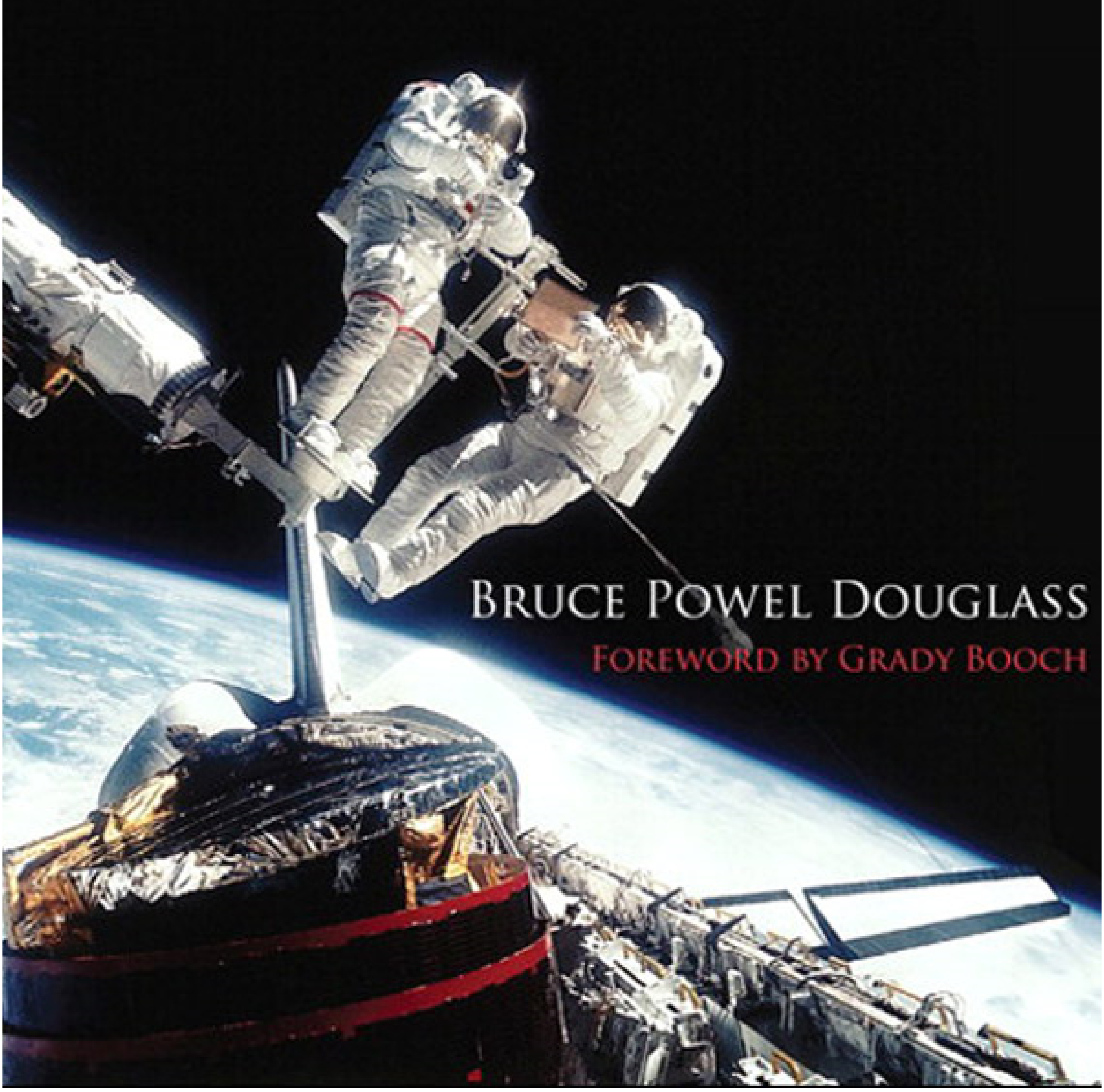
REAL-TIME AGILITY

The Harmony/ESW Method for Real-Time and Embedded Systems Development



BRUCE POWEL DOUGLASS

FOREWORD BY GRADY BOOCH

Praise for *Real-Time Agility*

"Regardless of your perceptions of Agile, this is a must read! Douglass's book is a powerful and practical guide to a well-defined process that will enable engineers to confidently navigate the complexity, risk, and variability of real-time and embedded systems—including CMMI compliance. From requirements specification to product delivery, whatever your modeling and development environment, *this is the instruction manual.*"

—Mark Scoville, software architect

"This book will provide you with the framework of agile development for real-time projects ranging from embedded systems to web-based, data collection applications. I wish I had this book three years ago when we began a real-time, embedded drilling control system project, but all my engineers will be getting copies now that it is available. And, for my academic colleagues, this is the perfect book for graduate seminars in applied software development techniques."

—Don Shafer, chief technology officer, Athens Group;
adjunct professor, Cockrell School of Engineering,
The University of Texas at Austin

"We have used Dr. Douglass's books on real-time (*Doing Hard Time*, *Real-Time UML*, and *Real-Time Design Patterns*) for years. His books are always informative, accessible, and entertaining. *Real-Time Agility* continues that tradition, and I can't wait to introduce it to my colleagues."

—Chris Talbott, principal software designer

"Until now, agile software development has been mostly applied within the IT domain. This book breaks new ground by showing how to successfully traverse the perceived chasm between agility and real-time development. Although embedded systems impose challenging constraints on development teams, you can always benefit from increasing your agility."

—Scott W. Ambler, chief methodologist/Agile, IBM Rational;
author of Agile Modeling

Real-Time Agility

The Harmony/ESW Method for Real-Time and Embedded Systems Development

Bruce Powel Douglass

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapols • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Douglass, Bruce Powel.
Real-time agility : the harmony/embedded process for real-time and embedded systems development / Bruce Powel Douglass.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-54549-7 (pbk. : alk. paper) 1. Agile software development.
2. Real-time data processing. 3. Embedded computer systems—Programming.
I. Title.

QA76.76.D47D68 2009
004'.33—dc22

2009011227

ISBN-13: 978-0-321-54549-7
ISBN-10: 0-321-54549-4
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, June 2009

This book is dedicated to my family. My sons, Scott and Blake, are fantastic. My step-daughter, Tamera, is nothing short of awesome. And my wife, Sarah, is simultaneously both beautiful and geeky—an intoxicating mixture!

Contents

Foreword

One of the things I have always admired about Bruce is his ability to take a complex, potentially deadly serious topic—in this case, real-time and embedded systems development—and make it interesting, approachable, and practical.

Bruce has contributed a large and important body of work to this domain. In his previous books on developing real-time systems, Bruce has attended to the issues of an underlying theory, best practices for modeling, and the codification of design patterns. In this present work, he turns his attention to the human elements: How does one develop quality real-time and embedded systems in a repeatable, predictable, reliable fashion? To that end, Bruce weds the evolving field of agile development with real-time development. He brings to the table considerable experience in developing and delivering real systems, and thus his observations on the specific needs of embedded systems are both relevant and credible.

As you'll see by reading this book, Bruce is somewhat of a Renaissance man. You don't often see a software book that contains code, UML models, some hairy mathematical formulas, and entertaining prose all in one package, but Bruce does pull it off. In reading his work, I often found myself nodding a vigorous yes, or being pleasantly jolted by his insights. Bruce's work is methodical, complete, and pragmatic. I hope that you will enjoy it as much as I have.

For as a classically trained musician raised by wolves, Bruce has certainly made a difference in this industry.

—Grady Booch

IBM Fellow

Preface

Back in 1996, I perceived a need for guidance on the development of real-time and embedded systems. By that time, I had built many such systems in various domains, such as medical, telecommunications, consumer electronics, and military aerospace, over the previous 20-odd years. I had developed a process known as ROPES (Rapid Object-Oriented Process for Embedded Systems) based on that experience. The development projects provided a cauldron for development (adding a bit of this and a scooch of that) and evaluation of the concepts and techniques. The evaluation of the techniques in real projects turned out to be invaluable because truth is often different from theory. What worked well was kept and what didn't was culled. Over time, ROPES was integrated with systems engineering (with the help of Dr. Hans-Peter Hoffmann), resulting in the Harmony process. Later, with the acquisition of I-Logix by Telelogic, Harmony was elaborated into a family of processes of which embedded systems development was a member (Harmony/ESW).

My original thought for providing guidance included producing a set of books on the topics of real-time theory, modeling real-time systems (with a companion book providing detailed exercises), developing real-time architectures, and efficient development techniques and processes. While that vision changed a bit over the next 12 years, I dutifully proceeded to begin capturing and elaborating the ideas, creating the examples, and writing the chapters. I had written several books before, so I had an idea of the extent of the work I had undertaken (although it turns out that I seriously underestimated the effort). In parallel with this writing effort, my "day job" kept me very busy consulting to customers in a variety of embedded domains; contributing to standards such as the UML, the UML Profile for Schedulability, Performance, and Time, SysML, and others; and speaking at dozens of conferences. I believe the primary effect of these activities was to significantly improve the quality of the practices through their repeated application in real projects.

Over the years, the resulting set of books realized the vision:

• *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Reading, MA: Addison-Wesley, 1999) was the "real-time theory" book that focused on the core concepts of real-time software-intensive systems.

• The *Real-Time UML* book has gone through two revisions so far (the third edition was published by Addison-Wesley in 2004) and focuses on how to analyze and design real-time systems with the UML.

• *Real-Time Design Patterns: Robust Scalable Architectures for Real-Time Systems* (Boston: Addison-Wesley, 2002) provided a taxonomy of views for real-time system architectures and a set of architectural design patterns that could be used to construct those architectures.

• *Real-Time UML Workshop for Embedded Systems* (Burlington, MA: Elsevier Press, 2006) was meant to be the "lab book" to accompany *Real-Time* UML. It walks the reader through a progressive series of exercises for both simple (traffic-light controller) and complex (unmanned air vehicle) system examples.

This book is meant to complete that initial vision. It represents the experience I have gained more than thirty years of developing real-time and embedded systems. In my work, I emphasized early measures of correctness and just enough process to achieve the goals long before Extreme Programming and agile methods became the buzzwords they are today. This book is focuses on agility, "traveling light" along the road to software development. Doing enough with process to improve the quality of the developed systems but not so much that the workflows become a burden to the developer is the key goal for the Harmony/ESW process. I firmly believe that a good process enables you to produce high-quality software in less time with fewer defects; poor processes result in either lower quality or much higher development cost.

Why Agile?

When you embark on a trip, you want to bring enough stuff with you to provide the necessities, comfort, and support for the trip and its intended purpose. However, you live within constraints about how much you can bring. Not only do the airlines limit the number of bags you can bring; they also limit the weight and size of each. Plus, you have to somehow get your luggage to the airport and from the airport to your destination. You must carefully select the items you bring so that you don't overburden yourself. When packing for a trip, you must carefully weigh the convenience of having more stuff against the cost and risk of carrying it around with you. Lots of people will give you advice, but the best advice will come from people who travel extensively for the same purposes that you do.

Developing software processes is very similar. Developing software is not about entering in the CPU operation codes in hexadecimal. It's not about writing the assembly, or even the high-level source code. It is about creating the software that meets the customer's needs and fits within the various constraints imposed by financial and technological limitations. It is about discovering the best way to organize and orchestrate the machine op codes to achieve the system's mission. It is about analysis and design. That is not to say that source or assembly code isn't important, but these are really sidebars to the fundamental concerns.

It turns out that software development is difficult. It requires invention on a daily basis. It is very hard to develop software that consistently does the right thing at the right time and does not have unintended side effects. Software development brings its own baggage, such as written documentation, review processes, change management processes, software development processes, testing processes, various work products, and tools.

Like the airline traveler, you must decide what baggage you need to bring along during the trip and what you can (and perhaps should) do without.

There are many software development processes from which to choose. The best of these are developed by thoughtful, smart, and *experienced people*. Far too many processes are defined by people who don't have the experience and won't have to develop software using the techniques and workflows that they come up with. The resulting processes are often incredibly cumbersome, bloated by extra documentation and ancillary work that is included "for completeness." It's like travel policies created and managed by people who don't actually travel, and therefore don't feel the pain that their policies entail. Such policies are rarely appreciated by the people who actually have to plan and travel.

Does this mean what many software developers have come to believe—that processes are at best burdens to be borne and at worst, impediments to the actual development of software? No; I believe process adds value. A good process provides guidance for the efficient development of high-quality software that meets the needs of the customer. Sadly, too many processes don't meet this need. They often require you to carry too much baggage along the way. Even worse, they often require you to carry items in your bags that don't aid you at all in achieving your goals.

Agile methods are a reaction to these heavyweight approaches to software development. Just as the efficient traveler needs just enough baggage, the effective software developer needs just enough process. Agile methods promote "traveling light"—using enough process to make you efficient and to create great software without spending extra effort performing tasks that don't add much value.

This book is about agile methods and how they can be applied to the development of real-time embedded-software-intensive systems. Many of these systems are large in scale and rigorous in their need for quality. Others are small and can be developed by a single person or a small team. Agile methods apply well to all such systems. The trick is to decide how much baggage you need to carry the elements you really need for the trip.

The delivery process detailed in this book is known as Harmony/ESW (for Embedded SoftWare). It is a member of the Harmony process family—a collection of best practices for software and systems development. Harmony/ESW uses a spiral development approach, incrementally developing the system with continuous execution, which provides immediate feedback as to the correctness and quality of the software being developed. It has been successfully applied to projects both small (3 people or fewer) and large (more than 100 people). It emphasizes the development of working software over the creation of documentation. It emphasizes correctness over paperwork and efficiency over artificial measures of completeness.

I hope you enjoy your trip.

Audience

The book is oriented toward the practicing professional software developer, the computer science major in the junior or senior year, project and technical leads, and software managers. The audience is assumed to have familiarity with the UML and basic software development and an interest in effective development of real-time and embedded software. Readers who need more information about UML or real-time concepts are referred to my other books, listed previously.

Goals

The primary goal of this book is to provide guidance for the efficient and effective development of real-time and embedded software. To this end, the first two chapters focus on the introduction of the basic concepts of agility, model-based development, and real-time systems. The next two chapters highlight the key principles and practices that enable rapid development of high-quality software and provide an overview of the Harmony/ESW process concepts. The remaining chapters take you through the process, providing detailed workflows, work product descriptions, and techniques.

About This Book

It should be noted that this book is about applying agile methods to the development of real-time and embedded systems. These systems, as discussed above, have some special characteristics and properties not found in traditional IT or desktop software. For that reason, some of the practices and implementation approaches will be a bit different from those recommended by some other authors.

In addition to being agile, the processes presented in this book also take advantage of other standards and techniques. One such is model-driven development (MDD). Model-driven development is the generic term for Model-Driven Architecture (MDA), an OMG[1] standard for developing reusable software assets that is being applied very effectively in aerospace and defense systems.

1. Object Management Group, not "Oh, my God," as some people believe.

Applying agile methods to modeling is not new. Scott Ambler's excellent book on agile modeling[2] provides a good discussion of agile methods in the context of using UML and the Rational Unified Process (RUP). This book differs in a couple of ways. First and foremost, this book applies both agile and MDD to the development of real-time and embedded systems. Many of these systems have special concerns about safety and reliability, and I will talk about how to address those concerns with agility. Second, this book uses the Harmony/ESW process as its basis, rather than the Unified Process. The processes are superficially similar, but the former focuses heavily on the special needs of real-time and embedded systems, particularly on quality of service, safety, and reliability and how they can be effectively managed. The third and final primary difference is the emphasis on the use of technology to automate certain aspects of development, especially in the use of highly capable modeling and testing tools. The process doesn't require high-end tools but takes advantage of them when it can.

2. Scott Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process* (New York: John Wiley & Sons, 2002).

In addition to standards, this book emphasizes the use of technology to make your development life better and easier. Just as few people insist on coding in assembler, today's modeling technology allows us to "raise the bar" in productivity by using tools to validate the correctness of models, generate application code, provide "automatic" documentation, enable trade-off analysis, and so on.

The value proposition of agile methods is significantly enhanced when it is used in conjunction with such standards and technologies. To be clear, the practices and process workflows presented in this book *are* agile and adhere to the basic goals and principles of agile methods, but they also embrace advances in processes and technologies to leverage that agility. The guidance in this book is both practical and proven and is the culmination of decades of personal experience developing, managing, or consulting to projects in this space. As a side note, most of the analysis and design examples in this book are of a Star Trek transporter system. Although I had to invent some physics for this, my son (a physics major himself) kept me from being too fast and loose with the laws of the universe. It was a fun design to create. Appendix A contains the requirement specifications for the system in case you want to implement it yourself.

Accessing the Harmony/ESW Process Content

This book discusses in detail the Harmony/ESW process content and how it can be used to implement agile methods for real-time and embedded systems development. By the time you read this, the Harmony/ESW process content should be integrated into the Rational Method Composer (RMC) content, available at www-01.ibm.com/software/awdtools/rmc/.

RMC is a flexible process management platform for authoring, managing, and publishing process content. It comes with an extensive library of best practices including the Rational Unified Process (RUP). It is used by companies and project teams needing to manage their development approaches to realize consistent, high-quality product development. The Web site includes the ability to download a trial copy or purchase the tool and related content.

Note: All process content on the above-mentioned Web site is the property of IBM.

Acknowledgments

While writing is rewarding, it is lots of work. I want to acknowledge the support of my wife, Sarah, through the months of burning the midnight photons while writing this book. I also want to thank Ray Blash at Rational/IBM for not keeping me out on the road consulting all the time and thus giving me enough time at home to work on the book. Additionally, thanks to Barbara Wood for her diligence in editing the proofs. She did an amazing job. All remaining defects are mine and I claim responsibility for them.

About the Author

Bruce Powel Douglass was raised by wolves in the Oregon wilderness. He taught himself to read at age 3 and calculus before age 12. He dropped out of school when he was 14 and traveled around the United States for a few years before entering the University of Oregon as a mathematics major. He eventually received his M.S. in exercise physiology from the University of Oregon and his Ph.D. in neurophysiology from the University of South Dakota Medical School, where he developed a branch of mathematics called autocorrelative factor analysis for studying information processing in multicellular biological neural systems.

Bruce has worked as a software developer in real-time systems for more than thirty years and is a well-known speaker, author, and consultant in the area of real-time embedded systems. He is on the advisory board of the Embedded Systems Conference, where he has taught courses in software estimation and scheduling, project management, object-oriented analysis and design, communications protocols, finite state machines, design patterns, and safety-critical systems design. He develops and teaches courses and consults in real-time object-oriented analysis and design and project management and has done so for many years. He has authored articles for many journals and periodicals, especially in the real-time domain.

He is the chief evangelist[1] for IBM Rational, a leading producer of tools for software and systems development. Bruce worked with various UML partners on the specification of the UM, both versions 1 and 2. He is a former cochair of the Object Management Group's Real-Time Analysis and Design Working Group. He is the author of several books on software, including *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Reading, MA: Addison-Wesley, 1999), *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Boston: Addison-Wesley, 2002), *Real-Time UML: Advances in the UML for Real-Time Systems, Third Edition* (Boston: Addison-Wesley, 2004), *Real-Time UML Workshop for Embedded Systems* (Burlington, MA: Elsevier Press, 2006), and several others, as well as a short textbook on table tennis.

1. Being a chief evangelist is much like being a chief scientist, except for the burning bushes and stone tablets.

Bruce enjoys classical music and has played classical guitar professionally. He has competed in several sports, including table tennis, bicycle racing, running, triathlons, and full-contact Tae Kwon Do (in which he holds a black belt), although he currently fights only inanimate objects that don't hit back.

Bruce does extensive consulting and training throughout the world, earning thousands of frequent-flier miles that he rarely has the time to use. If you're interested, contact him at Bruce.Douglass@us.ibm.com.

Chapter 1
Introduction to Agile and Real-Time Concepts

Different people mean different things when they use the term **agile**. The term was first used to describe a lightweight approach to performing project development after the original term, **Extreme Programming (XP)**,[1] failed to inspire legions of managers entrusted to oversee development projects. Basically, agile refers to a loosely integrated set of principles and practices focused on getting the software development job done in an economical and efficient fashion.

1. Note that important acronyms and terms are defined in the Glossary.

This chapter begins by considering why we need agile approaches to software development and then discusses agile in the context of real-time and embedded systems. It then turns to the advantages of agile development processes as compared to more traditional approaches.

1.1. The Agile Manifesto

A good place to start to understand agile methods is with the agile manifesto.[2] The manifesto is a public declaration of intent by the Agile Alliance, consisting of 17 signatories including Kent Beck, Martin Fowler, Ron Jeffries, Robert Martin, and others. Originally drafted in 2001, this manifesto is summed up in four key priorities:

2. http://agilemanifesto.org/. Martin Fowler gives an interesting history of the drafting at http://martinfowler.com/articles/agileStory.html.

• *Individuals and interactions* over processes and tools

• *Working software* over comprehensive documentation

• *Customer collaboration* over contract negotiation

• *Responding to change* over following a plan

To support these statements, they give a set of 12 principles. I'll state them here to set the context of the following discussion:

• Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

• Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

• Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

• Business people and developers must work together daily throughout the project.

• Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

• The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

• Working software is the primary measure of progress.

• Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

• Continuous attention to technical excellence and good design enhances agility.

• Simplicity—the art of maximizing the amount of work not done—is essential.

• The best architectures, requirements, and designs emerge from self-organizing teams.

• At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile methods have their roots in the XP (Extreme Programming[3]) movement based largely on the work of Kent Beck and Ward Cunningham. Both agile and XP have been mostly concerned with IT systems and are heavily code-based. In this book, I will focus on how to effectively harness the manifesto's statements and principles in a different vertical market—namely, realtime and embedded—and how to combine them with modeling to gain the synergistic benefits of model-driven development (MDD) approaches.[4]

3. See www.xprogramming.com/what_is_xp.htm or Kent Beck's *Extreme Programming Explained* (Boston: Addison-Wesley, 2000) for an overview.

4. A good place for more information about agile modeling is Scott Ambler's agile modeling

Web site, http://www.agilemodeling.com/.

> Note
>
> Agility isn't limited to small projects. Agile@Scale is an IBM initiative to bring the benefits of agility to larger-scale systems and projects. This initiative includes agile project tool environments such as Rational Team Concert (RTC; based on the Jazz technology platform). Interested readers are referred to www-01.ibm.com/software/rational/agile and www-01.ibm.com/software/rational/jazz/features.

1.2. Why Agile?

But why the need for a concept such as "agile" to describe software development? Aren't current software development processes good enough?

No, not really.

A process, in this context, can be defined as "a planned set of work tasks performed by workers in specific roles resulting in changes of attributes, state, or other characteristics of one or more work products." The underlying assumptions are the following:

• The results of using the process are repeatable, resulting in a product with expected properties (e.g., functionality and quality).

• The production of the goal state of the work products is highly predictable when executing the process in terms of the project (e.g., cost, effort, calendar time) and product (e.g., functionality, timeliness, and robustness) properties.

• People can be treated as anonymous, largely interchangeable resources.

• The problems of software development are infinitely scalable—that is, doubling the resources will always result in halving the calendar time.

As it turns out, software is *hard* to develop. Most existing development processes are most certainly not repeatable or predictable in the sense above. There are many reasons proposed for why that is. For myself, I think software is *fundamentally complex*—that is, it embodies the "stuff" of complexity. That's what software is best at—capturing how algorithms and state machines manipulate multitudes of data within vast ranges to achieve a set of computational results. It's "thought stuff," and that's hard.

The best story I've heard about software predictability is from a blog on the SlickEdit Web site by Scott Westfall called the "The Parable of the Cave" (see sidebar).[5] Estimating software projects turns out to be remarkably similar to estimating how long it will take to explore an unknown cave, yet managers often insist on being given highly precise estimates.

5. Used with permission of the author, Scott Westfall. The SlickEdit Web site can be found at http://blog.slickedit.com/?p207.

The Parable of the Cave

Two people stand before a cave. One is the sagely manager of a cave exploring company whose wisdom is only exceeded by his wit, charm, and humility. Let's call him, oh, "Scott." The other is a cave explorer of indeterminate gender who bears no resemblance to any programmers past or present that this author may have worked with and whose size may be big or little. Let's call him/her "Endian."

"Endian," said Scott in a sagely voice that was both commanding and compassionate, "I need you to explore this cave. But before you do, I need to know how long you think it will take, so that I may build a schedule and tell the sales team when the cave will be ready."

"Great Scott," replied Endian using the title bestowed upon Scott by his admiring employees, "how can I give you an answer when surely you know I have never been in this cave before? The cave may be vast, with deep chasms. It may contain a labyrinth of underwater passages. It may contain fearsome creatures that must first be vanquished. How can I say how long it will take to explore?"

Scott pondered Endian's words and after a thorough analysis that might have taken days for others but was completed in but a moment for him, he replied, "Surely this is not the first cave you explored. Are there no other caves in this district? Use your knowledge of those caves to form an estimate."

Endian heard these words and still his doubt prevailed. "Your words are truly wise," said Endian, "but even within a district the caves may vary, one from another. Surely, an estimate based on the size of another cave cannot be deemed accurate."

"You have spoken truly, good Endian," replied Scott in a fatherly, supporting tone that lacked any trace of being patronizing as certain cynical readers may think. "Here, take from me this torch and this assortment of cheeses that you may explore the cave briefly. Return ere the morrow and report what you have learned."

The parable continues like this for pages, as parables are known to do. Let's see, Endian enters the cave . . . something about a wretched beast of surpassing foulness . . . he continues on . . . hmm, that's what the assortment of cheeses were for. Ah! Here we go.

Endian returns to Scott, his t-shirt ripped and his jeans covered in mud. Being always concerned with the well-being of his employees, Scott offers Endian a cool drink, then asks, "Endian, what news of the cave? Have you an estimate that I can use for my schedule? What shall I tell the sales team?"

Endian considers all that he has seen and builds a decomposition containing the many tasks necessary to explore the cave based on his earlier reconnoitering. He factors in variables for risk and unknowns, and then he responds, "Two weeks."

In addition, the scope of software is increasing rapidly. Compared to the scope of the software functionality in decades past, software these days does orders of magnitude more. Back in the day,[6] my first IBM PC had 64kB of memory and ran a basic disk operating system called DOS. DOS fit on a single 360kB floppy disk. Windows XP weighs in at well over 30 million lines of code; drives hundreds of different printers, disks, displays, and other peripherals; and needs a gigabyte of memory to run comfortably. These software-intensive systems deliver far more functionality than the electronic-only devices they replace. Compare, for example, a traditional phone handset with a modern cell phone. Or compare a traditional electrocardiogram (ECG) that drove a paper recorder like the one I used in medical school with a modern ECG machine—the difference is remarkable. The modern machine can do everything the old machine did, plus detect a wide range of arrhythmias, track patient data, produce reports, and measure noninvasive blood pressure, blood oxygen concentration, a variety of temperatures, and even cardiac output.

6. I know I'm dating myself, but my IBM PC was my fifth computer. I still remember fondly the days of my TRS-80 model I computer with its 4kB of memory . . .

Last, software development is really invention, and invention is not a highly predictable thing. In electronic and mechanical engineering, a great deal of the work is conceptually simply putting pieces together to achieve a desired goal, but in software those pieces are most often invented (or reinvented) for every project. This is not to oversimplify the problems of electronic or mechanical design but merely to point out that the underlying physics of those disciplines is far more mature and well understood than that of software.

But it doesn't really matter if you believe my explanations; the empirical results of decades of software development are available. Most products are late.[7] Most products are delivered with numerous and often significant defects. Most products don't deliver all the planned functionality. We have become used to rebooting our devices, but 30 years ago it would have been unthinkable that we would have to turn our phones off, remove the batteries, count to 30, reinsert the batteries, and reboot our phones.[8] Unfortunately, that is the "state of the art" today.

7. See, for example, Michiel van Genuchten, "Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development," *IEEE Transactions on Software Engineering* 17, no. 6 (June 1991).

8. Much as I love my BlackBerry, I was amazed that a customer service representative recommended removing the battery to reboot the device *daily*.

To this end, many bright people have proposed processes as a means of combating the problem, reasoning that if people *engineered* software rather than hacked away at it, the results would be better. And they have, to a large degree, been better. Nevertheless, these approaches have been based on the premise that software development can be treated the same as an industrial manufacturing process and achieve the same results. Industrial automation problems are highly predictable, and so this approach makes a great deal of sense when the underlying mechanisms driving the process are very well understood and are inherently linear (i.e., a small change in input results in an equally small change in output). It makes less sense when the underlying mechanisms are not fully understood or the process is highly nonlinear. Unfortunately, software development is neither fully understood nor even remotely linear.

It is like the difference in the application of fuzzy logic and neural networks to nonlinear control systems. Fuzzy logic systems work by applying the concept of partial membership and using a centroid computation to determine outputs. The partial membership of different sets (mapping to different equations) is defined by set membership rules, so fuzzy logic systems are best applied when the rules are known and understood, such as in speed control systems.

Neural networks, on the other hand, don't know or care about rules. They work by training clusters of simple but deeply interconnected processing units (neurons). The training involves applying known inputs ("exemplars") and adjusting the weights of the connections until you get the expected outputs. Once trained, the neural network can produce results from previously unseen data input sets and produce control outputs. The neural network learns the effects of the underlying mechanisms from actual data, but it doesn't in any significant way "understand" those mechanisms. Neural networks are best used when the underlying mechanisms are not well understood because they can learn the data transformations inherent in the mechanisms.

Rigorously planned processes are akin to fuzzy logic—they make a priori assumptions about the underlying mechanisms. When they are right, a highly predictable scheme results. However, if those a priori assumptions are either wrong or missing, then they yield less successful results. In this case, the approach must be tuned with empirical data. To this end, most traditional processes do "extra" work and produce "extra" products to help manage the process. These typically include

• Schedules

• Management plans

• Metrics (e.g., source lines of code [SLOC] or defect density)

• Peer and management reviews and walk-throughs

• Progress reports

And so on.

The idea is that the execution of these tasks and the production of the work products correlate closely with project timeliness and product functionality and quality. However, many of the tasks and measures used don’t correlate very well, even if they are easy to measure. Even when they do correlate well, they incur extra cost and time.

Agile methods are a reaction in the developer community to the high cost and effort of these industrial approaches to software development. The mechanisms by which we invent software are not so well understood as to be highly predictable. Further, small changes in requirements or architecture can result in huge differences in development approach and effort. Because of this, empiricism, discipline, quality focus, and stakeholder focus must all be present in our development processes. To this end, agile methods are not about hacking code but instead are about focusing effort on the things that demonstrably add value and defocusing on efforts that do not.

1.3. Properties of Real-Time Embedded Systems

Of course, software development is hard. Embedded software development is harder. Real-time embedded software is even harder than that. This is not to minimize the difficulty in reliably developing application software, but there are a host of concerns with real-time and embedded systems that don’t appear in the production of typical applications.

An embedded system is one that contains at least one CPU but does not provide general computing services to the end users. A cell phone is considered an embedded computing platform because it contains one or more CPUs but provides a dedicated set of services (although the distinction is blurred in many contemporary cell phones). Our modern society is filled with embedded computing devices: clothes washers, air traffic control computers, laser printers, televisions, patient ventilators, cardiac pacemakers, missiles, global positioning systems (GPS), and even automobiles—the list is virtually endless.

The issues that appear in real-time embedded systems manifest themselves on four primary fronts. First, the optimization required to effectively run in highly resource-constrained environments makes embedded systems more challenging to create. It is true that embedded systems run the gamut from 8-bit processes in dishwashers and similar machinery up to collaborating sets of 64-bit computers. Nevertheless, most (but not all) embedded systems are constrained in terms of processor speed, memory, and user interface (UI). This means that many of the standard approaches to application development are inadequate alone and must be optimized to fit into the computing environment and perform their tasks. Thus embedded systems typically require far more optimization than standard desktop applications. I remember writing a real-time operating system (RTOS) for a cardiac pacemaker that had 32kB of static memory for what amounted to an embedded 6502 processor.[9] Now *that's* an embedded system!

9. It even had a small file system to manage different pacing and monitoring applications.

Along with the highly constrained environments, there is usually a need to write more device-driver-level software for embedded systems than for standard application development. This is because these systems are more likely to have custom hardware for which drivers do not exist, but even when they do exist, they often do not meet the platform constraints. This means that not only must the primary functionality be developed, but the low-level device drivers must be written as well.

The real-time nature of many embedded systems means that predictability and schedulability affect the correctness of the application. In addition, many such systems have high reliability and safety requirements. These characteristics require additional analyses, such as schedulability (e.g., rate monotonic analysis, or RMA), reliability (e.g., failure modes and effects analysis, or FMEA), and safety (e.g., fault tree analysis, or FTA) analysis. In addition to "doing the math," effort must be made to ensure that these additional requirements are met.

Last, a big difference between embedded and traditional applications is the nature of the so-called target environment—that is, the computing platform on which the application will run. Most desktop applications are "hosted" (written) on the same standard desktop computer that serves as the target platform. This means that a rich set of testing and debugging tools is available for verifying and validating the application. In contrast, most embedded systems are "cross-compiled" from a desktop host to an embedded target. The embedded target lacks the visibility and control of the program execution found on the host, and most of the desktop tools are useless for debugging or testing the application on its embedded target. The debugging tools used in embedded systems development are almost always more primitive and less powerful than their desktop counterparts. Not only are the embedded applications more complex (due to the optimization), and not only do they have to drive low-level devices, and not only must they meet additional sets of quality-of-service (QoS) requirements, but the debugging tools are far less capable as well.

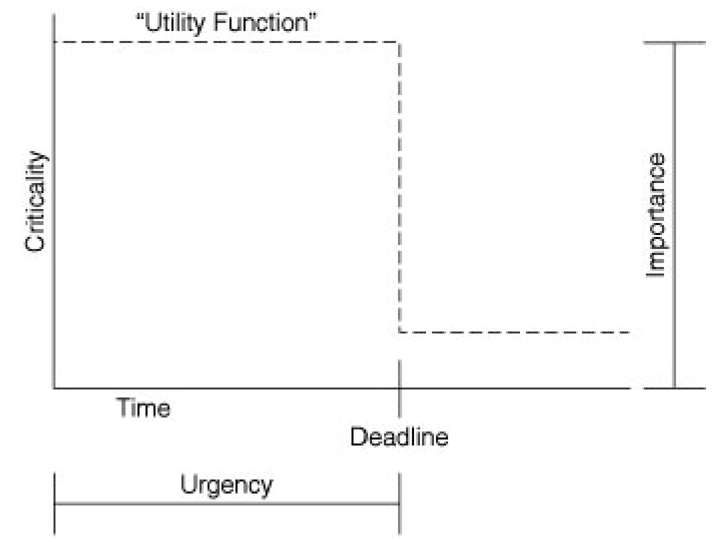
It should be noted that another difference exists between embedded and "IT" software development. IT systems are often maintained systems that constantly provide services, and software work, for the most part, consists of small incremental efforts to remove defects and add functionality. Embedded systems differ in that they are released at an instant in time and provide functionality at that instant. It is a larger effort to update embedded systems, so that they are often, in fact, replaced rather than being "maintained" in the IT sense. This means that IT software can be maintained in smaller incremental pieces than can embedded systems, and "releases" have more significance in embedded software development.

A "real-time system" is one in which timeliness is important to correctness. Many developers incorrectly assume that "real-time" means "real fast." It clearly does not. Real-time systems are "predictably fast enough" to perform their tasks. If processing your eBay order takes an extra couple of seconds, the server application can still perform its job. Such systems are not usually considered realtime, although they may be optimized to handle thousands of transactions per second, because if the system slows down, it doesn't affect the system's *correctness*. Real-time systems are different. If a cardiac pacemaker fails to induce current through the heart muscle at the right time, the patient's heart can go into fibrillation. If the missile guidance system fails to make timely corrections to its attitude, it can hit the wrong target. If the GPS satellite doesn't keep a highly precise measure of time, position calculations based on its signal will simply be *wrong*.

Real-time systems are categorized in many ways. The most common is the broad grouping into "hard" and "soft." "Hard" real-time systems exhibit significant failure if every single action doesn't execute within its time frame. The measure of timeliness is called a **deadline**—the time after action initiation by which the action must be complete. Not all deadlines must be in the microsecond time frame to be real-time. The **F2T2EA** (**F**ind, **F**ix, **T**rack, **T**arget, **E**ngage, **A**ssess) **Kill Chain** is a fundamental aspect of almost all combat systems; the end-to-end deadline for this compound action might be on the order of 10 minutes, but pilots absolutely must achieve these deadlines for combat effectiveness.

The value of the completion of an action as a function of time is an important concept in real-time systems and is expressed as a "utility function" as shown in Figure 1.1. This figure expresses the value of the completion of an action to the user of the system. In reality, utility functions are smooth curves but are most often modeled as discontinuous step functions because this eases their mathematical analysis. In the figure, the value of the completion of an action is high until an instant in time, known as the deadline; at this point, the value of the completion of the action is zero. The length of time from the current time to the deadline is a measure of the urgency of the action. The height of the function is a measure of the criticality or importance of the completion of the action. Criticality and urgency are important orthogonal properties of actions in any real-time system. Different scheduling schemas optimize urgency, others optimize importance, and still others support a fairness (all actions move forward at about the same rate) doctrine.

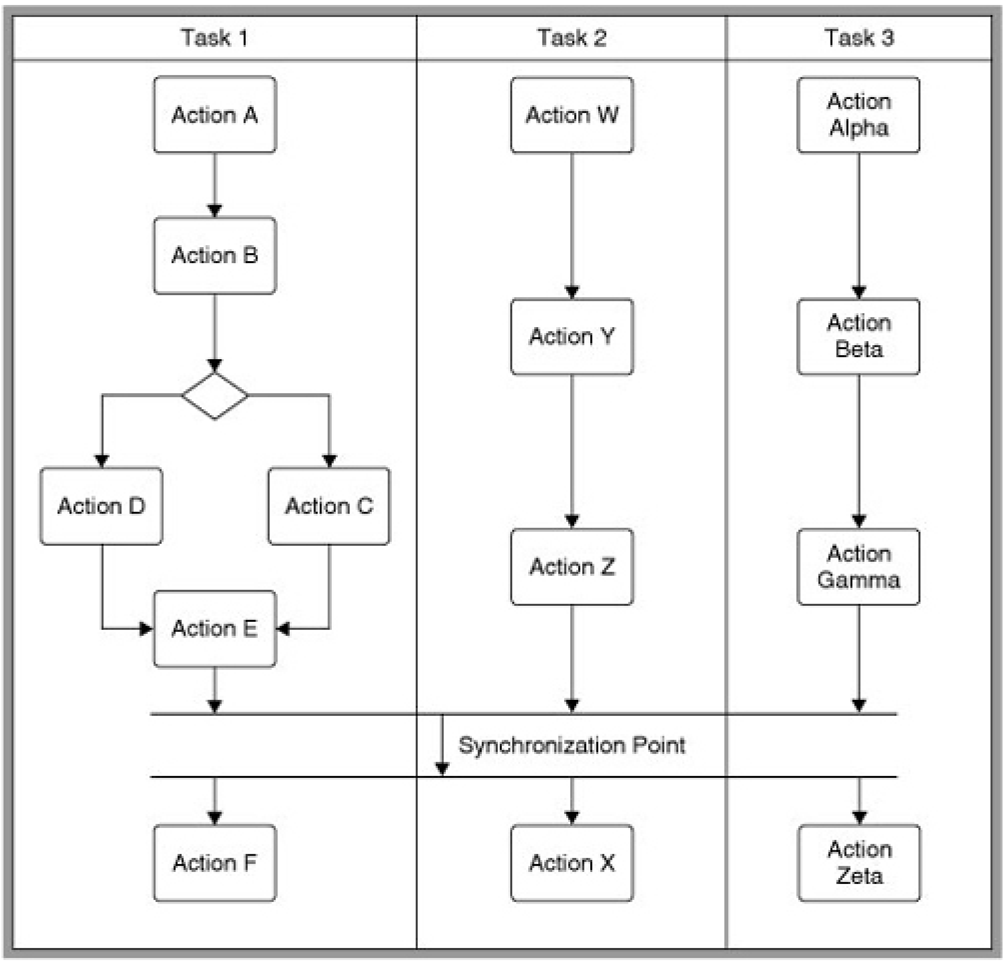
Figure 1.1 *Utility function*



Actions are the primitive building blocks of concurrency units, such as tasks or threads. A **concurrency unit** is a sequence of actions in which the order is known; the concurrency unit may have branch points, but the sequence of actions within a set of branches is fully deterministic. This is not true for the actions between concurrency units. Between concurrency units, the sequence of actions is *not* known, or cared about, except at explicit synchronization points.

Figure 1.2 illustrates this point. The flow in each of the three tasks (shown on a UML activity diagram) is fully specified. In Task 1, for example, the sequence is that Action A occurs first, followed by Action B and then either Action C or Action D. Similarly, the sequence for the other two tasks is fully defined. What is not defined is the sequence between the tasks. Does Action C occur before or after Action W or Action Gamma? The answer is *You don't know and you don't care*. However, we know that before Action F, Action X, and Action Zeta can occur, Action E, Action Z, and Action Gamma have all occurred. This is what is meant by a task synchronization point.

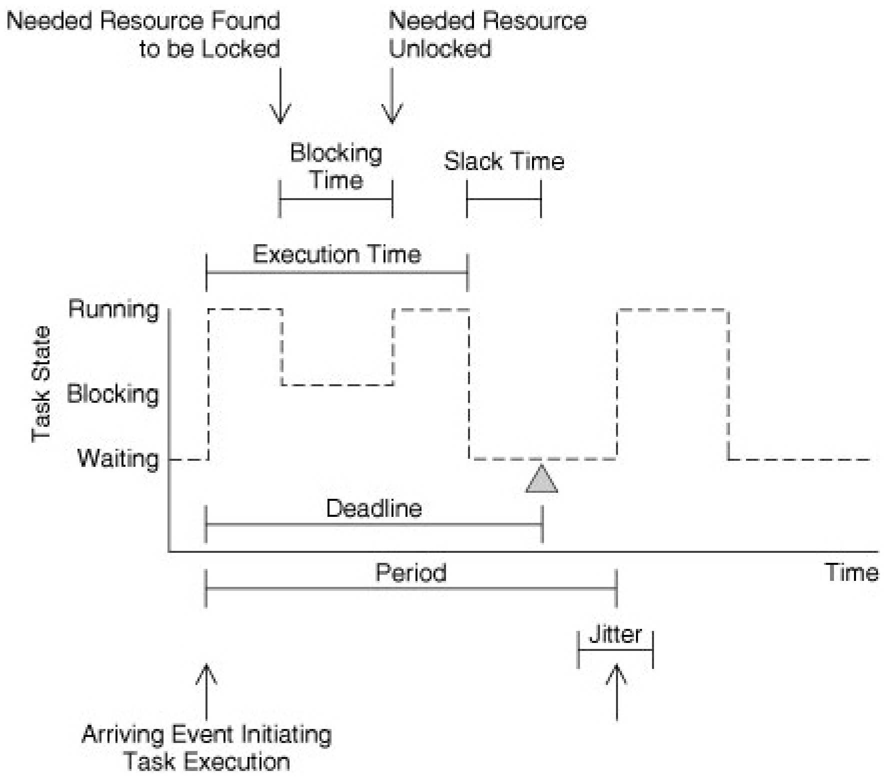
Figure 1.2 *Concurrency units*



Because in real-time systems synchronization points, as well as resource sharing, are common, they require special attention in real-time systems not often found in the development of IT systems.

Within a task, several different properties are important and must be modeled and understood for the task to operate correctly (see Figure 1.3). Tasks that are time-based occur with a certain frequency, called the **period.** The period is the time between invocations of the task. The variation around the period is called *jitter*. For event-based task initiation, the time between task invocations is called the **interarrival time.** For most schedulability analyses, the shortest such time, called the **minimum interarrival time,** is used for analysis. The time from the initiation of the task to the point at which its set of actions must be complete is known as the **deadline.** When tasks share resources, it is possible that a needed resource isn't available. When a necessary resource is locked by a lower-priority task, the current task must **block** and allow the lower-priority task to complete its use of the resource before the original task can run. The length of time the higher-priority task is prevented from running is known as the **blocking time.** The fact that a lower-priority task must run even though a higher-priority task is ready to run is known as **priority inversion** and is a property of all priority-scheduled systems that share resources among task threads. Priority inversion is unavoidable when tasks share resources, but when uncontrolled, it can lead to missed deadlines. One of the things real-time systems must do is bound priority inversion (e.g., limit blocking to the depth of a single task) to ensure system timeliness. The period of time that a task requires to perform its actions, including any potential blocking time, is called the **task execution time.** For analysis, it is common to use the longest such time period, the **worst-case execution time,** to ensure that the system can always meet its deadlines. Finally, the time between the end of the execution and the deadline is known as the **slack time.** In real-time systems, it is important to capture, characterize, and manage all these task properties.

Figure 1.3 *Task time*



Real-time systems are most often embedded systems as well and carry those burdens of development. In addition, real-time systems have timeliness and schedulability constraints. Real-time systems must be **timely**—that is, they must meet their task completion time constraints. The entire set of tasks is said to be **schedulable** if all the tasks are timely. Real-time systems are not necessarily (or even usually) deterministic, but they must be predictably bounded in time. Methods exist to mathematically analyze systems for schedulability,[10] and there are tools[11] to support that analysis.

10. See *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns and Real-Time UML: Advances in the UML for Real-Time Systems*, both written by me and published by Addison-Wesley (1999 and 2004, respectively).

11. For example, see http://www.tripac.com/ for information about the RapidRMA tool.

Safety-critical and high-reliability systems are special cases of real-time and embedded systems. The term **safety** means "freedom from accidents or losses"[12] and is usually concerned with safety in the absence of faults as well as in the presence of single-point faults. **Reliability** is usually a stochastic measure of the percentage of the time the system delivers services.

12. Nancy Leveson, *Safeware: System Safety and Computers* (Reading, MA: Addison-Wesley, 1995).

Safety-critical systems are real-time systems because safety analysis includes the property of **fault tolerance time**—the length of time a fault can be tolerated before it leads to an accident. They are almost always embedded systems as well and provide critical services such as life support, flight management for aircraft, medical monitoring, and so on. Safety and reliability are assured through the use of additional analysis, such as FTA, FMEA, failure mode, effects, and criticality analysis (FMECA), and often result in a document called the hazard analysis that combines fault likelihood, fault severity, risk (the product of the previous two), hazardous conditions, fault protection means, fault tolerance time, fault detection time, and fault protection action time together. Safety-critical and high-reliability systems require additional analysis and documentation to achieve approval from regulatory agencies such as the FAA and FDA.

It is not at all uncommon for companies and projects to specify very heavyweight processes for the development of these kinds of systems—safety-critical, high-reliability, real-time, or embedded—as a way of injecting quality into those systems. And it works, to a degree. However, it works at a very high cost. Agile methods provide an alternative perspective on the development of these kinds of systems that is lighter-weight but does not sacrifice quality.

1.4. Benefits of Agile Methods

The primary goal of an agile project is to develop working software that meets the needs of the stakeholders. It isn't to produce documentation (although documentation will be part of the delivered system). It isn't to attend meetings (although meetings will be held). It isn't to create schedules (but a schedule is a critical planning tool for all agile projects). It isn't to create productivity metrics (although they will help the team identify problems and barriers to success).[13] You may do all of these things during the pursuit of your primary goal, but it is key to remember that those activities are secondary and performed only as a means of achieving your primary goal. Too often, both managers and developers forget this and lose focus. Many projects spend significant effort without even bothering to assess whether that effort aids in the pursuit of the development of the software.

13. See Scott Ambler's discussion of acceleration metrics at www.ibm.com/developerworks/blogs/page/ambler?tag=Metrics.

The second most important goal of an agile project is to enable follow-on software development. This means that the previously developed software must have an architecture that enables the next set of features or extensions, documentation so that the follow-on team can understand and modify that software, support to understand and manage the risks of the development, and an infrastructure for change and configuration management (CM).

The benefits of agile methods usually discussed are:

• Rapid learning about the project requirements and technologies used to realize them

• Early return on investment (ROI)

• Satisfied stakeholders

• Increased control

• Responsiveness to change

• Earlier and greater reduction in project risk[14]

• Efficient high-quality development

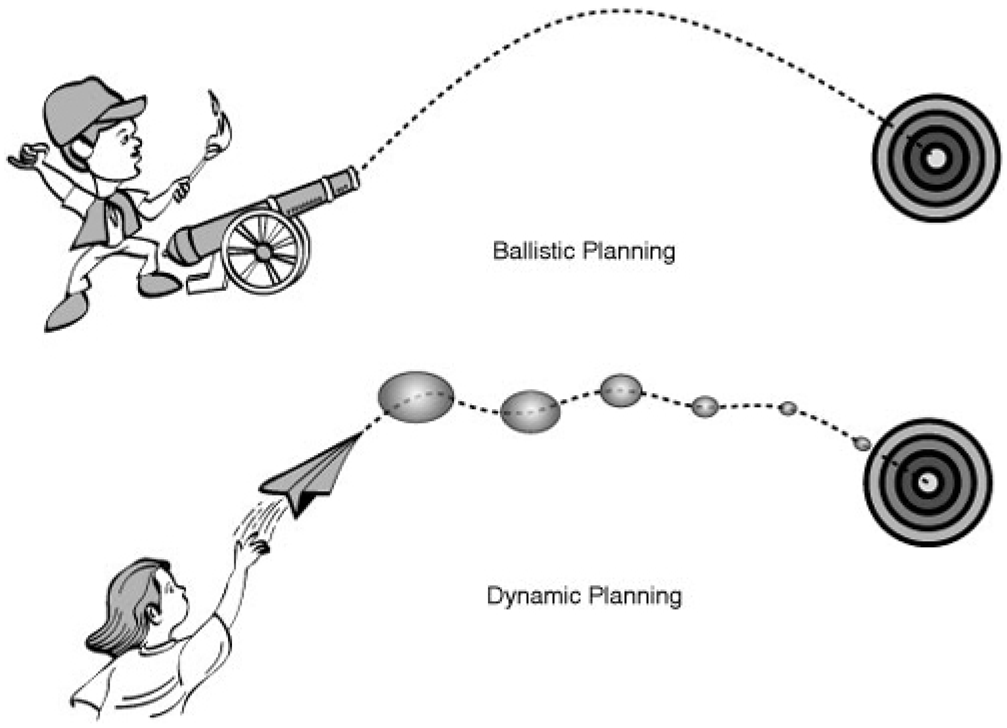
14. See, for example, http://www.agileadvice.com/.

These are real, if sometimes intangible, benefits that properly applied agile methods bring to the project, the developers, their company, the customer, and the ultimate user.

1.4.1. Rapid Learning

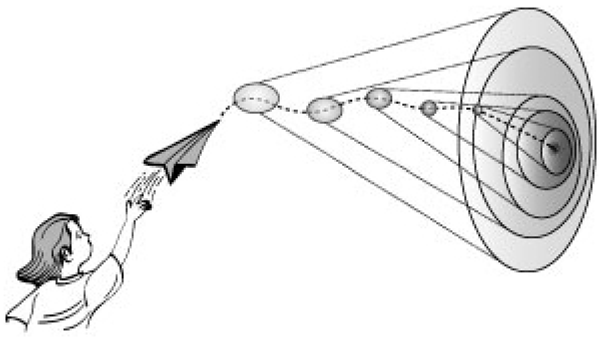
Rapid learning means that the development team learns about the project earlier *because they are paying attention*. Specifically, agile methods focus on early feedback, enabling dynamic planning. This is in contrast to traditional approaches that involve ballistic planning. Ballistic planning is all done up front with the expectation that physics will guide the (silver) bullet unerringly to its target (see Figure 1.4). Agile's dynamic planning can be thought of as "planning to replan." It's not that agile developers don't make plans; it's just that they don't believe their own marketing hype and are willing to improve their plans as more information becomes available.

Figure 1.4 *Ballistic versus dynamic planning*



Since software development is relatively unpredictable, ballistic planning, for all its popularity, is infeasible. The advantage of early feedback is that it enables dynamic planning. A Law of Douglass[15] is "The more you know, the more you know." This perhaps obvious syllogism means that as you work through the project, you learn. This deeper understanding of the project enables more accurate predictions about when the project will be complete and the effort the project will require. As shown in Figure 1.5, the ongoing course corrections result in decreasing the zone of uncertainty.

Figure 1.5 *Reduction in uncertainty*

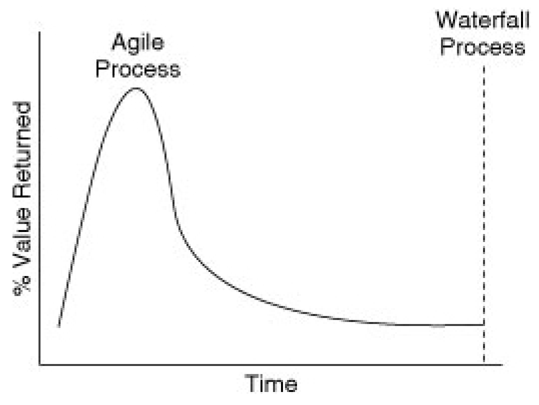


15. Unpublished work found in the Douglass crypt . . .

1.4.2. Early Return on Investment

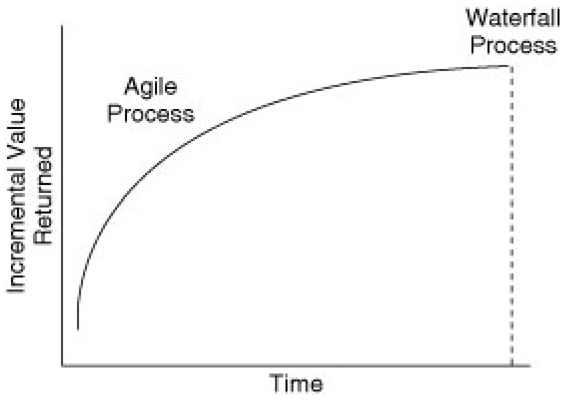
Early return on investment means that with an agile approach, partial functionality is provided far sooner than in a traditional waterfall process. The latter delivers all-or-none functionality at the end point, and the former delivers incremental functionality frequently throughout the duration of the development. As you can see in Figure 1.6, agile delivers high value early, with less incremental value as the system becomes increasingly complete, whereas the waterfall process delivers nothing until the end.

Figure 1.6 *Percent value returned over time*



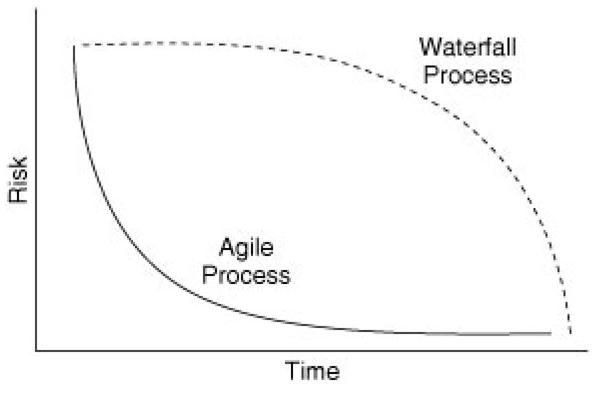
Another way to view this is by looking at incremental value over time, as shown in Figure 1.7. We see that an agile process delivers increasing value over time, whereas the waterfall process delivers no value until the end.

Figure 1.7 *Incremental value*



Delivering value early is good for a couple of reasons. First, if the funding is removed or the project must end early, something of value exists at the point of termination. This is not true for the waterfall process, but it is a primary value in an agile process. Additionally, delivering validated, if partial, functionality early reduces risk, as we see in Figure 1.8. Exactly how early deliveries do this is a topic we will discuss in more detail later, but let us say for now that because we validate each incremental build and we tend to do high-risk things early, we significantly and quickly reduce the project risk. The waterfall process reduces risk slowly at first because you only really know about the quality and correctness of things that you validate, and validation comes only at the end in a waterfall process.

Figure 1.8 *Risk over time*



How can we return incremental value for a system that is delivered externally, such as a cell phone or a missile? Every increment period (which the Harmony/ESW[16] process refers to as a **microcycle**), a system is designed, implemented, and validated in accordance with its mission statement. This mission statement identifies the functionality, target platform, architectural intent, and defect repairs to be included. The incremental functionality is organized around a small set of use cases running on an identified (but not necessarily final) target environment. Through the use of good engineering practice, we can encapsulate away the platform details and ensure that the delivered functionality is correct, given the current target. For example, for one tracking system, our team originally targeted laptops with simulated radars and created actual validated functionality on that environment. Over the course of the project, as hardware became available, we migrated to target hardware of the actual military systems. Through this approach, we had high-quality, testable software earlier than expected.

16. Harmony/Embedded Software. This is one of the members of the IBM Rational Harmony family of processes and is the basis of the content of this book. The process basics will be discussed at some length in Chapter 3.

1.4.3. Satisfied Stakeholders

Stakeholders are simply people who have a stake in the successful outcome of a project. Projects have all kinds of stakeholders. Customers and marketers are focused on the functional benefits of the system and are willing to invest real money to make it happen. Their focus is on specifying to the developers the needs of the users in a realistic and effective fashion. Managers are stakeholders who manage that (real or potential) investment for the company to achieve a timely, cost-effective delivery of said functionality. Their job is to plan and schedule the project so that it can be produced to satisfy the customer and meet the users' needs. The users are the ones who use the system in their work environment and need high-quality functionality that enables their workflow to be correct, accurate, and efficient. All these stakeholders care about the product but differ in the focus of their concern. The customers care how much they pay for the system and the degree to which it improves the users' work. The managers primarily care how much the system costs to develop and how long that effort takes. The users primarily care about the (positive) impact the system makes on their work.

Agile methods provide early visibility to validated functionality. This functionality can be demonstrated to the stakeholders and even delivered. This is in stark contrast to traditional preliminary design review (PDR) and critical design review (CDR) milestones in which text is delivered that describes promised functionality in technological terms. Customers can—and should—be involved in reviewing the functionality of the validated incremental versions of the system. Indeed, the functionality can be implemented using a number of different strategies, depending on what the process optimization criterion is. Possible criteria include the following:

• Highest-risk first

• Most critical first

• Infrastructure first

• Available information first

All other things being equal, we prefer to deliver high-risk first, because this optimizes early risk reduction. However, if the users are to deploy early versions of the system, then criticality-first makes more sense. In some cases, we deploy architectural infrastructure early to enable more complex functionality or product variations. And sometimes we don't have all the necessary information at our fingertips before we must begin, so the things we don't know can be put off until the necessary information becomes available.

1.4.4. Improved Control

Many, if not most, software projects are out of control, to some degree or another. This is largely because although projects are planned in detail, they aren't tracked with any rigor. Even for those projects that are tracked, tracking is usually done on the wrong things, such as SLOC delivered. Thus most projects are either not tracked or track the wrong project properties.

Project tracking requires the answers to three questions:

• Why track?

• What should be tracked?

• How should projects be tracked?

Why track? Project teams that don't know exactly *why* they are tracking project properties rarely do a good job. Only by identifying the goals of tracking can you decide what measures should be tracked and how to implement the tracking procedures.

The biggest single reason for project tracking is that plans are *always* made in the presence of incomplete knowledge and are therefore inaccurate to some degree. Tracking enables the project deviance from plan to be identified early enough to effectively do something about it. Projects should be tracked so that they can be effectively managed, replanned as appropriate, and even scrapped if necessary. You can effectively replan only when you know more than you did when the original plan was made, and that information can come from tracking the right things. Put another way, the fundamental purpose of tracking is to reduce uncertainty and thereby improve project control.

What should be tracked? Ideally, tracking should directly reduce uncertainty in the key project characteristics that relate to the cost, time, effort, and quality of the product; that is, tracking should directly measure cost, time to completion, effort to completion, and defect rates. The problem is that these quantities are not directly measurable.

So projects typically evaluate metrics that are measurable with the expectation that they correlate with the desired project quantities. Hence, people measure properties such as lines of code or defects repaired. The flaw in those measures is that they *do not* correlate strongly with the project criteria. If, at the end of the project, you remove lines of code during optimization, are you performing negative work and reducing the time, cost, or effort? If I don't know exactly how many lines of code I'm going to end up with, what does writing another 10,000 lines of code mean in terms of percent completeness? If I measure cyclomatic complexity, am I demonstrating that the system is correct? The answer is an emphatic *no*.

The problem with many of the common metrics is that while they are easy to measure, they don't correlate well with the desired information. This is because those metrics track against the project implementation rather than the project goal. If you want to measure completeness, measure the number of requirements validated, not the number of lines of code written. If you want to measure quality, measure defect rates, not cyclomatic complexity. The other measures do add incremental value, but the project team needs to focus on achievement of the ultimate goal, not weak correlates.

Agile methods provide the best metrics of all—working, validated functionality—and they provide those metrics early and often. Agile focuses on delivering correct functionality constantly, providing natural metrics as to the quality and completeness of the system over time. This in turn provides improved project control because true problems become visible much earlier and in a much more precise fashion.

1.4.5. Responsiveness to Change

Life happens, often in ways that directly conflict with our opinions about how it ought to happen. We make plans using the best available knowledge, but that knowledge is imprecise and incomplete and in some cases just wrong. The imprecision means that small incremental errors due to fuzziness in the data can add up to huge errors by the end of the project—the so-called butterfly effect in chaos theory.[17] Chaos theory is little more than the statement that most systems are actually nonlinear; by **nonlinear** we mean that small causes generate effects that are not proportional to their size. That sums up software development in a nutshell: a highly nonlinear transfer function of user needs into executable software.

17. See Edward N. Lorenz, *The Essence of Chaos (The Jessie and John Danz Lecture Series)* (Seattle: University of Washington Press, 1996).

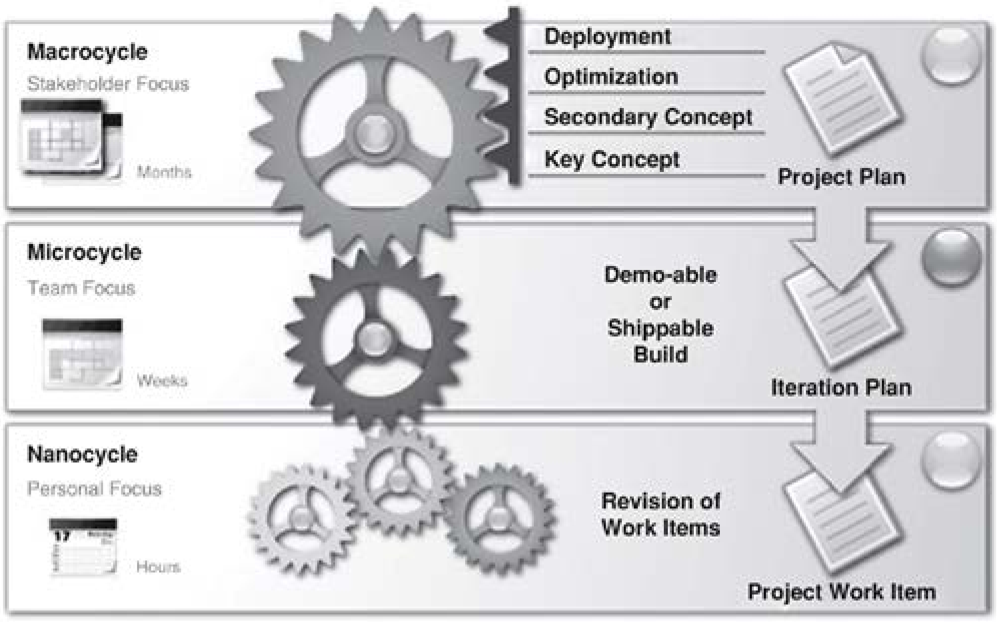
The incompleteness problem means that not only do we not know things very precisely, but some things we don't know at all. I remember one project in which I was working on a handheld pacemaker program meant to be used by physicians to monitor and configure cardiac pacemakers. It was a based on a Z-80-based embedded microcomputer with a very nice form factor and touch screen. The early devices from the Japanese manufacturer provided a BIOS to form the basis of the computing environment. However, once the project began and plans were all made, it became apparent that the BIOS would have to be rewritten for a variety of technically inobvious reasons. Documentation for the BIOS was available from the manufacturer—but only in Japanese. The technical support staff was based in Tokyo and spoke only—you guessed it—Japanese. This little bit of missing information put the project months behind schedule because we had to reverse-engineer the documentation from decompiling the BIOS. It wouldn't be so bad if that was the only time issues like that came up, but such things seem to come up in every project. There's always something that wasn't planned on—a manufacturer canceling a design, a tool vendor going out of business, a key person being attracted away by the competition, a change in company focus, defects in an existing product sucking up all the development resources, . . . the list goes on and on.

Worst, in some way, is that knowledge you have about which you are both convinced and incorrect. This can be as varied as delivery dates, effort to perform tasks, and availability of target platforms. We all make assumptions, and the law of averages dictates that when we make 100 guesses, each of which is 90% certain, 10 are still likely to be wrong.

Despite these effects of nonlinearity, incompleteness, and incorrectness, we still have to develop systems to meet the stakeholders' needs at a cost they're willing to pay within the time frames that meet the company's schedules. So in spite of the nonlinearity, we do our best to plan projects as accurately as possible. And how well do we do that? The answer, from an industry standpoint, is "not very well at all."

The alternative to plan-and-pray is to plan-track-replan. Agile methods accept that development plans are wrong at some level and that you'll need to adjust them. Agile methods provide a framework in which you can capture the change, adjust the plans, and redirect the project at a minimal cost and effort. The particular agile approach outlined in this book, known as the Harmony/ESW process, deals with work at three levels of abstraction, as shown in Figure 1.9.

Figure 1.9 *Harmony/ESW timescales*



The smallest timescale, known as the **nanocycle,** is about creation in the hour-to-day time frame. In the nanocycle, the developer works off of the work items list, performs small incremental tasks, and verifies that they were done properly via execution. In this time frame, small changes with local scope can be effectively dealt with in the context of a few minutes or hours.

The middle time frame is called the **microcycle** and focuses on the development of a single integrated validated build of the system with specified functionality. The microcycle time frame is on the order of four to six weeks and delivers formally validated, although perhaps limited, functionality. Changes with medium scope are dealt with in the formal increment review[18] and in the prototype mission statement that identifies the scope for the microcycle iteration.

18. Also known as the “party phase” because it is not only a review, but also a “celebration of ongoing success”—as opposed to a postmortem, which is an analysis designed to discover why the patient died.

The largest time frame is called the **macrocycle**. The macrocycle concerns itself with the beginning and end of the project and primary milestones within that context. The macrocycle is usually 12 to 24 months long and represents a final, or at least significant, customer delivery. At this scope, large-scale changes are managed that may result in significant project replanning.

1.4.6. Earlier and Greater Reduction in Project Risk

The last of the benefits we will discuss in this section has to do with reduction of project risks.

In my experience, the leading cause of project failure is simply ignoring risk. Risk is unavoidable, and attempts to ignore it are rarely successful. I am reminded of a company I consulted to that wanted help. The development staff of this medical device company had been working 55 to 60 hours per week *for 10 years* and had never made a project deadline. They asked that I come and see if I could identify why they were having such problems. As it happens, they did develop high-quality machines but at a higher-than-desirable development cost and in a longer-than-desirable time frame. They consistently ignored risks and had a (informal) policy of refusing to learn from their mistakes. For example, they had a history of projects for fairly similar devices, and it had always taken them five months to validate the machines. However, they, just as always, scheduled one month for validation. They refused to look at why projects were late and adjust future plans to be more reasonable.

In this context, risk means the same thing as it did in the earlier discussion of safety. It is the product of the severity of an undesirable situation and its likelihood. For a project, it is undesirable to be late or over budget or to have critical defects. We can reduce project risks by *managing them*. We manage them by identifying the key project risks and their properties so that we can reduce them. Risks are managed in a document called either a **risk list** or a **risk management plan.** As we will learn later, this risk list contains an ordered list of conditions, severities, likelihoods, and corrective actions known as **risk mitigation activities** (RMAs). These activities are scheduled into the iterations primarily in order of degree of risk (highest-risk first).

For example, if the risk is that CORBA[19] is too slow to handle the throughput required, an early prototype[20] should include some high-bandwidth data exchange and the performance can be measured. If it is found that CORBA does, in fact, provide inadequate performance, other technical solutions can be explored. Because the problem was discovered early, the amount of rework in that case will be less than in a traditional "Oh, God, I hope this works" development approach. In agile methods this kind of an experiment is known as a **spike**.[21]

19. Common Object Request Broker Architecture, an OMG standard.

20. A **prototype** is a validated build of the system produced at the end of an iteration microcycle. It contains a subset (but usually not all) of the real code that will ship in the system. Unless specifically described as such, we do not mean a **throwaway** prototype, which is an executable produced to answer a specific set of questions but will not be shipped in the final product.

21. In agile-speak, a spike is a time-boxed experiment that enables developers to learn enough about an unknown to enable progress to continue. See www.extremeprogramming.org/rules/spike.html.

The risk list is a dynamic document that is reviewed at least every iteration (during the party phase[22]). It is updated as risks are reduced, mitigated, or discovered. Because we're focusing attention on risk, we can head off an undesirable situation before it surprises us.

22. See Chapter 9.

1.4.7. Efficient High-Quality Development

High quality is achieved by the proper application of agile methods but in a different way from traditional industrial processes. This is again a dynamic, rather than a ballistic, approach. Agile achieves high quality through continuous execution, continuous integration, and continuous testing—begun as early as possible. Agile holds that the best way not to have defects in a system is not to systematically test them out but to not introduce them into the software in the first place (a topic I will address in more detail in upcoming chapters).

Efficiency is why most people in my experience turn to agile methods. In fact, agile methods have sometimes been thought to sacrifice quality and correctness in the pursuit of development efficiency. It is true that agile methods are a response to so-called heavyweight processes that emphasize paper analysis and ballistic planning over early execution and risk reduction. Nevertheless, agile emphasizes efficiency because it is a universal truth that software costs too much to develop and takes too long. A good agile process is *as efficient as possible* while achieving the necessary functionality and quality. Agile often recommends lighter-weight approaches to achieve a process workflow.

1.5. Agile Methods and Traditional Processes

Agile methods differ from traditional industrial processes in a couple of ways. Agile planning differs from traditional planning because agile planning is—to use the words of Captain Barbossa[23]—"more what you'd call a guideline." Agile development tends to follow a depth-first approach rather than the breadth-first approach of traditional methods. Another key agile practice is test-driven development (TDD), which pushes testing as far up front in the process as possible. Finally, agile embraces change rather than fearing it.

23. *Pirates of the Caribbean: The Curse of the Black Pearl* (Walt Disney Pictures, 2003).

1.5.1. Planning

It is a common and well-known problem in numerical analysis that the precision of a computational result cannot be better than that of the elements used within the computation.[24] I have seen schedules for complex system development projects that stretch on for years yet identify the completion time *to the minute*. Clearly, the level of knowledge doesn't support such a precise conclusion. In addition (pun intended ☺), errors accumulate during computations; that is, a long computation *compounds* the errors of its individual terms.

24. Assuming certain stochastic properties of the error distribution, of course.

If you are used to working in a traditional plan-based approach, agile methods may seem chaotic and intimidating. The problem with the standard waterfall style is that although plans may be highly detailed and ostensibly more complete, that detail is *wrong* and the computed costs and end dates are in error.

Further, not only is the information you have about estimates fuzzy at best, it is also usually systematically biased toward the low end. This is often a result of management pressure for a lower number, with the misguided intention of providing a "sense of urgency" to the developers. Sometimes this comes from engineers with an overdeveloped sense of optimism. Maybe it comes from the marketing staff who require a systematic reduction of the schedule by 20%, regardless of the facts of the matter. In any event, a systematic but uncorrected bias in the estimates doesn't do anything but further degrade the accuracy of the plan.

Beyond the lack of precision in the estimates and the systematic bias, there is also the problem of *stuff you don't know and don't know that you don't know*. Things go wrong on projects—*all* projects. Not all things. Not even most things. But you can bet money that *something* unexpected will go wrong. Perhaps a team member will leave to work for a competitor. Perhaps a supplier will stop producing a crucial part and you'll have to search for a replacement. Maybe as-yet-unknown errors in your compiler itself will cause you to waste precious weeks trying to find the problem. Perhaps the office assistant is really a KGB[25] agent carefully placed to bring down the Western economy by single-handedly intercepting and losing your office memo.

25. Excuse me, that should be *FSB* now.

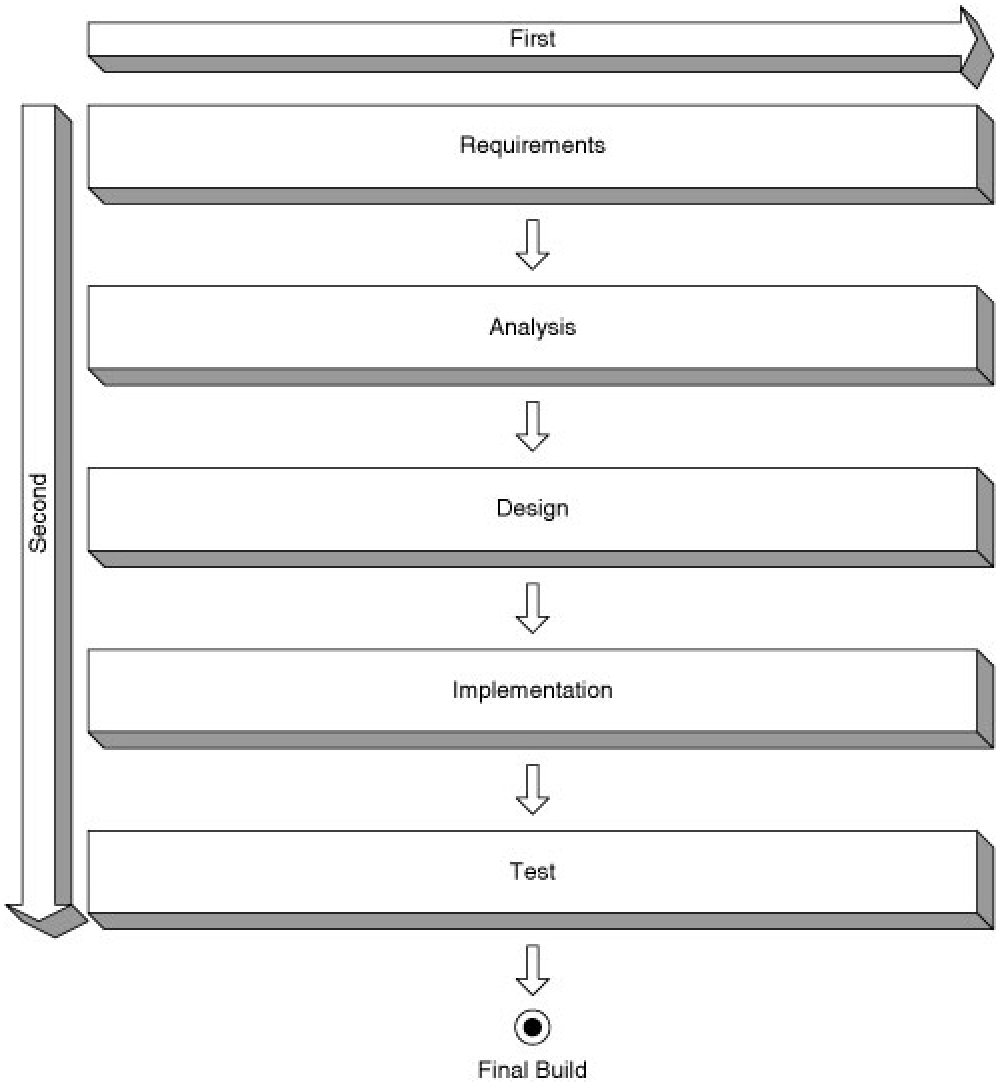
It is important to understand, deep within your hindbrain, that planning the unknown entails inherent inaccuracy. This doesn't mean that you shouldn't plan software development or that the plans you come up with shouldn't be as accurate as is needed. But it does mean that you need to be aware that they contain errors.

Because software plans contain errors that cannot be entirely removed, schedules need to be tracked and maintained frequently to take into account the "facts on the ground." This is what we mean by the term **dynamic planning**—it is planning to track and replan when and as necessary.

1.5.2. Depth-First Development

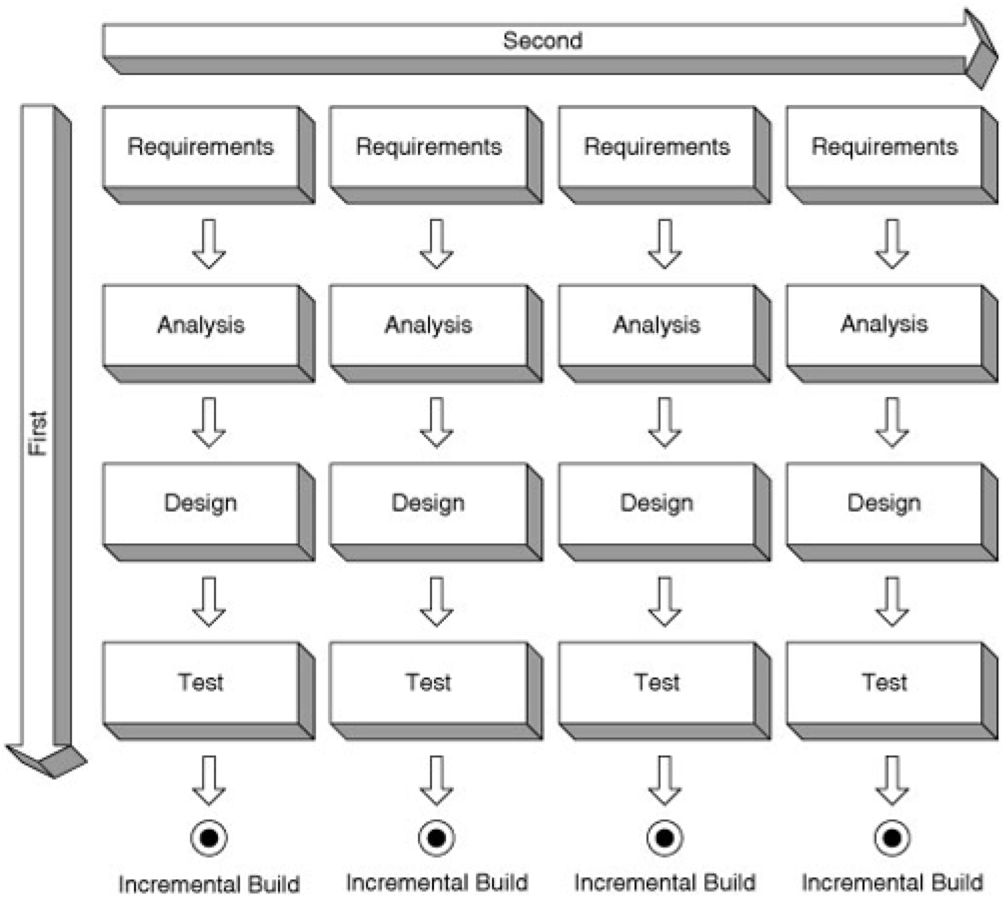
If you look at a traditional waterfall approach, such as is shown in Figure 1.10, the process can be viewed as a sequential movement through a set of layers. In the traditional view, each layer (or "phase") is worked to completion before moving on. This is a "breadth-first" approach. It has the advantage that the phase and the artifacts that it creates are complete before moving on. It has the significant *dis*advantage that the basic assumption of the waterfall approach—that the work within a single phase can be completed without significant error—has been shown to be incorrect. *Most* projects are late and/or over budget, and at least part of the fault can be laid at the feet of the waterfall lifecycle.

Figure 1.10 *Waterfall lifecycle*



An incremental approach is more “depth-first,” as shown in Figure 1.11. This is a “depth-first” approach (also known as spiral development) because only a small part of the overall requirements are dealt with at a time; these are detailed, analyzed, designed, and validated before the next set of requirements is examined in detail.[26] The result of this approach is that any defects in the requirements, through their initial examination or their subsequent implementation, are uncovered at a much earlier stage. Requirements can be selected on the basis of risk (high-risk first), thus leading to an earlier reduction in project risk. In essence, a large, complex project is sequenced into a series of small, simple projects. The resulting incremental prototypes (also known as **builds**) are validated and provide a robust starting point for the next set of requirements.

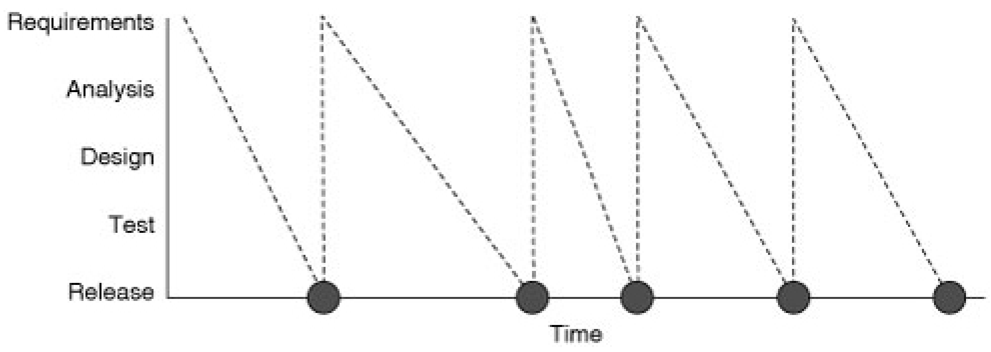
Figure 1.11 *Incremental spiral lifecycle*



26. The astute reader will notice that the "implementation" phase has gone away. This is because code is produced throughout the analysis and design activities—a topic we will discuss in much more detail in the coming chapters.

Put another way, we can "unroll" the spiral approach and show its progress over linear time. The resulting figure is a sawtooth curve (see Figure 1.12) that shows the flow of the phases within each spiral and the delivery at the end of each iteration. This release contains "real code" that will be shipped to the customer. The prototype becomes increasingly complete over time as more requirements and functionality are added to it during each microcycle. This means not only that some, presumably the high-risk or most critical requirements, are tested first, but also that they are tested more often than low-risk or less crucial requirements.

Figure 1.12 *Unrolling the spiral*



1.5.3. Test-Driven Development

In agile approaches, testing is the “stuff of life.” Testing is *not* something done at the end of the project to mark a check box, but an integral part of daily work. In the best case, requirements are delivered as a set of executable test cases, so it is clear whether or not the requirements are met. As development proceeds, it is common for the developer to write the test cases before writing the software. Certainly, before a function or class is complete, the test cases exist and have been executed. As much as possible, we want to automate this testing and use tools that can assist in creating coverage tests. Chapter 8 deals with the concepts and techniques for agile testing.

1.5.4. Embracing Change

Unplanned change in a project can occur either because of the imprecision of knowledge early in the project or because something, well, *changed.* Market conditions change. Technology changes. Competitors’ products change. Development tools change. We live in a churning sea of chaotic change, yet we cope. Remember when real estate was a fantastic investment that could double your money in a few months? If you counted on that being true forever and built long-range inflexible plans based on that assumption, then you’re probably reading this while pushing your shopping cart down Market Street in San Francisco looking for sandwiches left on the curb. We cope in our daily lives because we know that things will change and we adapt. This doesn’t mean that we don’t have goals and plans but that we adjust those goals and plans to take change into account.

Embracing change isn’t just a slogan or a mantra. Specific practices enable that embracement, such as making plans that specify a range of successful states, means by which changing conditions can be identified, analyzed, and adapted to, and methods for adapting what we do and how we do it to become as nimble and, well, *agile,* as possible.

In the final analysis, if you can adapt to change better than your competitors, then evolution favors *you*.[27]

27. As the saying goes, "Chance favors the prepared mind." I forget who said it first, but my first exposure to it was Eric Bogosian in *Under Siege 2: Dark Territory* (Warner Bros., 1995).

1.6. Coming Up

This chapter provided some basic background information to prepare you for the rest of the book. Agile approaches are important because of the increasing burden of complexity and quality and the formidable constraint of a decreasing time to market. Following the discussion of the need for agility, the context of real-time systems was presented. The basic concepts of timeliness, such as execution time, deadline, blocking time, concurrency unit, criticality, and urgency, are fundamental to real-time systems. Understanding them is a prerequisite to understanding how agile methods can be applied to the development of such systems. The discussion went on to the actual benefits of agile methods, such as lowered cost of development, improved project control, better responsiveness to change, and improved quality. Finally, agile and traditional methods were briefly compared and contrasted. Agile methods provide a depth-first approach that embraces change and provides a continual focus on product quality.

The next chapter will introduce the concepts of model-driven development. Although not normally considered "agile," MDD provides very real benefits in terms of conceptualizing, developing, and validating systems. MDD and agile methods work synergistically to create a state-of-the-art development environment far more powerful than either is alone.

Chapter 3 introduces the core principles and practices of the Harmony/ESW process. These concepts form the repeated themes that help define and understand the actual roles, workflows, tasks, and work products found in the process.

Then, in Chapter 4, the Harmony/ESW process itself is elaborated, including the roles, workflows, and work products. Subsequent chapters detail the phases in the Harmony/ESW microcycle and provide detailed guidance on the implementation of those workflows.

Chapter 2
Concepts, Goals, and Benefits of Model-Driven Development

One of the best things about agile is that it isn't a stand-alone process that requires that you follow it dogmatically. Actually, agile methods are not, strictly speaking, a process at all, but a coherent manner of thinking, planning, and practicing that can be applied to many different processes and technologies. The reason why this is so beneficial is that it can be combined with other processes and technologies that are worthwhile to create an overall methodological approach that is superior to the individual approaches performed independently. That is to say, agile methods are synergistic with many other approaches and technologies. One of these is Model-Driven Architecture, which has demonstrated significant advantages, particularly in complex systems development, when it comes to enabling the creation and maintenance of real-time and embedded systems.

2.1. What Is MDA?

The Object Management Group (OMG) is a nonprofit technology consortium that creates and maintains technology standards such as Common Object Request Broker Architecture (CORBA) and the Unified Modeling Language (UML). These standards are created by self-organizing teams that contain OMG members from industry, the academy, and the government, bound together by a common interest in a technology or domain. The OMG is primarily focused on enterprise-level integration and interoperability of technologies and domains. The UML will be used in this book as the lingua franca for capturing and representing models of software and systems. UML is a prominent standard but won't be described in this book, as there are many good books about it already.[1] But the capability of representing the system using one or more coherent models is just one aspect of interoperability. Another is an approach to modeling that enables interoperability, and that is the focus of another standard: Model-Driven Architecture (MDA).[2]

1. Interested readers are referred to my *Real-Time UML, Third Edition.*

2. MDA is a trademarked term; the generic term is model-driven development (MDD). We will treat the two as equivalent here.

Heterogeneous technologies exist today, and the amount of diversity in technologies, platforms, middleware, and execution environments is only increasing. A typical enterprise of interacting systems must contend with different forms of the following:

• Operating systems

• CPUs

• Networks

• Middleware

• Graphical user interface (GUI) development environments

• Programming libraries

• Programming languages

Reaching a consensus on the set of technologies is an impossible task. First of all, systems are owned by different companies and teams, and they have an investment in the technologies they've employed. Second, those systems often must fit into many different enterprise architectures, so the companies and teams select technologies they view as optimal for their system's needs. To be successful, enterprise architecture cannot focus on the consistency of implementation but must instead focus on interoperability of interfaces and semantic content.

MDA preserves the OMG's focus on integration and interoperability. The OMG recognizes that it is not only unreasonable but also impossible to expect everyone to adopt a single technological solution enterprise-wide. Solutions are adopted and implemented for many reasons, and different aspects of an enterprise require different kinds of solutions and different solution technologies and approaches. The enterprise architecture has the unenviable task of trying to make all these optimized point solutions work together in a smooth way. Each of these system solutions has its own means of distribution, mapping to different hardware solutions, with different concurrency and resource architectures, and with different needs for safety and reliability measures. Each system in such an enterprise has its own specialized concerns in addition to the need to interoperate with peer systems that work under different constraints.

MDA is a standard that focuses on the system or point solution and how it can be made to interoperate effectively. To enable this interoperability, it focuses on one key idea: *the separation of the specification of functionality from the specification of the implementation of that functionality*. This concept embraces interoperability in that an application can be easily ported to different execution, distribution, and deployment environments because its functionality is captured in such a way that it doesn't depend on those implementation concerns. Further—and this is very important for systems that live a long time, such as military and aerospace systems—as the underlying implementation technology evolves "underneath" the application, the functionality is far more easily maintained.

With MDA, the models are the key artifacts of the system. Models—and especially models captured in the UML—are formal representations of different aspects of the system being constructed. Note the use of the term *formal*. By that we indicate that we are not talking about drawings sketched on a napkin over lunch. By formal we mean a well-specified representation of aspects of the system. This applies to requirements as much as it does to implementation and everything in between. MDA encourages the use of formal models because they are "machine-processable"; that is, they can be manipulated by tools because their meaning is clear and unambiguous, at the level of abstraction presented.

The capability of automatically processing the models is an essential facet of the models because it not only saves considerable human effort but also reduces the cost while improving the quality of the software. Specifically, it means that it is possible to validate a model's semantic completeness and accuracy in an automated or semiautomated way. It is also possible to convert a model into other forms, such as:

> Note
>
> Note that the notion of clarity is always related to the level of abstraction. The C statement
>
> ```
> for (j=0; j<10; j++)
> printf(j);
> ```
>
> is clear at the level of abstraction of the statement but can be implemented in many different ways in the CPU's machine language.

• Into source code, a process commonly known as **code generation**

• From source code into a mode, a process commonly known as **reverse engineering**

• From a model in one form or at one level of abstraction into a model in a different form or at a different level of abstraction, a process commonly known as **model transformation**

2.2. Why Model?

When we talk about model-based development, we are implicitly contrasting it with a code-centric approach. For us to change our approach in such a radical fashion, there must be compelling reasons.

Source code is an inherently one-dimensional, highly detailed view of the structure of a system. This makes it extremely difficult to look at different points of view, such as the architectural aspects, the functionality, or the behavior. Large source-code-based systems are difficult to understand, and introducing new staff to such systems is expensive and time-consuming. Large-scale architectural pieces are not represented explicitly, and they must be inferred from reading thousands of lines of text. Source-code-based systems may embody their designs, but the designs are not directly expressed, so the correspondence between the design intent and the code is difficult to ensure. Further, there is no separation between the "essential" characteristics that absolutely must be in the system and the optional technological decisions added to implement the system. And last, the source code hides semantic relations to other parts of the system, and so many such systems are exceedingly fragile because they rely on constraints and relations that are not obvious. Over time, code-based systems decay because the implicit design becomes distorted, unstated assumptions and constraints are forgotten, and "good programming" practices such as encapsulation are violated.

Models enable us to explicitly and clearly state the functionality and design intent in a language that is at a higher level of abstraction than the source code. With the UML, a graphical modeling language, two-plus dimensions are available to show how the pieces are structured, how they behave, and how they relate to other parts of the system. The use of models can bring many advantages, including:

• Visualization

• Comprehension

• Communication

• Consistency and correctness

• Provability and testability

Visualization is enhanced with modeling because graphical models use two-plus dimensions to depict the functionality, structure, and behavior of the system. Graphical models enable you to construct views of different aspects (e.g., functional, structural, behavioral, constraints, etc.) and also to show these at different scales (e.g., system scale, subsystem scale, component scale, class scale, function scale, or data scale). A modeling tool can ensure that the semantics of all the views are consistent by maintaining the model repository automatically as you draw diagrams and add, remove, or modify semantic elements.

In addition, it is possible to explicitly capture "missions"—concepts about the system that are important to understand—and create diagrams that demonstrate those concepts. For example, you may create diagrams to show each of the following:

• *Concurrency architecture*—what the tasks are in the system, what resources they share, and how they interact

• *Distribution architecture*—how objects are allocated across address spaces, how they communicate and collaborate

• *Deployment architecture*—how software objects map to hardware

• *Safety and reliability architecture*—how redundancy is managed to make the system robust in the presence of faults

• *Subsystem architecture*—the large-scale pieces of the system and how they interact

• *Class taxonomies*—the generalization taxonomy of a related set of classes

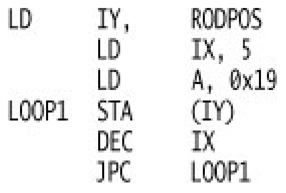
• *Collaborations*—how classes interact to realize a system-level capability

• *Class structure*—parts within a structured class

• *Instances and links* that exist in the running system at a specific point in time

Models aid in the initial construction of the system by providing clear views of what is to be accomplished. Later, during maintenance, these views enable developers to understand the possibly complex relations among the elements to isolate the points of change within the system. Models help in testing the system because it is clear how changes in one aspect impact others. Because the views enable the important functional, structural, and behavioral aspects to be clearly seen, it becomes easier to introduce new staff to the system and to communicate these aspects to relevant stakeholders.

Comprehension is enhanced through the use of models because, as discussed above, views may be constructed to show important or complex aspects of the system, at different levels of abstraction. A model is always an abstraction of the subject, permitting the modeler to focus on the significant aspects of a system. The art of modeling lies in rendering the important aspects while eliding the inessential ones. When this is done, we can see, for example, the conditions under which the control rods are in the reactor core, rather than seeing the following:



```
LD     IY,    RODPOS
       LD     IX, 5
       LD     A, 0x19
LOOP1  STA    (IY)
       DEC    IX
       JPC    LOOP1
```

It is important to be able to see and understand the implementation, but it is at least as important to be able to understand the semantics of the system or application without worrying about which CPU register was used as a loop variable. Because models can be viewed at different levels of abstraction, they provide an outstanding tool to help us understand just what the heck is going on in a way that the source code cannot.

Systems are specified, designed, and constructed by teams of people who must communicate to work together. Models improve communication by enabling different stakeholders to focus on the aspects of the system relevant to them, whether it is the required functionality, the structure, the behavior, test cases, and so on. Models enable views to be constructed that are consistent with the model semantics that concentrate on what the stakeholder needs to understand.

In a model-based system, code reviews are sometimes done, but models replace code as the primary work artifact. In reviews, diagrams depict different aspects of the relevant portions of the model for analysis, review, and discussion. For example, it is common to conduct reviews around the realization of a use case. The views (diagrams) used to form the basis of such a review might be:

• Use case diagram

• Use case details, including:

° Sequence diagrams

° State diagram for the use case

° Requirements diagram (SysML diagram used to show trace relations between requirements and model elements)

• Class diagram showing the collaboration of elements realizing the use case, with related behavioral views, including:

° Sequence diagrams

° State diagrams for reactive classes

° Activity diagrams for complex methods

Beyond reviews, the graphical views of the model can provide excellent assistance in understanding what the elements of the system are, how they relate to each other, and how they behave, both together and in isolation.

Some argue that source code is the best place to look for consistency and correctness, because compilers perform compile-time checking. Leaving aside for the moment weakly typed languages such as C, even strongly typed languages such as Ada can check for only a very basic level of consistency and correctness. Compilers can check that statements conform to the language syntax and, in the case of strongly typed languages, that type consistency is maintained. Compilers cannot efficiently check for ill-formed semantic constructs such as deadend states, unreachable states, or problem-domain-specific concerns. Models improve consistency and correctness by enabling the developer to focus on the semantic content without worrying about the implementation details.

While third-generation source code languages (3GLs) are unambiguous, they represent the application semantics munged together with implementation details. If the application fails, was it because the implementation was incorrect or because the design was flawed? With models, the functional intent of the system can be captured (typically with use case sequence and state diagrams), and these can form the basis for the validation tests when the system is developed. In a traditional, source-code-oriented approach, the tests are specified as textual documents and must be translated into manually performed tests or into a machine-executable form. With model-based development, the requirements can be captured in a representation that can be directly executed. The UML Testing Profile[3] provides a standard way to use UML to specify and characterize tests with the UML.

3. "The UML 2 Testing Profile Version 1.0," OMG document formal/05-07-07 at http://www.omg.org/.

While modeling has many advantages, the fact that you create models doesn't necessarily result in those benefits. In my own experience, to reap the advantages discussed above requires that the models be:

• Well formed, i.e., consistent with the modeling language in which they are specified (e.g., UML)

• Complete enough, i.e., unambiguously representing the breadth of the aspects of the system they exist to elucidate

• Semantically rich, i.e., containing precise statements regarding the structure, functionality, or behavior of the relevant elements

• Well organized, i.e., structured into parts that enable comprehension and decomposition for the teams involved in the creation or use of the model

• Executable, i.e., capable of being executed or simulated to ensure that the model meets its semantic intent at the specified level of abstraction

• Unambiguously and bidirectionally related to the code, i.e., the code is clearly a view of the model at a detailed level of abstraction and this relation is maintained automatically

It should be noted that I am not in any way arguing that source code is bad or unnecessary. It is clearly a crucial view of the system software. It is, however, insufficient for optimal software productivity and creativity. But to be useful, models should not be "napkin design" but instead should be semantically rich and detailed enough to meet the need. Used in this way, models have proven themselves to improve productivity and efficiency by enabling developers to focus on the right things at the right level of abstraction at the right time.

2.3. Key Concepts of MDA

MDA includes a number of fundamental concepts. Together, they provide a coherent view of OMG's view of how to create and maintain interoperable enterprise architectures.

2.3.1. Model

The OMG defines the term **model** to be "a description or specification of that system and its environment for some purpose." The salient aspects of a model are the following:

• A model is a simplification of the thing that it models

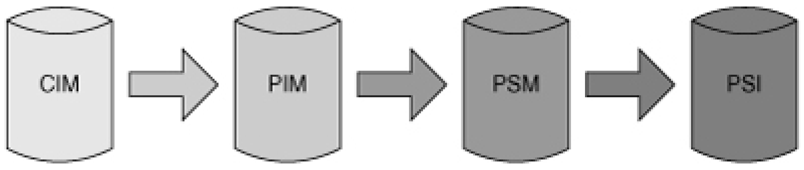
• A model has a purpose or intent

• The model focuses on the aspects relevant to its purpose

That is, it is common to have different models of the same thing for different purposes. MDA uses this concept to implement a well-established idea—the separation of the essential aspects of the thing from how those aspects are implemented.

MDA defines four uses for models in its context (see Figure 2.1). The **computation-independent model** (CIM) doesn't show the pieces within a system that perform the necessary computation but instead focuses on the required functionality of the system. The **platform-independent model** (PIM) focuses on the essential parts of the system and their essential behavior but does not focus on platform-specific details such as the operating system, CPU, or distribution infrastructure. The **platform-specific model** (PSM) includes the elided platform information and represents the system targeted to a specific execution environment. The **platform-specific implementation** (PSI) is the source code of the PSM and may be generated from the PSM automatically. Of these key models, the PIM and the PSM get most of the attention because it is there that automated transformations provide the most benefit.

Figure 2.1 *Key models in MDA*



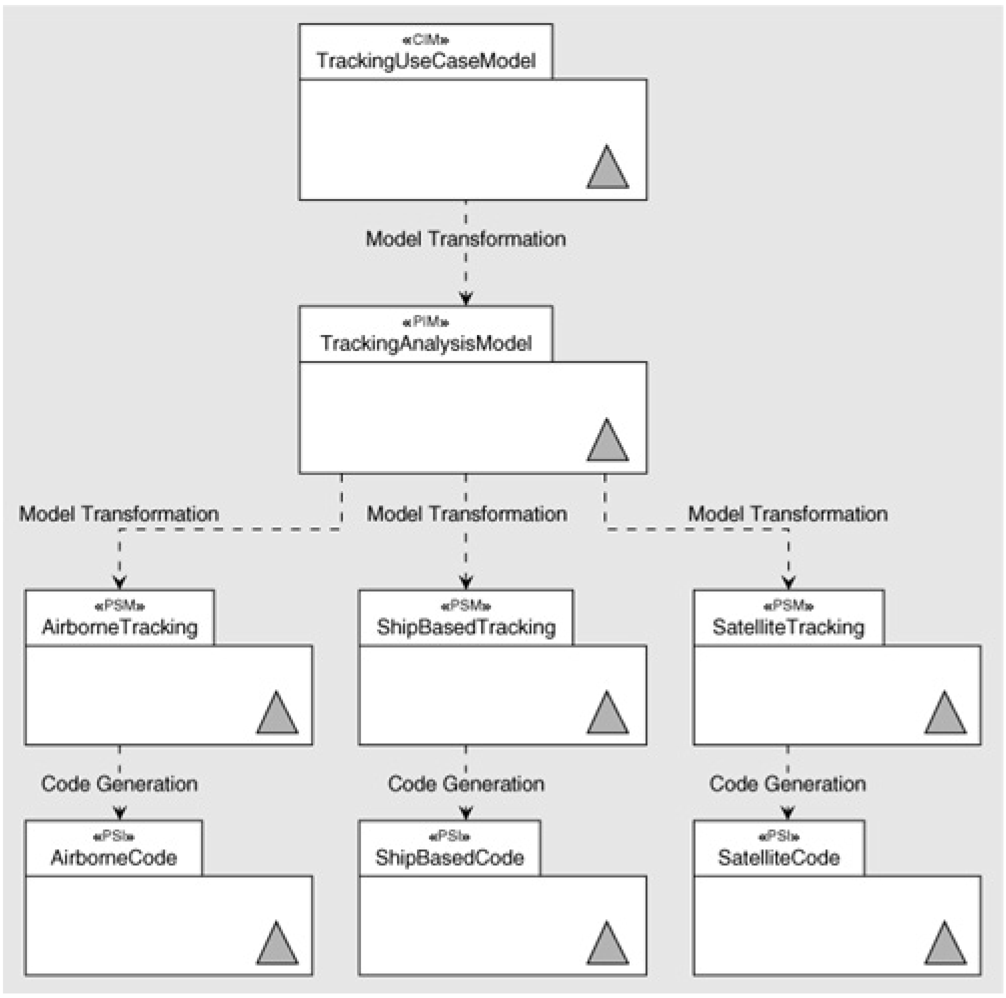
Models are usually represented in the UML, but other modeling languages can be used instead of, or in conjunction with, the UML model. For example, safety analysis is often performed using FTA or FMEA. These languages can be used along with UML models to supply useful views not available within the UML.

A minimalist UML model consists of three types of elements: classes (and their features and relations), state machines for some of the classes, and typical and exceptional interactions of instances of those classes, usually shown on sequence diagrams. In addition to these basic elements, a great deal more can be added, such as use cases, activity models, requirements and their relations, and so on.

It should be noted that the model content is held in the model repository and is exposed in diagrams. The diagrams are *not* the model but are really just useful views into the model repository. Thinking that the diagrams are the primary aspect is a mistake many new to UML make. It's not about the diagrams; it's about what's *on* the diagrams!

The models are created in sequence through a process known as model transformation. Figure 2.2 shows a typical set of models. The `TrackingUseCase-Model` represents the functionality and QoS requirements for a tracking system. It is given the stereotype «`CIM`» to indicate the type of model it is. The use case model contains traceability links from textual requirements to the model's use cases, interactions, and state machines (and associated diagrams). This CIM is transformed into an analysis model that represents the computational model that realizes the requirements.

Figure 2.2 *Model transformations*



The `TrackingAnalysisModel` has the stereotype «`PIM`» to indicate that it is a platform-independent model. The PIM contains essential structural elements (classes, types, and relations), interactions, algorithms, and state machines to realize the computations necessary to perform the tracking requirements. It does not include the platform-specific details such as the CPU, operating system, middleware, network structure, or even the specific sensor platforms used.

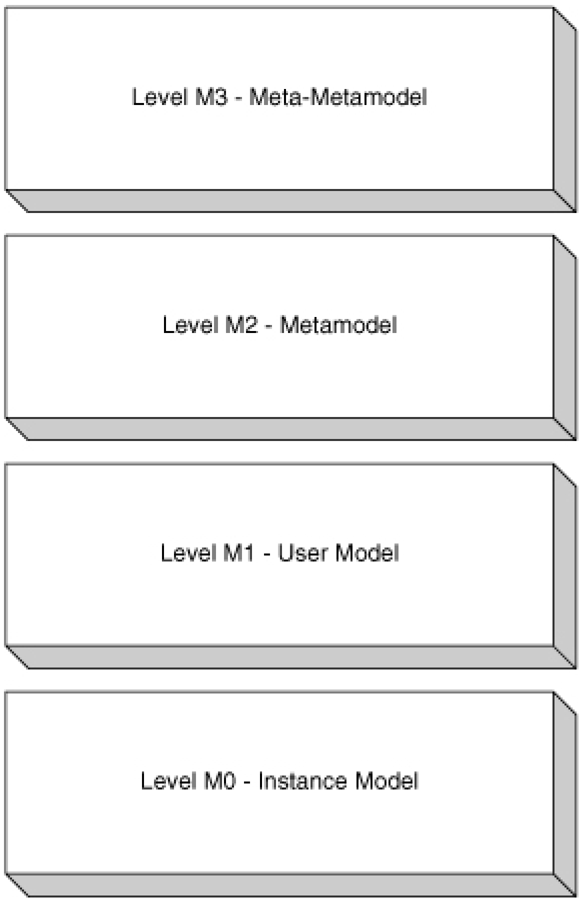
Normally, there is a single PIM realizing a given CIM. It is not uncommon, however, for multiple PSMs to realize a given PIM. Indeed, much of the value of the approach is realized only when this is the case. In this case, we've shown three different PSMs: One is meant for mounting on an airborne platform such as a helicopter, one is meant for mounting on a water vessel (ship), and one is meant for mounting on a satellite. The different platforms have different CPUs, operating systems, middleware, networks, and sensors, but all are derived from the same computational model (PIM).

Each PSM is usually implemented with a single PSI. The PSI contains all the code, data, and configurations necessary to implement the PSM.

2.3.2. Metamodel

A **metamodel** is a model of a model. In English, this means that the language used to define the system (UML) is itself defined by a formal semantic model. This brings the expressive power of modeling not only to the application domain but also to the specification of the language that users employ to design their systems. The UML is based on a four-tier metamodel architecture, as shown in Figure 2.3. The bottom level, level M0, is called the **instance model.** This model represents the elements that exist as the system runs on the real-world platform. The instance model is created by compiling the code generated from the level-M1 model, known as the **user model**. The CIM, PIM, and PSM are all user models and so are at level M1. Level M2 is the **metamodel,** and it defines the language in which the user model is captured. UML is a language defined by its level-M2 metamodel. The UML metamodel itself is defined in a more basic modeling language, which is the **meta-metamodel** (level M3). It is defined in terms of the MetaObject Facility (MOF) and is a language for defining metamodels.

Figure 2.3 *Four-level metamodel architecture*



As developers, we care mostly about the level-M1 model and use it to generate the level-M0 model. However, you may care about the other levels if you do model transformations.

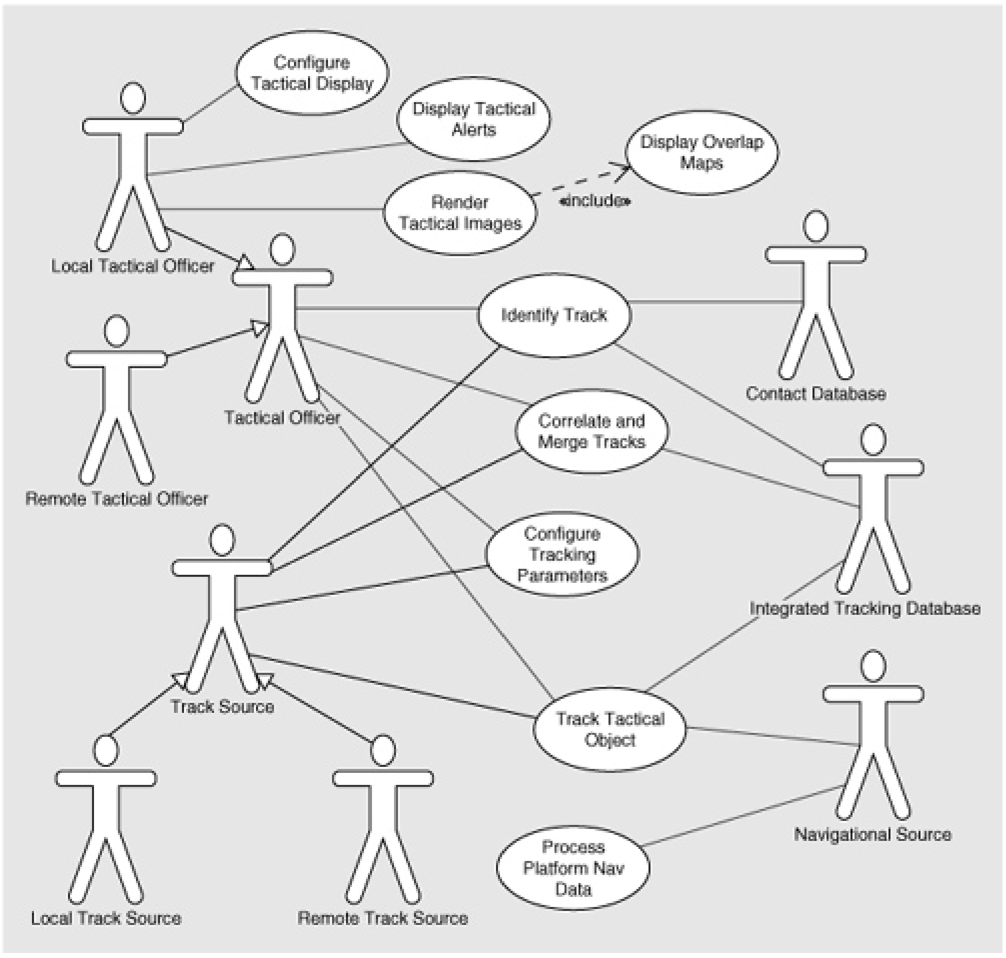
2.3.3. CIM

The CIM models the required functionality without identifying the system elements necessary to perform the computation to achieve it. That is to say, it represents a domain model, usually built by a requirements (or business) analyst or subject matter expert (SME).

The key element within the CIM is the use case. The use case represents a coherent usage of the system and serves as an organizational unit for requirements, as we will see in some detail later in Chapter 6, "Agile Analysis." Of course, the use case itself is just a named oval; we need far more information than that! The use case is bound to requirements via `«trace»` dependencies and is elaborated in the model with state machines detailing the system states and modes related to the execution of the system usage, and (usually many) interactions shown on sequence diagrams. The set of use case diagrams is like a table of contents for a book that defines the system requirements.

Figure 2.4 shows a typical use case diagram from such a CIM. The named ovals on the diagram represent the use cases. The stick figures represent actors—elements in the system environment that interact with the system during the use case execution. The lines between the actors and the use cases are associations, indicating that messages (events or data) are exchanged between the system and the actor when it executes the use case.

Figure 2.4 *CIM use case diagram*



2.3.4. PIM

The PIM is a computational model that is platform-independent. In older technology, this is called the **essential** or **analysis** model. The PIM contains the essential semantic elements, modeled as classes, their essential relation, and their essential behavior.

What is “essential” in this context? One way to think about it is that it is a structural or behavioral aspect that must be true in *any* acceptable design solution. The elements must be able to perform the required functionality by collaborating. For example, if you’re building a bank accounting system, you’ll need classes such as the following:

- Customer

◦ Attributes

- Name
- Tax ID
- Address
- Account

◦ Attributes

- Balance
- Interest rate
- Date opened
- Date closed

◦ Behaviors

- Credit
- Debit
- Open
- Close
- Transaction

◦ Attributes

- Amount
- Source account
- Target account

- `Date`
- `Time`
- `CreditTransaction` (subclass of `Transaction`)
- `DebitTransaction` (subclass of `Transaction`)
- `TransferTransaction` (subclass of `Transaction`)
- `InterestTransaction` (subclass of `Transaction`)
- `AccountHistory`

And so on. This information (and, of course, a great deal more) is needed by any bank to perform its accounting functionality. These classes also have inherent relations (`AccountHistory` aggregates `Transactions`, `Account` owns `AccountHistory`, `Customer` associates with `Accounts`, etc.). These elements have behavior, as dictated by accounting practice and by law. These elements aren't really free to vary between specific accounting systems, even though the networks that support information transfer may vary.

Similarly, a traffic-light control system has some essential structure (traffic sensors, control lights, etc.) and some required behavior to support different operational modes for the intersection.

The PIM represents these essential properties and elides the elements that can change—those will show up in the PSM. In the bank example, whether an ATM machine is connected via a wireless or a wired network is a part of the PSM and so will be detailed there.

The PIM is usually organized around the use cases but is modeled primarily as a set of class diagrams with support from interaction and behavioral diagrams. It is common—and recommended—that there is at least one diagram depicting the collaboration of PIM classes per use case. There may be many such diagrams, depending on the nature and complexity of the system and its use cases. In addition, other class diagrams may show other aspects of the PIM, such as class generalization taxonomies, contents of packages, internal structure of compound classes, and so on.

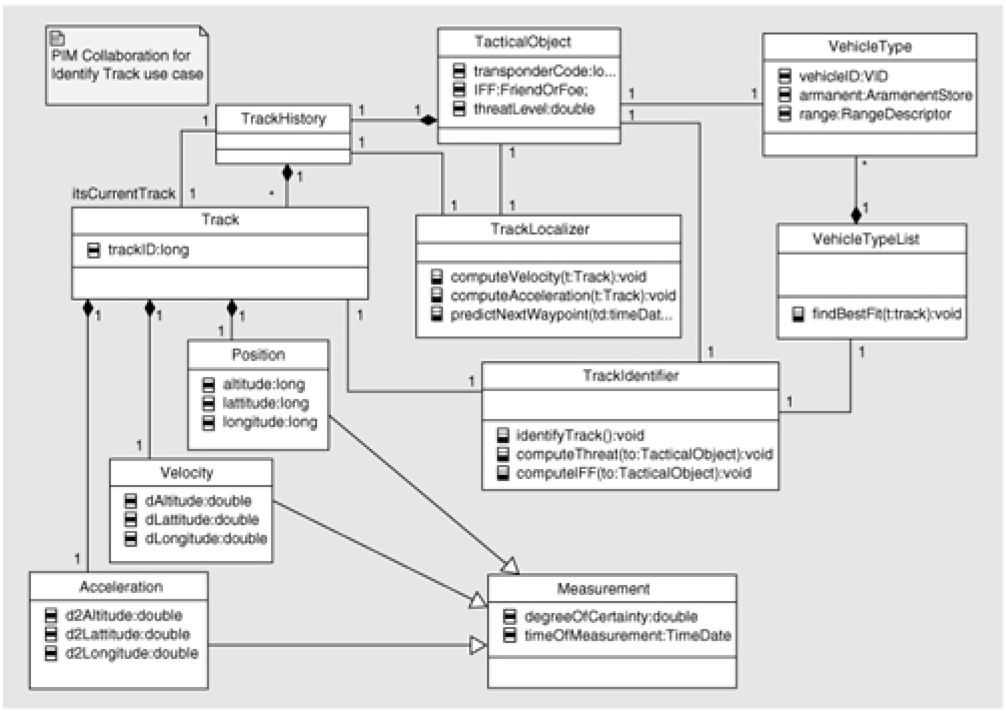
The PIM is correct only if it supports the functionality required by the use case. It is common to have a set of sequence diagrams representing the interactions of the system with the relevant actors for each use case. So, the PIM is correct if, and only if, it can reproduce, through execution, those use case sequence diagrams. Hence, it is highly recommended that the interactions among the collaborating elements be shown in sequence diagrams so that they can be compared to the original use case sequence diagram.

Other highly useful diagrams in the PIM are state machines for various reactive or stateful classes, and activity diagrams for complex operations and functions. State machines sequence the actions (such as invocations of methods) into permitted flows[4] based on the arrival of events of interest. Activity diagrams are like flowcharts on steroids and can depict complex algorithms in a way that is easier to comprehend than the implementation code. Activity diagrams are most commonly used to detail the behavior of class methods above a certain minimum complexity.

4. After all, landing and *then* lowering the landing gear constitutes a *crash*, so order matters!

Figure 2.5 shows a portion of a PIM class model realizing the use case `Identify Track` from Figure 2.4.

Figure 2.5 *PIM class diagram*



It is very important—and we will discuss this concept in great detail later—that the PIM be validated for correctness. Since the measure of correctness is that it can generate the use case sequence diagrams when it runs those scenarios, *the PIM must be executable*. No napkin drawings, no CRC[5] cards—real software that runs. The PIM won't have all the platform-related detail, but unless you can test it, you don't really know whether or not it is correct. And you can test *only* things that actually run.

5. **C**lass, **R**esponsibility, **C**ollaboration cards, a common note-card-based technique used in some circles.

It should be noted that while the MDA specifications talk about moving from the CIM to the PIM by model transformation, this is primarily done by expert human modelers rather than via automation. Later in this book, I will discuss a family of techniques known as “object identification strategies” to create the PIM from the CIM, but that is a human-driven, rather than an automated, process.

2.3.5. PSM

The PSM contains the original semantic content of the PIM but also platform, design, and technology decisions as well. The PIM contains only essential elements but omits these latter aspects. The PSM can best be thought of as a transformation of the PIM in which platforms, designs, and technologies are selected on the basis of achieving the system optimization goals.

Platforms are often chosen because they already exist in the system context and the application must deploy onto them to fit into the operational environment. However, it is often true that the developers have significant leeway to select platform aspects. In this case, platforms should be selected because they are optimal in some sense. Similarly, design solutions are optimizations of analysis solutions. Reusable design solutions are known as **design patterns**, so one way to think about the creation of the PSM is to apply the design patterns to the PIM. The PIM may need to exchange messages and data, but unless the application is an Ethernet router or something similar, exactly *how* the message delivery takes place isn’t an essential property of the PIM but it is a property of the PSM.

There are a number of ways to create the PSM from the PIM (see the section “Common Model Transformations” later in this chapter), but the basic workflow I recommend involves the explicit identification of the optimizations involved. Starting with a high-quality PIM, the steps for creating the PSM are:

1. Identify the optimization criteria, such as worst-case performance, throughput, reliability, weight, heat, design cost, manufacturing cost, etc.

2. Rank the optimization criteria in order of criticality.

3. Identify platforms, design patterns, and technologies that optimize the most important of those criteria at the expense of the least important.

4. Apply the design solutions to the PIM—this is where the different ways of transforming the

PIM occur.

5. Validation of the PSM:

a. Ensure that the PSM didn’t break the working functionality of the PIM.

b. Ensure that the PSM delivers the desired optimizations.

Thus, a good PSM (design) will maximize the weighted set of design criteria

$$optimalDesign = \max\left[\sum DegreeOptimized_j \times Weight_j\right]$$

where

$DegreeOptimized_j$ is the degree of optimization (a larger number is better) for design criteria j

$Weight_j$ is the relative importance of the design criteria j

This optimization is how we select one design approach over another; we select it because it is more optimal overall. The key point is that the PSM is the “design model,” and design is really all about optimization of the required functionality. Transforming the PIM into the PSM is therefore mostly about optimizing the PIM: The PSM is the selection of a particular design solution that optimizes the weighted set of design criteria with respect to the relative importance of each. Secondarily, the PSM is also focused on deploying the application into different operational environments.

Real-time and embedded systems care more about optimality than most. The term **real-time** means “predictably fast enough,” and the primary distinguishing property of real-time and embedded systems is that the qualities of service are not just desirable—they are essential for correctness. Some common design criteria for real-time and embedded systems include:

• Performance

° Worst-case performance

° Average-case performance

° Predictability of performance

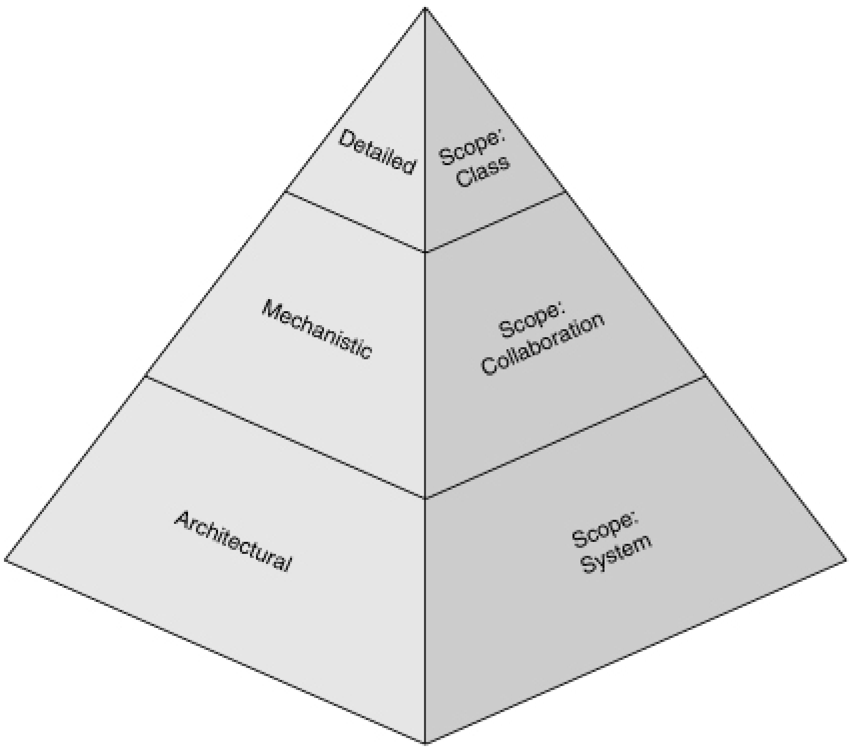
• Schedulability (ability to reliably meet the deadlines)

• Precision and accuracy

- Safety
- Reliability
- Robustness (fault immunity)
- Physical system properties
  - Weight
  - Heat
  - Power consumption
  - Size
- Cost
  - Recurring cost (i.e., cost per shipped item)
  - Design cost
- Maintainability
- Extensibility
- Time to market
- Certifiability (i.e., standards conformance)

In the Harmony/ESW process discussed in this book, the creation of the design PSM occurs at three levels of abstraction (see Figure 2.6). MDA itself focuses on the highest level, architectural design. The Harmony/ESW process identifies five key architectural aspects, and the PSM identifies platforms, design patterns, and technologies in each of these. The design decisions made at this level are called "**strategic**" because they affect most or all of the system. The architectural views organize the PIM collaborations into larger-scale entities. The architectural views in the Harmony/ESW process are discussed in the upcoming section "Harmony's Five Key Architectural Views."[6]

Figure 2.6 *Levels of design*



6. Architectural design pattern references include Frank Buschmann et al., *Pattern-Oriented Software Architecture*, vols. 1–4 (New York: John Wiley & Sons, 1996–2007), and my book *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Boston: Addison-Wesley, 2002).

In addition to the architectural aspects, design patterns can be applied at the level of collaborating classes. This is known as mechanistic design in the Harmony/ESW process. Design patterns here are below the radar of MDA per se. Mechanistic design patterns focus on optimizing the analysis collaborations that constitute the PIM.[7] Design patterns at this level focus on optimizing how the analysis classes interact.

7. The classic text for design patterns at the mechanistic level is Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1994).

Detailed design concentrates on optimizing individual classes, functions, and data elements. At this level of abstraction, the industry refers to the patterns of optimization as **design idioms.** Most classes are fairly simple, but there is a certain percentage in all systems, usually 3% to 5%, that require special attention. The topics of concern during detailed design include:

• Internal optimization of algorithms, for example:

° Average execution time

° Worst-case execution time

° Reusability

° Extensibility

° Memory usage

• Internal optimization of data structures

° Memory usage

° Read access time

° Write access time

° Persistence

• Internal safety and reliability measures

° Data redundancy

° Data validity checks

° Precondition and postcondition validation

° Error- and exception-handling mechanisms

• Optimization of state behavior through state behavioral patterns[8]

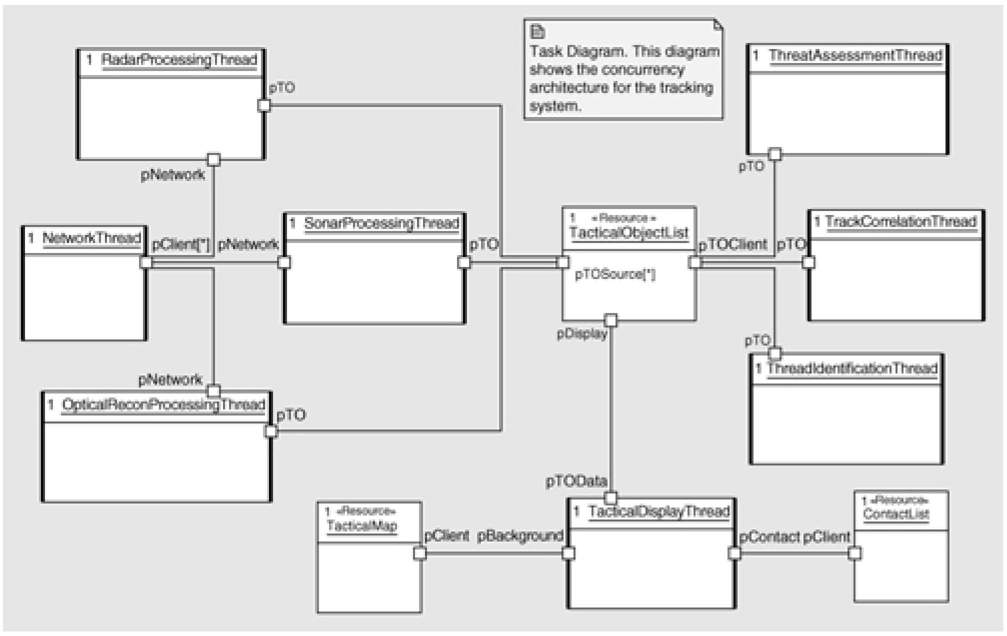
8. See my book *Doing Hard Time*.

• Implementation of associations and other relations

Usually the PSM has many diagrams associated with it that depict different aspects of the design.

Figure 2.7 shows a "task diagram," which is simply a class or object diagram showing the concurrency architecture. The tasks are shown as `«active»` objects in the figure. The UML models a concurrency unit such as a task or thread as an `«active»` object (or class). This element is a structured element containing internal parts (object roles met by instances of classes) that execute in the context of the encapsulating `«active»` object. The `«active»` object has the explicit responsibility for managing the event queue for the concurrency unit (typically there is one event queue per concurrency unit) and creates and destroys the operating system thread in which the internal parts execute. «active» objects are shown with a heavy border on the sides of the element.

Figure 2.7 *PSM concurrency diagram*



The other elements shown in the figure are stereotyped «Resource». These are passive elements that execute in the thread of the caller and are constrained in some way. Usually, there must be some mechanism for protecting the source from the mutual exclusion problems associated with simultaneous access to the resource in a multithreaded environment. This can be done in a number of ways, such as:

• *Critical regions*—disable task switching while the resource is being accessed

• Mutual exclusion semaphores—block access from competing threads while the resource is being accessed

• *Queuing requests*—filter the requests through a first-in-first-out (FIFO) queue or a priority FIFO

In this case, those mechanisms aren't shown, but they could be, if desired. Note that the selection of the tasks, the mapping of the PIM semantic elements into those tasks, the selection of the resources, and the mechanisms for protecting those resources are all optimization properties; they are selected because they provide the desired optimality of the system and so are part of the PSM and not the PIM.

2.3.6. PSI

The PSI isn't described in the original MDA documents from the OMG but has been added because it constitutes an important artifact for deployment of the functionality on a specific platform. The PSI isn't a model in the same sense as the CIM, PIM, and PSM. It refers to the code generated from a PSM that implements the PIM on a specific platform. It is discussed with some military programs such as the Single Integrated Air Picture (SIAP[9]) captured in the Integrated Architecture Behavioral Model (IABM) created by the Joint SIAP System Engineering Organization (JSSEO) and realized in the Open Architecture Track Manager (OATM[10]), a project with which I was involved.

9. See, for example, Basil Krikeles, Robert Merenyi, and John Brtis, "Use of MDA in the SIAP Program," at www.omg.org/news/meetings/workshops/MDA_2004_Manual/4-4_Krikeles_Merenyi_Brtis.pdf.

10. They do love their acronyms, don't they?

Probably the most common use of MDA is the generation of the PSI from the PSM (or even the PIM, skipping the explicit creation of the PSM). In fact, some tool vendors will tell you that MDA is all about specifying actions in an **action language** even though the MDA specifications don't even mention the term. These people clearly have something they're trying to sell. Having said that, let's examine the concept a bit.

Let's Talk about Action Language for a Moment

Actions are primitive statements such as augmenting a variable, creating an instance, or sending a message. Actions show up in three primary places in a UML mode: as actions on a state machine, as actions in an activity diagram, or as statements in a method body. The OMG specifies action semantics[11]—that is, the kinds of things that an action language must be able to state—but does not specify an action language. An action language meets the required action semantics but also provides a syntax. What most people mean when they use the term *action language* is **abstract action language**—a language that is not meant to be directly compiled into executable object code but rather is intended to be translated into a concrete action language for compilation.

11. See "Action Semantics for the UML," OMG document ad/2001-03-01, www.omg.org/docs/ptc/01-11-11.pdf.

As it happens, any 3GL such as C, C++, Java, or Ada can be used to express the action semantics for a UML model. The OMG does not currently define an action language, although there is some work under way in this regard,[12] but nothing is expected to emerge until late 2009 or possibly later.

12. See "Concrete Syntax for a UML Action Language: Request for Proposal," OMG document ad/2007-08-02, at http://www.omg.org/.

The vast majority of modelers use, and prefer to use, the same concrete language in their models as they plan to use for the implementation. However, there are some advantages to using an abstract action language. If you plan to implement the model in multiple concrete languages, or need to allow for the possibility of reimplementing the model in a different concrete language, then an abstract action language provides some real benefit.

Abstract action languages are not without their drawbacks. First, for right now, they are vendor-specific and will be until and unless the OMG releases a standard. That ties your models to a specific tool even though UML models may be exchanged if the tools adhere to the XMI standard (in the upcoming section "XMI"). Second, many developers don't want to learn another language at the same level of abstraction as the implementation language in which they are already experts. An even more serious issue is the difficulty in debugging. If you discover a problem in the PSI, you can't change the PSI and import those changes into the model, because the actions are specified in a different language. You need to fix the abstract action language and then forward-engineer to test the code. There may not be an obvious mapping between the abstract action language and the implementation language, so you will have to think about how to cast the problem in the action language to produce the design implementation language result. Since the two languages are at the same level of abstraction, this may be more trouble than it's worth. Last, the most serious complaint is that unless the action language includes a decompiler (translator from the implementation language to the action language), you can never touch the PSI without breaking the connection between the model and the code. While forward engineering (generation of code from models) is the primary workflow that must be supported, it seems draconian, as well as unnecessary, to completely disallow the reverse workflow.

As I have mentioned, most developers using MDA today use the target implementation language as a concrete action language in the model. This does mean that if the model must be retargeted toward a different implementation language there is some work to do, but using a concrete action language eases the debugging while simultaneously enabling reverse engineering when necessary.

2.3.7. Model Transformation

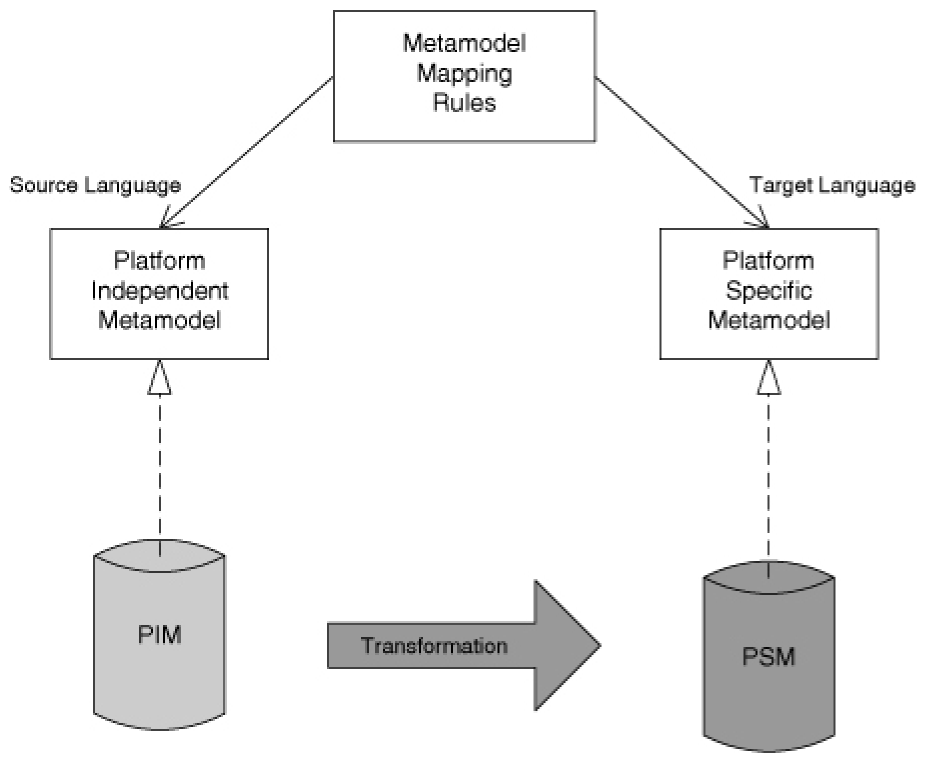
With all these models around, how does one ensure consistency? MDA approaches this problem through **model transformation.** Models may be transformed manually, to be sure, but emphasis is placed on automating these transformations as much as possible. Primarily models are forward-transformed (CIM to PIM to PSM to PSI), but sometimes backward transformations are performed as well.

There are many ways these model transformations can be done. The most useful are by metamodel mapping, by marking and transforming the model, and by elaborating the model with design patterns.

Metamodel Transformations

Metamodel transformations are done by creating PIM- and PSM-specific metamodels and transforming a metaclass from one into the metaclass for the other. The basic idea is shown in Figure 2.8. The PIM is captured in a domain-specific modeling language (most likely a UML profile) that contains domain concepts for the platform-independent application semantics. The PSM is defined using a different metamodel (also likely a UML profile). The mapping rules identify which metamodel elements in the PSM are created from which metamodel elements in the PIM.

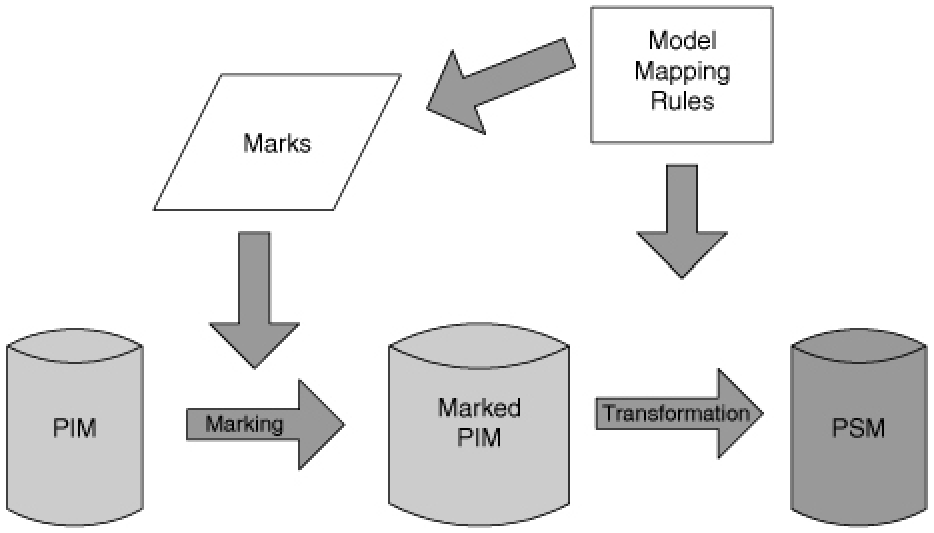
Figure 2.8 *Metamodel transformations*



The disadvantage of this is the work involved in creating two different metamodels and the mapping rules that define the transformation between them, as well as constructing the translator. The advantage of the approach is that once this work is done, many metamodels can be transformed easily. This approach is not used as often as the other approaches discussed here.

Another approach is to mark the one model with "design hints" and use a translator that uses the design hints to create the PSM by applying transformational rules to the marked elements (see Figure 2.9). This is very similar to the previous metamodel transformational approach discussed but is less work to implement. In this approach, marks are added to the source model (usually the PIM). These marks are almost always either stereotypes (such as the `«trace»` and `«active»` stereotypes mentioned previously), tags (user-defined name-value pairs), or constraints (user-defined "well-formedness" rules). These three elements form the lightweight extension mechanisms in the UML used to define profiles, a topic that will be discussed soon.

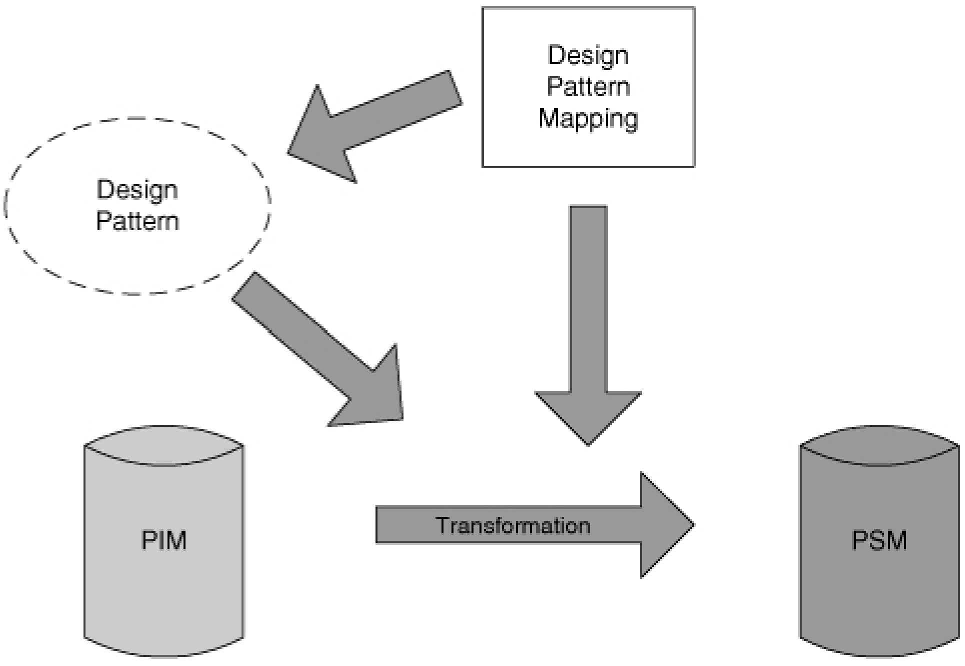
Figure 2.9 *MDA model transformations*



For example, some classes in the PIM might be marked as **`«distributed»`** and the translator might generate CORBA or DDS (Data Distribution Service) interface description language (IDL) for the marked elements. Or an association on a class might be marked **`«IPC»`** to indicate interprocess communications.

The third, and most common, approach for generating the PIM is through the manual or semiautomated application of design patterns (see Figure 2.10). A design pattern is a generalized solution to a commonly recurring problem; that is, to be a design pattern, it must be generalizable and must still make sense when the specifics of its application are removed. It must also address a concern that reappears in a variety of contexts; that is, it must be **reusable.**

Figure 2.10 *Design pattern model transformations*



Another useful definition for a design pattern is that it is a **parameterized collaboration.** It is a set of collaborating object roles, some of which are the formal parameters of the pattern. These parameters are object roles typed by classes that will be replaced by classes from the PIM. The process of substituting the actual parameters (classes from the PIM) for the formal parameters (classes in the pattern) is called **pattern instantiation.**

A design pattern has a number of important aspects. The **problem** is a statement of the goals of the design pattern, specifically: What design problem does the pattern address? The **applicability** defines the environmental or operational circumstances that enable the pattern to be effectively used. The **solution** is the pattern itself. The solution is usually shown as a collaboration of structural elements (types and classes) with relations tying them together plus a set of interaction views showing how the elements interact dynamically. In addition, state machines and activity diagrams may show complex state or algorithmic behavior if helpful. The **consequences** are arguably the most important aspect of the pattern because ultimately these decide whether the pattern is an appropriate choice. The consequences are a set of benefits and costs of using the pattern. Because design (and design patterns) is all about optimality, whenever you optimize one aspect, you deoptimize some other. The consequences enable the developer to make good pattern selections.

The application of design patterns can be automated by tools. For example, the Rhapsody tool (from IBM Rational) applies a number of design patterns automatically (although this can be configured). Some of the design patterns Rhapsody automatically implements include the following:

• Container-Iterator Pattern

• Event Queue Pattern

• Guarded Call Pattern

• State Pattern

• CORBA Broker Pattern (supports CORBA, COM[13], DCOM[14])

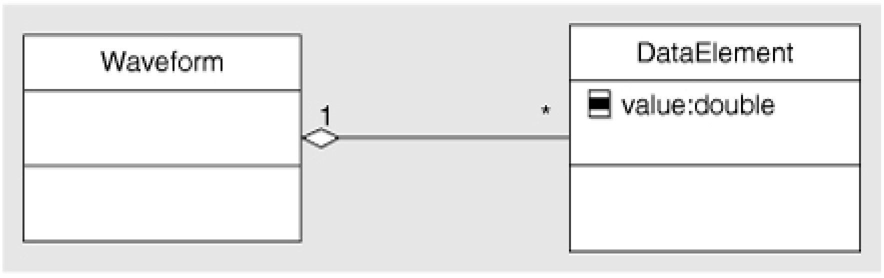
13. Component Object Model.

14. Distributed Component Object Model.

• Data Bus Pattern (supports DDS)

For example, for the classes in Figure 2.11, code can be generated that uses built-in or Standard Template Library (STL) containers and iterators to manage the collection. This is done automatically by Rhapsody and serves as a useful, but simple, example of design pattern automation. The header file for the `Waveform` class is shown in Listing 2.1, and the implementation file is shown in Listing 2.2. You can see that `OMCollection<DataElement*>` and `OMIterator<DataElement*>` are added to manage the collection of `DataElements`.

Figure 2.11 *Container-Iterator Pattern example*



Listing 2.1 `Waveform.h`

```
        #ifndef Waveform_H
        #define Waveform_H

        //## auto_generated
        #include <oxf\oxf.h>
        //## auto_generated
        #include <oxf\omcollec.h>
        //## link itsDataElement
        class DataElement;

        //## package pattern

        //## class Waveform
        class Waveform {
            ////    Constructors and destructors    ////

        public :

            //## auto_generated
            Waveform();

            //## auto_generated
            ~Waveform();

            ////    Additional operations    ////

            //## auto_generated
            OMIterator<DataElement*> getItsDataElement() const;

            //## auto_generated
            void addItsDataElement(DataElement* p_DataElement);

            //## auto_generated


    void removeItsDataElement(DataElement* p_DataElement);

    //## auto_generated
    void clearItsDataElement();

protected :

    //## auto_generated
    void cleanUpRelations();

    ////    Relations and components    ////

    OMCollection<DataElement*> itsDataElement;  //## link itsDataElement

    ////    Framework operations    ////

public :

    //## auto_generated
    void _addItsDataElement(DataElement* p_DataElement);

    //## auto_generated
    void _removeItsDataElement(DataElement* p_DataElement);

    //## auto_generated
    void _clearItsDataElement();

};
#endif
```

Listing 2.2 `Waveform.cpp`

```
//## auto_generated
#include "Waveform.h"
//## link itsDataElement
#include "DataElement.h"
//## package pattern

//## class Waveform
Waveform::Waveform() {
}

Waveform::~Waveform() {
    cleanUpRelations();
}

OMIterator<DataElement*> Waveform::getItsDataElement() const {
    OMIterator<DataElement*> iter(itsDataElement);
    return iter;
}
```

```
void Waveform::addItsDataElement(DataElement* p_DataElement) {
    if(p_DataElement != NULL)
        {
            p_DataElement->_setItsWaveform(this);
        }
    _addItsDataElement(p_DataElement);
}

void Waveform::removeItsDataElement(DataElement* p_DataElement) {
    if(p_DataElement != NULL)
        {
            p_DataElement->__setItsWaveform(NULL);
        }
    _removeItsDataElement(p_DataElement);
}

void Waveform::clearItsDataElement() {
    OMIterator<DataElement*> iter(itsDataElement);
    while (*iter){
        (*iter)->_clearItsWaveform();
        iter++;
    }
    _clearItsDataElement();
}

void Waveform::cleanUpRelations() {
    {
        OMIterator<DataElement*> iter(itsDataElement);
        while (*iter){
            Waveform* p_Waveform = (*iter)->getItsWaveform();
            if(p_Waveform != NULL)
                {
                    (*iter)->__setItsWaveform(NULL);
                }
            iter++;
        }
        itsDataElement.removeAll();
    }
}

void Waveform::_addItsDataElement(DataElement* p_DataElement) {
    itsDataElement.add(p_DataElement);
}

void Waveform::_removeItsDataElement(DataElement* p_DataElement) {
    itsDataElement.remove(p_DataElement);
}

void Waveform::_clearItsDataElement() {
    itsDataElement.removeAll();
}
```

It should be noted that sometimes some models may be only implicitly created. For example, Rhapsody can generate PSI directly from a PIM through the application of design patterns. During this transformation (called “code generation”), Rhapsody internally generates the PSM (which it calls the “simplified model”) and then generates code from the PSM. Normally, this PSM is not exposed to the modeler, but it can be, if the developer wants to see, store, or manipulate it. For example, in the previous code samples, the PSM is not explicitly exposed.

It is also common to manually elaborate the PIM into the PSM by adding design patterns by hand. This can be done because some patterns are difficult to automate, for example. In general, I recommend a combination of both automated and manual transformations to create the PSM.

Common Model Transformations

A number of specific model transformations are commonly done as development work progresses. The MDA specification and usage focus on the PIM-to-PSM translation, but there are several more as well.

CIM to PIM

This transformation is so common that it is a basic part of the development process. The CIM captures and organizes the requirements into use cases and related forms (e.g., sequence diagrams, state machines, activity diagrams, and constraints). The analysis model, or PIM, identifies the essential structural elements and relations necessary to perform the semantic computations of the application. This is commonly called "realizing the use case."

This transformation is problematic to automatically transform, so virtually everyone does it through an object discovery procedure. Development processes differ in how this procedure is performed. In the Harmony/ESW process, a technique called object identification strategies is used to ferret out the essential elements of the PIM. Continual execution is also used to ensure that we've done a good job. This translation will be discussed in more detail in Chapter 6, "Agile Analysis."

PIM to PIM

PIM-to-PIM mappings are usually ones of refinement in which more details are added to a more general PIM. This most commonly occurs when it is desirable to create a family of PSMs that have common design properties; in that case, the common design properties can be put in a refined PIM even though they properly belong in a PSM. For example, a PIM might include a common subsystem or concurrency architecture structure, even though the details of how the subsystems are deployed and how the concurrency units are implemented may be omitted from the refined PIM.

PIM to PSM

This is probably the most common focus in MDA. This is traditionally the mapping from the essential analysis model to the platform-specific design model. In the Harmony/ESW process, this is done at the three levels of design but MDA focuses mostly on the architectural views. To this end, Harmony/ESW defines five key views of architecture that organize and orchestrate the elements of the PIM within the PSM. This will be discussed later in the section "Harmony's

Five Key Architectural Views.”

PSM to PIM

This “backward” mapping is largely a matter of mining an existing design for essential elements or stripping out platform, technology, and design patterns to uncover the essential model hiding within the PSM. This transformation is useful when moving from an existing system design to a family of products. It is often used in conjunction with the PSI-to-PSM (reverse-engineering) transformation to identify the essential elements hidden within an existing code base. This can be an essential part of refactoring a model when the PIM doesn’t exist.

PSM to PSI

Other names for this transformation are “code generation” and “model compiling.” A number of modeling tools can generate code from the model. Tools can be classified into three primary types in this regard. Nongenerative tools don’t generate any code from the model. Such tools are commonly known as “drawing tools.” The second category of tools generates code frames; these tools generate classes and class features (e.g., attributes and empty operations for the developer to elaborate) but don’t deal with state machines or activity diagrams. The final category contains so-called behavioral tools and generates code from the behavioral specifications (i.e., state machines and activity diagrams) as well structural elements. This last category of tools tends to be the most capable but also the most expensive.

Within the generative tools, a number of different techniques are employed for generating the code. Two primary kinds are rule-based tools and property-based tools. Rule-based tools use a set of rules, captured in some human-readable (and proprietary) form, to model the code generation. A translator (compiler) applies these rules to construct the output code. For example, one rule might be that when an association end is encountered in a class, it is implemented as a pointer whose name is the role end label. Property-based tools are model compilers that provide a set of user-defined properties to tune the code generation. These tools are usually somewhat less flexible than rule-based tools but are usually far easier to use.

PIM to PSI

It is possible that the model translator skips the step of producing the PSM and directly outputs the PSM. It is common that internally this translation takes place in multiple phases, and one of these phases constructs an interim PSM but the intermediate PSM is thrown away. This is a viable strategy for production of the final design, but it is also extremely useful for testing and debugging the PIM before design is even begun. This is a common way to work with Rhapsody, for example. The code generators in Rhapsody have built-in design rules out of the box and so partially complete PIMs may be executed by generating the code, compiling it, and running the generated executable. In this case, the default design decisions are not really exposed to the user and they are, in some sense, the simplest design choices. These decisions work fine for the purpose of testing and debugging the PSM and are usually changed and elaborated during the design phase.

PSI to PSM

The PSI-to-PSM model transformation can be done in two primary ways: reverse engineering and **round-trip engineering.** Reverse engineering occurs when you construct a PSM model from a code base for which no model exists. Round-trip engineering occurs when minor modifications are made to the code generated from a model. Reverse engineering is normally done incrementally, a piece at a time, but still only once for a given code base. Round-trip engineering is done more frequently as developers modify the generated code.

Reverse engineering is extraordinarily helpful, but it is not without problems. The first problem is that many times the developers are not satisfied with the generated model because, for perhaps the very first time, they can directly see the design. Often, that design isn't very good; hence the consternation. In addition, some aspects of the model may not be easy to generate. For example, while it is rather easy to generate the structural model from the code, few reverse-engineering tools identify and construct state machines from the underlying code. It is a difficult problem, in general, because there are so many different ways that state machines may be implemented.

Another problem can arise if the code generated forward from the constructed model doesn't match the original code because of differences in the translation rules. To be correct, the original and subsequently generated code must be functionally equivalent, but there is often a desire to have them look the same as well. Some reverse-engineering tools fare better than others on this score.

In general, I recommend an incremental approach to reverse-engineering a code base:

1. First, in code, identify and separate the large-scale architectural pieces of the system.

2. Reverse-engineer one architectural element:

a. Integrate that element back with the remaining code, including the forward-engineered code from the reverse-engineered model.

b. Validate the resulting code.

3. Reverse-engineer the next architectural element.

The basic practice of agile methods is to develop in small incremental steps and validate that the resulting code works before going on. This practice is very successful with reverse-engineering code bases.

Round-trip engineering is a much simpler problem because the model already exists. Most code generators mark places where developers are allowed or not allowed to make changes in the code, and the code generators insert markers to facilitate the round-trip engineering. The advantage of this approach is that the developer can work in the code when that is appropriate without breaking the connection between the model and the code. Some tools allow only forward code generation and do not support round-trip engineering. Although I agree that primarily code should be forward-generated, I believe that completely disallowing modifications to the source code is a draconian measure best avoided.

2.4. MDA Technologies

The MDA standard relies on a core set of technologies, including MOF, UML, SysML, CWM, and profiles.

2.4.1. MOF

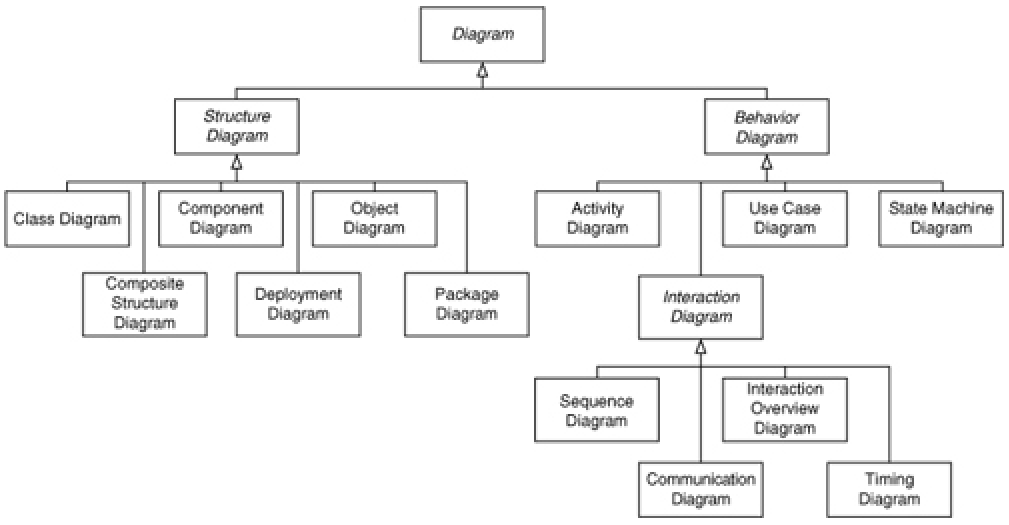
The MOF forms the core basis for all modeling within the OMG family of standards, including the UML. MOF is a kind of universal modeling language in which other modeling languages may be defined; in fact, MOF can be thought of as the language in which the UML is defined.

MOF uses the same basic syntax as UML models, so you can use UML tools to create MOF models. MOF is used to create other modeling languages, such as the Common Warehouse Metamodel (CWM) and the CORBA Component Model (CCM). Most developers don’t need to directly work in MOF, but it’s there in the conceptual underpinnings of UML if needed.

2.4.2. UML

The UML is enormously successful; in fact, it is the de facto standard software modeling language. I have contributed to the UML standard and so I have a keen interest in its success. It is a mostly graphical modeling language consisting of many different kinds of diagrams, as shown in Figure 2.12.

Figure 2.12 *UML diagram types*



Strictly speaking, these are not all different diagram types. Some constitute diagram uses; for example, a structure diagram is a class diagram whose purpose is to show the internal structure of a composite class, and a package diagram is a class diagram, the purpose of which is to show some aspect of the model organization.

The three key diagrams are the class, state, and sequence diagrams. The other diagrams have their place and add value, but any system can be constructed from these three basic diagram types.

We won't say much about the UML per se in this book, although we will use it extensively. Interested readers are referred my books *Real-Time UML* for exposition and explanation of how to use the UML and *Real-Time UML Workshop for Embedded Systems* (Burlington, MA: Elsevier Press, 2006) for detailed exercises with solutions from the real-time and embedded domain.

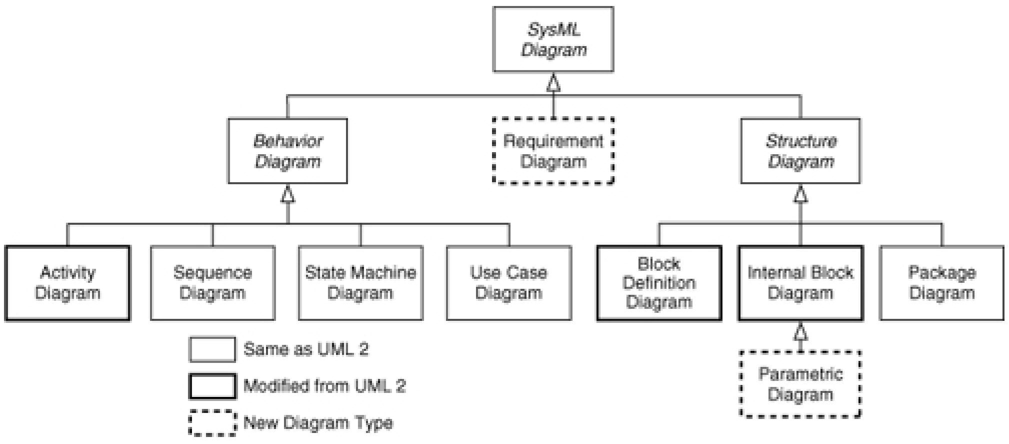
2.4.3. SysML

Strictly speaking, the Systems Modeling Language (SysML) is a UML Profile; that is, it is a minor extension/elaboration/subset of the UML. I also contributed to the SysML standard. The SysML is an attempt to bring the power of UML modeling to the systems engineering domain. Historically, systems engineers have rejected the UML because it is too “software-oriented.” The SysML standard changed the names of key elements and is now experiencing great success with systems engineering groups.[15] SysML simultaneously subsets the UML (for example, SysML does not use UML deployment diagrams but uses block—i.e., class—diagrams) for this purpose and extends the UML by adding new semantic extensions (e.g., modeling continuous behavior), new diagram types (e.g., parametric diagrams for showing parametric values and their relations), and some model libraries (e.g., SI definitions model library for standard international types).

15. OK, we did some other things as well, such as elaborating activity diagrams to be continuous in time and value (a genuinely valuable extension), but the primary thing that made the UML acceptable to systems engineers was changing the term *Class* to *Block*. Go figure ☺.

Figure 2.13 summarizes the primary inclusions and extensions SysML makes to the UML.

Figure 2.13 *SysML diagram types*



2.4.4. XMI

One of the good things about the MOF-based standards is that they provide model interchange standards. This is valuable for a couple of reasons. First, it means that, in principle, you are not bound to a particular vendor with all your corporate intellectual property. This is huge, because companies have millions of dollars invested in their UML models. If a tool vendor goes out of business, or if another tool vendor arises that better meets the developer needs, the

XML Model Interchange (XMI) standard provides an accepted common means for bringing your models, and the intellectual property they represent, to the new tool. It also means that supporting analysis tools, such as RAPID RMA from Tri-Pacific Software[16] (a performance and schedulability analysis tool), have a standard way of reading user models so that they can perform specialized functions and analysis. The current version of XMI, 2.1, has finally met the users' request to provide diagram interchange in addition to the interchange of semantic elements. Unfortunately, due to varying degrees of adherence to the XMI standard, XMI interchange is not as seamless as it should be. However, most users don't use XMI frequently, so the pain is manageable.

16. www.tripac.com/html/prod-fact-rrm.html.

2.4.5. CWM

The Common Warehouse Metamodel (CWM) is primarily concerned with metadata management and data transformations. It actually contains a set of metamodels for relational databases, record structures, XML, and data transformations. Its purpose is to enable interchange of warehouse and business intelligence metadata between various tools and repositories. CWM is based on UML, MOF, and XMI.

2.4.6. Profiles

A **UML Profile** is a version of the UML specialized for some special purpose or domain. A profile is a coherent set of lightweight extensions to existing UML elements, inclusions and omissions from the UML standard, and additional elements. A profile must be consistent with the base definition of the UML. It is not allowed to extend the metamodel directly. It is considered an extension at the M1 level, although we find it more useful to consider at level M1.5. The extension mechanisms include:

• Stereotypes

• Tags

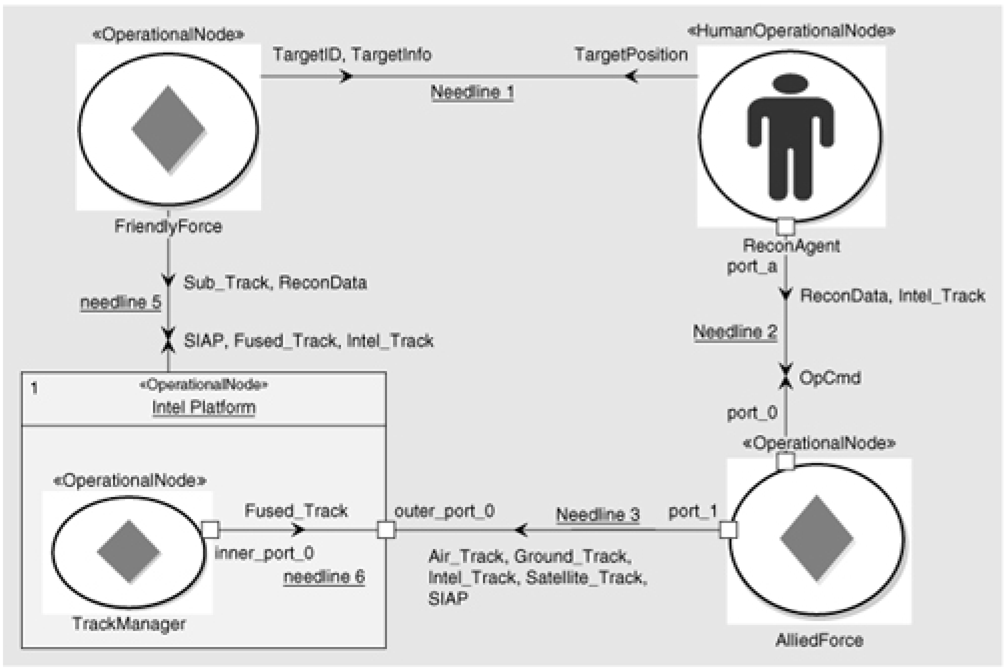
• Constraints

• Model libraries

A **stereotype** is a special "kind of" metaclass. Typically, it identifies an element that is used in a specific way, has additional associated metadata, or has additional rules for usage.

Stereotypes are normally identified by attaching the stereotype name in guillemets, but you are also allowed to add new notations. New notations can include both specialized graphics symbols for the stereotyped element as well as new diagrammatic types based on existing diagrams. For example, Figure 2.14 shows a DoDAF[17] Operational View-2 (OV-2) diagram, which is a stereotype of a class diagram. On it we see **operational nodes** and **human operational nodes** (stereotypes of instances) connected via **needlines** (stereotype of links). On the needlines, we see information exchanges (stereotypes of UML information flows). Underneath, the elements have valid UML specifications, but the stereotype allows you to cast the problem in the vocabulary of its domain.

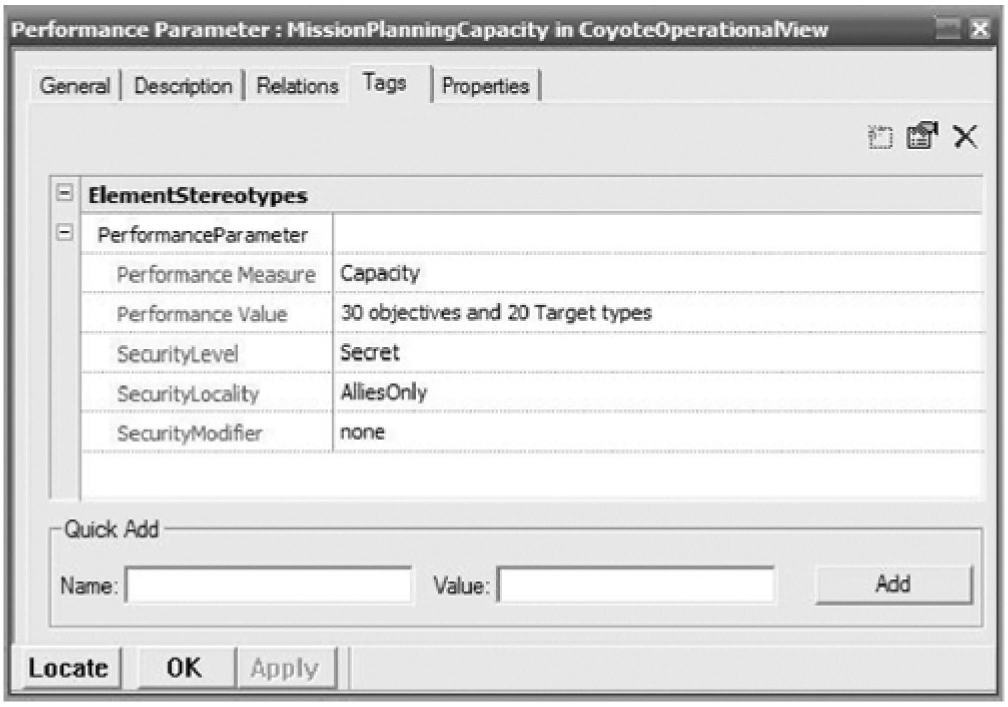
Figure 2.14 *DoDAF OV-2 diagram*



17. Department of Defense Architecture Framework.

Stereotypes often contain additional metadata, stored in tags. A tag is a name-value pair. Associating a set of tags with a stereotype means that any stereotyped element also contains those tags. For example, the same DoDAF profile has a stereotype constraint called a `PerformanceParameter` used for modeling different kinds of qualities of service. Performance parameters have tags as shown in Figure 2.15.

Figure 2.15 *Performance parameter tags*


Performance Parameter : MissionPlanningCapacity in CoyoteOperationalView

ElementStereotypes	
PerformanceParameter	
Performance Measure	Capacity
Performance Value	30 objectives and 20 Target types
SecurityLevel	Secret
SecurityLocality	AlliesOnly
SecurityModifier	none

Other profiles have other tags. The UML Profile for Schedulability, Performance, and Time (SPT) has a large set of stereotypes with associated tags for capturing performance and schedulability metadata. Because the underlying model is UML, these can be exported in XMI to be used by performance analysis tools.

Constraints are a common way to assign values to tags. A **constraint** is a user-defined "well-formedness" rule—a rule that defines a criterion for a well-formed model element. Tags are often associated values in constraints. These constraints are usually declarative and can be validated only by testing the system, often on its ultimate target platform (PSM).

A **model library** is a set of predefined constructs, such as the SI definitions within the SysML profile that provide standard data types for typical measures of length, mass, time, current, temperature, and so on; these include units such as meters, kilograms, seconds, amperes, kelvins, and others.

A profile is a coherent set of these extensions—stereotypes, tags, constraints, and model libraries—all tied to the underlying UML metamodel.

Common profiles in the real-time and embedded domain include:

• *SysML*—a profile of the UML intended to provide the modeling power of UML to systems engineers

• *The UML Profile for SPT*—a profile intended to provide standard ways of representing performance metadata for analysis

• Modeling and Analysis of Real-Time and Embedded Systems (MARTE)—a profile intended to replace and extend the SPT profile to cover other concerns of embedded systems beyond just performance (not yet released; currently in finalization)

• *UML Profile for DoDAF and MoDAF (UPDM)*—a profile intended to provide a standard means of representing DoDAF (U.S.) and Ministry of Defense Architecture Framework (UK) models in the UML (not yet released; currently in finalization)

• *UML Testing Profile*—a profile intended to provide ways of representing test vectors, test suites, and test fixtures to facilitate the testing of UML models and to bring the expressive power of UML to the testing domain.

2.5. Benefits of MDA

MDA brings a number of significant benefits to the developer, including the agile developer. Modeling, well performed, enhances visibility and understandability of the analysis and design. In addition to the benefits of standard UML modeling, MDA provides benefits of its own, especially portability, reusability, and isolation from technological churn.

2.5.1. Portability

In this context, portability means the capability of moving an application from one execution environment to another. MDA clearly improves portability because the PIM provides all the functionality in an inherently reusable form. This is because the platform and technological details are missing from the PIM. Once a high-quality PIM is created, it can be put on different platforms through model transformations to create PSMs for the target environments.

2.5.2. Reusability

One of the problems with source code as a repository for intellectual property is that it contains essential functionality munged together with implementation details. For most companies building real-time and embedded systems, their key intellectual property is not platform-related; it is in the state or algorithmic specification of the essential behavior. A company that makes flight control systems has a core interest not in network architectures, but in how flight systems work. A company that makes routers has a core interest in network architectures but not in operating systems. Each of these companies wants to be able to create its core intellectual property in such a way that it can be used on other platforms or when the nature of the existing platforms changes.

MDA provides a clear means to achieve this goal: creating *inherently reusable* intellectual property within the PIM and reusing it by specifying the PSM that includes platform and technological details.

2.5.3. Isolation from Technology Churn

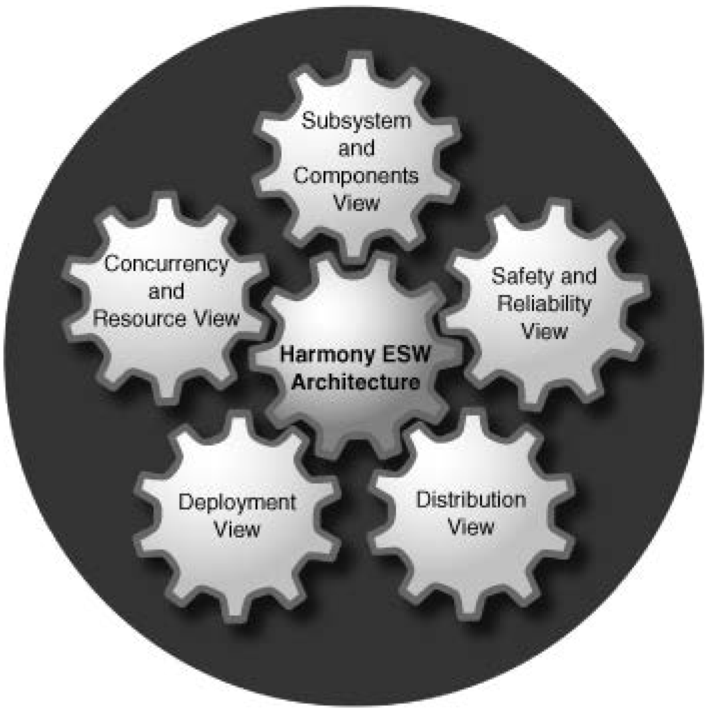
Another point of view on reusability comes from the fact that technology changes, and it changes quickly. New network infrastructures, new CPUs, and new operating systems appear all the time. The core semantics of the applications don't, for the most part, change nearly as quickly. If you work in a domain in which systems must be maintained for a long period of time—such as military and aerospace—then you need isolation from the constant churn of technology. It is too expensive to completely reengineer a system because a CPU or network becomes obsolete. Having the essential semantic intellectual property captured in the PIM makes it far easier to integrate new technological and platform solutions.

2.6. Harmony's Five Key Architectural Views

The UML is a generic modeling language and expressly does not contain content that is process-dependent. Thus, it has weak notions about what should be considered architecture and how to go about creating it. On the other hand, the Harmony/ESW process is in the business of providing that guidance and has very strong opinions about what constitutes an architecture and how to go about creating it.

As shown in Figure 2.16, Harmony/ESW centers its architectural concern around five key views:

Figure 2.16 *Harmony/ESW five key views of architecture*


Subsystem and Components View
Concurrency and Resource View
Safety and Reliability View
Harmony ESW Architecture
Deployment View
Distribution View

• Subsystem and component architecture

• Concurrency and resource management architecture

• Distribution architecture

• Safety and reliability architecture

• Deployment architecture

These architectural aspects are expressly identified because in my experience they play a key role in the efficiency and quality of the delivered system. Harmony organizes its architecture into these aspects because the computer science literature is organized in the same way. It should also be noted that these aspects are a part of architecture, architecture is a part of design, and design optimization is part of the PSM. Thus, these concerns usually do not appear in the PIM but instead only in the PSM.

It is normal to create one or more diagrams expressing each of these architectural viewpoints; thus, your models have a “subsystem diagram,” a “task diagram,” a “distribution diagram,” and so on. These are usually nothing more than class diagrams with a mission to show specific aspects of the architecture; that is, they will show the model elements relevant to the architectural viewpoint in question but not more than that.

In an incremental development process such as Harmony/ESW, not all of these viewpoints may be present in all of the prototypes.[18] Early prototypes may have only the subsystem architecture in place and only later will the other viewpoints be added. When the various architectural aspects are added is dependent upon the schedule and iteration plans for the project—a topic I will discuss in more detail in Chapter 5, “Project Initiation.”

18. Remember that in this context a **prototype** is a “working, validated version of the system” that may be incomplete. Nevertheless, the code that it does contain is the code that will be eventually delivered.

2.6.1. Subsystem and Component Architecture

A subsystem is the largest-scale architectural unit within a system. Both components and subsystems are kinds of classes in the UML that serve a similar purpose: provide large-scale organizational units for elements within the running system. The subsystem and component architecture focuses on:

• The identification of the subsystems and components

• The allocation of responsibilities to the subsystems and components

• The specification of interfaces, both offered and required, between the subsystems

The subsystem architecture is usually shown on one or more “subsystem” (class) diagrams that contain the subsystems, ports, and interfaces. There are different styles in use for constructing the class diagram. One very common approach is to create a structure diagram representing the system containing the subsystems as parts with connectors linking them together (see Figure 2.17). Another is to show the subsystems as classes on a class diagram with their ports and interfaces (see Figure 2.18). A third way is to depict each subsystem on its own diagram surrounded by the interfaces it offers or requires (see Figure 2.19). A fourth way is to just focus on the interfaces themselves (see Figure 2.20). Each of these diagram styles focuses on an important aspect of the subsystem architecture. It is not uncommon to show multiple aspects of the subsystem architecture by providing different diagrams. I, for example, typically use all of these for systems of even moderate complexity.

Figure 2.17 *Subsystems as system parts*

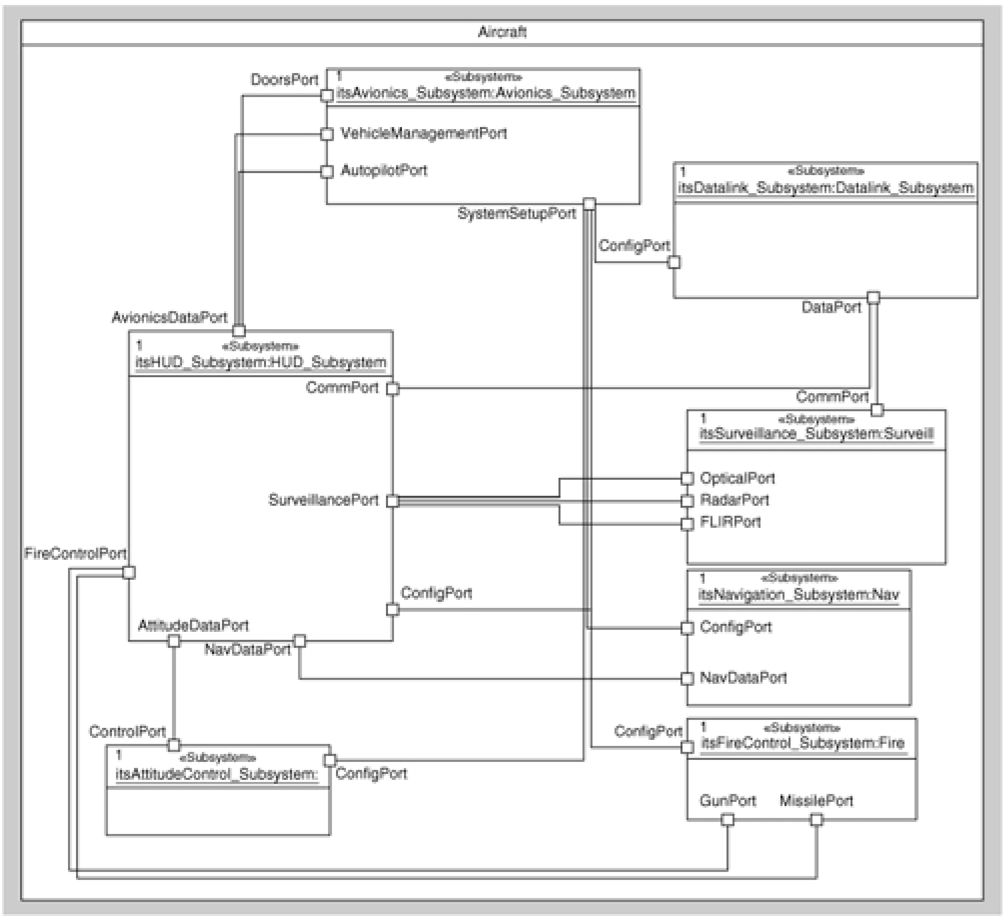

Aircraft
1 «Subsystem» itsAvionics_Subsystem:Avionics_Subsystem
DoorsPort
VehicleManagementPort
AutopilotPort
SystemSetupPort
1 «Subsystem» itsDatalink_Subsystem:Datalink_Subsystem
ConfigPort
DataPort
AvionicsDataPort
1 «Subsystem» itsHUD_Subsystem:HUD_Subsystem
CommPort
SurveillancePort
FireControlPort
ConfigPort
AttitudeDataPort
NavDataPort
CommPort
1 «Subsystem» itsSurveillance_Subsystem:Surveill
OpticalPort
RadarPort
FLIRPort
1 «Subsystem» itsNavigation_Subsystem:Nav
ConfigPort
NavDataPort
ControlPort
1 «Subsystem» itsAttitudeControl_Subsystem:
ConfigPort
ConfigPort
1 «Subsystem» itsFireControl_Subsystem:Fire
GunPort
MissilePort

Figure 2.18 *Subsystem classes with interfaces*

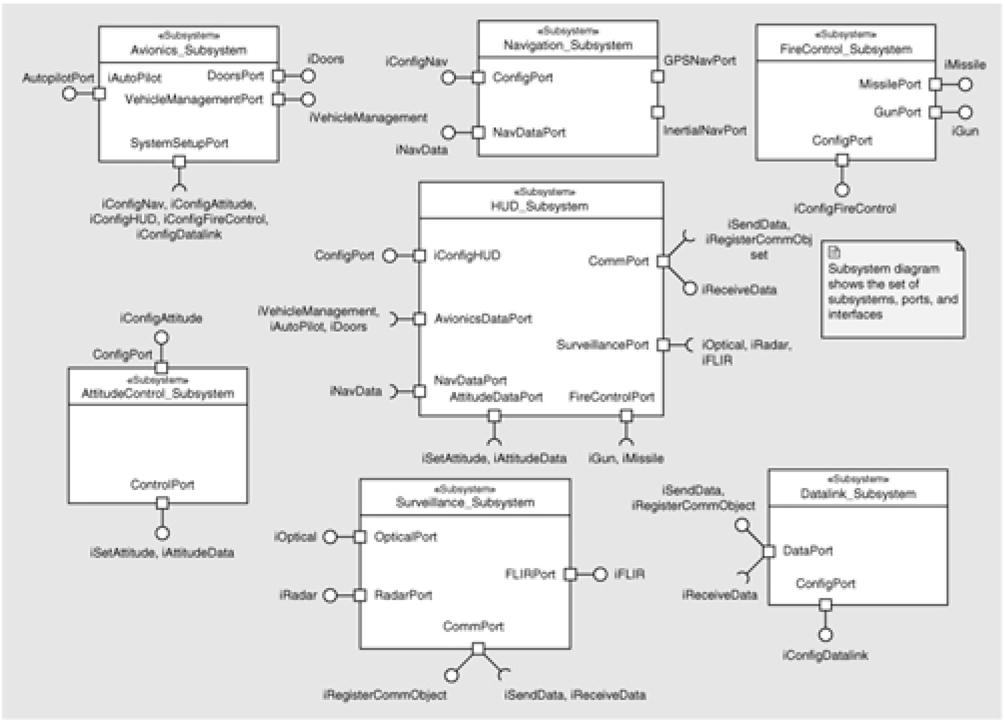


Figure 2.19 *Subsystem with interface details*

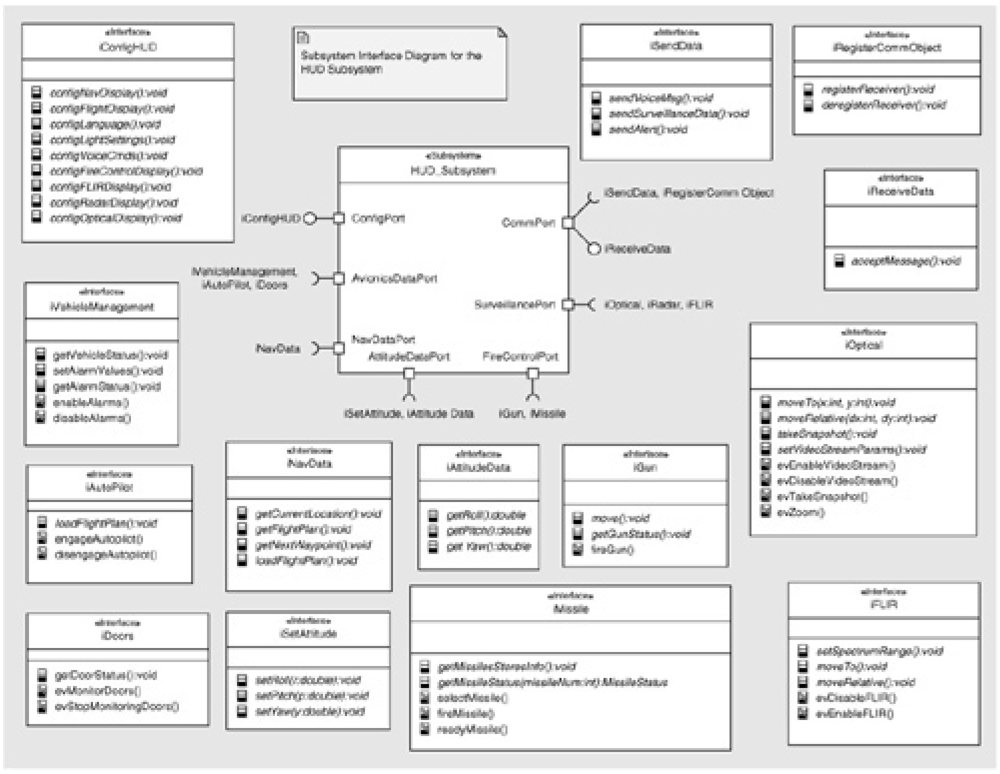
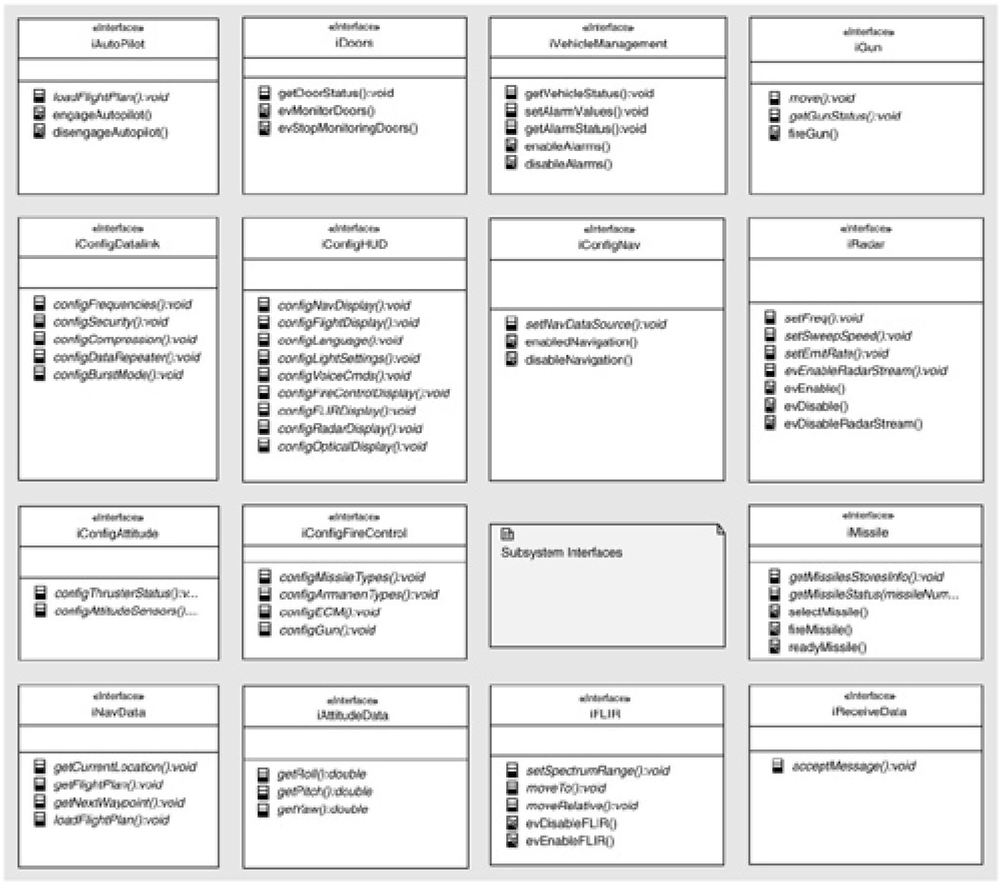


Figure 2.20 *Subsystem interfaces overview*



2.6.2. Concurrency and Resource Management Architecture

The concurrency and resource management architecture has a tremendous impact on the performance of the system. This architectural view focuses on:

• The identification of the concurrency units

• The mapping of the concurrency units to OS tasks and threads

• The specification of the concurrency units’ metadata, such as:

○ Priority

○ Period

○ Execution time

○ Blocking time

• Specification of scheduling mechanisms

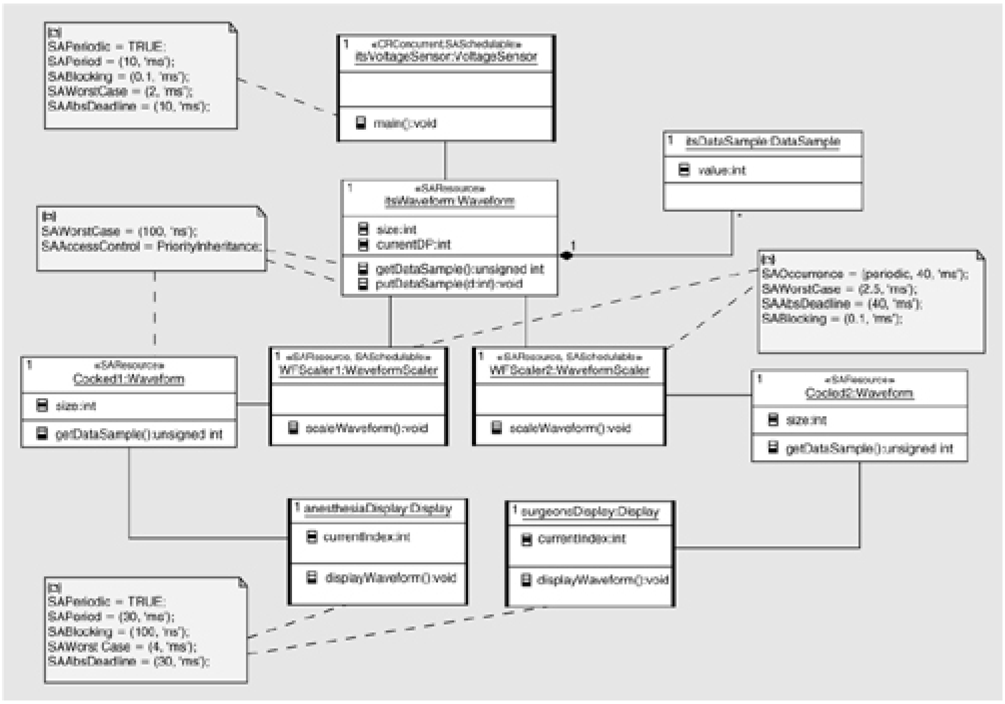
• Identification of resources

• Specification of resource-sharing patterns

The concurrency architecture is usually shown on one or more “task” (class) diagrams containing the active classes, performance metadata in constraints, and possibly other concurrency mechanisms, such as queues and semaphores.

Figure 2.21 shows a typical task diagram with the «`active`» classes (classes with heavy side borders), resources (stereotyped from «`SAResource`», the SPT profile), and concurrency-related metadata in constraints. In this case, there are three «`active`» classes with a total of five «`active`» instances, since there are two instances of the `WaveformScaler` and `Display` tasks.

Figure 2.21 *Task diagram*



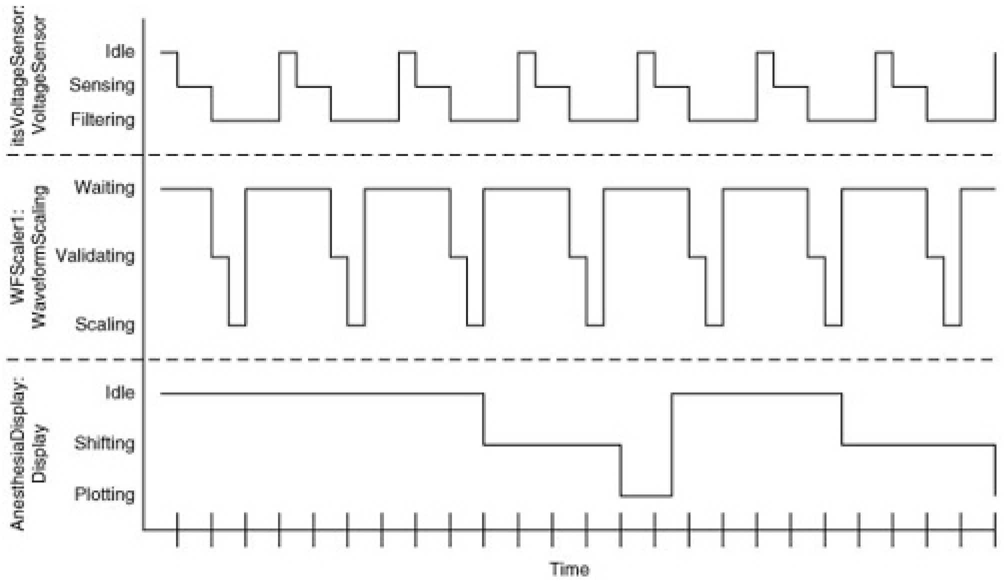
{c}
SAPeriodic = TRUE;
SAPeriod = (10, 'ms');
SABlocking = (0.1, 'ms');
SAWorstCase = (2, 'ms');
SAAbsDeadline = (10, 'ms');
1 «CRConcurrent,SASchedulable»
itsVoltageSensor:VoltageSensor
main():void
1 itsDataSample:DataSample
value:int
*
1 «SAResource»
itsWaveform:Waveform
size:int
currentDP:int
getDataSample():unsigned int
putDataSample(d:int):void
1
{c}
SAWorstCase = (100, 'ns');
SAAccessControl = PriorityInheritance;
{c}
SAOccurrence = (periodic, 40, 'ms');
SAWorstCase = (2.5, 'ms');
SAAbsDeadline = (40, 'ms');
SABlocking = (0.1, 'ms');
1 «SAResource»
Cooked1:Waveform
size:int
getDataSample():unsigned int
1 «SAResource, SASchedulable»
WFScaler1:WaveformScaler
scaleWaveform():void
1 «SAResource, SASchedulable»
WFScaler2:WaveformScaler
scaleWaveform():void
1 «SAResource»
Cooked2:Waveform
size:int
getDataSample():unsigned int
1 anesthesiaDisplay:Display
currentIndex:int
displayWaveform():void
1 surgeonsDisplay:Display
currentIndex:int
displayWaveform():void
{c}
SAPeriodic = TRUE;
SAPeriod = (30, 'ms');
SABlocking = (100, 'ns');
SAWorst Case = (4, 'ms');
SAAbsDeadline = (30, 'ms');

In addition to the task diagram, UML timing diagrams are useful for showing changes in state or value over linear time. Timing diagrams are particularly useful for viewing the correlation of state changes to concurrently executing objects. Sadly, few tools support timing diagrams,[19] but they can be constructed using tools such as Microsoft Excel.

19. I introduced timing diagrams for such systems in my book *Real-Time UML, First Edition* in 1997 and wrote the initial draft for the UML 2 specification for timing diagrams, so I have a vested interest ☺.

Figure 2.22 shows a typical timing diagram with concurrently executing instances. The ordinate values are the states of the instances and the abscissa axis represents time.

Figure 2.22 *Timing diagram*



itsVoltageSensor: VoltageSensor
Idle
Sensing
Filtering
WFScaler1: WaveformScaling
Waiting
Validating
Scaling
AnesthesiaDisplay: Display
Idle
Shifting
Plotting
Time

2.6.3. Distribution Architecture

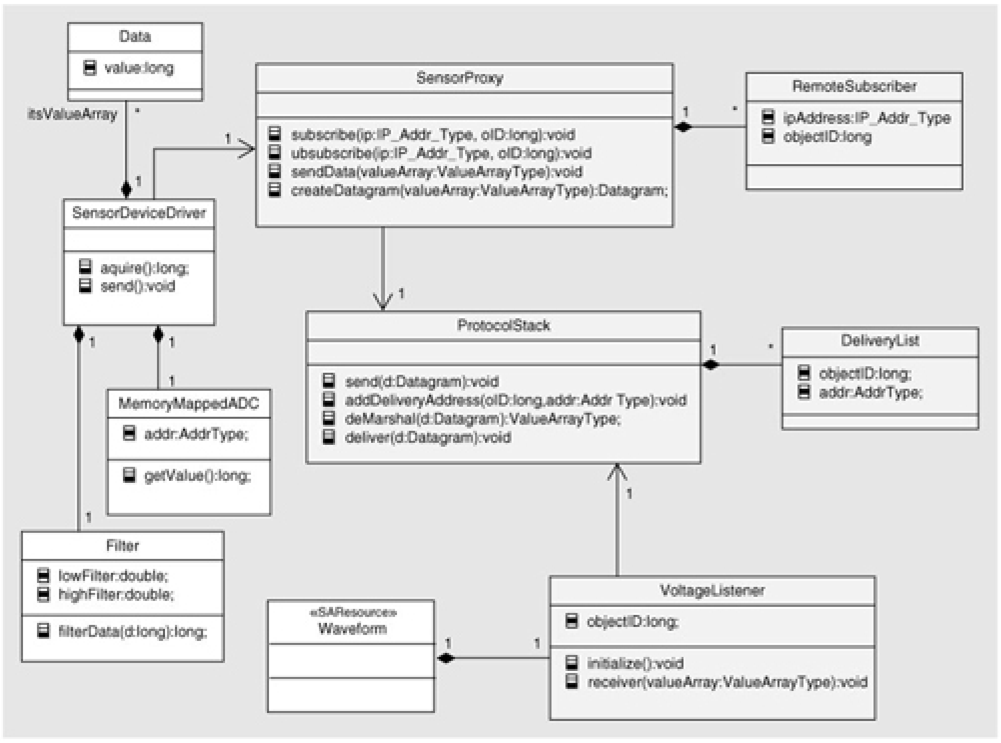
The distribution architecture centers on the technologies and techniques by which objects executing in different memory spaces find each other, communicate, and collaborate. Technologies include the network hardware and infrastructure as well as the communication protocol stacks. Different patterns can be layered on top of the protocol stack to provide different benefits (at different costs). Common patterns include the Proxy, Data Bus, Broker, and Port Proxy patterns,[20] but there are many others.

20. See my book *Real-Time Design Patterns* and Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture,* vol. 2, *Patterns for Concurrent and Networked Objects* (New York: John Wiley & Sons, 2000).

The distribution architecture is usually shown on one or more class diagrams that include the elements that manage the marshaling, transmission, and delivery of information or events.

Figure 2.23 shows a typical example of a class diagram with distribution information added. The classes added to implement the distribution patterns are shown with shading.

Figure 2.23 *Distribution diagram*



2.6.4. Safety and Reliability Architecture

Safety and reliability aren't necessary aspects for all systems but they are for many real-time and embedded systems. The safety and reliability architecture is concerned with the identification, isolation, and correction of faults during system execution. The safety aspect centers on doing the right thing, since safety is defined to be "freedom from accidents or losses."[21] Safety does not concern itself with the delivery of services, only the protection from harm. An **accident** is a loss of some kind, usually referring to harm to one or more persons, but it can also refer to equipment damage or monetary loss. A **hazard** is a condition that will lead to an accident, such as operating an aircraft with insufficient fuel. The **severity** of the accident is a quantitative measure of how serious the accident is or could be. Different safety standards have different quantification schemes for severity. The **likelihood** of an accident is a measure of how often an accident occurs with the hazard present. Likelihood is typically measured as a percentage of occurrences (e.g., 0.3% likelihood means that the accident will likely occur three times out of 1000 executions) or as a stochastic value of the likely execution time necessary for an accident to occur. The **risk** associated with an accident is the product of the likelihood of an accident and its severity.

21. Leveson, *Safeware.*

Reliability, on the other hand, concentrates on the delivery of services. Reliability is almost always measured as a stochastic value, such as mean time between failure (MTBF).

Reliability and safety are often opposing concerns. If a system has a fail-safe state (a condition under which the system is always free from accidents), then if there is a detected problem, the safest thing to do is to go to that state. Most often, services are either not delivered or only a small subset of services is delivered in the fail-safe state, so going to this state decreases the system's reliability. If there is no fail-safe state, then usually improving the reliability also improves the safety. In any case, detailed analysis is required.

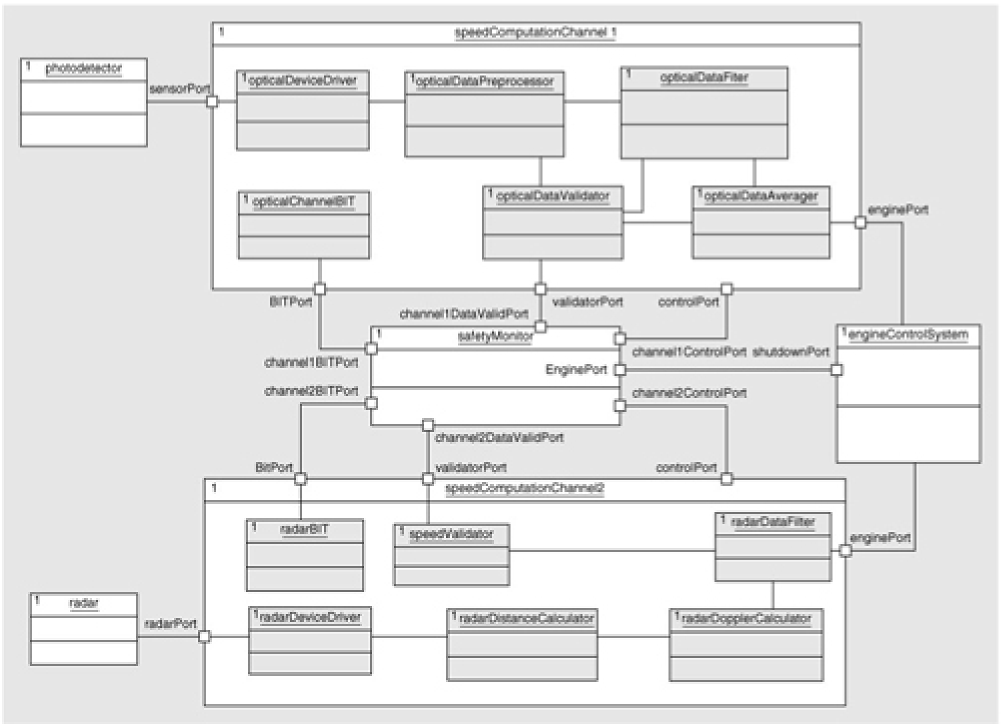
The primary views for the safety and reliability architecture are class diagrams showing the redundant elements with behavioral or interaction diagrams illustrating the fault identification and response behaviors. In addition to these UML views, it is customary to add other analytical means, such as FTA or FMEA. A **hazard analysis** is a common document for safety-critical or high-reliability systems. The hazard analysis correlates faults, fault identification means, fault tolerance times, fault corrective measures, fault identification times, and fault action times.

Since safety and reliability architecture fundamentally requires redundancy, the patterns for safety-critical systems and high-reliability systems focus on how to use redundancy in different ways to achieve different effects. Two primary approaches for architectural redundancy include homogeneous and heterogeneous redundancy. Both these types of patterns are based on the Channel Pattern,[22] in which a channel is a subsystem that contains parts that transform data in a series of steps from raw acquisition through controlling the external environment. The advantage of the Channel Pattern is that adding redundancy is simply a matter of replicating the channel and providing logic that controls the execution of the different channels. In homogeneous redundancy, the channels are identical. For this reason, homogeneous redundancy provides protection against failures ("It used to work but it broke" kind of faults) but not against systematic faults (design or coding errors). Heterogeneous redundancy, on the other hand, uses a different design or design team to create the different channels and so provides some protection against both failures and errors.

22. See my *Real-Time Design Patterns*. Another good reference is Robert Hanmer, *Patterns for Fault Tolerant Software* (New York: John Wiley & Sons, 2007).

Figure 2.24 shows a heterogeneous architecture for speed control of a high-speed train. One channel uses an optical sensor that times the passage of marks affixed to the wheel while the other channel computes the Doppler shift of a radar beamed down at the track. Both channels have steps for data validation, and both also have parts that execute periodic built-in tests (BITs).

Figure 2.24 *Safety and reliability architecture*



2.6.5. Deployment Architecture

In general, the deployment architecture specifies the responsibilities of elements from different engineering disciplines and how those elements interact among the different disciplines. In real-time and embedded systems, the relevant engineering disciplines include software, digital electronics, analog electronics, and mechanical, optical, and chemical engineering. In the simple case, in which the hardware is already predefined, the deployment architecture is limited to identifying how the software maps onto the digital electronics.

The deployment architecture is related to the distribution architecture, but they have distinctly different concerns. The distribution architecture concentrates on the technologies and techniques for object communications, whereas the deployment architecture focuses on the allocation of responsibility to different engineering disciplines and how elements from the disciplines collaborate to achieve the system functionality.

In systems with hardware and software codevelopment, the job of the deployment architecture definition is usually relegated to the systems engineer. This is normally part of what is called **architectural analysis** and produces a work product called a **trade study** in which different hardware/software deployment allocations, technologies, and platforms are evaluated against a weighted set of design criteria (often known as measures of effectiveness [MOE]).[23]

23. For a simple but well-respected approach called the "weighted objective method," see Nigel Cross, *Engineering Design Methods: Strategies for Product Design, Third Edition* (New York: John Wiley & Sons, 2000).

Two primary approaches to the allocation of software to digital electronic hardware are **asymmetric** (static design-time allocation) and **symmetric** (dynamic runtime allocation). Asymmetric allocation is simpler by far but less flexible and less fault-tolerant than dynamic allocation.

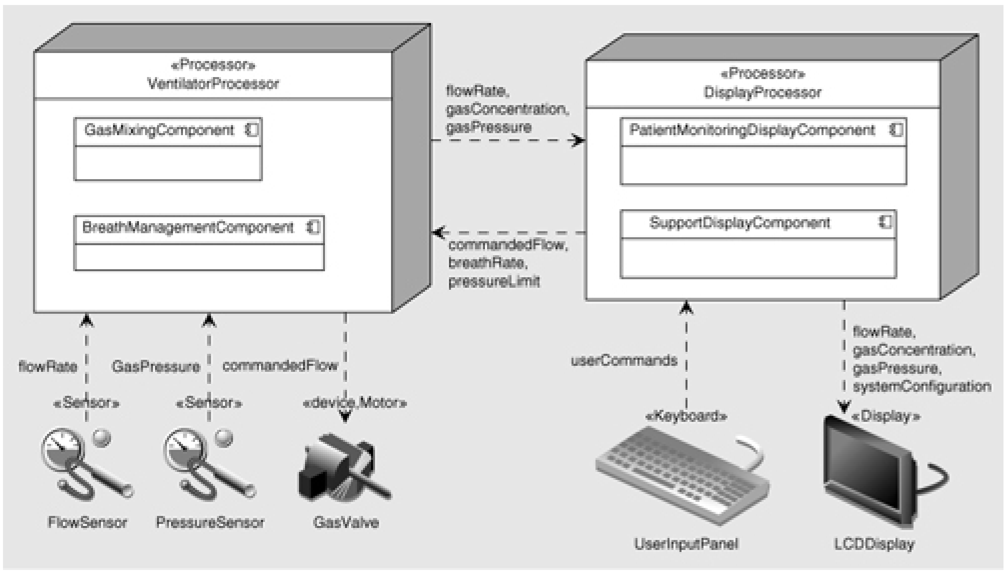
The UML provides a "deployment diagram" as a distinct diagrammatic type. However, it has weak semantics and is not very expressive. For this reason, the SysML profile declined to use the predefined UML deployment diagram and instead uses class ("block") diagrams for this purpose. However, to be fair, if the system is using commercial off-the-shelf (COTS) or previously developed hardware, the standard UML deployment diagram may be adequate.

The standard deployment diagram provides two primary structural element types: nodes (hardware elements) and artifacts (software architectural pieces existing at runtime). Components are mapped onto the hardware elements either by placing them within the node or by linking the components to the nodes with `«deploy»` dependency relations. Nodes specialize in many ways. In the simple case, a node may be a `«device»` (hardware that doesn't execute software you develop) or a `«processor»` (onto which you can map your component artifacts).

In addition, flows can be used between the elements to show information exchange.

Figure 2.25 shows a typical UML deployment diagram. Note that `«device»` nodes are more precisely stereotyped and special icons are used for some of them. Again, this is common in UML deployment diagrams.

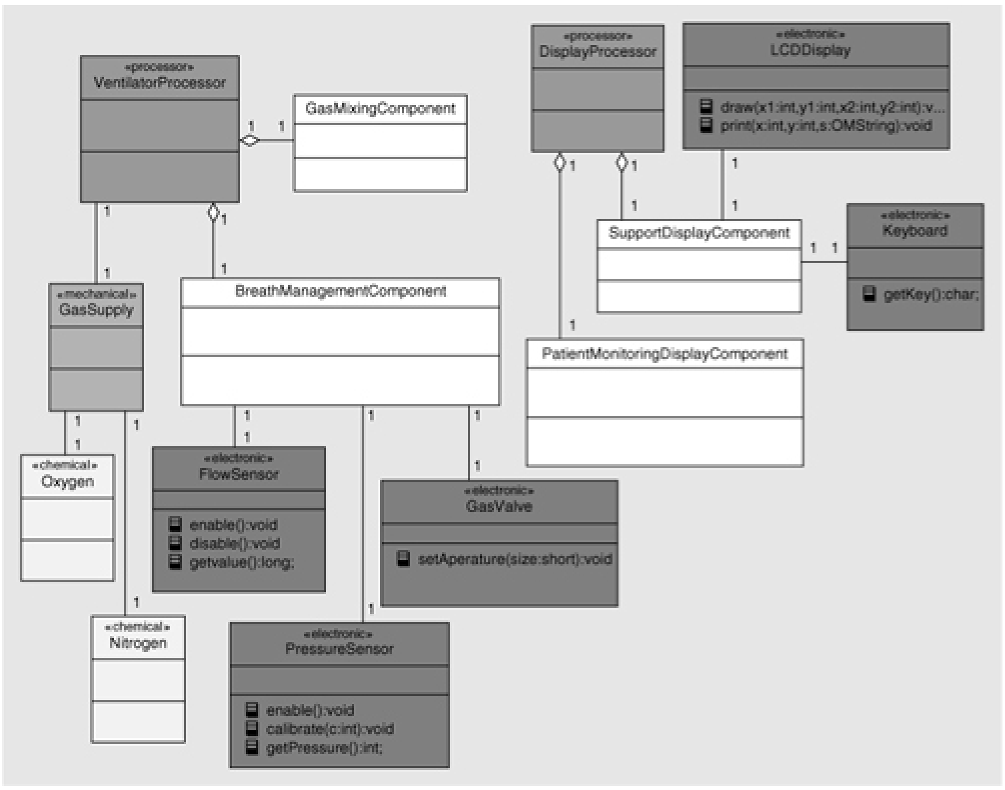
Figure 2.25 *UML deployment diagram*



Alternatively, a class diagram can be used. The advantage is that the class diagrams have richer semantics, and more precision can be used to depict the services and data features of the hardware elements.

Figure 2.26 shows elements from different disciplines, including mechanical and chemical elements. These are shown both with shading (for nonsoftware elements) and stereotypes. Some of the electronic elements are shown with services.

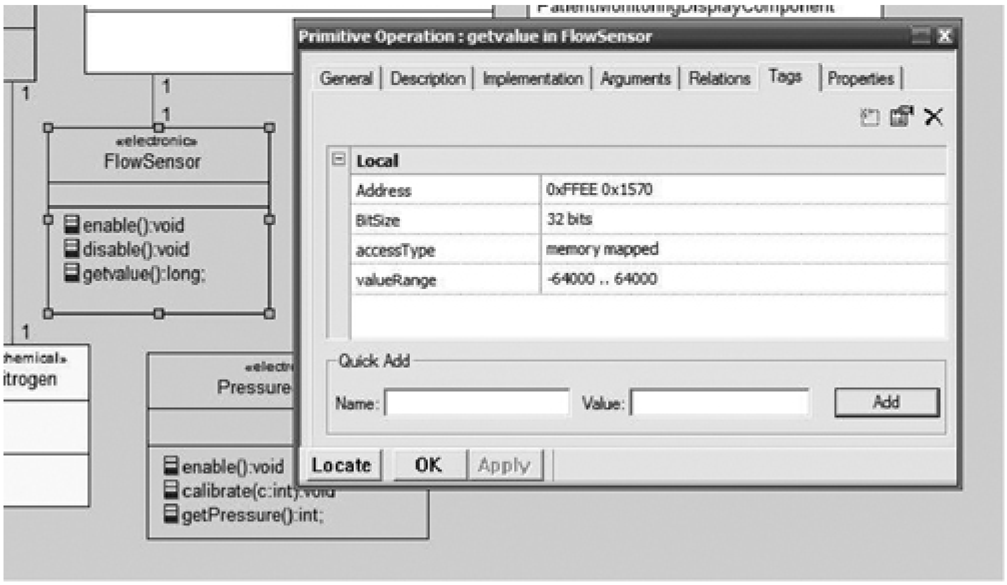
Figure 2.26 *Deployment with a class diagram*



The details of these services—such as whether they are memory-, port-, or interrupt-mapped, the bit allocations of the values, and so on—can be specified in tags defined for those operations.

Figure 2.27 shows that the service `getValue()` provided by the electronic flow sensor is memory-mapped at address 0xFFEE 0x1570, is 32 bits in size, and has a value range of –64000 to 64000. You can see that it is easy to access important interface detail when class diagrams are used instead of the standard UML deployment diagrams.

Figure 2.27 *Tags specifying interface details*



Local	
Address	0xFFEE 0x1570
BitSize	32 bits
accessType	memory mapped
valueRange	-64000 .. 64000

2.6.6. Secondary Architectural Views

There are, of course, architectural decisions to be made beyond those described in the five key architectural views discussed above. Harmony/ESW relegates them to be secondary architectural views because while in any specific system they may be important, they are rarely as fundamental as the decisions made in the five key views.

Some important architectural viewpoints include the following:

- Information assurance

- Data management

- QoS management

- Error- and exception-handling policies

- Service-oriented architecture (SOA)

These architectural views are pattern-driven as well in Harmony/ESW. Information assurance is concerned with managing data-related risks, including security (freedom from inappropriate release of secure data and protection from hostile attack).[24] There are a number of ways to secure data and to prevent system attacks.[25] Data management refers to policies for storing, retrieving, and managing data, including persistence, validation, and storage. QoS management refers to a whole host of qualities, including execution time, precision, accuracy, and data robustness. Error- and exception-handling policies identify the set of fault conditions and lay out precise actions (including displayed user messages). Of course, this architectural aspect must align with the safety and reliability architecture. SOA usually entails architectural components for a service repository, service identification, and service brokering. SOA is normally exclusively an enterprise (system of systems) architectural concern and so may not appear in the development of individual systems.

24. *Security* and *safety* are distinctly different terms in English but share a common word in German (*sicherheit*). So, German readers should be aware that I mean something different from safety here.

25. See, for example, Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad, ***Security Patterns: Integrating Security and Systems Engineering*** (New York: John Wiley & Sons, 2006).

2.7. Coming Up

In this chapter we explored the concepts and benefits of models and a model-driven development approach. Models provide the ability to view different aspects of a system (e.g., functional, structural, behavioral, and interactive) and examine those aspects at different levels of abstraction (e.g., system, subsystem, component, task, primitive element). MDA, a specific set of standards within the OMG, defines a means for organizing models that provides benefits of reusability and maintainability for complex systems composed of heterogeneous technology. The CIM defines the requirements model, which focuses on stakeholder needs. The PIM focuses on the essential aspects of the system that must be present. The PSM maps the PIM to a target platform by including a set of technologies and design patterns. The PSI is the code generated from the PSM. Model translation enables developers to move quickly and confidently from one model to the next.

The Harmony/ESW process defines five key views of architecture: the subsystem and component view, the distribution view, the concurrency and resource view, the distribution view, and the deployment view. Through the use of models, not only can diagrams for each of these views be created and maintained, but their consistency is ensured through the use of a model repository. MDD (or MDA, if you prefer) is an enabling technology, independent from and synergistic with agile methods.

The next chapter provides an overview of and justification for the Harmony/ESW process in terms of core principles and how those principles are realized in practice. Chapter 4, “Process Overview,” goes on to show how the principles and practices manifest themselves in the process at a high level, the iterations and their phases, timescales, concepts, and core principles. Subsequent chapters will drill down into the process details and provide detailed guidance for how to execute and manage the process. Chapter 5, "Project Initiation," discusses project initiation and how to start up a real-time agile project, including the artifacts you may (or may not) need, depending on the nature of the system being developed. Chapter 6, "Agile Analysis," talks about how to do agile analysis and focuses on creating an iteration plan for the microcycle, clarifying the requirements with the CIM, and creating and validating the PIM. Chapter 7, "Agile Design," is all about design and the creation of the PSM. This chapter includes all three levels of design—architectural, mechanistic, and detailed. Chapter 8, "Agile Testing," drills down into testing, particularly continuous integration strategies and end-of-the-microcycle validation testing. Finally, Chapter 9, "Agile Process Optimization," handles how the agile process self-optimizes by identifying and managing issues related to schedule, architecture, risks, workflows, and the process itself.

Chapter 3
Harmony/ESW Principles and Practices

The essential principles and practices of the Harmony/ESW process drive the process content because a process is just a recipe for achieving a goal. It specifies roles (who does the work), tasks (what the work is), workflows (the sequences of tasks), and work products (the stuff the recipe creates). The elements of a process are selected and organized on the basis of its underlying principles. A **principle** is a fundamental generalization that is accepted almost as an axiom, a statement assumed to be true and that can be used as a basis for reasoning and acting. Practices are the concrete realization of principles. A **practice** is a customary or common way of acting that is consistent with the principles. Practices are elements that are used as templates for behaviors during the performance of tasks within a process. Thus, in order to understand why the Harmony/ESW process is as it is, it is important to understand its conceptual underpinning, its principles and practices. These principles and practices are really variants on just a few key ideas such as dynamic planning, getting continuous feedback on the quality and progress of the work being done, and the use of abstraction (modeling) to achieve the product goals.

3.1. Harmony Core Principles

The Harmony/ESW process is designed to realize a set of guiding principles. Understanding these principles will help you understand how the workflows of the Harmony/ESW process are organized and why.

The Harmony core principles include the following:

• Your primary goal: Develop working software.

• You should always measure progress against the goal, not against the implementation.

• That is, your primary measure of progress is *working software.*

• The best way not to have defects in your software is not to put them there in the first place.

• Continuous feedback is crucial.

• Five key views of architecture define your architecture.

• Secondary architectural views supplement your architecture.

• Plan, track, and adapt.

• The leading cause of project failure is ignoring risk.

• Continuous attention to quality is essential.

• Modeling is crucial.

Each of these principles is explained in the following sections.

3.1.1. Your Primary Goal: Develop Working Software

The most important single principle is that your goal as a developer is to create high-quality software that meets the needs of the customers. This premise is so basic that people lose sight of it all the time. It is very common for developers and managers to spend inordinate effort and time on activities that don't directly (or sometimes even indirectly) contribute to this primary goal. A consequence of this principle is that almost every task you do should contribute directly to the specification, creation, testing, or deployment of demonstrably high-quality software. And by *demonstrably* I mean "demonstrably through formal or informal execution."

This principle influences the way in which you will work. On a daily basis, your primary work should center around:

• Understanding what you need to do to add working functionality to your software *today*

• Focusing on getting what you have to execute *today*

• Debugging the software that exists *today*

• Extending functionality with small incremental changes and then executing and debugging *today*

• Hiding from activities that don't contribute to getting your software working *today*

What about other necessary things, such as attending meetings, writing documentation, and filling out time cards? This principle doesn't mean that you shouldn't do those ancillary activities. In fact, many of them are necessary for your project and for your business. But you must always keep this in mind: *Most of your time should be spent doing activities that directly contribute to the development of working software*. A good rule of thumb is that over the life of the project, you should spend 80% of your time doing activities that contribute to the delivery of working software; at least 50% of your time should be spent performing activities that *directly* contribute to the development of working software.

Of course, that doesn't mean that you're always working in Emacs[1] pounding out code. There are many activities that contribute to the development of working software:

• Capturing your requirements in requirements documents

• Capturing your requirements in use case models

• Creating your PIM

• Identifying and characterizing your design optimization criteria

• Applying design patterns to create your PSM

• Generating your code from your model (or hand-writing code when appropriate or necessary)

• Debugging your model

• Developing and applying test vectors for formal testing

1. My wife, Sarah, is an Emacs zealot—I mean *enthusiast*—but she's willing to admit that if you absolutely *must*, you can also write code with vi if Emacs isn't available.

It is important to remember that this is the most important single premise. If you retain nothing else from this book, remember to generate, compile, and run your software several times per day and avoid doing work that interferes with your ability to do that.

3.1.2. Measure Progress against the Goal, Not the Implementation

As a part of planning the project, the tasks necessary to develop the desired system are identified. This will include activities such as writing requirements, writing lines of code, removing defects, and so on. However, these tasks aren't the goals of the project; they are how we've decided to achieve the goals. Because we've developed plans that schedule and sequence these tasks, it is perhaps natural to use the completion of these tasks as measures of progress.

Natural, perhaps, but misguided.

The approach of measuring progress by tracking implementation leads to counting requirements, counting lines of code, and counting defects. The problem with that is that these things measure how well we are working against the plan and *not* how well we are meeting our goals. Plans are good, but realizing the plan is not identical to achieving the goal. Certainly, we hope the two are strongly correlated, but plans are usually wrong, to some degree or another. This is because of things we don't know and also because of things we know that are incorrect or change.

Counting lines of code against expectation is an easy measure but doesn't accurately reflect how close we are to realizing the goal of delivering the product. First of all, we may have an idea how many hundreds of thousands of lines of code we expect, but we don't know the actual number until we are done. So if I think I'm going to create 400,000 lines of code and I have 420,000, how close to done am I? Further, if I'm optimizing the code and go from 420,000 lines to 380,000 lines, have I done "negative work" that I can save up and spend later? If I spent a couple of weeks adding zero net lines of code but remove all existing defects, have I made zero progress against the goal?

One more time: Tracking where you are against your plan is important, but it is not a reliable or accurate measure of how close you are to realizing your goal of delivering the product.

What you need are measures of progress that map directly to the project goals. Systems exist to provide functionality, and the primary goal is to deliver working systems. Therefore, your measures of progress should directly measure the degree to which that goal is met. The best measure, then, is the amount of working functionality that is validated and "delivered" in the current project state. If you model your requirements as use cases, then a coarse measure is the number of use cases validated in the currently executing version of the system. If you use text-based requirements, then a finer-grained measure is the number of requirements validated in the currently executing version of the system.

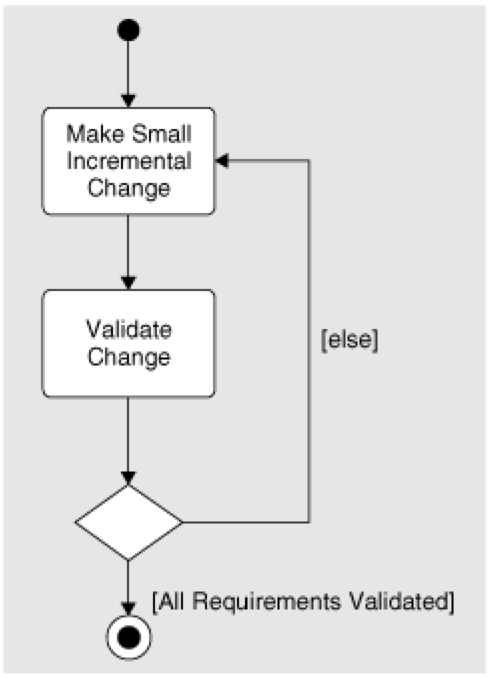
3.1.3. Your Primary Measure of Progress Is *Working Software*

This principle, that the amount of correctly implemented functionality is your primary measure of progress, is a direct and inescapable corollary of the previous principle, but it is so important that it is worth expressing again. You truly know that you're making progress only when the software you're executing today includes demonstrably more (or better) functionality than it did the last time you measured. This principle has some practical consequences.

The first one is that for the amount of working (e.g., debugged or validated) functionality to be a useful measure of progress, you need to measure it *often*—at least daily. This implies that you will do small incremental changes and execute your model several times per day. Although not strictly required, modeling tools that either generate executing code or at least simulate the behavioral semantics of your model greatly facilitate achieving your goal and providing the measures of progress. The generation of code can be done manually, just as you can type in the assembly language equivalent to your C statements, but most people find it more efficient to let the model (or C) compiler do that part of the job.

This principle also means you need to begin executing your models very early in the development cycle. Thus you may end up with 10,000 lines of code realizing a use case representing 50 requirements, but you need to begin executing the model realizing one or two of those requirements even though you may have only 30 lines of code. From a practical standpoint, this means that your basic workflow is as shown in Figure 3.1. A more detailed form of this workflow will be shown in the next chapter, but this illustrates the basic idea of making progress as a series of small incremental changes, each of which is validated before you move on.

Figure 3.1 *Basic nanocycle workflow*



3.1.4. Don’t Put Defects in Your Software!

The best way not to have defects in your software is not to put them there in the first place.

You're probably thinking, "Yeah, but how do I do that?" The way that the Harmony/ESW process recommends is shown in Figure 3.1. The reason that software developed in a more traditional fashion has so many defects is because the software isn't executed very often (so defects have the opportunity to be created and to breed[2]), the design or code isn't tested against its expected behavior frequently, or the design or code isn't defensively oriented, a concept known as "defensive development."

2. I swear that software defects breed like . . . well, like bugs.

Frequent Execution

The best way to immediately identify defects as you are developing your application is to immediately run it. If you're doing code-based agile development, this means that you write a small function or a portion of a larger function, add a variable or two, and run it. It is not uncommon to run your (incomplete but compilable) application multiple times per hour. It is not really any different if you move up the abstraction tree and work in models. You'll add a class, maybe add or change an operation or two, add or change an attribute or two, maybe add a relation—and then run the system or portion thereof. This allows you to immediately identify when you do something dumb. When I develop model-based real-time applications, I typically execute those (incomplete but compilable) models every 10 or 15 minutes. Personally, I do this with the Rhapsody tool from IBM Rational, but other tools can be used as well. If the model compilers weren't as good as they are, I would still perform the same basic workflow, but I'd be writing the code (manually) simultaneously with creating the UML models. I'm glad I don't have to, though, because of the current state of the art in modeling tools.

Executing against Expectations

Executing the partially developed system frequently, however, provides only minimal value. What is really important is not only that it compiles and executes but that it also does the "right thing." This requires that we compare the results of the execution against our expectations. So, what should we expect?

Expectations can be classified as external or internal. External expectations have to do with the collaboration of the element with other elements. Internal expectations have to do with statement-by-statement flow, state changes, and value changes during the execution. External expectations really have to do with meeting the requirements or the requirements realization identified with the use case analysis.

A use case is detailed with scenarios, showing flows of control and data of the system with actors (elements in the system environment), and with state machines specifying the set of such flows. The two views must be consistent; that is, a scenario is a path through the use case state machine, and the state machine is the set of all possible such paths. The level of detail is “black-box,” meaning that the system is just one element of the collaborating set of elements and it is not really possible to see the internal behavior of the system at that level. Thus, a scenario is a sequence of messages between the system and the actors that occur in a specific order and pass specific data values. Change the order or change the messages and you have a different scenario. So, executing the system against expectations boils down to re-creating these use case scenarios, or some portion thereof, from the execution. When the execution of the system elements produces the original use case scenarios, then the elements are doing the right thing. *That* is our criterion of correctness.

Defensive Development

Defensive development is another important practice. The idea behind defensive development is, if not paranoia exactly, at least a deep-seated mistrust in software that you don’t develop yourself. Every function, operation, class, and so on has a set of things it expects to be true. These are formally known as **preconditional invariants.** It is (relatively) easy to write software if you assume these things are true. But what if they’re not?

If the preconditions are not true, then your software will fail in some way or another unless it takes appropriate corrective actions. The steps in defensive development are the following:

1. Explicitly state the software element’s preconditional invariants. In UML, this is commonly done using constraints.[3]

3. A **constraint** is a user-defined “well-formedness” rule. Such constraints may be preconditional invariants (e.g., there is enough memory to create this object), behavioral (e.g., the user will always press the button before turning the knob), structural (e.g., the server always exists and is always ready for a request), or QoS (e.g., the worst-case execution time for the service is 10 ms).

2. Identify the correct responses of the software element when each of the preconditions is violated.

3. Add logic to verify that the preconditions are met before relying on them to be true.

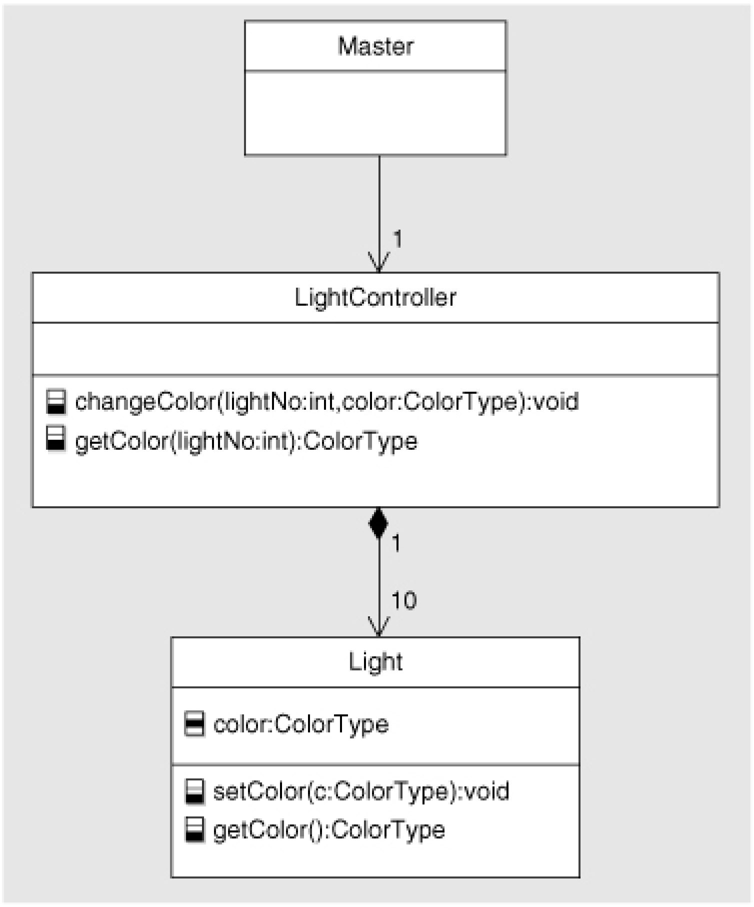
4. Add logic to implement the appropriate corrective actions if a precondition isn’t met.

5. Validate the “sunny-day” (preconditions are true) behavior.

6. Validate the “rainy-day” (preconditions are not true) behavior.

The basic idea of defensive development is, then, that you check your assumptions before relying on them, and if they are not true, you take appropriate corrective action. For example, consider the simple model shown in Figure 3.2 with the corresponding header source code in Listing 3.1. In this model, the `LightController` manages a collection of lights and provides two operations for this purpose. The `changeColor` operation sets the color of a specified light to the specified color, while the `getColor` operation returns the current color of the specified light. Even in this simple example, the `LightController` operations have preconditions that, if violated, will lead to unpredictable behavior.

Figure 3.2 `LightController` *class diagram*


Master
1
LightController
changeColor(lightNo:int,color:ColorType):void
getColor(lightNo:int):ColorType
1
10
Light
color:ColorType
setColor(c:ColorType):void
getColor():ColorType

The `changeColor` operation takes two parameters. The first identifies the light for which the color is to be changed. In the figure we see that the multiplicity of the lights is 10, and that corresponds to the array of pointers to `Light` class. The type of the `lightNo` is int, which has a much larger extent than 0 .. 9. What will happen if the caller request is to set the light value to −1000, 257, or even 10? Something bad, no doubt. The second parameter is the color value. We see in the code listing that this is expressed as an enumerated type with the extent `[BLACK,` `RED, BLUE, GREEN, YELLOW, WHITE]`. However, this is C++, so the type is treated implicitly as an int subtype. So what happens if the caller invokes the service with an invalid color value, such as –1 or 255?

In defensive development, the system element explicitly checks the preconditions and takes the appropriate action if the preconditions are violated. The downsides of defensive development include both runtime and design-time costs:

• The time it takes to dynamically check the preconditions (runtime)

• The extra code space required to implement the precondition checks and corrective actions (runtime)

• The additional development work required to explicitly identify the preconditions and analyze the system to determine what should happen if the precondition is violated (design time)

Listing 3.1 `LightController` Class Header File

```
#ifndef LightController_H
#define LightController_H
enum ColorType {
    BLACK,
    RED,
    BLUE,
    GREEN,
    YELLOW,
    WHITE
};

class LightController {
    ////    Constructors and destructors    ////

public :

    //## auto_generated
    LightController();

    //## auto_generated
    ~LightController();

    ////    Operations    ////
    //## operation changeColor(int,ColorType)
    void changeColor(int lightNo, const ColorType& color);

    //## operation getColor(int)
    ColorType; getColor(int lightNo);

    ////    Additional operations    ////

    //## auto_generated
    int getItsLight() const;

protected :

    //## auto_generated
    void initRelations();

    ////    Relations and components    ////

    Light itsLight[10];      //## link itsLight

};

#endif
```

Obviously, the caller expected the `LightController` to do *something* even if it was poorly specified. Some possible corrective actions include the following:

• Modify the operations so that they return an error code as an int, with 0 indicating success and 1 indicating an error,[4] and otherwise ignore the request.

4. This is a standard C coding idiom—just another reason to love C ☺.

• Throw an exception indicating the kind of error (e.g., `INVALID_LIGHT_ID or INVALID_COLOR`) and let the caller catch the error.

• Accept the out-of-range value but realize it with defaults (e.g., `LightNo` 0 and color `BLACK`).

• Quietly discard the request with no error indication and take no action.

• `LightController` identifies and handles both out-of-range errors.

• `LightController` handles the `LightNo` range errors and the `Light` itself handles the color range errors.

To illustrate defensive development, let us let the `LightController` handle the out-of-range `LightNo` by throwing an exception, and, just to be different, let's let the `Light` validate the color and use the default `BLACK` when the value is out of range. This leads to the code for the `LightController` shown in Listing 3.2 and the code for the `Light` class in Listing 3.3.

Listing 3.2 `LightController` Class Implementation File

```
#include "LightController.h"

//## class LightController
LightController::LightController() {
    initRelations();
}

LightController::~LightController() {
}
```

```
ColorType* LightController::getColor(int lightNo) {
    //#[ operation getColor(int)
    if (lightNo>=0 && lightNo<10)
       return itsLight[lightNo]->getColor();
    else
      throw INVALID_LIGHT_ID;
    //#]
}

int LightController::getItsLight() const {
    int iter = 0;
    return iter;
}

void LightController::initRelations() {
    {
        int iter = 0;
        while (iter < 10) {
            ((Light*)&itsLight[iter])->_setItsLightController(this);
            iter++;
        }
    }
}

void LightController::changeColor(int lightNo, const ColorType& color) {
    //#[ operation changeColor(int,ColorType)
    if (lightNo >=0 && lightNo <10)
      itsLight[lightNo]->setColor(color);
    else
      throw INVALID_LIGHT_ID;
    //#]
}
```

Listing 3.3 `Light` *Class Implementation File*

```
#include "Light.h"
//## package Chapter3

//## class Light
Light::Light() {
}

Light::~Light() {
}

ColorType* Light::getColor() {
    //#[ operation getColor()
     if color is out of range, set it to BLACK
    if (color<BLACK || color>WHITE)
        color = BLACK;
    return color;


        //#]
    }

    void Light::setColor(const ColorType& c) {
        //#[ operation setColor(ColorType)
        if (c>=BLACK && c<=WHITE)
            color = c;
        else
            color = BLACK;
        //#]
    }
```

3.1.5. Continuous Feedback Is Crucial

A principle closely related to the previous one is that you need to get feedback that the system is doing the right thing on a continuous or at least a highly frequent basis. A good rule of thumb is that the developer should never be more than minutes away from demonstrating that what he or she has done so far is right. This is in sharp contrast to more traditional uses of modeling where the modeling phase goes on for months or years before implementation begins. It is vastly more effective to make small incremental changes, run your model, and compare it against the expectations *all the time*. For example, let's suppose we want to build a state machine that specifies complex behavior. I recommend building the complex behavior in a series of small incremental steps of increasing functionality and validating the behavior of each small increment. It is common to do this by adding a flow at a time and validating the state machine as each flow is added. Primary flows are added first, then secondary, and last error detection/correction flows. Let's get more specific.

Suppose we want to build a state machine that accepts a four-digit PIN code for an ATM machine. There are a number of cases that should be implemented in that state machine:

• The PIN should have four digits.

• The card swiped should be linked to a valid account.

• The PIN code should match the customer account PIN code stored.

• The Enter key causes the PIN code to be validated.

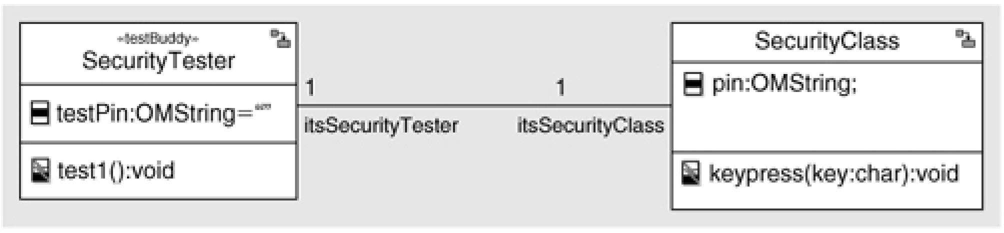
• The Cancel key aborts the operation.

• The Backspace key deletes the last character entered, if the entered PIN code contains at least one digit.

Even though this is a pretty simple state machine, the most effective way to implement it is to do so incrementally.

Step 1 accepts digits and builds a string of digits terminating when the user presses the Enter key but doesn't worry about incorrect string length or canceling or validating the PIN. The class diagram for the partial system is shown in Figure 3.3.

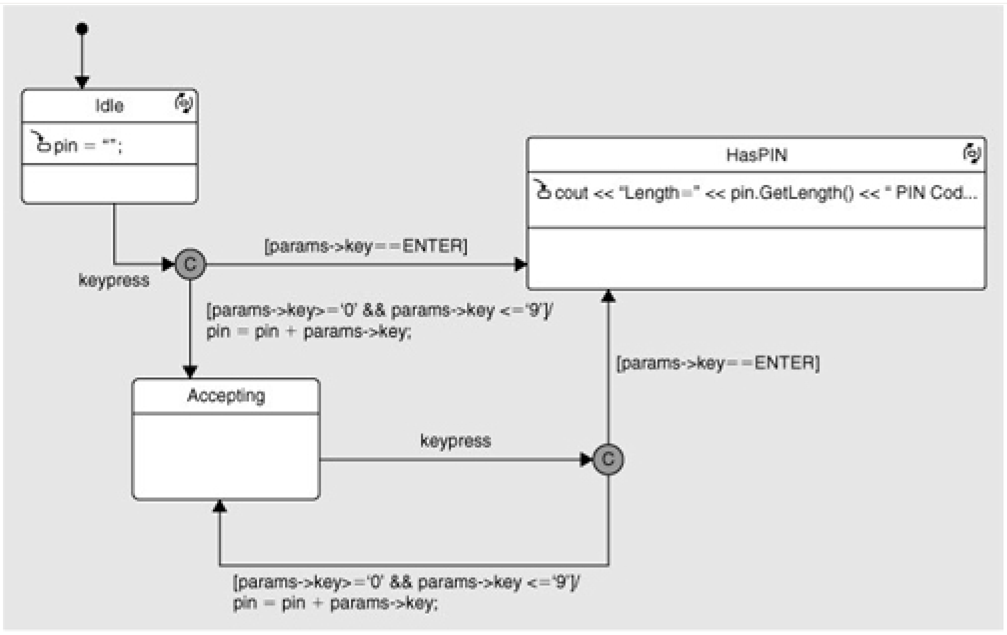
Figure 3.3 `SecurityClass` (step 1)



In order to compare the execution against expectation, the model below uses a `<<testBuddy>>` class to force the test case. This will be elaborated as the functionality of the `SecurityClass` increases.

The `SecurityClass` accepts a keypress event that carries with it the key the user pressed (0 .. 9, Enter, or Cancel). The state machine for the first step is shown in Figure 3.4.

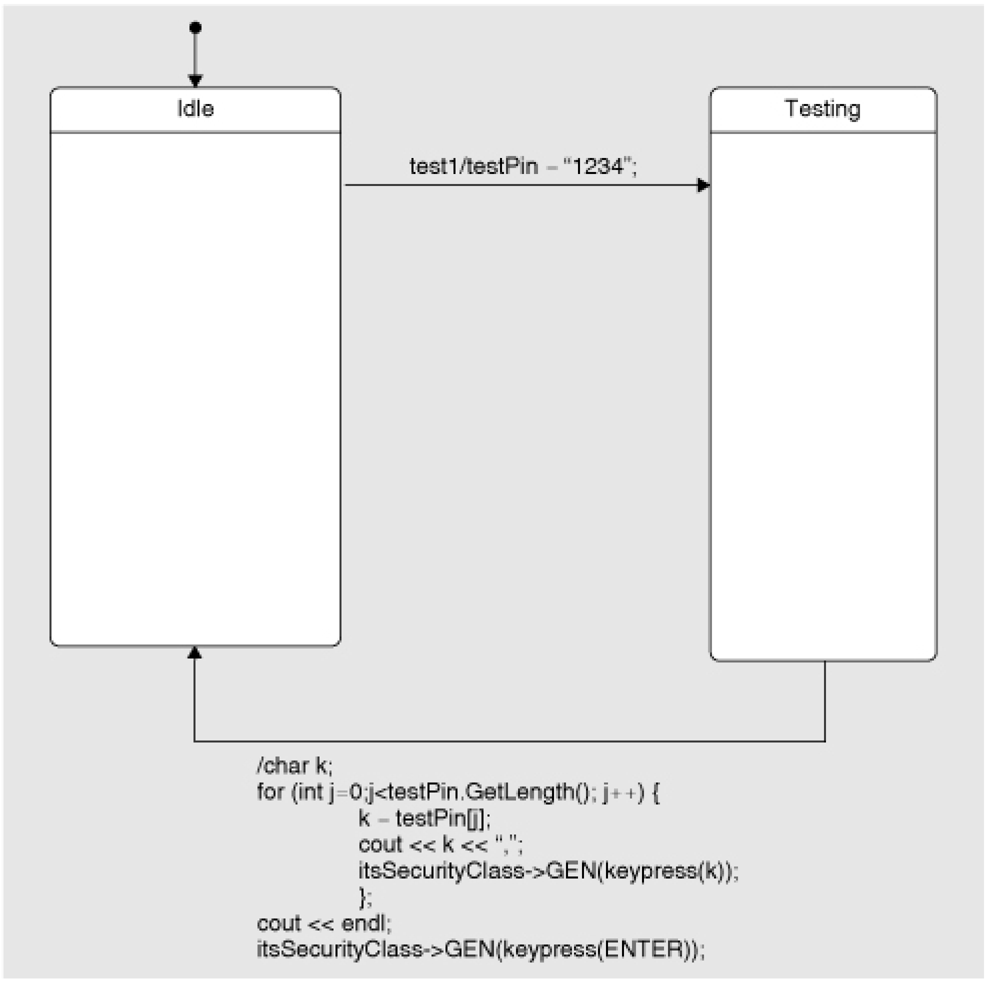
Figure 3.4 `SecurityClass` *state machine (step 1)*



The tool used to create and execute this model (Rhapsody) automatically creates a structure called `params` that contains the passed data values. This event takes a single char parameter called `key`, so to get that value, the model uses `params->key`. The `HasPIN` state has an entry action (an action executed when the state is entered) that puts the length and contents of the PIN code out to standard output. The Boolean conditions enclosed in square brackets are known as **guards** in UML and must be true for that transition path to be taken once the event has occurred.

The state machine for the `SecurityTester` class is very simple at this point (see Figure 3.5). The idea is to create test cases, each of which is triggered by a single event. The state machine sets the value of a local string attribute. The transition back to the `Idle` state is "null-triggered," so it fires as soon as the `Testing` state is entered. The action list simply walks through the string and sends the values of the string to the `SecurityClass` via multiple keypress events. Once it has sent all the characters, it sends an `Enter` key (#declared in the model to be a special character `e`) to complete the test.

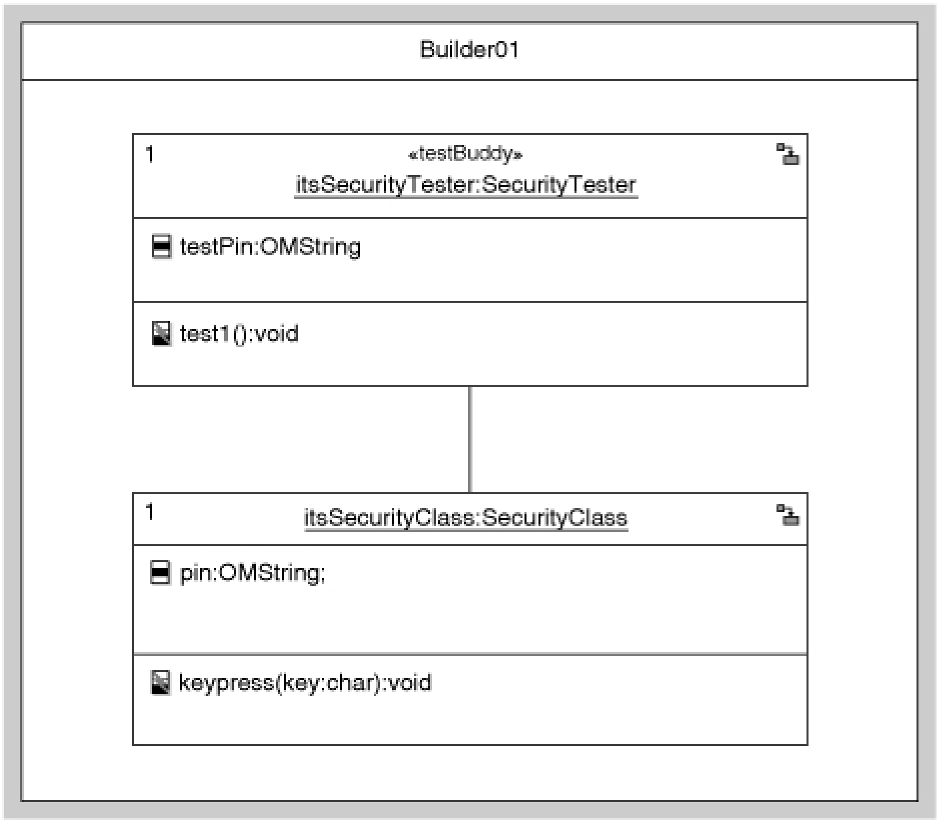
Figure 3.5 `SecurityTester` *state machine (step 1)*



```
/char k;
for (int j=0;j<testPin.GetLength(); j++) {
            k = testPin[j];
            cout << k << ",";
            itsSecurityClass->GEN(keypress(k));
            };
cout << endl;
itsSecurityClass->GEN(keypress(ENTER));
```

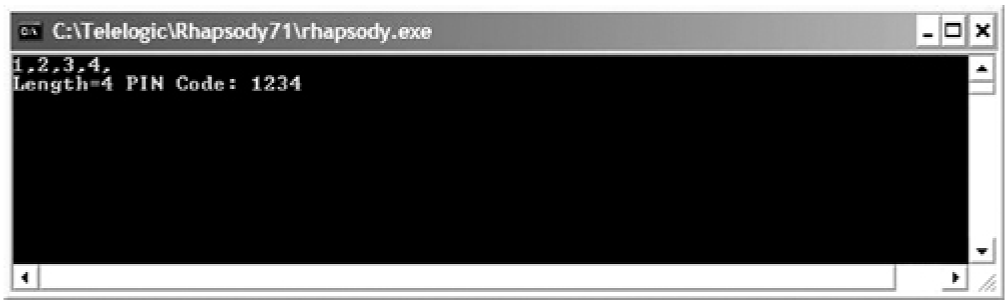
To execute the model, it is simply a matter of generating the code, creating an instance of each of the two classes, running the model, and inserting event `test1`. To simplify this, I created a `Builder` structured class that contains the instances of the two classes with a link between them. Then I only need to create an instance of the `Builder` class in `main ()`, and it will create its parts and link them together (see Figure 3.6), but I could have just written the three lines of code to do that in `main ()`.

Figure 3.6 `Builder` *class (step 1)*



OK, now we can execute the model. We can inspect the value of the attributes or look at the output sent to cout, as shown in Figure 3.7.

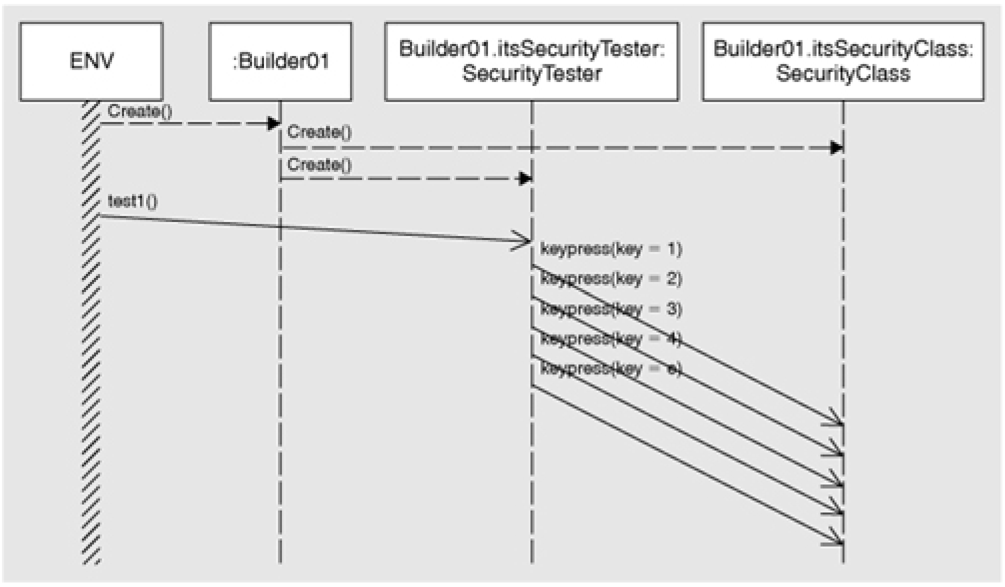
Figure 3.7 *Model execution (step 1)*



```
1,2,3,4,
Length=4 PIN Code: 1234
```

A generally more useful view is to view the execution as an “animated sequence diagram,” that is, a sequence diagram automatically generated from the execution of the system. The advantage of this can be seen in Figure 3.8; the sequence diagram shows the sequence of message exchange as well as the values being passed.

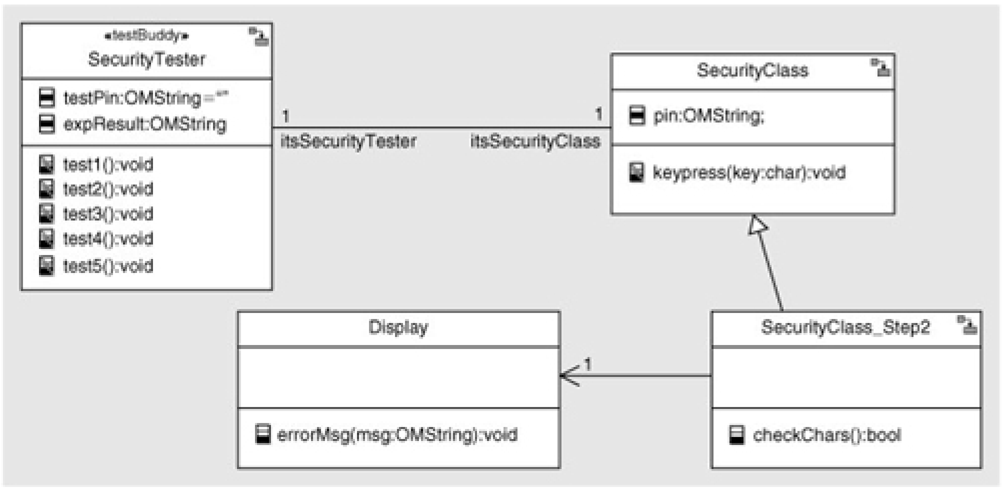
Figure 3.8 *Model execution shown as sequence diagram (step 1)*



Other execution views are possible, but these two illustrate the point.

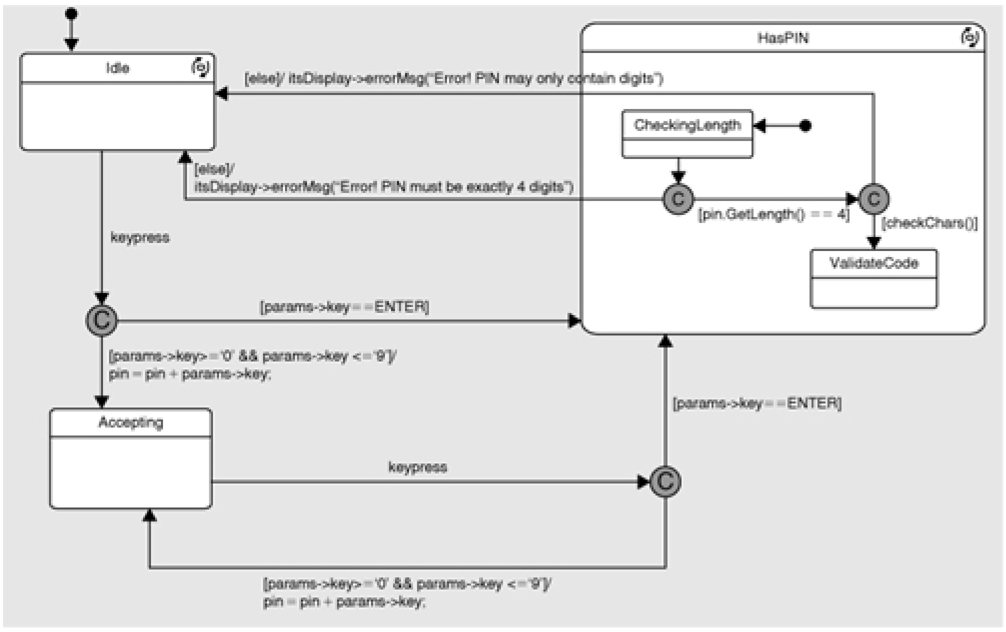
In the next increment, we can add checks to ensure that the PIN is exactly four digits in length and to handle invalid characters (nondigits). The class diagram elaborates to Figure 3.9. Note that the `SecurityClass_Step2` is where we add the additional functionality in this model, but in your development, it will almost always be done “in place” in the `SecurityClass`. The model was constructed in this way to preserve the increment history clearly.

Figure 3.9 `Security` model (step 2)



Most of the change is in elaborating the state machine of `SecurityClass_Step2`, creating the `Display` class, and adding more test cases to the `Security Tester`. Figure 3.10 shows the elaborated state machine for `SecurityClass_Step2`.

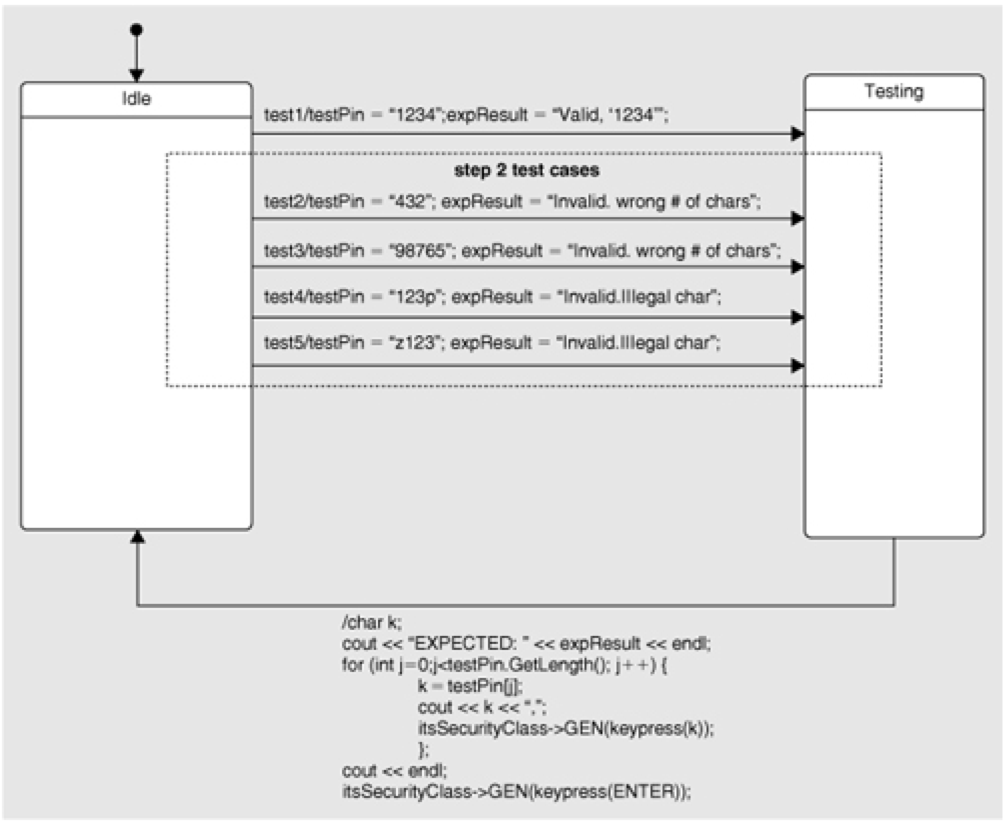
Figure 3.10 `SecurityClass state` *machine (step 2)*



Idle
HasPIN
[else]/ itsDisplay->errorMsg("Error! PIN may only contain digits")
CheckingLength
[else]/
itsDisplay->errorMsg("Error! PIN must be exactly 4 digits")
C
[pin.GetLength() == 4]
C
[checkChars()]
ValidateCode
keypress
C
[params->key==ENTER]
[params->key>='0' && params->key <='9']/
pin = pin + params->key;
[params->key==ENTER]
Accepting
keypress
C
[params->key>='0' && params->key <='9']/
pin = pin + params->key;

The `<<testBuddy>>` class is elaborated simultaneously with the model so that we can immediately validate that the model is doing the right thing so far. Figure 3.11 shows the `SecurityTester` state machine.

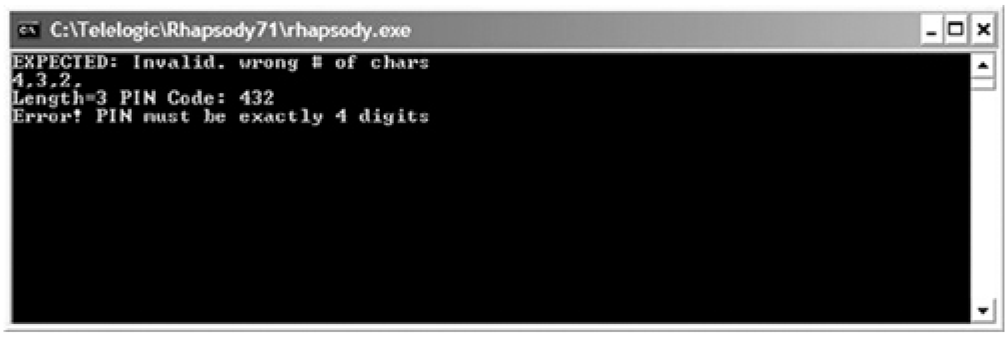
Figure 3.11 `SecurityTester` *state machine (step 2)*


Idle
Testing
test1/testPin = "1234";expResult = "Valid, '1234'";
step 2 test cases
test2/testPin = "432"; expResult = "Invalid. wrong # of chars";
test3/testPin = "98765"; expResult = "Invalid. wrong # of chars";
test4/testPin = "123p"; expResult = "Invalid.Illegal char";
test5/testPin = "z123"; expResult = "Invalid.Illegal char";

```
/char k;
cout << "EXPECTED: " << expResult << endl;
for (int j=0;j<testPin.GetLength(); j++) {
            k = testPin[j];
            cout << k << ",";
            itsSecurityClass->GEN(keypress(k));
            };
cout << endl;
itsSecurityClass->GEN(keypress(ENTER));
```

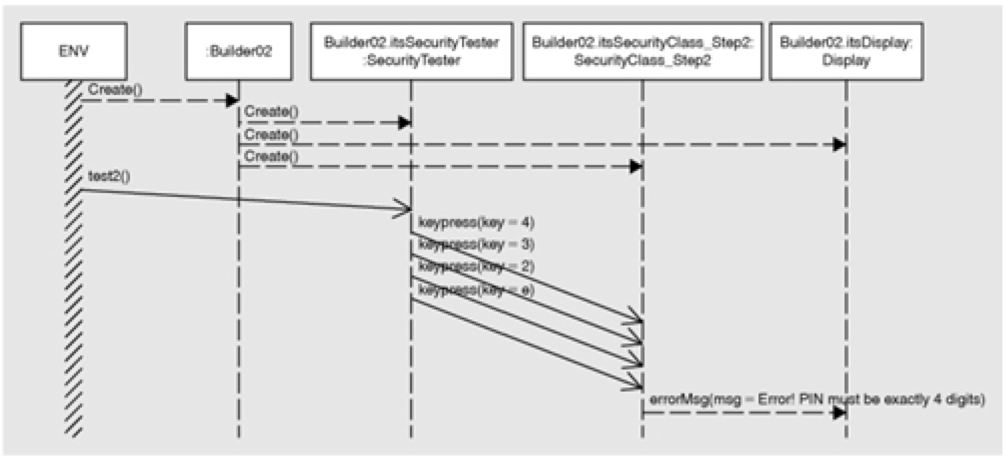
If we run `test2` with the wrong number of characters, we see that the model correctly discovers the error and reports it. The output window is shown in Figure 3.12 and the sequence diagram resulting from the execution in Figure 3.13.

Figure 3.12 *Model execution output window (step 2)*



```
EXPECTED: Invalid. wrong # of chars
4,3,2,
Length=3 PIN Code: 432
Error! PIN must be exactly 4 digits
```

Figure 3.13 *Sequence diagram of model execution (step 2)*


ENV
:Builder02
Builder02.itsSecurityTester :SecurityTester
Builder02.itsSecurityClass_Step2: SecurityClass_Step2
Builder02.itsDisplay: Display
Create()
Create()
Create()
Create()
test2()
keypress(key = 4)
keypress(key = 3)
keypress(key = 2)
keypress(key = e)
errorMsg(msg = Error! PIN must be exactly 4 digits)

The next increment (Figure 3.14 and Figure 3.15) adds the Cancel key processing and also, if the PIN passes the validation checks, compares the PIN with the stored PIN and allows the user to continue with the transaction. This requires elaboration of the `SecurityClass` (or in this case the next subclass since we want to preserve these increments for pedagogical purposes), the `Display` class, and, of course, the `SecurityTester` class. In addition, we added a `PINDB` (PIN database) class to manage the customer information.

Figure 3.14 `Security` *model (Step 3)*

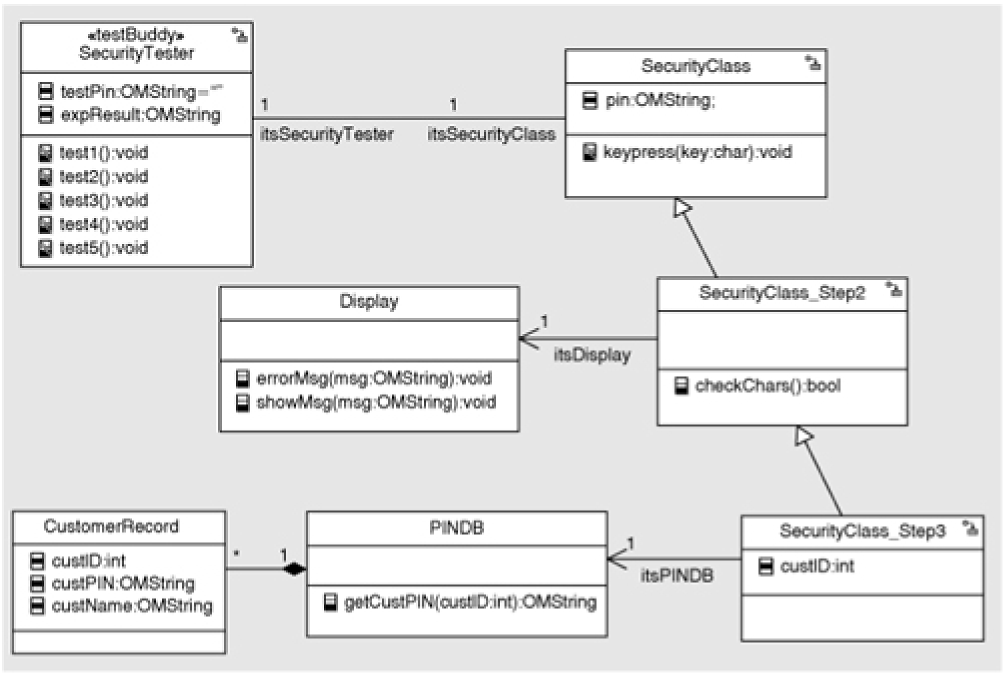
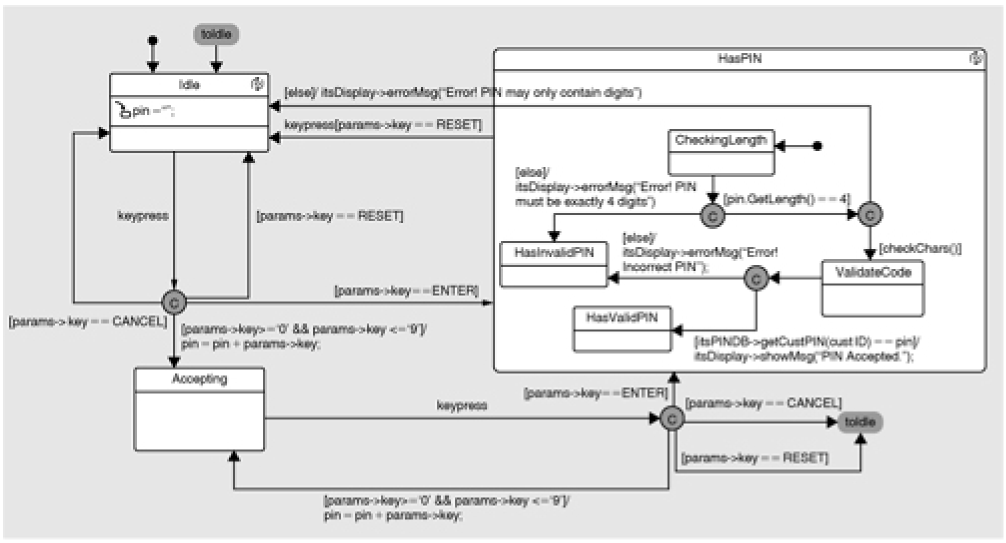
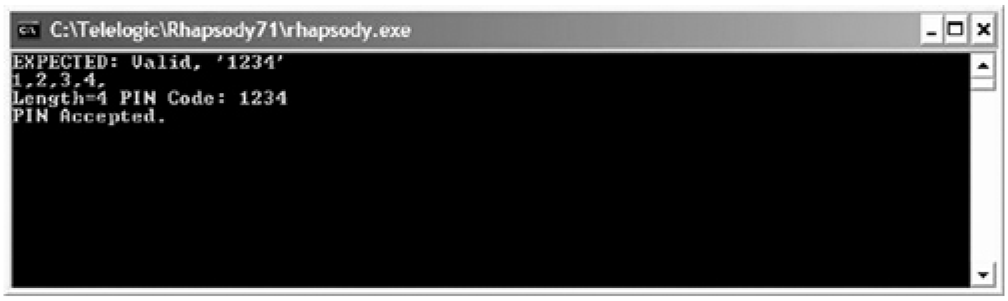


Figure 3.15 `SecurityClass` *state machine (step 3)*



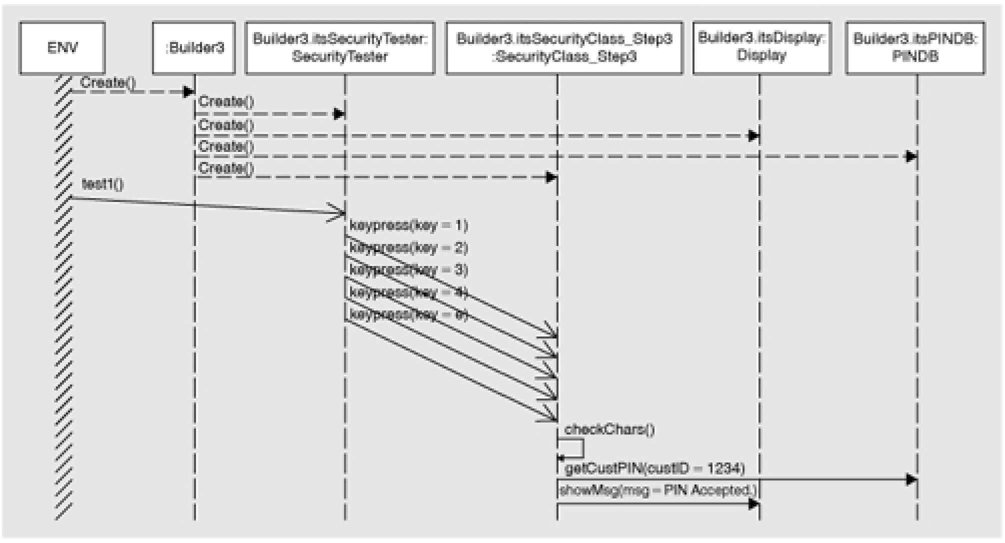
In this case, the execution is shown in Figure 3.16 and Figure 3.17. Of course, by now there are over a dozen test cases to run, but we will show only one here.

Figure 3.16 *Execution output (step 3)*


C:\Telelogic\Rhapsody71\rhapsody.exe

```
EXPECTED: Valid, '1234'
1,2,3,4,
Length=4 PIN Code: 1234
PIN Accepted.
```

Figure 3.17 *Model execution animated sequence diagram (step 3)*



Clearly, during the development of a real ATM system, this process would continue until each scenario of each use case was added and validated.[5] However, this should be enough to provide a flavor of the workflow shown in Figure 3.1. You can see how the collaboration grows in the number and complexity of the elements with each increment. Validating the increment via execution ensures that you're meeting your expectations and not making mistakes that will be more difficult to find later.

5. Exercise left to the reader ☺.

3.1.6. The Five Key Views of Architecture

In the Harmony/ESW process, analysis is all about the specification of the essential properties of a system or element, while design takes that analysis and optimizes it based on the identified and weighted design criteria. Harmony/ESW looks at design at three levels of abstraction:

• *Architectural design*—optimizations of the system that affect most or all elements of the system

• *Mechanistic design*—optimization of the system at the level of an individual collaboration (i.e., a set of object roles working together to realize a single use case)

• *Detailed design*—optimization of the system at the primitive element level (class, type, or function)

Architecture, being an aspect of design, focuses on the optimization of the analysis model but does so at a global scope; that is, architectural decisions are made to optimize the model at a gross, overall level.

The Harmony/ESW process identifies five key views of architecture (see Figure 3.18):

• *Subsystem and component architecture*—the identification of the largest-scale pieces of the system, their responsibilities, and their interfaces

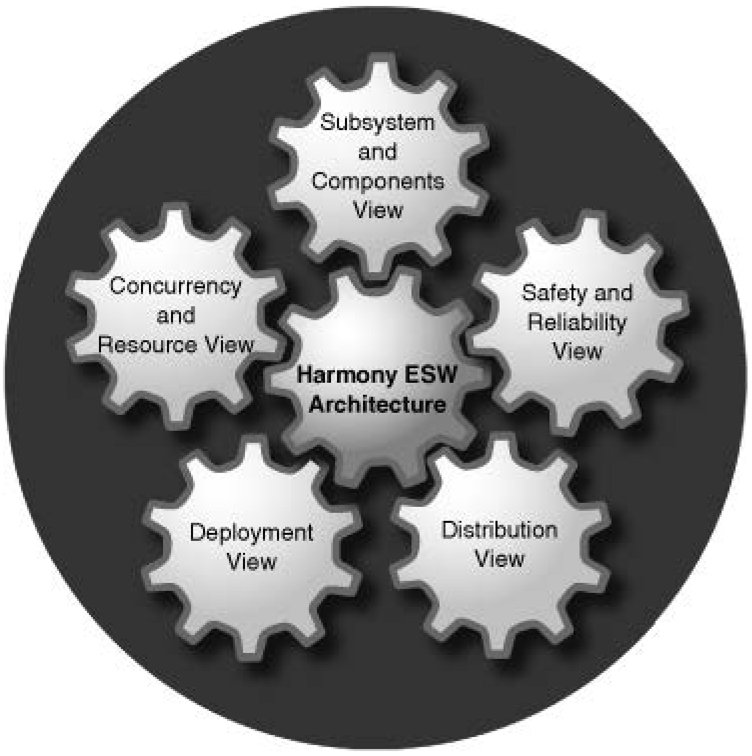
• *Distribution architecture*—the key strategies, patterns, and technologies for how objects will be dispersed across multiple address spaces, how they will find each other, and how they will collaborate

• *Concurrency and resource management architecture*—the identification of the concurrency units (e.g., tasks or threads), how they will be scheduled and arbitrated, the mapping of the semantic ("working") objects of the system into the concurrency units, how these units will synchronize and rendezvous, and how these units will share resources

• *Safety and reliability architecture*—the identification, isolation, and correction of faults at runtime, primarily through the management of different kinds and scales of redundancy and error-handling policies

• *Deployment architecture*—how the different engineering disciplines, such as software, electronic, mechanical, chemical, and optical, will collaborate; the responsibilities of the elements from the different disciplines; and the interfaces between elements of different disciplines

Figure 3.18 *Key architectural views*


Subsystem and Components View
Concurrency and Resource View
Safety and Reliability View
Harmony ESW Architecture
Deployment View
Distribution View

These views are considered key because they usually have the most significant impact on the form, structure, and behavior of the system as a whole. The architecture is decomposed into these particular views because the literature on the subject has “self-organized” into this taxonomy; that is, the computer science and engineering literature focuses on views more or less independently. Thus, it is easy to find self-contained patterns in the concurrency and resource architecture that are independent from the patterns to be found in the distribution architecture, which are independent from the patterns to be found in the safety and reliability architecture, and so on. Because of the nature of this independence, it is usually simple to mix design patterns from these different views together. The system architecture is formed as the union of the design and technology decisions made in each of these primary architectural views.

For example, here are just a few design patterns for the different architectural views:[6]

• Subsystem and component architecture

° Layered Pattern

° Microkernel Pattern

° Recursive Containment Pattern

° Pipes and Filters Pattern

° Virtual Machine Pattern

• Distribution

° Proxy Pattern

° Broker Pattern

6. For the details of these patterns, see my *Real-Time Design Patterns* or the series of books by Buschmann et al., *Pattern-Oriented Software Architecture.*

° Data Bus Pattern

° Shared Memory Pattern

° Port Proxy Pattern

• Concurrency and resource management

° Half-Sync/Half-Async Pattern

° Active Object Pattern

° Message Queuing Pattern

° Guarded Call Pattern

° Rendezvous Pattern

° Ordered Locking Pattern

° Simultaneous Locking Pattern

° Priority Inheritance Pattern

• Safety and reliability

° Fixed Block Allocation Pattern

° Channel Pattern

° Triple Modular Redundancy Pattern

° Protected Single Channel Pattern

° Heterogeneous Redundancy Pattern

° Monitor-Actuator Pattern

• Deployment architecture

° Asymmetric Pattern

° Symmetric Pattern

° Semi-symmetric Pattern

Most of these patterns can be freely matched with patterns from the other key architectural views, but sometimes a pattern will cross the boundaries of these views.

3.1.7. Supplementing Your Architecture with Secondary Architectural Views

While the architecture of a system is primarily based on the five key views, other optimization concerns appear at the architectural scope. In some systems, these aspects may be as crucial as any of the five key views, but in most systems, they are secondary concerns. To be considered an architectural optimization, the design decision must be made above the level of individual collaborations. This means that it can affect multiple elements in several collaborations or it can be concerned with coordinating multiple collaborations. Some common secondary architectural concerns include

• *Information assurance*—Information assurance focuses on managing information-related risk. One aspect of this is **security,** which is defined as "freedom from intrusion or compromise of data privacy or integrity."[7] Information assurance architecture is primarily concerned with data confidentiality and data integrity.[8]

7. In German, *safety* and *security* are both translated as *sicherheit,* but in English the words have different meanings.

8. See Schumacher et al., *Security Patterns.*

• *Data management*—This architectural concern has to do with system-wide policies, patterns, and technologies for storing, manipulating, validating, and managing data.

• *QoS management*—This view focuses on identifying, monitoring, and improving performance dynamically at runtime.

• *Error and exception handling*—This view is closely aligned with the key safety and reliability view but has a narrow focus on the ontological classification of errors and exceptions and the specification of corrective action for each error or exception type.

The secondary architectural views can be important for particular systems but generally have less impact than the five key views of architecture. Nevertheless, they are strategic in the sense that they attempt to optimize the system design at an overall level. You may well find additional architectural concerns for the systems in your own specific problem domain.

3.1.8. Plan, Track, and Adapt

Planning can be categorized as either ballistic or dynamic. Ballistic planning refers to creating and executing a plan without monitoring progress against the plan. This is a very common approach but it rarely has good results. Dynamic planning refers to creating and executing a plan with continuous (or at least frequent) monitoring and adapting the plan as necessary to adjust when progress against plan isn't as expected. In software and system projects, dynamic planning is *always* better than ballistic planning.

Dynamic planning is a recurrent theme in the Harmony/ESW process. It shows up in the BERT scheduling process and the ERNIE schedule-tracking process (to be discussed in Chapter 5, "Project Initiation"). It also shows up in the increment review ("party") phase of the incremental development lifecycle. The basic principle is summed up by DeMarco and Lister when they say, "You don't know what you don't measure."[9] It is crucial for project management to collect meaningful feedback on the progress of a project and then effectively use that information to replan when necessary.

9. Tom DeMarco and Timothy Lister, *Peopleware: Productive Projects and Teams* (New York: Dorset House Publishing, 1999).

Metrics are a large topic and are very frequently misapplied, resulting in a waste of time and effort. Good metrics are ones that are easy to measure and reflect important factors of progress or quality. Too often, metrics are adopted that are easy to measure but don't translate into meaningful information; lines of code is such a poor metric. Good metrics should measure —directly or indirectly—progress against project *goals*. Some good metrics include

• Number of requirements identified, realized, or validated

• Number of use cases identified, realized, or validated

• Number of defects identified or removed

• Number of hours spent on work activities

• Work activities completed

The other aspect—just as crucial as tracking—is adapting. The earlier you adapt to being off course, the smaller (and cheaper) the correction is likely to be. This does not mean that the entire project structure should be redesigned daily. Instead I simply mean that course corrections must be timely to be effective.

3.1.9. The Leading Cause of Project Failure Is Ignoring Risk

Many, many projects fail because bad situations arise, such as loss of staff or funding, technical difficulties, supplier failures, poor or unfeasible requirements, low-quality development, bad management, or low morale. Sadly, far too many projects fail when the failure could have been avoided by actively looking ahead and removing problems before they became manifest. Risk, as defined in previous chapters, is the product of the severity of a (bad) situation (known as a hazard) and its likelihood. To reduce risk, you must do the following:[10]

10. See the SEI paper by Ronald P. Higuera and Yacov Y. Hames, "Software Risk Management" (SEI Technical Report CMU/SEI=96=TR-012) at www.sei.cmu.edu/pub/documents/96.reports/pdf/tr012.96.pdf.

• *Identify*—This step discovers risks while they are still potential conditions that have not yet occurred. Proactive risk reduction steps can be performed only for known risks.

• *Analyze*—This step characterizes and quantifies the risk. This is done by estimating the severity and likelihood of the hazard and computing the risk by multiplying these quantities. This crucial step allows you to focus on the most serious project risks.

• *Plan*—Risk planning identifies and schedules actions that

° Plan for a risk contingency should the risk become manifest

° Avoid or mitigate the risk by reducing its severity or its likelihood

° Combine the previous approaches

• *Track*—This step monitors the status of the project with respect to the hazard and the effectiveness of the RMAs. Tracking allows you to identify changes in the risk (such as increasing likelihood or decreasing severity), which may prompt you to change your risk mitigation actions.

• Control—This is the “dynamic planning” part of risk management. Risk control is all about identifying and correcting deviations from what you want in what you have. If the measures you have taken are not adequately addressing the risk, this step results in modified plans and actions.

• *Communicate*—Communication allows teams to effectively work together to track and control identified risks, as well as to dynamically identify and handle new risks as they emerge within the project.

In the Harmony/ESW process, risk management is a workflow that begins early in project planning and lasts until the product is delivered and possibly beyond. Project initiation includes the identification, analysis, and planning parts of risk management. Once the project is running, risk management is a part of executing the work items associated with the RMAs. Tracking and controlling are done implicitly within the microcycles and explicitly in the “party phase” (microcycle increment review).

For projects that contain circumstances of significant risk, those hazards are typically managed in a **risk management plan** (sometimes known as a **risk list**). The risk management plan lists the hazards in order of risk and for each risk identifies metadata about the risk, such as:

• Hazard description

• Severity

• Likelihood

• Risk

• Risk status

• Risk assessment metric(s)

• Mitigation action(s), including

° What actions are to be performed

° When the actions will be performed

° Who will perform the actions

° How the results of the actions will be assessed

° Who will assess the results

The risk management plan will be discussed more in Chapter 5, when I talk about project initiation.

3.1.10. Continuous Attention to Quality

A common misconception about agile methods is that they are an excuse to “hack.” I find that the opposite is true; I find that really using agile methods frees me to focus on the aspects of the system that actually impact quality. Kent Beck[11] notes that there are two kinds of quality: internal and external. Internal quality is measured by the developers, whereas external quality is measured by the customers. These are clearly not the same, because these different groups understand different things by the term and use different assessment methods, although there is significant overlap.

11. Beck, *Extreme Programming Explained*.

Developers usually think of “quality = correctness,” “quality = robustness,” and “quality = absence of defects.” The correctness criterion means that the software produces the computational outputs that you expect with the degree of precision you expect. The robustness criterion focuses on handling violations of preconditional invariants (such as out-of-range values, lack of sufficient memory, and device failures) properly. Related to the previous criterion, the absence of defects means that the software contains few, if any errors. Developers usually do not consider meeting requirements or system usability to be issues of quality.

Customers most often judge quality as “quality = meets requirements,” “quality = meets expectations,” or “quality = usability.” Customers who pay for, but do not themselves use, the system want to ensure that the requirements are met, even if those requirements are incorrect, inconsistent, or inappropriate. If the customer is a system user who is replacing an existing system with a new one, he or she is likely to view the new system as high-quality if it does exactly what the old one does, including replicating all its warts and wrinkles. End users are also likely to view the usability of the system as a primary determinant of its quality because they are focused on trying to use the system to accomplish their goals. The “principle of least surprise”[12] is often cited as a measure of software quality.

12. This principle states that when users attempt to use the software in a new (for them) way, the software acts in a manner most often expected.

There are, of course, other criteria that can be (and are) considered important to software quality, depending on viewpoint. Some of these are:

• Understandability (of program usage, of messages, of system behavior, etc.)

This is crucial not only for system maintenance and enhancement but also for getting the system to work in the first place. This bullet specifically refers to the structuring and organization of the materials as well as “embedded documentation” (i.e., comments).

• Good documentation

Documentation is important to facilitate understanding (the preceding bullet). This bullet refers to external documentation, such as models and the reports generated from models and the results of analysis tools (e.g., lint).

• Portability

Portability refers to the ease with which a software application can be retargeted to run properly in a different environment. This can be as mundane as taking care of Endian issues or as broad as completely revamping the tasking model or UI.

• Maintainability

Maintainability refers to the ease with which software defects can be repaired and the software can be enhanced. Many embedded systems, such as flight avionics, have lifetimes measured in decades and can expect many rounds of repair and upgrades.

• Conciseness

Conciseness refers to the succinctness or “getting to the point” of the design, code, and documentation. Too often documentation addresses the wrong concerns or does so in a roundabout fashion.

• Testability

Software that is easy to test is generally of much higher quality than software that isn’t. Easy-to-test software will be tested more thoroughly and at a lower cost.

• Reliability

Reliability is a stochastic measure of the percentage of time a system can deliver services. This means not only that the software will continue to deliver services in the absence of faults, but that the system is robust in the presence of faults. Many real-time and embedded systems have very high reliability requirements.

• Efficiency (of time, memory, or other resources)

Of course, most real-time and embedded systems are concerned about efficiency. Most such systems have highly constrained execution environments in terms of both resources (such as memory) and time. While I believe many embedded developers are overly concerned with efficiency (and insufficiently concerned with correctness), efficiency remains one of the primary criteria for quality.

• Capacity

The capacity of a system refers to the load a system can bear. Normally, this is measured in terms of the maximum number of elements managed simultaneously, such as the maximum number of targets that can be tracked by a targeting system or the maximum number of calls managed by a router.

• Throughput

Throughput is similar to capacity but refers to the maximum number of elements processed per unit time, and so may be thought of as the first time derivative of capacity.

• Security

Security, a specific concern of the more general information assurance concept, has to do with the protection of information or resources from intrusion, espionage, or inappropriate access. Not all systems have security requirements, but those that do provide national or consumer protection of secret or sensitive information.

Quality is really all of these things, which is why, of course, software engineers are paid so well.[13]

13. Hint, hint, to all you managers out there!

The concern about agile methods is that because documentation and ancillary work activities in traditional industrial processes are deemphasized, it appears that software quality may suffer. However, these ancillary aspects are deemphasized because they are *ancillary*. Remember the first principle: Working software is your primary goal! To achieve that, you need to measure progress against that goal (next two principles). The feedback needs to be continual and immediate (next two principles). So you can see that the whole point of the first five principles, and the practices that implement them, is software quality.

One practice that is used to improve quality is the notion of “testing first.” This practice emphasizes developing the test cases prior to actually writing the software. This forces developers to think about what should happen in different circumstances and leads them down the path of implementing more robust behavior.

For model-based development, test cases are usually captured as sequence or activity diagrams, complete with input data values and expected output. Alternatively, you can develop `<<testBuddy>>` classes that encapsulate the test cases and the ability to execute them under developer control. In either case, when you run a portion of the model, you apply the test cases to ensure that portion of your model is correct. The tests are applied continually as the software analysis or design evolves. The test cases grow in depth and breadth as the functionality of the system grows.

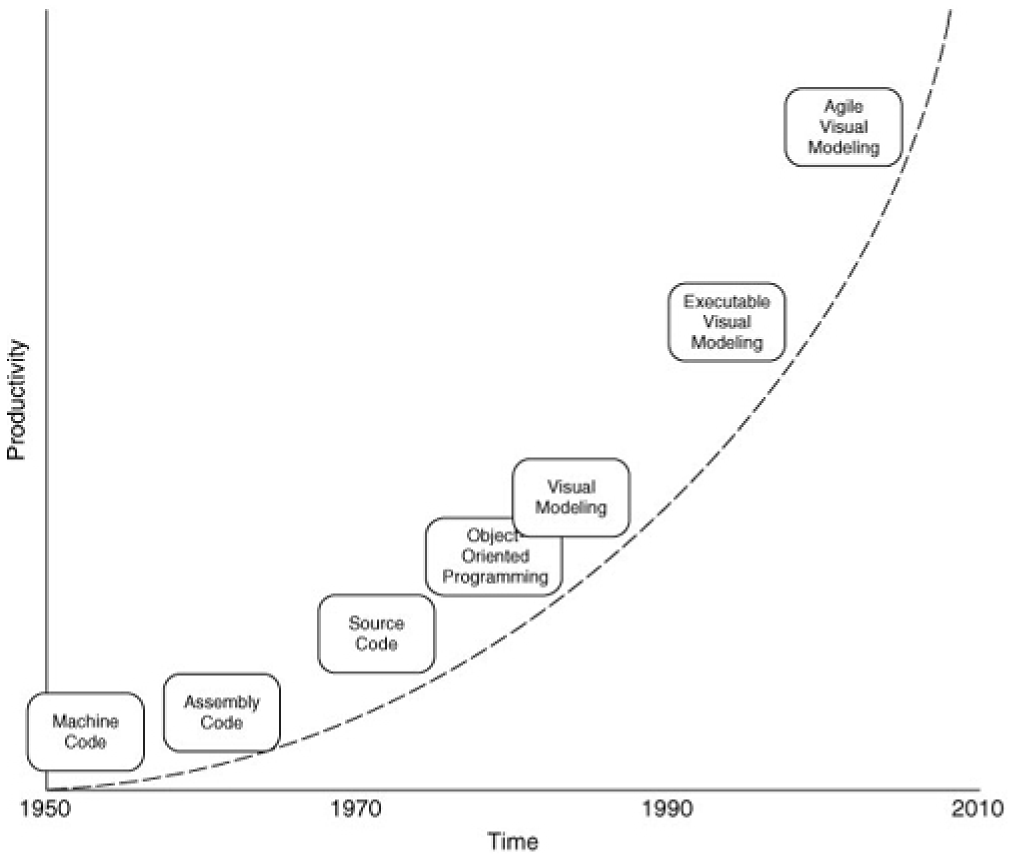
For example, for the ATM security example discussed previously, we created a `<<testBuddy>>` class called `SecurityTester` for just this purpose (refer to Figure 3.3). The very simple first model contained a simple test case (refer to Figure 3.5). As the functionality of the system grew, we added more test cases to the `SecurityTester` (refer to Figure 3.11) to validate the correctness and robustness of the system. The new test cases used illegal characters and invalid PIN code lengths. Early focusing on the test cases concentrated our attention on ensuring that the system continued to do the right thing, even when the input data was wrong.

3.1.11. Modeling Is Crucial

Many developers of embedded (and nonembedded) software view source code as the primary development artifact. However, I believe that they are missing the boat. Ultimately, it is *delivered functionality* that is the primary development artifact. For software-intensive systems, that is the compiled object code executing on the target platform. Source code is important solely as a means to achieve the primary goal. So an important question arises: What is the best way to get there?

In the beginning there was machine code, and it was cumbersome (see Figure 3.19). Quickly, early developers defined an abstraction of the machine’s native language called **assembly language,** which allowed the developer to use mnemonics to remember the machine instructions (such as RET) instead of remembering the hexadecimal code to which it corresponded (such as 0xC9). As a result, developers could be more productive, but the main impediment to developing complex functionality was the need for the developer to mentally cast the problem from the real world (e.g., “compute missile trajectory”) into the corresponding language of the machine, which consists of instructions to store or retrieve a byte of memory, increment an 8-bit value, add two 8-bit values, jump to a memory location, and so on. The conceptual difference between these worlds is immense, and connecting all the dots was a huge conceptual burden on the developer. This limited the complexity of the functionality that could be reasonably developed.

Figure 3.19 *Evolution of software development*



In the 1970s, along with bell-bottom jeans, came the invention of source-level languages. Because a source-level language was machine-independent, as long as a compiler existed to generate the correct machine instructions for the target machine, applications could be written that were largely machine-independent. Even more important, though, was the ability to move up the conceptual food chain into the domain of computer science. Rather than developing applications using the extremely limited concepts of the actual machine, developers could now develop them using the language of computer science. This includes concepts such as a variable, function, function parameters and return values, stacks, and heaps. This was a huge advance in the capability of the programmer because the conceptual difference between the problem domain and the computer science domain was vastly smaller than that between the problem domain and the machine. There were initial concerns about the efficiency of those newfangled “compilers,” and there was a great deal of debate about which was better—assembly or source code. Over time (several years, in fact), source code won out because the use of source code enabled developers to create more functionality in less time with fewer defects. This advantage won out over pure execution efficiency (the main argument of the assembly language camp). Also over time, compilers became better at producing more efficient code, and now it is relatively difficult for hand-written assembly code to outperform the output of a good optimizing compiler.

Yet, while the ability of developers improved an order of magnitude with the advent of source code and compilers, the demands for capabilities in software-intensive systems grew much faster. More or less simultaneously, object-oriented development and visual modeling appeared on the scene, coming out of different camps. Object orientation promised to move the level of abstraction from computer science concepts to real-world concepts through the introduction of the **class** concept, in which a class represents a real-world or problem-domain conceptual entity, binding together both the information the element possesses along with the behavior it can perform on that data. Early OOP languages, such as Smalltalk, drew rather little attention until C++ appeared in 1985. C++ became very popular because of the low learning curve for C programmers, and it could be learned incrementally from that standpoint. Of course, it has been repeatedly said that “the worst thing about C++ is C,” meaning that while it has a lower learning curve for C programmers than does Smalltalk and is more runtime-efficient, its required compatibility hampers it. From that community, and the arising need for write-once-debug-everywhere languages for the Internet, Java was born. Nevertheless, OOP has proven itself superior to structured languages because the former allows direct representation of the problem-domain semantics, both structure and behavior.

Visual modeling arose from the need in the structured programming world to understand just what the heck they were doing. Data flow diagrams, entity relation diagrams, and structure charts provided a graphical means for showing the organization of structured programming. People used such graphical tools inexpertly, however, creating massive wall coverings of graphics. While there was certainly some benefit from visual modeling, hindsight shows us that early visual modeling failed to live up to its hype because of some key issues.

First, and probably most critical, was the inability to truly assess the quality of the graphical design without actually building the system. Developers spent at minimum several months constructing their view of how the architecture and design should look with no verification along the way that what they were doing was a good idea. Months or years later, when they actually had to implement the design, they naturally found problems. In most cases, this resulted in the abandonment of the graphical design as the implementation began to deviate more and more profoundly from the graphically stated intent.

Compounding this problem was the second primary issue: the lack of automated synchronization between models and the implementation. The fundamental point of view was that developers had to manage two independent work products—the model and the code—and their job was to connect them in their heads and manually make changes to the two to ensure their consistency. Thus, the graphical models were completely disconnected from the implementation; if you changed the implementation and wanted to keep the graphical models in sync, you had to spend significant effort to manually do so. Even project teams with the best of intent ultimately failed to maintain the two work products in sync.

Also adding to the first issue was that these graphical modeling approaches were used in traditional industrial development processes. In these traditional, testing was performed at the end of the project. This meant that other than laborious and error-prone manual review, the models could not be assessed for quality or correctness until after the implementation was already constructed; and even then, the implementation usually differed significantly from the graphical model.

The next step in the evolution of software development was to use more robust modeling languages that were based on the OOP paradigm. By this time, object-oriented approaches had proven themselves superior to structured development approaches. By the late 1980s, there were several different graphical schemas for representing object-oriented models. In the 1990s, a consolidation effort was undertaken under the auspices of the OMG to create a unified modeling language. The first released version of UML (version 1.1) surfaced in 1996. Since then, UML has undergone a number of minor revisions (limited to bug fixes) and one major release in 2005. I had the opportunity to contribute to both major releases of the UML.[14]

14. Yes, timing diagrams in UML are my fault; I originally defined them in *Real-Time UML, First Edition* in 1998 and finally wrote them into the UML in 2003.

UML is important for a number of reasons. For one thing, it allows OOP developers[15] to represent their software functionality, structure, and behavior using a standard language. Previously there had been an incompatible set of modeling languages, and UML unified them, really allowing OOP to gain the benefits of visual modeling.

15. I want to point out that while UML is object-oriented, it can be, and *is*, used successfully for structured projects as well, and it provides all the benefits of an executable visual modeling language in either case.

At least as important was the firm semantic basis for the UML. The UML has a well-specified four-tier metamodel defining the language, although not how the language is used.[16] The result of this is that UML is inherently *executable*. At the lowest level of behavioral primitives, actions are specified in an action language (such as C, C++, or Java). These behavioral primitives are organized and orchestrated into larger-scale behavioral entities such as activities (on activity diagrams) and states (on state machines). Regardless of how the internal metamodel structures the “hairy underbelly” of the UML, the benefit to developers is that they can model some portion of their system and then test it via execution immediately. Only the higher-end tools support behavioral model execution in UML, but the benefits are substantial, particularly in that now graphical languages can be effectively used in an agile way.

16. UML is a language and not a methodology; as such, UML does not contain process guidance. That's up to books like this ☺.

Because of the well-defined semantics of the UML, a number of tool vendors have solved the problem of keeping the models and the code in sync. It is now common to forward-generate source code from a well-written UML model. It is also possible, with tools such as Rhapsody, to create a model from source code; this is known as reverse engineering. However, it should be noted that reverse engineering can be disappointing because it actually shows developers clearly, perhaps for the first time, what the structure of the software actually is. In addition, it is not only possible, but highly recommended, that as the graphical model or the source code changes, the tools automatically update the work products to keep them in sync, a process known as **round-tripping.** I firmly believe that rather than having two fundamentally independent artifacts that you must manually keep in sync, you should have one artifact (the model) contained within a single repository; the different diagrams are merely different views of elements of that repository. Class diagrams show the structural aspects; activity and state diagrams show the behavioral aspects. *Source code is nothing more or less than a detailed structural point of view of the contents of the repository.* It is *not* the job of the developer to run back and forth between the model and the code and manually synchronize them. The views should be dynamically linked to the repository content, so that if a change is made in one view, it changes the content of the repository and all relevant views change *automatically*. This is such an important concept that I coined the term **dynamic model-code associativity** (DMCA) for it in 1996.[17]

17. Kind of "rolls off the tongue," doesn't it?

The thing to note about Figure 3.19 is that there is a direct correlation between the alignment of the degree of abstraction with the problem space and productivity. The closer the developer moves to "programming in the language of the real world," the more productive that developer is. Tools have also significantly reduced the workload of the developers—a darned good thing, too, since the functionality of delivered systems continues to grow exponentially. Early on, assemblers eliminated the need for the developer to memorize the machine operation codes, and assembly language debuggers raised the level of abstraction for testing and debugging. Source code compilers and debuggers performed a similar level of productivity enhancement in the seventies and eighties. Now, UML model compilers furnish the same benefits for the developer at the graphical model level of abstraction.

The end result of all this is that many of the customers to which I consult report 30% or more improvement in their productivity and efficiency. This varies from 0% benefit up to 80% benefit based on a wide set of factors, but the most important is *how they use the models and modeling languages*. Not all processes are equally productive.[18] With an agile, constant model-execution approach, most customers report highly significant productivity improvements.

18. I guess that's a major point of this book.

3.1.12. Optimizing the Right Things

In the Harmony/ESW process, analysis is focused on getting the functionality right, whereas design concentrates on achieving various qualities of service (i.e., optimization). Problems arise because developers either optimize too early or because they spend too much effort optimizing the wrong things.

Many designers make the mistake of paying too close attention to optimization patterns and schemes and not enough to getting the right functionality. This results in highly efficient but incorrect designs. The Harmony/ESW process addresses this issue by constructing an analysis (PIM) model in the analysis phase of the microcycle and then performing optimization. In the case that some or another design solution must be chosen in order to get the analysis model executing, the simplest possible design solution is preferentially selected. This ensures that at least when optimization does take place, the basic functionality is correct.

Many developers optimize their software without truly understanding what really needs to be optimized or how best to optimize it. The important thing to remember about optimization is that it is always a trade-off—when you optimize some qualities of service, you inevitably deoptimize some other aspect of the system.

Prior to optimization, it is important to understand why you are optimizing. "To make it better, duh!" is not really a very satisfying answer if you take into account that you cannot simultaneously optimize all aspects of a system. You may be optimizing because the system has very tight timing, schedulability, or memory constraints. You may be optimizing because you have a short time-to-market need or because you need to maximize reuse. You may be optimizing because you're building a safety-critical, high-reliability, or high-security system. Each of these aspects of the system can be thought of as a design criterion—that is, it is one of the criteria by which the "goodness" of the design will be judged. The previous chapter includes a list of common design optimization criteria in the section on the PSM.

In a given system, these design criteria will be more or less important; we call the relative importance of a design criterion its **weight.** The weight may be normalized (so that the sum of the weights equals some value) or not, but the basic measure of a good design is that it maximizes the sum of the product of the degree of optimization of the design aspect and its weight. Mathematically, this is shown in the following equation:

$$optimalDesign = \max\left[\sum DegreeOptimized_i \times Weight_i\right]$$

The degree optimized for each design aspect changes with respect to the design pattern, idiom, or technology employed. For example, precalculating computation results may save significant time if those results are referenced more frequently than they are computed, but at the cost of lowering the degree of optimization of memory. The overall improvement of using the design idiom is a function of the relative frequency of reading the value versus its computation, the amount of time required to perform the read versus the amount of time required for the computation, and the relative importance of runtime speed versus runtime memory usage.

I certainly recognize the criticality of optimal designs, having been involved in real-time system development for more decades than I care to recount. But because of that experience, I recognize that optimization is effective only if it optimizes the right things. As discussed in the previous chapter, this optimization is done at three different levels of abstraction: architectural, mechanistic, and detailed (see Figure 2.6). This topic will be discussed in some detail in Chapter 7, “Agile Design.”

3.2. Harmony Core Practices

As a result of the core principles identified in the previous section, Harmony/ESW provides a set of core practices that are realized by the process roles while performing process tasks, resulting in process work products. In this context, a practice is a standard way of working to perform tasks. As such, these practices contribute to many tasks in the Harmony/ESW workflows. They will be discussed more thoroughly in the coming chapters, but I will discuss them briefly here.

The Harmony core practices include the following:

• Incrementally construct

• Use dynamic planning

• Minimize overall complexity

• Model with a purpose

• Use frameworks

• Prove the system under development is correct—continually

• Create software and tests at the same time

• Apply patterns intelligently

• Manage interfaces to ease integration

• Use model-code associativity

These core practices are explained in the following sections.

3.2.1. Incrementally Construct

Incremental construction is a key practice in the Harmony/ESW process. In the small nanocycle scale, this construction take place every few minutes in the same fashion as the security example in Figure 3.3. With powerful model-based tools, executing your model really is as simple (and almost as fast) as pushing the Generate/Make/Run button. Experience has shown that this is—by far—the most effective way to model software. The traditional approach of modeling for a few months and then trying to get the system to compile for a few weeks or months isn't effective despite its predominance.

At the larger scale, incremental construction appears at the microcycle scale. This brings together different architectural units (subsystems or components) as well as different engineering disciplines (e.g., software, electronic, mechanical, and chemical) into an integrated version of the product that provides some level of functionality. This artifact is known either as the **increment** or, more commonly, the **prototype.** To be clear, what Harmony/ESW means by the term *prototype* is "an integrated, validated version of the system that may not be complete." It contains *real code* that will be shipped in the product, just not always *all* of the code. Some people refer to throwaway versions of the system constructed to prove or test a concept as a prototype, but that is not what is meant in the Harmony/ESW process. There is a place for such things, but when necessary, the process always refers to such work products as **throwaway prototypes.** For example, I've created UIs for medical and airborne systems using Visual Basic to present the interface to users for feedback. The UI prototype is used to gather requirements and get feedback on the UI concept and is then discarded. That is an example of a valid use for a throwaway prototype. In this book, and in the Harmony/ESW process, whenever the term *prototype* is used alone, it always refers to a version of the real system.

A single microcycle takes the existing prototype and adds functionality to it. This functionality is specified by a small number of use cases. A single microcycle usually requires between four and six weeks to construct and results in an incremental prototype and, possibly, a list of minor defects to be fixed in downstream microcycles. Some practitioners like to have shorter cycles, sometimes as short as a week, while others might go for three to four months—but the main point is that the time taken to produce a prototype is an order of magnitude smaller than the time for the overall project.

The work done within a microcycle is defined by the **prototype mission** (a work product to be discussed in more detail in Chapter 6, "Agile Analysis"). In short, the prototype mission summarizes the goals of the work to be done during the microcycle. The primary goal is to identify:

- The set of requirements (bound to use cases) to be realized and validated

- The set of risks to be reduced

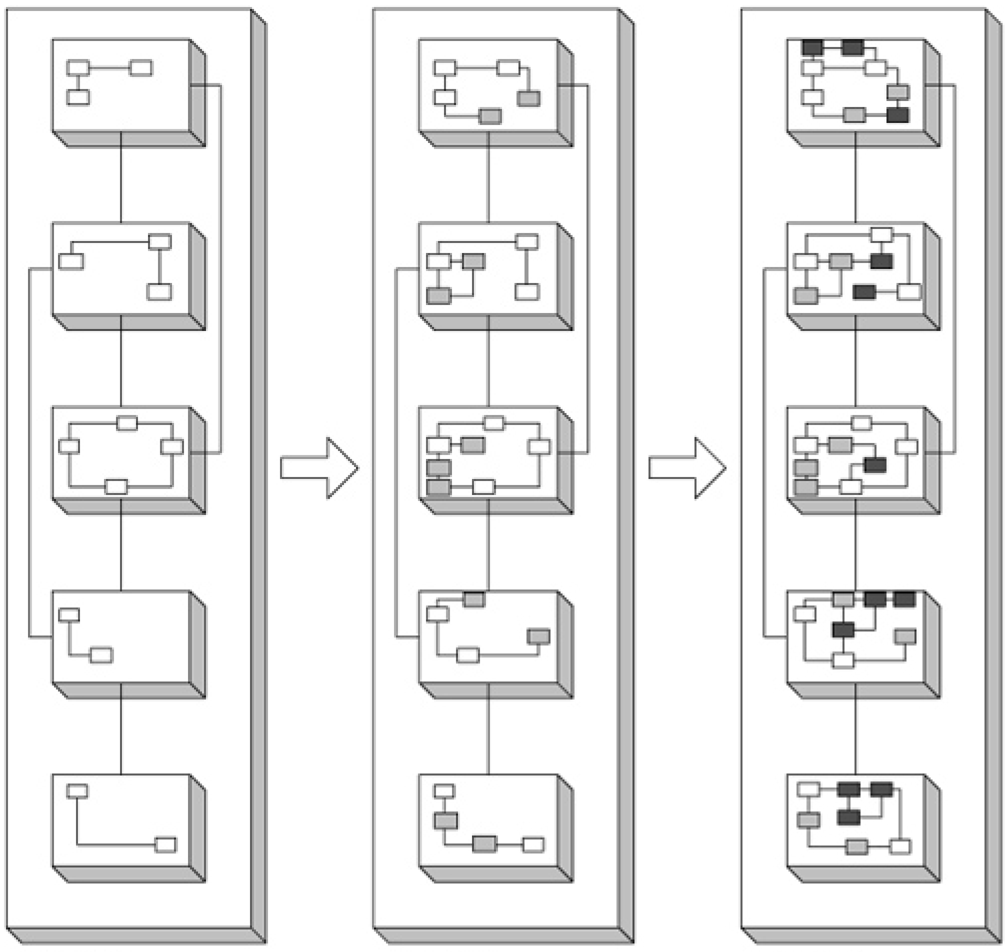
- The set of defects to be repaired

- The architectural intent of the prototype

- The target platform(s) on which the prototype will be validated

The prototype gains functionality and completeness over successive microcycle iterations. At the end of each microcycle, regression testing is performed to ensure that previously validated functionality hasn't been broken by the additions, as well as validation testing of the new capabilities of the system. An example is shown schematically in Figure 3.20. Basically, the same elements are retained as the new functionality is added. In practice, there is a small amount of reorganization of existing elements that takes place during the development of the next prototype; this is known as **refactoring.**

Figure 3.20 *Prototype evolution*



As I have mentioned, it is not just software that is integrated in the prototype; the work products of other engineering disciplines are added together as well, as specified in the prototype mission. Hardware may be at various stages of maturity in different prototypes. For example, early prototypes may be run on laptops with software simulations of sensors and actuators. Later prototypes may use some hand-breadboarded circuits, and still later prototypes may have wire-wrapped or factory-produced boards. It is similar for the other disciplines, such as mechanical engineering. In one system I was involved with, the first prototype had painted cardboard enclosures. Over time, the mechanical systems matured to include hand-manufactured mechanicals with real materials and, finally, first-run factory-produced mechanical parts. Each engineering discipline is focused on producing its contribution to the next prototype, based on the prototype mission.

3.2.2. Use Dynamic Planning

As I discussed in Chapter 1, "Introduction to Agile and Real-Time Concepts," agile methods differ from traditional methods in a number of ways, but one of the most important is that agile methods embrace the concept of dynamic planning. The basic idea of **dynamic planning** is to iterate the sequence “Plan, track, adapt” frequently. As the project progresses, you learn more about the nature of your project, the size of the features, the effort involved, and when you can expect to be done. These are things that you typically have some idea about before you begin but about which you lack perfect, precise knowledge.[19] Certainly, it helps to have a plan that reflects reality early on, but even if the initial plan isn’t very accurate, properly applied dynamic planning will rapidly converge on reality.

19. That’s why they call it *estimating*.

Doubt it? Consider the Newton-Raphson iteration, a numerical analysis method for solving differentiable equations.[20] In this analysis technique, the idea is to solve numerically for an equation of the form

$$f(x) = 0$$

20. Apology to math-phobes: I have a personality flaw; I was trained as a scientist and a math geek so I think about things in terms of equations. I go to support groups but the best I can say is that I’m a “recovering mathaholic.”

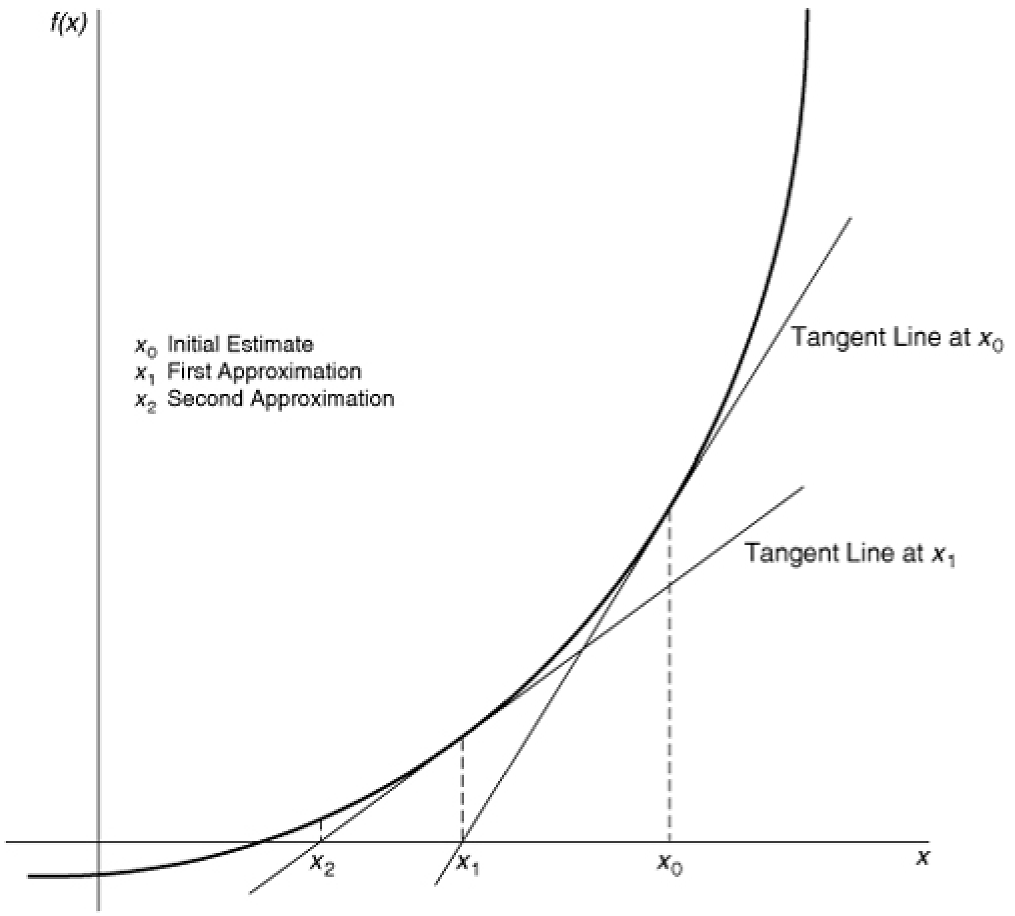
when a closed-form analytic solution isn’t feasible.[21] The basic approach is to make an initial guess (x_0). If we are anywhere in the neighborhood of the solution where $f(x)$ is actually zero, then $f'(x)$ should be a good approximation of $f(x)$, where $f'(x)$ is the derivative of $f(x)$. The line tangent to the curve $f(x)$ at the point x_0, $T(x)$, is simply

$$T(x) = f'(x_0)(x - x_0) + f(x_0)$$

21. That’s math for “I dunno what the answer is, dude!”

as you can see in Figure 3.21. The value of x at which the $T(x)$ crosses the x axis (i.e., $T(x) = 0$) is our next estimate of the answer (x_1). This is called the *zero* of $T(x)$. This can be easily calculated as

Figure 3.21 *Newton-Raphson iteration*



$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The Newton-Raphson iteration then repeats the process until we are sufficiently close to the value; that is, $f(x_n)$ is sufficiently close to zero. In practice, the Newton-Raphson iteration usually converges very rapidly, requiring only a few iterations to get very close to the solution.

Dynamic planning is the scheduling equivalent of the Newton-Raphson iteration. We make our best guess as to the answer, we experimentally determine how close we are and how fast we're getting there, and we replan based on that result. The key for us is the termination of $f'(x)$, the first derivative of the product completion function. This is done through the notion of **velocity.**

Velocity is simply the rate of progress on project completion. To use the notion of velocity, we need a quantitative measure of the work elements to be done, so that as we complete them we can determine where we are (with respect to project completion) and how fast we're approaching our goal. A common agile approach is to use **story points**[22] or **use case points.**[23] It is important to understand that these are estimation techniques and are therefore inaccurate (up to 40% variance is common). However, the point of dynamic planning is that accuracy isn't required—the tracking aspect of dynamic planning will compute the error, and the replanning can take into account this variation.

22. Mike Cohn, *Agile Estimation and Planning* (Upper Saddle River, NJ: Prentice Hall, 2006).

23. Based on the work of Gustav Karner and elaborated in Geri Schneider and Jason P. Winters, *Applying Use Cases: A Practical Guide* (Reading, MA: Addison-Wesley, 1998).

Even more important than absolute accuracy in the effort or time estimates is the consistency with which they are applied; that is, if you have applied points to estimate the time to completion for various work items (e.g., product features), it is not so important that the absolute value of the points be correct, but it *is* important that the relative assignment of points to different work items be specific. It doesn't matter if a point corresponds to two days of work, but it does matter that if one work item has a value of eight points, it takes about twice as long to complete as a work item assigned four points.

This is important because approaches such as COCOMO tend to fail because (1) they are applied only once for an initial estimate of the total project time or effort and (2) the actual "size" of the points varies greatly because of factors (such as business environment overhead) that are either not or poorly accounted for. As Cohn says, "Velocity is the great equalizer." As long as the points relate to a consistent time frame (which is usually the case), the actual value of that time frame is of secondary importance.

The practice of dynamic planning is governed by a set of (sub)practices:

- Planning by goals (features), not implementation (activities)
- Estimating relative size (e.g., use case points)
- Computing relative accomplishment with velocity
- Scheduling work items of an appropriate size
- Prioritizing your work items list
- Understanding the uncertainty
- Replanning often

Let's consider each point.

Planning by Goals, Not Implementation

Planning by implementation in software development is the most common approach. That doesn't, however, make it a good idea. For example, it is common to plan for a certain number of lines of code per day. Instead, I recommend that the project maintain a prioritized **work items list** that contains the information about the backlog of work items that must be addressed. Work items can refer to new features to be added, risk reduction activities, defects to be repaired, and so on. The details of the work items list will be discussed in Chapter 5.

Estimating Relative Size (e.g., Use Case Points)

Different companies, even different project teams within the same company, will quantify the time necessary to perform a work item differently. As mentioned above, the important thing is the consistency of the mapping from points to time. This consistency can be achieved by using the same estimator or by achieving consensus among the team members as to the point values for a set of work items.

Computing Relative Accomplishment with Velocity

Velocity is computed from the work items completed within a period of time. The Harmony/ESW process computes velocity primarily at the microcycle points and compares it with the expected velocity (as identified by the project schedule). Velocity is important because it represents "truth on the ground," a measure of actual progress as opposed to the ideal progress usually represented in schedules. The closer the plan and the actuality are, the more accurately the effort and completion dates of the project can be known.

Scheduling Work Items of an Appropriate Size

Ideally, work items are in the range of half a day to two days. One of the reasons that this is the ideal is that estimators will be far more accurate at this timescale than for work items expected to require, say, six months. The other is that because the work items are completed rapidly, the project gains velocity information much sooner than with larger-scale work tasks. This does mean that larger-scale work efforts will often require decomposition. So, rather than "Support the Blue Fox multimode RADAR," work items might be much smaller, such as:

• Support basic transfer on the 1553 bus.

• Support 1553 bus schedules.

• Support commanded built-in test.

• Support basic command (*x*).

• Support sector blanking.

• Visualize RADAR plot.

• Display RADAR target data.

And so on.

Prioritizing Your Work Items List

Not all work items are equally important. The priority of a work item reflects either its urgency or its criticality. In either case, it is almost always preferable to work on higher-priority work items at the expense of lower-priority ones. Although this rule shouldn't be slavishly followed —for example, you may want to complete a feature by performing a short lower-priority work item rather than begin a longer but unrelated higher-priority work item—it remains a good rule of thumb.

Understanding the Uncertainty

If you admit that schedules are always an exercise in estimating stuff that you don't know, then by extension you admit there is uncertainty in your schedule. As discussed in Chapter 1, knowledge improves as the project progresses,[24] so that subsequent plans will be more accurate than earlier ones. Uncertainty can be accounted for in several ways. One way is to allow the end date to vary. With this approach, the computed end date has an associated expected range of variation that diminishes as the end date approaches. Alternatively, you can fix the end date but vary the delivered functionality. This is often captured by identifying features as either required or "upside" (to be delivered if there is adequate time). Another approach is to vary the effort applied to the work, either as a percentage of available effort or by adding resources. What you cannot do—although many managers repeatedly try—is to fix all three aspects simultaneously.

24. Provided, of course, that you actually *look*.

Replanning Often

Because you learn from project execution, you can use that increasing knowledge to make increasingly accurate plans. This is known as **replanning.** In the Harmony/ESW process, this replanning occurs at a specific point in the microcycle, although intermediate replanning can be done more frequently if desired. The microcycle contains a short activity known as the increment review or, more popularly, the "party phase." During this phase, one of the things reviewed is the project schedule. A typical microcycle is in the range of four to six weeks but may be as short as two weeks or as long as four months. During this schedule review the team looks at the project accomplishments against what was planned, computes the project velocity, and replans to take actual project progress into account.

3.2.3. Minimize Overall Complexity

As long as requirements are met, simpler is better. Simpler requires less effort to create. Simpler is easier to get right. Simpler is more understandable. Simpler is more maintainable. Of course, having said that, we spend a lot of time developing systems that have quite complex functionality. One of the things we want from our designs is that they be as simple as possible, but no simpler than that.

As a general rule, we can have complexity in the large (architecture or collaboration level) or in the small (class, function, or data structure level). In some cases, we improve overall simplicity by making individual elements slightly more complex because it greatly simplifies the interaction of those elements. Or, in other cases, we improve overall simplicity by simplifying some very complex elements but at a cost of increasing the number of elements and their relations in the collaboration.

Complexity comes in two kinds. First, there is **inherent complexity.** This is the complexity that results from the fundamental nature of the feature being developed. Without changing the feature, you are stuck with the inherent complexity. The second kind is **incidental complexity.** This is the complexity resulting from the idioms, patterns, and technologies chosen to realize the feature. This latter kind of complexity is under your control as a developer. You can trade off complexity in the small for complexity in the large in your effort to minimize the overall complexity.

For example, consider the collaboration shown in Figure 3.22. It contains a class `Vaporizer` for delivering anesthetic drugs and has the complex state machine shown in Figure 3.23. Note that the structural diagram is simple because the elements within it are complex.

Figure 3.22 *Sample collaboration*

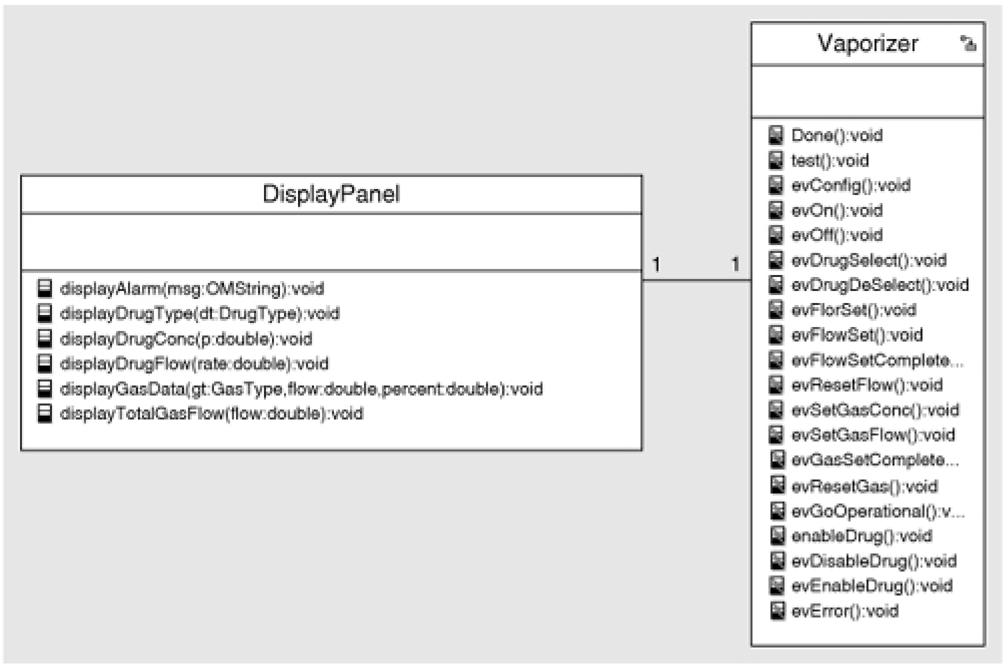
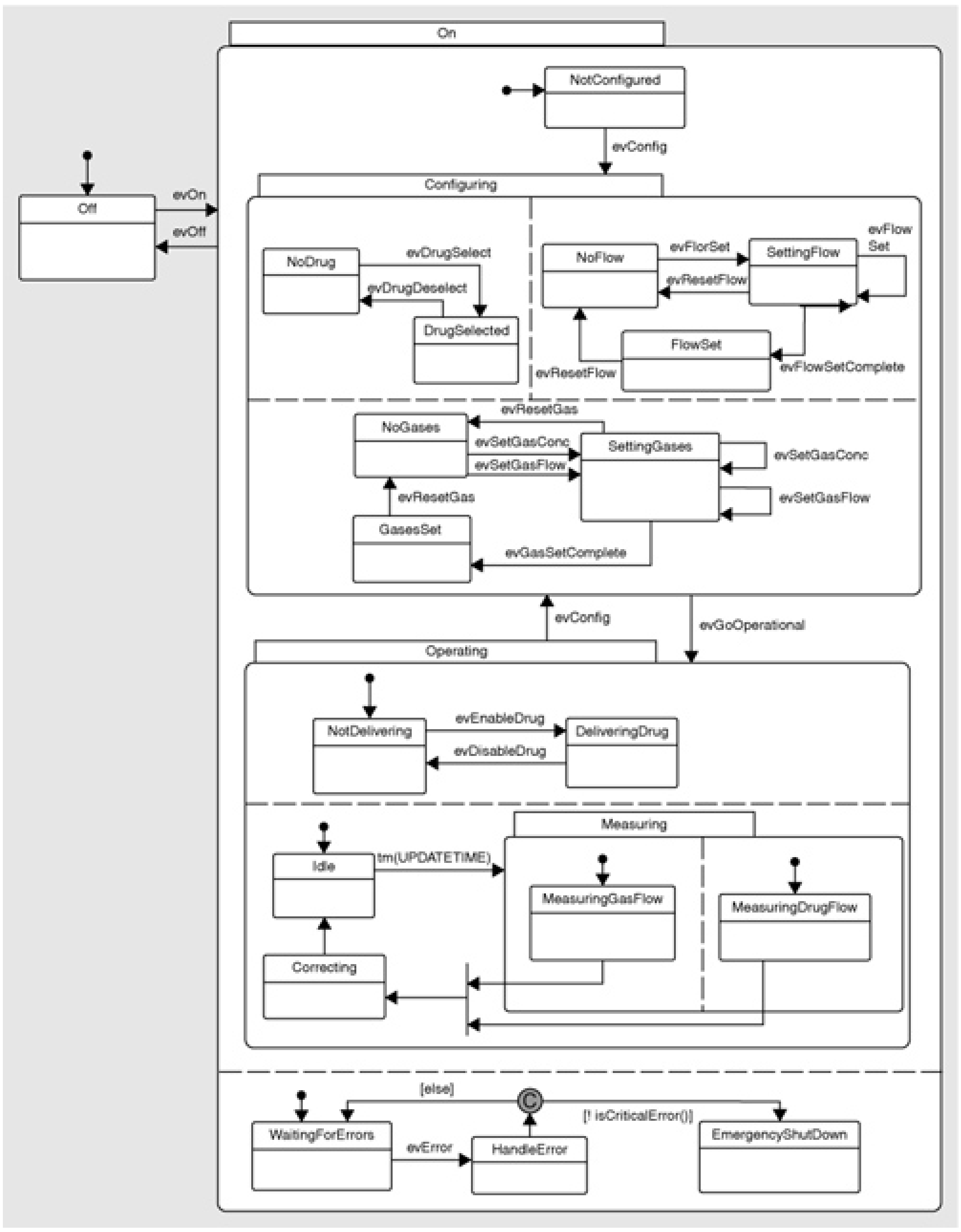
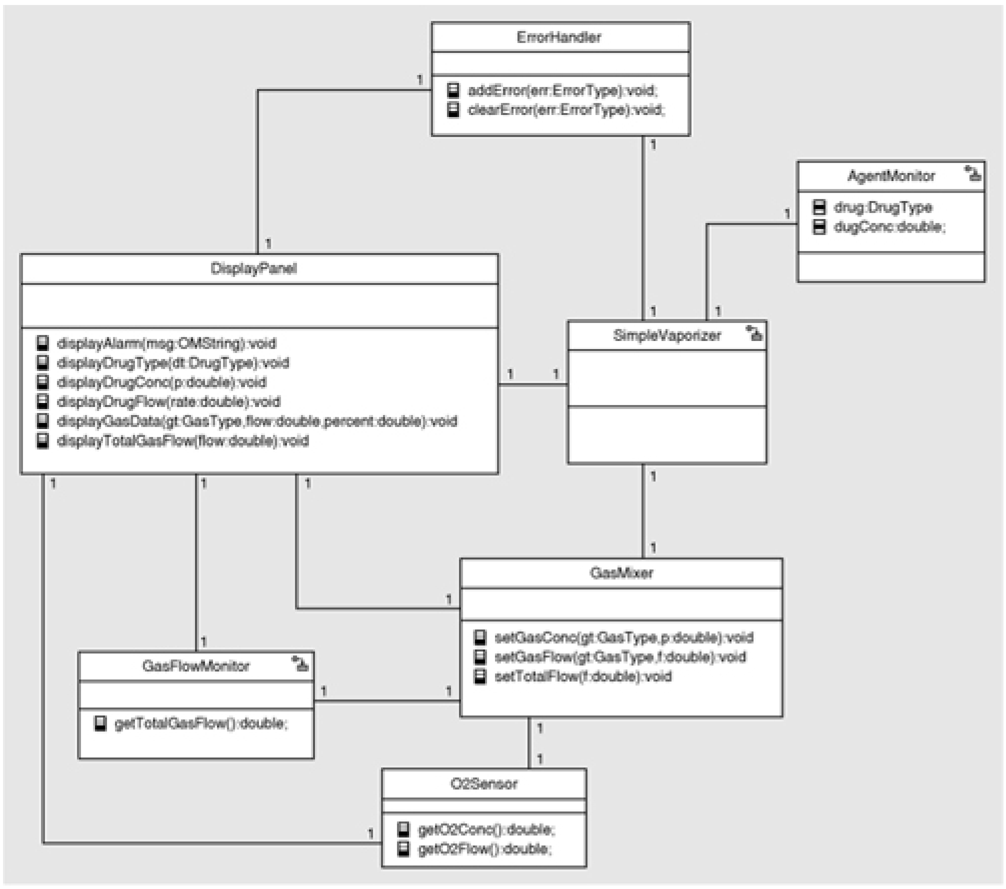


Figure 3.23 *Class* `Vaporizer` *state machine*



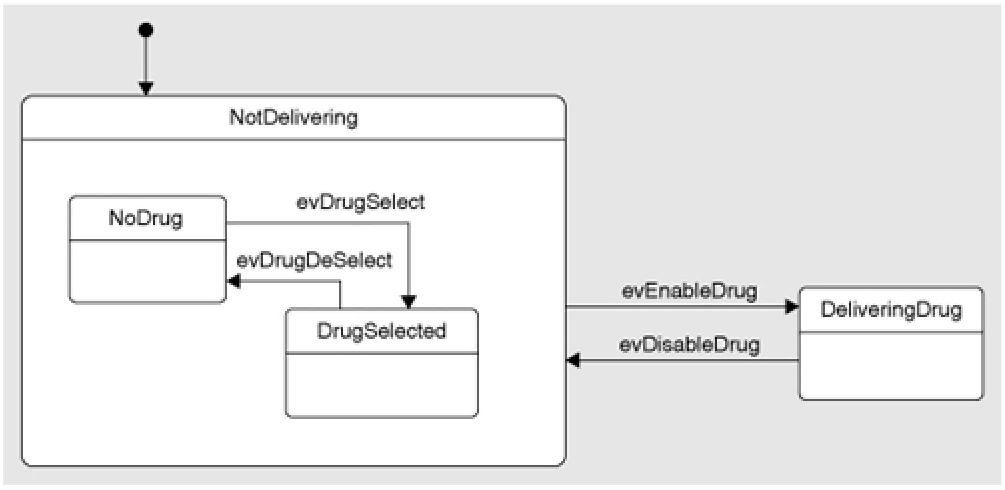
We can simplify the `Vaporizer` class at the expense of complicating the collaboration by extracting either composite or and-states within the class and making them separate classes. Figure 3.24 shows the resulting more complex collaboration. Note the icon in the upper right-hand corner of various classes; this indicates that the class has a state machine. Portions of the state machine from the original `Vaporizer` class now populate these various classes. Each class in the collaboration has a narrower focus than the previous `Vaporizer` class, but the overall collaboration is more complex. Figure 3.25 shows the much simpler state machine for the `SimplifiedVaporizer` class. The other state machines are of a similar complexity.

Figure 3.24 *Elaborated collaboration*



ErrorHandler
addError(err:ErrorType):void;
clearError(err:ErrorType):void;
AgentMonitor
drug:DrugType
dugConc:double;
DisplayPanel
displayAlarm(msg:OMString):void
displayDrugType(dt:DrugType):void
displayDrugConc(p:double):void
displayDrugFlow(rate:double):void
displayGasData(gt:GasType,flow:double,percent:double):void
displayTotalGasFlow(flow:double):void
SimpleVaporizer
GasMixer
setGasConc(gt:GasType,p:double):void
setGasFlow(gt:GasType,f:double):void
setTotalFlow(f:double):void
GasFlowMonitor
getTotalGasFlow():double;
O2Sensor
getO2Conc():double;
getO2Flow():double;
1

Figure 3.25 *State machine for class* `SimplifiedVaporizer`



3.2.4. Model with a Purpose

A model is always a simplification of the thing in the real world that it represents. As George Box says, “All models are wrong but some are useful.”[25] For a model to be useful, it must represent the aspects of the elements relevant to the purpose of the model. In addition, we need to represent aspects of that model in views (diagrams) that present that information in a fashion that is understandable and usable for our purposes.

25. George Box, “Capability Maturity Model Integration (CMMI®) Version 1.2 Overview” (2007), at www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview07.pdf.

One of the time-honored (and fairly useless) idioms for creating diagrams is to invoke the 7 ± 2 rule. The idea is adapted from neurolinguistics—we know that human short-term memory can hold 7 ± 2 things. In the seventies, someone misinterpreted this to mean that data flow diagrams should have no more than seven things on them, and if there are more than seven things, the right thing to do is to create a hierarchy.

This is, of course, stupid[26] and has created countless examples of extremely hard-to-follow diagram structures. The Harmony/ESW process recommends an entirely different practice:

26. Not that I have an opinion or anything! ☺

Each diagram should have a single key principle or aspect it is trying to represent; the diagram will contain all elements contributing to that principle or aspect but no more. This mission should be explicitly stated on the diagram unless obvious.

The mission for some diagrams—such as state charts—is obvious (specify the behavior of a `Classifier`) but for the other diagrams it is not. Some common missions for class diagrams include the following:

• *Show the concurrency architecture*—what tasks are in the system, what resources they share, and how they interact.

• *Show the distribution architecture*—how objects are split across address spaces, how they communicate and collaborate.

• *Show the deployment architecture*—how objects map to hardware.

• *Show the safety and reliability architecture*—how redundancy is managed to make the system robust in the presence of faults.

• *Show the subsystem architecture*—the large-scale pieces of the system and how they interact.

• *Show the interfaces supported and required* by a subsystem.

• *Show a class taxonomy*—the generalization taxonomy of a related set of classes.

• *Show a collaboration of roles*—how a set of classes interact to realize a system-level capability.

• *Show the structure of a composite class*—the parts within the structured class.

• *Show the instances and links* at a specific point in time.

Whenever you create a diagram, you should do so with an explicit purpose. This mission is usually placed in a comment. I generally place that comment in the upper-left or upper-right corner of the diagram. For example, Figure 3.26 shows the subsystem architecture for an anesthesia machine. Figure 3.27 shows a sequence diagram from the same model that depicts a scenario from one of the use cases. The mission appears in the comment that names and describes the scenario, along with the pre- and postconditions. Figure 3.28 shows a UML package diagram showing the organization of the model. To recap, each diagram should have an explicit purpose, and that purpose should be clearly stated on the diagram.

Figure 3.26 *Subsystem diagram with mission statement*

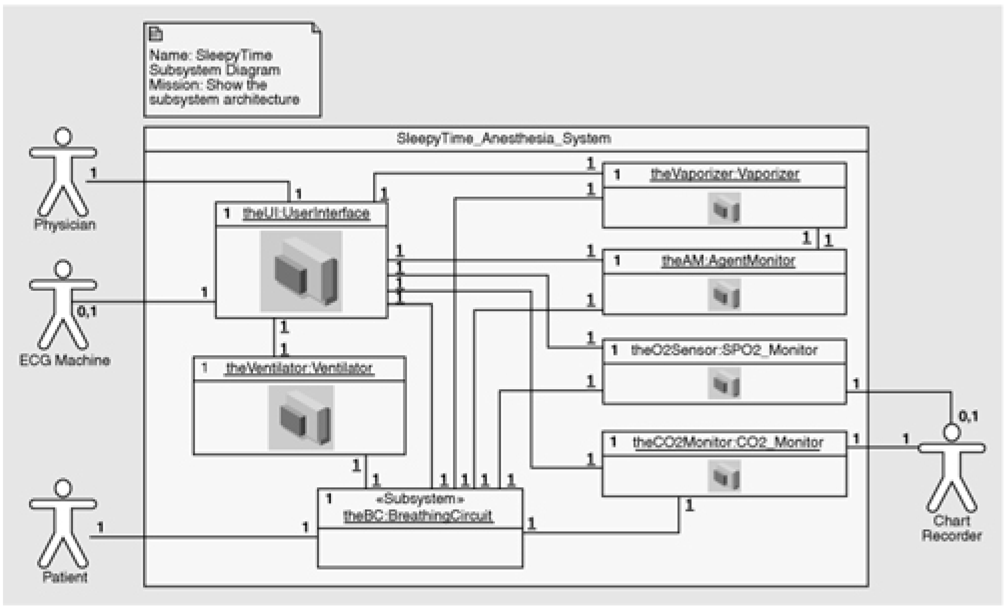


Figure 3.27 *Sequence diagram with mission statement*

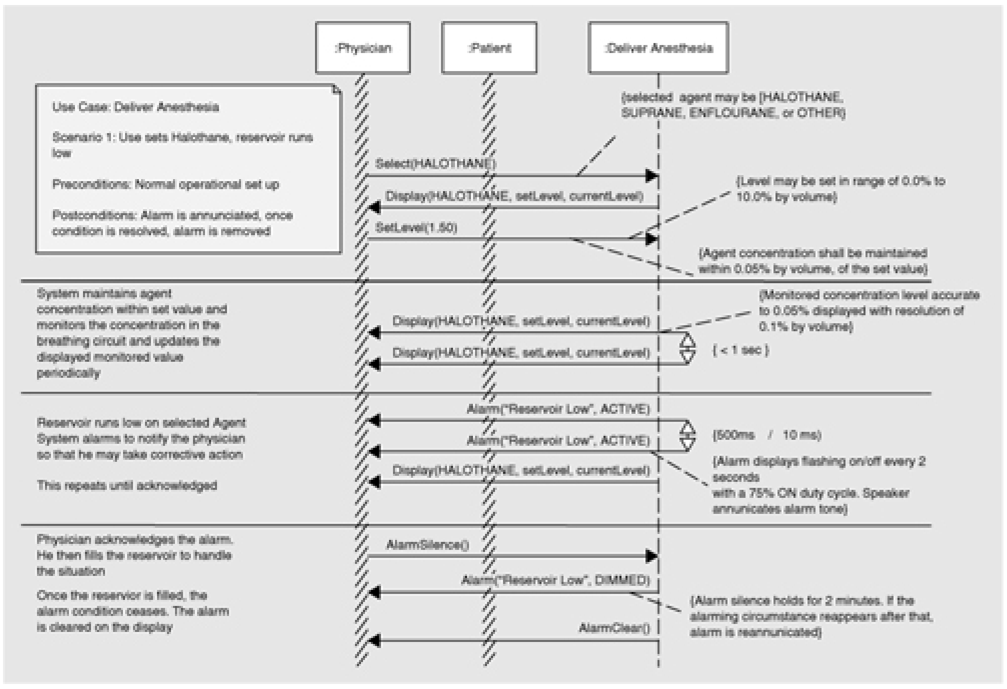
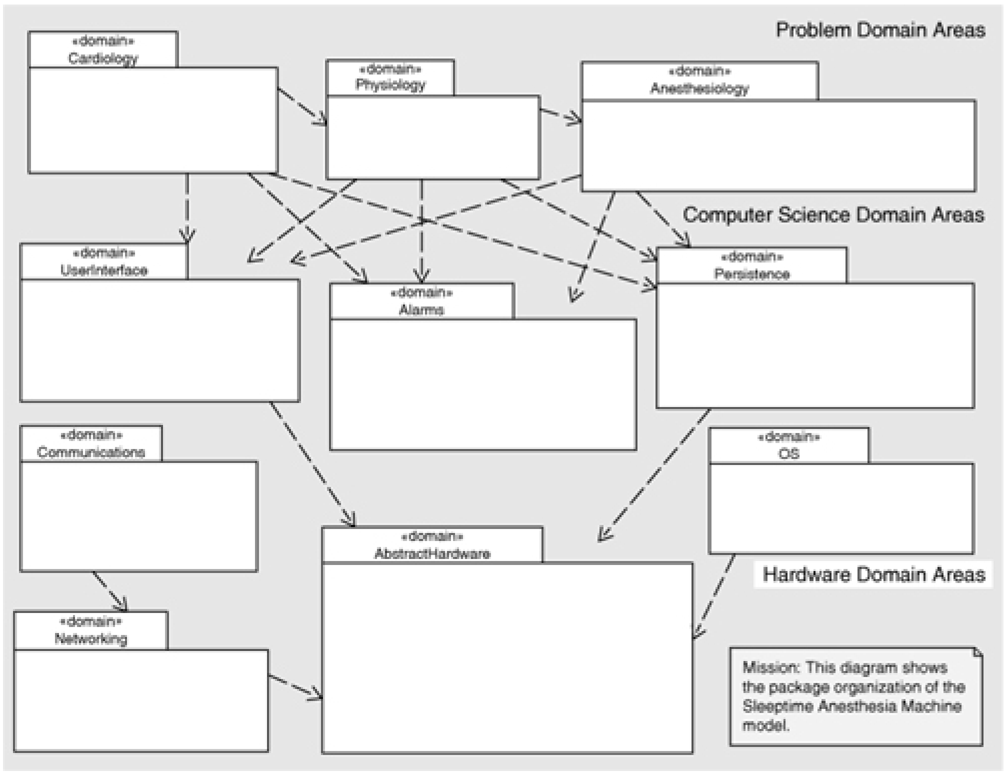

:Physician
:Patient
:Deliver Anesthesia
Use Case: Deliver Anesthesia
Scenario 1: Use sets Halothane, reservoir runs low
Preconditions: Normal operational set up
Postconditions: Alarm is annunciated, once condition is resolved, alarm is removed
{selected agent may be [HALOTHANE, SUPRANE, ENFLOURANE, or OTHER}
Select(HALOTHANE)
Display(HALOTHANE, setLevel, currentLevel)
{Level may be set in range of 0.0% to 10.0% by volume}
SetLevel(1.50)
{Agent concentration shall be maintained within 0.05% by volume, of the set value}
System maintains agent concentration within set value and monitors the concentration in the breathing circuit and updates the displayed monitored value periodically
{Monitored concentration level accurate to 0.05% displayed with resolution of 0.1% by volume}
Display(HALOTHANE, setLevel, currentLevel)
Display(HALOTHANE, setLevel, currentLevel)
{ < 1 sec }
Reservoir runs low on selected Agent System alarms to notify the physician so that he may take corrective action
This repeats until acknowledged
Alarm("Reservoir Low", ACTIVE)
Alarm("Reservoir Low", ACTIVE)
Display(HALOTHANE, setLevel, currentLevel)
{500ms / 10 ms}
{Alarm displays flashing on/off every 2 seconds with a 75% ON duty cycle. Speaker annunicates alarm tone}
Physician acknowledges the alarm. He then fills the reservoir to handle the situation
Once the reservoir is filled, the alarm condition ceases. The alarm is cleared on the display
AlarmSilence()
Alarm("Reservoir Low", DIMMED)
{Alarm silence holds for 2 minutes. If the alarming circumstance reappears after that, alarm is reannunicated}
AlarmClear()

Figure 3.28 *Package diagram with mission statement*



The Harmony/ESW process identifies the missions as candidates (there are, of course, many more). However, in practice, almost every model will have one or more diagrams for each of the five views of architecture discussed earlier in the chapter, plus at least one class diagram per use case showing the collaboration of elements realizing that use case.

3.2.5. Use Frameworks

There are several available definitions for frameworks. The best definition that I've seen is that:

a framework is a partially completed application, pieces of which are customized by the user to complete the application.[27]

27. Gregory F. Rogers, *Framework-Based Software Development in C++* (Upper Saddle River, NJ: Prentice Hall, 1997).

Frameworks provide a number of significant interrelated advantages for the quick development of robust applications:

• They provide a set of general ways to accomplish common programming tasks. This minimizes the complexity of the system because there are fewer idioms to learn in order to understand the application structure and behavior.

• They provide service classes that perform many of the common housekeeping chores that make up most of all applications. This frees the developer to concentrate on domain-specific issues and problems.

• They provide a means of large-scale reuse for applications.

• They provide a common architectural infrastructure for applications.

• They can greatly reduce the time to market for building brand-new applications.

• They can be specialized for domains so that they can capture domain-specific idioms and patterns.

One area in which frameworks have shown their value is in the development of Windows applications. Originally, Windows applications were handcrafted C applications filled with complex and obscure Windows API calls to do everything imaginable. Windows applications were built totally from scratch each time. The development experience for Windows programming was that it was slow, painful, and error-prone—and the quality of the resulting Windows applications reflected this truth. Frameworks like the Microsoft Foundation Classes (MFC) and Borland's Object Windows Library (OWL) changed all that. Now, writing Windows applications with drop-down and pop-up menus, bitmaps, multiple document windows, and even database access, TCP/IP sockets, animation, and Object Linking and Embedding (OLE) interfaces is almost as easy as writing DOS applications.

The tremendous decrease in perceived development effort is the result of a single primary factor: the use of frameworks. The use of frameworks also enabled a secondary technology, that of component-based development, to flourish by providing a common infrastructure in which to plug.

The same benefits are applicable in the world of real-time embedded systems development. For the most part, real-time embedded systems development is now like Windows programming was a decade ago: slow, error-prone, and painful. However, through the development and use of frameworks that provide a common infrastructure, the development of real-time systems can be greatly improved. These improvements will manifest themselves in improved quality and functionality and decreased development time.

3.2.6. Prove the System under Development Is Correct—Continually

This practice works hand in hand with the first practice, incremental construction. This key practice states that the system under development should never be more than minutes away from execution (and thereby being demonstrated to be correct). This is done even with early, very incomplete models. This practice applies as soon as we begin realizing requirements in the phase of the microcycle called object analysis, in which we identify the essential classes, types, and their behavior and relations. It continues through architecture definition and optimization, collaboration-level optimization, and through detailed design optimizations. We execute and exercise our software after each small incremental change to make sure that the previous functionality wasn't broken, and that the new functionality we've added works as we expect.

The earlier `Security` class example in this chapter (refer to Figure 3.3 through Figure 3.17) illustrates this point. Remember: The best way not to have defects in your system is not to put defects in your system. The very best way to achieve this is to execute your evolving software beginning in the first hour of the very first day that you start laying down classes, data, or functions.

This practice is a good idea even in the absence of model execution tools in which the models are translated manually into equivalent code. However, model-based execution speeds up and enhances this effort significantly. Imagine if you had to design in C and then switch over to vi and write the equivalent assembly code. Isn't it better to just execute the C code directly? The same is true with the UML. The purpose of this book is not to sell or even promote tools (not even powerful modeling tools), but if you have them, they can certainly make your life easier!

3.2.7. Create Software and Tests at the Same Time

The previous practice emphasized that software should be created in small incremental steps with continual validation that the software, so far, does the right thing. We recommend that you use model-based execution several times per day, but you can do it with hand-written source code as well. One of the keys of continual executions is that you're running the software to prove not that it compiles, but that it is correct. This means that you must have some concept about what correct *is*.

While traditional approaches emphasize testing at the end just prior to system release, this has proven to be the second-most-expensive approach to testing.[28] When defects remain in the software, it becomes more difficult to identify them. Worse, it becomes much more expensive to remove them, because the longer a defect is present in your system, the more time the software has to build up dependencies on the flaw, resulting in more places in your software that need to be repaired. It's just better and easier to not have defects in the software in the first place.

28. The most expensive is to test *after* you ship to the customer.

To achieve this goal, the Harmony/ESW process recommends the practice of developing the tests immediately prior to, or in conjunction with, the software under test. That way, tests are not an afterthought—"Oh, yeah, I guess we should test, eh?"—but something that is done as an integral part of developing the software in the first place.

Representing Tests

The UML Testing Profile represents tests primarily as sequence or activity diagrams. Additionally, test cases can be represented in code or other model elements. The Harmony/ESW process recommends creating `«testBuddy»` classes. These are classes that exist solely to test classes that are or will be contained in the shipped system, but are not themselves shipped with the system. They are configuration-managed along with the shippable model elements and normally reside in the same package in the model or within a package nested therein. Refer to Figure 3.3 for an example of a `«testBuddy»` class. There are also model-based testing tools, such as Test Conductor from IBM Rational, that can automatically apply tests defined as sequence diagrams, activity diagrams, flowcharts, state diagrams, or code segments.

There are many different kinds of tests that are useful. Not all may be applicable at a given time. For example, performance tests are rarely performed on analysis models because they are not yet optimized. Some of the kinds of tests identified in the Harmony/ESW process are:

• *Functional*—tests the behavior or functionality of a system or system element

• *QoS*—tests the “performance” of a system or system element, often to measure its performance against its performance requirements. Such tests measure the time for an individual action or for a large number of actions in sequence or in parallel. For example, if a device driver is planned to handle 1000 messages/second, then stress testing might consist of testing sustained throughputs of 900 messages/second and 1000 messages/second as well as measuring the time for the processing of a single message.

• *Range*—tests values within a data range

• *Statistical*—tests values within a range by selecting them stochastically from a probability density function (PDF)

• *Boundary*—tests values just at the edges of, just inside, and just outside a range

• *Coverage*—tests that all execution paths are executed during a test suite. This can be done at the model level (looking for execution of all transitions in a state machine or all control flows in an activity diagram), at the source code level (looking for all calls, if/then/else, switch/case, and try/catch clauses), or at the assembly language level.

• *Stress*—tests data that exceeds the expected bandwidth of a system or system element. Stress testing tries to break the system or element by giving it bandwidth range out of its design specification to ensure the system can gracefully handle overload situations.

• *Volume,* also known as “load testing”—tests the system with large amounts of data that meet or exceed its design load

• *Fault seeding*—tests whether the system properly handles a fault intentionally introduced to the system

• *Regression*—normally a subset of previously passed tests; tests that a modification to a system did not introduce errors into previously correctly functioning systems

Early testing focuses primarily on functional testing. Later testing will include boundary, stress, volume, and regression testing. It is important to test the software with boundary and out-of-range values and conditions to ensure the system is robust. It is important to select tests that are appropriate for the level of maturity and completeness. Certainly, by the time the product is shipped, all those kinds of tests must be run and passed.

3.2.8. Apply Patterns Intelligently

Optimization is important, particularly for real-time and embedded systems. However, unguided and undirected optimization is as likely to have poor results as good optimization. This is because whenever you optimize one aspect of a system, you necessarily deoptimize some other aspect. Therefore, it is critical that the aspects of the design that improve it for its intended purpose be optimized. The Harmony/ESW process achieves this goal by applying design optimizations with a specific workflow, namely:

1. Identify the design (optimization) criteria.

2. Rank the design criteria in order of criticality to the system success.

3. Identify design patterns, idioms, or technologies that optimize the most important design criteria at the expense of the least important.

4. Apply the design patterns, idioms, or technologies.

5. Validate the design solution to

a. Ensure that previously working functionality wasn't broken during the optimization process

b. Ensure that the optimization goals have been achieved

In Harmony/ESW, most design work is done through the application of design patterns. Design patterns, as mentioned in Chapter 2, "Concepts, Goals, and Benefits of Model-Driven Development," are reusable design solutions for recurring problems. Some design patterns are added by elaboration—manually modifying the model to include the pattern elements—while others are added by using a model-based translator. Both approaches work fairly well. The main thing is to understand what you are trying to optimize, understand the pros and cons of your optimization approach, and then apply the patterns correctly.

3.2.9. Manage Interfaces to Ease Integration

For larger systems, integration of elements has long been a common point of failure. This has been true of different systems that are supposed to talk to each other, of architectural units of a single system, and even of engineering disciplines within an architectural subsystem. The two basic practices in Harmony/ESW are:

• All stakeholders of an interface—especially both providers and consumers—use the same definition of the interface.

• Continuously integrate elements that share an interface so that any inconsistency in interface use can be detected and fixed as early as possible.

The first of these may seem obvious, but it is not at all uncommon for different stakeholders to use copies of the actual interface, and that these copies are separately maintained. Eventually, these copies fall out of sync with each other and chaos ensues later during integration.

The two primary ways that Harmony/ESW realizes the practice of using the same interface is managing architectural interfaces of a system as a resource to be shared among the stakeholders and, for multidisciplinary systems, creating class diagrams that capture the interface as a shared resource.

A UML model is often represented as a set of shared models. In fact, I typically recommend that no more than 7 to 10 engineers share a single model. This means that all large projects are represented in a set of interrelated models rather than a single monolithic repository. This practice is a reasonable compromise between cohesion within a model and interoperability of cohesive units. Models within a project can interact in a couple of different ways: "by value" or "by reference." The former means that one model loads a copy of the other but is free to modify it without changing the original. The latter means that one model loads another in a read-only form; in order for the second model to be modified, it must be edited directly. In this case, the client model is automatically updated when the server model is changed.

It is in this latter form that models share the interfaces and common data types in the shared model. By loading the shared model by reference, they are ensured of working against the actual interface.

These interfaces are specified early in the architecture and then "frozen" in CM. This allows the teams to progress against a known and specified interface. The problem is, of course, that the interfaces are likely to be wrong in some detail or another. However, if, when a defect is discovered in the interface, the stakeholders get together and renegotiate that interface and then freeze it again, the problem is minimized. There may be a small amount of rework necessary when an interface is revised, but experience has shown that this is a tiny amount of effort compared to the more common practice of letting the interfaces "float" until system integration.

The other interface issue is the interaction of different engineering disciplines. For the readers of this book, the primary concern is the software-electronics interface. As discussed in Chapter 2, I don't recommend the use of UML deployment diagrams for this purpose because they are “dumbed-down” class diagrams. Full class diagrams used for this purpose, such as Figure 2.26, are much richer and more expressive. When the interface details, such as:

• Type (e.g., memory-mapped, port-mapped, or interrupt-mapped)

• Location (memory address, port, or interrupt number)

• Size (e.g., number of bits)

• Data representation and valid data range

• Other preconditional invariants

are captured, the interface becomes well specified for both the software and the electrical engineer. This diagram and the associated metadata form a contract for the engineering disciplines involved.

Besides managing the interfaces per se, Harmony/ESW eliminates (or at least mitigates) integration problems by pursuing a practice of **continuous integration.** Continuous integration is achieved by having all developers submit their software daily or more often. Then at least once per day, the configuration items (CIs) are integrated and tested to ensure that someone’s changes haven’t broken the build. The key is to develop in small units called work items that can be completed in a day or even less.

The developer workflow for continuous integration is to:

1. Implement the work item

2. Validate the work item in the developer context

3. Update the developer context with the baseline

4. Validate the work item in the context of the baseline

5. Submit the update to the configuration manager

The configuration manager then builds and tests the system at least once per day. His or her workflow is to:

1. Create a build from updated elements from (possibly multiple) developers

2. Run an integration test suite

3. If the system passes the tests, create a new baseline

4. Make the new baseline available to the developers

In parallel, the configuration manager also creates new integration tests over time as new functionality is delivered in the evolving baseline.

3.2.10. Use Model-Code Associativity

In the 1980s, the point of view that modeling tools took was that there were two fundamentally unrelated artifacts: the model and the code. And it was the developer's job to traverse the two and manually keep them in sync. Experience has shown that this never happens over the long term; eventually the models and the code begin to deviate, and at some point the models will be thrown away.

Harmony/ESW takes a different viewpoint. It views the model as a cohesive set of interrelated metadata that is exposed to humans through a variety of views. Class diagrams show the structure of model elements. Use cases show functionality. Sequence, timing, and communication diagrams show interactions among model elements. State machines and activity diagrams show behavior. Code is nothing more or less than a detailed structural view of the system.

Each of these views is *dynamically linked* to the metadata repository. The dynamic nature of the link means that if the data in that repository is changed because the developer is working in a particular view (e.g., class, state machine, or even code), then all the views that represent the changed information are automatically updated. Thus, when I work in the class or state diagram, the code changes. Automatically. When I work in the code, the class and/or state diagrams change. Again, automatically.

For the most part, the Harmony/ESW process practices forward engineering, that is, automatically creating the code from the analysis and design models. Sometimes you will find it useful to modify the code and you want those code changes to be reflected in the model. Automatically.

This is important. For models to be truly useful, they *must* reflect the functionality, structure, and behavior of the actual system. And we know that you won't update the models manually to reflect changes made in the code. Oh, you'll want to. You may even believe that you will. But we all know that you won't.[29] With a tool that implements dynamic model-code associativity, models and the code will remain in sync.

29. You never call, either, even though I wait by the phone . . .

3.3. Coming Up

This chapter introduced the forces driving the content and guidance of the Harmony/ESW agile process. The process includes a set of core principles from which the practices discussed in this chapter are derived. The most crucial goal is the first: The primary goal of a software developer is to develop software. Several of the remaining goals are consequences of this. These goals include paying continual attention to quality, getting continuous feedback, optimizing the right things, and using the amount of working functionality as your primary measure of progress.

Other goals center around a core concept of the use of abstraction. To support this, there are goals about the use of abstraction and modeling. This is especially true for architecture, the strategic optimization decisions that structure your system. The Harmony/ESW process identifies five key views of architecture and a set of secondary architectural views that augment them.

The third set of goals focuses on the fidelity of project information. The fact remains that software development schedules and plans are always exercises in estimating things that you don't actually know. The Harmony/ESW process emphasizes using the information you have but planning to track progress and revising the plans frequently as you gain more insight into the project. In addition to *planning to replan*, the Harmony/ESW process notes that the leading cause of project failure is ignoring risk. Therefore, the process has recommended practices for identifying and removing risk during the project.

The Harmony/ESW process incorporates a number of practices to realize these goals. Harmony/ESW is a model-driven, architecture-centric process that emphasizes the creation of semantically correct models from which code can be directly generated by humans or by model compiler. This generation takes place several times per day, and the code so developed is then executed informally—a process known as **debugging**—and formally—a process known as **unit testing.** A key practice is the incremental construction of the system, making small incremental changes and validating that the system is defect-free before moving on. The use of design patterns emphasizes the reuse of proven solutions for optimization of the system structure, behavior, or QoS.

The next chapter discusses the process per se. The focus will be on the workflows of the process, emphasizing the incremental microcycle spiral. Each of the phases of the microcycle is then discussed in more detail with the roles, tasks, and work products identified. Subsequent chapters will drill down into the nitty-gritty of how Harmony/ESW performs project initiation (Chapter 5), analysis (Chapter 6), design (Chapter 7), test (Chapter 8), and process optimization (Chapter 9).

Chapter 4
Process Overview

The Harmony/ESW process is the result of decades of work, by myself and others. I've tried very hard to incorporate new ideas that help and cut out deadweight when ideas didn't work out. Previously, the process was known as the Rapid Object-Oriented Process for Embedded Systems (ROPES), later Harmony, and now is a key member of the Harmony process family. As mentioned in previous chapters, Harmony/ESW focuses on the development of certain kinds of systems, specifically software-intensive real-time and embedded systems that are often highly focused on efficiency, cost, safety, and reliability.

The detailed tasks, workflows, roles, work products, and guidance will be provided in later chapters. This chapter will serve as an introduction to the basic ideas behind those details.

4.1. Why Process at All?

Why indeed?

Software and systems are intensely social activities, despite the lack of social skills found in many of its practitioners. This is because most systems are complex, complicated, and consist of tens of thousands (on the small side) to multiple millions of lines of code. Highly skilled technical experts must collaborate to end up with a product that performs as it should and when it should. That turns out to be much more difficult than you might otherwise suppose.

For a tiny system—a few thousand lines of code—a single talented engineer can construct the logic and data necessary. In this case, the quality, correctness, and robustness of the system are in direct proportion to the skill of the engineer. Process isn't needed, because the highly skilled engineer is able to focus his or her efforts toward a fruitful end. Such developers have an internalized process that works for them, or they wouldn't be known as "highly skilled." However, these internalized processes can be extremely idiosyncratic and unique to the individual.

Most systems are far too complex to be efficiently constructed by a single engineer. For a team of engineers to pool their efforts, they need to understand how to divide the work, work toward a common goal, and merge the fruits of their labors efficiently into the final product.

They must agree on how to partition the work so that each piece will contribute to the common goal as well as properly interact with the other pieces, and do so in a predictable time frame.

Process is the means by which correlated interaction among the engineers takes place. Harmony/ESW defines process as “a set of coherent workflows constructed of tasks performed by worker roles resulting in desired work products.” A **methodology** consists of a primary language in which the semantics of the system are specified and a process that guides the developers in the use of that language.

There are, of course, many ways to divide and structure the work, but not all approaches work equally well. As discussed in Chapter 1, “Introduction to Agile and Real-Time Concepts,” most current processes are based in factory automation, with the basic assumptions about infinite scalability and infinite predictability built in. The waterfall process is the epitome of the industrial process point of view:

1. Define the requirements for the system to be constructed.

2. Analyze those requirements.

3. Design the system.

4. Test the system to find any minor flaws that have been introduced.

5. Ship the system.

The basic assumptions of the industrial process model are that each phase in the process can be completed without making any essential mistakes, and that any minor mistakes made are linear in nature, with easily identified, localized, and point-repairable faults. We know from decades of software development in many different domains of expertise that this is clearly not the case. Software is chaotic and highly nonlinear. It is not infinitely scalable; systems 10 times the size are easily 100 times harder to develop. It is not highly predictable, not only because the basic “stuff” of software is complexity captured as a set of rules written in formal languages, but also because the world for which we develop software is very dynamic. It is difficult to produce software ahead of changes in the environment that invalidate our assumptions or starting conditions.

Software development is far more invention than it is manufacturing. This has led to the development of incremental development approaches in which systems are constructed in smaller, easier-to-develop pieces with increasing functionality and fidelity. Barry Boehm’s[1] spiral approach has been around for quite a number of years, but most projects are still developed with a linear industrial manufacturing process perspective.

1. See Barry Boehm, *Software Engineering Economics* (Englewood Cliffs, NJ: Prentice Hall, 1981).

A process is an interrelated set of concepts. The keys ones are:

• *Role*—the "hat" a person wears while having a specific scope of responsibility

• *Task*—an element of work with a predefined set of inputs and outputs, which may be

° Work products

° Conditions

° Results

• *Workflow*—a defined sequence of tasks

• *Work product*—an artifact produced by a role while performing a task

These concepts are summarized in Figure 4.1. In this illustration you can see some secondary concepts as well, such as

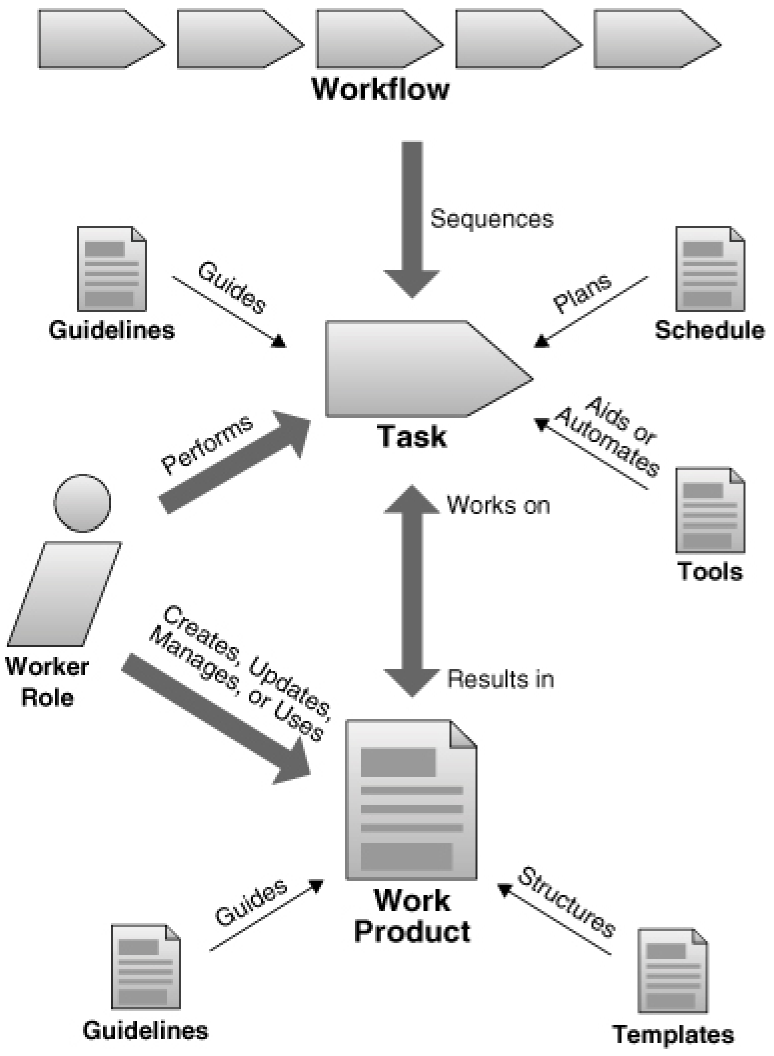
• *Schedule*—identifies the date, time, and effort for tasks and allocates the tasks to specific people

• *Tools*—either aid or automate the execution of a task

• *Guidelines*—provide help in how a work product should be created or managed or a task should be executed

• *Templates—provide predefined structures for work products*

Figure 4.1 *Core process concepts*



There are, of course, many more elements of a process. We will get into these as we progress.

Process contains a series of tasks and guidance on how to perform those tasks. But what differentiates a good process from a bad one? Harmony/ESW has a clear standard: *A good process is one that allows you to develop more functionality, in less time, at a lower cost, and with fewer defects.*

4.1.1. Harmony/ESW Process Introduction

The Harmony/ESW process is unique in the software development industry in that it is:

• Agile

• Model-based

• High-quality

- Architecture-centric
- Requirements-driven
- Focused on QoS
- Safety- and reliability-directed
- Optimized for real-time and embedded systems

By *agile* we mean simultaneously lightweight in its overhead and dynamic in its planning and responsiveness. In this context, *lightweight* refers to focusing our effort on the tasks that provide substantial benefit to the goal of product development with the minimum ceremony necessary to meet the project needs. *Dynamic* means that we actively seek feedback as to our actual effectiveness and modify our plans to improve that effectiveness. These topics were discussed in some detail in Chapter 1.

As discussed in some detail in Chapter 2, "Concepts, Goals, and Benefits of Model-Driven Development," Harmony/ESW is *model-based*. Models are the primary work product(s) produced. Models provide a vastly more powerful means of capturing the system semantics because we can succinctly represent different aspects of the system semantics, such as functionality, structure, behavior, and performance (more precisely known by the more encompassing term *quality of service* [QoS]). In addition, we can visualize that representation at different levels of abstraction from system context to subsystem architecture, all the way down to primitive (nondecomposable) elements such as simple classes, functions, and data. Despite the variety of viewpoints that we can bring to bear, we can also ensure consistency among the elements by using a (semi)rigorous language, such as the UML, and tools to manage the model within a repository. MDA, or the more generic MDD, recommends construction of a specific set of models that have proven their worth in enhancing correctness, reuse, portability, and performance. These are, of course, the CIM, PIM, PSM, and PSI.

High quality is achieved through the continuous gathering of quality data, primarily through the execution of partially complete models.[2] A key practice of agile methods is to never be more than minutes away from being able to demonstrate the correctness of the system so far.[3] Harmony/ESW implements this practice primarily through the use of model-based execution. This has many advantages. For example, as stated in Chapter 1, the best way not to have defects in the system is not to put those defects there in the first place. Early feedback means that we identify defects immediately (or darn close to it) and remove them when it is easy and inexpensive to do so. Second, model-based execution allows us to visualize the functionality using the same technology that we use to design it—analogous to using a C debugger to visualize the behavior of code rather than figuring it out from debugging the generated assembly code. The quality aspect is especially enhanced when coupled with automatic code generation, because with a high-quality model compiler, the code always reflects the design. If you modify the code directly, good modeling tools provide automated means to incorporate those changes back into the model.[4]

2. See the Harmony/ESW principle "Continuous attention to quality is essential" and the key practice "Create software and tests at the same time" in Chapter 3.

3. See the Harmony/ESW principle "Continuous feedback is crucial" and the practice "Prove the system under development is correct—continually," ibid.

4. See the Harmony/ESW practice "Use model-code associativity," ibid.

You can think about *architecture* as "the set of strategic design decisions that affect most or all of the system." Harmony/ESW is *architecture-centric* and focuses enormous attention on the five key architectural views.[5] Experience has shown that these architectural aspects—the subsystem and component architecture, the concurrency and resource management architecture, the distribution architecture, the safety and reliability architecture, and the deployment architecture—have a profound impact on the performance, scalability, robustness, and maintainability of the final system. There are secondary architectural views, such as security, data management, dynamic QoS management, and SOA (service-oriented architecture). These make a significant impact in some systems, but generally not as universal as the impact of the five key views. The Harmony/ESW design process, of which architecture is an important part, is oriented around the selection and application of technology and design patterns. Harmony/ESW provides strong guidance on how best to identify appropriate technology and design patterns to create robust, scalable architectures.

5. See the Harmony/ESW principle "Five key views of architecture define your architecture," ibid.

Requirements are important because they define the goal and the end point of the development effort. Requirements serve as the arbiter of correctness of the work and the delivered system. Harmony/ESW is *requirements-driven* in that the identification of elements in the analysis and design models is guided continuously by meeting the requirements. As we will see, this is done by creating the CIM to represent the requirements in an inherently validatable, verifiable form. This aids not only in getting the requirements correctly captured but also in guiding the day-to-day and minute-to-minute work of the developer. The CIM is organized around use cases that depict coherent uses of the system. The use cases are detailed, as we will see in Chapter 5, "Project Initiation," with informal natural language statements, formal behavioral specifications with state machines, and scenarios capturing example actor-system interactions. The PIM (analysis model) and PSM (design model) are correct to the extent that they can reproduce those semantics.

On thing that sets the Harmony/ESW process apart from other processes is the attention paid to and focus on QoS.[6] Of course, Harmony/ESW focuses on real-time and embedded systems, and QoS is the primary distinguishing characteristic of such systems. Harmony/ESW is *QoS-focused* and provides strong guidance on how best to capture and use QoS metadata. Some of the different kinds of qualities of service managed by the Harmony/ESW process include

6. See the Harmony/ESW practice “Apply patterns intelligently,” ibid.

• Execution time

° Worst case

° Average case

° Read time

° Write time

° Access time

° Slack time

° Blocking time

° Deadlines

° Utility functions

• Event recurrence

° Period

° Jitter

° Interarrival time

° Burst length

• Schedulability

• Predictability

• Memory usage

• Robustness

• Reliability

• Safety

• Risk

Not all real-time and embedded systems are safety-critical or high-reliability. Another aspect of the Harmony/ESW process that sets it apart is the degree to which it is *safety- and reliability-directed*. The process provides guidance on how to analyze and model safety and reliability designs. Safety-critical and high-reliability products must be able to identify, isolate, and correct faults during system execution. This requires special technologies and design patterns[7] that use redundancy in different ways to achieve the design goals of the system. Projects that don't have these aspects can omit the additional related tasks, work products, and redundant aspects that otherwise must be present. Analytic techniques include FTA, FMEA, and FMECA. As we will see in Chapter 6, "Agile Analysis," the process includes guidance in how to perform these analyses and incorporate the results in the design.

7. See my *Real-Time Design Patterns*.

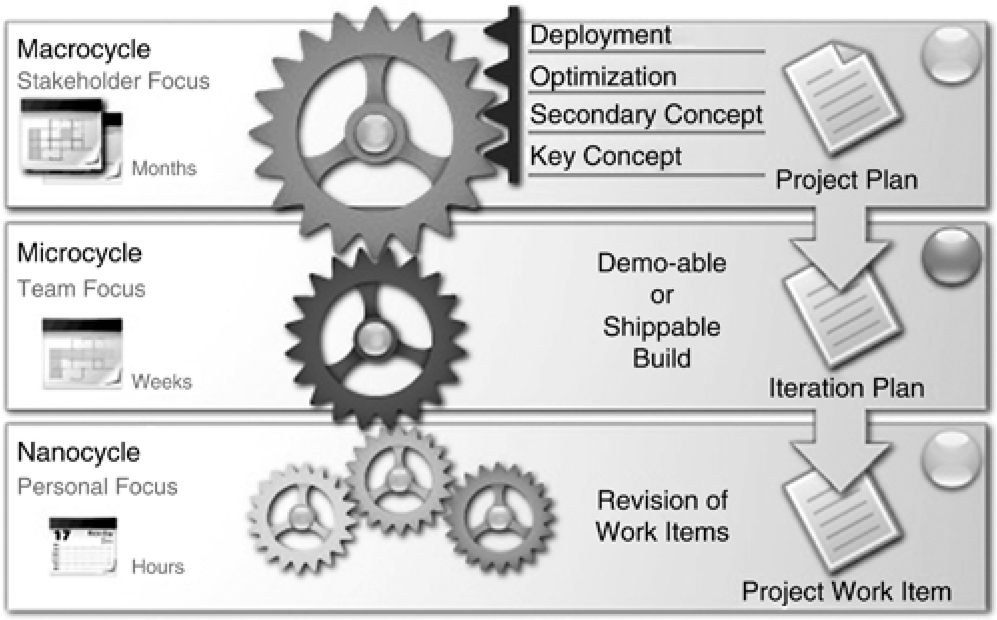
The Harmony/ESW process, as a software and systems development process, applies to all kinds of software. Nevertheless, it is *optimized* for the development of real-time and embedded systems. There is a special focus on the concerns of such systems, especially modeling and managing processing efficiency, qualities of service, device-driver-level components, embedded architectures, safety, and reliability.

A process is ultimately a set of roles performing sequenced tasks within a set of workflows that result in a set of work products. The guidance provided by the process applies at different timescales, in the same sense that managing your career is done differently if the concern is what you need to accomplish today, versus what you need to accomplish for the project, versus your five-year goals. The next section discusses the three levels of timescale of importance to the Harmony/ESW process.

4.2. Harmony Time Frames

The Harmony/ESW process views project time at three timescales (see Figure 4.2). Why do we care? We care because different project planning and development work takes place at each of these timescales.

Figure 4.2 *Harmony/ESW time frames*



The **macrocycle** is the largest-scale focus of the process and has the entire life of the project squarely in its sights. This point of view identifies key project milestones, planned future system enhancements, and customer release schedules. Typical macrocycle scope ranges from about nine months on the short side to several years on the longer side, usually a couple of years on average. In one military project I was involved with, the macrocycle scope extended for 25 years.

The macrocycle is loosely divided into four overlapping macrophases. Each macrophase contains a number of microcycles, each of which produces a validated version of the system under development. The macrophases are characterized by their primary focus of concern. The fact that these macrophases overlap is meant to convey the idea that while the mission statements of the microcycles are highly project-specific, their emphasis tends to change in a standard way throughout the course of the project. It is also meant to emphasize the notion that this is highly variable depending on the project characteristics.

• The first macrophase focuses on *key concepts*. These key concepts may be key product features or capabilities, key architectural aspects, key technologies, or key project risks. As mentioned, this macrophase will contain a number of microcycles, during which increasingly complete versions of the system that address, elaborate, or resolves these concerns are built. If your customer is traditionally minded and wants a PDR, this will normally take place at or around the end of the first macrophase.

• The second macrophase centers on *secondary concepts*. Once the primary concerns have been addressed, the secondary ones must be dealt with. By the end of the set of microcycles within this macrophase, all architectural concerns have been addressed, so traditional CDRs, if required, take place around the end of the second macrophase.

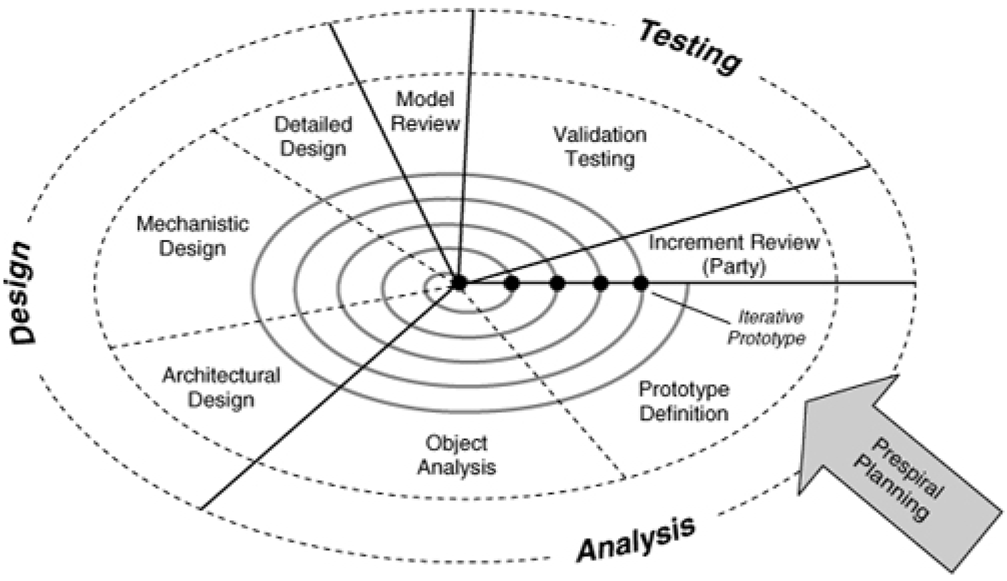
• The third macrophase concentrates on *optimization concerns*. This is not to say that optimization doesn't take place in the other macrophases, merely that this is a primary emphasis of the microcycles contained within this macrophase. That means that most analysis is complete and design is more the target of the work activities.

• The last macrocycle focuses on *deployment concerns*. These concerns deal with cross-discipline integration of software, electronics, mechanical and chemical aspects, as well as deployment in the customer environment. It is not uncommon to stage "test flights" of the system (sometimes literally) in this phase.

While the macrophases certainly have "themes" that they emphasize, each microcycle in each macrophase contains analysis, design, implementation, and test work activities.

The next timescale down is the **microcycle**. The microcycle is all about producing a single build of the system that integrates functionality and engineering disciplines into a validated version of the system. Harmony/ESW is an incremental, spiral development approach at its core, and the microcycle is just an instance of that spiral. Figure 4.3 shows the main spiral of the microcycle approach schematically. The key concept of the spiral approach is to develop the system incrementally at a frequent rate and to validate each increment. The next section will discuss the phases in the spiral in some detail, but it is useful to note that the microcycle is basically just a project-in-the-small. A microcycle normally requires four to six weeks to complete, although the range may be one week to as much as four months. In each microcycle, new requirements are elucidated, analyzed, designed, implemented, and *validated*. The increments, known as **incremental prototypes,** are not hacked-together code for demonstration to the customer or to management (although they can be used for that purpose) but contain the *real code* that will ultimately ship. The incremental prototypes, particularly the early ones, are merely incomplete. The requirements that they implement are truly implemented in a high-quality, validated way, but early prototypes don't implement all the requirements.

Figure 4.3 *Harmony spiral*



Each microcycle is organized around a **mission**—a statement of intent and goals—culminating in a validated version of the system with identified functionality. The mission statement, as we will see in Chapter 6, is organized around several concerns:

• The system capabilities (organized into use cases) to be realized

• The project risks to be reduced

• The (minor) defects from earlier prototypes to be repaired

• The architectural intent of the prototype

• The target platforms to be supported

The work to achieve this mission is represented in the list of work items scheduled to be performed during the microcycle. In agile methods, the work items list is a backlog of future and potential work to be done.

In the beginning of the project, **prespiral planning** precedes the first instance of the spiral (microcycle). In this phase the vision of the system to be ultimately produced is constructed, the development environment—such as CM, compilers, and modeling tools—is installed, resources are allocated, and the schedule is constructed. The schedule is organized primarily around the specific microcycles with validated incremental releases at the end of each microcycle. The microcycle is split into a number of activities usually performed in sequence. These are:

• *Analysis*—the specification of the essential properties of the system in the context of the microcycle. This is divided into two subphases:

° *Prototype definition*—the identification of the scope of work within the microcycle, including the requirements to be realized

° *Object analysis*—the creation of a running model of the functional properties for the prototype

• *Design*—the optimization of the object analysis model. This takes place at three levels of abstraction:

° *Architectural design*—the strategic, global optimization of the system

° *Mechanistic design*—the optimization of use case collaborations

° *Detailed design*—the optimization of primitive elements of the design (classes, functions, types, and data structures)

• *Model review*—a review of the work artifacts (models, source code, unit test results, etc.)

• *Testing*—the validation of the model and source code against the requirements of the system and the mission of the microcycle

• *Increment review* ("*party*")—the analysis of the project progress and identification of improvements in the process and environment

The **analysis** phase identifies essential elements of the system. In the first subphase —**prototype definition**—the requirements to be realized within the spiral are identified, characterized, and clustered into use cases, and the microcycle mission statement, identified earlier, is produced. The use case model corresponds to the CIM discussed in Chapter 2.

Object analysis is the second subphase. It concentrates on identifying the essential classes, objects, functions, and data necessary to realize the requirements. It is organized around realizing the set of use cases identified in the microcycle mission. This subphase has source and object code as well as the PIM as outputs, and by its end, each use case is realized by a set of unit tested (although not validated) collaborations of software elements. The emphasis on the object analysis model is realizing the functional requirements, not the QoS or performance requirements.

The next phase is **design.** In the Harmony/ESW process, design is all about optimization. Design is divided into three subphases, differentiated on the basis of scope of concern. **Architectural design** identifies and applies design decisions that optimize the system at an overall level. **Mechanistic design** optimizes individual collaborations, each of which realizes a single use case. **Detailed design** optimizes primitive classes, functions, and variables and has the smallest scope of concern. Design in the Harmony/ESW process is largely design-pattern-based and places emphasis on identifying exactly what needs to be optimized, identifying design patterns and/or technologies that achieve that optimization at an acceptable cost, applying those design decisions, and verifying the resulting model. The primary outputs are the optimized design model (also known as the PSM in MDA parlance) and the resulting source and object code.

Both the analysis and design phases produce code as an output. They also produce white-box unit-level tests that are applied throughout development. This results in a high-quality model and code base for subsequent work and possible delivery to the customer.

In parallel with object analysis and design, in the “Prepare for validation” phase the validation test suite is created and/or updated, the test plans and strategies are detailed, and any test fixtures (e.g., simulators) needed during the validation test that takes place at the end of the microcycle are created.

Following the design phase, there is frequently a **model review**. Model reviews are expensive and potentially very time-consuming, so many agile authors recommend that they not be performed. They do, however, provide value, particularly for large projects or systems that have high reliability and/or safety concerns. To make this review as efficient as possible, we defer the review until after design (and all associated unit tests) so that the review can focus on key functionality and not on obvious implementation errors. Sometimes a model review will also be done between object analysis and design; this can add value as well, especially in large projects.

Validation is the next phase after the design phase and any subsequent review. Validation formally tests the system to ensure that it meets the requirements of the system from a black-box viewpoint. Validation consists of reapplying at least a subset of previous validation tests (i.e., regression tests) and applying the test vectors that are derived from the requirements and use cases realized in the current prototype.

In parallel with the primary phases, the “Control project” activity manages the project by performing quality assurance activities, managing safety and reliability issues (if relevant to the project), and updating the risk management plan, schedule, and process, as appropriate.

The last activity in the microcycle is the **increment review** (also known as the **party phase**). The party phase is one of the primary points in which the Harmony/ESW process improves itself based on feedback gathered during the spiral.

In the Harmony/ESW process, the project proceeds incrementally. What that means is that a validated version of the system is produced during each prototype. This is called “prototype-based spiral development," which is discussed in the next section.

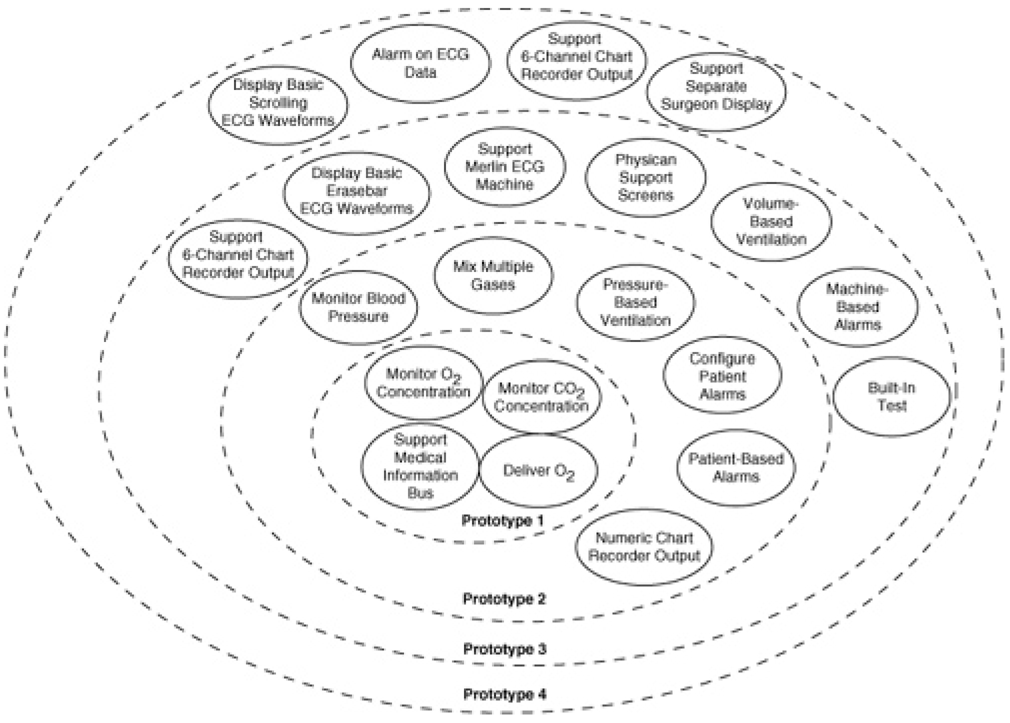
4.3. Prototype-Based Spiral Development

All three timescales described in the preceding section are important, but the incremental microcycle is a central viewpoint from a planning and delivery standpoint. The key work product at this level is known as the **prototype.** A prototype, also known as an **increment,** is a high-quality version of the system that is fully validated against the requirements that it implements. The system is constructed incrementally as a series of such prototypes, each adding more functionality, and each is completed at the end of a microcycle.

A couple of key aspects of the prototype are important to note. First, the prototype is not something you hack together to test a concept. It contains the actual code that will ship in the product. Early prototypes are simply less complete than later ones. Each microcycle adds more features and capabilities to the system. This is mostly a matter of addition to the existing prototype functionality, but there is also optimization of that functionality and there is a certain amount of reorganization of that functionality. Optimization takes place in each microcycle in its design phase, but as the project progresses, there is an increasing emphasis placed on optimization, as discussed in the preceding section on the macrocycle phases. The second aspect is the reorganization of existing functionality, commonly known as **refactoring**. This is a natural outcome of the incremental development lifecycle. Experience has shown that if the process is followed, the refactoring tends to be a rather small effort and not a huge deal, even though developers who haven't used an incremental lifecycle before find the prospect intimidating.

Mostly, the prototype functionality grows in microcycle spurts, like growth rings on a tree. For example, Figure 4.4 shows the incremental evolution of an operating room patient ventilator; the dashed ovals represent the prototypes, and the small ovals inside are the use cases realized and validated for that prototype.

Figure 4.4 *Incremental prototype elaboration*


Alarm on ECG Data
Support 6-Channel Chart Recorder Output
Support Separate Surgeon Display
Display Basic Scrolling ECG Waveforms
Display Basic Erasebar ECG Waveforms
Support Merlin ECG Machine
Physican Support Screens
Volume-Based Ventilation
Support 6-Channel Chart Recorder Output
Mix Multiple Gases
Monitor Blood Pressure
Pressure-Based Ventilation
Machine-Based Alarms
Monitor O_2 Concentration
Monitor CO_2 Concentration
Configure Patient Alarms
Built-In Test
Support Medical Information Bus
Deliver O_2
Patient-Based Alarms
Prototype 1
Numeric Chart Recorder Output
Prototype 2
Prototype 3
Prototype 4

In a fully incremental lifecycle such as Harmony/ESW, the process is rather smoother than that. The growth rings really reflect the formal validation of the prototypes. The actual functionality of the system grows in (approximately) daily increments through the practice of continuous integration, as discussed briefly in the previous chapter. With continuous integration, the developer performs small work items, such as adding an aspect of a product feature, tests it at the unit level, and then integrates it into the baseline. Once this baseline has passed a set of tests performed by the configuration manager, these changes are made available to the development staff as a whole. The integration tests are normally a subset of the validation tests and are updated as new features are added to the prototype baseline. Formal validation takes place at scheduled intervals—that is, at the end of each microcycle—to ensure that the quality of the system is increasing or at least is maintained.

The product is therefore essentially constructed in a series of small subprojects, each of which results in a validated version of the system with a specified level of functionality. This functionality is organized by product "feature" or "use case"; that is, each microcycle produces an intent document, known as the microcycle mission statement, that outlines the goals of the microcycle. The most important of these are the new features or use cases to be added, but other goals, such as defects to be fixed, project risks to be reduced, architectural aspects to be added, and platforms to be supported, are important as well.

The reason for developing the product incrementally is that it is far easier to obtain systems of high quality—or equivalent quality at a lower cost—by creating high-quality pieces of the system all the way through. As I said before, *the best way not to have defects in a system is not to put defects in the system*. When the product is developed a little bit at a time, and quality is addressed all the way through, far fewer defects get embedded in the software.

Integration and validation are vastly accelerated because of a greatly reduced need to fix bugs.

In a traditional industrial process, validation comes only at the end of the entire process. The complete system is in place, but there has been little in the way of validation or testing before that. In an effort to improve quality, manual desk checking (code inspection) and reviews are typically done, but in fact, those quality assurance measures are significantly less effective than running the system and testing what it does against what you expect it to do. The incremental construction of the system as a set of increasingly capable prototypes emphasizes execution and testing and performs those activities as early as possible.

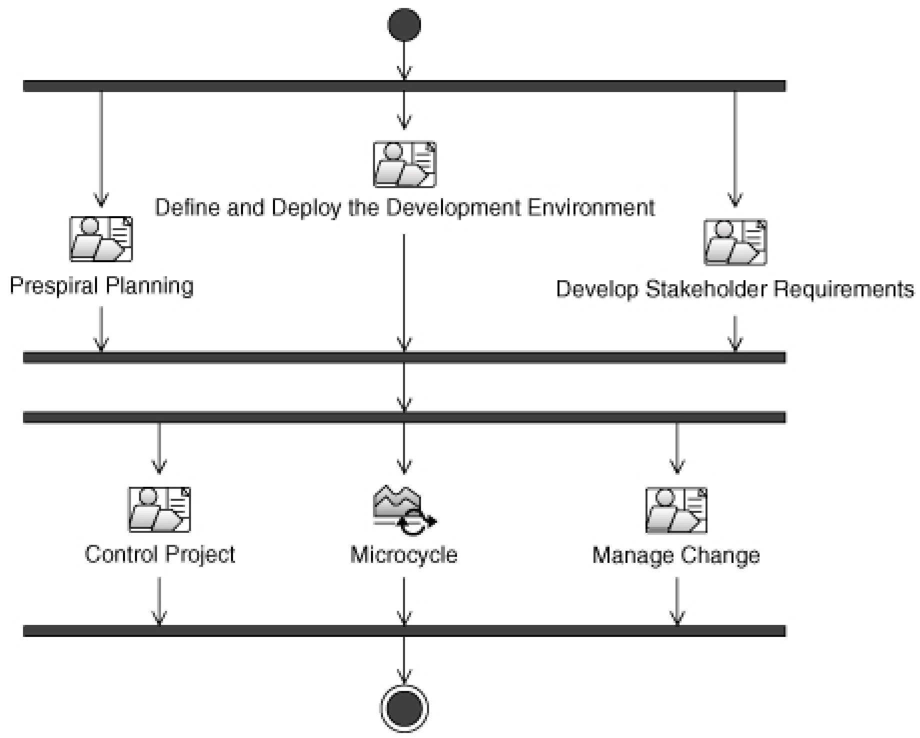
For an 18-month project, testing starts as early as the very first day. Validation occurs at the end of the first microcycle, typically four to six weeks into the project. Testing continues to be done every day, and formal validation is repeated at the end of each microcycle. In a typical project, this means that validation is actually performed 12 to 18 times, but each time at a far lower cost than the single validation of a traditional process. The end result is a higher-quality system in less time with less effort.

The Harmony/ESW process is captured using UML activity diagrams and can be shown at different levels of abstraction. The next section will introduce this notation to discuss the process workflow details.

4.4. Harmony Macrocycle Process View

While most of the action from the developer point of view takes place within the microcycle, the macrocycle provides a context into which the microcycle fits. Figure 4.5 shows the highest-level task view for the Harmony/ESW process.[8] This figure is, of course, a UML activity diagram. This is the most common way to depict process workflows. The heavy horizontal lines are forks and joins; the activities executing between them run in parallel with each other, often with feedback among themselves. Thus the three activities—prespiral planning, defining and deploying the development environment, and developing stakeholder requirements—are done in parallel. It doesn't matter which starts or finishes first. However, the three subsequent activities—controlling the project, managing change, and the microcycle itself—do not start until those three tasks are complete.

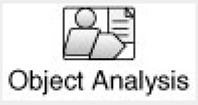
Figure 4.5 *Harmony macrocycle view*


Define and Deploy the Development Environment
Prespiral Planning
Develop Stakeholder Requirements
Control Project
Microcycle
Manage Change

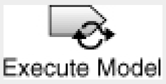
8. The process workflows are captured from the Harmony/ESW content in the Eclipse Process Framework (EPF) open-source tool. In the EPF, activities are groupings of tasks, and an iteration is a kind of activity. All the elements on this figure are activities and the microcycle is an iteration. In the next figure, the elements are tasks contained within the prespiral planning activity. Tasks are refined by specifying the steps they require or optionally use for completion. For more information about EPF, or to download it, visit www.eclipse.org/epf. The EPF Composer is an open-source, lightweight version of the commercial product Rational Method Composer (RMC). For more information visit www.ibm.com/developerworks/rational/products/rup/.

A Note about Process Workflow Notation

The diagrams used to depict workflow in this book are created (mostly) in the Eclipse Process Framework (EPF) Composer (an open-source process capture tool available at www.eclipse.org/epf) or with the more powerful Rational Method Composer (RMC). The icon for an activity looks like this:

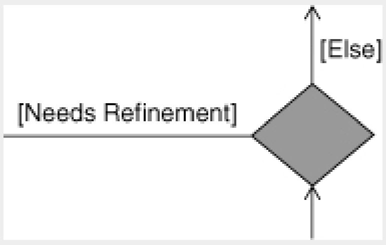


It represents a workflow containing, usually, multiple tasks. A task uses a different icon:

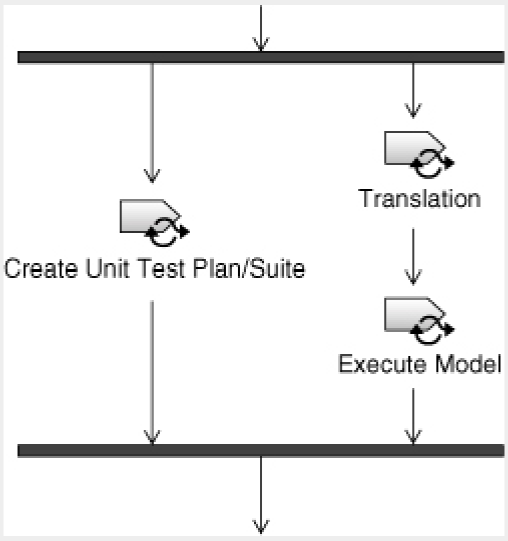


A task is a unit of work that is described by a series of steps, is performed by one or more roles, and (usually) has inputs and outputs.

Activities and tasks can be sequential (connected by flow arrows) and may lead to operators such as a branch:



which selects which subsequent task or activity to do, a fork or join:



in which case the subsequent activities or tasks are executed in parallel.

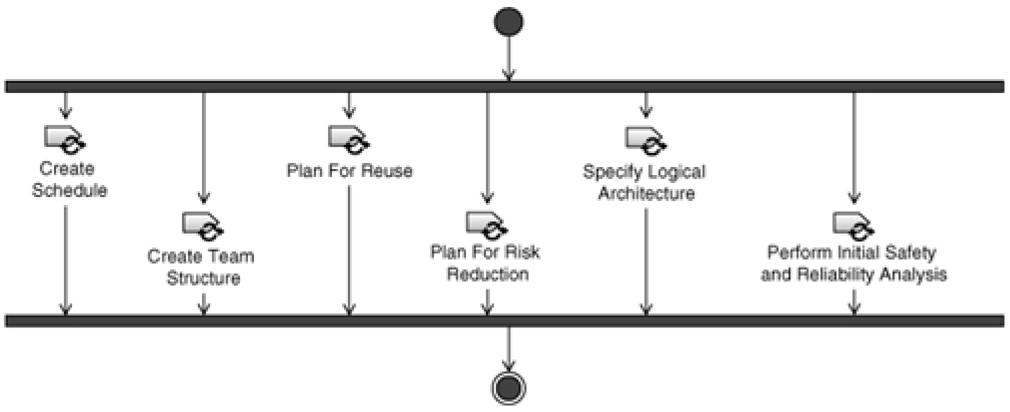
Once the microcycle starts up, two parallel activities—“Control project” and “Manage change”—take place in parallel with it. For the most part, these activities are performed by different roles from the ones within the microcycle, but they are concurrent in the technical sense; that is, the tasks and steps within the concurrent activities are sequential within themselves, but the ordering between the concurrent activities isn’t known or even of much interest.

I will briefly describe each of these activities next. The microcycle will be discussed in more detail in the following section.

4.4.1. Prespiral Planning

Prespiral planning takes place before any development work can begin. This activity will be the subject of the entire next chapter. Nevertheless, you can see in Figure 4.6 the tasks that are involved. These tasks will be discussed in some detail in the next chapter.

Figure 4.6 *Prespiral planning*


Create Schedule
Create Team Structure
Plan For Reuse
Plan For Risk Reduction
Specify Logical Architecture
Perform Initial Safety and Reliability Analysis

4.4.2. Defining and Deploying the Development Environment

Before the microcycle work can begin, the work environment must be defined and established. This activity won't be discussed in much detail in this book. The tasks involved include

• Tailoring the process

It is not uncommon for a project to have special needs or intent, requiring small modifications to the approach. For example, a given project may be safety-critical, requiring additional work products, tasks, and roles; or it may be done in conjunction with outsourcing, requiring additional monitoring and management tasks. In this task, the particular nature of the project is taken into account, and any process customization required is made.

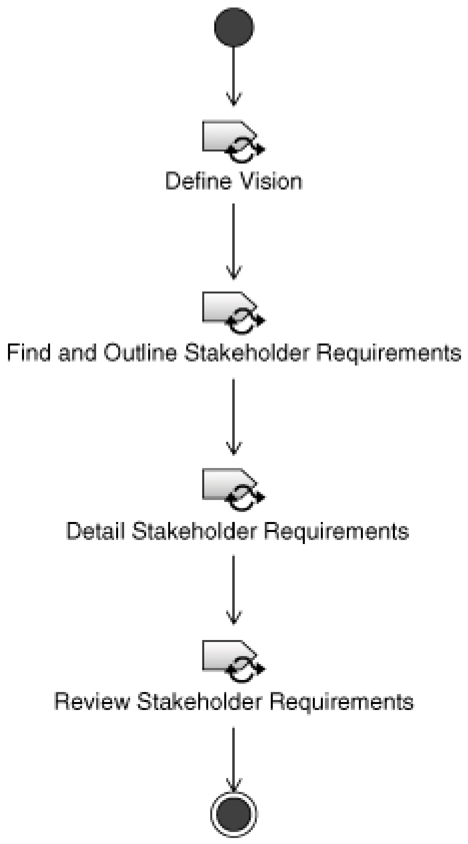
• Installing, configuring, and launching the development tools

Developers use an entire suite of tools that cover the range from word processing to modeling, compiling, testing, and CM. They must be installed, configured, and initialized so that the environment is ready when development begins. Launching the development environment often includes training the developers on the use of the tools.

4.4.3. Developing Stakeholder Requirements

Stakeholder requirements are requirements from the stakeholder point of view. In this case, the stakeholder point of view combines the customer (the one paying for the system) and the user. These requirements are usually nontechnical in the sense that they describe or define a stakeholder need and how the product will satisfy that need in its operational context. Stakeholder requirements should be stated from a black-box point of view and not specify how the product will be developed or its internal structure or behavior. The tasks involved in this activity are shown in Figure 4.7.

Figure 4.7 *Developing stakeholder requirements*

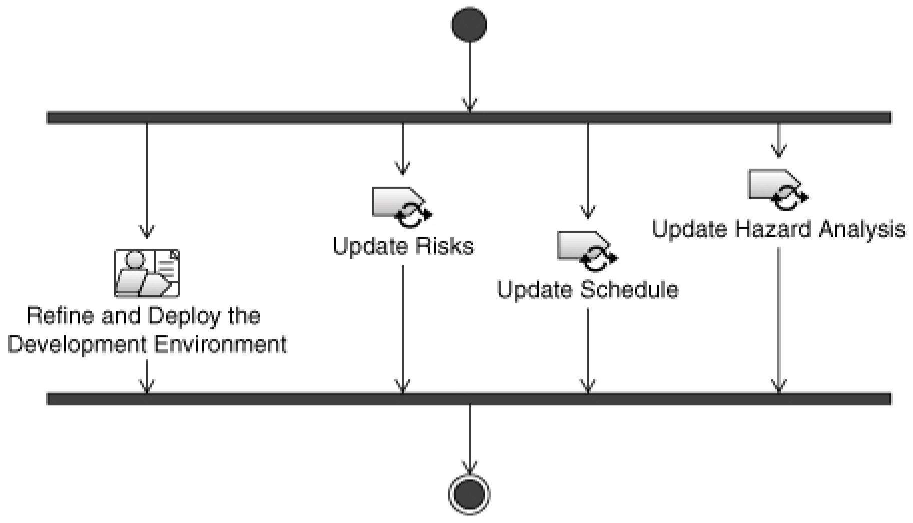

Define Vision
Find and Outline Stakeholder Requirements
Detail Stakeholder Requirements
Review Stakeholder Requirements

Two primary artifacts are created during this activity. The vision is a statement of overall needs, goals, requirements, and constraints of the system at a high level of abstraction. The stakeholder requirements are an elaborated version of these elements without getting into the technical specification of the product (that technical specification is provided by the system specification).

4.4.4. Controlling the Project

Controlling the project (see Figure 4.8) is an activity done in parallel with the microcycle. It contains management tasks for tracking and updating the risk management plan, the schedule, and—if the project is safety-critical—the hazard analysis.[9] One of the key principles of the Harmony/ESW process is "Plan, track, adapt." It is this crucial activity that performs this vital project function.

Figure 4.8 *Controlling the project*

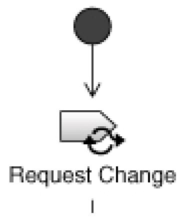


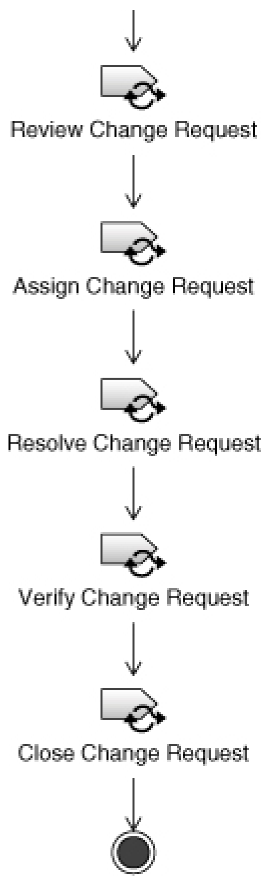
9. The hazard analysis is a key document for safety-critical systems. This work product, its creation, and its management will be discussed in more detail in the next two chapters.

4.4.5. Change Management

This activity captures the project necessity to create and manage change requests. It includes the creation and review of the change request as well as its assignment, resolution, and verification, as shown in Figure 4.9. The change management activity not only handles defects, but also manages other changes due to evolving scope, misunderstood requirements, and so on.

Figure 4.9 *Change management*

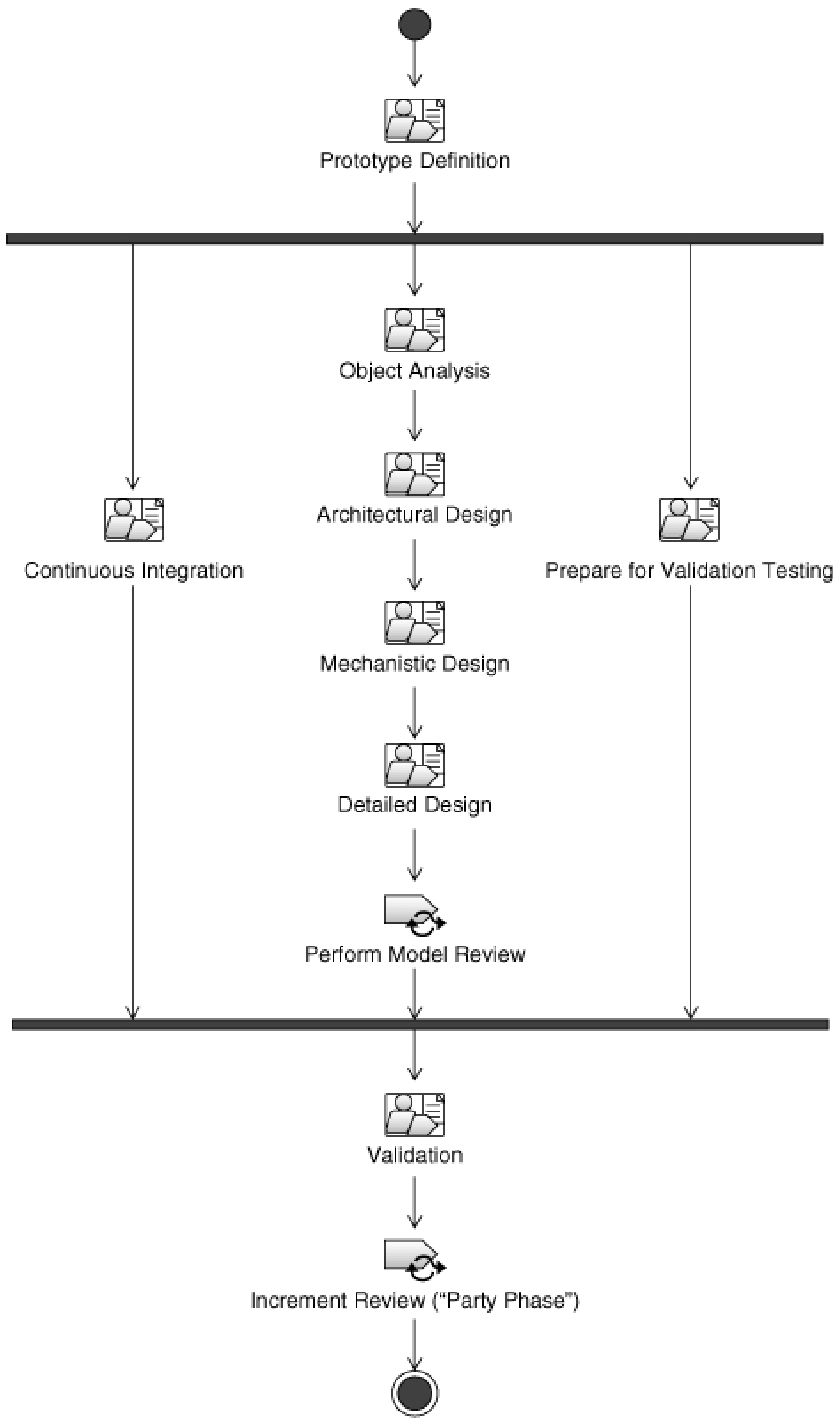




4.5. Harmony Spiral in Depth

Figure 4.3 shows the basic flow of the spiral process. Figure 4.10 presents it a bit more formally, showing the activities that are sequential and those that are concurrent. The Harmony/ESW spiral has three primary phases—analysis (comprising prototype definition and object analysis), design (containing architectural, mechanistic, and detailed design), and validation—plus a few other support activities. This section will discuss the roles, work products, tasks, and workflows for each. The Harmony/ESW process differs from a traditional process in a number of ways. For example, Harmony/ESW is incremental, so that analysis, design, and other work are applied repeatedly to add functionality over time. In contrast, in a traditional process all of the analysis for the entire project is done before any design begins. Perhaps an even more striking difference is the absence of a phase devoted to writing code. A traditional process has an analysis phase followed by a design phase during which no code is written. Following design, an implementation phase contains all the code-writing and unit test activities of the developers. The Harmony/ESW process doesn't need such a phase because code is being written and unit-tested throughout the analysis and design phases.

Figure 4.10 *Harmony microcycle (formal)*



The absence of a traditional coding phase may seem odd, but it is a natural consequence of the agility of the Harmony/ESW process. Very early on in the process, models are developed in analysis; we know about the quality of these models only when they are executed. As a result of this execution, we generate the actual (if early) code for the system in parallel with the models. Usually, this code is generated automatically from the models, but even if the developer is using a very lightweight modeling tool, it is crucial that he or she be able to assess the model's quality, and this is possible only through execution. In addition, either immediately preceding the modeling or simultaneously with it, unit tests are created and modeled (or coded) as well to ensure that the model is doing the right thing at the right time. Later in design, the models are optimized against the weighted set of design criteria. Just as before in the analysis phase, it is crucial that the quality of the design be assessed through model execution and unit testing, resulting in the source code. The model (and code) baseline is continuously updated to include unit- and integration-tested elements, resulting in an updated baseline at least once per day on average. As a result, there is no need for a traditional coding phase; by the time the developer gets to the point where it would be done, the code is already there.

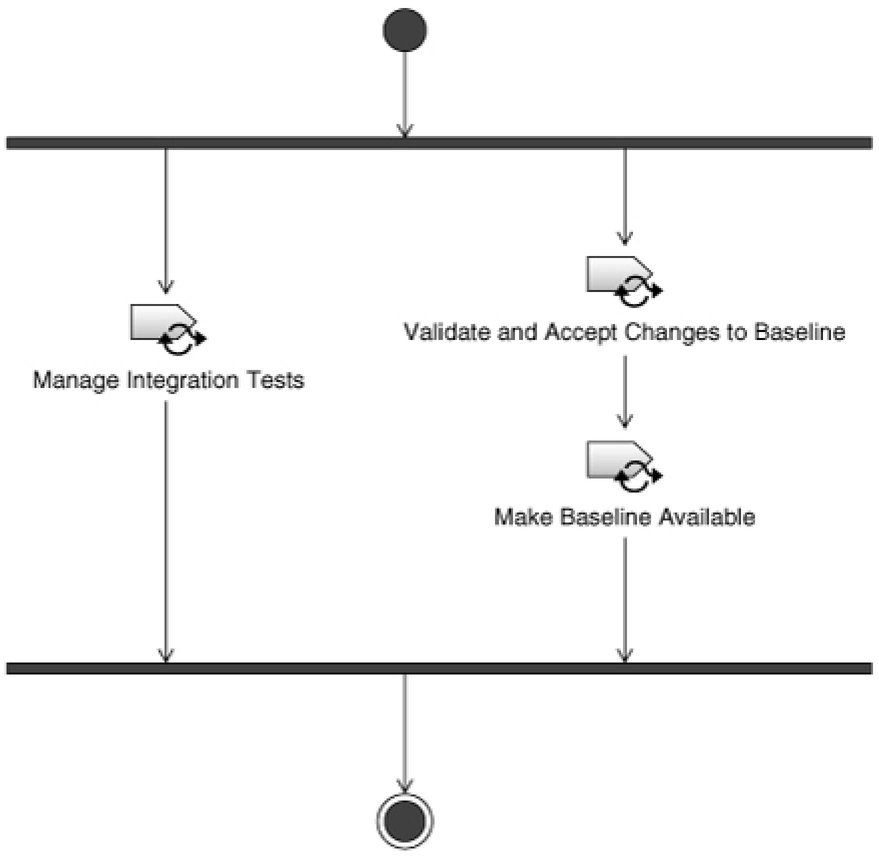
While many agile adherents discourage the use of reviews, I have found that performing a model review can have a number of benefits. It is not strictly required, but it improves understanding of the architecture and how the collaborations work within the architecture, and it can identify subtle problem-domain defects not otherwise identified through unit and integration testing.

The increment review has great value to the project and is strongly recommended. The increment review is short—typically half a day to a day in length—and is the key for process improvement. It is the point in the spiral where issues with the architecture, project risks, schedule accuracy, and project execution are evaluated and addressed.

4.5.1. Continuous Integration

Continuous integration is performed in parallel with development activities. In practice, this means that each developer will test the software and submit changes on a daily or more frequent basis. The configuration manager will perform integration tests (a subset of the validation tests) and, if everything works, update the current baseline for all developers to use. In addition, the integration manager manages the set of integration tests, adding more as new capabilities are added to the system, as well as any necessary integration test fixtures. This workflow is shown in Figure 4.11.

Figure 4.11 *Continuous integration workflow*

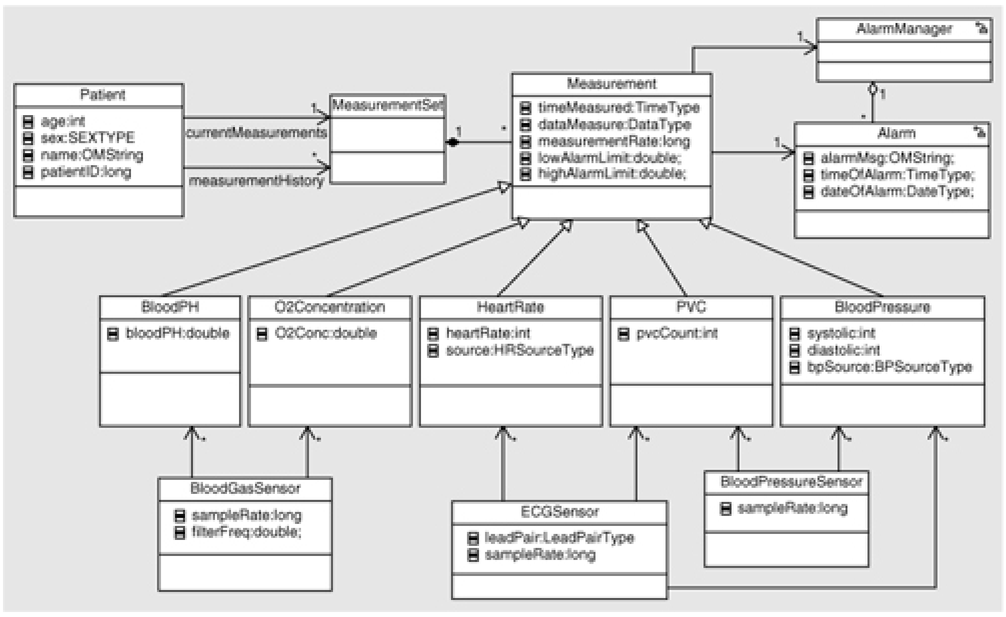


4.5.2. Analysis

The analysis phase of the microcycle is intended to identify the properties and characteristics of the system that are essential. By the term *essential* we mean that any acceptable product release has these properties. If the properties are free to vary—such as using a CAN bus instead of Ethernet, or VxWorks instead of the Integrity operating system—then they do not belong in the analysis model.

For example, a patient-monitoring system probably would identify a model such as the one shown in Figure 4.12. The classes `Patient`, `HeartRate`, `PVC` (preventricular contraction count), `BloodPressure`, `O2Concentration`, and so forth are essential for any such system. However, the technology used for communication protocols, distribution middleware such as CORBA or DDS, the operating system, and even the concurrency scheduling such as cyclic executive or multitasking preemption are all *inessential*. Different choices could be made without changing the functionality of the system. Technologies and design decisions are related to optimality and therefore are relegated to design, not analysis.

Figure 4.12 *Patient monitor analysis model*



Harmony has two subphases within analysis. The first, **prototype definition,** concerns itself with identifying the tasks to be performed within the current microcycle. This includes, most importantly, definition of the requirements and their mapping to product use cases, but there are other concerns as well, such as the project risks to be reduced, identified defects to be repaired, and so on. In MDA terms, this activity focuses on the CIM.

The second subphase is **object analysis.** This phase focuses on identifying the essential model elements for the use cases being realized within the current microcycle. This object analysis model is known as the PIM in the MDA standard. The set of elements realizing a use case is called a **collaboration.** The analysis model must execute and be (functionally) unit-tested during its development—a key notion of the agile approach—and is done in what Harmony/ESW refers to as the nanocycle.

Prototype Definition

Prototype definition is all about specifying the work in the current microcycle. The work goals of the microcycle are summarized in the microcycle mission statement. The primary goal of the microcycle is to identify

• The use cases or features to be realized and validated in the next prototype

• The existing defects to be repaired

• The identified risks to be mitigated

• The architectural intent of the prototype to be realized

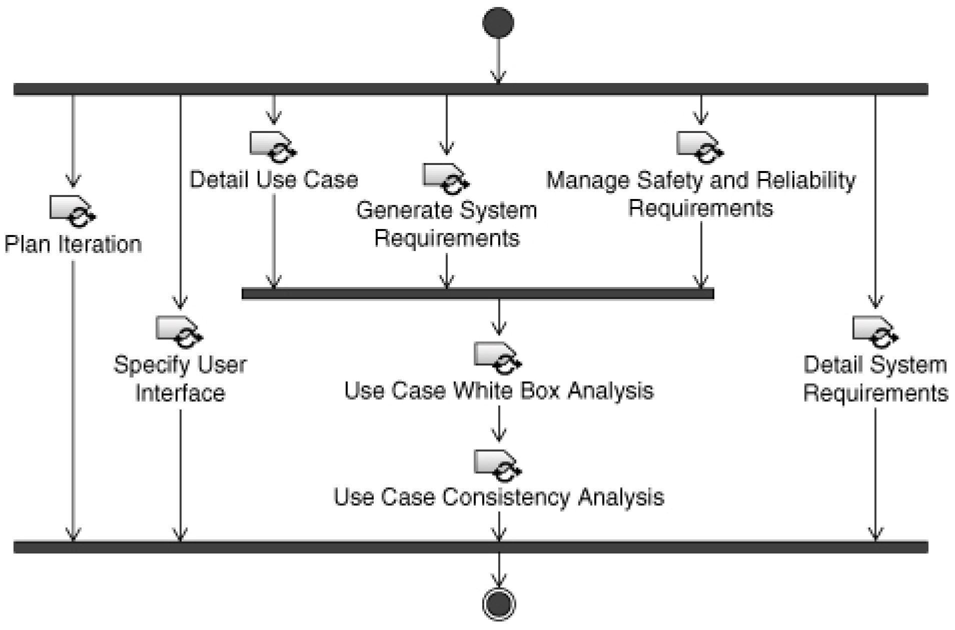
• The target platforms to be supported

The work items list contains the backlog of work to do. Most of this work is organized around realizing the use cases, but the other goals are also represented. Elements from the work items list are selected for work within the current microcycle to realize its goals.

At the start of prototype definition, the use cases are named and given a one-paragraph description, (or imported from the Stakeholder Requirements, if available) but the detailed system requirements are not specified. They will be specified and traced to the included use cases. The use cases are detailed with sequence diagrams, activity diagrams, state machines, and constraints.

Figure 4.13 shows the basic workflow for defining the prototype. The detailed steps of the tasks and the work products used, created, and modified will be discussed in more detail in Chapter 6.

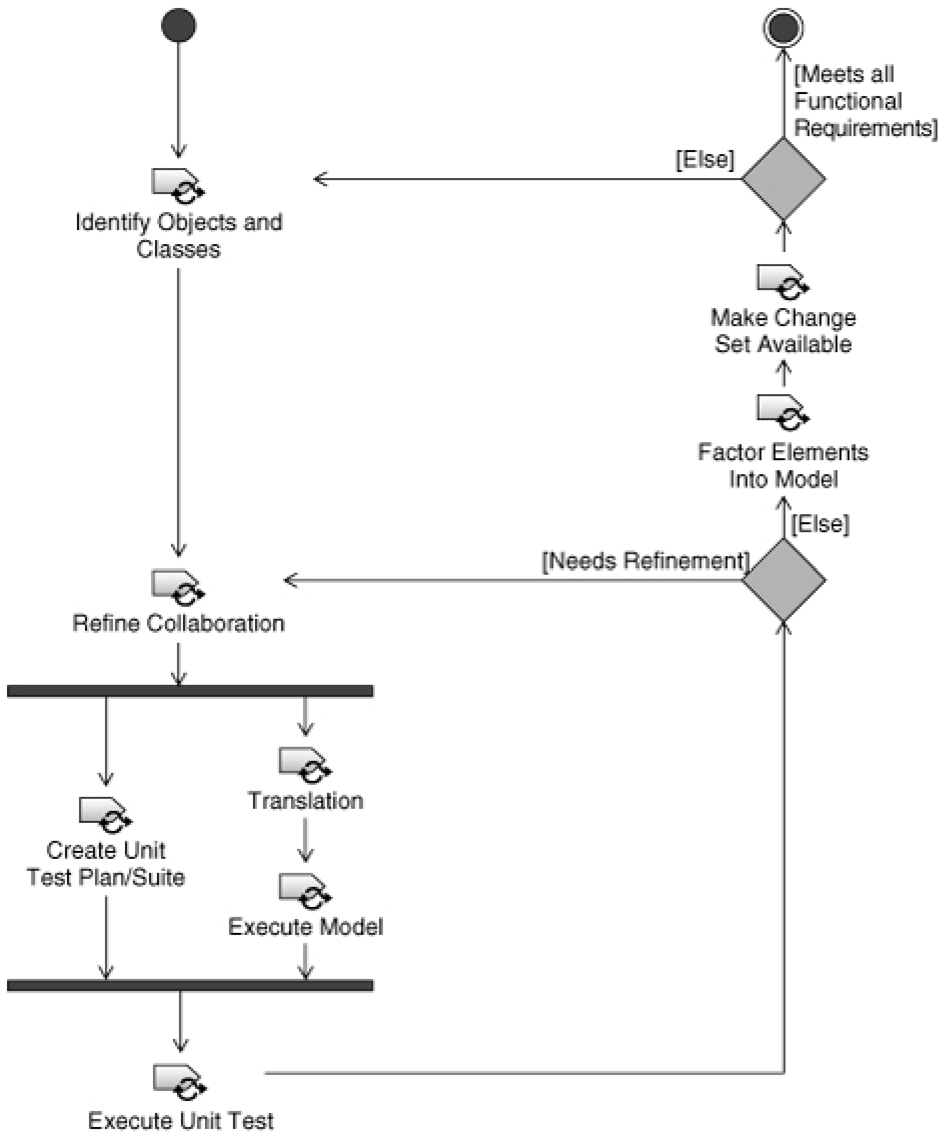
Figure 4.13 *Prototype definition workflow*


Detail Use Case
Generate System Requirements
Manage Safety and Reliability Requirements
Plan Iteration
Specify User Interface
Use Case White Box Analysis
Detail System Requirements
Use Case Consistency Analysis

Object Analysis

Object analysis (Figure 4.14) identifies the essential classes, types, and relations and specifies their essential behavior. Object analysis is done on a per-use-case (or per-feature) basis. At the end of this subphase, each use case will be realized by an analysis collaboration that meets all of the functional requirements of that use case. In a small project, the use cases will be realized sequentially, whereas a larger project will work on multiple use cases simultaneously, one per team.

Figure 4.14 *Object analysis workflow*


[Meets all Functional Requirements]
[Else]
Identify Objects and Classes
Make Change Set Available
Factor Elements Into Model
[Else]
[Needs Refinement]
Refine Collaboration
Translation
Create Unit Test Plan/Suite
Execute Model
Execute Unit Test

The analysis work proceeds incrementally with frequent submissions to the configuration manager to update the project baseline. This requires continuous execution and testing of the evolving collaboration. To this end, the developer creates and applies unit tests in parallel with the addition and refinement of classes and other model elements. The task of making the change set available consists of submitting the changes to the configuration manager for integration and test. If these tests are passed, then the submitted changes become part of the new baseline available to all developers. Object analysis will be discussed in detail in Chapter 6.

4.5.3. Design

Design is all about optimization in the Harmony/ESW process. This means that we are going to focus on issues such as delivered qualities of service (e.g., worst-case performance, throughput, bandwidth, reliability, and safety) and other product design criteria such as maintainability, manufacturability, reusability, and time to market. We do this at three levels of abstraction: architectural, mechanistic, and detailed. All levels of design use a similar workflow:

1. Identify the design criteria of importance.

2. Rank them according to criticality.

3. Select design patterns and technologies that optimize the most important ones at the expense of the least important.

4. Apply the design decisions.

5. Validate that the design doesn't break existing functionality and that it achieves the desired optimizations.

As with object analysis, design continuously translates the models into code and performs informal debugging and more formal unit testing on the system. Design is discussed in detail in Chapter 7, "Agile Design."

Architectural Design

Architectural design optimizes the system at an overall level by selecting architectural design patterns and technologies. Harmony/ESW identifies five primary views of architecture as well as a number of secondary views. These views—along with the secondary views—are clearly visible in Figure 4.15. The primary views are

• Subsystem and component architecture

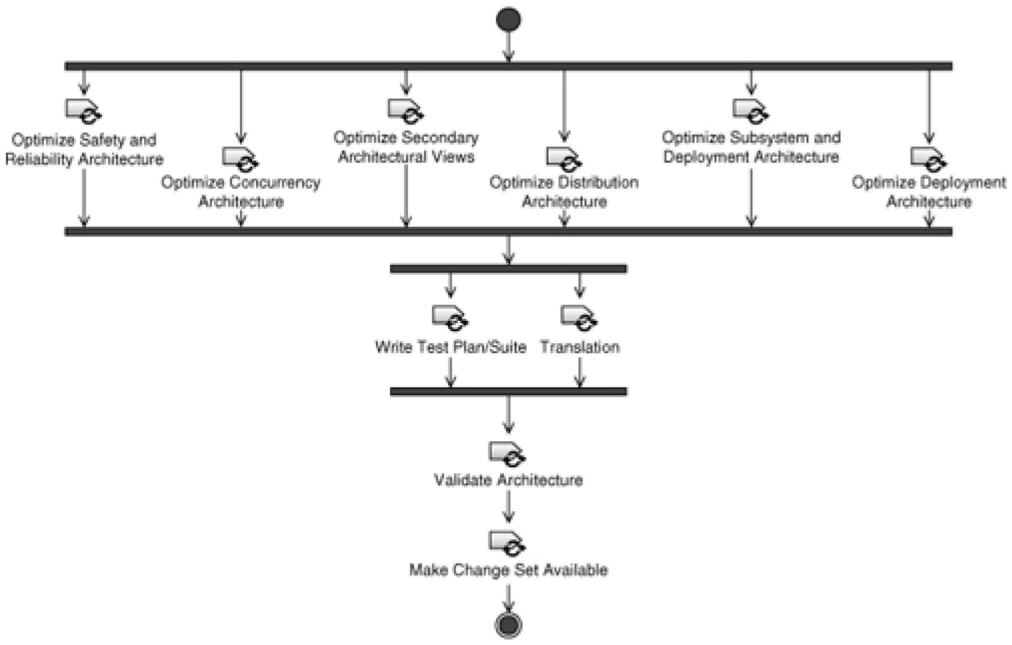
• Distribution architecture

• Safety and reliability architecture

• Concurrency architecture

• Deployment architecture

Figure 4.15 *Architectural design workflow*


Optimize Safety and Reliability Architecture
Optimize Concurrency Architecture
Optimize Secondary Architectural Views
Optimize Distribution Architecture
Optimize Subsystem and Deployment Architecture
Optimize Deployment Architecture
Write Test Plan/Suite
Translation
Validate Architecture
Make Change Set Available

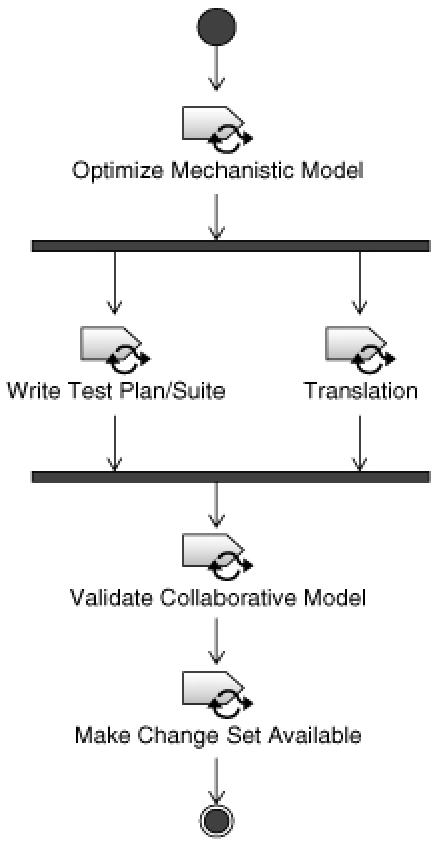
It is important to note that not all these architectural concerns are addressed in every prototype, but they will be by the time the product is ready for release.

The patterns used for architectural design are large-scale, such as those in my book *Real-Time Design Patterns* or Buschmann et al.'s *Pattern-Oriented Architecture*, vol. 1, *A System of Patterns*.

Mechanistic Design

Mechanistic design optimizes the system at the collaboration level. Each use case is realized by a single collaboration of elements, possibly spread across multiple subsystems. Mechanistic design optimizes that collaboration and has limited or no effect on other collaborations within the system. The workflow shown in Figure 4.16 is repeated within the microcycle for each use case being realized. Small projects may do this sequentially; large groups will have teams do this concurrently.

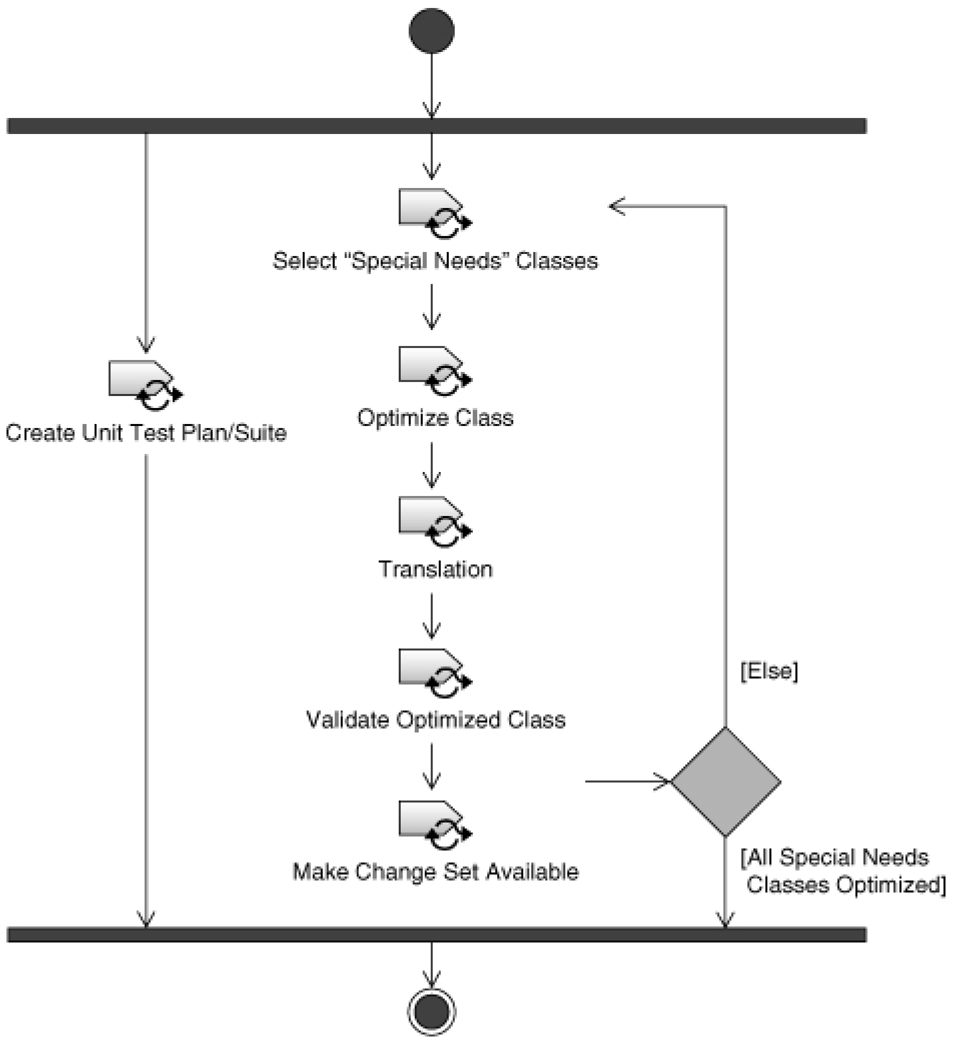
Figure 4.16 *Mechanistic design workflow*


Optimize Mechanistic Model
Write Test Plan/Suite
Translation
Validate Collaborative Model
Make Change Set Available

Detailed Design

Detailed design is the lowest-scale design in the Harmony/ESW process. It focuses on optimizing individual classes, functions, and data structures. Normally, only a small percentage of the elements of a model require special attention; this may be because the element is part of a high-bandwidth path, it has extraordinarily high reliability or safety standards, or it is unusually complex. The workflow shown in Figure 4.17 is repeated for all such "special needs" classes.

Figure 4.17 *Detailed design workflow*


Select "Special Needs" Classes
Create Unit Test Plan/Suite
Optimize Class
Translation
[Else]
Validate Optimized Class
Make Change Set Available
[All Special Needs Classes Optimized]

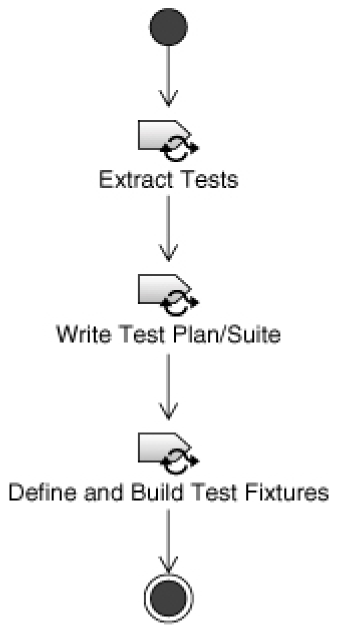
4.5.4. Prepare for Validation

Once the use cases for the current prototype are detailed and the system requirements identified in the prototype definition phase, preparations for validation can begin. This activity runs in parallel with object analysis and the design phases of the microcycle.

For the most part, the use case state machines and scenarios derived from those state machines become the functional test cases for the prototype validation. There are normally other test cases for performance, and QoS, safety, reliability, usability, and robustness tests are added as well. Those tests are specified during preparation for validation. In addition, any necessary test fixtures, simulators, and test documentation are created. The goal is to be ready for a formal validation test by the end of the design phase.

Figure 4.18 shows the simple workflow for this activity, which will be elaborated in Chapter 8, "Agile Testing."

Figure 4.18 *Prepare for validation workflow*



4.5.5. Model Review

The problem with reviews—and the reason that many agile methods folks discourage their use—is that they are expensive and often don't add significant value. Although optional, model reviews can add value, but only if they are performed on models at the right level of maturity and if they are performed in the right way.

The Harmony/ESW process bases its model review approach on Fagan inspections.[10] The inspection technique was identified as a way to get early quality in lieu of testing. Since the Harmony/ESW process performs continuous testing and integration, it is arguably of less value than in processes that push all testing and integration to the end of the project. Nevertheless, inspections can disseminate information about the structure, behavior, and functionality of software elements to a larger audience than its developers and can identify concerns with problem-domain semantics in the model.

10. See Michael Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* 15, no. 3 (1976), 182–211, available at www.research.ibm.com/journal/sj/153/ibmsj1503C.pdf.

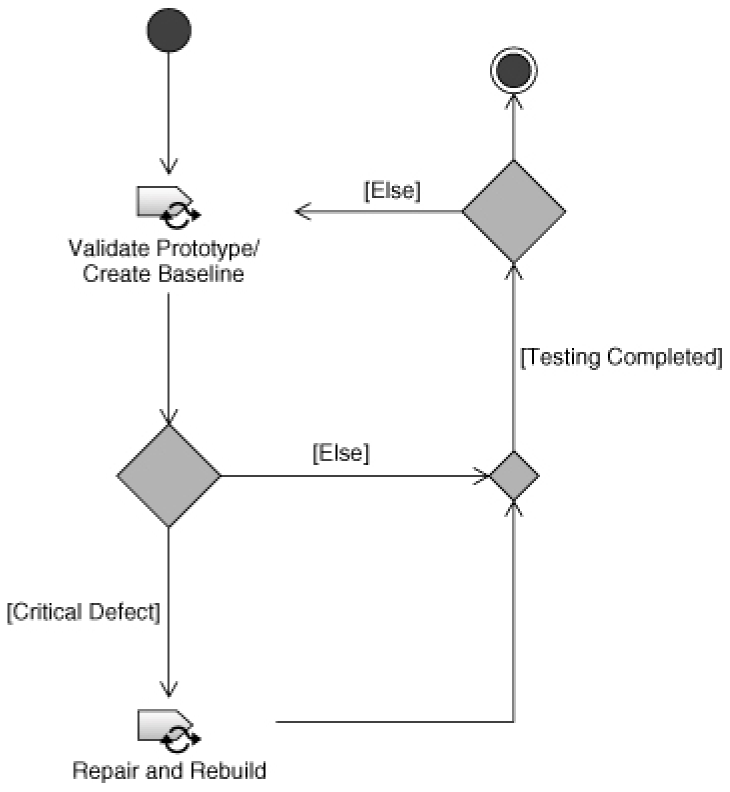
The key requirement for model reviews to take place is that the model is already of high quality prior to a review. This means that the model under review has been unit-tested. Remember that you've got 6 to 20 people in a room, each costing over $100/hr in overhead, so it is hugely expensive to identify obvious coding errors and mistakes in a review. It is far more cost-effective for the developers to find those problems themselves during the translation and unit testing. In a similar vein, the model review is run in a particular fashion, specifically deferring resolution of identified issues to outside the meeting. So we want not to find issues such as “That line of code is missing a semicolon—that won’t even compile!” during the review, but we *do* want to find issues such as “In that mode, the safety standard requires that the control rods be fully embedded in the nuclear reactor core.”

Model reviews are discussed in Chapter 9, “Agile Process Optimization,” along with iterative project reviews.

4.5.6. Validation

Validation is a formal procedure meant to ensure that the product meets its requirements. In the Harmony/ESW process, it applies the test vectors specified in the “Prepare for validation” activity to the prototype; by this time all of the use cases in the microcycle mission statement have been realized, some defects from previous iterations have been fixed, and some number of risks have been reduced. The basic workflow is shown in Figure 4.19. Validation is discussed in some detail in Chapter 8.

Figure 4.19 *Validation workflow*



4.5.7. Increment Review (Party Phase)

The increment review (it's a *party* not a *postmortem!*) examines the project progress, compares it to expectations, and looks at changes and new risks. The purpose is to make frequent course corrections to keep the project on track, and it is one of the primary ways in which the Harmony/ESW process realizes the principle of dynamic planning. This task solicits feedback from the developers and project contributors and then looks to see how it can make the process work better. The primary areas of focus are

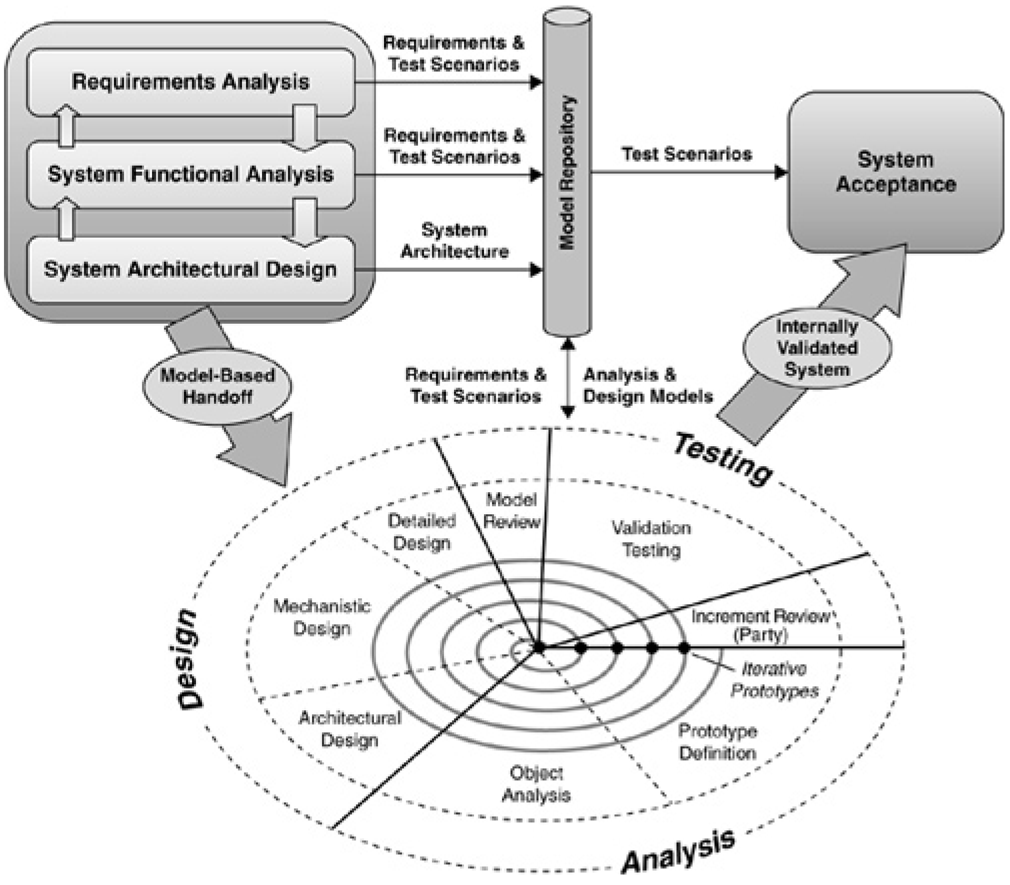
- Process efficiency
- Prototype quality
- Schedule
- Risk mitigation
- Architecture scalability

This activity is the primary topic of Chapter 9.

4.6. What about Systems Engineering?

Systems engineering is not a part of Harmony/ESW per se. A slightly different development process within the Harmony family takes Harmony/SE and combines it with Harmony/ESW to form a complete Harmony process that includes both. This process has a hybrid-V-cycle lifecycle (see Figure 4.20) in which systems engineering precedes software and hardware development, but once the handoff from systems engineering takes place, software (and hardware) are developed incrementally. In fact, the software microcycle is almost identical to the Harmony/ESW microcycle. There are a few minor changes in the prototype definition subphase of the analysis phase of the Harmony/ESW microcycle to account for the fact that systems engineering has already defined the use case model and made hardware-software decomposition decisions. Beyond that, the microcycle is identical to its description above.

Figure 4.20 *Hybrid-V spiral*



The hybrid-V lifecycle is appropriate when there is significant hardware/software codevelopment. If the hardware already exists or is well understood, the full spiral lifecycle, the focus of this book, is more optimal.

4.7. What about CMMI?

CMMI stands for Capability Maturity Model Integration and is a standard owned by the Software Engineering Institute (see www.sei.cmu.edu/cmmi). It is a process improvement approach to provide development organizations with the essentials of effective development processes. The fundamental premise of CMMI is that the quality of systems is strongly influenced by the process used to develop them. The benefit of using CMMI is that processes can be improved to result in more cost-effective and higher-quality systems.

There is a great deal of confusion about CMMI in the software development industry. It is either idolized or demonized, depending on the group speaking. But really, all CMMI does is describe the characteristics of an effective process. CMMI is *not* a process itself. The CMMI product suite is a collection of process models,[11] training materials, and methods of process appraisal generated from the CMMI framework.

11. In this context, a process model is a structured collection of practices.

CMMI has different models for development, acquisition, and supplier management. The part that pertains to development is called CMMI-DEV. Results posted on the CMMI Web site claim 34% median improvement in cost and 50% median improvement in schedule with data from 30 different organizations.

CMMI classifies the process maturity of development organizations into five levels:

1. *Initial*—Processes are ad hoc and chaotic. Projects do not have a stable development environment, and success often depends upon heroic efforts or personnel rather than on proven approaches and methods.

2. *Managed*—Projects are planned and executed in accordance with those plans. Projects have properly skilled people with enough resources to achieve their product goals.

3. *Defined*—Processes are well characterized and understood and are specified in standards, procedures, tools, and methods. Processes are tailored to specific project needs and are more rigorously defined than in level 2.

4. *Quantitatively managed*—Companies and projects have quantitative objectives specified for quality and process performance and use this data to facilitate process management. Often this quantitative data is statistically analyzed, and this adds a level of predictability beyond what is found in level-3-compliant companies.

5. *Optimizing*—The organization continuously improves its processes by applying the results of

quantitative measurement to incremental and continuous evolution.

CMMI focuses on ensuring that the development organization does what it claims it does in terms of processes; hence the emphasis on appraisals. In some circles, CMMI has a bad name, but I believe that is because people not involved in development have defined processes that are extremely heavyweight and then used CMMI methods to ensure compliance. The problem is the first part—the definition of expensive, paper-driven processes—not with the compliance to CMMI. CMMI is not anti-agile, although many implementations might lead you to that conclusion. CMMI is about doing what you say and improving processes.

For example, the Team Software Process (TSP), championed by Watts Humphrey, incorporates most of what's needed for CMMI level-5 compliance. It has been used by the Naval Air Systems Command (NAVAIR) AV-8B Joint System Support Activity to accelerate the climb to CMMI level-4 compliance to 60% faster than typical.[12] In another example, Intel's IT department streamlined its CMMI processes to be 70% shorter.[13] Also, Science Applications International Corporation (SAIC) combined Scrum (an agile-based process) and CMMI to achieve and sustain CMMI level-3 compliance.[14]

12. www.stsc.hill.af.mil/crosstalk/2004/01/0401Pracchia.html.

13. "Why Do I Need All That Process? I'm Only a Small Project," *CrossTalk: The Journal of Defense Software Engineering* (February 2008).

14. Liz Barnett, "Adopting Development Processes," Forrester Research (March 25, 2004), www.forrester.com/Research/Document/Excerpt/0,7211,34039,00.html.

CMMI provides feedback as to how to improve processes. As Kent Beck says, "In software development, optimism is a disease; feedback is the cure." The Harmony/ESW process can be implemented at different levels of CMMI maturity. It incorporates the key features for level-5 compliance, if desired. For a detailed mapping of the Harmony/ESW elements to CMMI, see Appendix B, "Harmony/ESW and CMMI: Achieving Compliance."

4.8. Combining Agile, MDA, and Harmony

It should be noted that while Harmony/ESW is certainly agile, it goes beyond agile methods. That is because agile methods are not a process by themselves but are a cohesive set of strategies used to realize a process. The three timescales of the Harmony process shown in Figure 4.2 cover the entire development lifecycle for a project, whereas agile methods are almost entirely focused on the nanocycle level. Harmony/ESW provides guidance at all three timescale levels.

Beyond agility, modeling in general, and UML in particular, has proven to be enormously beneficial not only in the specification and design of systems but also in maintaining and reusing the intellectual property codified in those models. MDA is the OMG standard providing guidance on the development of models that facilitate interoperability of heterogeneous systems and that are reusable in a variety of different execution contexts.

Harmony also specifies how to perform systems engineering and provides detailed guidance on how to organize and structure model-based handoffs of system specifications to software development.

Harmony, well, *harmonizes* the different technologies and approaches into a cohesive development process.

4.9. Coming Up

This chapter provided an overview of the Harmony/ESW process. Harmony/ESW is applicable to general software and system development but really focuses on the needs and concerns of real-time and embedded projects; that is, it provides especially strong guidance on modeling and managing processing efficiency, qualities of service, device-driver-level components, embedded architectures, safety, and reliability. The Harmony/ESW process provides guidance at three timescales. The macrocycle concerns itself with key project milestones and the organization of the iterations (microcycles). The microcycle focuses on the realization of a small set of functionality and incrementally creates validated versions of the evolving system. The microcycle is divided into analysis, design, review, test, and party phases. The analysis activity identifies, models, and generates source code for the essential functionality of the system. The design phase optimizes the functional model against the weighted set of design criteria, at three levels of abstraction: architectural, mechanistic, and detailed. The model review task reviews the work products. Testing formally validates the system against the mission statement of the microcycle. The increment review ("party") examines the project's progress against the plan and identifies process and environmental areas to be improved. The third timescale, the nanocycle, focuses on the hourly or daily tasks performed by the developers.

This concludes the introductory section of the book. The next few chapters detail the process tasks, roles, and work products based on the Harmony/ESW principles and practices. Chapter 5 focuses on project initiation. This includes specifying the product vision, key stakeholder requirements, and setting up the development environment. Chapter 6 drills down into analysis, which includes system requirements and use cases—referred to in the MDA as the

CIM—and also creates collaborations of software elements that realize the functionality specified by those requirements. Chapter 7 discusses how the analysis models can be optimized in different ways and at different levels of abstraction to produce highly efficient real-time and embedded designs. Chapter 8 discusses the Harmony/ESW approach to testing, including unit-level (something referred to in earlier chapters) but also integration and validation. The last chapter, Chapter 9, talks about process improvement and how the Harmony/ESW process achieves this through the use of periodic process reviews and model reviews. For interested readers, Appendix B provides mapping from CMMI to the Harmony/ESW process.

Chapter 5
Project Initiation

What needs to be done before you start banging out the software implementation? In the Harmony/ESW process, an activity known as **project initiation** specifies the workflows and work products that precede actual software development. Project initiation precedes system requirements analysis because detailing those requirements is a part of the project, although it includes developing the product vision and the stakeholder requirements. The primary concerns of project initiation include:

• Prespiral planning (see Figure 5.1)

° Creating the schedule

° Creating the team structure

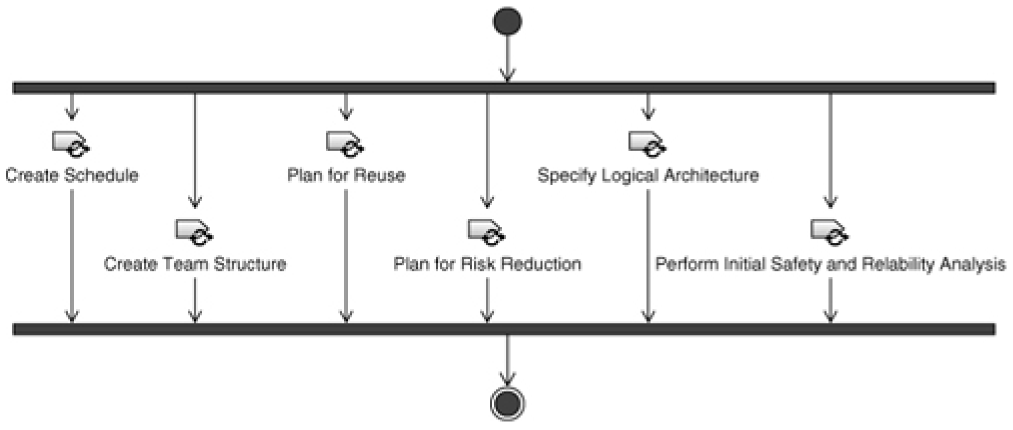
° Planning for reuse

° Planning for risk reduction

° Specifying the logical architecture

° Performing an initial safety and reliability analysis (for safety-critical or high-reliability projects)

Figure 5.1 *Prespiral planning*


Create Schedule
Create Team Structure
Plan for Reuse
Plan for Risk Reduction
Specify Logical Architecture
Perform Initial Safety and Reliability Analysis

• Developing the stakeholder requirements (see Figure 5.8)

° Defining the product vision

° Finding and outlining stakeholder requirements

° Detailing the stakeholder requirements

° Reviewing stakeholder requirements

• Defining and deploying the development environment (see Figure 5.11)

° Tailoring the process to the needs of the project as summarized in the software development plan or its equivalent

° Installing and configuring development tools, including compilers, debuggers, editors, emulators, and modeling tools

° Installing and configuring the CM environment

In addition, this chapter will discuss the philosophy and techniques for continuous configuration management.

5.1. What Do You Need to Get Started? The Baby Bear Plan

It’s not true that agile developers don’t plan. Agile developers *do* plan—they just don’t entirely *believe* their plans. Project initiation is about three primary things: setting up the development team along with the development environment, understanding what the customer needs, and planning the project.

Setting up the development team and its environment is clear enough. The team members must be selected, organized, and tasked. They must be given the set of tools necessary for them to individually perform their work tasks as well as for them to collaborate with their teammates to achieve the overall project goals. This will be discussed in some detail later in the chapter.

The customer needs are specified in the stakeholder requirements. As we will see later in this chapter, this consists of a high-level product vision and somewhat more detailed stakeholder requirements. The stakeholder requirements capture the needs of the customer entirely from the customer’s point of view and not at all in terms of the system development technology. The requirements levied on the system in terms of the technology are called the **system requirements,** a topic that will be discussed in the next chapter.

The project plans will be reviewed and updated throughout the project; in some cases more detail will be added, and in other cases a plan that has proven inadequate will be redeveloped. Despite the fact that plans change—for all the reasons I've outlined in the earlier chapters—plans still provide focus, direction, and priority for the people doing the development work. Developers need plans so that they are collectively working toward a common goal, but these plans will change, in at least minor ways. Thus, developers need *enough* planning to give them adequate direction, focus, and priority, but not *so much* planning that they waste time detailing plans beyond the fidelity of their available information.

Agile planning is much like the story of Goldilocks and the Three Bears. Planning in a traditional industrial process fully defines all potential contingencies, resulting in huge monolithic and often impenetrably unreadable documents. These are the "Papa Bear" plans. Projects that try to avoid planning altogether lack cohesion. How do the developers know what tasks they should perform and what work products are needed? Planless (or nearly so) projects have the "Momma Bear" plans—too light, not enough support. Good plans are an optimization of the opposing forces describing the job at hand (what needs to be done, who shall do it, when it needs to be done), what is known, and the work necessary to produce the plans. Plans that are complete enough and lightweight enough are the "Baby Bear" plans—they're *just right*.

The previous chapter discussed the overall workflows for the Harmony/ESW process. Figure 4.5 shows the highest-level overview of the process. Three activities are performed before software development begins: prespiral planning, developing stakeholder requirements, and defining and deploying the development environment. Initial planning takes place in the prespiral planning activity, discussed in the next section. This activity develops the initial schedule, risk management plan, team organization, model structure, and so on to enable downstream development.

5.2. Prespiral Planning

Figure 5.1 graphically shows the activities commonly performed as a part of the prespiral planning activity. The fact that these are shown as executing in parallel implies three things:

• Parallel activities may be done by different people.

• Parallel activities may be done in any order with respect to each other.

• In some cases, parallel activities need not be performed.

First, the activities may be done by different people. After all, the skill sets for creating a good schedule are different from those needed to plan for risk reduction or to perform the initial safety and reliability analysis. Second, the order in which these tasks are performed isn't really important. In fact, some of these tasks must be done concurrently because they share information. For example, a risk management plan identifies some work tasks to reduce risks that must show up in the schedule; thus, the scheduling task and the risk reduction planning task must interoperate and share information. Last, not all of the tasks need to be performed at all. For example, the initial safety and reliability analysis is really relevant only for safety-critical and high-reliability product development. Similarly, not every project or company wants to pay to create reusable assets. Projects that are highly familiar and introduce no significant risks may eschew the creation of a risk management plan.

The tasks in the prespiral planning activity include:

• Creating the schedule

• Creating the team structure

• Planning for reuse

• Planning for risk reduction

• Specifying the logical architecture (project structure)

• Performing the initial safety and reliability analysis

Each of these topics will be discussed in turn.

5.2.1. Creating the Schedule

Creating a schedule is always an exercise in estimating stuff you don't know, and this is especially true in software development. Nevertheless, a schedule adds important information and is as necessary for business planning and decision making as it is for project management. A schedule is a sequence of work tasks, loaded with resources (people), resulting in usable work products, and is quantified with time, duration, and effort metadata. It is the central document for project management because it coordinates the people, tasks, and work products with dates and costs (effort). It can be used to determine whether or not the business should even undertake the project. A study a few years ago found that over 30% of all software projects are abandoned prior to delivery. I believe that many of them were abandoned when (or usually well after) it became apparent that it wasn't justifiable to complete the project based on expected return. Of course, for those projects, it is even better to scuttle the project before significant work has been done, but to do that requires an accurate schedule.

That being said, one of the problems with scheduling is that many managers assume infinite fidelity of information and infinite scalability of the work tasks. They don't bat an eye when the scheduling tool announces, down to the minute, when a five-year project will be complete. It doesn't seem odd to them that if I take a one-person-year project and assign 16,000 people to it I come up with a delivery date of one minute from now. Sadly, it seems that common sense isn't really all that common.

The Harmony/ESW process has two related workflows:

• Bruce's Evaluation and Review Technique (BERT) constructs a schedule from estimates in a particular way.

• The Effect Review for Nanocycle Iteration Estimation (ERNIE) tracks estimates against actuals and provides feedback to improve the schedule and estimation accuracy.

Tracking and controlling the project are dealt with in detail in Chapter 9, "Agile Process Optimization," but I will discuss the scheduling aspects in this section.

Before we can construct a schedule, we've got to have some idea of what product is being built, what its primary features are, and what resources we can bring to bear during the project. The initial schedule must take into account the product vision, stakeholder requirements, and identified risks and RMAs. The basic steps involved in scheduling are

1. Identifying the desired functionality

2. Identifying the key risks and RMAs

3. Planning the set of microcycles and resulting prototypes

4. Evaluating the schedule

5. Reworking the schedule until it is acceptable

Later refinement of schedules, done in conjunction with detailing the system requirements, will detail the plans within each microcycle iteration. However, we don't have enough information in prespiral planning to have any realistic chance of doing a good job of that, so the detail will be added later. But before we talk about constructing the schedule, let's talk about how to construct estimates.

Bruce's Evaluation and Review Technique (BERT)

Software developers are notorious for their inability to estimate how long their work will take. Really. There are lots of reasons for this (see Chapter 1, "Introduction to Agile and Real-Time Concepts"), but despite the difficulties, there are real benefits from accurate estimates and schedules. BERT is the approach that I've developed to provide me with accurate estimates from which I can construct reasonable schedules.

Schedules are sequences of work tasks, called **work items** (formerly known as **estimable work units**, or EWUs, but that's too hard to pronounce). Ideally, these are small—one to a few days in length—although for early schedules they tend to be much larger. The most important single estimated value is the "50th-percentile estimate," abbreviated as $E_{50\%}$. This is the amount of effort the task is estimated to require 50% of the time; that is, half of the time the developer will finish early, and half of the time the developer will finish late. The central limit theorem of statistics shows that a schedule composed of $E_{50\%}$ estimates will end up on time overall.

However, that's not the only value you need. You also need a measure of the degree of certainty inherent in the measure. For this reason, I also ask my engineers to provide me with the pessimistic estimate that they will able to meet 80% of the time ($E_{80\%}$) and the optimistic estimate that they will be able to meet only 20% of the time ($E_{20\%}$). The difference between these latter two estimates is a measure of the perceived degree of certainty of $E_{50\%}$.

Not all engineers are equally accurate in their estimations. We also need a measure of the accuracy of the estimator. Over time, this value can be computed by comparing $E_{50\%}$ against the actual effort required for a set of estimates. The estimator accuracy is called the estimator confidence factor (E_c). Ideal estimators will have an E_c value of 1.0. Engineers who estimate consistently too low will have an E_c greater (often significantly greater) than 1.0. Some managers achieve similar (but less accurate) results by multiplying all estimates by a "fudge factor" of 2 or 3. Others go to the next level of unit (two hours is changed to two days, three days becomes three weeks, and so on). I prefer to use the demonstrated historical accuracy of each estimator. The construction of the E_c is discussed along with the ERNIE process, later in this chapter.

Once these values have been quantified, we can construct the most accurate estimate to be used in the primary schedule (E_{used}):[1]

1. Remember the mantra "Math is my friend. Math is my friend..."

Equation 5.1 *Constructing the used estimate*

$$E_{used} = \frac{E_{20\%} + 4E_{50\%} + E_{80\%}}{6} E_c$$

Astute (and mathematically inclined) readers will note that the quotient is simply a linear approximation to a Gaussian distribution curve. This computed estimate takes into account the mean value, the shape of the distribution, and the historical accuracy and bias of the estimator.

Interestingly, this works out to be mathematically equivalent to the approach suggested by Tom DeMarco and Timothy Lister.[2] They argue that estimates are usually composed of $E_{0\%}$—that is, a minimal value that can never actually be achieved—and they provide guidance on adding realism to such estimates.

2. Tom DeMarco and Timothy Lister, *Waltzing with Bears: Managing Risk on Software Projects* (New York: Dorset House Publishing, 2003).

For early schedules, the BERT method is applied at the use case level, where the use cases are identified from the stakeholder requirements. These are, of course, much larger than one or two days in length, but not as large as you might think. Remember from the previous chapter that a single microcycle normally implements anywhere from two to seven (an average of three for a 10-person team) use cases within a four- to six-week period. That includes detailed system requirements specification, construction of the analysis model, optimization during design, and validation. Since an average team includes fewer than 10 people, the scope for a typical use case is then about 10 person-weeks per use case. Use cases obviously larger than this can be decomposed into “part” use cases of this scale with the `«include»` dependency. For example, on one project, the project team decomposed the use case `Track Tactical Objects` into four use cases, each done in a subsequent microcycle.

In later, more detailed scheduling, we want to use smaller work items, such as individual sequence diagrams or parts thereof. However, in the full spiral Harmony microcycle, the system requirements are not detailed until the iteration in which those requirements are realized, so the detailed planning must wait until we have more information.

Story Points

An interesting, if qualitative, approach is suggested by Mike Cohn.[3] He recommends the assignment of arbitrary unit “story points” to the estimate of work items. His statement is that it doesn’t matter how the points map to hours or days as long as they are consistent among all the estimated items; that is, a 10-point work item should take twice as long as a different, 5-point work item. While that has some merit, ultimately we need to assign real resources to the job and real budgets to the project, so the point-to-time mapping must eventually be made.

3. Mike Cohn, *Agile Estimation and Planning* (Upper Saddle River, NJ: Prentice Hall, 2005).

The advantage of story points is that they model effort in relative terms. This makes them useful for estimating the effort of one use case or feature with respect to another. It is also an easy way to track progress as story points per unit time. This is known as **project velocity.** This approach has proven useful in IT environments where there is no "delivered system" that is packaged and handed off to manufacturing; instead, the IT infrastructure is updated in an incremental way.

The disadvantage of story points is that they model effort in relative terms. In most real-time and embedded projects, time and cost estimates must be constructed for management to make intelligent decisions about starting or bidding on a project. These projects ultimately deliver a system to manufacturing or to a customer at a (planned) point of time for a (planned) cost. To make good business decisions, absolute measures are required.

Use Case Points

Use case points are a size *x* complexity estimator, similar to the COCOMO technique championed by Barry Boehm.[4] In both approaches project effort is estimated by guessing the size and complexity of the overall project. My experience is that a good estimator can do a better job than COCOMO, but some people swear by it. The primary use to which I put such estimation techniques is as a secondary check of the schedule length I come up with via the BERT method. The use case points method is similar to COCOMO but formulates the problem in terms of use cases.

4. Barry Boehm et al., *Software Cost Estimation with COCOMO II* (Upper Saddle River, NJ: Prentice Hall, 2000).

The steps involved in applying use case points are the following:

1. Determine the number and complexity of each actor (environmental object).

2. Determine the number and complexity of each use case.

3. Adjust for technical and project complexity factors.

4. Adjust for environmental complexity factors.

5. Compute adjusted use case points.

6. Apply project- or company-specific scaling factors to computed predicted hours.

Step 1: Determine the Number and Complexity of Each Actor (Environmental Object)

An actor is an object outside the scope of the system being built that interacts with it in important ways. This step determines the **unadjusted actor weight**—the sum of complexity weights of the actors. Typically, actor complexity is a value in the range of 1 to 3:

• *Simple* (value = 1)—used for simple message exchange with the system, such as a human actor pushing a button or a memory-mapped actuator interface being set to a value

• *Moderate* (value = 2)—used for simple or predefined and purchased communication protocols, such as if you construct a simple serial data link protocol or purchase a commercial TCP/IP stack

• *Complex* (value = 3)—used for complex interactions such as when you must construct a GUI or use middleware such as CORBA or DDS

Equation 5.2 shows the computation of the actor weight for *j* actors.

Equation 5.2 *Unadjusted actor weight*

$$ActorWeight_{unadjusted} = \sum_{j} ActorComplexity_{j}$$

Step 2: Determine the Number and Complexity of Each Use Case

Step 2 computes the weight of each use case against a measure of the complexity of the use case. This is also typically divided into three levels of complexity:[5]

5. The actual method uses extremely tiny use cases instead of more reasonably sized ones. The complexity factors are computed with 1 to 5 messages for simple complexity, 6 to 10 for moderate, and more than 10 for complex. Since these are unreasonably tiny, I've upped the

ante by using similar values but referring to scenarios instead of messages to compensate. This is my experience based on the use of agile model-based development, which improves the developers' efficiency significantly.

• *Simple* (value = 5)—used when the use case is captured in 1 to 5 scenarios, each of a short (fewer than 10 messages) length

• *Moderate* (value = 10)—used when the use case is captured in 6 to 15 scenarios, each of which is moderate (10 to 20 messages) in length

• *Complex* (value = 15)—used when the use case is captured in more than 20 scenarios of moderate or greater length, or when sequence diagram decomposition is required

Of course, more divisions may be used if necessary. Equation 5.3 shows the computation for *k* use cases.

Equation 5.3 *Unadjusted use case weight*

$$UseCaseWeight_{unadjusted} = \sum_k UseCaseComplexity_k$$

The unadjusted use case points are then computed by adding the actor weights to the use case weights, as shown in Equation 5.4.

Equation 5.4 *Unadjusted use case points*

$$UCP_{unadjusted} = ActorWeight_{unadjusted} + UseCaseWeight_{unadjusted}$$

The unadjusted use case points are a measure of the size of the project. Now we have to adjust for its complexity. We do so for both environmental factors and for technical factors. In both cases, we assess the project against the degree to which the complexity factor is true and multiply that value by a measure of how much work that factor entails (its "pain-in-the-butt" value).

Step 3: Adjust for Technical and Project Complexity Factors

There are a number of technical factors, as shown in Table 5.1. The technical adjustment factor is the sum of the products of the technical factor weights and the degree to which those factors appear, as shown in Equation 5.5.

Table 5.1 *Technical Adjustment Factors*

Technical Factor	Weight	Assigned Degree (0–5)	Adjusted Factor
Distributed system	2	x	$2x$
Real-time/performance-critical	1	x	x
End user efficiency	1	x	x
Complex internal process	1	x	x
Reusability of code/model	1	x	x
Ease of use required	0.5	x	$\frac{x}{2}$
Ease of installation	0.5	x	$\frac{x}{2}$
Portability	1	x	x
Ease of maintenance	1	x	x
Concurrency	1	x	x
Security issues	1	x	x
Direct access for third parties	1	x	x
Special user training requirements	1	x	x

Equation 5.5 *Technical factor*

$$T_{factor} = \sum_{m} TWeight_m \times TDegree_m$$

The actual adjustment is then computed with an equation constructed by performing a linear regression of the actual effects found by observing real projects, as shown in Equation 5.6.

Equation 5.6 *Computed technical complexity*

$$Complexity_{technical} = 0.1 \times T_{factor} + 0.6$$

Step 4: Adjust for Environmental Complexity Factors

The other complexity factor requiring attention is the complexity of the environment. Table 5.2 shows the weighting for the environmental factors.

Table 5.2 *Environmental Complexity*

Environmental Factor	Weight	Assigned Degree (0–5)	Adjusted Factor
Familiarity with project model	1.5	x	$1.5x$
Domain experience	0.5	x	$\frac{x}{2}$
UML experience	1	x	x
Lead analyst capability	0.5	x	$\frac{x}{2}$
Motivation	1	x	x
Stable requirements	2	x	$2x$
Part-time staff	−1	x	$-x$
Difficult programming language	−1	x	$-x$

The environmental factor and associated environmental complexity are calculated in a similar fashion to the technical complexity. The environmental factor, like the technical factor, is merely the sum of the products of the severity of the factor and the degree to which it is true, as shown in Equation 5.7.

Equation 5.7 *Environmental factor*

$$E_{factor} = \sum_{n} EWeight_n \times EDegree_n$$

The environmental complexity is computed by applying a different formula, which was derived in the same way (see Equation 5.8).

Equation 5.8 *Environmental complexity*

$$Complexity_{env} = -0.03 \times E_{factor} + 1.4$$

Step 5: Compute Adjusted Use Case Points

The next step is computationally simple: The value of the adjusted use case points is the product of the unadjusted use case points, the actor complexity, and the environmental complexity. This is shown in Equation 5.9.

Equation 5.9 *Adjusted use case points*

$$UCP_{adjusted} = UCP_{unadjusted} \times Complexity_{technical} \times Complexity_{env}$$

Step 6: Apply Project- or Company-Specific Scaling Factors to Computed Predicted Hours

The last step is where "the miracle" occurs. Computationally, this is very easy. Simply multiply the efficiency by the number of adjusted use case points, and *voilà!*

To compute the hours required (Equation 5.10), the average developer efficiency in hours per adjusted use case point is needed. Ideally, this value should be based on the historical data for the project team or at least the company. That generally means that the first time you use it, you must guess at the complexity. Gustav Karner, the developer of the approach at Rational (now IBM), uses 20 development hours per adjusted use case point. Gautam Banerjee reports good results with efficiency values ranging from 15 to 30.[6]

6. Gautam Banerjee, "Use Case Points: An Estimation Approach" (August 2001), at www.bfpug.com.br/Artigos/UCP/Banerjee-UCP_An_Estimation_Approach.pdf.

Equation 5.10 *Estimated hours*

$$Estimate_{hours} = Efficiency\left[\frac{hours}{UCP_{adjusted}}\right] \times UCP_{adjusted}$$

All this may seem complex, but it is very easy to put it into a spreadsheet to calculate the project estimate. That's the good part.

The bad part is that the method is very sensitive to initial conditions, including use case and complexity weighting factor assessments. All published reports with this method that I've found have used unusually tiny use cases consisting of a single scenario with only a few messages, unlike real use cases that often contain dozens of complex scenarios.

It is possible to estimate your efficiency (so that you can estimate your software project) even when previous similar projects weren't done using use cases. But it will require some work. Go back through a small set of projects for which you have the overall effort data and the requirements documents. Do a use case analysis on each project, and use Equation 5.10 to back-estimate the efficiency from the computed adjusted use case points and the actual effort expended.

The bottom line is that the technique may add some value for checking your overall schedule, but not until you have gathered some historical data to compute your group's efficiency.

Constructing the Schedule

It is important to understand why we have a schedule and use this as guidance in its construction. The main reason is that we want to be able to accurately predict how much cost (mostly in terms of effort) and how much time a project will take. This allows for strategic business planning, such as whether or not to even build this product and how the launch will fit into the larger business picture. It also allows for tactical planning, such as when we need to make resources available, when to launch marketing campaigns, how much money we need to allocate to the project, and when manufacturing must be able to accept the implementations. To achieve this purpose, the schedule must be accurate and realistic.

Another use for a schedule is to inform customers when they can expect to see the product. In some environments customers are more forgiving than in others, but in my experience, it is best to have a tradition of meeting or exceeding customer expectations. To achieve this goal, the schedule must be pessimistic, so that most of the time the customer's expectations will be met or exceeded.

Some managers believe that a schedule can serve another purpose: to provide a sense of urgency to the developers. Clearly, reason such managers, the developers are slacking off, so they need a little whipping to motivate them. This is the purpose of the "motivational schedule." This requires the schedule be "aggressive" (i.e., unrealistic) in the hope that the developers will devote extra effort (e.g., overtime) to make it happen. There is nothing wrong with having a "stretch goal" schedule, but you must keep in mind that it is, by definition, *not likely to occur* and should not therefore be used as the basis for planning.

The same schedule cannot be used for all three purposes. To this end, we will construct multiple schedules, one for each of the goals.

The primary or working schedule is constructed by scheduling the microcycles in sequence or in parallel, depending on the assigned resources. The estimates used to construct the initial version of the schedule will be solely use-case-based (or feature-based), where a use case is completely realized within a single microcycle, but possible multiple use cases may be so realized—again depending on use case size and available resources. The E_{used} estimates are used for the primary schedule since this is simply the most accurate set of numbers available.

But, while the working schedule is the most accurate, it means that fully half of the time we will be late (although we'll be early half of the time as well). I propose, therefore, the construction of a second schedule, called the "customer schedule." This is composed of the $E_{80\%}$ estimate (specifically, $E_{80\%} \times E_c$ for each estimate). This schedule is the one shared with the customer. It will be met 80% of the time, resulting in generally pleased customers.

On the other hand, there's nothing wrong with having an aggressive goal, as long as you keep $firmly$ in mind that it is a goal that isn't likely to be met. This "goal schedule" is composed of the optimistic estimates times the estimator confidence factor, that is, $E_{20\%} \times E_c$. It is perfectly reasonable to offer incentives to meet such a schedule, such as bonuses and the like, but it is important to remember that this schedule will be met only 20% of the time. What you want to avoid at all costs is using the goal schedule as the actual schedule. As mentioned, many managers do this in a misguided attempt to create a sense of urgency in the developers but then later somehow forget that the schedule wasn't realistic in the first place—that's why we call it the goal schedule and not the working schedule.

A comment I often get when proposing this is "But we have to bid on the project; if our bid is too high (with the customer schedule), even if it is correct, we won't get the work." To this comment I have two replies. First, if it costs you more to develop the product than you get paid, then it is false economy to win the bid for a too-low value. Second, if the contract allows for cost overruns, then you can underbid the contract.

Remember, we provide the customer schedule to customers to allow them to plan to receive the product; we work against the working schedule because that's the most accurate number we have. We provide incentives to meet the goal schedule because there are usually business advantages to coming in early.

Effect Review for Nanocycle Iteration Estimation (ERNIE)

I believe that one of the biggest reasons that, as an industry, we're so terrible at estimation is that people make estimates and never go back to see how they did against those predictions. As DeMarco and Lister point out,[7] you can't track what you don't measure. And you can't improve what you don't track. The ERNIE method is used to track work actuals against predicted (scheduled). The steps involved in the ERNIE method are simple enough:

• Use BERT to construct estimates.

• Track all estimates against complete actuals.

• Periodically recompute the estimator confidence factor, E_c.

• Reward estimation accuracy over optimism.[8]

• Accept realistic, even if undesirable, estimates.

7. DeMarco and Lister, *Peopleware*.

8. Remember the Law of Douglass (there are many) that states, “Optimism is the enemy of realism.”

When I manage projects, I have my staff track their own estimation accuracy, but the project manager can do this as well. I recommend a spreadsheet such as that shown in Table 5.3.

Table 5.3 *Estimation Tracking Spreadsheet (in hours)*

Date	Task	$E_{20\%}$	$E_{50\%}$	$E_{80\%}$	Unadjusted Used	E_c	E_{used}	Actual	Dev.	% Diff
9/15/09	UI	21	40	80	43.5	1.75	76.1	57	17	0.425
9/17/09	Database	15	75	200	85.8	1.75	150.2	117	42	0.56
9/18/09	Database conversion	30	38	42	37.3	1.75	65.3	60	32	0.842
9/20/09	User manual	15	20	22	19.5	1.75	34.1	22	2	0.1

The estimator confidence factor should be recomputed periodically, once the estimator has made enough estimates and worked enough of those work items to make a reasonable assessment of his or her accuracy (minimum four). This calculation is shown in Equation 5.11 for *n* estimates.

Equation 5.11 *Recomputing E_c*

$$E_c = 1 + \frac{1}{n}\sum_{j=1}^{j=n}\left[\frac{Actual_j - EstimatedMean_j}{Actual_j}\right]$$

Computing a new E_c value from Equation 5.11, we use the value from the “Actual” column and subtract the value from the E column (since that’s the single value we’re trying to optimize)

50%

and get

$$E_c = 1.0 + (0.425 + 0.56 + 0.842 + 0.1)/4 = 1.48$$

In this example, the engineer went from an estimator confidence factor of 1.75 to a factor of 1.48 (a significant improvement). This E_c value will be used to adjust the "Unadjusted Used" computed estimate for insertion in the schedule. It is important to track estimation success in order to improve it. In order to improve a thing, it is necessary to track it.

Checklist for Scheduling

There always is, or should be, a question about the reasonableness of a schedule. To address this, the Harmony/ESW process provides the following checklist:

• Evaluation of your scheduling history

° Do half of your projects come in early?

° Do half of your projects come in under planned cost?

° On average, do project costs meet plans?

° Does delivered functionality usually meet plans?

° Is it *unusual* for a project to be 10% or more over schedule or over budget?

• Evaluation of the current schedule

° Have you created all three recommended schedules?

° Does the current schedule seem feasible?

° Are the estimates made using the BERT method?

° Can the end date be met if half of the estimated work items come in late?

° Have you taken identified high-risk aspects into account in your schedule?

° Does the schedule include RMAs for the high-risk aspects of your project?

° Have you scheduled slack time?

° Do you have the resources available to perform the work items according to the plan?

° Does this schedule compare with the actual time, effort, and cost for similar projects at your company?

° Does the schedule compare favorably with secondary scheduling methods such as COCOMO or use case points?

° Does the schedule meet the customer's needs and expectations?

Schedules are crucial project work products because they provide estimates for project cost and effort and also provide a means to track the progress of the project against the plan. This is problematic because schedules are always estimates of things you don't actually know. For this reason, the schedule created here is considered an initial plan that will be updated frequently throughout the project.

5.2.2. Creating the Team Structure

There is a strong correlation between the team structure and the model organization (aka the logical model). On one hand, the teams are formed because they make coherent sense and the model is organized to allow the teams to work together effectively. On the other hand, the model is organized to optimize a number of other properties as well, which may result in shuffling teams and team members about.

It is crucial that the teams work with strong internal cohesion, and this requires them to be small and singularly focused. It is also crucial that the teams encapsulate their internal workings and provide well-structured means to coordinate with other teams. The overall project is a "team of teams,"[9] and the project manager orchestrates the teams so that they work toward a common goal, as stated in the schedule of work activities.

9. In large-system development, it may well be "teams of teams of teams..."

The ideal team size seems to be between 5 and 10 and, again ideally, each team works primarily on a single model. The interaction between teams is then done largely though the interactions of multiple models, each representing some aspect of the final product. Put another way, given a specific set of models containing the various work products of the project, each model is created and manipulated by a single team. Admittedly, some projects use smaller models and have teams work on multiple models simultaneously. This can work as well. What appears not to work very well is to have many people working on a single monolithic model.

For project success, it is important to recognize the individual differences in ability and expertise among the team members and allocate such resources where they can do the most good. This flies in the face of the “interchangeable cog” mentality that some managers apply, but managers who do take the time and effort to engage their personnel effectively tend to have much higher productivity than those who do not.

5.2.3. Planning for Reuse

Reuse can provide significant strategic advantage for a company, including lowering the cost for new products, improving their time to market, reducing resource requirements over the long term, and enabling the development of software product lines. However, the creation of reusable assets has definite tactical disadvantages as well. Specifically, there is a great deal of anecdotal evidence to suggest that creation of reusable assets carries with it on the order of three times the cost of similarly functioned purpose-built components. Technically, reuse isn’t all that difficult to achieve, but it requires a strategic focus and the acceptance of short-term costs for longer-term gain. The reason why reuse isn’t more common is simply that many businesses are unwilling to sacrifice tactical market penetration and pay higher initial project costs to achieve strategic market and project advantage over a time frame several times that. For this reason, planning for reuse is an optional step in prespiral planning. It should be undertaken when the business is willing to invest in long-term success.

The basic actions for planning for reuse include:

- Identifying reuse needs and goals

- Identifying opportunities for reuse

- Estimating the cost of constructing reusable assets

- Determining which reusable assets to construct

- Evaluating the impact of reuse on the schedule

- Specifying how reusable assets will be managed

- Specifying how existing assets will be reused in the current project and how newly constructed reusable assets will be reused in the future

- Writing the reuse plan

• Updating the schedule to reflect planned reuse

These actions culminate in the **reuse plan.** The reuse plan captures these major aspects of reuse. The reuse plan specifies “design constraints,” either on the degree of reuse of existing components, on the approach to reusing them, or on how the new design must be reusable in the future. Different levels of reuse are possible; in general, the more abstract and larger the scope, the more benefit there is to reuse. Some assets that may be reused include:

• Applications that can be plugged into different enterprise architectures

• Application frameworks that provide infrastructure for a set of applications

• Profiles that provide common types and metadata tags for similar applications

• Components that provide application building blocks

• Subsystems that provide multiple-discipline (i.e., mixed hardware/software) system-level building blocks

• Libraries that provide common services within components

• Models, containing

° Shared models (e.g., domains)

° Packages

° Model elements

° Classes

° Use cases

° State machines

° Activity diagrams

° Sequence diagrams

° Activity diagrams

° Source code

The reuse plan identifies the levels of reuse expected and the mechanisms by which the elements will be reused. In general, the creation of reusable elements is more expensive than the creation of special-purpose elements. For small-scale reuse, it appears to be around a factor of 3 more expensive. For larger-scale reuse, it can cost between 3 and 10 times the cost of a special-purpose element of equivalent behavior. However, that cost is largely a one-time cost, so with adequate reuse, the additional effort can lower long-term cost.

5.2.4. Planning for Risk Reduction

Risk reduction is a crucial aspect of project planning for most projects. Risk is the product of the severity of an undesirable situation (a hazard) and its likelihood. For projects, the most common risks have to do with adopting new technologies, using inexperienced teams, developing systems of greater size and scope, using new manufacturing or development methods, or using new and unfamiliar tools. Most projects are introducing something new in at least one of these aspects,[10] and it only makes sense to think about the things that can go wrong with the project and to plan corrective measures to address them. If your company has a history of constructing similar systems on time and on budget, and there is nothing fundamentally new in the current project, then it makes sense to skip this step.

10. Otherwise, the company could just reuse the existing system!

This task focuses on initial risk reduction planning, but it should be emphasized that risk reduction is an ongoing task that occurs through most or all of the project. Periodically, risks will be explicitly reevaluated and explored to ensure that the project can proceed successfully.

The steps involved in risk reduction planning include:

• Identifying key project hazards

• Quantifying hazard severity

• Determining the likelihood of these key project hazards

• Computing the project risks

• Ranking the project hazards in terms of risk

• Specifying RMAs for key project risks

• Writing the risk management plan

The project hazards are, again, undesirable conditions, such as:

• Selected middleware (e.g., CORBA) that is too large or too slow

• Staff turnover resulting in the loss of key personnel

• Loss of project funding

• New tools (e.g., compilers) that contain defects

• Lack of familiarity with new languages (e.g., UML or a new source-level language)

• Lack of efficiency with new tool sets (e.g., changing CM tools)

• Lack of team motivation

• Lack of management commitment

• Lack of team availability due to company imperatives

• Customer requirements that don't stabilize long enough to be met

• A schedule that is unrealistic at the outset

• Crucial vendors (e.g., for memory boards or CPUs) that stop producing required parts

• Partners or subcontractors that fail to meet objectives

• Loss of backups due to on-site storage concurrent with a fire

Obviously, depending on the system, some of these hazards are worse than others. Hazards that are quite severe, even if relatively unlikely, may warrant special attention. Furthermore, hazards that aren't that severe but are highly likely may be equally worrisome. It is the product of the severity and the likelihood of the hazard that constitutes the risk to the project that must be considered. I generally rank the severity in a range from 0 (no impact) to 10 (catastrophic impact), and the likelihood in a range from 0.0 (impossible) to 1.0 (certain). In some cases, the likelihood can be measured from previous projects, but often it can only be estimated.

Once each project hazard is quantified in terms of severity, likelihood, and risk, then the hazards can be ranked in order of risk. The *key risks* are the set of risks above a specified threshold. This threshold is set depending on the sensitivity of the project to risk. For a pilot project, the threshold might be set quite high—say, at 9.0. For a make-or-break-the-company project, the threshold might be set quite low—perhaps at 5.0. Most projects are somewhere in between.

RMAs (risk mitigation activities) are work efforts designed to reduce the risk. For example, if you're not sure that CORBA is fast enough, early on, perhaps in one of the first three prototypes, you would write a performance-critical flow to use the communication infrastructure (including CORBA) so that the speed can be measured. For new and unfamiliar technologies, pilot projects or training can be scheduled. For all key risks, such RMAs must be identified *and scheduled*. I recommend that this be done, to the degree practical, high-risk first. The highest-risk items are the most likely to have a severe detrimental effect on project success. Lower-risk mitigation activities can be done later, once the higher risks have been dealt with.

The **risk management plan** brings together the risk details, including the hazard, likelihood, severity, computed risk, RMAs, responsible party, when it is planned to be addressed, and resolution. This is a living document, updated throughout the project and reevaluated at least once per microcycle, during the increment review (party) activity.

5.2.5. Specifying the Logical Architecture

What we mean by the term **logical architecture** in the Harmony/ESW process is the principles, practices and work products for organizing the things that exist at *design time;* that is, for model-based development projects, logical architecture is the same thing as *model organization*. We have a great deal of experience in model organization principles that work and don't work. Model organization is initially constructed in conjunction with team organization in prespiral planning. The models may require some reorganization later, and the same is true of the teams, but it is common for a good initial model organization to continue throughout a development project and subsequent product evolution cycles.

In simple-enough systems, you can pretty much do anything you like and still succeed. Once you have a system complex enough to require more than one person, then it begins to matter what you do and how you do it. Once there are *teams* of people in place, it matters a great deal how the work is organized for the teams to effectively work together.

Why Model Organization?

The reasons for worrying about model organization are to:

• Allow team members to contribute to the model without losing changes made by other team members, or in some other way corrupting your model

• Allow team members to use parts of the model to which they need access but which they are not responsible for developing

• Provide for an efficient build procedure so that it is easy to construct the system

• Be able to locate and work on various model elements

• Allow the pieces of the system to be effectively reused in other models and systems

At first blush, it may appear that the first two issues—contributing and using aspects of a common model—are dealt with by CM. This is only partially true. CM does provide locks and access controls and can even perform model-based merging for concurrent development. However, this is a little like saying that C solves all your programming problems because it provides basic programmatic elements such as assignment, branching, looping, and so on. CM does not say anything about what model elements ought to be CIs, or when, or under what conditions. Effective model organization uses the project CM infrastructure but provides a higher-level set of principles that allow the model to be used effectively.

The UML provides two obvious organizational units for CIs: the **model** and the **package.** A UML package is a model element that "contains" other model elements. It is essentially a "bag" into which we can throw model elements that have some real semantic meaning in our model, such as use cases, classes, objects, types, functions, variables, diagrams, and so on. However, UML does not provide any criteria for what should go into one package versus another. So while we might want to make packages CIs in our CM system, this begs the question as to what policies and criteria we should use to decide how to organize our packages—what model elements should go into one package versus another.

One simple solution would be to assign one package per worker. Everything that Sam works on is in SamPackage, everything that Julie works on is in JuliePackage, and so on. For very small project teams, this is, in fact, a viable policy. But again, this begs the question of what Sam should work on versus Julie. It can also be problematic if Susan wants to update a few classes of Sam's while Sam is working on some others in SamPackage. Further, this adds artificial dependencies of the model structure on the project team organization. This will make it more difficult to make changes to the project team (say, to add or remove workers) and will really limit the reusability of the model elements.

It makes sense to examine the user workflow when modeling or manipulating model elements in order to decide how best to organize the model. After all, we would like to optimize the workflow of the users as much as possible, decoupling the model organization from irrelevant concerns. The specific set of workflows depends, of course, on the development process used, but there are a number of common workflows:

• Requirements

° Working on related requirements and use cases

• Detailing a use case

• Creating a set of scenarios

• Creating the specification of a use case via state machine or activity diagram

• Mapping requirements (e.g., use cases) to realizing model elements (e.g., classes)

• Realizing requirements with analysis and design elements

° Elaborating a collaboration realizing a use case

° Refining collaborations in design

° Detailing an individual class

• Designing the architecture

° Logical architecture—working on a set of related concepts (classes) from a single domain or subject area, defining a common area for shared types and data structures, and so on

° Physical architecture—working on a set of objects in a single runtime subsystem or component

• Construction and testing

° Translation of requirements into tests against design elements

° Execution of tests

° Constructing the iterative prototypes from model elements at various stages in the project development

• Planning

° Project scheduling, including work products from the model

When working on related requirements and use cases, the worker typically needs to work on one or more related use cases and actors. When detailing a use case, a worker will work on a single use case and detailed views—a set of scenarios and often either an activity diagram or a state machine (or some other formal specification language). When elaborating a collaboration, the user will need to create a set of classes related to a single use case, as well as refine the scenarios bound to that use case. These workflows suggest that one way to organize the requirements and analysis model is around the use cases. Use packages to divide the use cases into coherent sets (such as those related by generalization, `«include»`, or `«extend»` relations, or by associating with a common set of actors). In this case, a package would contain a use case and the detailing model elements: actors, activity diagrams, state machines, and sequence diagrams.

The next set of workflows (realizing requirements) focuses on classes, which may be used in either analysis or design. A **domain** in the Harmony/ESW process is a subject area with a common vocabulary, such as UI, device I/O, or alarm management. Each domain contains many classes and types, and system-level use case collaborations will contain classes from several different domains. Many domains require rather specialized expertise, such as low-level device drivers, aircraft navigation and guidance, or communication protocols. It makes sense from a workflow standpoint (as well as a logical standpoint) to group such elements together because a single worker or set of workers will develop and manipulate them. Also, this simplifies the use of such model elements because they often must be reused in coherent groups. Grouping classes by domains and making the domains CIs may make sense for many projects.

Architectural workflows also require effective access to the model. Here, the architecture is broken up into the logical architecture (organization of types, classes, and other design-time model elements) and the physical architecture (organization of instances, objects, subsystems, and other runtime elements). The logical architecture is to be organized first by models and second by packages (such as those representing domains and the subsystem architecture). If the model is structured this way, then each domain, subsystem, or component is made a CI and assigned to a single worker or team. If the element is large enough, then it may be further subdivided into subpackages of finer granularity based on subtopic within a domain, subcomponents, or some other criterion such as team organization.

Testing workflows are often neglected in the model organization, although usually to the detriment of the project. Testing teams need only read-only access to the model elements under test, but nevertheless they do need to somehow manage test plans, test procedures, test results, test scripts, and test fixtures, often at multiple levels of abstraction. Testing is often done at many different levels of abstraction but can be categorized into three primary levels: unit testing, integration, and validation. Unit-level testing is usually accomplished by the owner of the model element under test or a “testing buddy” (a peer in the development organization). The tests are primarily white-box, design, or code-level tests and often use additional model elements constructed as test fixtures. In the Harmony/ESW process, these tests are constructed and applied throughout the development of the model. It is important to retain these testing fixture model elements so that as the system evolves, we can continue to test the model elements. Since these elements are white-box and tightly coupled with the implementation of the model elements, it makes the most sense to colocate them with the model elements they test. So, if a class `myClass` has some testing support classes, such as `myClass_tester` and `myClass_stub`, they should be located close together.

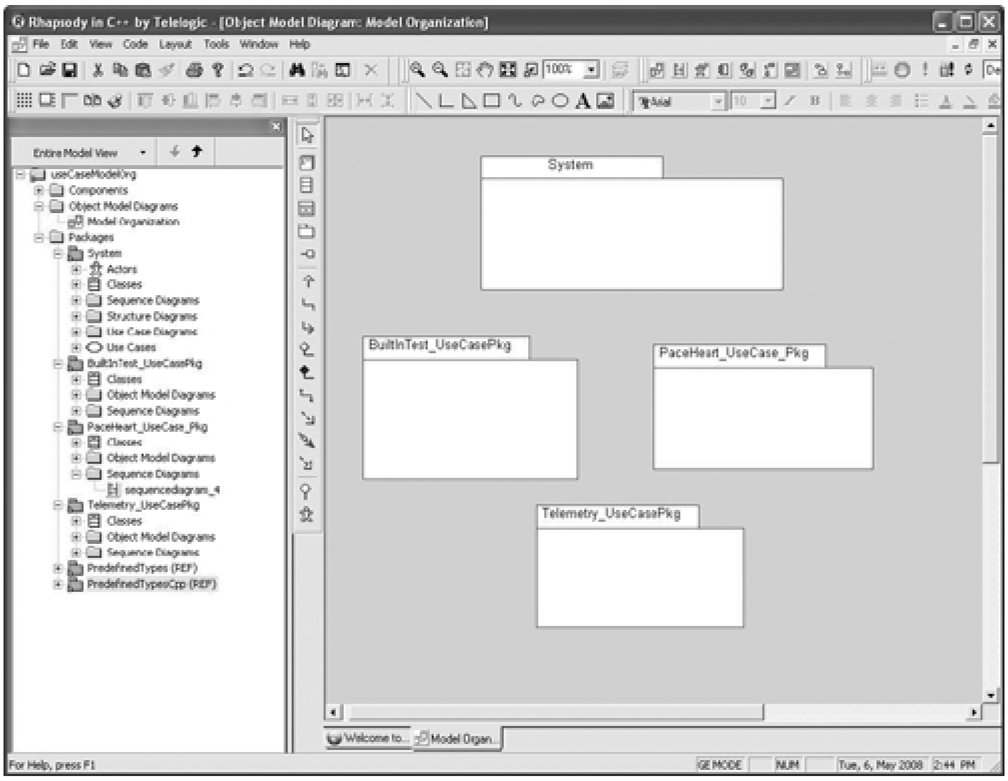
Integration and validation tests are not so tightly coupled as at the unit level, but clearly the testing team may construct model elements and other artifacts to assist in the execution of those tests. These tests are typically performed by *different workers* from the creators of the model elements they test. Thus, independent access is required, and they should be in different CIs. Harmony/ESW's continuous integration is done at least daily during the object analysis and various design phases. The tests the build is required to pass before general release to the baseline (and the other workers) are created and updated by the configuration manager as new functionality is released to the baseline. Validation is done toward the end of each microcycle, usually by a separate team of testers. These tests are almost entirely "black-box" tests and are based on the validation tests for previous microcycles (regression testing) and the use case analysis of the current microcycle.

It is important to be able to efficiently construct and test prototypes during the development process. This involves tests both against the architecture and against the entire prototype's requirements. There may be any number of model elements specifically constructed for a particular prototype that need not be used anywhere else. It makes sense to include these in a locale specific to that build or prototype. Other artifacts, such as test fixtures that are going to be reused or evolved and that apply to many or all prototypes, should be stored in a locale that allows them to be accessed independently from a specific prototype.

Specific Model Organization Patterns

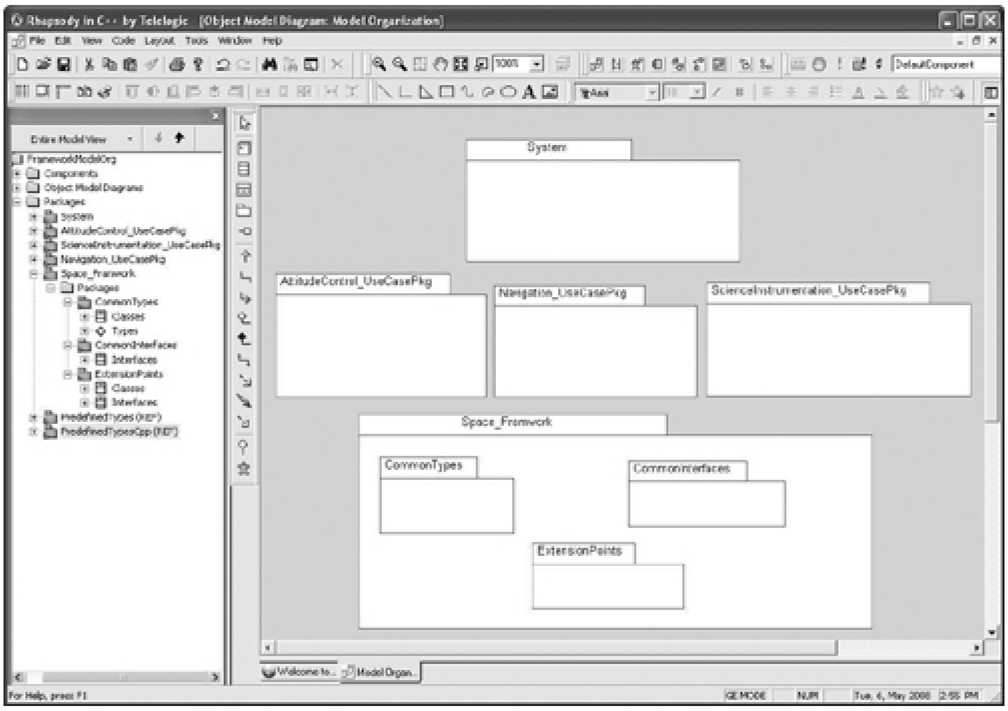
In the preceding discussion, we saw that a number of factors influence how we organize our models: the project team organization, system size, architecture, how we test our software, and our project lifecycle. Let us now consider some common ways to organize models and see where they fit well and where they fit poorly. The model organization shown in Figure 5.2 is the simplest organization we will consider. The system is broken down by use cases, of which there are only three in the example. The model is organized into four high-level packages: one for the system level and one per use case. For a simple system with 3 to 10 use cases and perhaps one to six developers, this model can be used with little difficulty. The advantages of this organization are its simplicity and the ease with which requirements can be traced from the high level through the realizing elements. The primary disadvantages of this approach are that it doesn't scale up to medium or large-scale systems. Other disadvantages include difficulty in reuse of model elements and the fact that there is no place to put elements common to multiple use case collaborations, hence a tendency to reinvent similar objects. Finally, there is no place to put larger-scale architectural organizations in the model, and this further limits its scalability to large systems.

Figure 5.2 *Use-case-based model organization*



The model organization shown in Figure 5.3 is meant to address some of the limitations of the use-case-based approach. It is still targeted toward small systems but adds a framework package for shared and common elements. The framework package has subpackages for usage points (classes that will be used to provide services for the targeted application environment) and extension points (classes that will be subclassed by classes in the use case packages). It should be noted that there are other ways to organize the framework area that work well, too. For example, frameworks often consist of sets of coherent patterns; the subpackaging of the framework can be organized around those patterns. This organization is particularly apt when constructing small applications against a common framework. This organization does have some of the same problems with respect to reuse as the use-case-based model organization.

Figure 5.3 *Framework-based model organization*



As mentioned early on, if the system is simple enough, virtually any organization can be made to work. Workflow and collaboration issues can be worked out ad hoc, and everybody can get their work done without serious difficulty. A small application might be 10 to 100 classes realizing 3 to 10 use cases. Using another measure, it might be on the order of 10,000 or so lines of code.

One of the characteristics of successful large systems (more than, say, 300 classes) is that they are architecture-centric; that is, architectural schemes and principles play an important role in organizing and managing the application. In the Harmony/ESW process, there are two primary subdivisions of architecture: logical and physical.[11] The logical model organizes types and classes—things that exist at design time—whereas the physical model organizes objects, components, tasks, and subsystems—things that exist at runtime. When reusability of design classes is an important goal of the system development, it is extremely helpful to maintain this distinction in the model organization.

11. Bruce Douglass, "Components: Logical and Physical Models," *Software Development* magazine, December 1999.

The next model organization, shown in Figure 5.4, is suitable for large-scale systems. The major packages for the model are:

• System

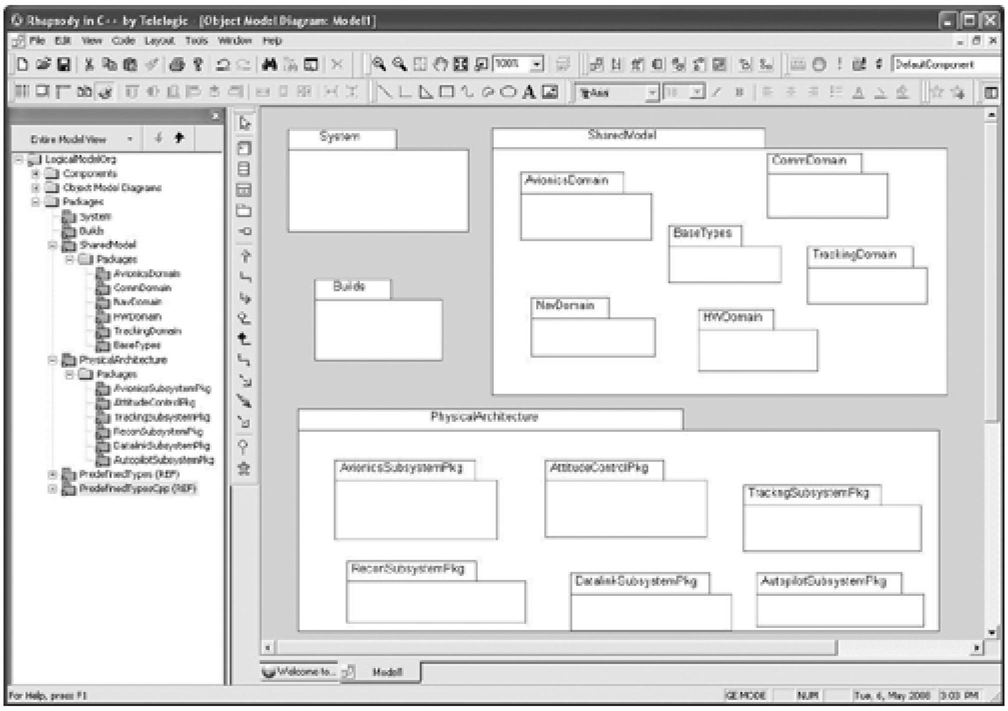
• Physical architecture model

• Subsystem model

• Shared model

• Builds

Figure 5.4 *Logical model-based model organization*



The System package contains elements common to the overall system: system-level use cases, subsystem organization (shown as an object diagram), and system-level actors. This model is often created by the systems engineers, and it models the requirements, structure, and behavior of the system as a whole.

The `SharedModel` is organized into subpackages called domains and contains elements to be shared by the subsystems. Each domain contains classes and types organized around a single subject matter, such as UI, alarms, hardware interfaces, bus communication, and so on, that is shared by two or more subsystems. Domains have domain owners—those workers responsible for the content of a specific domain. Every *shared* class in the system ends up in a single domain (elements specific to a single subsystem reside in the corresponding subsystem model). Class generalization hierarchies almost always remain within a single domain, although they may cross package boundaries within a domain.

The `PhysicalArchitecture` package is organized around the largest-scale pieces of the system: the subsystems. In large systems, subsystems are usually developed by independent teams, so it makes sense to maintain this distinction in the model. Subsystems are constructed largely of instances of classes from multiple domains but also contain elements specific to that particular subsystem. Put another way, each subsystem contains (by composition) objects instantiated from different domains in the system and objects unique to that subsystem. These packages contain the interface elements necessary to invoke the services and data owned by the subsystems, but not the design or implementation of those subsystems. That detail is located within the subsystem models.

The next major package is `Builds`. This area is decomposed into subpackages, one per prototype. This allows easy management of the different incremental prototypes. Also included in this area are the test fixtures, test plans, procedures, and so forth used to test each specific build for both the integration and validation testing of that prototype.

The `Subsystem` model (not shown in Figure 5.4) is actually a set of separate models, each of which contains the results of the analysis and design activities of a single subsystem team. It usually imports the portion of the requirements model allocated to that specific subsystem from the `System` model and uses those requirements as a specification for downstream engineering. It is in these models that the analysis and design of the system components reside.

The advantage of this model organization is that it scales up to very large systems very nicely because it can be used recursively to as many levels of abstraction as necessary. The separation of the logical and physical models means that the classes in the domains may be reused in many different deployments, while the use of the physical model area allows the decomposition of system use cases to smaller subsystem-level uses cases and interface specifications.

The primary disadvantage that I have seen in the application of this model organization is that the difference between the logical and physical models seems tenuous for some developers. The model organization may be overly complex when reuse is not a major concern for the system. It also often happens that many of the subsystems depend very heavily (although never entirely) on a single domain, and this model organization requires the subsystem team to own two different pieces of the model. For example, guidance and navigation is a domain rich with classes, but it usually also applies to a single subsystem.

Checklist for Logical Architecture

A number of aspects must be taken into account to create a good logical organization. These include the following:

• Have you taken into account the scope and scale of the project?

• Are there 10 or fewer developers allocated to work on a single model?

• Are the team members allocated to a single model colocated?

• Have you identified the elements likely to be shared among models?

• Have you defined the structure for the shared model to enable sharing of common elements (e.g., interfaces, common types, common classes)?

• Have you defined a common structure for the subsystem models?

• Have you identified a common means by which legacy components will be reused?

• Have you taken into account how each individual component will be built and tested?

• Have you taken into account how the prototypes (system builds) will be integrated and constructed?

• Have you identified the CIs (units of CM)?

• Have you identified the strategy to be used for traceability of requirements into the models?

A good logical architecture will enable smooth workflow among the developers; allow them to efficiently share elements of common interest; and support building, testing, and delivering the final system. Suboptimal logical architectures will cause developers to step on each other's work constantly, inhibiting the sharing of common elements and making it difficult to construct and test the system. It pays to give some up-front thought to how you want the teams of developers to work together. Remember, though, that this is inherently an incremental process, so the logical model can be changed over time as necessary.

5.2.6. Performing the Initial Safety and Reliability Analysis

This task within the prespiral planning activity needs to be performed only for safety-critical or high-reliability system development. When such projects are known, the inherent safety and reliability issues must be identified and addressed so that appropriate safety and reliability requirements can be identified. The steps for this task include the following:

• Identify the hazards

• Quantify the hazards in terms of likelihood and severity

• Compute the risks (likelihood × severity)

• Perform an initial safety analysis with FTA

• Perform an initial reliability analysis with FMEA

• Identify safety and reliability control measure requirements

• Create the initial hazard analysis

• Update the requirements to include safety and reliability requirements

Terms, Definitions, and Basic Concepts

Reliability is a measure of the "up time" or "availability" of a system; specifically, it is the probability that a computation will successfully complete before the system fails. It is normally estimated with MTBF. MTBF is a statistical estimate of the probability of failure and applies to stochastic failure modes. Electrical engineers are familiar with the "bathtub" curve, which shows the failure rates of electronic components over time. There is an initial high failure rate that rapidly drops to a low level and remains constant for a long time. Eventually, the failure rate rises rapidly back to the initial or higher levels, giving the characteristic "bathtub" shape. This is why electrical components and systems undergo the burn-in process. The high temperature increases the probability of failure, thereby accelerating the bathtub curve. In other words, the components that are going to fail early do so even earlier (during the burn-in). The remaining components or systems fall into the low-failure basin of the bathtub curve and so have a much higher average life expectancy.

Reducing the system downtime increases reliability by increasing the MTBF. Redundancy is one design approach that increases availability because if one component fails, another takes its place. Of course, redundancy improves reliability only when the failures of the redundant components are independent.[12] The reliability of a component does not depend upon what happens after the component fails. Whether the system fails safely or not, the reliability of the system remains the same. Clearly the primary concern relative to the reliability of a system is the availability of its functions to the user.

12. Strict independence isn't required to have a beneficial effect. Weakly correlated failure modes still offer improved tolerance to faults over tightly correlated failure modes.

Another term used loosely is **security.** Security deals with permitting or denying system access to appropriate individuals and preventing espionage and sabotage. A secure system is one that is relatively immune to attempts, intentional or not, to violate the security barriers set in place. Security is a primary aspect of the more general concern of managing information-related risks, a subject known as **information assurance.**

Safety is distinct from both reliability and security. A safe system is one that does not incur too much risk to persons or equipment. A **hazard** is an event or condition that can occur but is undesirable. Risk is defined in terms of both the severity and the likelihood of the hazard. The failure of a jet engine is unlikely, but the consequences can be very severe. Thus the risk of flying in a plane is tolerable; even though it is unlikely that you would survive a crash from 30,000 feet, it is an extremely rare occurrence. At the other end of the spectrum, there are events that are common but are of lesser concern. There is a risk that you can get an electric shock from putting a 9V battery in a transistor radio. It could easily occur, but the consequences are small. Again, this is a tolerable risk.

A risk is the chance that something bad will happen. Nancy Leveson[13] defines risk to be

13. Leveson, *Safeware*.

a combination of the likelihood of an accident and the severity of the potential consequences.

Or, more precisely:

$$risk = probability\ of\ failure \times severity$$

The "something bad" is called a **mishap** or **accident** and is defined to be damage to property or harm to persons.

Ms. Leveson goes on to define a *hazard* as:

a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object) will inevitably lead to an accident (loss event).

Hazards arise in five fundamental ways:

- Release of energy
- Release of toxins
- Interference with life support functions

• Supplying misleading information to safety personnel or control systems

• Failure to alarm when hazardous conditions arise

The unsafe release of energy is perhaps the most common threat. Energy occurs in many forms, including chemical, electrical, atomic, potential, and kinetic. Chemical energy can result in fires and explosions. Electrical energy can result in electrocutions. Atomic energy can lead to immediate death or agonizing radiation sickness. Airplane crashes release enormous kinetic energy. Anytime a large amount of energy is being controlled, protecting humans from its inadvertent sudden release is the primary safety concern.

The release of toxins is important in many environments today, particularly in the medical and chemical industries. Many of the worst catastrophes in recent times have been caused by the release of toxins. The Bhopal chemical accident released a huge cloud of methyl isocyanate from a Union Carbide plant, killing at least 2000 people in the surrounding area.

Incidents can also occur when the system interferes with a process necessary to sustain life. The most obvious examples are with medical products, such as patient ventilators and heart-lung machines. However, life support systems also maintain human-compatible environments in airplanes, submarines, and spacecraft. Even the failure of a thermostat in a sufficiently harsh environment[14] could contribute to the loss of life.

14. Northern Minnesota comes to mind.

Many safety-critical systems are continuously or periodically monitored by humans and include humans within the safety loop. A monitoring system that actively misleads a human contributes to a hazardous condition. Imagine an ECG monitor that displays a cardiac waveform but, because of a software defect, displays the same waveform data repeatedly, ignoring new data. The patient could enter cardiac arrest but the trusted ECG system would display only old data. An attending physician might well conclude that the patient is fine because the supposedly real-time data shows a good patient condition. Offline diagnostic systems can also contribute to an unsafe condition. Many deaths in hospitals occur because of mislabeling of patient medications and laboratory results. These errors actively mislead the personnel responsible for safety-related decisions and contribute to the resulting incidents.

The previous example dealt with actively misleading the human in the loop. Passive systems that fail to alarm can be just as deadly. Failure of pressure and temperature alarms in a nuclear power plant can lead to reactor leaks. Many people fail to recognize that the reverse is also true—too many “nuisance” alarms can hide important safety-related conditions. Exactly this kind of problem contributed to the Therac-25 incidents,[15] where the system falsely alarmed frequently and reported obscure error codes. The operators soon learned to ignore the nuisance alarms and to silence them immediately. Additionally, the Therac-25 system made no distinction between critical and routine alarms. Too many false alarms frequently led to operators disabling alarms altogether.

15. The Therac-25 story is of a radiation treatment device whose malfunction killed a number of patients. Read the complete story in Leveson, *Safeware*.

In some systems, the number of alarms can be so great that responding to them takes all the operator's attention, leaving none to deal with the underlying problems. Alarms with cryptic messages can be worse than having no indications at all—they do not suggest appropriate corrective action yet distract the operator from his or her safety-related tasks.

Safety-Related Faults

Hazards can occur either because a system was designed to unsafe specifications or because of faults, that is, the nonperformance of a system to achieve its intended function within its specifications. Faults are normally categorized as systematic faults (aka errors or defects)[16] or random faults (aka failures). Failures are events occurring at specific times. Errors are more static and are inherent characteristics of the system, as in design errors. A fault is an unsatisfactory system condition or state. When the fault is visible—that is, it results in a demonstrably incorrect result—then the fault is said to be *manifest*. When the fault is present but not visible, it is said to be *latent*. Thus, failures and errors are different kinds of faults and, as we will see, are generally addressed with different solutions.

16. Although some developers refer to such characteristics as *features*...

Faults can affect a system in a variety of ways:

• *Actions*—inappropriate system actions taken or appropriate actions not taken

• *Timing*—actions taken too soon or too late

• *Sequence*—actions skipped or done out of order

• *Amount*—inappropriate amount of energy or reagent used

Safety Is a System Issue

Many systems present hazards. The systems can identify and address them, or they can ignore them. However, note that safety is a “system” issue. A system can remove an identified hazard, or reduce its associated risk, in many ways. For example, consider a radiation therapy device; it has the hazard that it may overradiate the patient. An electrical interlock activated when the beam is either too intense or lasts too long is one design approach to reduce risk, also known as a **safety control measure.** The interlock could involve a mechanical barrier or an electric switch. Alternatively, the software could use redundant heterogeneous computational engines (verify the dosage using a different algorithm) to verify the setting before permitting the dose to be administered. The point is that either the *system* is safe or it isn’t. Not the software. Not the electronics. Not the mechanics. The system is safe only when the combination of such elements is safe as a unified whole. Of course, each of these impacts the system safety, but it is ultimately the interaction of all these elements that determines system safety.

Random Faults versus Systematic Faults

All types of components can contain design defects. These are errors in the component design or implementation that can lead to mishaps. However, not all can have *failures*. Notably, software does not fail. If it does the wrong thing, it will always do the wrong thing under identical circumstances. Contrast that with electrical components reaching end of life or mechanical switches breaking. The designs of such components may have been fine, but they no longer meet their design characteristics. It makes sense, then, to divide faults into systematic and random faults. Errors are systematic faults—they are intrinsic in the design or implementation. Writing the FORTRAN statement

```
DO I=1.10
```

rather than

```
DO I=1,10
```

is a transcription error[17]—an inadvertent substitution of a period for a comma. A single inadvertent semicolon has been known to bring down an entire mainframe![18]

17. One that resulted in the destruction of a Venus probe.

18. Been there, done that! Oops! ☺

The term *failure* implies that something that once functioned properly no longer does so. Failures are random faults that occur when a component breaks in the field. Failures are normally defined as arising from a stochastic process arising from a probability density function that describes their distribution.

Hardware faults may be systematic or random; that is, hardware may contain design flaws, or it may fail in the field. Random faults occur only in physical entities, such as mechanical or electronic components. End-of-life failures are common in long-lived systems, but the probability of random faults is well above zero even in brand-new systems. The likelihood of such failure is estimated from a probability distribution function, which is why they are called “random faults.” Random faults cannot be designed away. It is possible to add redundancy for fault detection, but no one has ever made a physical component that cannot fail.

Software faults are always systematic because software neither breaks down nor wears out. The problem is that the number of true states of any software application is, for all practical purposes, infinite. That means that all possible combinations of conditions cannot be fully evaluated or tested. For this reason, we advocate the concept of **defensive design.** All software services assume preconditions—passed parameters are within a valid range, variables have not been corrupted by electromagnetic interference (EMI), there is memory available to satisfy a request, and so on. In defensive design, the software service itself assumes the responsibility for validating such assumptions rather than relying on other services to always operate correctly. The resulting software detects (and responds to) precondition violations that typical software does not, and so can result in systems that are far safer and more reliable.

Many engineers routinely remove runtime range checking from shipped programs because testing supposedly removes all faults. However, runtime checking can provide the only means within a system to identify a wide variety of systematic faults that occur in rare combinations of circumstances.

Single-Point Faults

Devices ought to be safe when there are no faults and the device is used properly. Most experts consider a device “safe,” however, only when any single-point failure cannot lead to an incident; that is,

the failure of any single component or the failure of multiple components due to any single failure event should not result in an unsafe condition.[19]

19. Whether or not the system must consider multiple independent faults in its safety analysis depends on the risk. If the faults are sufficiently likely and the damage potential sufficiently high, then multiple-fault scenarios must be considered. I once worked with a spacecraft design that was required to be dual-point-fault-safe.

For example, consider total software control on a single CPU for a patient ventilator. What happens if the CPU locks up? What if EMI corrupts memory containing the executable code or the commanded tidal volume and breathing rate? What if the ventilator loses power? What if the gas supply fails? What if a valve sticks open or closed?

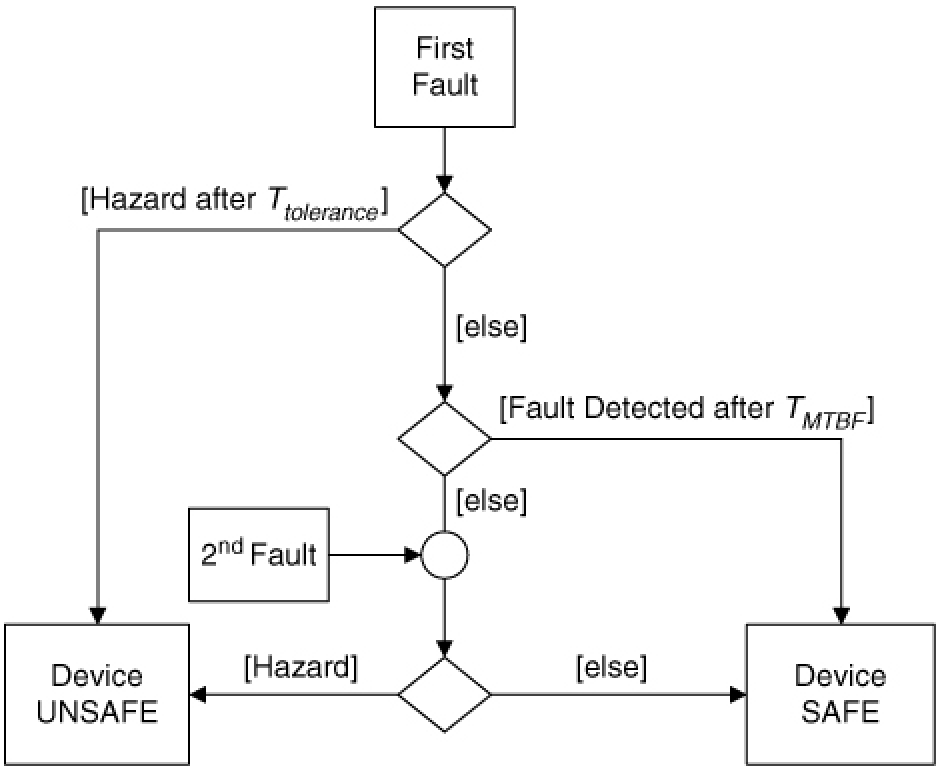
Given that an untoward event can happen to a component, one must consider the effect of its failure on the safety of the system. If the only means of controlling hazards in the software-controlled ventilator is to raise an alarm on the ventilator itself, then the means of control may be inadequate. How can a stalled CPU also raise an alarm to call the user's attention to the hazard? This fault, a stalled CPU, affects both the primary action (ventilation) and the means of hazard control (alarming). This is a common mode failure, that is, a failure in multiple control paths due to a common or shared fault.

The German safety assessment organization TUV uses a single-fault assessment tree for determining the safety of devices in the presence of single-point failures.[20]

20. German Electrotechnical Commission of the German Standards Institute, *VDE-0801 Principles for Computers in Safety-Related Systems*, 1990.

In Figure 5.5, $T_{tolerance}$ is the fault tolerance time of the first failure (the time that the fault can be tolerated without incident). T_{MTBF} is the time after which a second fault is likely (which can be estimated by the MTBF).

Figure 5.5 *TUV single-fault assessment*



If the purpose of runtime testing is to identify faults, then testing must be repeated periodically. The fault tolerance time determines the maximal period of the test, that is,

$$T_{test} < T_{tolerance} \ll T_{MTBF}$$

In other words, the time between tests must be less than the fault tolerance time. If the system cannot guarantee test completion within this time frame, then it must provide some other mechanism for fault identification and control.

A hazard analysis is a document that identifies faults, resulting hazards, and the hazard control measures. The hazard analysis requires periodic review during the development process. The development process must also track identified hazard control measures forward into design, implementation, and validation testing. Design decisions add failure modes to the hazard analysis.

For safety analysis, you cannot consider the probability of failure for the single fault. Regardless of how remote the chance of failure, a safe system continues to be safe in the event of *any* single-point failure. This has broad implications. Consider a watchdog circuit in a cardiac pacemaker. A watchdog is a circuit that requires periodic service or it forces the system to go into a fail-safe state. A watchdog must use a different time base from the CPU running the software. If the same crystal drives both the CPU and the watchdog, then the watchdog cannot detect a doubling or tripling of the pacing rate in the event of a crystal failure. Pacing the heart at 210 beats per minute is an unsafe condition for anyone. Crystals are typically reliable components, but they do fail. When they fail, the system must continue to be safe.

Common Mode Fault

The pacemaker example above illustrates a **common mode fault.** A common mode fault is a single fault that affects multiple parts. Safety measures must not have any common mode faults in common with the processes for which they seek to ensure safety. If a medical linear accelerator uses a CPU to control the radiation dose, then the safety mechanisms that reside on that CPU should not lead to an unsafe condition if that CPU fails.

Latent Faults

An undetected fault that does not, by itself, lead to a hazardous condition but could, in combination with another fault, lead to such a condition is known as a **latent fault.** Since safety measures improve safety only when they function correctly, a measure can be relied upon only if it is known to be valid. Put another way, if a system cannot routinely validate that a safety measure is working properly, then the safety of the system cannot rely upon that measure. For example, consider a medical linear accelerator that has a safety measure that involves dropping a radiation-opaque curtain if an overdose condition is detected. If that safety measure doesn't work, it doesn't directly lead to an accident. However, in combination with a fault in the shutoff timer, it would result in one. If the safety measure isn't tested, then the system cannot know about the latent fault. A subsequent failure in the shutoff timer 10 years later could lead to an accident. The correct action is for the system to periodically test the curtain functionality. This test should be performed much more frequently than the anticipated MTBF of the curtain. For example, if the MTBF is 4000 hours of use, the system could test the curtain at each power-up and at the monthly maintenance service. Only if this is done can the curtain provide a safety measure against the single-point failure of the shutdown timer.

Fail-Safe State

The concept of a fail-safe state is central to many safety-critical system designs. The fail-safe state is a condition known to be safe although not necessarily one in which the system delivers services. Nancy Leveson has identified several different types of safe failure modes:

• *Off state*

° *Emergency stop*—immediately cutting power

° *Production stop*—stopping as soon as the current task is completed

° Protection stop—shutting down immediately but not by removing power

• *Partial shutdown*—going to a degraded level of functionality

• *Hold*—no functionality, but safety actions are taken automatically

• *Manual or external control*—system continues to function but only via external input

• *Restart*—system is rebooted or restarted

The problem domain usually disallows several of these choices. An engine in an airborne aircraft cannot merely be shut down in the event of a failure (off), unless there is another engine that can take over. In the case of unmanned space vehicles, the fail-safe state usually is to blow up the rocket (hold). Attended medical devices often shut down and alarm the user (emergency stop), although sometimes they will enter a monitor-only condition (partial shutdown). When a person enters a hazardous area, a robot control system may finish the current task before shutting down to protect persons and equipment (production stop). The fail-safe state can be determined only by examining the purpose of the system and the context in which it executes.

A safety-critical system may have several different fail-safe states to handle failures in different control or data paths. A detailed analysis of the failure modes of the system determines the most appropriate fail-safe states.

Achieving Safety

The single most fundamental safety design concept is the separation of safety channels from nonsafety channels. This is the **Firewall Pattern.** A **channel** is a static path of data and control that takes some information from sensors and produces some output control of actuation. Any fault of any component of the channel constitutes a fault of the entire channel. A channel can be a control-only path with no feedback, or it can be a tightly coupled control loop, including both sensors and actuators.

One application of the firewall concept involves isolation of all nonsafety-related software and hardware components from those with safety responsibility. Since developing safe subsystems is much more difficult than otherwise, this separation can usually be economically justified. The separation of safety-critical components simplifies their design and implementation, making them more tractable.

Another example of the firewall idea is the separation of control from its correlated safety measure, as in the linear accelerator example above. This requires some kind of redundancy. The redundancy can be small-scale, as in the protection of local data using ones-complement multiple storage or cyclic redundancy checks (CRCs) on stored data. The redundancy can be large-scale as well, replicating an entire subsystem chain. The large-scale redundancy is commonly done in the context of a safety pattern or safety framework.[21]

21. See, for example, my book *Real-Time Design Patterns*.

This redundancy can be **homogeneous** or **diverse.**[22] Homogeneous redundancy uses exact replicas of channels to improve safety. Homogeneous redundancy protects against only random failures. Commercial airplanes often use triple-modular redundancy (TMR) in which they have three such replicas for flight control to protect against single-point component failures. The idea is that if we concern ourselves with only single-point failures, and the channels have no common mode fault, then in the presence of a fault, two of the channels will always agree and only one will contain the fault. The two channels that agree outvote the erroneous one, so the correct action is taken. Diverse redundancy uses different means to perform the same function. It is called “diverse” because the redundancy is not achieved through simple cloning of the channel but through a different design or implementation.

22. Diverse redundancy is also called “heterogeneous.”

Redundant storage of data offers a simple example. In homogeneous redundant storage the data may be stored three times and compared before use. In diverse redundant storage a second copy of the data may be stored in ones-complement format or a CRC stashed with the data. An industrial process can be homogeneously redundant when the control loops are replicated on identical computers. It can be diversely redundant when a proportional-integral-derivative (PID) control loop is used on one computer and a fuzzy logic or neural network algorithm on another to solve the same problem.

Diverse redundancy is the stronger of the two because it protects against systematic as well as random faults. The idea is that both channels may have faults, but if a different design is used, they won’t be the *same* fault, and so the channels won’t fail in the same way or at the same time. If a software flaw in the flight control computer turns the plane upside down when you cross the equator,[23] having three different computers containing the same code won’t help on those flights to Rio de Janeiro.

23. I just hate when that happens.

Software can be redundant with respect to either data or control or both. Data redundancy can be as simple as storing multiple copies, or as complex as needed. Different types of redundancy provide varying degrees of protection against different kinds of faults. Many different mechanisms identify data corruption, such as:

• Parity

• Hamming codes

• Checksums

• CRCs

• Homogenous multiple storage

• Complement multiple storage

Simple 1-bit parity identifies single-bit errors, but not which bit is in error; nor does it protect against multiple-bit errors (that is, errors in even numbers of bits will not be detected). Hamming codes contain multiple parity bits to identify n-bit errors and repair (n-1)-bit errors. Checksums simply add up the data within a block using modulo arithmetic. CRCs provide good data integrity checks and are widely used in communication systems to identify data stream corruption. CRCs have the advantage of fairly high reliability with a small size and low computational overhead.

The data may simply be stored in multiple locations and compared prior to use. A stronger variant of multiple storage is to store the data in ones-complement form. The ones-complement form is a simple bit-by-bit inversion of the original data. This latter form protects against certain hardware faults in RAM such as stuck bits.

Redundant software control replicates controlling algorithms. The system compares the results from the replicates prior to control signal use. Homogeneous control redundancy is of no use in the detection of software faults. Diverse redundancy requires different algorithms computing the result, or the same algorithm implemented by different teams.[24] The redundant algorithm can be less complex than the primary one, if it only needs to provide reasonableness checks.

24. Unfortunately, the faults found in redundant systems written by different teams are not entirely statistically independent (see Leveson, *Safeware*). This can be mitigated somewhat by purposely selecting different algorithms when possible.

Reasonableness checks may be simple and lightweight. If the primary system algorithm has a simple inverse operation, the reasonableness check may simply invert the result and compare the answer with the initial data. Algebraic computation usually, although not always, can be inverted.

In many cases, the inverse operation may not exist or may be too complex to compute. Instead of algorithmic inversion, a reasonableness check may perform an alternative forward calculation using a different algorithm, such as using a fuzzy logic inference engine to check a PID control loop. In this kind of redundancy, it is not always necessary for the secondary system to have the same fidelity or accuracy as the primary. It may be necessary to check only that the primary system is not grossly in error. In all but the highest-risk-category devices, a lightweight, but less accurate, reasonableness check may be sufficient.

Redundancy can implement either feedback error detection or feed-forward error correction. Feedback error detection schemes identify faults but do not attempt to correct the action. Instead, they may either attempt to redo the processing step that was in error or terminate processing by signaling the system to go to a safe shutdown state.

Some systems use feedback error detection to identify when they should enter a fail-safe state. Many systems do not have a fail-safe state. For example, it may be unsafe for an error in a computational step of a flight control computer to shut down the computer while flying at 35,000 feet. It may not be a good idea for an unattended patient ventilator to shut down when it detects an error.

Many systems do have a fail-safe state. A nuclear reactor can safely shut down by inserting its control rods into the core to dampen the nuclear reaction. An attended ventilator can cease ventilation, provided that it alerts the attending physician. However, there may be nonsafety reasons for not forcing the fail-safe state. A high frequency of false negative alarms lowers the availability of the system, possibly to unacceptable levels. In this case, retrying the computation may be a better choice, depending on the risk associated with continuing or stopping.

Feed-forward error correction schemes try to correct the error and keep processing. This is most appropriate when there is significant risk to shutting the system down, or when the fault’s cause is unambiguous and correctable. A common implementation of feed-forward error correction is to reconstruct correct data from partially corrupted values.

Identifying the Hazards

The first step in developing safe systems is to determine the hazards of the system. Recall that a hazard is a condition that could allow a mishap to occur in the presence of other nonfault conditions. In a patient ventilator, one hazard is that the patient will not be ventilated, resulting in hypoxia and death. In an ECG monitor, electrocution is a hazard. A microwave oven can emit dangerous (microwave) radiation, cooking the tissues of the user. It is not uncommon for embedded systems to expose people to many potential hazards.

Naturally, a normally functioning system should provide no unacceptable hazards; they should have all been handled in some way. The identification of the potential hazards is the first step; the compilation of the hazards forms the initial hazard analysis.

The hazard analysis is a document written at the same time, or even preceding, the system specification. This living document is continuously updated throughout the development process. It contains:

• The identified hazards, including

° The hazard itself

° The level of risk

° The tolerance time—how long the hazardous condition can be tolerated before the condition results in an incident

• The means by which the hazards can arise

° The fault leading to the hazard

° The likelihood of a fault

° The fault detection time

• The means by which the hazards are handled

° How the hazard is detected and mitigated

° The fault reaction (exposure) time

Such a table might look like Table 5.4.

Table 5.4 *Hazard Analysis*

Hazard	Fault	Severity (1 (low)–10 (high)	Likelihood (0.0–1.0)	Computed Risk	Time Units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe?
Hypoventilation	Breathing tube disconnect	10	0.2	2	minutes	5	0.5	Blood oxygen sensor	2	2.5	TRUE
Hypoventilation	Ventilator timer error	10	0.2	2	minutes	5	0.5	Independent pressure sensor with alarming	2	2.5	TRUE
Hypoventilation	Gas supply failure	10	0.4	4	minutes	5	0.05	Ventilator incoming gas pressure sensor	2	2.05	TRUE
Hypoxia	Gas mixer failure	10	0.6	6	minutes	5	0.05	Inspiratory limb O_2 sensor	2	2.05	TRUE
Hyperventilation	Ventilator timer error	8	0.1	0.8	minutes	20	0.5	Blood oxygen sensor	2	2.5	TRUE
Overpressure	Pump failure; expiratory tube blockage	10	0.3	3	ms	200	10	Secondary pressure sensor with autorelease valve	5	15	TRUE

In Table 5.4 we can see that hypoventilation is a severe hazard, but one that can be tolerated for about five minutes. Several different faults can lead to this particular hazard.

The first is that the ventilator just quits working. We see that the system includes a secondary pressure alarm that detects the fault in 30 seconds and by 35 seconds raises an alarm to the user. This is an appropriate means for handling the hazard since normal operation requires attendance by a qualified user. This would be an inappropriate means of control for an unattended ventilator.

Another fault that can occur is that the user fails to properly intubate the patient; instead of inserting the endotracheal tube in the trachea, the user manages to put it in the esophagus. The CO_2 monitor detects this hazard within 30 seconds. The breathing gas mixture delivered to the patient lacks significant CO_2, but CO_2 is present in high levels in normally expired gas. An insufficiently high level of CO_2 in the expiratory gas means that the patient is not expiring into the breathing circuit. Thus, a CO_2 monitor is an appropriate means for handling the hazard, again provided that an operator is in attendance.

The third fault identified is that the user attaches the breathing circuit hose to the wrong connectors of the ventilator. This can easily happen in an operating room where there may be dozens of hoses lying around in the crowded area of the anesthesiologist. Designing different sizes of connectors eliminates this fault entirely.

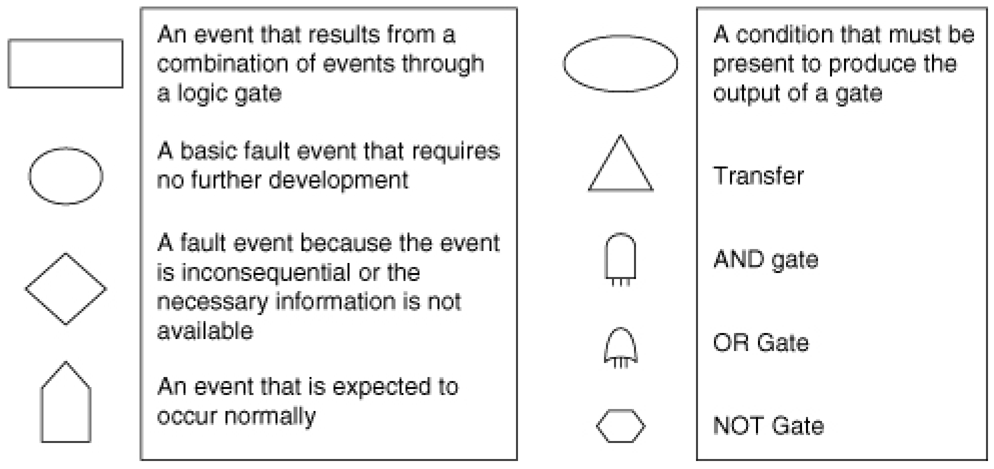
The next hazard is that the patient's lungs are overinflated, producing pulmonary barotrauma. This is a serious hazard that can also lead to death, particularly in neonates who lack a strong rib cage. Note that this fault cannot be tolerated for more than 50 ms, so alarming is an inappropriate measure for handling the condition. Here, a secondary mechanical overpressure valve releases the pressure before it can rise to dangerous levels. The response time for the valve is 10 ms, well below the tolerance limit of 50 ms.

Determining the Faults Leading to Hazards

The hazard analysis lists the hazards, as discussed above. Once hazards are identified, the faults causing the hazards must be determined. FTA is a common method for analyzing faults. An FTA graphically combines fault conditions using Boolean operators: OR, AND, NOT, and so on. The typical use is to begin with an unsafe system state and work backward to identify the causal conditions that allowed it to happen. The analysis can be done from obvious fundamental faults and propagated forward, as is done with FMEA, discussed later.

The symbols identified in Figure 5.6 represent the Boolean equations for the combination and propagation of faults into hazards. The AND gate outputs a logical TRUE if and only if all of its inputs are true—that is, the precursor events have occurred or conditions are present. The OR gate outputs a logical TRUE if any of its antecedents are true. The NOT gate outputs a TRUE only if its antecedent is false. The AND and OR gates may take any number of precursor events or inputs greater than one. The NOT takes only a single input.

Figure 5.6 *FTA symbology*



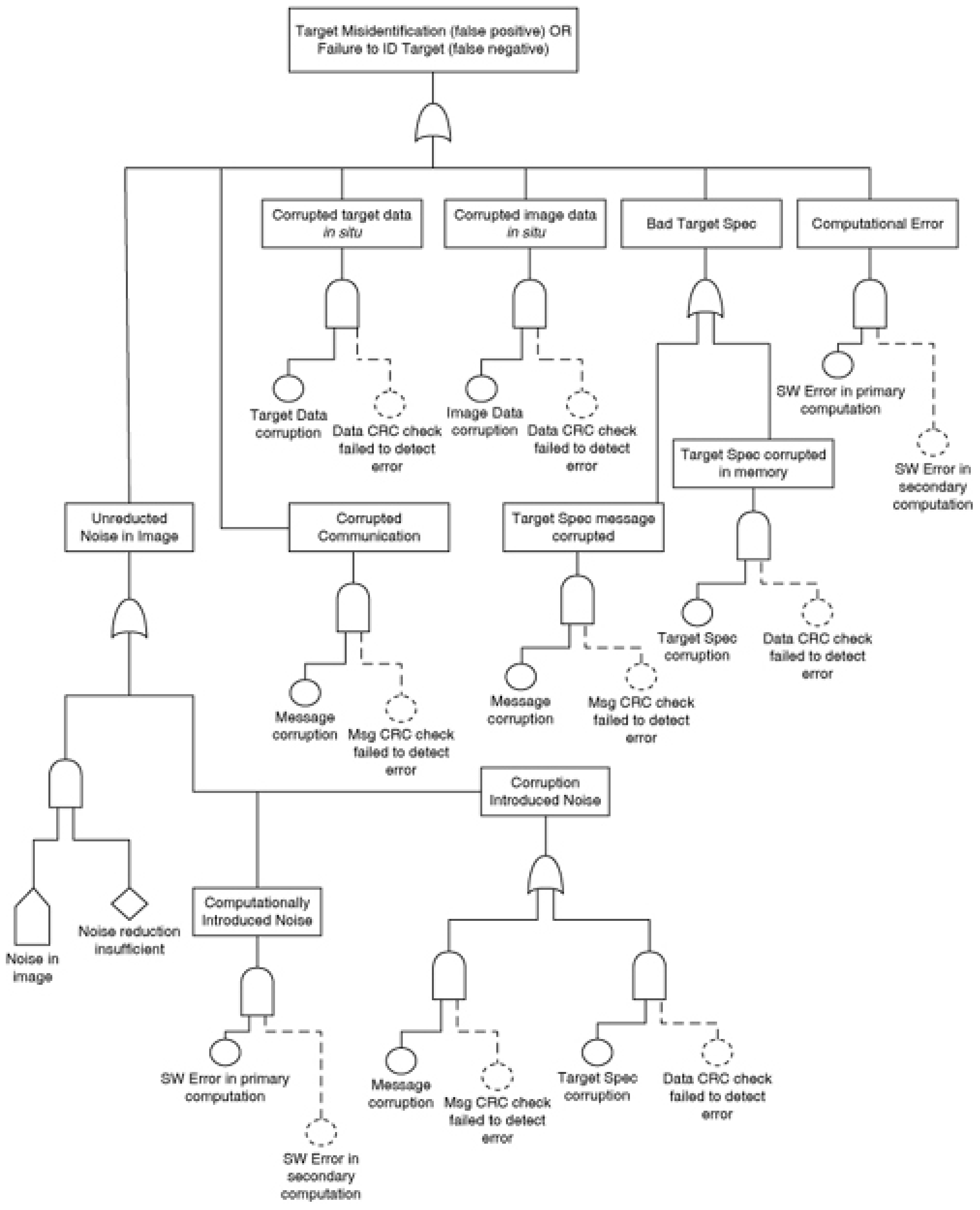
An event that results from a combination of events through a logic gate
A basic fault event that requires no further development
A fault event because the event is inconsequential or the necessary information is not available
An event that is expected to occur normally
A condition that must be present to produce the output of a gate
Transfer
AND gate
OR Gate
NOT Gate

FTA is useful because it shows the combination of conditions required for an accident to occur. The risk of the accident can be reduced by introducing safety control measures that either reduce the likelihood of a fault or the severity of the hazardous condition. Safety measures are sometimes called “AND-ing conditions” because they are combined with an AND operator to the original conditions, so that for the accident to occur *both* the original fault must occur AND the safety measure must fail. It is typical to create a separate FTA diagram for each accident or hazard you want to avoid.

An example[25] is shown in Figure 5.7. In this hazard, the unmanned air vehicle (UAV) identifies targets. Shooting at the wrong thing or *not* shooting at the right thing are hazards we want to avoid. The FTA shows the combination of conditions necessary for this hazard to be realized. Also shown (in dashed lines) are the safety measures. These create the AND-ing conditions, making it much less likely that the accident will occur.

25. Taken from my book *Real-Time UML Workshop for Embedded Systems*.

Figure 5.7 *Unmanned air vehicle target misidentification hazard*



Target Misidentification (false positive) OR Failure to ID Target (false negative)
Corrupted target data *in situ*
Corrupted image data *in situ*
Bad Target Spec
Computational Error
Target Data corruption
Data CRC check failed to detect error
Image Data corruption
Data CRC check failed to detect error
SW Error in primary computation
SW Error in secondary computation
Target Spec corrupted in memory
Unreducted Noise in Image
Corrupted Communication
Target Spec message corrupted
Target Spec corruption
Data CRC check failed to detect error
Message corruption
Msg CRC check failed to detect error
Message corruption
Msg CRC check failed to detect error
Corruption Introduced Noise
Noise in image
Noise reduction insufficient
Computationally Introduced Noise
SW Error in primary computation
SW Error in secondary computation
Message corruption
Msg CRC check failed to detect error
Target Spec corruption
Data CRC check failed to detect error

FTA starts at hazards and tries to identify underlying precursor faults. FMEA starts with all components and their failure modes and looks forward to determine consequences. A related technique, FMECA, is also common. These latter techniques are more commonly applied to reliability analysis than to safety analysis. Many electrical engineers are familiar with FMEA from reliability assessment, so it is a well-known technique.

FMEA considers a number of fields for relevant fault information and analysis, including:

• *Process step*—description of where in the system execution the failure occurs

• *Potential failure modes*—description of the modes of failure

• *Potential failure effects*—description of the effects of the failure

• *Potential causes*—description of the causes that led to the failure

• *Resolution*—recommendations to handle, obviate, or mitigate the failure

• *Assignee*—the person responsible for identifying and executing the resolution

In addition, the following quantitative data is also captured:

• Severity, for example, measured in the range 1 to 10, from 1 = no effect to 10 = catastrophic

• Likelihood, for example, measured in the range 1 to 10: 1 = 1 in 1,000,000; 2 = 1 in 20,000; 3 = 1 in 5000; 4 = 1 in 2000; 5 = 1 in 500; 6 = 1 in 100; 7 = 1 in 50; 8 = 1 in 20; 9 = 1 in 10; 10 = 1 in 2

• Detectability, for example, measured in the range 1 to 10: 1 = 100%; 2 = 99%; 3 = 95%; 4 = 90%; 5 = 85%; 6 = 80%; 7 = 70%; 8 = 60%; 9 = 50%; 10 = < 50%

• Risk = severity × likelihood × detectability

The quantitative data may be entered for both pre-actions and post-actions—that is, prior to and after failure mitigation strategies and design elements are in place—to permit the analysis of the improvement due to the resolutions performed. It is common to use a spreadsheet to represent all the data related to a specific fault or process-step failure within a single row.

Other techniques have been applied to safety analysis, such as flowcharts, cause-effect graphs, and cause-consequent diagrams, but not as widely.

Determine the Risks

Risk levels are specified in a number of different industry-specific standards. FDA identifies three risk classes—minor, moderate, and major—using the definitions in Table 5.5. European standard IEC 651508 defines four safety integrity levels (SILs) and defines required analyses and measures for level 1 (lowest) through level 4 (highest). The FAA uses the RTCA standard DO-178B to specify 5 levels of risk, from level E (no safety effect) up to level A (catastrophic failure).

Table 5.5 *FDA Risk Classes*

FDA Level of Concern	Definition
Minor	Failures or latent design flaws would not be expected to result in injury or death.
Moderate	Failures or latent design flaws result in minor to moderate injury.
Major	Failures or latent design flaws result in death or serious injury.

Defining the Safety Measures

A safety measure is a behavior added to a system to handle a hazard. There are many ways to handle a hazard:

• *Obviation*—The hazard can be made physically impossible

• *Education*—The hazard can be handled by educating the users so that they won't create hazardous conditions through equipment misuse

• *Alarming*—The hazard is announced to users when it appears so that they can take appropriate action

• *Interlocks*—The hazard can be removed by using secondary devices and/or logic to intercede when a hazard presents itself

• *Internal checking*—The hazard can be handled by ensuring that a system can detect that it is malfunctioning prior to an incident

• *Safety equipment*—The hazard can be handled by the users wearing equipment such as goggles and gloves

• *Restriction of access*—Only knowledgeable users have access to potential hazards

• *Labeling*—The hazard can be handled by labeling, for example: High Voltage—DO NOT TOUCH

There are many considerations when applying a means of control to a hazard, such as:

• Fault tolerance time

• Risk level

• Presence of supervision of the device: constant, occasional, unattended?

• Skill level of the user: Unskilled or expert users? Trained or untrained?

• Environment in which the system operates

• Likelihood of the fault that gives rise to the hazard

• Exposure time to the hazard due to the detection and response times of the means

• Scope of the fault's effects: Can the condition that induced the fault also affect the means of control?

A control measure must factor in all these considerations to effectively handle a fault condition.

Create Safe Requirements

A reliable system is one that continuously meets its availability requirements. A safe system is one that prevents mishaps and avoids hazardous conditions. Specifying a safe system often means specifying negations, such as

The system shall not energize the laser when the safety interlock is active.

or

The system shall not pass more than 10 mA through the ECG lead.

Good specifications do not unnecessarily constrain the design. Safety requirements handle hazards that are intrinsic to the system functionality in the context of its environment. Design decisions introduce design hazards—for example, a design that requires a high-voltage input introduces the risk of electrocution. Requirement specifications generally do not address design hazards. Design hazards must be added to the hazard analysis and tracked during the development process. In the context of this chapter and task—initial safety and reliability analysis—the focus is on the intrinsic concerns of safety and reliability and the addition of appropriate safety requirements to the stakeholder requirements specification. Design-specific reliability and safety concerns will be addressed later in the system requirements specification and directly in the design itself.

Checklist for Initial Hazard Analysis

Creating a complete and accurate hazard analysis is a complex, often arduous, task. The primary concerns are the following:

• Does the hazard analysis represent all of the essential hazards of the system?

• Is each hazard quantified as to severity and likelihood?

• For each hazard, are primary faults identified?

• Are faults quantified with a fault tolerance time?

• Are safety or reliability control measures identified for each hazard and each fault?

• Are the control measures quantified with identification and control action times, and is the sum of those values less than the fault tolerance time?

• Are the control measures commensurate with the level of risk, according to relevant standards?

• Has each control measure resulted in one or more requirements in the requirements specification?

The hazard analysis isn't required for all projects. It is a work product specific to safety-critical projects. For those projects to which it applies, it is a living document, initially created during prespiral planning, but it is updated to reflect additions and changes that take place as a natural part of design and development activities.

5.3. Developing Stakeholder Requirements

The stakeholder requirements are "high-level" in that they focus almost exclusively on the concerns of a particular stakeholder—the customer or user of the system. This activity is concerned with two primary work products: the product vision and the stakeholder requirements document. Figure 5.8 shows the workflow for this activity.

As shown in the figure, the primary tasks for this activity are:

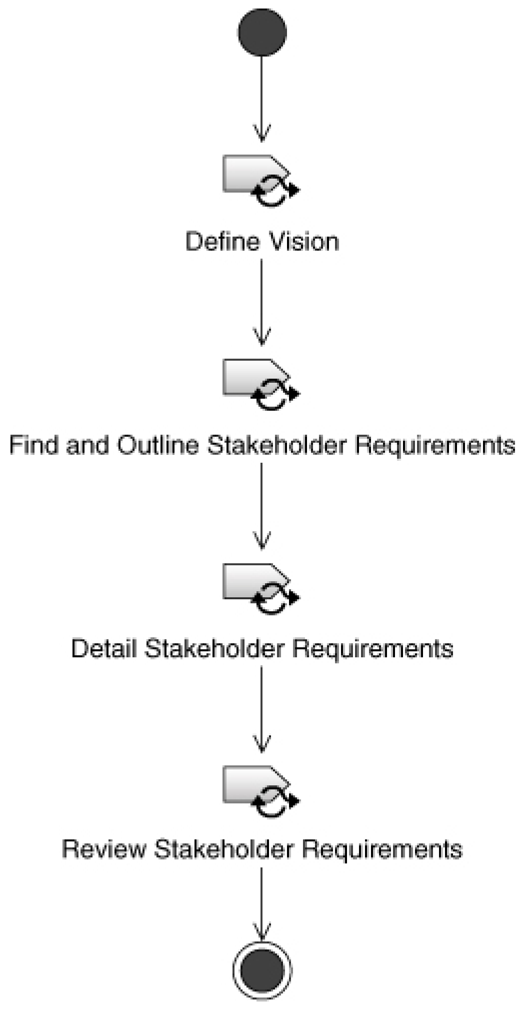
• Defining the product vision

• Finding and outlining stakeholder requirements

• Detailing stakeholder requirements

• Reviewing stakeholder requirements

Figure 5.8 *Developing stakeholder requirements*


Define Vision
Find and Outline Stakeholder Requirements
Detail Stakeholder Requirements
Review Stakeholder Requirements

These tasks result in two primary work products: the product vision (a high-level overview of the system and its benefits) and the stakeholder requirements, a rather more detailed statement of stakeholder needs.

5.3.1. Defining the Product Vision

The **product vision** establishes the context of the system to be built. This includes the relevant stakeholders, what they expect from the system, the execution environment, and the system boundaries (defining what is inside the system and what is outside) and identifies the primary features. Each feature is characterized only to the extent that the purpose, intent, and usage of the feature are clear. This document doesn't include a detailed specification of those features but sets scope and overall expectations. The product vision serves as a nonnormative overview of the product scope and intent.

Checklist for Product Vision

To be useful, the product vision must clearly establish the context of the system but not go beyond an overview level of detail. The primary concerns to be addressed with the product vision are as follows:

• Is the customer's need well understood?

• Is the list of stakeholders complete and correct?

• Is there agreement on the boundary and scope of the proposed system?

• Have you identified the constraints on the problem solution, including political, economic, time, and environmental?

• Have all the key features been identified and are they consistent with the constraints?

• Can someone unfamiliar with the project understand the stakeholders' need and system scope from reading the project vision?

The product vision may be skipped for small projects, but it provides a valuable overview for many different stakeholders, such as customers, marketers, managers, team leaders, architects, testers, and developers.

5.3.2. Finding and Outlining Stakeholder Requirements

A requirement is defined by the IEEE[26] as:

1. A condition or capability needed by a user to solve a problem or achieve an objective

2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document

3. A documented representation of a condition or capability as in (1) or (2)

26. IEEE Computer Society, IEEE 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology* (1990), E-ISBN 0-7381-0391-8.

Informally, we take this to mean that requirements define either what the user needs (stakeholder requirements) or what the system must do (system requirements). Many people adopt the standard that requirements are "shall" statements that identify clearly a singular aspect of the need or usage or the system behavior. In any event, good requirements are correct, clear, complete, consistent, verifiable, and feasible.

The main objective of the task of finding and outlining stakeholder requirements is to clearly and unambiguously define the requirements the system must fulfill. These requirements must be stated in the vocabulary of the customer or operational environment and not in terms of the technology used to design the system. The requirements must be clear, precise, and testable and, at all times, reflect the stakeholder needs. The stakeholder requirements identify the individual requirements for each of the features identified in the product vision.

These requirements must be grouped into use cases; a use case is a usage of the system with a particular intent. It is common that many pages of requirements are clustered together within a single stakeholder use case. The use case serves as a "chapter" or organizational unit of stakeholder requirements. Use cases should be named with verb phrases, such as "Track Tactical Objects," rather than nouns.

The use cases can be identified and used as organizational elements in the textual specification in tools such as DOORS or even Word, but they should also be graphically represented in a UML modeling tool to take advantage of the semantic expressiveness and precision of the UML.

The steps involved in creating the stakeholder requirements include:

• Identifying and capturing problem-domain terms

• Creating the document organizational structure

• Gathering the stakeholder requirements

• Adding a description for each use case

• Defining the use of legacy systems (actors)

• Defining the user roles (actors)

The key to a good stakeholder requirements specification is that each requirement is clear, unambiguous, and testable. Organization into use cases is also important, as is a clear demarcation of the boundary of the system (the stuff you have to create).

Stakeholder or System Requirement?

Stakeholder requirements are in user vocabulary and state a stakeholder need. System requirements state how the system will meet a stakeholder need. That all sounds fine in the abstract, but what does this mean in practice? The primary difference between the two is the degree of precision in the system requirements; system requirements are either stakeholder requirements or are derived from them. Consider the following requirement categorization for a medical ventilator:

• [stakeholder and system] The user shall be able to set the operational mode to pressure-based, volume-based, or jet-ventilation modes.

• [stakeholder] The user shall be able to support tidal volumes appropriate for neonates and adults.

• [system] In volume-based mode, the user shall be able to set the tidal volume to values between 100 ml and 1000 ml in 1 ml increments.

• [stakeholder] The system shall deliver the volume with high accuracy.

• [system] In volume-based mode, the accuracy of volume delivery shall be ±2 ml.

• [stakeholder] The respiration rate shall be settable to support all ranges of medical needs.

• [system] The respiration rate shall be user-settable in the range of 4 to 30 breaths/min.

• [stakeholder] The respiration rate shall be delivered with moderate accuracy.

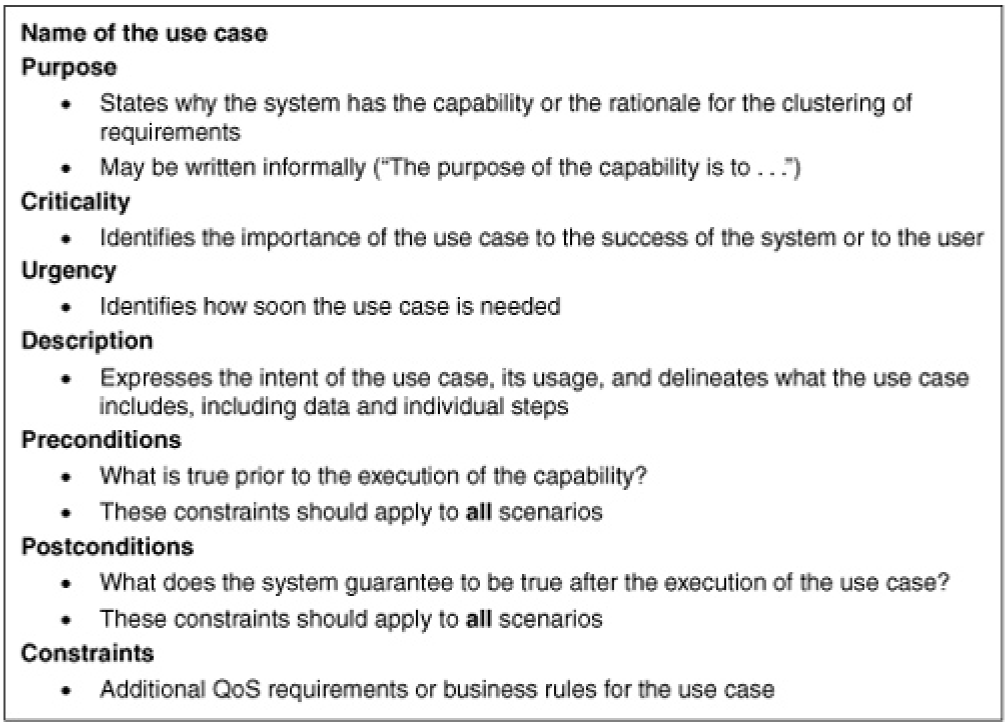
• [system] The accuracy of the delivered respiration rate shall be ±2 seconds.

5.3.3. Detailing the Stakeholder Requirements

Detailing the stakeholder requirements document fleshes out the specifications. Stakeholder use cases are detailed primarily by associating them with textual requirements. Other aspects of requirements are elucidated, including rationale and traceability (to the product vision document).

In the model, each use case is detailed with a description and may be more precisely detailed with sequence diagrams (showing exemplary interactions of the use case) and a state machine (showing all such scenarios merged together). A recommended template for a use case description is given in Figure 5.9.

Figure 5.9 *Use case description*



Name of the use case

Purpose

- States why the system has the capability or the rationale for the clustering of requirements
- May be written informally ("The purpose of the capability is to . . .")

Criticality

- Identifies the importance of the use case to the success of the system or to the user

Urgency

- Identifies how soon the use case is needed

Description

- Expresses the intent of the use case, its usage, and delineates what the use case includes, including data and individual steps

Preconditions

- What is true prior to the execution of the capability?
- These constraints should apply to **all** scenarios

Postconditions

- What does the system guarantee to be true after the execution of the use case?
- These constraints should apply to **all** scenarios

Constraints

- Additional QoS requirements or business rules for the use case

Some authors include additional fields in the description, such as primary and secondary scenarios, actors, and so on. I do not recommend that for the simple reason that since I model the use case diagrammatically, the diagrams (and hence the model) contain that information. I don't recommend putting the same information in multiple places when it cannot be maintained as a single element because this leads to "dual maintenance." For example, if I change the primary scenario or an actor, then I must update both the requirements specification text and the model. I'd really rather have to make the change only once.

5.3.4. Reviewing Stakeholder Requirements

Following the creation of the stakeholder requirements, the development team and the customer must agree on what is being built. There is nothing worse than spending months to years building the wrong thing! The review of the stakeholder requirements normally entails a formal walk-through/review of the textual requirements as well as the use case model. The sequence diagrams can add a great deal of clarity to the often massive amounts of text. Customers can be easily taught the fundamentals of sequence diagrams and most find it a useful way to understand how the requirements interact. As a general rule, I do not expose use case state machines to the customer, since they require significant technical expertise to understand.

Checklist for Stakeholder Requirements

The stakeholder requirements document is optional but can provide value for negotiation with the customer. In many business domains, such as military systems development, the system requirements specification is used for negotiation with the customer (e.g., military procurement organization). When it is created, it is important that it establish a common expectation among the various stakeholders about the needs the system will address. The concerns addressed by a good stakeholder requirements specification include the following:

• Are the requirements written using customer (purchaser or user) vocabulary?

• Are the requirements quantified with urgency and criticality?

• Are the requirements consistent with each other?

• Are the requirements atomic (that is, one need stated per requirement)?

• Are the requirements clear and unambiguous?

• Are the requirements precise?

• Is the performance aspect of the requirements clearly and precisely stated?

• Are the required range and precision of the information clearly and precisely stated, where appropriate?

• Are the requirements verifiable?

• Do the requirements avoid dictating system design?

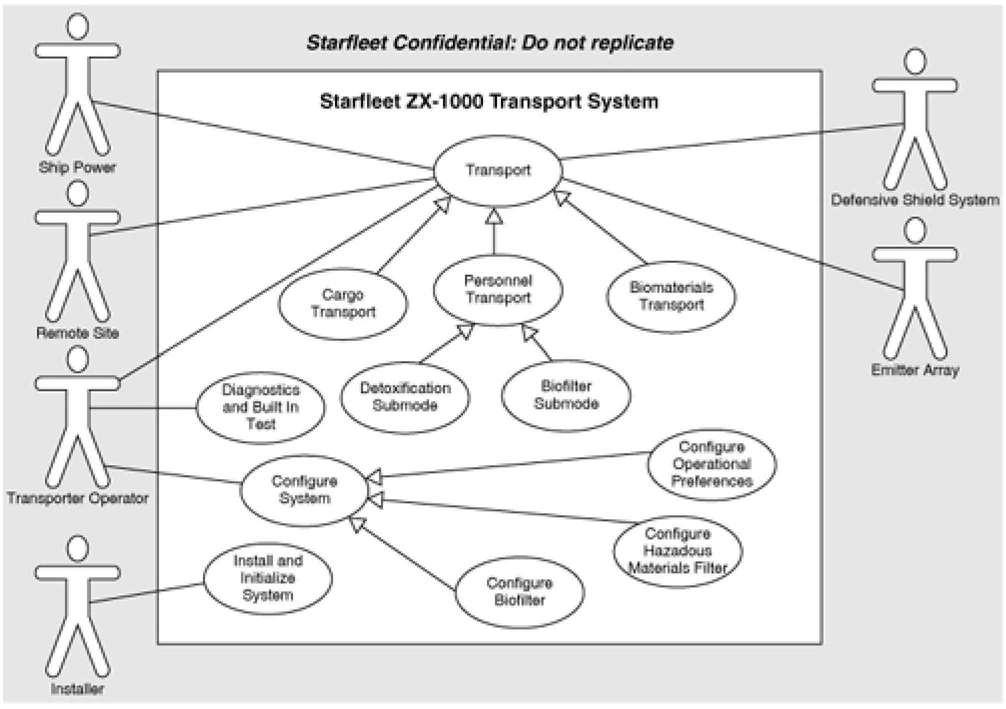
• Are the requirements traced to the product vision?

Note that in cases where the stakeholders are technically and systems-oriented, this document is sometimes omitted and the requirements focus will be on the product vision, the system use cases[27] (identified in a use case model in this activity), and the system requirements (discussed in the next chapter). When the customers and users are nontechnical and less system-oriented, the stakeholder requirements provide a vital bridge to capture the needs of the stakeholders which the system must ultimately satisfy. At the very least, a use case model is needed that identifies the use cases and gives a paragraph description for each, even if the use cases are not detailed more thoroughly than that. For example, Figure 5.10 shows the use case diagram for a Starfleet ship-based matter transporter.[28]

27. The system use cases, by and large, are the stakeholder use cases. They differ in the degree of precision in the use case details, but the use cases themselves are the same. Sometimes, system-specific use cases are added to support technical features not readily apparent to the stakeholder, such as performing a BIT.

28. The system specification for this device is given in Appendix A.

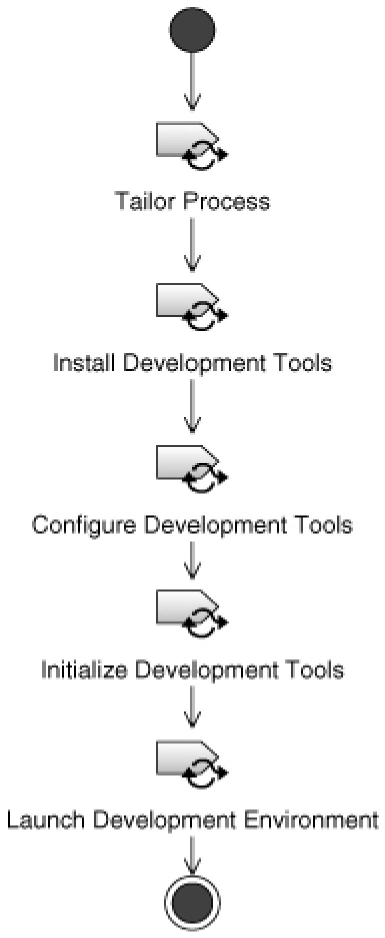
Figure 5.10 *Transporter use cases*



5.4. Defining and Deploying the Development Environment

Before a project can begin, the development environment must be defined, implemented, and configured. The development environment includes the various development tools and the computing infrastructure to be used (both individual computing workstations and any needed servers). It may already exist from previous projects, but even then it is usually necessary to create new baselines for CM, identify the directory structures for work products, and so forth. In addition, it is necessary to install the development process so that developers know what they are doing, when they are doing it, and what work products they are producing. If the process is already in place, it may need to be tailored to take into account project details. The workflow for this activity is shown in Figure 5.11.

Figure 5.11 *Define and deploy the development environment*


Tailor Process
Install Development Tools
Configure Development Tools
Initialize Development Tools
Launch Development Environment

The tasks within this activity address the establishment of a work environment to effectively produce the system under development. These tasks include:

• Tailoring the process

This task customizes the organizational development process for the specific project to take into account peculiar features of the team, system, customer, or contract.

• Installing the development tools

This task acquires and installs the tools necessary for the team, including word processing, CM, modeling, compiling, editing, and testing tools.

• Configuring the development tools

This task configures the selected tools to work within the project and team environment.

• Initializing the development tools

This task initializes the tools for the team.

• Launching the development environment

This task launches the development environment for the team to begin work.

5.4.1. Tailoring the Development Process

This task configures the existing development process to be optimal for the current project, resulting in a document that is usually known as the **software development plan.** The SDP is often a large, monolithic textual document depicting the development phases and tasks the workers will execute, the roles they will play, and the work products they will generate. I prefer to use a hyperlinked Web site hosted on the development team's server that provides the guidance. Open-source tools, such as the EPF Composer[29] or the commercial RMC from IBM, manage process content and can publish this content as Web sites of this kind. In fact, all the graphical process views of the Harmony/ESW process shown in this book are snapshots taken from the output of one or the other of these tools.

29. See www.eclipse.org/epf for the Composer tool as well as exemplary processes, tutorials, and guides.

Different projects often have different process requirements. Some common project/product variations that may profit from process tailoring include:

• Embedded/not embedded

• Real-time/not real-time

• Safety-critical/not safety-critical

• High-reliability/not high-reliability

• Large/small projects

• Colocated development/remote development locations

• Update existing product/create new product

• Significant reuse of legacy/“clean slate” projects

• High project risk/low project risk

• Regulated industry/not regulated

• High use of technical (e.g., modeling) tools/low use of technical tools

• High degree of customer oversight/low degree

• Hardware-software codevelopment/software only

The tailoring might be done either by the project leader or by the company process group. If you’re using a standard textual SDP, tailoring the process involves creating a copy of the baseline SDP and modifying, adding, or removing sections. If you’re using a tool such as EPF Composer or RMC, then your process guidance is already organized into a library of reusable best practices, so tailoring the process is a matter of updating (via specialization, addition, or subtraction) of existing process content and republishing the Web site that defines the process content. The introduction Web page for the Harmony/ESW published content looks like Figure 5.12.

Figure 5.12 *Harmony/ESW published process Web page*

Introduction to Harmony/ESW

- The Telelogic Harmony Process Family
- Harmony/ESW Process Context
- Harmony/ESW Timeframes

Welcome to Harmony/ESW

Harmony/ESW is a member of the The Telelogic Harmony Process Family specifically for Embedded Software development.

Harmony/ESW connects the dots between people, process, tools and best practices to provide a complete solution for embedded software development teams.

The Harmony/ESW Process is generally applicable to software and systems development, but is optimized for the development of software-intensive real-time and embedded systems. Harmony/ESW is:

- **Agile**: embodies the use of key concepts and guiding principles of agile development
- **Efficient**: emphasis is on tasks and work products that add significant to the development
- **High quality**: stresses on continual validation of correctness and completeness throughout development
- **Requirements-driven**: development tasks concentrate on identifying and meeting stakeholders needs

5.4.2. Installing, Configuring, and Launching Development Tools

The development tools include any requirements management tools (e.g., DOORS), compilers, assemblers, linkers, debuggers, IDEs (e.g., Eclipse, or RTOS IDE), modeling tools, and so on that the developers will use to develop the software. In addition, managers will need project management software (e.g., Microsoft Project) to schedule and track progress. Of course, it isn't enough just to install the tools and their license managers. Many times the tools must be configured. This includes defining a common set of properties so that work products developed separately can work together. To configure the development environment, these common properties must be identified, then deployed across the project team. Of course, the deployment of this configuration must be tested to ensure that it actually works before dumping it on the team.

5.4.3. Installing and Configuring the CM Environment

CM is a development tool, but it is special in that it is the primary means for integration of the work of multiple people. I recommend that a separate worker role, the "configuration manager," maintain control over the processes used to build and verify the integration of the software development work products. This is achieved through the use of CM tools (e.g., ClearCase or Synergy). These tools can require complex configuration, so specialist training for the CM manager is usually required.

The CM plan documents the procedures the developers use to work on their desktops (e.g., to check in or check out project components on which to work) as well as the rules that apply to when and how components can be submitted back into the CM environment and the software baseline. A recommend approach to CM, known as “continuous integration,” is discussed in the next section.

5.5. Continuous Integration

CM is a crucial activity in projects that involve the creation of multiple work products by multiple developers—that is, pretty much all projects. Although this is a part of the microcycle iteration, which is detailed in subsequent chapters, it applies to all the primary microcycle activities and is performed concurrently with them. For this reason, it makes sense to discuss it before drilling down into the microcycle details.

Integration is the bringing together of separate components into a cohesive whole (usually a “build”). Waterfall processes perform integration at the end of the development. This identifies all kinds of problems that really should have been identified far earlier. The Harmony/ESW process performs a **continuous** integration as soon as software development begins. Typically, a build is created and tested for consistency daily, throughout the project. This means that integration problems are identified and fixed far earlier and, because the root causes are much easier to find with a small incremental change, at a far lower cost.

5.5.1. What Is CM?

CM is a technical management activity that focuses on establishing and maintaining the consistency of a product’s content, behavior, physical attributes, and performance. It was originally defined by the U.S. Department of Defense as a technical management discipline in the 1950s.[30] Since then, it has become a foundational bedrock on which almost all software and systems projects are based. CM includes the identification of CIs, management of CI revisions and versions, provision for accounting and verification of versions and revisions of CIs, and support for construction of versions of the product with consistent sets of appropriate revisions of CIs.

30. See, for example MIL-HDBK-61A, “Military Handbook: Configuration Management Guidance, Revision A” (February 7, 2001), available at http://assist.daps.dla.mil/quicksearch/basic_profile.cfm?ident_number=202239.

The integrated, coherent set of CIs is referred to as a **baseline.** In high-quality development environments, this baseline is integrated, compiled, and tested for basic functionality before it is made generally available to the engineering staff. In some cases, different teams may work on multiple baselines simultaneously and merge them periodically. In other cases, the CM manager may maintain a singular baseline against which all of the developers work. In any case, one of the main challenges of CM is maintaining the interoperability of the CIs in the context of the baseline. In high-quality CM environments, this is done with frequent integration/test cycles of the baseline. Because the cost of performing CM poorly is high (weeks or months lost to getting the system to integrate), the Harmony/ESW process recommends continuous integration, that is, integrating and performing sanity testing on the baseline at least daily.

Classically, change management is a part of CM as well, but I prefer to think of them and model them as related, but distinct, activities. Change management is discussed in Chapter 9.

CIs are any project work products that must be controlled—essentially, all work products. Requirements engineers write and maintain requirements specifications. Use case analysts create and maintain use case models. Developers (aka "software modelers") develop analysis and design models (including unit test cases) and use them to generate code. Testers write test plans, test cases, and test fixtures. All of the artifacts are CIs, often multiple CIs, that must be managed and controlled. In this book I am primarily concerned with the software analysis and design elements: UML models, source code, test cases, and the like.

Traditional CM involves managing independent CIs and integrating them at the end of the project. Tradition has failed to keep up with the demand for highly capable, and hence complex, system functionality. Harmony/ESW recommends a process known as **continuous configuration management,** in which CIs are integrated and validated continuously throughout the development process. This is described in the next subsection.

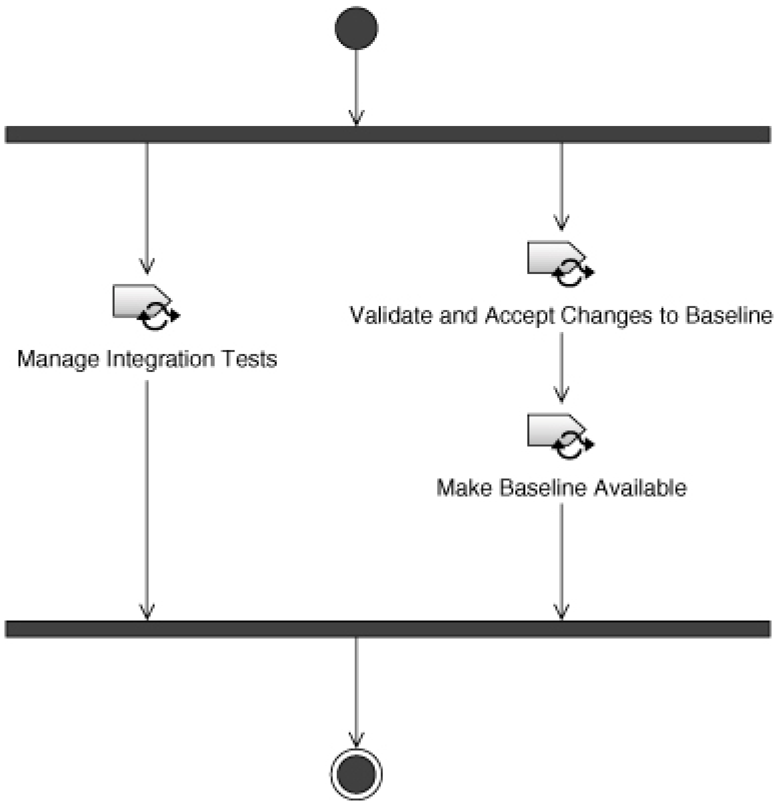
5.5.2. Continuous Configuration Management

Two primary roles are involved in CM. The first is the *creator* role that creates and/or modifies the CIs. The second is the *CM manager* role that accepts the CIs, integrates, and validates them.

The creator creates or modifies the CI and then submits the CI in a task known as "Make change set available." The creator must perform quality assurance activities on his or her own desktop prior to releasing the CI. For software elements, this involves both unit testing and making sure that the element interacts properly with the baseline used by the development staff. The creator wants to ensure that the changes he or she has made to the elements don't break the baseline. The creator does this by loading a copy of the baseline on his or her desktop and performing a build with some set of tests.

The CM manager receives the submitted CIs (also known as “change sets”). Typically, the CM manager receives multiple CIs from multiple engineers. He or she must then take those updated CIs, integrate, and validate them. If the CIs break the build or the existing functionality of the product, the CIs are kicked back to the developers for repair. The workflow for the CM manager role is shown in Figure 5.13.

Figure 5.13 *Continuous configuration management*



Validating and Accepting Changes to the Baseline

This task involves adding the CIs into the CM workspace and building the resulting system. In continuous configuration management, this is done at least daily. This first step ensures that no obvious syntactic or interface errors exist within or between the CIs. It also involves loading the resulting binary image onto the appropriate target environments. These may be target processors, desktop computers playing the role of the target processors, or simulators.

The next step for this task is to run a set of integration tests. These are sometimes referred to as “smoke tests” (as in “See if the machine bursts into flames”). These tests focus on two aspects. The first is the integration of the architectural software elements (components or subsystems). Test cases will emphasize testing various services invoked across the component boundaries. The second set of test cases focuses on functionality. The test cases also exist to ensure that previously existing functionality isn't broken by new changes and that new functionality works as expected. The integration tests are usually a subset of the validation tests because they must be run frequently, so execution time and ease of execution are important issues.

If the test fails, then, with the developers' help, the defects are isolated into their respective CIs, and those CIs are kicked back to their creators for repair. If the tests succeed, then a new baseline with the new changes is created and released to the team at large. This means that the new changes become available to developers only when they have demonstrably not broken the system and have at least continued to provide existing functionality. In this way, the team can move forward effectively. The worst case occurs when a single team member can cause the entire team to halt because his or her defects have broken everyone's ability to construct the system. If this occurs, it is a sign of poor CM procedures and poor developer quality assurance (unit testing).

Making the Baseline Available

The task of making the baseline available is mostly a matter of updating the CM workspace so that when developers check out CIs, they get the new elements. The task involves updating the change descriptions so that developers can see the current interface and functionality contents of the baseline. It may also involve updating installation and deployment instructions and release notes, as necessary and appropriate.

Managing Integration Tests

In parallel with baseline oversight, test cases for the integration must be managed as well. In the beginning, there are no tests, obviously, so the configuration manager must define a minimal set of tests that can be used to differentiate good change set submissions from poor ones. The product subsystem and component architecture (see Chapter 7, "Agile Design") defines the connection points of the components and their interfaces. The microcycle mission identifies which interfaces, and which services within the interfaces, will be created during the initial prototype development. Subsequent microcycles will have their own mission statements and will identify the interfaces, functions, features, and use cases to be added. Further, as a result of prototype definition (Chapter 6, "Agile Analysis"), sequence diagrams representing required interactions at the architectural level detail the use cases being realized. Some set of these can be used for the CM integration tests.

Within this context, the configuration manager works with the developers to identify the initial set of tests to get the baseline started. Over time, as more features are realized and more interfaces are used, test cases are added to the integration tests. This is an ongoing task performed by the CM manager role in parallel with the continuous integration of the evolving baseline.

5.6. Coming Up

This chapter hashed out what is done during project initiation for Harmony/ESW projects. Three key activities are performed here, prior to the start of software development: prespiral planning, creating the stakeholder requirements, and defining and deploying the development environment.

Prespiral planning includes creating the initial schedule and cost estimates, organizing the project team, organizing the work products, planning for reuse, risk reduction, and product safety and reliability. Stakeholder requirements specify stakeholder needs and are typically stated at both overview (product vision) and detailed (stakeholder requirements) levels of abstraction. These will later trace to system requirements and ultimately to system design and test work products. Last, the development environment must be defined and deployed to the project team. This includes tools, processes, and training for the team.

Once these predecessor activities are complete, we're ready to start the project. The project is run as a series of incremental development efforts, each expanding and elaborating on the efforts that came before. This incremental cycle is known as the microcycle (see Figure 4.10).

The microcycle consists of three primary tasks done in sequence. First, analysis details the use cases and the system requirements to be implemented in the current microcycle. This microcycle phase focuses on the essential properties of the product for those use cases, resulting in a CIM and a PIM, also known as the use case model and object analysis model. This workflow is described in the next chapter.

The second step in the microcycle is design. This phase focuses on the optimization of the PIM against the weighted set of design criteria, including but not limited to the product QoS (e.g., performance). This takes place at three levels of design: architectural, mechanistic, and detailed. The design phase is covered in Chapter 7.

Before going on to validation testing, it is common (although not necessarily required) to conduct model-based (and to a lesser degree source code) reviews. The main purpose of these reviews is not to inject quality—after all, these reviews come well after unit testing—but to ensure adherence to the architectural guidelines and to disseminate information about the internal content of components to interested parties. This topic is discussed in Chapter 9.

The last primary phase of the microcycle is testing. This phase performs detailed black-box validation testing of the prototype against the current requirements captured within the use cases. In addition, regression testing is performed by applying at least a subset of the test cases from previous microcycle iterations. This is covered in some detail in Chapter 8.

There is an activity performed in parallel with the primary microcycle phases called “Prepare for validation testing.” In this parallel activity, test cases and testing infrastructure are created and modified to support the effort of the final testing phase of the microcycle. This topic is also discussed in Chapter 8.

Finally, at the end of the microcycle, there is a short review of the project status, known as the increment review, or more informally as the “party phase.” In this half-day to two-day effort, the project performance against plan is examined. The party begins in Chapter 9.

Chapter 6
Agile Analysis

In this chapter, we drill down inside the microcycle. The microcycle, you will no doubt recall from Chapter 4, "Process Overview," is the core iterative component of the entire Harmony/ESW process. Before we begin the microcycle iterations, the overall process (shown in Figure 4.5) shows that both prespiral planning and the "Develop stakeholder requirements" activities have been completed. This means that at minimum, the following things are true:

• A schedule containing the microcycles, their expected completion dates, and their allocation to resources has been created.

• The team is organized and structured.

• A risk reduction plan, with identified RMAs, has been created.

• The stakeholder requirements have been developed and organized into use cases or features.

• The use cases to be realized in the current microcycle have been selected from the stakeholder requirements.

• The model structure has been created.

In addition, where appropriate, the following additional tasks have been performed:

• If the project will create reusable assets or include significant reuse of previously developed elements, then a reuse plan is in place.

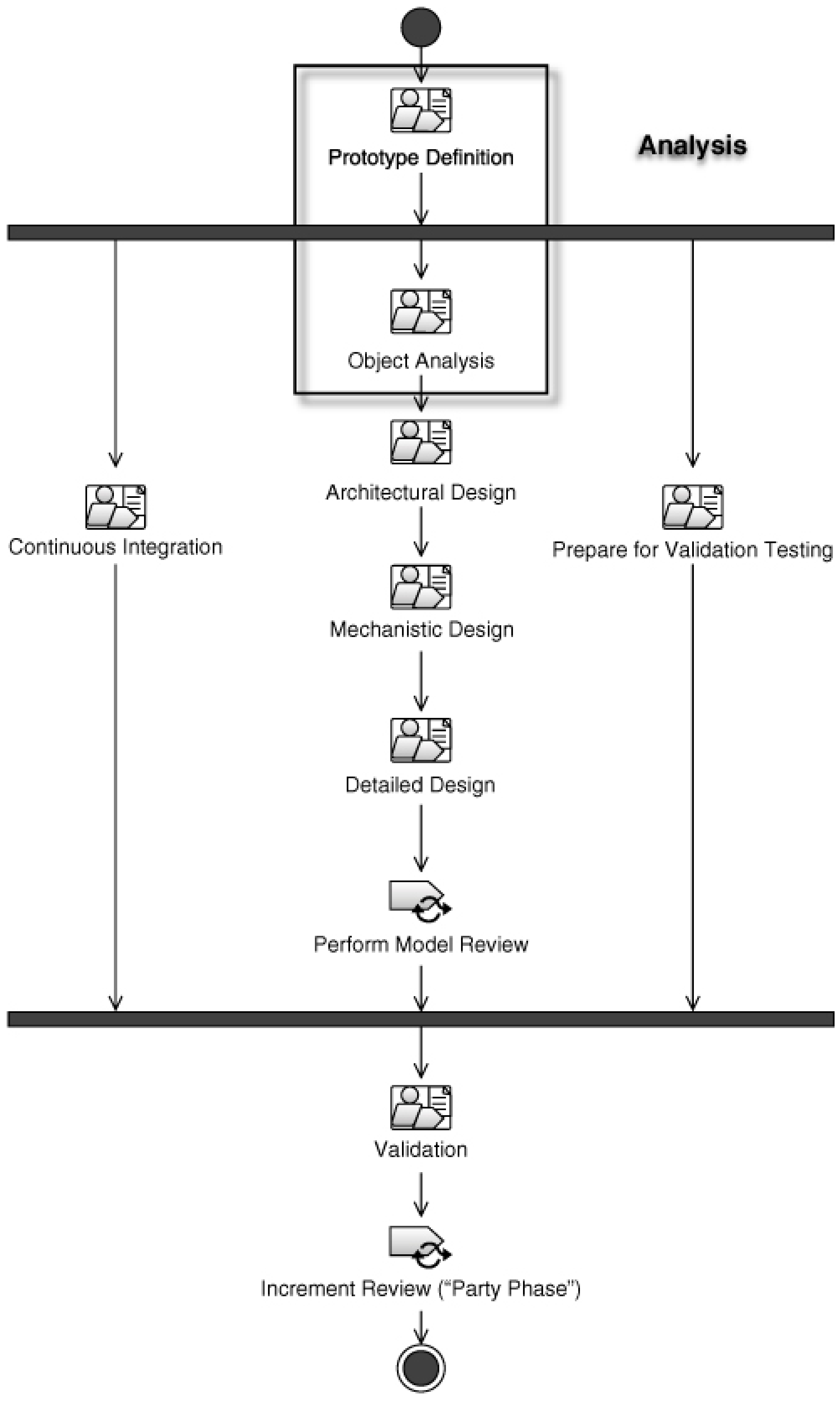
• If the system is either safety-critical or high-reliability, then an initial safety and/or reliability analysis has been performed with appropriate artifacts, such as FTA, FMEA, and hazard analysis.

Each microcycle focuses on the construction of a high-quality, validated realization of some functional aspects of the system; that is, the microcycle realizes, in terms of requirements, analysis, design, and implementation, some small number of use cases in the evolving system. The microcycle takes a large, complex problem and breaks it down into a set of linearly separable subproblems, each of which is an order of magnitude simpler and/or smaller than the entire product. The evolving product is incrementally constructed via the creation of these smaller builds, each of which adds new functionality. At critical milestone points, the prototype[1] can be released outside the development team to customers or users for evaluation or deployment.

1. Remember from Chapter 1 that we use the term *prototype* to mean a validated version of the real system that may lack some of the intended functionality. These prototypes are not meant to prove a point and then be discarded; instead, they contain the *real code* to be shipped in the system. The prototype is created incrementally throughout the project until it is complete and can be released as a commercial product or as a component in a released system.

Analysis is the first phase of that microcycle. As can be seen in Figure 6.1, analysis consists of two activities. The first of these—prototype definition—specifies and elucidates in detail the requirements to be designed, implemented, and validated in the current microcycle. This use case model and the (optional) system requirements specification are the primary forms of the requirements. In MDA terms, this is known as the CIM (computation-independent model) because it specifies what must be done but not the computational elements needed to do it.

Figure 6.1 *Analysis in the Harmony/ESW microcycle*



The second activity is object analysis. The work in this activity constructs a working functional model of the system, functional in the sense that it correctly implements the functionality specified in the prototype definition. Within object analysis, source code is generated, compiled, and unit-tested, resulting in a high-quality, tested version of the system. The object model resulting from object analysis is known in MDA terms as the PIM (platform-independent model) because it emphasizes functionality but defers platform-specific and optimization issues. The work isn't done at the end of object analysis, since the model still requires design optimization. That topic is discussed in Chapter 7, "Agile Design."

In keeping with the principles and practices of agile methods, the analysis work is done with continuous or at least highly frequent execution, code generation, and unit testing. At the end of the analysis phase, the use case model organizes and details the requirements, representing them with model-based concepts such as sequence diagrams and state machines. The object model realizes these requirements, or at least the functional ones, with class diagrams, sequence diagrams, state machines, and activity diagrams. Unit-level test vectors are usually represented with sequence, activity, or state diagrams. Throughout analysis, the project baseline is continuously integrated and tested to ensure that all of the components under development collaborate properly.

6.1. Prototype Definition

Prototype definition, the first activity in the analysis phase, consists of a number of individual tasks (see Figure 6.2). Most of them are order-independent and so are shown as parallel activities. A few of them, such as "Specify user interface" and "Manage safety and reliability requirements," are optional, depending on the nature of the project and product. At the end of the prototype definition activity, the requirements to be realized in the current microcycle are clear, precise, unambiguous, and complete to the best of our ability to make them so.

Figure 6.2 *Prototype definition workflow*

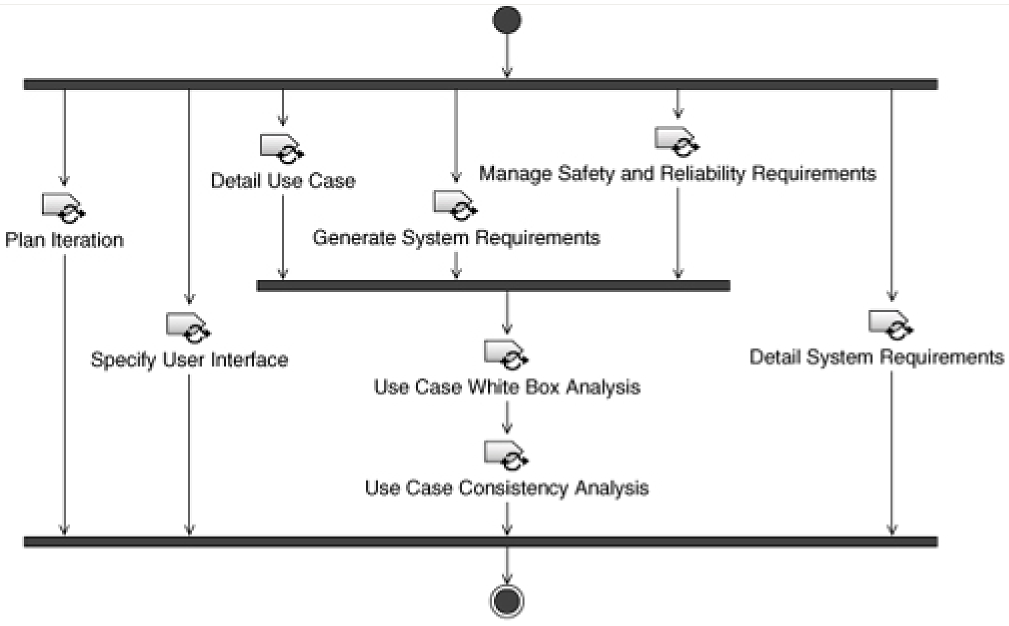

Plan Iteration
Detail Use Case
Manage Safety and Reliability Requirements
Generate System Requirements
Specify User Interface
Use Case White Box Analysis
Detail System Requirements
Use Case Consistency Analysis

Figure 6.2 may seem a bit intimidating as it contains a large number of tasks. Remember, though, that the scope of the prototype definition activity is to produce a use case model for a very small number of use cases—typically two to five—and must be completed in a week or less. The overall microcycle time frame, including the design, implementation, and test, is on the order of four to six weeks, and this activity is just one of the pieces that must be completed in that time frame. It is crucial that the work stay within that modest scope.

Let's discuss each of these tasks in turn.

6.1.1. Plan the Iteration

The purpose of this task is to document the elements of the prototype to be created and enumerate the work items to be completed. This includes detailing the schedule for the scope of the current prototype—typically the next four to six weeks. The purposes of the work done are to:

• Select the use cases from the stakeholder requirements to be realized

• Update the risk management plan

• Prioritize elements in the work items list

• Detail the schedule for the current iteration

• Write the microcycle mission statement

The scope of the iteration plan is the upcoming microcycle, typically only a few weeks long. Nevertheless, this document guides the work and focus for that period.

Selecting the Use Cases

The first step is to select the use cases to be realized. The overall project schedule already plans which use cases should be realized, but this step reevaluates and/or confirms those choices. If different use cases or features are to be realized, then the overall plan must be updated as well.

Given that there are potentially many use cases for the system but only a few can be done within a given microcycle, how do we decide which use cases should be done now and which should be deferred? Many criteria can be justifiably used to select the order in which use cases are realized in different prototypes. These criteria include

• Risk

• Infrastructure

• Availability (of information or needed resources)

• Criticality

• Urgency

• Simplicity

The first of these, risk, is my primary criterion. If all other things are equal, I prefer to resolve high-risk use cases as early as possible. Ignoring risk, you may recall from earlier discussions, is the leading cause of project failure. I prefer to have an exponential washout risk reduction curve such as the one in Figure 1.8. This is best achieved by attacking the highest-risk use cases first. These use cases may be high-risk because of any number of issues, including unfamiliar technology, lack of information, lack of specific required resources, technical difficulty, or ambiguous requirements.

On the other hand, some use cases provide the technical infrastructure required for the development of others. A use case such as `Perform Basic Communication` provides infrastructure required for the use case `Provide Secure Encrypted Communication`. In addition, some use cases are just too large to be constructed in the standard microcycle time frame. Such use cases can be decomposed with the `«include»` relation into “part” use cases that *can* be done within a single microcycle. For example, a use case such as `Track Tactical Objects` may be decomposed into a series of four or five use cases providing incremental functionality, with one of them being realized in each successive microcycle. Thus, some use cases may be scheduled early because their realization provides functionality required by more complex ones.

In many projects, certain features may be well understood from a requirements point of view, while others are not yet settled. In such a case, it is reasonable to realize the more stable requirements while others work on refining the customer’s needs for the unsettled ones. This allows the development team to make progress while those other details are being resolved. In other cases, resources, such as personnel with specialized training or devices (for instance, hardware platforms or simulators), required for implementing or testing a use case may not be available until later in the project. Such use cases can be deferred until the required resources become available.

The criticality of the use case is a measure of the value a use case brings to the user; the higher the criticality, the greater the value to the user. The urgency of a use case refers to how soon that capability is needed. While these two parameters are independent, a combination of the two can be used to select use cases for early realization. Less critical or urgent use cases can be deferred.

The last criterion is simplicity. This is appropriate only when introducing the process or development technology to a team. A simple use case gives the developers an opportunity to practice the development approach on an easier problem to gain experience and confidence.

Most of the system use cases are none other than the stakeholder use cases, but they differ in that the former contain more precise requirements. These more precise requirements typically include specific quantitative ranges and qualities of service. The systems (or software) engineer adds that technical detail during the prototype definition activity. If the stakeholder requirements are already very precise because the stakeholders are technically skilled, then these use cases and textual requirements can be used more or less directly. If not, then system requirements must be derived from stakeholder use cases by adding the necessary precision. In addition, some use cases may be added to the system use case model that don’t directly impact a stakeholder need but are added for some technical purpose. Installation, configuration, power-on test, and built-in test use cases are often added at the system level, for example.

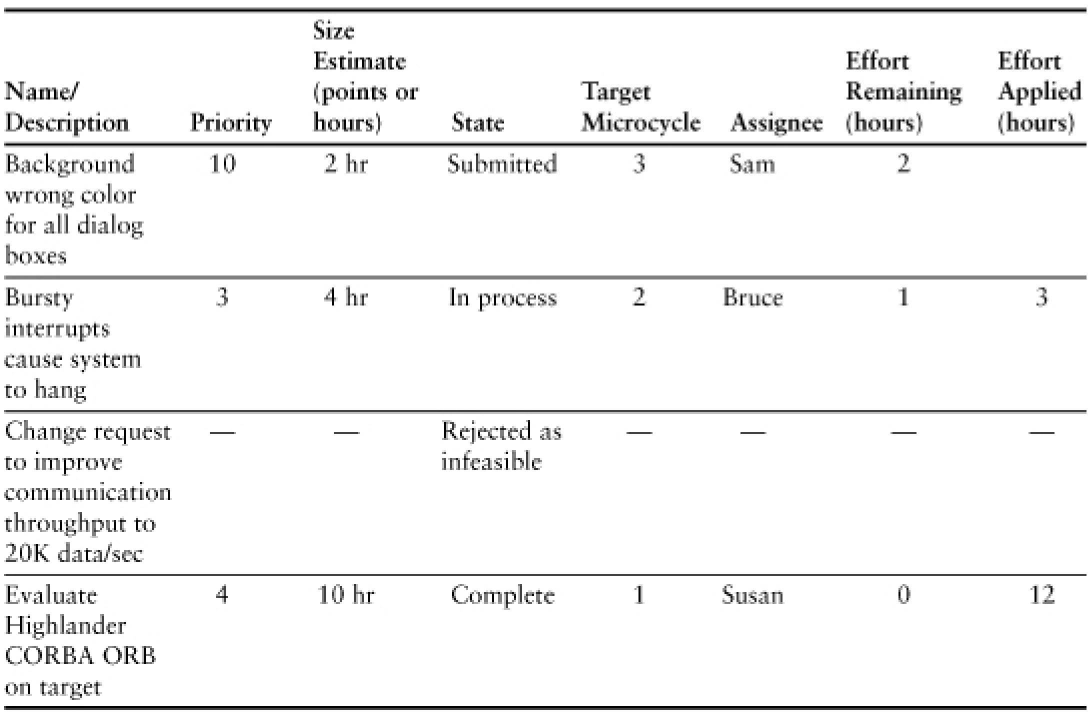
Updating the Risk Management Plan

This step looks at the risk management plan primarily to select the RMAs appropriate for the current microcycle and add them to the work items list. In addition to that, however, the list of risks should be examined to see if it is current. Any new risks should be added to it and quantified by their severity and likelihood. Resolved risks should be marked as such. In addition to the development work, risk must be mitigated in each microcycle. These RMAs must be scheduled into the detailed microcycle plan.

Prioritizing Elements in the Work Items List

The work items list is nothing more than a prioritized list of work to be done. I follow the convention of using a scale of 1 to 10, the lower number being a higher priority. Most of these work items will be known defects that need repair, change requests, and so forth. This list is often maintained in a spreadsheet such as that shown in Table 6.1. This format allows sorting on different columns (such as assignee, priority, state, or target microcycle) and filtering as desired.

Table 6.1 *Work Items List*



Name/ Description	Priority	Size Estimate (points or hours)	State	Target Microcycle	Assignee	Effort Remaining (hours)	Effort Applied (hours)
Background wrong color for all dialog boxes	10	2 hr	Submitted	3	Sam	2	
Bursty interrupts cause system to hang	3	4 hr	In process	2	Bruce	1	3
Change request to improve communication throughput to 20K data/sec	—	—	Rejected as infeasible	—	—	—	—
Evaluate Highlander CORBA ORB on target	4	10 hr	Complete	1	Susan	0	12

Typical states for work items might be:

• Submitted

• Pending

• In progress

• Complete

• Rejected

The work items list is an important work product since it guides the work to be done within the microcycle. The key elements are the requirements and related artifacts (e.g., use cases), but other important elements include known defects to be repaired and RMAs.

Detailing the Schedule for the Current Iteration

The microcycle schedule is a detailed schedule of the current microcycle, so it is limited to a few weeks in scope. This schedule must identify the planned start time, effort, and sequence for the work items to be resolved and the developer work to be performed during the microcycle. The schedule must also assign specific personnel to work items. The project manager will track this schedule frequently (normally daily) so that problems and roadblocks can be identified and resolved as quickly as possible. The techniques and methods for creating this schedule are the same as those discussed in Chapter 5, “Project Initiation,” for the overall project schedule.

Writing the Microcycle Mission Statement

The microcycle mission statement can be an informal document, but it is important because it lays out the crucial aspects of the work to be done within the current microcycle. The microcycle mission statement summarizes all of the information gathered in the other steps of the “Plan iteration” task. These are:

• The list of use cases to be realized

• The list of work items to be realized in this microcycle, including

° The list of RMAs

° The list of defects to be repaired

° The list of changes to be made

• The target platforms to be supported

• The architectural intent for this prototype

• Any external items required for this microcycle

The target platform is mentioned because it often changes in the course of a project. Early prototypes may run only on the desktop or lab system because of the lack of supporting hardware. Over time, as the target platforms, sensors, and actuators become available, they can be added into the target environment.

The architectural intent is discussed in more detail in the next chapter, but for now it is enough to note that the Harmony/ESW process identifies five primary architectural views as well as a number of views of secondary importance. The primary views are:

• Subsystem and component view

• Concurrency and resource view

• Distribution view

• Safety and reliability view

• Deployment view

Secondary views, such as information assurance (e.g., security), data management, and dynamic QoS management, are also legitimate candidates for architectural intent.

Not all of these views are necessarily reflected in the final system, let alone within a specific prerelease prototype. For example, the first prototype might specify the subsystem and component architectural view, but the concurrency view might be deferred until microcycle 3, and the distribution view might be added in microcycle 4. The safety and reliability view might never be added if the system has no safety-critical or high-reliability requirements. This aspect of the microcycle mission statement clearly identifies the architectural intent for the prototype to be produced in this microcycle.

Note

A subsystem is a large-scale architectural element that allocates requests for services to internal parts for processing. This is essentially the same definition as a component. In the UML, a subsystem is a metasubclass of a component. In my usage, a subsystem is the first level of decomposition of a system, whereas components represent the decomposition of subsystems; that is, systems contain subsystems, and subsystems contain components.

External items that may be required for a microcycle include computing environments (such as embedded computer boards), sensors, actuators, simulators, or test equipment (for example, in-circuit emulators or logic analyzers). The microcycle mission statement must explicitly identify such needs to ensure that they are visible to the project manager and other project staff.

6.1.2. Specifying the User Interface

Not all embedded systems have user interfaces, so this task is optional. However, for systems with a significant UI, it provides crucial guidance on how to create the operator-machine interface. The human factor engineer, a trained professional in the analysis of user workflows, typically performs this task, which has three main steps:

1. Analyze user workflows.

2. Create the UI prototype.

3. Write the UI specifications.

User workflow analysis[2] seeks to understand how users interact or will interact with the system to perform their work functions. For example, what are the tasks that a UAV pilot must perform? Do the tasks differ depending on mission state? Do they differ depending on mission type? What is the information needed to perform the detailed tasks? How should that information be organized? What is the priority of that information? What can the system do to support decision making and pilot action? This analysis will result in a set of user interaction concepts that will drive the specification of the UI.

2. See, for example, JoAnn T. Hackos and Janice C. Redish, *User and Task Analysis for Interface Design* (New York: John Wiley & Sons, 1998), or Larry L. Constantine and Lucy A. D. Lockwood, *Software for* Use: A Practical Guide to the Models and *Methods of Usage-Centered Design* (Reading, MA: Addison-Wesley, 1999).

It is not uncommon to construct a “throwaway” prototype that illustrates the user interaction concepts. This executable system can be demonstrated to users for the elicitation of feedback. Such a prototype is “throwaway” in the sense that it is not shipped to the user (it has no real functionality and isn’t scalable to support the user need), but it can persist throughout the project as an executable user interface specification. For one medical project, I wrote about 30,000 lines of Visual Basic for an anesthesia system to demonstrate how the user (the anesthesiologist) could perform the various duties of configuring a device for a medical procedure, performing different kinds of procedures, and interacting with other remote systems (such as labs and the hospital information network). This code was not shipped with the system but did provide valuable feedback on the efficacy and acceptability of the UI concepts.

The primary output of this task is the UI specifications. This document details the user interaction concepts; lays out the screens, widgets, data fields, and user controls; and describes the constraints within which they must work. The behavior of the UI element is best described in formal language such as activity diagrams or state machines.[3] Use cases that interact with human users via these UI concepts must be realized consistently with this UI specification in addition to their functional and QoS requirements.

3. An interesting reference in this regard is Ian Horrocks, *Constructing the User Interface with Statecharts* (Reading, MA: Addison-Wesley, 1999).

6.1.3. Detailing the Use Cases

Detailing the use cases entails expanding them from the stakeholder requirements, which are cast entirely in terms of user vocabulary and user need, to include the necessary system precision. Only the use cases to be realized within the current prototype are elaborated at this time. Care must be taken to ensure that the new use cases are still focused on a black-box perspective and on system requirements, not on implementation. The later task "Use case white-box analysis," described below, allocates portions of the use cases to internal system architectural elements. This task is entirely focused on deriving the system use cases from the stakeholder use cases and interacts strongly with the parallel task, "Detail system requirements."

The Harmony/ESW process emphasizes model or system execution as the principal means for ensuring that the development team is doing the right thing. This is also true with respect to requirements; that is, the Harmony/ESW process emphasizes creating *executable use case models* as the way to get high-quality, correct, and consistent requirements. This is especially important if the stakeholder requirements are vague and ambiguous; if the system has novel and perhaps poorly understood needs; if the system provides significantly new technology, UI concepts, or functionality; or if the cost of poor requirements is high.[4] If the system is low-risk and a minor extension to already well-understood system concepts, then the execution may be omitted.

4. Transportation, medical, military, and aerospace systems typically have a very high cost for requirements that are unclear, imprecise, or ambiguous. In these domains, executable requirements models are highly encouraged.

The steps involved in detailing the use cases are:

1. Identify the primary ("sunny-day") scenarios.

2. Capture the main flows in an activity diagram (optional).

3. Identify the secondary and exception ("rainy-day") scenarios.

4. Validate the scenarios with the stakeholders.

5. Specify the use case state machine.

6. Establish traceability to the stakeholder requirements and use cases.

Identifying the Primary ("Sunny-Day") Scenarios

A scenario is an exemplar for a use case; that is, it is a path through the use case that captures a specific set of messages in a specific order, annotated with optional constraints. If you change either the messages or their order, then it becomes a different scenario. A use case is normally detailed with between a half-dozen and several dozen primary scenarios, and an equal or greater number of secondary scenarios. Each scenario should have at least three messages, but most will have between a half-dozen and two dozen messages. Also, because the point of view of these scenarios is black-box, the only elements allowed to send or receive messages are the system (or the use case—either may be used in this context) and the actors.[5]

5. An **actor** is an element in the system environment with which the system interacts. When a use case is drawn with an association to an actor, it means that the system and that actor exchange messages during the course of executing that use case.

By far the most common way to represent scenarios in the UML model is as sequence diagrams.[6] It is sometimes useful to use UML 2's interaction operators such as *opt* (optional), *alt* (alternative), *loop,* or *parallel.* These allow more expressiveness in the sequence diagrams but are basically a structured way to show more than one scenario on a single diagram. While this is OK to some degree, it can be easily overdone. I recommend nesting these operators no more than three deep; more than that makes the scenario very difficult to understand. Further, because these scenarios will be used downstream both for driving the engineering work and for test cases, it is better to have more but simpler scenarios than fewer but impenetrably complex ones.

6. Since this book focuses on process, I don't explain the syntax or semantics of the UML elements discussed. For an in-depth tutorial on the topic, see my *Real-Time UML, Third Edition,* and for practice building real-time and embedded systems, refer to my book *Real-Time UML Workshop for Embedded Systems.*

Messages can be synchronous or asynchronous, as appropriate. Individual messages or sets of messages can be restricted by constraints. In this context, these constraints are usually QoS (e.g., maximum execution time, required bandwidth, or reliability), data limitations (e.g., subranges or default values), or pre- and postconditions. The point is to understand the flow of the scenario well enough to support the identification of internal elements and their behavior as well as the testing of those elements.

I also recommend a comment on the diagram (although some people prefer to put this in the internally stored description field for the diagram) that contains an overview of the diagram, including:

• Name of the use case being detailed

• Name of the sequence diagram

• Name of the scenario

• Description

• Preconditions

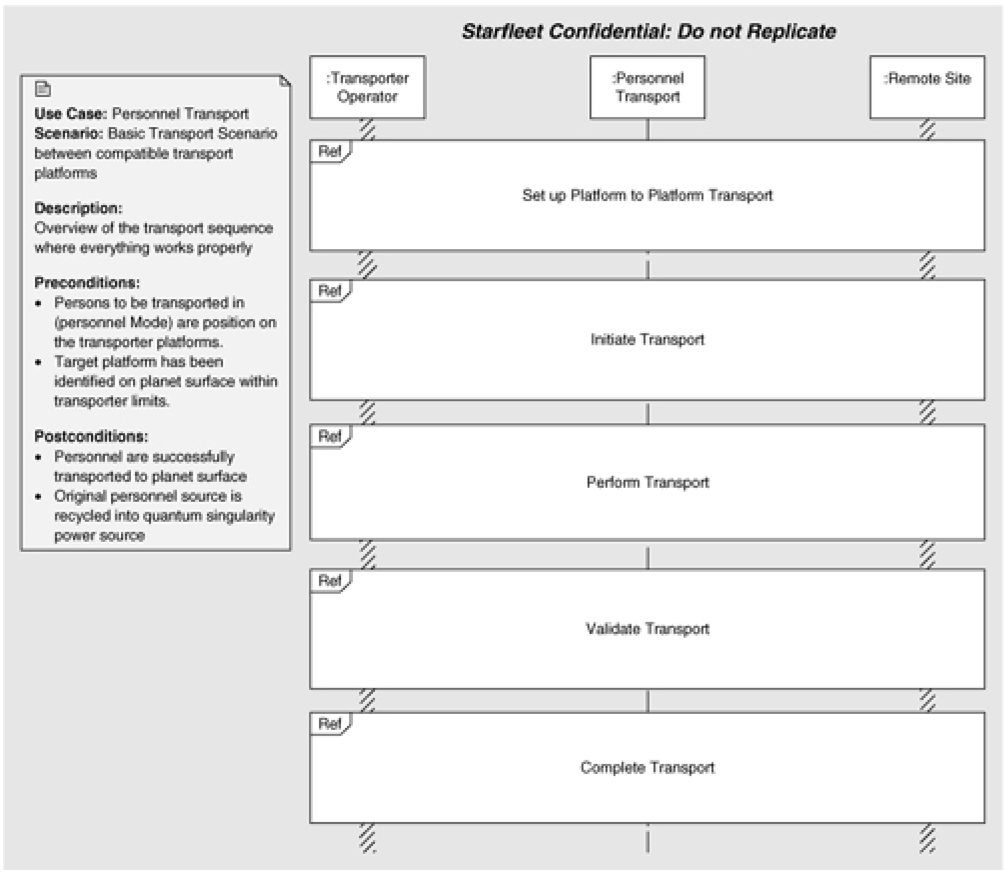
• Postconditions

• Other constraints not otherwise visible on the diagram

> **Note**
>
> Most of the examples in this and later chapters are for the Starfleet ZX-1000 Transporter System. This is a matter transport device that works by creating a quantum tunnel, binding quarks from the source location to the target, and transporting the matter stream via the quantum tunnel. The requirements specification for this system is in Appendix A.

Figure 6.3 shows the high-level sunny-day scenario for the `Personnel Transport` use case shown in Figure 5.10 from the previous chapter. Because this scenario is rather long, it is decomposed into steps, each of which is captured in a nested scenario. These scenarios are shown in Figures 6.4 through 6.8. The scenarios for simpler use cases will often fit on a single page, but for a medium- to large-scale system, these figures representing a single use case scenario are typical in terms of their length and complexity. I'm not going to kid you—complex systems are *complex*. And messing around with the 4000 petajoules required for transport is a highly safety-critical concern; one major mistake and there goes the space-time continuum. (But at least your luggage always arrives with you!)

Figure 6.3 *Basic Personnel Transport scenario (high-level)*



Use Case: Personnel Transport
Scenario: Basic Transport Scenario between compatible transport platforms

Description:
Overview of the transport sequence where everything works properly

Preconditions:
- Persons to be transported in (personnel Mode) are position on the transporter platforms.
- Target platform has been identified on planet surface within transporter limits.

Postconditions:
- Personnel are successfully transported to planet surface
- Original personnel source is recycled into quantum singularity power source

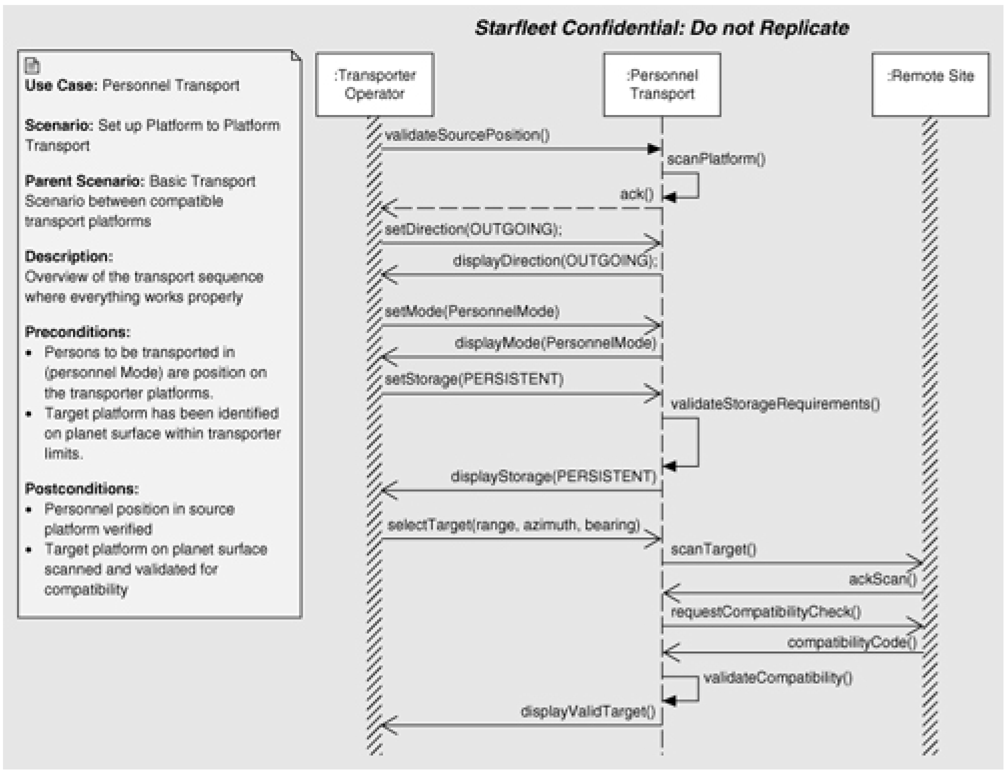
The actual use case would have scenarios not only for the sunny-day platform-to-platform transport but also for the platform-to-site (no target platform) and site-to-platform (no source platform) scenarios and various other variants. In addition, there would certainly be several dozen error scenarios to show what happens if the personnel are not properly within the transport chamber, if the platform is already locked out, if the range is too far or the interstellar medium too dense, if the rematerialization error rate is too high, and so on. These figures show only one of the sunny-day cases.

Most technical books show you trivial examples. In this book, I've tried to give real-world-scale examples; this means that I will be unable in such a short book to show you the entire model for the system. It also means that many times the diagram will not fit conveniently on a single page. The scenarios shown in this chapter and the later diagrams are typical for a use case of this scale. Remember that each use case typically represents anywhere from 4 to 20 pages of requirements.

Figure 6.3 shows the “master” for the scenario, in the sense that the subsequent figures are represented by the sequence diagram references (Ref blocks) in the figure.

Figure 6.4 shows the first referenced sequence diagram. It focuses on setting up for a platform-to-platform transport.

Figure 6.4 *Set Up Platform-to-Platform Transport scenario*



Use Case: Personnel Transport

Scenario: Set up Platform to Platform Transport

Parent Scenario: Basic Transport Scenario between compatible transport platforms

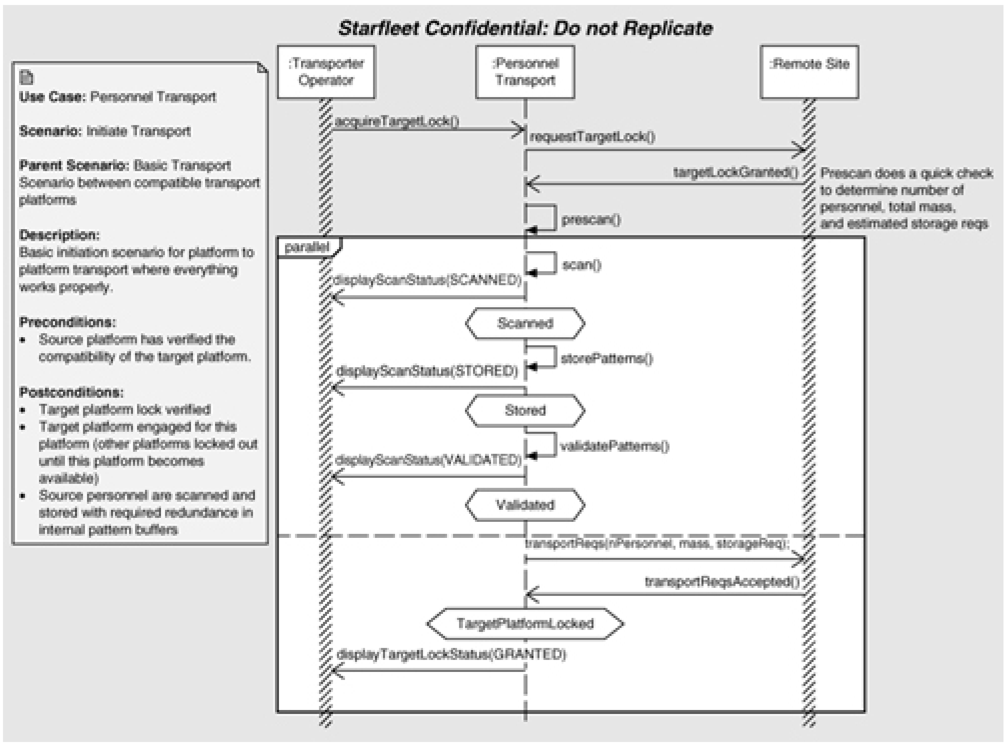
Description:
Overview of the transport sequence where everything works properly

Preconditions:
- Persons to be transported in (personnel Mode) are position on the transporter platforms.
- Target platform has been identified on planet surface within transporter limits.

Postconditions:
- Personnel position in source platform verified
- Target platform on planet surface scanned and validated for compatibility

Figure 6.5 is the second referenced sequence diagram. It focuses on initiation for transportation.

Figure 6.5 *Initiate Transport scenario*


Starfleet Confidential: Do not Replicate
:Transporter Operator
:Personnel Transport
:Remote Site
acquireTargetLock()
requestTargetLock()
targetLockGranted()
Prescan does a quick check to determine number of personnel, total mass, and estimated storage reqs
prescan()
parallel
scan()
displayScanStatus(SCANNED)
Scanned
storePatterns()
displayScanStatus(STORED)
Stored
validatePatterns()
displayScanStatus(VALIDATED)
Validated
transportReqs(nPersonnel, mass, storageReq);
transportReqsAccepted()
TargetPlatformLocked
displayTargetLockStatus(GRANTED)

Use Case: Personnel Transport

Scenario: Initiate Transport

Parent Scenario: Basic Transport Scenario between compatible transport platforms

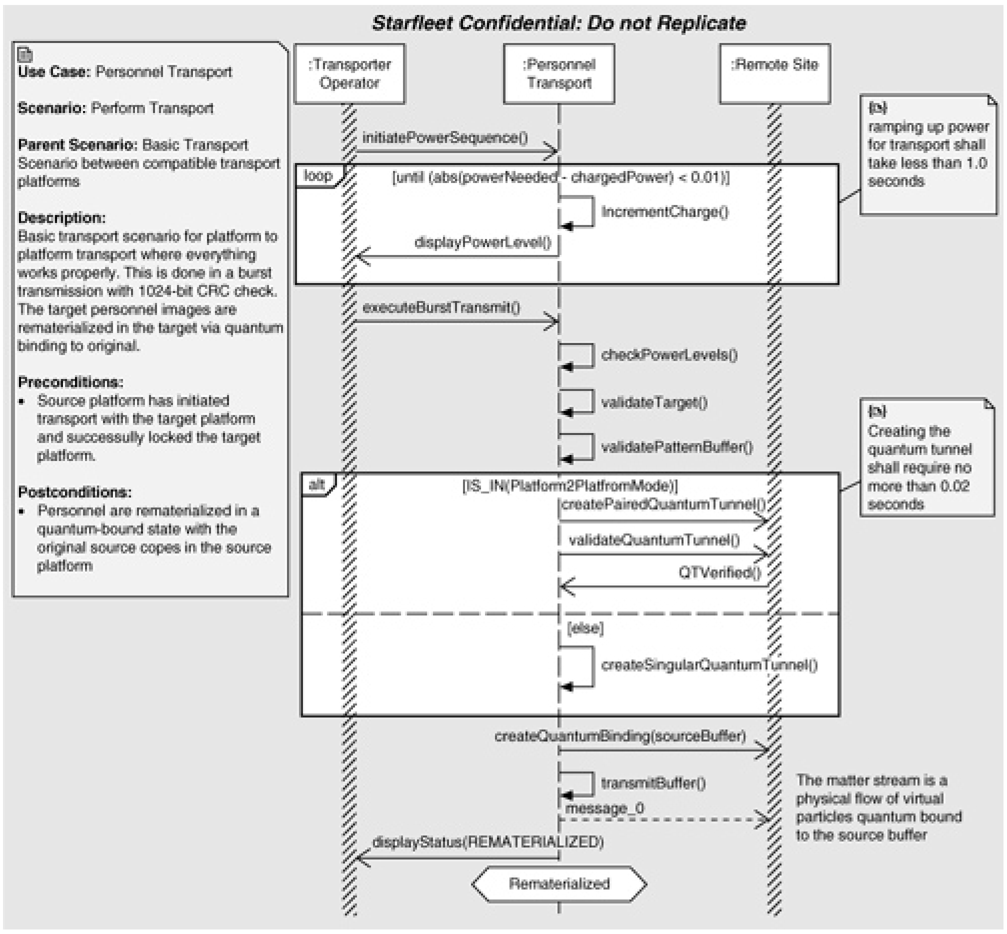
Description:
Basic initiation scenario for platform to platform transport where everything works properly.

Preconditions:
- Source platform has verified the compatibility of the target platform.

Postconditions:
- Target platform lock verified
- Target platform engaged for this platform (other platforms locked out until this platform becomes available)
- Source personnel are scanned and stored with required redundance in internal pattern buffers

Figure 6.6 is the third referenced sequence diagram. It shows how the transport itself is actually accomplished as an interaction of the system with the actors.

Figure 6.6 *Perform Transport scenario*


Starfleet Confidential: Do not Replicate
:Transporter Operator
:Personnel Transport
:Remote Site
initiatePowerSequence()
loop
[until (abs(powerNeeded - chargedPower) < 0.01)]
IncrementCharge()
displayPowerLevel()
{s}
ramping up power for transport shall take less than 1.0 seconds
executeBurstTransmit()
checkPowerLevels()
validateTarget()
validatePatternBuffer()
{s}
Creating the quantum tunnel shall require no more than 0.02 seconds
alt
[IS_IN(Platform2PlatfromMode)]
createPairedQuantumTunnel()
validateQuantumTunnel()
QTVerified()
[else]
createSingularQuantumTunnel()
createQuantumBinding(sourceBuffer)
transmitBuffer()
message_0
The matter stream is a physical flow of virtual particles quantum bound to the source buffer
displayStatus(REMATERIALIZED)
Rematerialized

Use Case: Personnel Transport

Scenario: Perform Transport

Parent Scenario: Basic Transport Scenario between compatible transport platforms

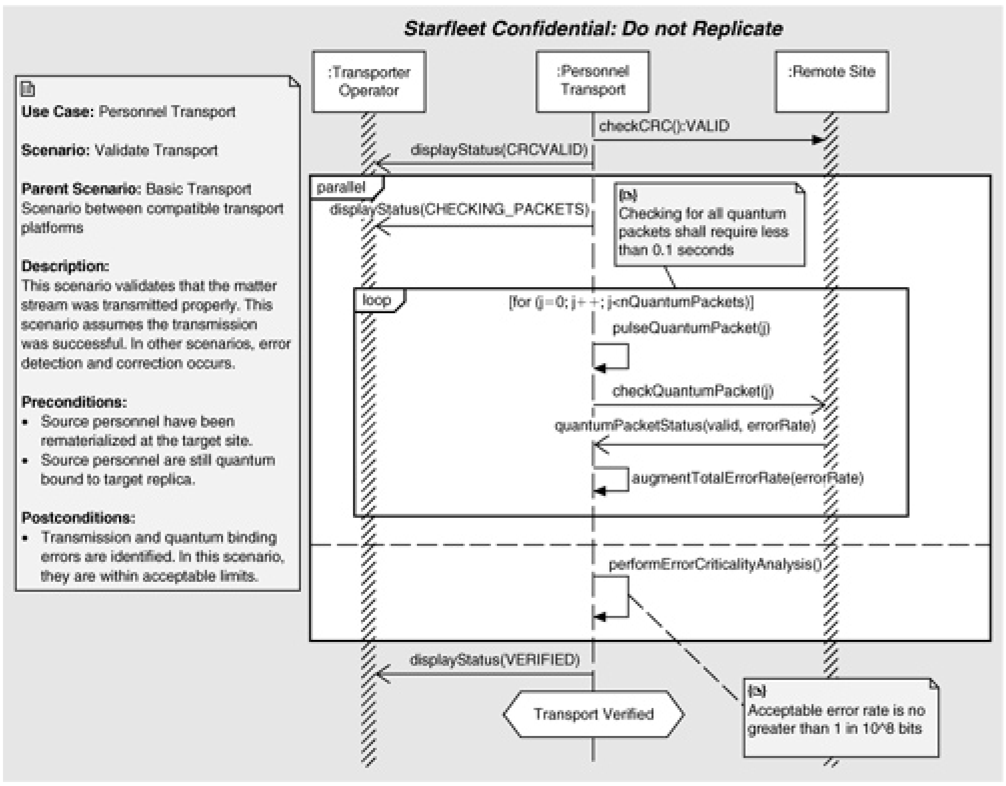
Description:
Basic transport scenario for platform to platform transport where everything works properly. This is done in a burst transmission with 1024-bit CRC check. The target personnel images are rematerialized in the target via quantum binding to original.

Preconditions:
- Source platform has initiated transport with the target platform and successully locked the target platform.

Postconditions:
- Personnel are rematerialized in a quantum-bound state with the original source copes in the source platform

Figure 6.7 is the fourth referenced sequence diagram. It shows how the completed transport is validated as complete and correct.

Figure 6.7 *Validate Transport scenario*


Starfleet Confidential: Do not Replicate
:Transporter Operator
:Personnel Transport
:Remote Site
checkCRC():VALID
displayStatus(CRCVALID)
parallel
displayStatus(CHECKING_PACKETS)
loop
[for (j=0; j++; j<nQuantumPackets)]
pulseQuantumPacket(j)
checkQuantumPacket(j)
quantumPacketStatus(valid, errorRate)
augmentTotalErrorRate(errorRate)
performErrorCriticalityAnalysis()
displayStatus(VERIFIED)
Transport Verified

Use Case: Personnel Transport

Scenario: Validate Transport

Parent Scenario: Basic Transport Scenario between compatible transport platforms

Description:
This scenario validates that the matter stream was transmitted properly. This scenario assumes the transmission was successful. In other scenarios, error detection and correction occurs.

Preconditions:
- Source personnel have been rematerialized at the target site.
- Source personnel are still quantum bound to target replica.

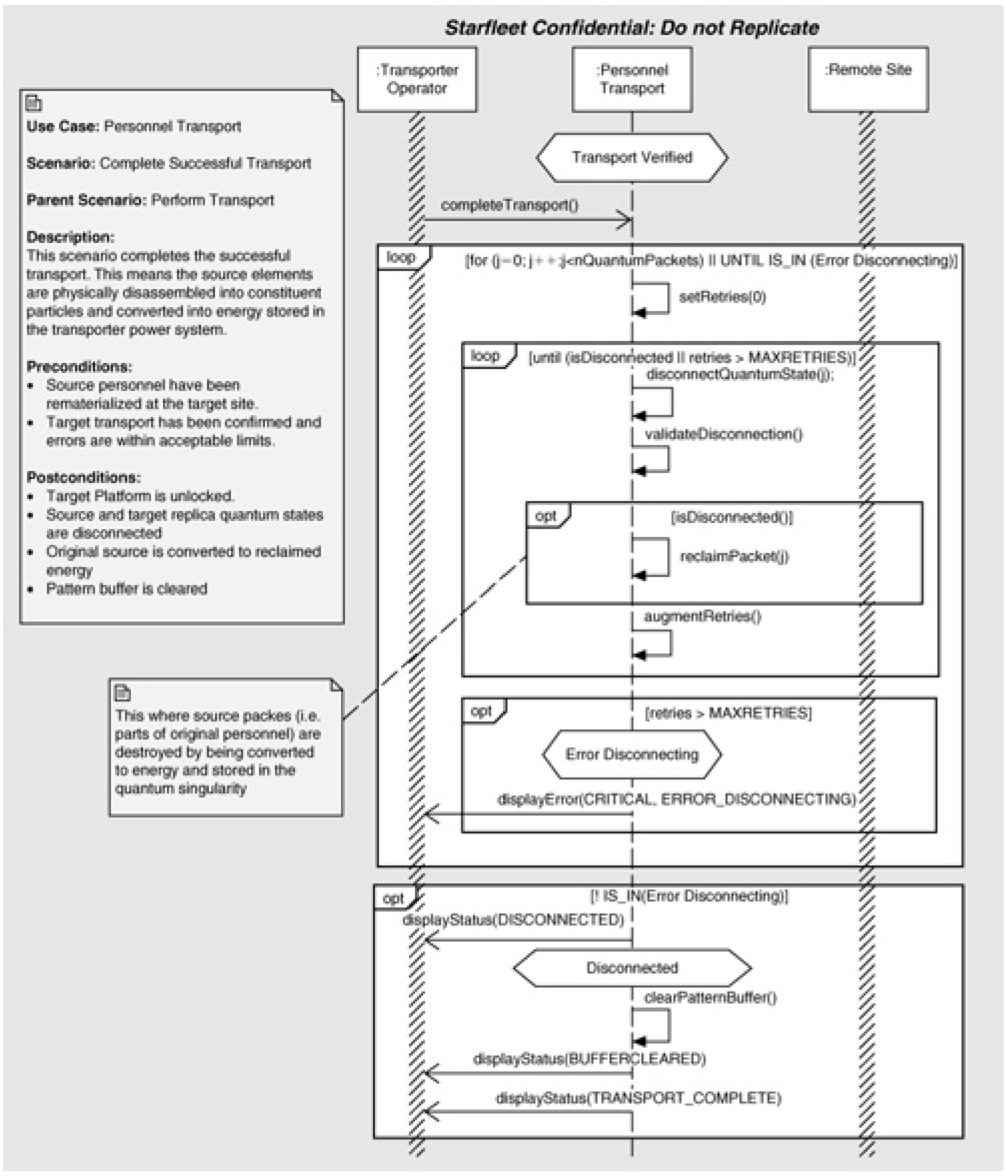
Postconditions:
- Transmission and quantum binding errors are identified. In this scenario, they are within acceptable limits.

{t≤}
Checking for all quantum packets shall require less than 0.1 seconds

{t≤}
Acceptable error rate is no greater than 1 in 10^8 bits

Figure 6.8 is the last sequence diagram referenced in Figure 6.3. It shows how the completion and cleanup of the (successful) transport occur.

Figure 6.8 *Complete Successful Transport scenario*


Starfleet Confidential: Do not Replicate
:Transporter Operator
:Personnel Transport
:Remote Site
Transport Verified
completeTransport()
loop
[for (j=0; j++;j<nQuantumPackets) II UNTIL IS_IN (Error Disconnecting)]
setRetries(0)
loop
[until (isDisconnected II retries > MAXRETRIES)]
disconnectQuantumState(j);
validateDisconnection()
opt
[isDisconnected()]
reclaimPacket(j)
augmentRetries()
opt
[retries > MAXRETRIES]
Error Disconnecting
displayError(CRITICAL, ERROR_DISCONNECTING)
opt
[! IS_IN(Error Disconnecting)]
displayStatus(DISCONNECTED)
Disconnected
clearPatternBuffer()
displayStatus(BUFFERCLEARED)
displayStatus(TRANSPORT_COMPLETE)

Use Case: Personnel Transport

Scenario: Complete Successful Transport

Parent Scenario: Perform Transport

Description:
This scenario completes the successful transport. This means the source elements are physically disassembled into constituent particles and converted into energy stored in the transporter power system.

Preconditions:
- Source personnel have been rematerialized at the target site.
- Target transport has been confirmed and errors are within acceptable limits.

Postconditions:
- Target Platform is unlocked.
- Source and target replica quantum states are disconnected
- Original source is converted to reclaimed energy
- Pattern buffer is cleared

This where source packes (i.e. parts of original personnel) are destroyed by being converted to energy and stored in the quantum singularity

Capturing the Main Flows in an Activity Diagram (Optional)

To understand the major (sunny-day-only) flows, many people like to create an activity diagram of the flows. The actions of the activity diagram are the services invoked by the messages in the different scenarios. The branch points indicate different scenarios; each branch point indicates a variant in which a different path is taken based on a different message arriving or due to different conditions (e.g., state or value causing a different action to be executed).

This activity diagram is used as a scratch pad and is usually discarded once the set of scenarios, paths, actions, and branch points are understood. The activity diagram usually does not show the error (rainy-day) scenarios. It is used as a stepping-stone to identify additional scenarios and to move to the normative specification, the use case state machine.

Identifying the Secondary and Exception ("Rainy-Day") Scenarios

The rainy-day scenarios capture the flows that occur when something unexpectedly goes wrong, such as when a precondition is violated, a failure occurs, or an error becomes manifest. The rainy-day scenario usually shows how the unexpected event or condition is detected and what corrective actions should be taken.

The difficult part of drawing the rainy-day scenarios is that there are normally two orders of magnitude more rainy-day scenarios than sunny-day ones. This is dealt with by two primary approaches. The first is the notion of **exception set.** An exception set is a set of exception conditions that, while unique, are identified and handled in exactly the same way.

For example, imagine a patient ventilator that delivers a shaped breath with a certain frequency (respiration rate), volume (tidal volume), pressure, and gas mixture. The exception condition is that no gas is actually delivered to the patient. This can be attributed to a number of root causes:

• The gas supply is depleted.

• The gas supply delivery valve breaks.

• The gas hose disconnects from the gas supply.

• The gas hose disconnects from the ventilator gas input.

• The gas hose has a kink or obstruction.

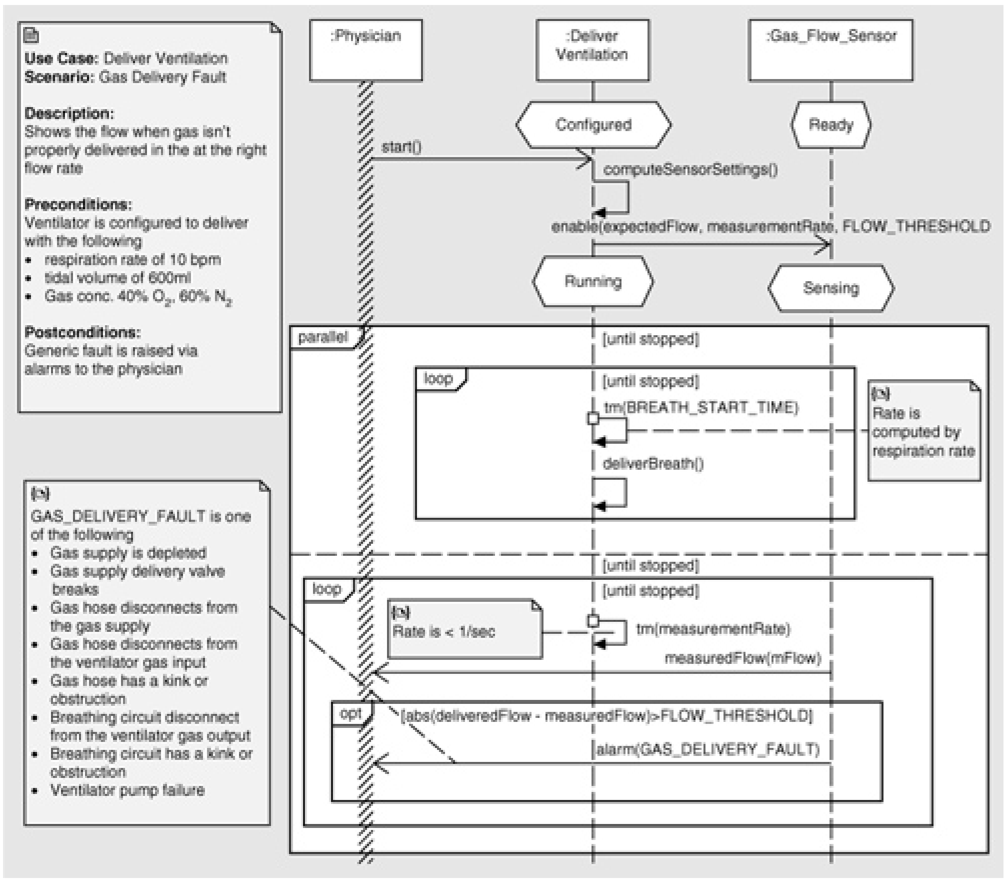
• The breathing circuit disconnects from the ventilator gas output.

• The breathing circuit has a kink or an obstruction.

• The ventilator pump fails.

In all these cases, both the detection means (flow sensor on the breathing circuit) and the corrective actions (alarm to raise the awareness of the attending physician) may be the same. In that case, only one scenario is necessary for all of these faults. The fault should be given a general name, such as `GAS_DELIVERY_FAULT`, and a constraint should be added to indicate the actual concrete faults that belong to this set. This is illustrated in Figure 6.9.

Figure 6.9 *Ventilator gas delivery fault scenarios*


Use Case: Deliver Ventilation
Scenario: Gas Delivery Fault
Description:
Shows the flow when gas isn't properly delivered in the at the right flow rate
Preconditions:
Ventilator is configured to deliver with the following
• respiration rate of 10 bpm
• tidal volume of 600ml
• Gas conc. 40% O_2, 60% N_2
Postconditions:
Generic fault is raised via alarms to the physician
{C}
GAS_DELIVERY_FAULT is one of the following
• Gas supply is depleted
• Gas supply delivery valve breaks
• Gas hose disconnects from the gas supply
• Gas hose disconnects from the ventilator gas input
• Gas hose has a kink or obstruction
• Breathing circuit disconnect from the ventilator gas output
• Breathing circuit has a kink or obstruction
• Ventilator pump failure
:Physician
:Deliver Ventilation
:Gas_Flow_Sensor
Configured
Ready
start()
computeSensorSettings()
enable(expectedFlow, measurementRate, FLOW_THRESHOLD
Running
Sensing
parallel
[until stopped]
loop
[until stopped]
tm(BREATH_START_TIME)
deliverBreath()
{C}
Rate is computed by respiration rate
[until stopped]
loop
[until stopped]
{C}
Rate is < 1/sec
tm(measurementRate)
measuredFlow(mFlow)
opt
[abs(deliveredFlow - measuredFlow)>FLOW_THRESHOLD]
alarm(GAS_DELIVERY_FAULT)

You can see that there are far more exception cases than normal cases. For just the one primary scenario illustrated in Figure 6.3 to Figure 6.8, look at all the things that can obviously go wrong:

• The personnel can be misaligned on the platforms.

• The transporter platform scanner can fail.

• The power can fail.

• The pattern buffer can fail.

• The targeting system can fail to lock.

• The targeting scanner can fail.

• The target can be incompatible.

• The system can fail to get a target lock grant.

• The storage of the pattern can fail.

• The stored pattern can fail to validate.

• The target platform can reject the transport requirements.

• The quantum tunnel can fail to stabilize.

• The quantum binding can fail.

• The quantum packets can fail CRC check.

• The error rate can be too high.

• The quantum binding can fail to unbind (error disconnecting).

• The packets can fail to reclaim.

• The pattern buffer can fail to clear.

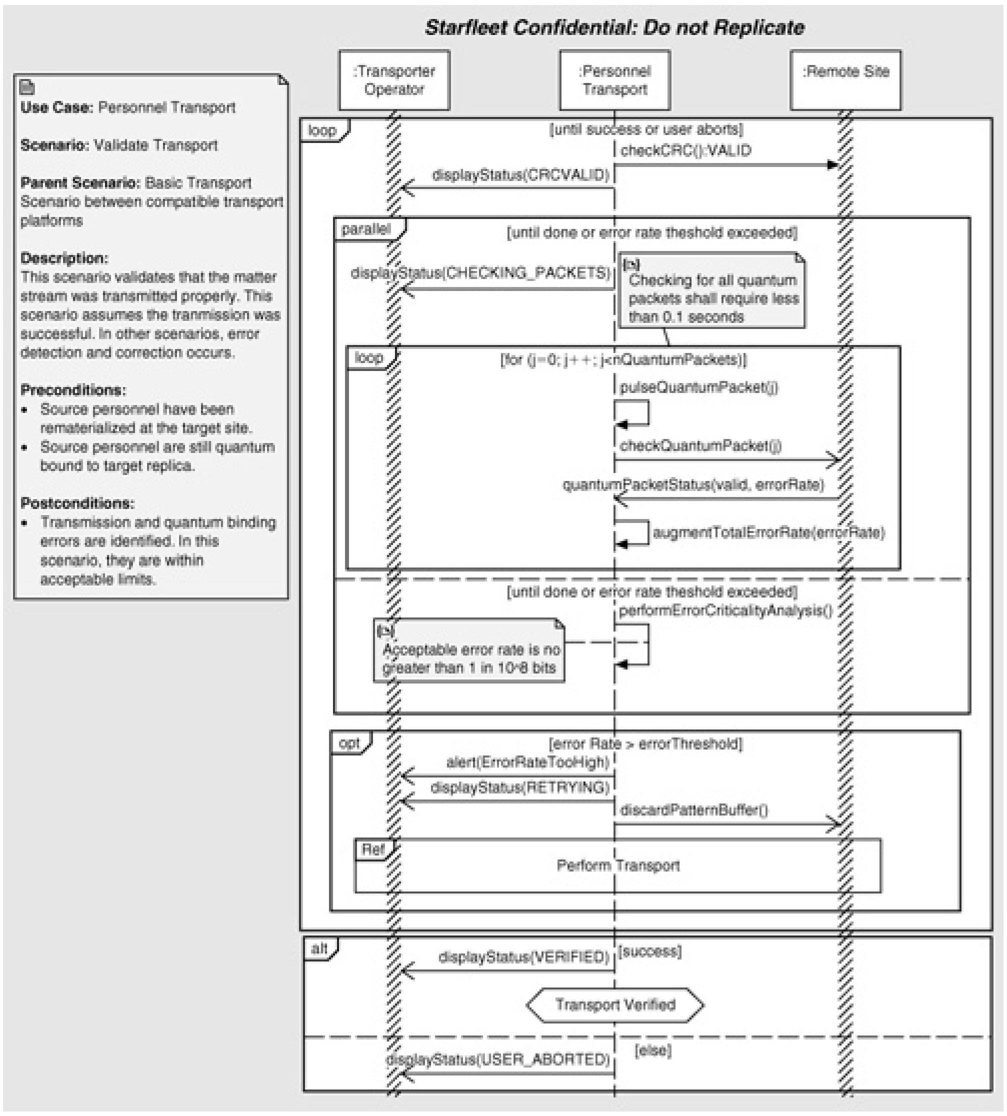
• The quantum singularity powering the whole system can explode, destroying the space-time continuum.

And those are just *some* of the essential fault scenarios!

Let's consider the situation in which the transporter scenario Validate Transport (Figure 6.7) fails because the error rate is too high (say, 1 in 10^7 bits instead of the limit of 10^8 bits). In that case, what's a Starfleet engineer to do? "Bend over and kiss your *sa-hut*[7] goodbye!" isn't a very satisfying answer, regardless of how much fun it might be to say. Figure 6.10 shows the scenario for this case, a variant of Figure 6.7. Note that this scenario includes a reference to Figure 6.6 to show that the source pattern buffer is retransmitted.

7. See the Klingon Language Institute on the galactic Web, http://www.kli.org/.

Figure 6.10 *Validate Transport high error rate scenario*


Starfleet Confidential: Do not Replicate
:Transporter Operator
:Personnel Transport
:Remote Site
loop
[until success or user aborts]
checkCRC():VALID
displayStatus(CRCVALID)
parallel
[until done or error rate theshold exceeded]
displayStatus(CHECKING_PACKETS)
Checking for all quantum packets shall require less than 0.1 seconds
loop
[for (j=0; j++; j<nQuantumPackets)]
pulseQuantumPacket(j)
checkQuantumPacket(j)
quantumPacketStatus(valid, errorRate)
augmentTotalErrorRate(errorRate)
[until done or error rate theshold exceeded]
performErrorCriticalityAnalysis()
Acceptable error rate is no greater than 1 in 10^8 bits
opt
[error Rate > errorThreshold]
alert(ErrorRateTooHigh)
displayStatus(RETRYING)
discardPatternBuffer()
Ref
Perform Transport
alt
displayStatus(VERIFIED)
[success]
Transport Verified
[else]
displayStatus(USER_ABORTED)

Use Case: Personnel Transport

Scenario: Validate Transport

Parent Scenario: Basic Transport Scenario between compatible transport platforms

Description:
This scenario validates that the matter stream was transmitted properly. This scenario assumes the tranmission was successful. In other scenarios, error detection and correction occurs.

Preconditions:
- Source personnel have been rematerialized at the target site.
- Source personnel are still quantum bound to target replica.

Postconditions:
- Transmission and quantum binding errors are identified. In this scenario, they are within acceptable limits.

The second way that rainy-day cases can be dealt with doesn't involve drawing every scenario, but rather ensuring that the use case state machine represents every case. Do you need to do this? Absolutely—provided that you want to specify what the system should do in fault situations. This kind of depth of analysis is required for all essential faults (faults that can be present in all acceptable design solutions) and especially in safety-critical or high-reliability systems. Later in design, more fault scenarios will be added to address the inessential faults—the faults that are potentially introduced because of specific design decisions. The creation of the use case state machine will be discussed shortly.

Validating the Scenarios with the Stakeholders

One of the best aspects of using sequence diagrams to capture scenarios is that nontechnical stakeholders (e.g., customers and marketing staff) can be readily taught to read and understand them. This makes them more valuable than written textual specifications for stakeholder review for this purpose.[8] The customers and users know their intended workflows and are often domain experts, so they can provide invaluable feedback as to the correctness and reasonableness of the scenarios.

8. However, it is best to have both sequence diagrams and the stakeholder requirements text for this purpose.

Specifying the Use Case State Machine

The use case state machine is a normative specification of the system executing the use case. Each scenario may be considered to be a transition path within that state machine. This includes both sunny-day and rainy-day scenarios. For complex systems, this can be . . . well, *complex*. For this reason, I recommend the use of the UML state machine features to help manage the complexity, using features such as nested states, and-states (states executing in parallel), and submachines (nested states shown on separate diagrams with a link from the primary composite state in the original diagram).

Messages from the actors to the system will show up as *events* triggering transitions on the state machine. Messages from the system to the actors are *actions* on the state machine. Testable conditions used to select branches from transition junctions or to qualify when a transition should be taken are shown as *guards* on the relevant transitions.

If you want to create an executable use case, then I recommend that you create a class that represents the use case and specify that class with the state machine. UML does permit use cases to be specified by state machine, but there are some technical advantages to using a class for this purpose. For one, a number of tools support code generation and execution of classes but not use cases. Second, UML doesn't provide a notation to show connection points (ports) on use cases, but there is such a notation for classes. This is valuable because it means that you can use the class diagram to show the interfaces provided and required by the different ports connected to different actors. This information is valuable in the construction of the interface documentation that specifies how the actors and the system will interact.

Figure 6.11 shows the high-level state machine for the use case `Personnel Transport`. The icons at the bottom of several of the states on that diagram indicate that they have nested state machines (known as **submachines**) shown in other diagrams. These are shown in Figure 6.12 through Figure 6.17.

Figure 6.11 *Platform-to-Platform Transport use case state machine*

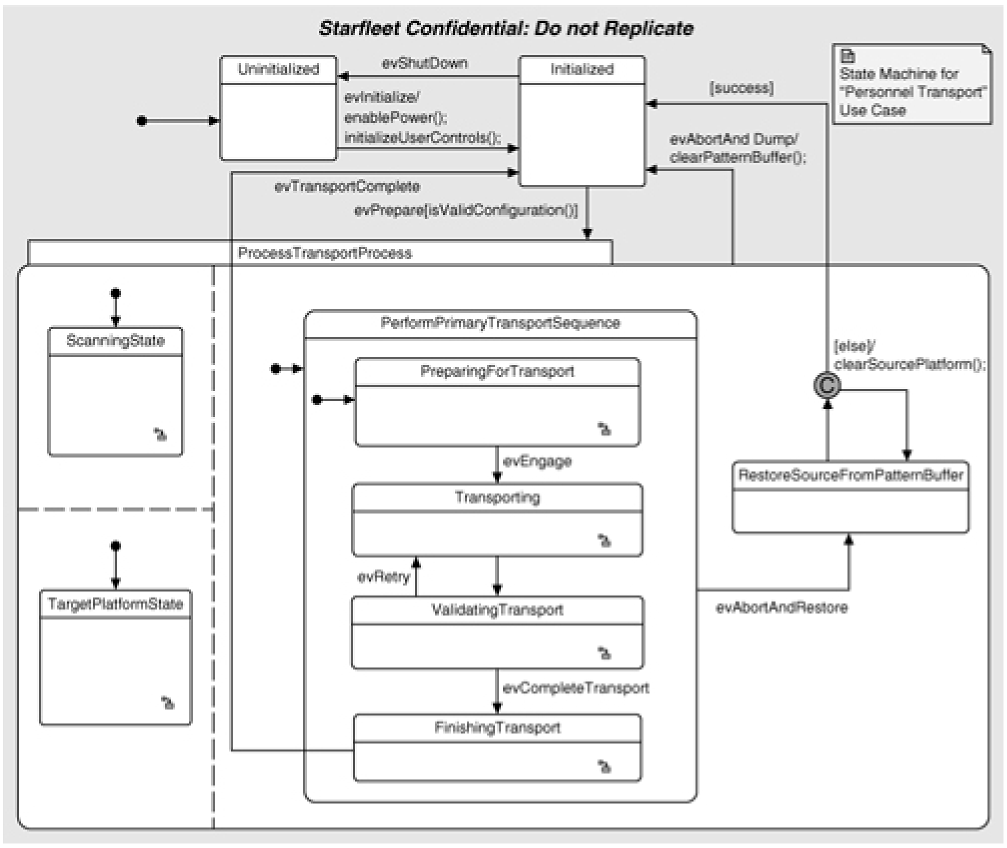


Figure 6.12 shows the submachine for the `ScanningState`. It focuses on the orthogonal processing of scanning the source location for material to transport.

Figure 6.12 `ScanningState` *state machine*

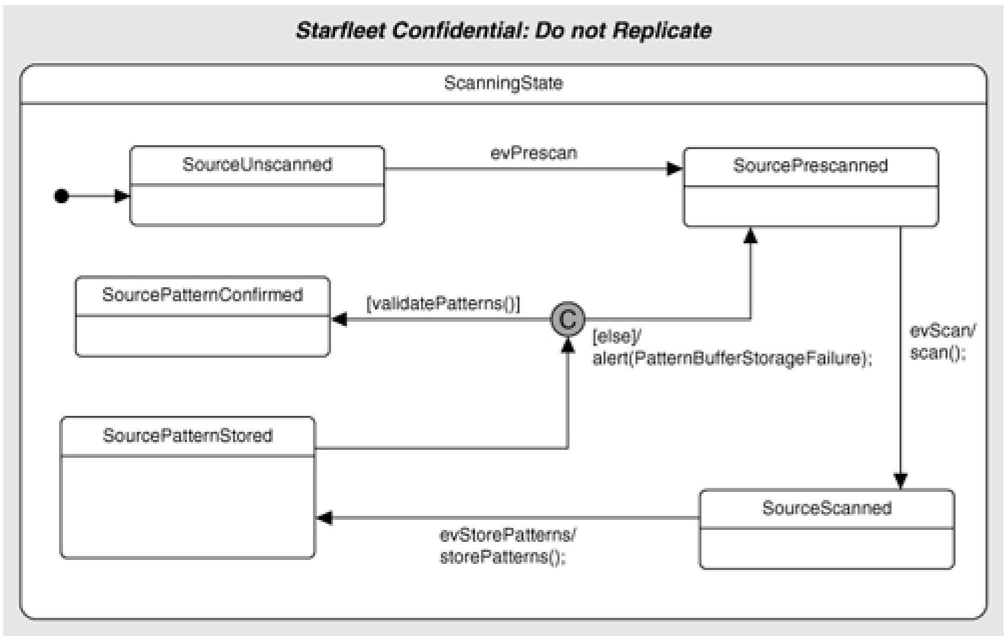


Figure 6.13 shows the submachine for states of the target platform.

Figure 6.13 `TargetPlatformState` *state machine*

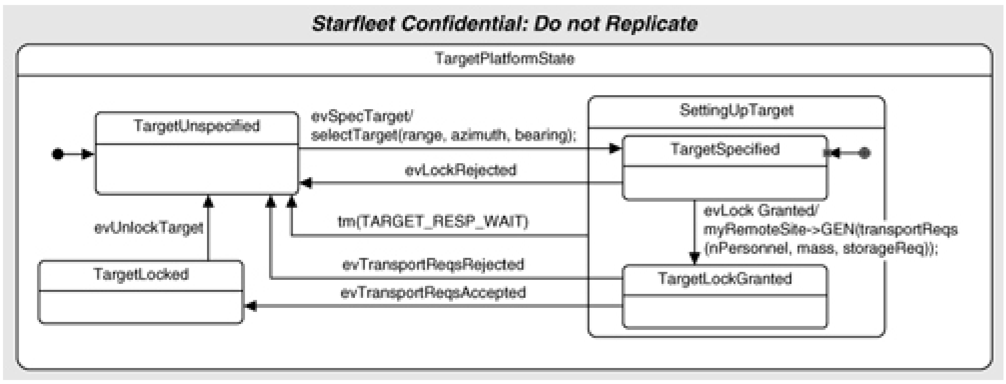


Figure 6.14 shows the details of the submachine for the `PreparingForTransport` state. The and-states execute in parallel. You can see that the state machine manages a number of different concerns regarding the preparation for transport, such as the validity of the source platform, the transportation mode, the pattern storage, whether the remote platform exists, and whether the matter stream is incoming or outgoing.

Figure 6.14 `PreparingForTransport` *state machine*

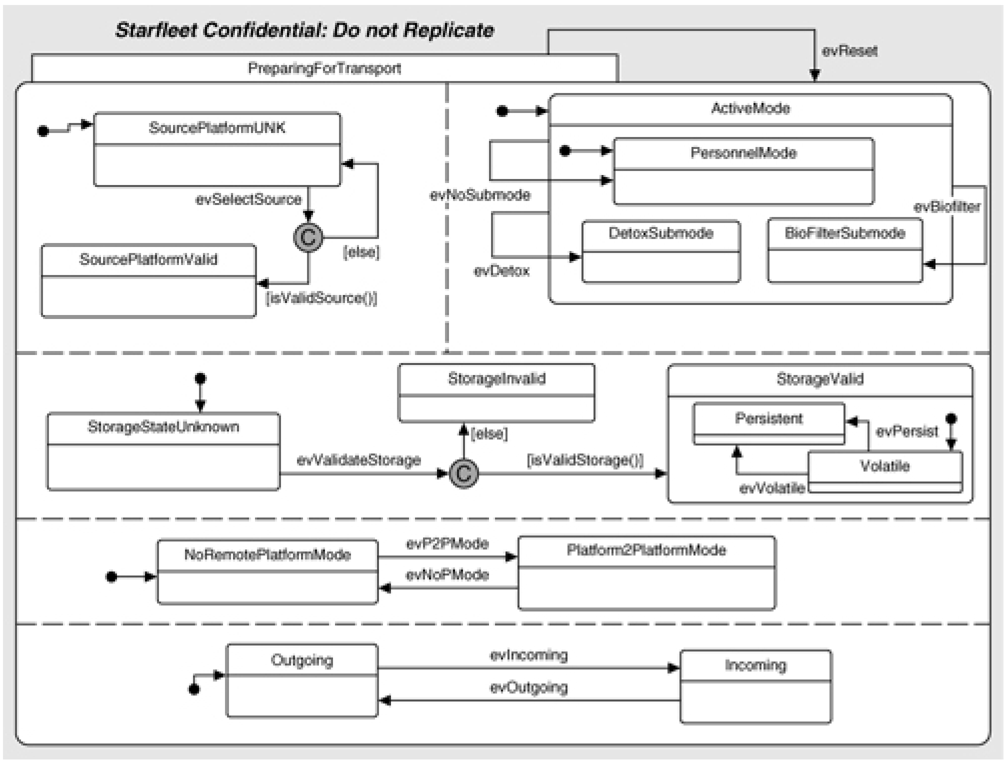
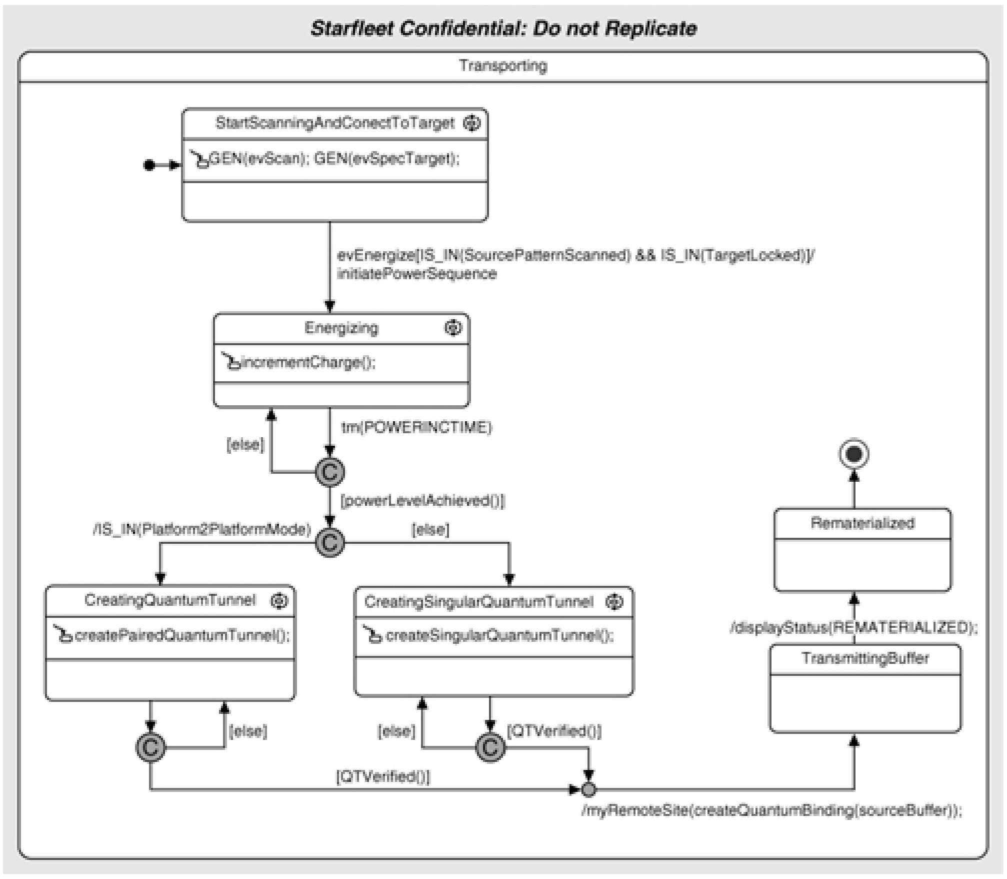


Figure 6.15 shows the submachine for the transport itself, sequencing the activities that perform the actual matter transportation.

Figure 6.15 `Transporting` *state machine*



Starfleet Confidential: Do not Replicate
Transporting
StartScanningAndConectToTarget
GEN(evScan); GEN(evSpecTarget);
evEnergize[IS_IN(SourcePatternScanned) && IS_IN(TargetLocked)]/ initiatePowerSequence
Energizing
incrementCharge();
tm(POWERINCTIME)
[else]
[powerLevelAchieved()]
/IS_IN(Platform2PlatformMode)
[else]
CreatingQuantumTunnel
createPairedQuantumTunnel();
CreatingSingularQuantumTunnel
createSingularQuantumTunnel();
[else]
[else]
[QTVerified()]
[QTVerified()]
/myRemoteSite(createQuantumBinding(sourceBuffer));
TransmittingBuffer
/displayStatus(REMATERIALIZED);
Rematerialized

Following the transportation, the result must be validated. Figure 6.16 shows the how this occurs in terms of checking each quantum packet for errors and performing error criticality analysis to determine success or failure.

Figure 6.16 `ValidatingTransport` *state machine*

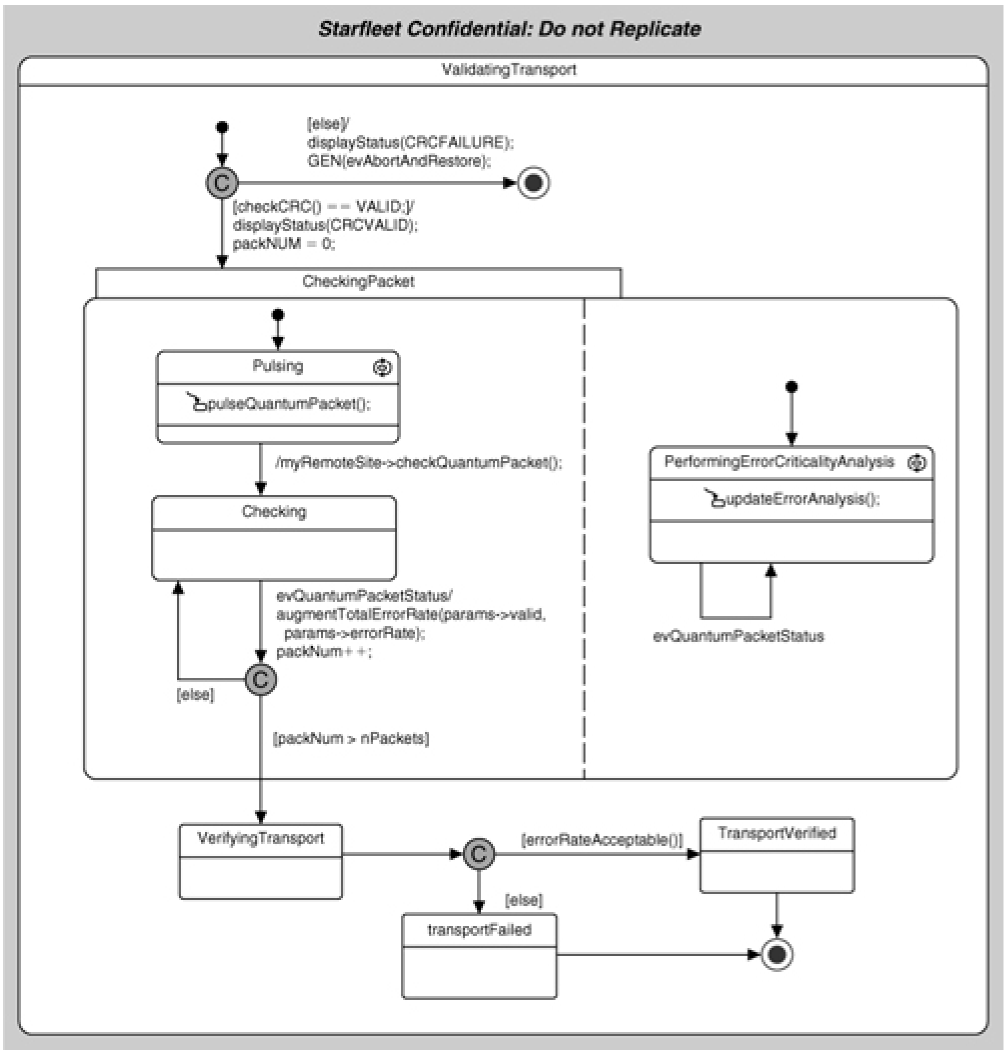
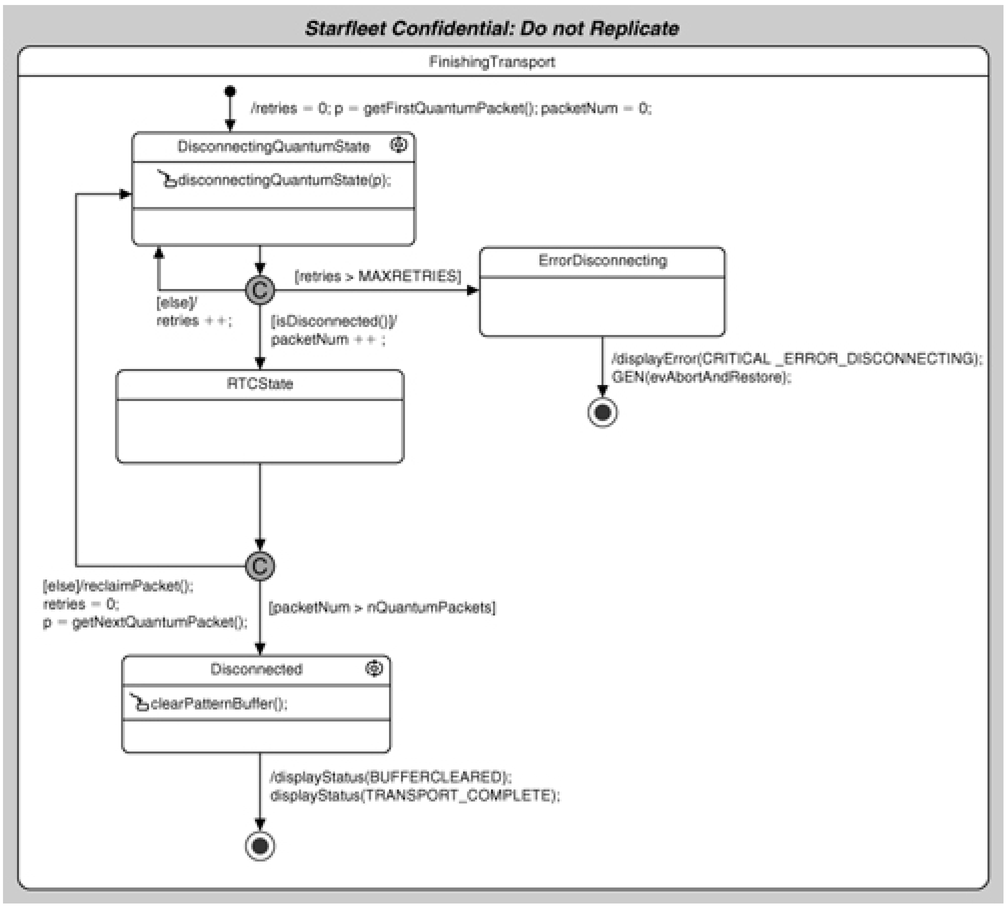


Figure 6.17 shows the sequence of actions necessary to complete the matter transportation process, including both successful and unsuccessful terminations.

Figure 6.17 `FinishingTransport` *state machine*


Starfleet Confidential: Do not Replicate
FinishingTransport
/retries = 0; p = getFirstQuantumPacket(); packetNum = 0;
DisconnectingQuantumState
disconnectingQuantumState(p);
[retries > MAXRETRIES]
ErrorDisconnecting
C
[else]/
retries ++;
[isDisconnected()]/
packetNum ++ ;
/displayError(CRITICAL _ERROR_DISCONNECTING);
GEN(evAbortAndRestore);
RTCState
C
[else]/reclaimPacket();
retries = 0;
p = getNextQuantumPacket();
[packetNum > nQuantumPackets]
Disconnected
clearPatternBuffer();
/displayStatus(BUFFERCLEARED);
displayStatus(TRANSPORT_COMPLETE);

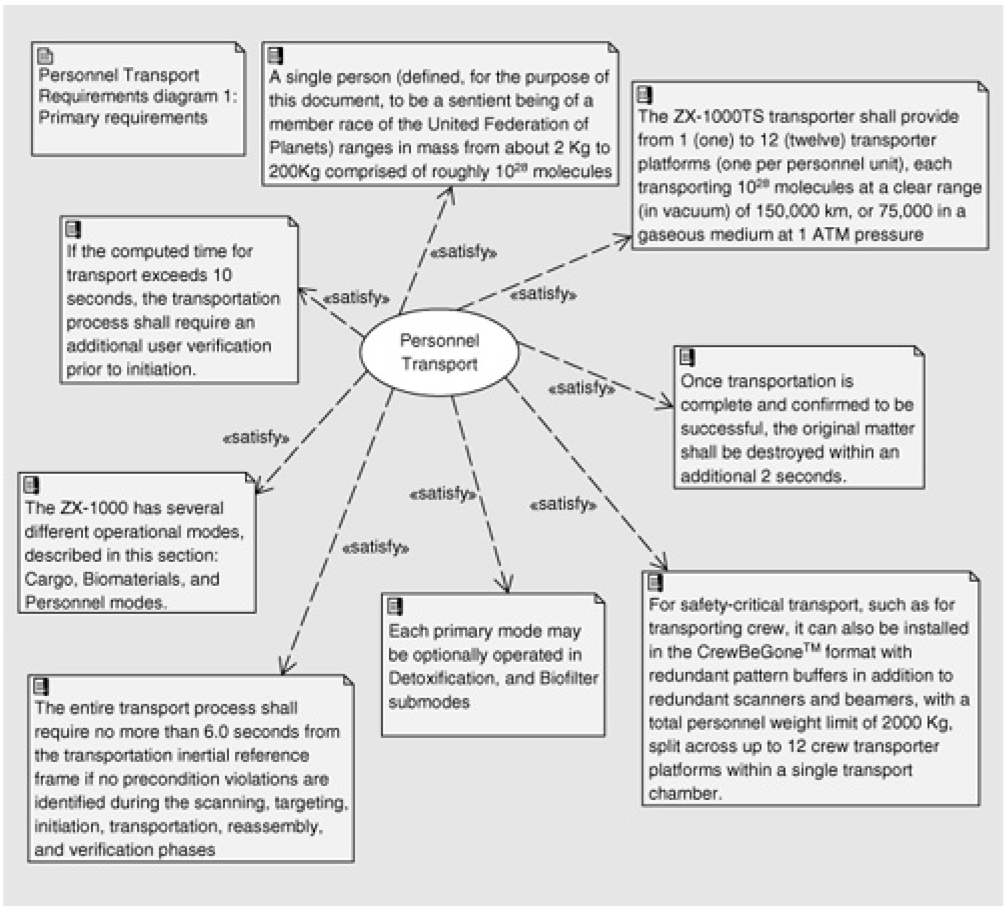
Establishing Traceability to the Stakeholder Requirements and Use Cases

Traceability is useful for both change impact analysis and to demonstrate that a system meets the requirements or that the test suite covers all the requirements. It is also useful to demonstrate that each design element is there to meet one or more requirements, something that is required by some safety standards, such as DO-178B.[9]

9. DO-178B is an RTCA standard used for avionics and other safety-critical domains. See http://www.rtca.org/.

Traceability can be done either through the creation of requirements diagrams, such as the one shown in Figure 6.18; with traceability tools, such as Telelogic DOORS[10] from IBM Rational; or through the creation of requirements traceability matrices, such as that shown in Figure 6.19.

Figure 6.18 `Personnel Transport` *requirements diagram*


Personnel Transport Requirements diagram 1: Primary requirements
Personnel Transport
«satisfy»

A single person (defined, for the purpose of this document, to be a sentient being of a member race of the United Federation of Planets) ranges in mass from about 2 Kg to 200Kg comprised of roughly 10^{28} molecules

The ZX-1000TS transporter shall provide from 1 (one) to 12 (twelve) transporter platforms (one per personnel unit), each transporting 10^{28} molecules at a clear range (in vacuum) of 150,000 km, or 75,000 in a gaseous medium at 1 ATM pressure

If the computed time for transport exceeds 10 seconds, the transportation process shall require an additional user verification prior to initiation.

Once transportation is complete and confirmed to be successful, the original matter shall be destroyed within an additional 2 seconds.

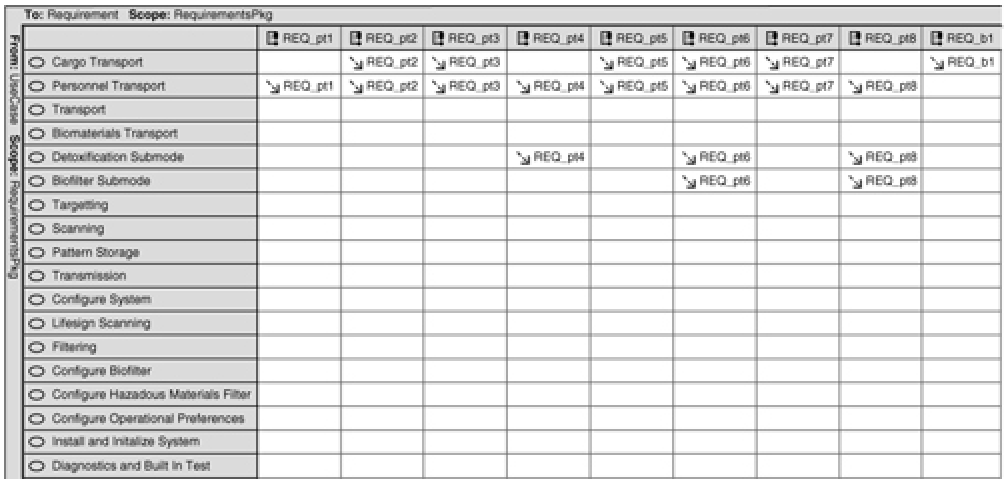
The ZX-1000 has several different operational modes, described in this section: Cargo, Biomaterials, and Personnel modes.

Each primary mode may be optionally operated in Detoxification, and Biofilter submodes

For safety-critical transport, such as for transporting crew, it can also be installed in the CrewBeGone™ format with redundant pattern buffers in addition to redundant scanners and beamers, with a total personnel weight limit of 2000 Kg, split across up to 12 crew transporter platforms within a single transport chamber.

The entire transport process shall require no more than 6.0 seconds from the transportation inertial reference frame if no precondition violations are identified during the scanning, targeting, initiation, transportation, reassembly, and verification phases

Figure 6.19 *Requirements Traceability Matrix*



To: Requirement **Scope:** RequirementsPkg

From: UseCase **Scope:** RequirementsPkg

	REQ_pt1	REQ_pt2	REQ_pt3	REQ_pt4	REQ_pt5	REQ_pt6	REQ_pt7	REQ_pt8	REQ_b1
Cargo Transport		REQ_pt2	REQ_pt3		REQ_pt5	REQ_pt6	REQ_pt7		REQ_b1
Personnel Transport	REQ_pt1	REQ_pt2	REQ_pt3	REQ_pt4	REQ_pt5	REQ_pt6	REQ_pt7	REQ_pt8	
Transport									
Biomaterials Transport									
Detoxification Submode				REQ_pt4		REQ_pt6		REQ_pt8	
Biofilter Submode						REQ_pt6		REQ_pt8	
Targetting									
Scanning									
Pattern Storage									
Transmission									
Configure System									
Lifesign Scanning									
Filtering									
Configure Biofilter									
Configure Hazadous Materials Filter									
Configure Operational Preferences									
Install and Initalize System									
Diagnostics and Built In Test									

10. DOORS is the preeminent requirements management tool in use in the real-time and embedded industry.

6.1.4. Generating System Requirements

This task creates the system requirements specification from the stakeholder requirements for the use cases to be realized in the current spiral. Use cases deferred to later microcycles are not within the scope of this effort. The purpose of the system requirements is to enable a system that meets the stakeholders' needs to be developed. This usually means that more precision, particularly in qualities of service and data type and ranges, is required than in the stakeholder requirements. As mentioned before, if the stakeholders are problem-domain experts (e.g., physicians for a medical system, or Starfleet engineers for a transporter system), then the users may already be accustomed to this degree of precision. If that is true, then the stakeholder and system requirements become a single document.

6.1.5. Managing Safety and Reliability Requirements

This task is essentially a follow-on from the "Perform initial safety and reliability analysis" activity from the prespiral planning activity. The same analytical means (e.g., FTA and FMEA) are used, and the hazard analysis is augmented to include the more precise requirements specified at the system level. Based on the system characteristics, additional hazards may be introduced; these will require additional safety measures that will be delineated in the system requirements.

6.1.6. Use Case White-Box Analysis

So far, the requirements have been "black-box"—that is, they've all been specified from the external viewpoint only. Stakeholder requirements are expressed in user vocabulary and focus on customer and user needs, whereas system requirements focus on specifying the system properties to meet those needs. In both cases, the requirements still can't "see" inside the system at all and are solely concerned with input/output data transformations, requestable services, and externally visible behaviors. White-box analysis begins to allocate the requirements into large-scale architectural elements (subsystems or components).

This is necessary for large projects that use teams of teams to build systems. It is very common to have different teams develop different subsystems and an integration team bring those architectural elements together for validation. When this is true, the subsystem teams need very clear specifications of what *they* are developing. Textually, these can be "subsystem specifications"—the same as the system specifications but limited in scope to a single subsystem. Large-scale subsystems have the same need for specifications as do systems, and so we will construct a use case model for each of the subsystems.

Note

If the subsystems are relatively simple, it may only be necessary to decompose system services into messages to various subsystems and allocate those services to the subsystems. They may not need to be clustered into use cases at the subsystem level. In general, if there are only a couple of dozen messages allocated to a subsystem, they can be managed without resorting to subsystem-level use cases. However, when the number of messages to which a subsystem must respond exceeds 50, then clearly subsystem use cases are called for.

Furthermore, be aware that if the project is small enough that the system doesn't require teams of teams, this step can be skipped and the team can go on to object analysis at this point.

The actions involved in white-box analysis include

• Understanding the subsystem architecture

• Bottom-up allocation or top-down allocation

• Detailing the subsystem-level use cases

• Validating the subsystem-level use cases

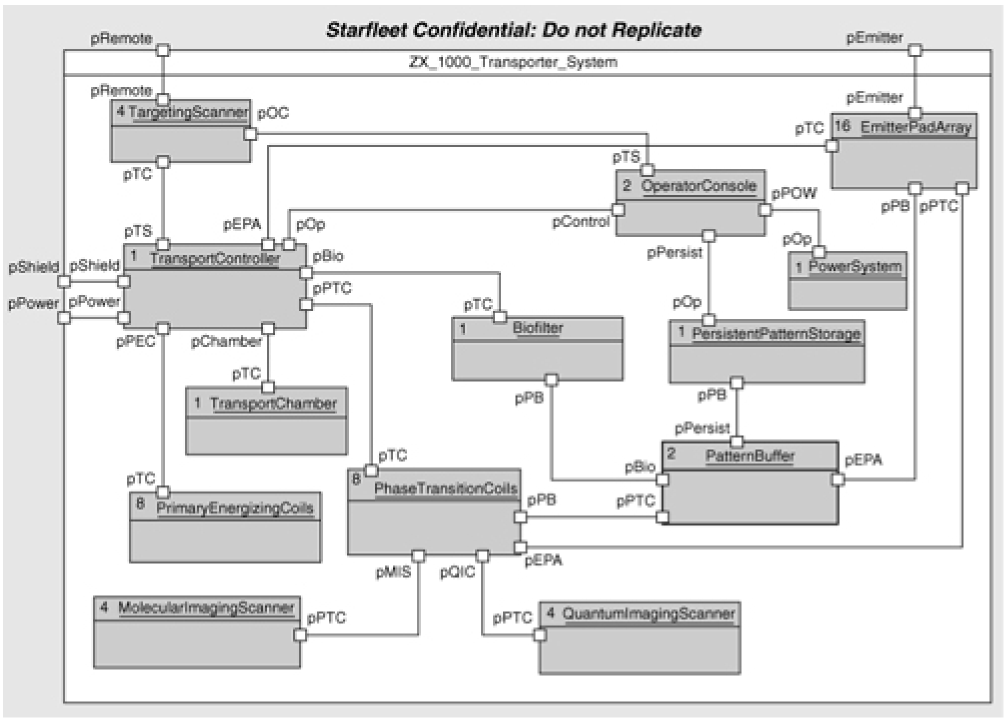
These steps are explained in detail in the following sections.

Understanding the Subsystem Architecture

A subsystem is a large-scale architectural element that allocates requests for services to internal parts for processing. A subsystem contains elements that are cohesive around a common goal and are more tightly coupled internal to the subsystem than to elements in other subsystems. For example, a medical anesthesia device may have subsystems such as User Control Panel, Ventilator, Drug Delivery, Patient Monitor, ECG, and Gas Delivery. Each subsystem may have 3 to 10 engineers working on it. For these engineers to work effectively, they need to understand, clearly and precisely, the responsibilities of their subsystem and the interfaces (both provided and required) among the subsystems.

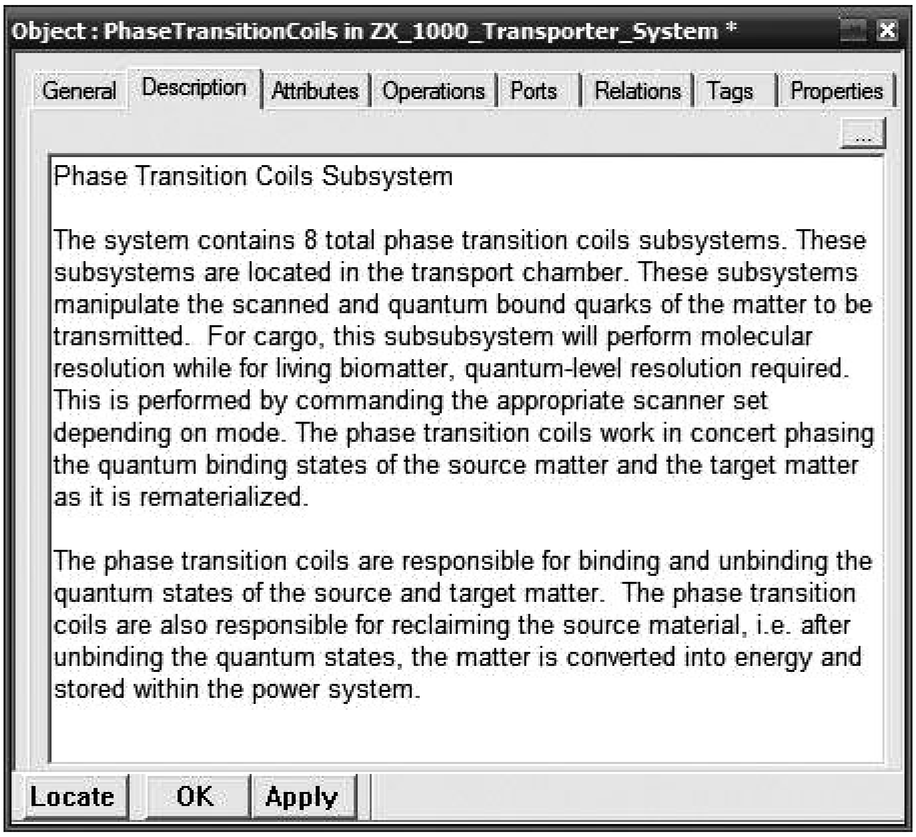
In terms of UML, a subsystem is nothing more than a structured class (class with parts). A subsystem diagram is nothing more than a class or structure diagram that shows the architectural elements. Figure 6.20 shows the subsystem architecture for the transporter system. Each subsystem has a description of responsibilities and a set of connection points (ports) that are defined by the messages that they support.

Figure 6.20 *ZX-1000 Transporter subsystem architecture*



For example, Figure 6.21 describes the responsibilities of the Phase Transition Coils subsystem. Each subsystem will have a similar description.

Figure 6.21 *Sample subsystem description*



Phase Transition Coils Subsystem

The system contains 8 total phase transition coils subsystems. These subsystems are located in the transport chamber. These subsystems manipulate the scanned and quantum bound quarks of the matter to be transmitted. For cargo, this subsubsystem will perform molecular resolution while for living biomatter, quantum-level resolution required. This is performed by commanding the appropriate scanner set depending on mode. The phase transition coils work in concert phasing the quantum binding states of the source matter and the target matter as it is rematerialized.

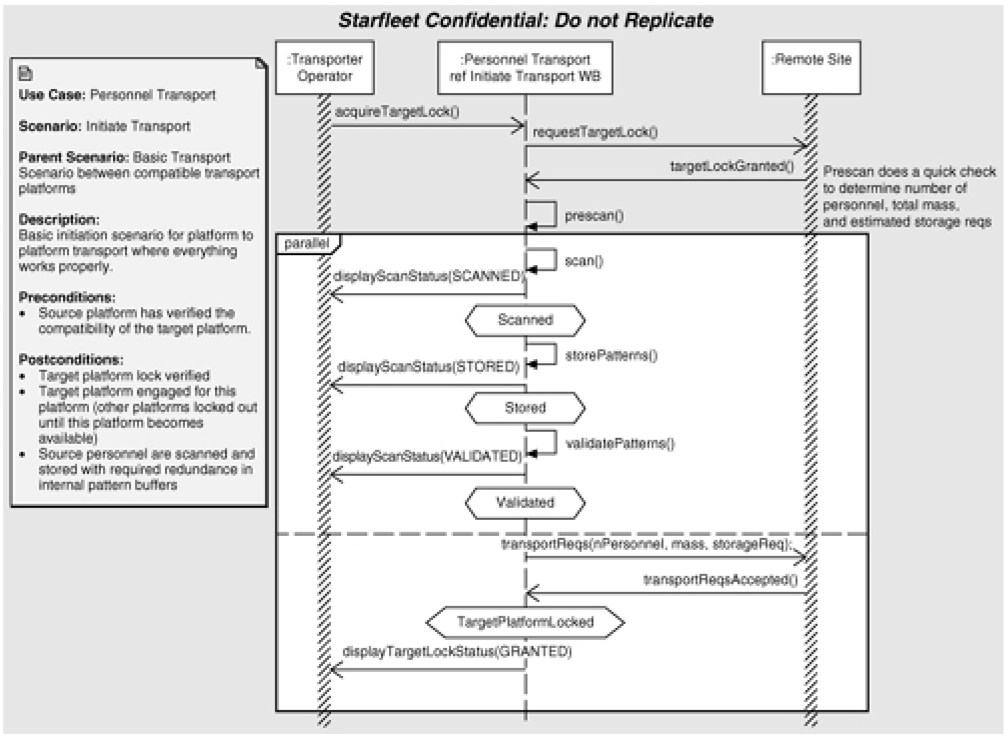
The phase transition coils are responsible for binding and unbinding the quantum states of the source and target matter. The phase transition coils are also responsible for reclaiming the source material, i.e. after unbinding the quantum states, the matter is converted into energy and stored within the power system.

Bottom-Up Allocation

Bottom-up allocation means that the system-level sequence diagrams are decomposed to show the contribution of the different subsystems. The easiest way to do this is to use the "lifeline decomposition" feature of UML for every system-level sequence diagram for the use case under consideration, to show how the subsystems work together to realize that scenario. Consider the scenario shown in Figure 6.5, Initiate Transport. How do the subsystems work together to realize that?

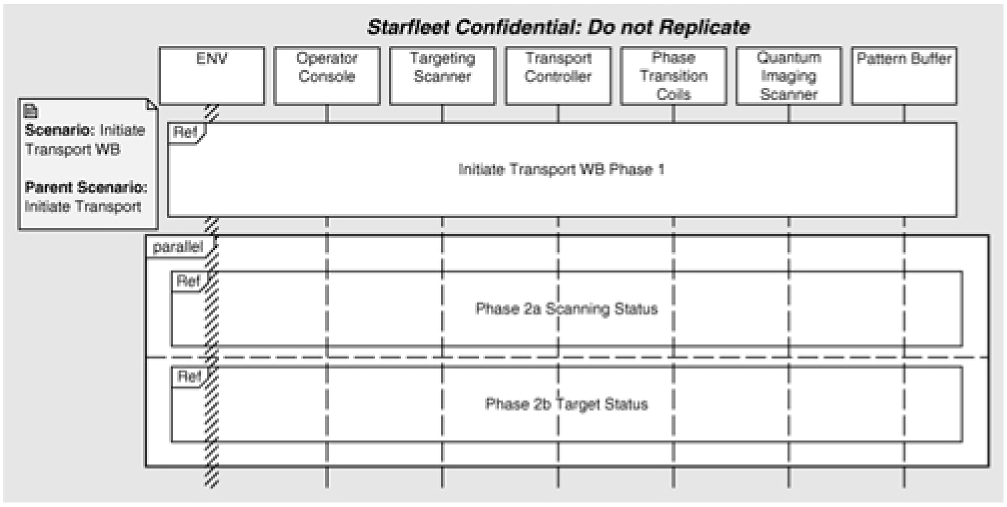
Figure 6.22 shows the same scenario as Figure 6.5, but note that the use case lifeline includes the link "ref Initiate Transport WB." This tells us that the lifeline is decomposed; that is, the very same scenario is shown in a more detailed perspective in another diagram. This diagram shows the internal subsystem interactions necessary to realize this scenario.

Figure 6.22 *White-box scenario ready to be decomposed*


Starfleet Confidential: Do not Replicate
Use Case: Personnel Transport
Scenario: Initiate Transport
Parent Scenario: Basic Transport Scenario between compatible transport platforms
Description: Basic initiation scenario for platform to platform transport where everything works properly.
Preconditions: • Source platform has verified the compatibility of the target platform.
Postconditions: • Target platform lock verified • Target platform engaged for this platform (other platforms locked out until this platform becomes available) • Source personnel are scanned and stored with required redundance in internal pattern buffers
:Transporter Operator
:Personnel Transport ref Initiate Transport WB
:Remote Site
acquireTargetLock()
requestTargetLock()
targetLockGranted()
Prescan does a quick check to determine number of personnel, total mass, and estimated storage reqs
prescan()
parallel
scan()
displayScanStatus(SCANNED)
Scanned
storePatterns()
displayScanStatus(STORED)
Stored
validatePatterns()
displayScanStatus(VALIDATED)
Validated
transportReqs(nPersonnel, mass, storageReq)
transportReqsAccepted()
TargetPlatformLocked
displayTargetLockStatus(GRANTED)

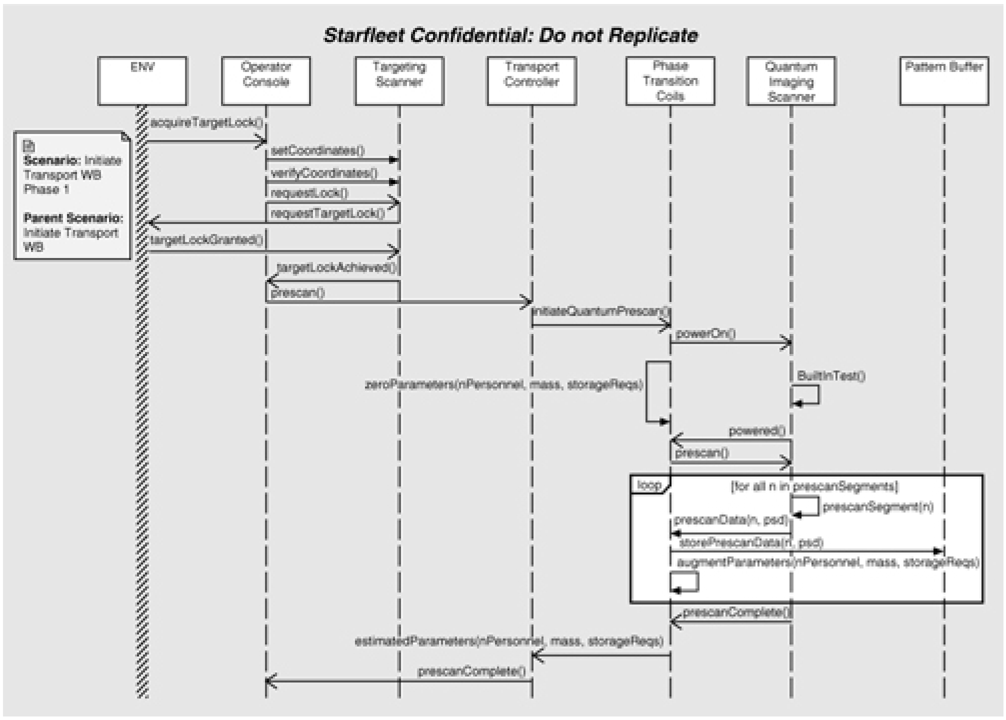
The referenced scenario is shown in Figure 6.23. Note that multiple subsystems from Figure 6.20 show up as lifelines on this diagram. Also note the ENV lifeline at the left—this is the "glue" that binds this scenario to the nondecomposed one; that is, messages going into the use case lifeline in the parent scenario come out of the ENV lifeline in the decomposed one. Similarly, messages coming out of the use case lifeline in the parent scenario go into the ENV lifeline in the decomposed one. In this particular case, no messages are shown because Figure 6.23 is further decomposed into more sequence diagrams.

Figure 6.23 *Initiate Transport white-box scenario*



The first of these—Phase 1—is shown in Figure 6.24. This diagram shows how the subsystems interact up to the point of the parallel interaction fragment operator in the parent. We can see the involvement and responsibilities of the `Operator Console`, `Targeting Scanner`, `Phase Transition Coils`, and so on.

Figure 6.24 *Initiate Transport Phase 1 white-box scenario*



The next two figures (Figure 6.25 and Figure 6.26) show the internal processing of the two parallel regions in Figure 6.23. The point is, though, that the white-box scenarios show the allocation of services and responsibilities to the different subsystems. When this is done for all the scenarios within a use case, the set of services allocated to a subsystem becomes a part of its provided interfaces. The messages that are sent from one subsystem to another become a part of its required interfaces. When a subsystem offers an interface that another requires, the two can be connected across compatible ports.

Figure 6.25 *Initiate Transport Phase 2a white-box scenario*

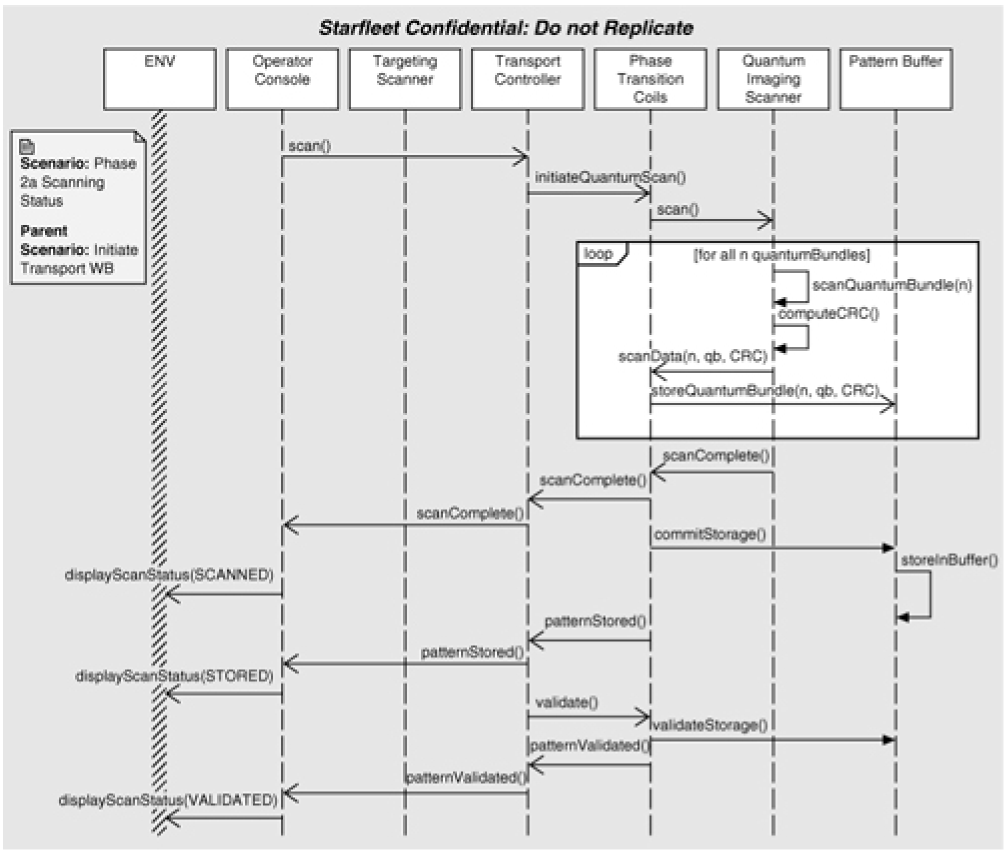
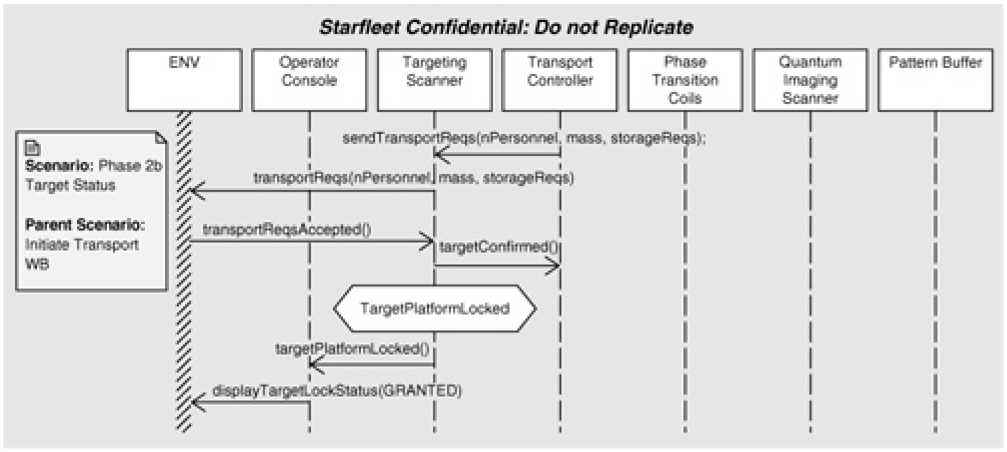


Figure 6.26 *Initiate Transport Phase 2b white-box scenario*


Starfleet Confidential: Do not Replicate
ENV
Operator Console
Targeting Scanner
Transport Controller
Phase Transition Coils
Quantum Imaging Scanner
Pattern Buffer
Scenario: Phase 2b Target Status
Parent Scenario: Initiate Transport WB
sendTransportReqs(nPersonnel, mass, storageReqs);
transportReqs(nPersonnel, mass, storageReqs)
transportReqsAccepted()
targetConfirmed()
TargetPlatformLocked
targetPlatformLocked()
displayTargetLockStatus(GRANTED)

When this strategy is used, the requirements for the subsystems begin to emerge. Once this is done for the set of use cases to be realized within a microcycle, the services allocated to each subsystem can be "clustered up" into use cases, if necessary. This is a powerful way to work, but some people prefer to work "top-down" by decomposing the use cases.

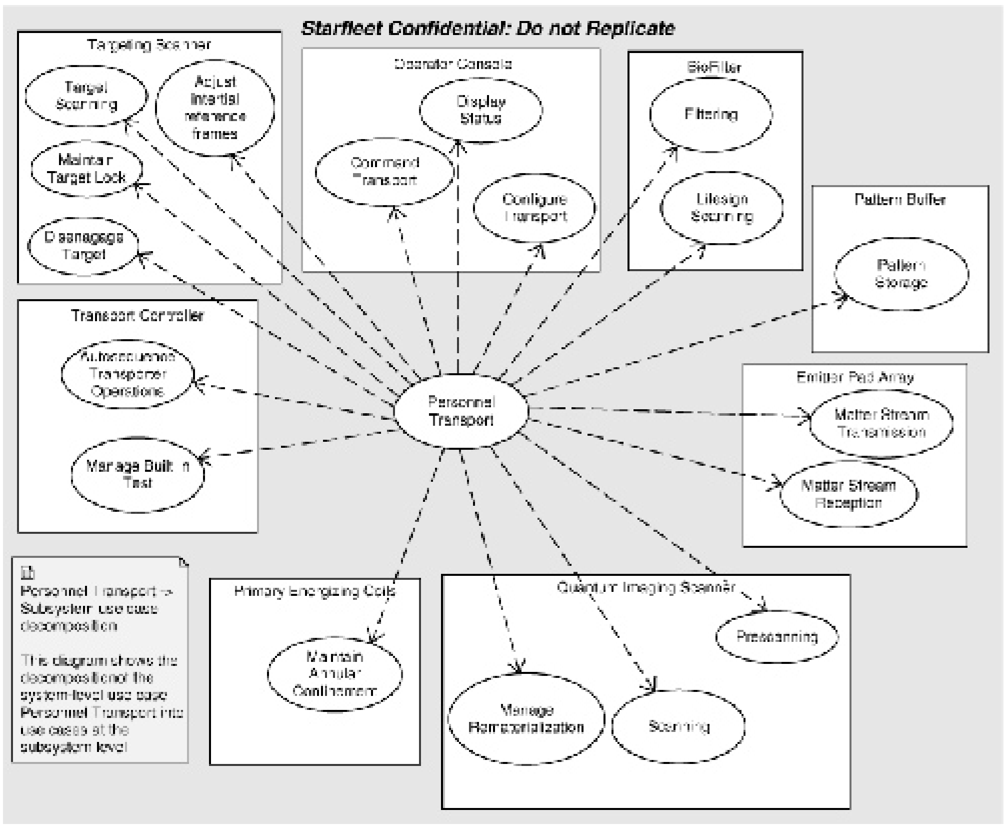
Top-Down Allocation

The top-down allocation strategy decomposes use cases at the system level down to the subsystem level. In UML, this is done with the `«include»` stereotype of dependency. By applying the decomposition to each use case in the microcycle, this step constructs a use case model for each subsystem. Each of these subsystem-level use cases must then be detailed using exactly the same techniques as discussed in Chapter 5 to construct the detailed use case model for the subsystem. This use case model is essentially the subsystem requirements specification.[11]

11. If desired, a textual document can be created in concert with the subsystem use case model, just as the system requirements specification is created in concert with the system use case model. This is discussed in the step "Detail system requirements."

Figure 6.27 shows the approach for one system-level use case of our transporter system: `Personnel Transport`. Each of the included subsystem-level use cases provides part of the necessary functionality. Care must be taken to ensure that each of the subsystem-level use cases meets the criteria for a good use case (multiple scenarios each with multiple messages, multiple pages of requirements bound, etc.). At the subsystem level, these use cases must be detailed; that is, they will be elucidated with scenarios, state machines, and (optionally) activity diagrams. The subsystem teams will use their use case model as the basis for engineering that subsystem.

Figure 6.27 *System use case decomposition into subsystem use cases*



Detailing the Subsystem-Level Use Cases

A subsystem in a large system is a "system" from the perspective of the team building it. Remember, we want a system use case model to understand precisely what is needed at the system level. Therefore, when we have a system complex enough to warrant subsystem decomposition, the subsystem teams need a similar thing, only focused on their scope of interest; that is, the subsystem teams need a detailed use case model that provides a detailed specification for their subsystems.

Each of the use cases identified in the previous steps must be detailed—if the functionality of the subsystem is complex enough to have use cases, those use cases warrant a full definition. This topic was discussed earlier in this chapter and also in Chapter 5. Each use case should have a set of sequence diagrams and at least a state machine specification that includes error and fault detection and handling. The steps to achieve it are exactly the same as for the system use case.

Validating the Subsystem-Level Use Cases

Validation of the subsystem-level use cases can be as simple as walking through the scenarios with the relevant stakeholders (e.g., SMEs and the architect) to make sure that they make sense. Or it can be as complex as constructing an executable use case model to demonstrate that the subsystem use case model can replicate its contribution to the system-level use cases. The former approach is appropriate for a low-risk project, and the latter is more appropriate either for projects that are high-risk or when the cost of a requirements error or misunderstanding is high. It is common to construct executable requirements models for transportation, military, medical, and aerospace systems.

6.1.7. Use Case Consistency Analysis

Ideally, use cases are *orthogonal;* that is, there is no opportunity for them to be inconsistent because they are loosely coupled with others. Sometimes, however, the requirements are not linearly separable, and this causes coupling between the use cases. When use cases are tightly coupled, there is an opportunity for the requirements in different use cases to conflict. This step is intended to address those situations.

A less robust way to do this (which may be fine in some circumstances) is by review. The SMEs and the architect can review the details of the tightly coupled use cases to look for conflict. A more robust way is to construct executable models for the use cases and *execute* them together. If they are in conflict, then one or more expectations will not be met by the combined output of the use case executions.

6.1.8. Detailing System Requirements

System requirements form an important document that adds precision to the stakeholder requirements. Indeed, if the stakeholders are technically proficient in their problem area, the system requirements *are* the stakeholder requirements. But for the most part, the stakeholder requirements will include requirements such as “Blood oxygen concentration will be monitored continuously with a high degree of accuracy,” while the equivalent system requirement might be “Blood oxygen concentration will be measured at least once per second with an accuracy of ±2%.”

Other than that, the procedure for creating the system requirements specification is very much the same as for the stakeholder requirements specification discussed in the previous chapter. This document is best stored within a requirements management tool such as DOORS because such tools provide traceability to the product vision and stakeholder requirements as well as to the downstream engineering solution, such as the classes, objects, functions, and data in the object analysis and design models.

Care must be taken to ensure that design does not creep into the system requirements. These requirements should still be "black-box"; they are just at a higher level of precision than typical stakeholder requirements. The most common failure of system requirements is specifying *how* the design should achieve the requirements. That is clearly a task of downstream engineering and not within the purview of the system requirements.

The system requirements are developed in parallel with the other tasks within the prototype definition activity. This means that they are developed side by side with the use case model, the hazard analysis, and the user interface requirements. By the end of the prototype definition activity, the requirements are sufficiently well formed that it is clear what needs to be engineered during the current microcycle.

Note

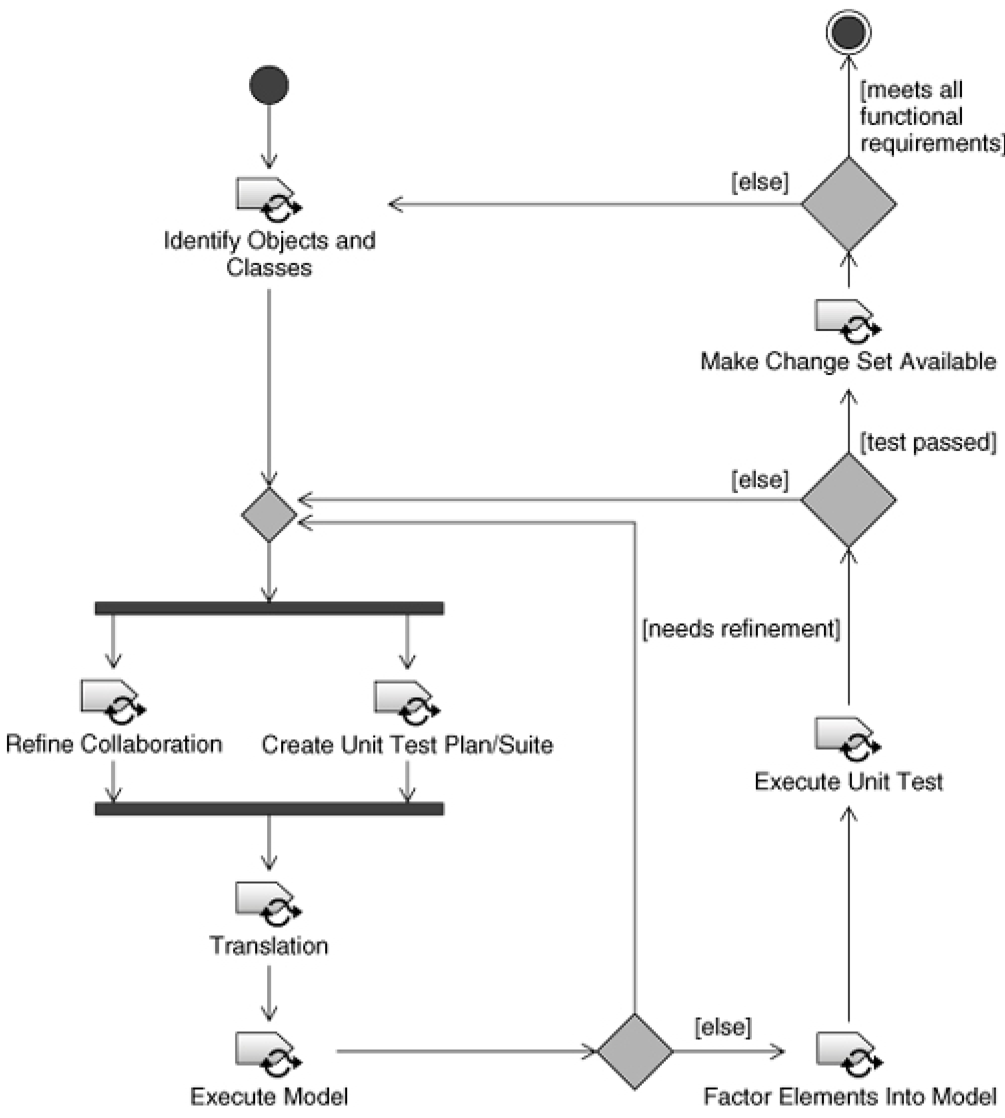
The Harmony/ESW process recommends that the system requirements be done as an incremental activity; that is, each microcycle elaborates the set of system requirements to be realized within that microcycle, as scoped by the use cases contained within it. Use cases that are allocated to later microcycles are deferred until those microcycles are reached.

6.2. Object Analysis

Object analysis follows prototype definition. The purpose of prototype definition is to specify the requirements, whereas the purpose of object analysis is to produce an **analysis model** that realizes all of the functional requirements specified for the use cases allocated to the current microcycle.

Figure 6.28 shows the workflow for the object analysis activity. The most salient points of the workflow are the three loops that appear from the conditional connectors. Collectively, these inner loops constitute the *nanocycle* within object analysis. Object analysis most assuredly does *not* proceed by creating dozens or hundreds of classes with the intent of getting them to work later. It proceeds by making small incremental changes to the model (tasks "Identify objects and classes" and "Refine collaboration"), generating code (task "Translation"), debugging (task "Execute model"), formally unit testing (tasks "Create unit test plan/suite" and "Execute unit test"), and submitting, at least once daily, the tested software to the configuration manager for inclusion in the evolving baseline (task "Make change set available"); that is, you identify a class or even a feature of a class (such as an attribute, state, event, or variable), construct some tests, add the changes into the collaboration, execute it, test it, and if it works, make some more changes. You'll submit this unit-tested software (model and code) to the configuration manager frequently to include in the baseline.

Figure 6.28 *Object analysis workflow*


[meets all functional requirements]
[else]
Identify Objects and Classes
Make Change Set Available
[test passed]
[else]
[needs refinement]
Refine Collaboration
Create Unit Test Plan/Suite
Execute Unit Test
Translation
[else]
Execute Model
Factor Elements Into Model

I emphasize the workflow because that is not what most people have done historically when given an opportunity to model. What usually happens is that developers spend weeks or months modeling before beginning to try to get the model converted into code or running. That approach has proven itself to be expensive and error-prone. Remember our *Agilista* credo: The best way not to have defects in your software is not to put them in there in the first place. This means constant, or at least highly frequent, execution, at least 10 times per day and often two or three times that often. Furthermore, unit testing isn't postponed until all the code is written. Unit testing is also done incrementally. If I have a couple of methods and an attribute to a class, *at the same time,* I add one or more tests to demonstrate that the model is right so far. These tests are applied as the model is executed. They ensure that the model works now and are reapplied later to make sure that later changes haven't broken the functionality.

The other thing to note about the object analysis model is that it meets the functional requirements but usually not the QoS requirements. The object analysis activity is all about defining the essential properties of the class model of the solution but is not focused on achieving optimality. Optimization is a job for design, the subject of the next chapter. In this chapter, I will focus on how to create the object analysis model.

> Tip
>
> Far too many people focus on creating optimal code before the code even works, resulting in very fast and only mostly right software. It is better to get the software correct, *demonstrably correct*, and then optimize it.

Harmony/ESW defines a basic five-step workflow for developing software, detailed in Figure 6.28.

1. Make a small incremental change to the model.

2. Generate unit tests to include your change.

3. Generate source code.

4. Verify, through execution and unit testing, that the model and code are right so far.

5. Repeat until done.

6.2.1. Identifying Objects and Classes

The object analysis model (aka PIM) is an executable model of class specifications connected with various relations. The first challenge in constructing the object analysis model is to identify objects and classes, their behavior (e.g., operations and state machine actions), and their data (attributes). The Harmony/ESW process provides a set of strategies for this purpose. Each of these strategies finds a subset of the elements. Each strategy finds some elements that are found by other strategies, but not all elements are found by all strategies. It is usually enough to employ two to four strategies to find all of the relevant elements in the analysis model. Which you use is a matter of both personal preference and the problem to be solved;

that is, some strategies work better in some problem domains than in others. It is also important that you remember to avoid thinking too much about design. The point here is to create what is known as a **domain model**—a model of the essential concepts and their essential behavior and data.

These "object identification strategies"[12] include:

• Underline nouns and noun phrases.

• Identify causal agents.

• Identify services to be performed.

• Identify events to be received.

• Identify information to be created or manipulated.

• Identify transactions.

• Identify real-world things to be represented.

• Identify resources to be managed.

• Identify physical devices.

• Identify key concepts and abstractions.

• Identify persistent data.

• Execute scenarios.

12. To see these strategies used "in action," see my *Real-Time UML Workshop for Embedded Systems*.

Let's consider each of these briefly.

Underlining Nouns and Noun Phrases

If you have a textual specification or concept document, underlining the nouns and noun phrases can lead to the identification of:

• Objects

• Classes

• Attributes

• Actors

but it can also identify:

• Synonyms of elements already identified

• Uninteresting objects—elements for which you will not write software

This is a common approach but is problematic in an incremental, spiral process. Problems arise because the increments implement only a subset of the entire system. Unless the textual specification is organized around the use cases, it is difficult to focus only on the nouns or noun phrases relevant to the specific use case under consideration. Nevertheless, *if* you have a well-written textual specification *and* it is organized around the use cases, then the strategy can be effective.

Identifying Causal Agents

Causal agents are things that cause results to occur; for example, a button on an elevator system "causes" the elevator to be selected and dispatched to the floor; the arrival of the elevator at the floor "causes" the elevator door and floor door to interlock; the resulting interlock "causes" the doors to cycle open, and so on. Identification of causal streams can lead to the objects and classes that provide the causes and others that realize the effects.

Identifying Services to Be Performed

Systems provide various services, some visible to an actor, some not. Every executing service is owned by a single instance that is typed by a single class; some of these services result in the sequenced execution of other services in other instances, forming a "call tree." In any event, ultimately, the services will be allocated to a single instance or be decomposed into services that will be. Following the service execution chain can identify the objects and classes that "own" and provide those services. This strategy can also lead to the identification of attributes with the question "What information does this service need or produce?"

Identifying Events to Be Received

Events are related to services in that they can invoke the execution of services. The “Identify events to be received” strategy is very similar to “Identify services to be performed” but may be more obvious in reactive systems, that is, systems that receive and react to events of interest. Every event is generated by a class and is received and processed.

Identifying Information to Be Created or Manipulated

Data must be stored as attributes of classes, so the identification of data leads one to identify the class that creates, owns, manages, or destroys that data. This strategy can also identify services provided by the owner classes with the question “How is this information used?”

Identifying Transactions

A transaction is the reification of an interaction between objects as an object itself. Transactions are used when an interaction is complex or stateful, or when it persists for a period of time and must be managed. Transactions are used in reliable message delivery, connection-oriented messaging, and staged interactions, or whenever an interaction between objects must be remembered (such as in a deposit into a banking account).

Identifying Real-World Things to Be Represented

Systems must often represent real-world elements as interfaces, data representations, simulations, or resources. Physical devices (important enough to be a strategy on their own) are most often represented as interfaces to electronics but may be simulated in some cases. Data representations stand in for some specific aspects of these real-world things; for example, a banking customer class might hold the customer’s name, bank ID, tax billing number, phone number, and address. Resources are represented as classes with a finite capacity of some kind, such as a list of available parts or a fuel tank with a specific quantity being held within its capacity. These are represented within the system as objects, typed by classes.

Identifying Resources to Be Managed

Resources are objects that have finite capacity. They can represent physical elements (such as memory available, fuel in a fuel tank, or missiles in a missile launcher) or abstractions (such as the number of available connections or the number of waiting messages), but they are represented as objects in the system.

Identifying Physical Devices

Physical devices are a special kind of real-world thing. Most common by far, physical devices are represented by interfaces or device drivers; that is, a single object "knows" how to actually talk to the physical sensor or actuator, and any object in the system that must access the physical device does so through this interface object. Sometimes, however, the physics (or other characteristics) of a physical device must be simulated, such as when the device is not yet available for integration, or when you're modeling physical properties, such as a bullet's trajectory in a video game or the response of an airframe to adjustment of flight control surfaces (the "kinematic model").

Identifying Key Concepts and Abstractions

Key abstractions are the antithesis of physical devices; these are things that are conceptual, but nevertheless essential in the domain of discourse. For example, a *track* in a tracking application, an *account* in a banking application, a *window* in a GUI application, a *thread* in an operating system, a *function* in a compiler, and so on are all conceptual entities in their respective domains. Every application domain has an interrelated set of concepts that must be represented as objects and classes.

Identifying Persistent Data

The persistent data strategy looks for data that is important to remember for long periods of time, possibly between hard restarts of the system. Such data may be user names and passwords, configuration data, maps and geolocation data, and so forth. This data must be represented as objects and classes within the model.

Executing Scenarios

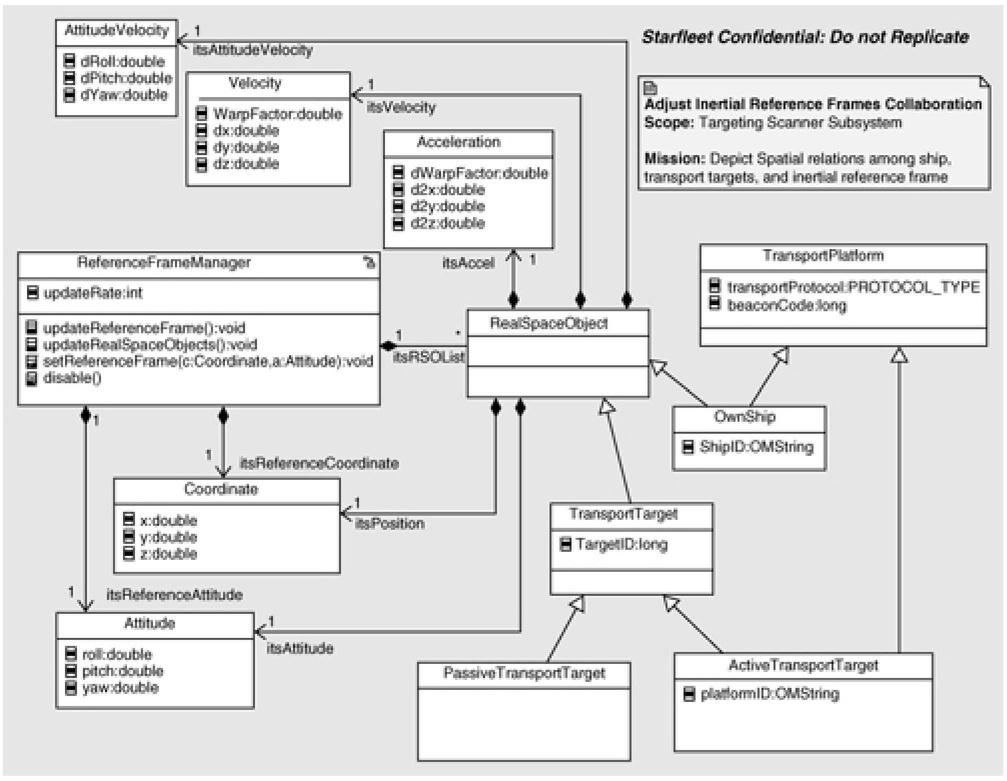
The "Execute scenarios" strategy is a personal favorite; it seeks to identify objects by walking through or executing use case scenarios and adding objects (as lifelines) to the scenario to "flesh it out." The messages to the lifelines become services provided by those objects, invoked either by calls (synchronous rendezvous) or event receptions (asynchronous rendezvous). Lifelines (objects) are added as you walk through the scenario. When a message is sent between objects, an association is drawn between them on a class diagram. Scenarios may be walked through from the beginning or at any "interesting" part. This is an invaluable strategy for identifying objects, services, and relations.

6.2.2. Example from the Targeting Scanner Subsystem

Using several of the strategies, the following analysis model of the Targeting Scanner subsystem was created. The classes, state machines, and activity diagrams that define the structure and behavior of the subsystem are shown in the following figures. Instead of one massive and unreadable diagram, the diagrams are organized by use case collaboration. The use cases for the Targeting Scanner subsystem are shown in Figure 6.27.

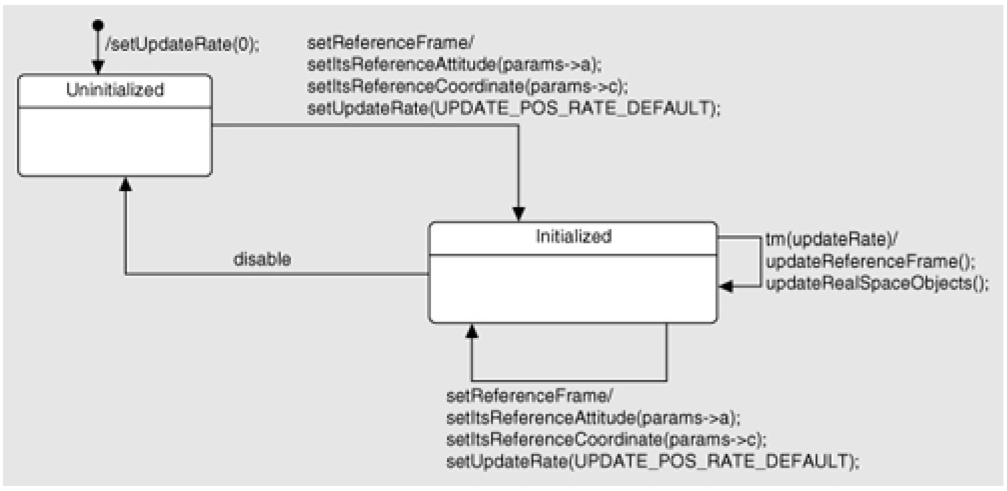
The first of the class diagrams is Figure 6.29. The need for this diagram and its elements was discovered by looking at the first few messages in Figure 6.24. Clearly, we need to have coordinates for at least three things—the transporter platform for the ship, the target destination, and the "zero" or reference frame—and possibly for other targets we may want to save and use. The reference frame is normally set to the center of the primary star (in case there is more than one) with an arbitrary "north" setting orthogonal to it. Along with the position, we need to know the orientation (attitude) in space for at least these "real space objects" as well as their positional and attitudinal velocity and positional acceleration. We must also support different transport protocols (the Klingons and the Vulcans can never agree on anything). We must support both passive and active target destinations, and the algorithms are somewhat different. This collaboration is shown in Figure 6.29.

Figure 6.29 *Adjust Inertial Reference Frames collaboration*



The problem with real space objects is that they're always moving and rotating with respect to all the other objects in space. For this reason, we must update the relative positions of all objects on a frequent basis; this is done in the state machine for the `ReferenceFrameManager`, shown in Figure 6.30. The `tm()` event[13] periodically occurs, invoking the update services.

Figure 6.30 *State machine for* `ReferenceFrameManager`

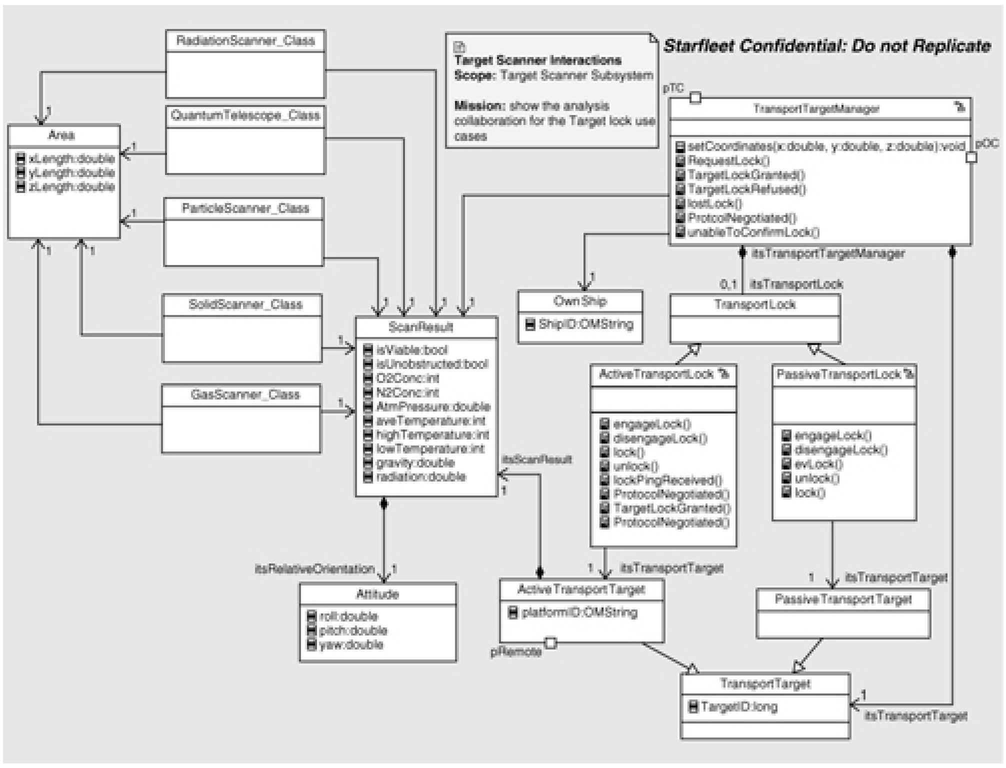


/setUpdateRate(0);
Uninitialized
setReferenceFrame/
setItsReferenceAttitude(params->a);
setItsReferenceCoordinate(params->c);
setUpdateRate(UPDATE_POS_RATE_DEFAULT);
Initialized
tm(updateRate)/
updateReferenceFrame();
updateRealSpaceObjects();
disable
setReferenceFrame/
setItsReferenceAttitude(params->a);
setItsReferenceCoordinate(params->c);
setUpdateRate(UPDATE_POS_RATE_DEFAULT);

13. The `tm(recurrenceTime)` fires `recurrenceTime` time units after entering the state. In this case, the transition leads back to the same state, so this transition fires periodically at that rate.

The other use cases for the Targeting Scanner subsystem deal with scanning and managing a target lock. The class diagram for these collaborating elements is shown in Figure 6.31. The key elements are the various targeting scanners, `TransportTargetManager`, and the two kinds of transport locks. The targeting scanner classes can be identified by the physical device strategy (among others), the `TransportTargetManager` by the service or causal agent strategies, and the transport locks by the transaction strategy.

Figure 6.31 *Target lock collaboration*



The purpose of the remote scanners is to determine the physical conditions of the target location. To this end, a number of different scanners are required, for scanning the area for electromagnetic radiation, particle density, gravity, obstructions (solids), and pressure gas concentrations (for crew survivability concerns). The scanner classes all contribute to the scan results and interact with the `TransportTargetManager` (see Figure 6.32). The scanning algorithm is shown in the activity diagram (really just a flowchart in this case) in Figure 6.33.

Figure 6.32 *Targeting Scanner interactions*

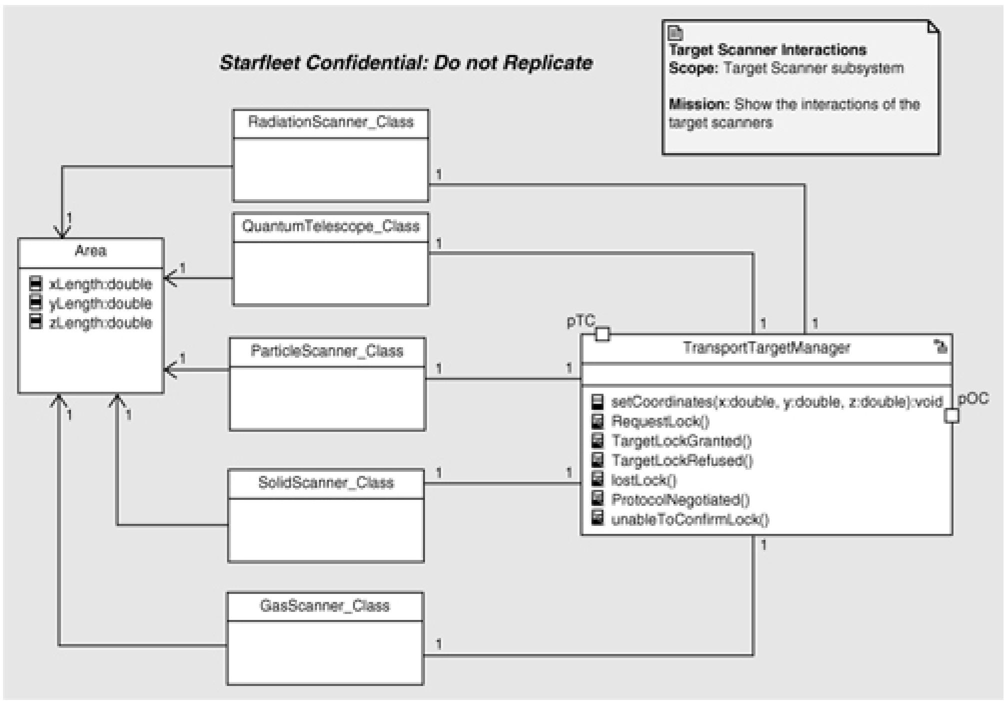
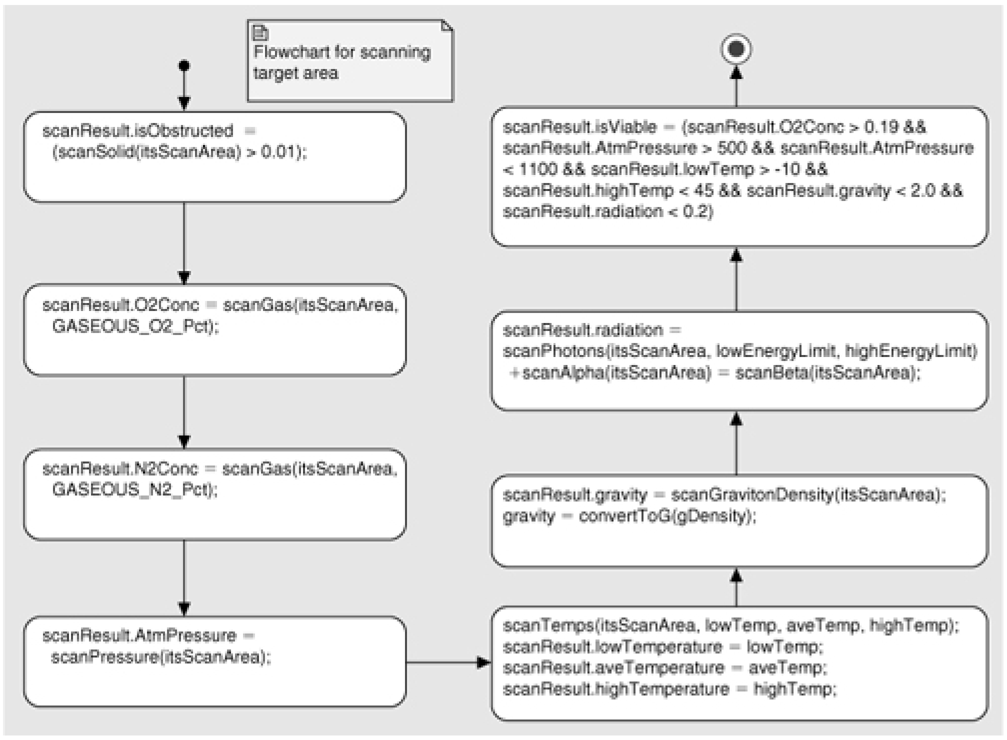


Figure 6.33 *Activity diagram for scanning*



```
scanResult.isObstructed =
  (scanSolid(itsScanArea) > 0.01);

scanResult.O2Conc = scanGas(itsScanArea,
  GASEOUS_O2_Pct);

scanResult.N2Conc = scanGas(itsScanArea,
  GASEOUS_N2_Pct);

scanResult.AtmPressure =
  scanPressure(itsScanArea);

scanTemps(itsScanArea, lowTemp, aveTemp, highTemp);
scanResult.lowTemperature = lowTemp;
scanResult.aveTemperature = aveTemp;
scanResult.highTemperature = highTemp;

scanResult.gravity = scanGravitonDensity(itsScanArea);
gravity = convertToG(gDensity);

scanResult.radiation =
scanPhotons(itsScanArea, lowEnergyLimit, highEnergyLimit)
 +scanAlpha(itsScanArea) = scanBeta(itsScanArea);

scanResult.isViable = (scanResult.O2Conc > 0.19 &&
scanResult.AtmPressure > 500 && scanResult.AtmPressure
< 1100 && scanResult.lowTemp > -10 &&
scanResult.highTemp < 45 && scanResult.gravity < 2.0 &&
scanResult.radiation < 0.2)
```

The `TransportTargetManager` has the responsibility for acquiring and maintaining a target lock. To this end, it owns a `TargetLock`, which is either passive (no transporter platform at the target) or active (the target is another transporter platform). Figure 6.34 shows the state machine for the `TransportTargetManager` class.

Figure 6.34 `TransportTargetManager` *state machine*

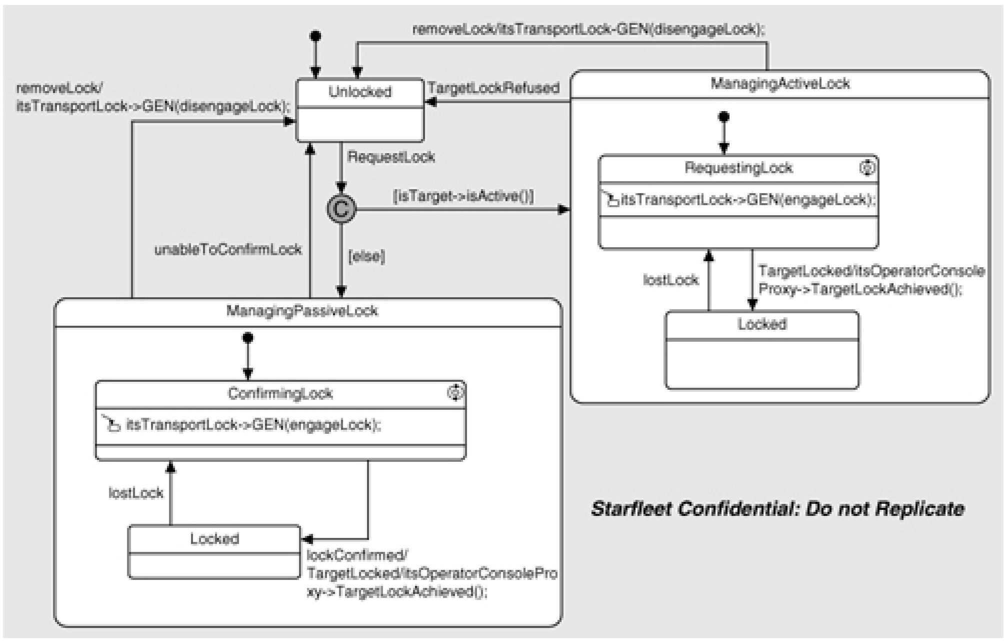
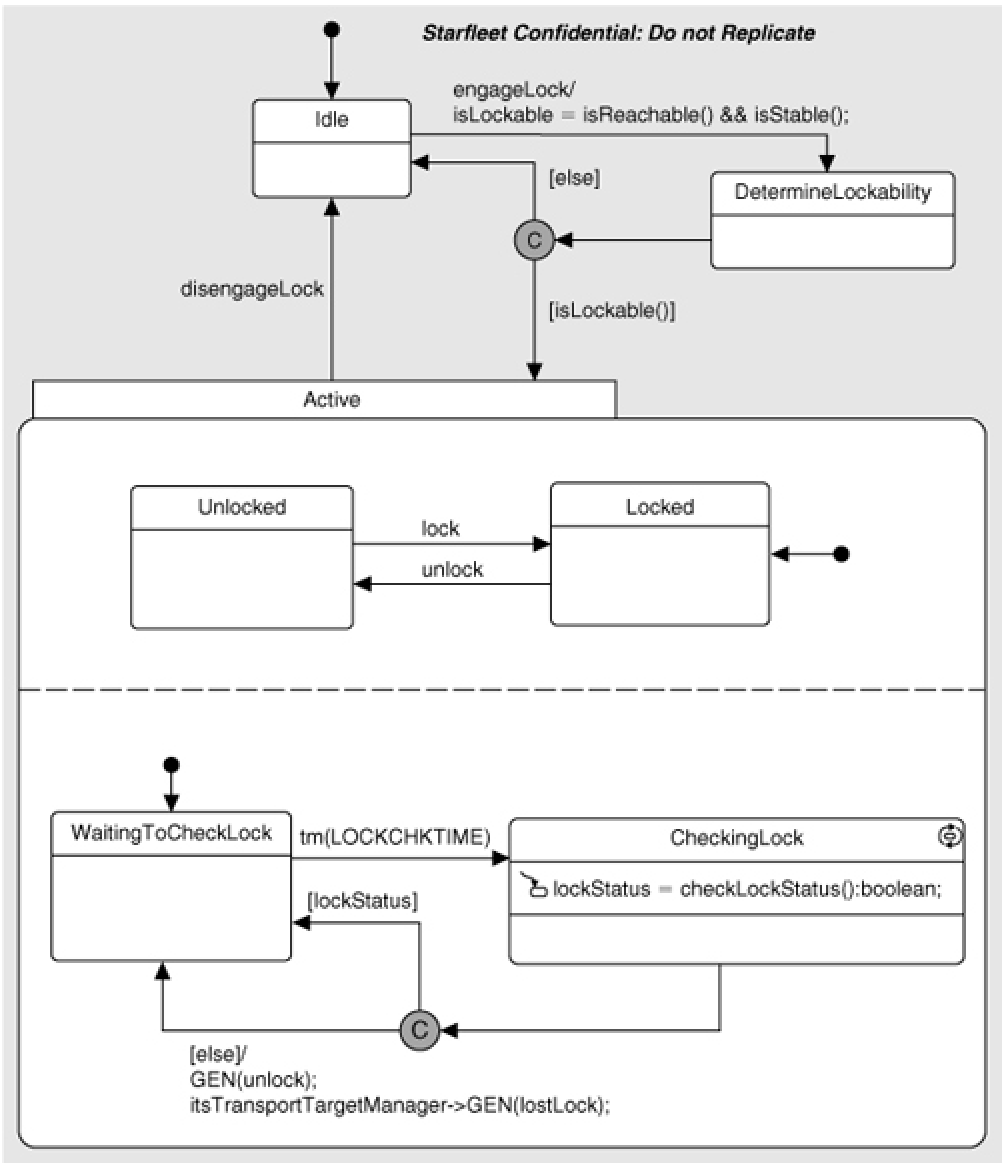

removeLock/itsTransportLock-GEN(disengageLock);
removeLock/
itsTransportLock->GEN(disengageLock);
Unlocked
TargetLockRefused
ManagingActiveLock
RequestLock
[isTarget->isActive()]
RequestingLock
itsTransportLock->GEN(engageLock);
unableToConfirmLock
[else]
lostLock
TargetLocked/itsOperatorConsole
Proxy->TargetLockAchieved();
Locked
ManagingPassiveLock
ConfirmingLock
itsTransportLock->GEN(engageLock);
lostLock
Starfleet Confidential: Do not Replicate
Locked
lockConfirmed/
TargetLocked/itsOperatorConsolePro
xy->TargetLockAchieved();

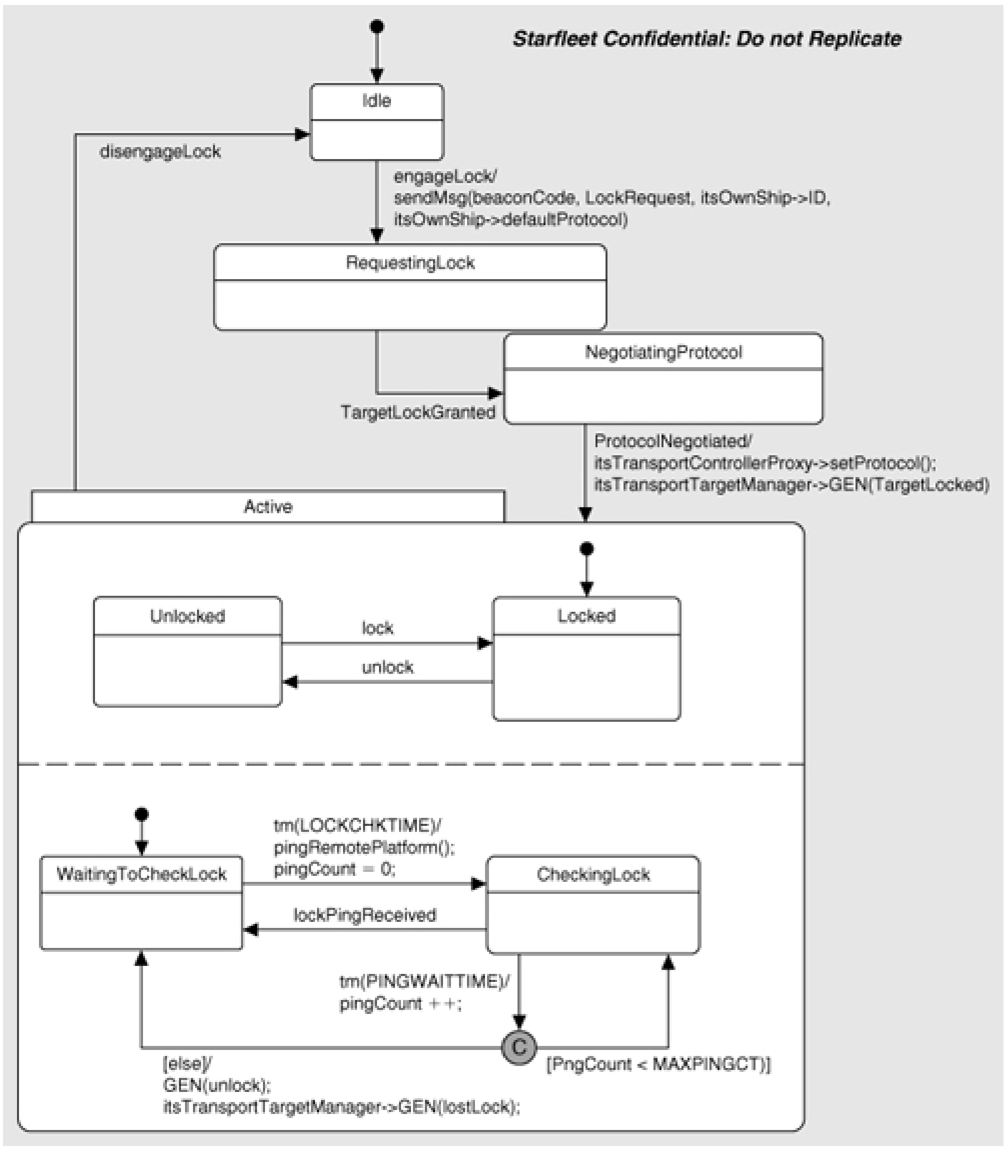
Figure 6.35 shows the state machine for the `PassiveTransportLock` class. Because it is passive, we need to determine whether or not it is lockable; that is, it must be within range and must be stable (low relative velocity, attitude change rate, and acceleration). To maintain the lock, the lower and-state periodically checks that we can still track the target location. If not, then the lock is lost, the target becomes unlocked, and the `TransportTargetManager` must deal with it.

Figure 6.35 `PassiveTargetLock` *state machine*



The `ActiveTargetLock` state machine is shown in Figure 6.36. It has a similar structure to Figure 6.35 but because the target is a transporter platform, it must agree to the lock and to the negotiated transport protocol. The lower and-state maintains the lock, again by periodically checking if the lock is valid. This is done by sending a "ping" to the target platform and receiving an acknowledgment. If a ping is not received within a specified time frame (`PINGWAITTIME` nanoseconds), the ping is resent. This continues until either the acknowledgment is received or the retry count is exceeded. If the retry count is exceeded, this constitutes a loss of target lock.

Figure 6.36 `ActiveTargetLock` *state machine*



You can see that the elements in these diagrams are "essential" in the sense that the nature of the problem dictates their necessity. We haven't specified the communication protocol, concurrency threads, distribution across multiple processors, redundancy for safety and reliability, or any of a number of other optimizations. That will be done in design. In the analysis model, we focus on the essential elements and their behavior.

6.2.3. Refining the Collaboration

Once one or a small number of elements have been identified, they can be entered into the evolving collaboration of classes and other elements. This task is relatively simple: Insert the identified element(s) into the collaboration, add or modify relations so that it can interact with other elements, and refine it and the collaborating elements so that they work together to achieve their composite functional goals. The refinement will include adding or modifying relations (e.g., association or generalization), attributes, operations, event receptions, states, state action lists, and control flow within operations.

One aspect of refinement is known as “defensive development.” The key concept is to create “bulletproof”[14] software. This is done by explicitly stating the pre- and postconditions of all services and adding behavior that either verifies that the preconditions are met or handles the violation appropriately when they are not. If an operation for a class sets the patient weight (in kilograms) for closed-loop control of drug delivery, then this operation should ensure that the value is reasonable. For example, 100 kg is perfectly reasonable, but what about 1000 kg? How about –1 kg? What should be done if the precondition isn’t met? Alert the user? Measure the data again? Restart the system? Defensive development forces you to think about what the preconditions are and what the software needs to do if they are violated. The unit tests (discussed next) should exercise the collaboration with both normal and invalid data.

14. Perhaps “phaser-proof” might be more appropriate in this context.

6.2.4. Creating the Unit Test/Suite

Unit testing consists of applying inherently white-box test cases to your collaboration. In the Harmony/ESW process, this is *not* something done at the end of development but is rather applied constantly as the software is incrementally evolved. Initially, a single class will be tested, and so the scope of the test cases is very small. Over time, as the software scope grows, so does the scope of the unit tests.

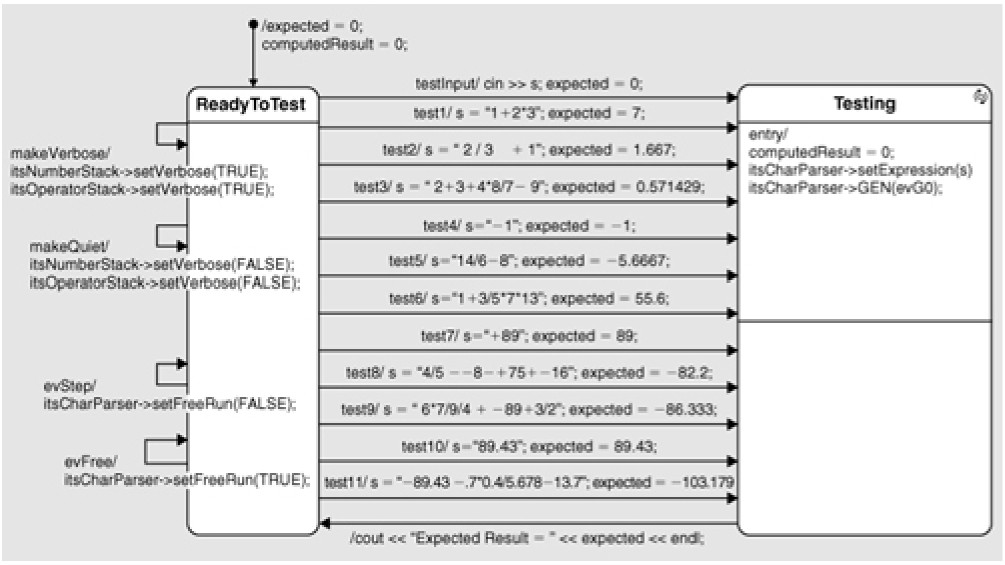
A unit test plan specifies the set of test cases. The advantage of creating this specification is that it is easy to see if different kinds of tests are missing. Some common test case types relevant to unit testing are listed in Chapter 3 in the section “Create Software and Tests at the Same Time.”

The tests themselves can be done by creating «`testBuddy`» classes that exist to test the deliverable parts of the model or by using tools to automatically create them. The test buddies are created by the developer and configuration-managed with the other model elements. These classes exist only to verify analysis or design classes, and they are not shipped in the delivered product. Tools such as Rhapsody Test Conductor can use sequence or activity diagrams to specify test cases and automatically execute them. Both of these approaches are better than manual testing because the tests can be configuration-managed with the model and provide an evolving test platform for your software.

Figure 6.37 shows an example state machine of a «`testBuddy`» class for a calculator application. Each different test is executed by sending the appropriate event to the testing buddy, and it sets up the execution environment and then performs the test. The transition back to the

`ReadyToTest` state prints out both the computed and the expected test results. This makes it easy to reapply the tests as more functionality is added to the collaboration.

Figure 6.37 «`testBuddy`» *class state machine*


/expected = 0;
computedResult = 0;
ReadyToTest
Testing
entry/
computedResult = 0;
itsCharParser->setExpression(s)
itsCharParser->GEN(evG0);
testInput/ cin >> s; expected = 0;
test1/ s = "1+2*3"; expected = 7;
test2/ s = " 2 / 3   + 1"; expected = 1.667;
test3/ s = " 2+3+4*8/7− 9"; expected = 0.571429;
test4/ s="−1"; expected = −1;
test5/ s="14/6−8"; expected = −5.6667;
test6/ s="1+3/5*7*13"; expected = 55.6;
test7/ s="+89"; expected = 89;
test8/ s = "4/5 −−8−+75+−16"; expected = −82.2;
test9/ s = " 6*7/9/4 + −89+3/2"; expected = −86.333;
test10/ s="89.43"; expected = 89.43;
test11/ s = "−89.43 −.7*0.4/5.678−13.7"; expected = −103.179
/cout << "Expected Result = " << expected << endl;
makeVerbose/
itsNumberStack->setVerbose(TRUE);
itsOperatorStack->setVerbose(TRUE);
makeQuiet/
itsNumberStack->setVerbose(FALSE);
itsOperatorStack->setVerbose(FALSE);
evStep/
itsCharParser->setFreeRun(FALSE);
evFree/
itsCharParser->setFreeRun(TRUE);

6.2.5. Translating the Model into Code

The translation task produces source-level code that corresponds to the model. This means that you must constantly do a little bit of modeling, a little bit of test case generation, a little bit of coding, and a little bit of testing. Don't wait weeks or months before you get around to generating and executing the code for the model. Evolve the code in parallel with the model. Of course, this is easiest in a tool that generates all the code for you, but even if you don't have such a tool, this is a crucial part of the process.

Note

I use the Rhapsody tool from IBM Rational to generate the full executable code from the model. Other tools can be used, although many produce only skeletons of the code structures into which you must manually type the source code. However, even if you have to write all the code by hand, this is a crucial step in the nanocycle; you should never be more than minutes away from executing the model in its current state.

This task also includes compilation and linking of the source code into an executable application that can be debugged and tested. The compile and link steps will include any legacy or third-party source code and components required for execution. It isn't necessary to compile and link the entire system, just enough to achieve the purpose of the execution. In this case, the purpose is to demonstrate the functional correctness of the model at its current level of completion.

The transporter system is implemented in Object Var'aq (OV), the object-oriented Klingon programming language.[15] The OV language is owned by the Object Management Federation Group (OMFG), and the details of the syntax are still being specified as of this writing. (I'm still working on the OV model compiler, but I hope it will be available before the ship sails . . . ☺)

15. It looks a lot like C++ except that most of the warning messages begin with the phrase "This pointer has no honor!" Fatal error messages state, "Today is a good day for this program to die!" In addition, Object Var'aq data casts have different tenses, including Command (downcast), Imperative (upcast), and Ultimate Imperative ("Die, Romulan scum!" cast) forms. And, of course, the OV-compatible keyboard also doubles as Bat'leths (see http://www.klingonimperialweaponsguild.org/), which is why Klingon programmers never get their performance reviews when they are at their desks.

6.2.6. Executing the Model

Once the code is generated, compiled, and linked, it must be run. This execution is focused but informal, a process known as "debugging." During debugging the execution of the model is examined and compared against expectations. If the elements seem correct, then the elements are factored into the full model in the next task.

Since the execution is focused, the purpose of the execution must be identified so that you can set up the appropriate execution environment (e.g., with external hardware and/or software) to achieve this purpose. Then you can informally explore the execution by assigning values to attributes, inserting events, calling operations, and so forth and comparing the results with your expectations.

6.2.7. Factoring the Elements into the Model

Often, modelers work in a temporary work area while the elements being added are unstable. Once the elements are stabilized, they can be added to their rightful locations within the model(s) to allow them to work with the full collaboration(s). Formal unit testing is best performed once the model elements are where they ultimately belong.

6.2.8. Executing Unit Tests

This step executes the unit tests specified earlier. Some of these unit tests may be from long ago (days, weeks, or even longer) and others may be freshly added. It is important to ascertain that the new elements work properly, but also that the changes don't break existing, working functionality.

If unit tests fail, then the elements will need to be repaired before going on to the next step and making the changes available to the development team as a whole.

6.2.9. Making the Change Set Available

This task submits the changes to the configuration manager for evaluation and subsequent release to the evolving baseline. The set of updated or new elements is known as the **change set.** The change set may be quantified as to the work and effort involved for tracking purposes. Descriptions of the changes are typically added as well so that the evolution of the software can be tracked. Work items are marked as completed, and the change set is submitted to the configuration manager.

At this point, the software is usually incomplete, but it is as correct as possible. It is important that the changes do not break the baseline, so the configuration manager performs integration tests to ensure that the system will not only build but also will properly perform against a limited set of functional tests. This process is described in Chapter 5 in the section on "Continuous Integration."

6.3. Coming Up

At the end of the analysis phase, we have achieved a number of important development goals. First, we've specified and delineated what is in the current prototype. Mostly, that's a matter of really understanding the use cases and system requirements that are included within the current microcycle. However, the purpose also includes the RMAs to be performed and any existing defects to be repaired.

Second, we've constructed an object analysis model that realizes the functional requirements of the specified use cases. This object analysis model consists of classes and objects that are *essential* for the functional correctness of the software. Optimality is not a focus, though, and that will be dealt with in the following design phase.

Third, we’ve generated real, deployable source code. This source code may not be sufficiently optimal, but it *is* functionally correct, robust code. Both the model and the source code are verified using formal unit tests (since they are always in sync, verifying the code simultaneously verifies the model). The unit tests are also configuration-managed and exist with the software that is to be delivered in the final system.

Fourth, we’ve done continuous integration—that is, integration of the evolving baseline—at least daily. This process ensures that integration will not be a huge looming risk downstream because the system *always* integrates.

The next step is to take this functionally correct software and optimize it. That’s the job of design, discussed in the next chapter.

Chapter 7
Agile Design

In the Harmony/ESW process, analysis is concerned with functional correctness. To this end, in prototype definition, we detail the use cases and identify the scope of the work to be done within the current microcycle. We then create an object analysis model that executes, produces deployable code, is formally unit-tested, and is integrated at least daily. Before we start designing, we already have high-quality, functionally correct software with which to work. From the analysis work, we have a number of work products completed (in the context of the current microcycle):

• The microcycle mission statement outlining the scope, intent, and target platforms for the current microcycle effort

• A detailed microcycle schedule and work items list identifying the expected work effort and allocation to personnel

• The system specification for the use cases to be realized in this microcycle, including use cases, scenarios, use case state machines, and textual system requirements

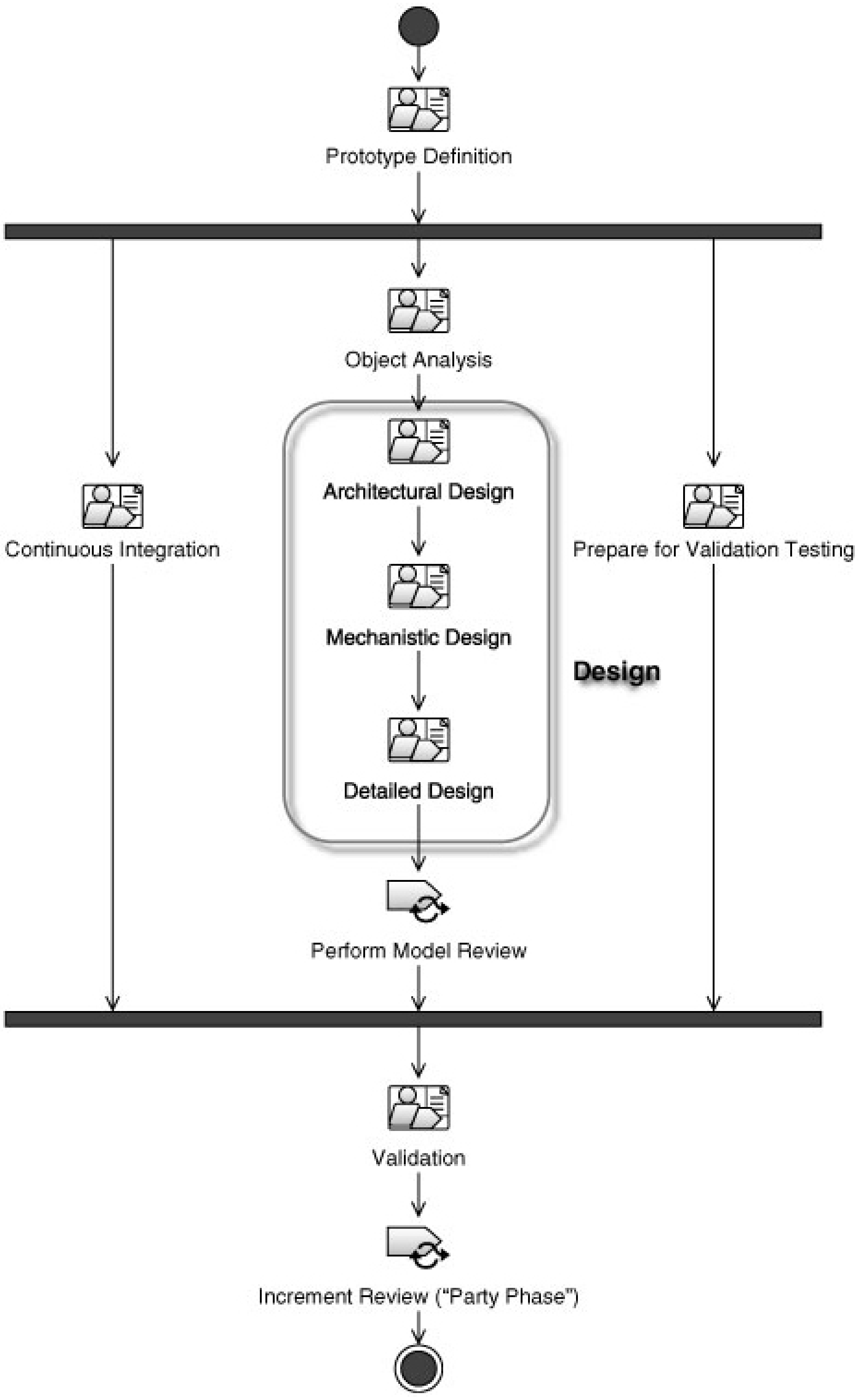
• An object analysis model that functionally realizes the use cases for the microcycle

• Source code generated from the object analysis model

• Unit test cases and fixtures

Now, in design, our goal is to make the software optimal—fast enough, maintainable enough, reliable enough, and so on. The Harmony/ESW design process works at three levels of abstraction (see Figure 7.1), but all levels are concerned with optimization. Architectural design optimizes the system at a gross level by employing design decisions that affect most or all of the system. It does this by considering optimization of different aspects, such as subsystem architecture or concurrency architecture. The next level down is mechanistic design. This level of design optimizes at collaboration scope, where a collaboration is a set of object roles working together to realize a single use case. The smallest scope in design is called detailed design. At this level, individual classes, functions, and data structures are optimized.

Figure 7.1 *Design in the Harmony/ESW microcycle*



7.1. Optimization and the Use of Design Patterns

The Harmony/ESW process has strong guidance on how to perform design. In general, design is applied only against high-quality, functioning software. Optimizing too soon is a common mistake, resulting in very efficient but incorrect systems. Don't go there.

In addition, optimization is the focus of many a developer who isn't sure exactly what to optimize. In the Harmony/ESW process, optimization has a goal: Optimize the system against the set of design criteria weighted by the criticality of each.

7.1.1. Design Patterns

Experienced developers find when they approach a new problem to solve that the situation usually has something in common with a solution they have already either created or seen created by others. The actual problems are not identical and the identical solution will rarely solve the new problem, but the problems are still similar, so a similar solution will probably work. The "similar solution"—generalized and formalized—is called a **design pattern.** Creating design patterns is a problem of abstracting a design solution into its fundamental aspects and removing the problem-context-specific details. This abstracted solution can then be reapplied in different problem contexts that share some aspects of the basic problem the pattern addresses.

Of the two fundamental concerns associated with patterns, the first has to do with the application of patterns. The problem of identifying the nature of the problem and examining the pattern "library" for the best ones to apply is called **pattern hatching.**[1] And, as John Vlissides, author of the excellent book by that name, points out, this name implies that we're not creating something new but "developing from preexisting rudiments." These preexisting rudiments are our captured design patterns that we can use to construct solutions that work in novel circumstances.

1. John Vlissides, *Pattern Hatching: Design Patterns Applied* (Reading, MA: Addison-Wesley, 1998).

The other issue, of course, is the identification and capture of new patterns to add to the library. This process I call **pattern mining**. It involves the abstraction of the problem to its essential properties, creating a generic solution, and then understanding the consequences of that solution in the problem context in which the pattern applies.

Patterns are not just software reuse, but rather a kind of concept reuse. Most patterns are design patterns.[2] Design is always an optimization of an analysis model, and design patterns are always a general concept of how to optimize an analysis model in a particular way with particular effects.

2. There are also analysis patterns—ways of structuring models within business domains or

subject matters. See Martin Fowler, *Analysis Patterns: Reusable Object Models* (Reading, MA: Addison-Wesley, 1996).

Optimization is a fickle partner. Optimization always entails improving some aspects of a system at the expense of others. For example, some patterns will optimize reusability at the expense of worst-case performance. Other patterns will optimize safety at the expense of elevating the system recurring cost (cost per shipped item). Whenever you optimize one set of aspects, you deoptimize others. These "pros and cons" constitute the benefits and costs of design patterns.

One of the big mistakes many developers make is focusing too much (and too early) on optimization, and they often optimize the wrong thing. By "focusing too much" I mean that they don't focus enough on getting the functionality correct in all circumstances. Often, if the developers get it "mostly right," they move on to making it as fast or as small as possible. It needs to be fast or small enough, but only if it does the right thing! By "too early" I mean that the first efforts should be getting the functionality right—not only for typical cases but for *all* cases. The principle of defensive design states that a software element such as a class or operation should always validate its preconditional invariants. This means that incoming parameter ranges should be checked, resource locks must be respected, and out-of-resource (e.g., memory) conditions should be handled. Only then should optimization take place. Additionally, developers are not very good at selecting parts of applications to optimize. Much of the time, developers optimize aspects of the application that have little or no impact on the performance or suitability of the system. It is best to create a properly functioning application and use performance measurement tools to identify where the application is using up time or resources, then use that information to target the design effort.

This concern about when, how, and what to optimize has a huge impact on the use of design patterns. A design pattern is "a generalized solution to a commonly occurring problem." To be a pattern, the problem must recur often enough to be usefully generalizable. The solution must also be general enough to be applied in a wide set of application domains. If it applies only to a single application domain, then it is probably an analysis pattern. An analysis pattern is similar to a design pattern but is a generalized organization of elements that apply to the essential model within a specific application domain such as finance or aerospace. Analysis patterns define ways of organizing problem-specific object analysis models within a single application domain. Martin Fowler's book on the subject[3] provides some examples of domain-specific analysis patterns.

3. Ibid.

According to Gamma et al.,[4] a pattern has four important aspects:

4. Gamma, Helm, Johnson, and Vlissides, *Design Patterns*.

• Name

The name provides a “handle” or means to reference the pattern.

• Purpose

The purpose provides the problem context and the QoS (quality-of-service) aspects the pattern seeks to optimize. The purpose identifies the kinds of problem contexts where the pattern may be particularly appropriate.

• Solution

The solution is the pattern itself.

• Consequences

The consequences are the set of pros and cons of the use of the pattern.

The pattern name brings us two things. First, it allows us to reference the pattern in a clear, unambiguous way, with the details present but unstated. Second, it gives us a more abstract vocabulary with which to speak about our designs. The statement “The system uses a Layered structural architecture with Message Queuing Concurrency distributed across a Symmetric deployment with a Broker Pattern” has a lot of information about the overall structure of the system because we can discuss the architecture in terms of these patterns.

The purpose of the pattern brings into focus the essential problem contexts required for the pattern to be applicable and what qualities of service the pattern is attempting to optimize. This section specifies in which situations the pattern is appropriate and in which situations it should be avoided.

The solution, of course, is the most important aspect. It identifies the elements of the pattern and their roles in relation to each other. These elements are replaced, extended, or subclassed by the application elements to instantiate the pattern.

The consequences are important because we always make trade-offs when we select one pattern over another. We must understand the pros and cons of the pattern to apply it effectively. The pros and cons are usually couched in terms of improvement or degradation of some qualities of service as well as a possible elaboration of problem contexts in which these consequences apply.

7.1.2. Applying Design Patterns

Analysis is driven by what the system must do, whereas design is driven by how well the system must achieve its requirements. A design pattern is a way of organizing a design that improves its optimality with respect to one or more design criteria. Most of the design criteria fall within the realm of QoS. Some of the qualities of service that may be optimized by design patterns are:

• Performance

º Worst case

º Average case

º Predictability

• Schedulability

• Throughput

º Average

º Sustained

º Burst

• Reliability

º With respect to errors

º With respect to failures

• Safety

• Reusability

• Distributability

• Portability

• Maintainability

• Scalability

• Complexity

• Resource usage (e.g., memory)

• Energy consumption

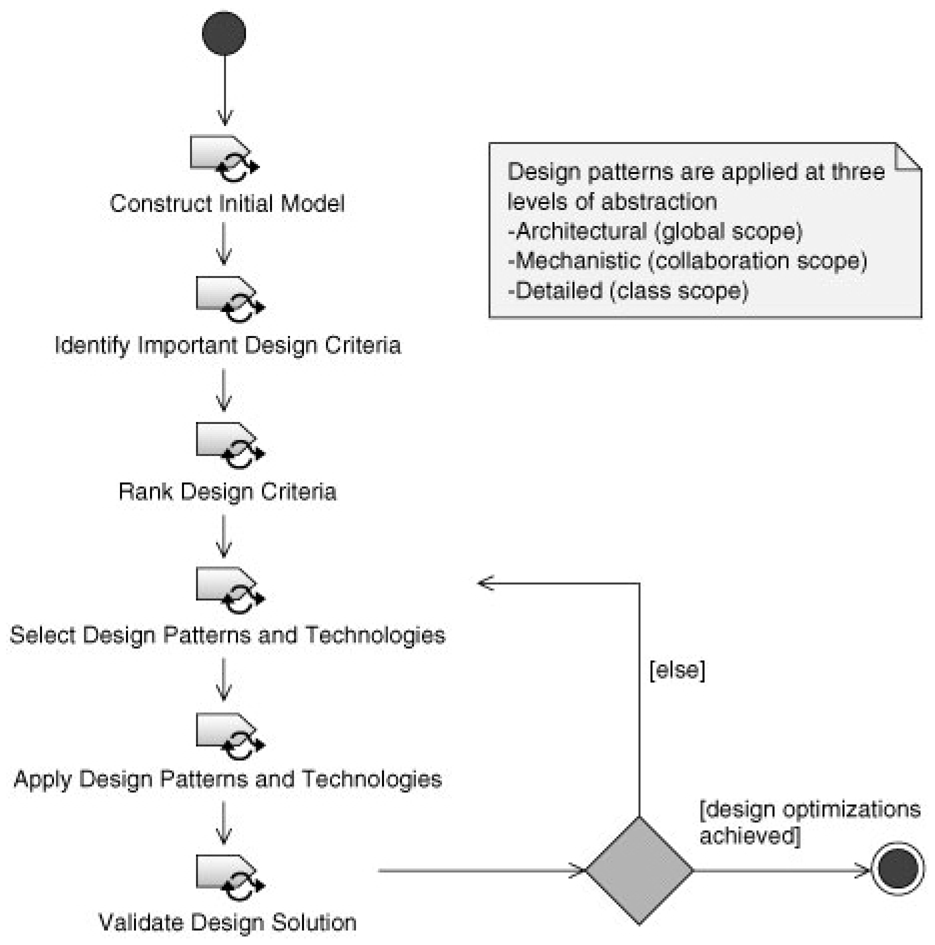
• Recurring cost (i.e., hardware)

• Development effort and cost

Overall, with real-time and embedded systems we care a great deal about achieving design criteria. One of the ways we do this is by reapplying generalized design solutions that have worked well in the past. To effectively do this, we must understand the nature of the optimizations we need and compare them to the optimizations provided by various design patterns. But as a part of that, we need to understand the relative importance to the success of our system of achieving the different design criteria. Is a small memory footprint more or less important than faster execution? Is it more important to have fast response to incoming events or to be able to easily distribute objects across different processors? Design is all about *trade-offs,* and you can make good trade-off decisions only when you know the relative importance of the different design criteria.

Figure 7.2 shows the (abstracted) workflow for design. This workflow changes slightly for the different design abstraction levels, but these are the basic steps in optimizing the design in all three levels of abstraction.

Figure 7.2 *Basic design workflow*


Construct Initial Model
Identify Important Design Criteria
Rank Design Criteria
Select Design Patterns and Technologies
Apply Design Patterns and Technologies
Validate Design Solution
Design patterns are applied at three levels of abstraction
-Architectural (global scope)
-Mechanistic (collaboration scope)
-Detailed (class scope)
[else]
[design optimizations achieved]

1. The first step, “Construct initial model,” is done in analysis and is known as the object analysis model, essential model, or PIM.

2. The next step is to identify the important design criteria, such as those listed above, that are important for the particular optimization scope.

3. Then, these criteria must be ranked and weighted in order of criticality. This is a crucial step because it allows you to explicitly define the relative importance of the different optimization concerns.

4. Based on this weighted ranking of criteria, we select patterns and technologies (such as bus architectures or operating systems) that optimize the more important criteria at the expense of the least.

5. We apply the patterns, in a step known as “pattern instantiation,” usually by replacing the elements that serve as the formal parameters of the pattern with elements from our analysis model.

6. Last, we validate our design solution. We do this to ensure two things. First, we want to make sure that we haven’t broken the already working functionality of the system. Second, we

want to ensure that we've achieved our optimization goals.

For example, Figure 7.3 shows the relative criticality ranking of several different design criteria for a particular system. If we rank the design criteria for a system according to Figure 7.3, then we might use patterns[5] such as the following:

• Static Priority Pattern

This pattern provides good responsiveness to events (worst-case performance) but is more complex and less inherently predictable than some scheduling methods.

• Fixed Block Memory Allocation Pattern

This pattern improves reliability by ensuring that memory fragmentation is never an issue but at the cost of wasting memory.

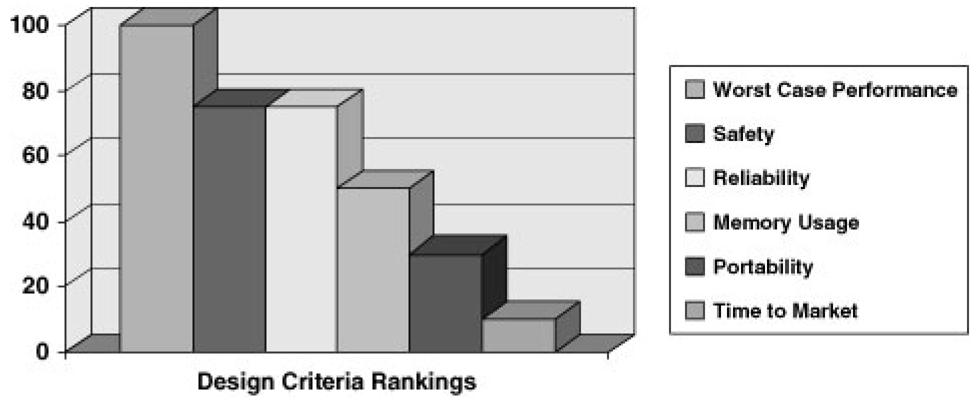
• Channel Pattern

This pattern creates architectural subsystems that provide "sense-to-actuation" end-to-end functionality that can improve reliability and safety by providing a basic mechanism for large-scale redundancy.

• Triple Modular Redundancy (TMR) Pattern

This pattern uses three channels and a voting mechanism to improve reliability but with increased recurring cost, power consumption, heat, and weight.

Figure 7.3 *Design criteria ranked by criticality (1)*



5. For these specific patterns, see my *Real-Time Design Patterns*.

If, instead, we take the *very same system* but optimize it to a different set of design criteria and rankings, we would use different patterns and end up with a functionally equivalent but very differently performing system. If we use the weighted set of criteria shown in Figure 7.4, then we might apply patterns such as the following:

• Cyclic Executive Scheduling Pattern

This concurrency architecture pattern is highly predictable and reliable but is demonstrably suboptimal in terms of responsiveness to events.

• Dynamic Memory Pattern

This memory pattern comes “out of the box” with modern compilers but is subject to memory leaks and memory fragmentation. It optimizes simplicity at the expense of reliability.

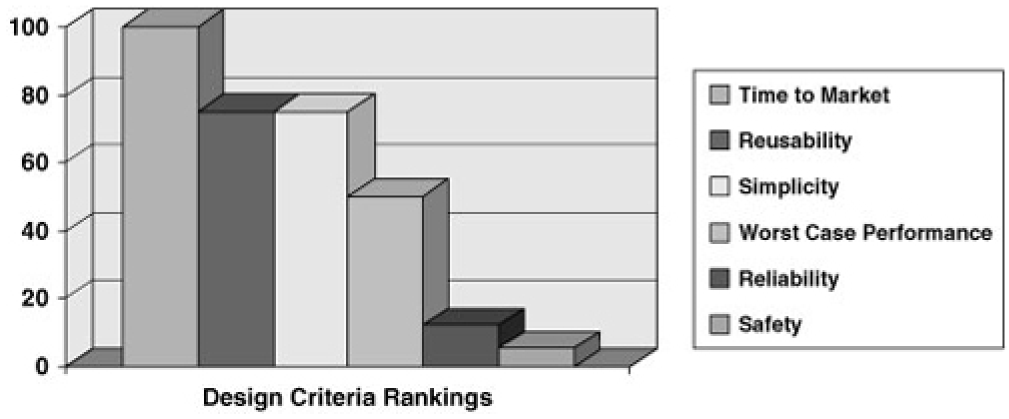
• Recursive Containment Pattern

This pattern provides a very simple recursively self-similar architectural organization but doesn’t lend itself well to the addition of redundancy.

• Virtual Machine Pattern

This pattern constructs the application as sitting on top of a virtual machine; porting the application to a new environment is a matter of porting the virtual machine and not the application. This pattern enhances portability (a kind of reusability) at the expense of runtime efficiency.

Figure 7.4 *Design criteria ranked by criticality (2)*



Let’s now look at the three design activities and see how they use design patterns to achieve the necessary optimizations.

7.2. Architectural Design

Architectural design identifies strategic design decisions that optimize the system at a gross, overall level. If you look in the software engineering literature, the topic of architectural design has self-organized into a number of different areas of concern. Five of these areas form the primary architectural views in the Harmony/ESW process. There are, naturally enough, other areas of concern that may be important for different kinds of systems. However, these secondary views have less impact on the structure of the system as a whole.

7.2.1. Primary and Secondary Architectural Views

Figure 7.5 shows the primary architectural points of view within the Harmony/ESW process. These different views were discussed in some detail in Chapter 2, "Goals and Benefits of Model-Driven Development," and there's no reason to repeat that discussion here. It is enough to recite the primary concerns of these views:

• Subsystem and component view

This view focuses on the identification, responsibilities, and interfaces of the large-scale architectural units, known either as subsystems or as components.

• Concurrency and resource view

This view is concerned with the identification of concurrency units (tasks or threads), their scheduling, the mapping of "passive" elements to the concurrency units, specification of scheduling and timeliness metadata (e.g., worst-case execution time), identification of resources, and how they are reliably shared across concurrency boundaries.

• Safety and reliability view

This view centers on the identification, isolation, and correction of faults at runtime, most often through the management of redundancy.

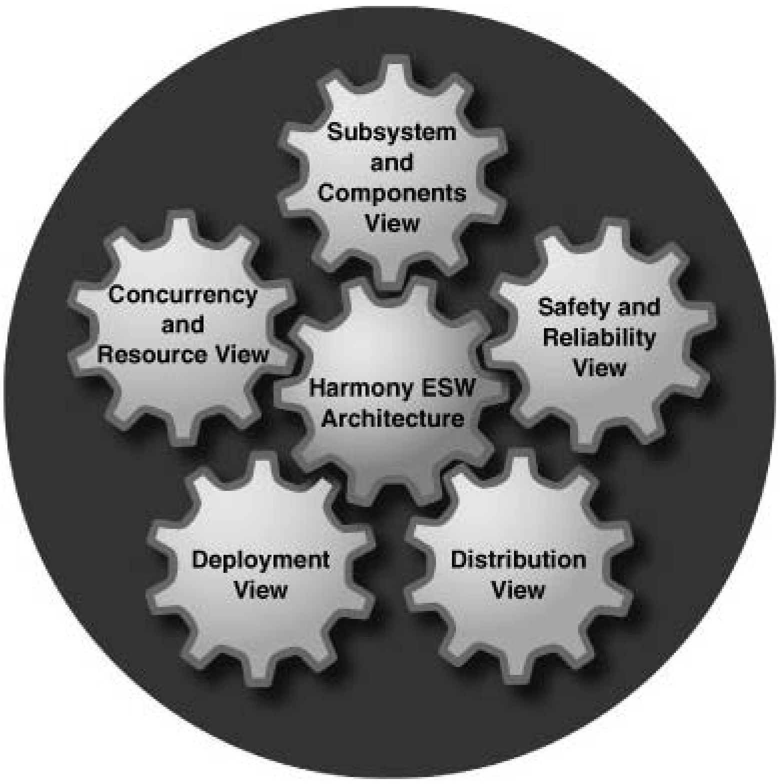
• Distribution view

This view concentrates on the technologies and techniques by which objects may be distributed across multiple address spaces and by which they collaborate to achieve system goals (e.g., realize system use cases).

• Deployment view

This view focuses either on the mapping of the software to the digital electronics (software-only perspective) or on the relative responsibilities, interactions, and interfaces among elements from different engineering disciplines, such as mechanical, analog electronics, digital electronics, optical, chemical, and software (system perspective).

Figure 7.5 *Primary architectural views*

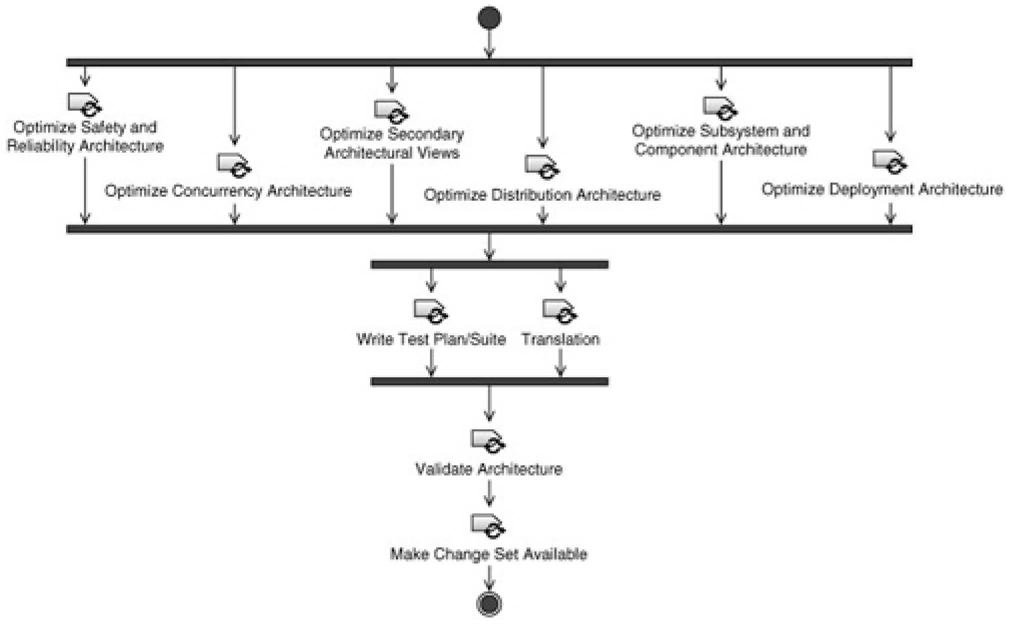

Subsystem and Components View
Concurrency and Resource View
Safety and Reliability View
Harmony ESW Architecture
Deployment View
Distribution View

The architectural views are important because they allow us to focus on more or less independent aspects of the strategic design. Each has its own set of technologies and design patterns. In fact, the computer science literature has already self-organized into these separate topics. The system "architecture" is really the sum of the decisions made in each of these areas of concern.

7.2.2. Architectural Design Workflow

Figure 7.6 presents the workflow for architectural design in Harmony/ESW. The tasks between the two heavy horizontal lines are done optionally and in parallel. By "optionally" I mean that within a given prototype, there may be no design decisions made for that architectural viewpoint. Over time, most of these views will be addressed in the architecture, although a couple—such as safety and reliability and distribution—are not always required, depending on the nature of the system under development.

Figure 7.6 *Architectural design workflow*


Optimize Safety and Reliability Architecture
Optimize Concurrency Architecture
Optimize Secondary Architectural Views
Optimize Distribution Architecture
Optimize Subsystem and Component Architecture
Optimize Deployment Architecture
Write Test Plan/Suite
Translation
Validate Architecture
Make Change Set Available

7.2.3. Optimizing Subsystem and Component Architecture

This task is usually done relatively early in the development project because it is used as a means to manage the teams involved in building the system. For large projects, it is common to have an interdisciplinary team per subsystem, plus a team focusing on the shared assets and another on overall system integration concerns. Once the subsystems are identified and their requirements (and use cases) and interfaces clarified, the subsystem teams can work more or less independently.

The steps involved in this task are:

1. Identify and rank the design criteria.

2. Identify the subsystems.

3. Decompose via the top-down approach (alternative 1):

a. Decompose system use cases into subsystem use cases to allocate responsibilities to the subsystems.

b. Decompose system use case scenarios into subsystem-level scenarios showing the necessary interaction of the subsystems.

c. Cluster messages between subsystems into interfaces.

4. Integrate via the bottom-up approach (alternative 2):

a. Allocate services and responsibilities to the subsystems.

b. Cluster service invocations, requests, and responses "up" into interfaces.

5. Specify physical interfaces to realize the logical interfaces.

6. Put interfaces in the shared model and under CM.

Of course, this workflow may have been done during the prototype definition activity in the analysis phase, as discussed in the previous chapter.[6] This is done, for example, if a separate systems engineering group is responsible for the subsystem architecture. In environments in which the software team does the subsystem definition, it can wait until the architectural design phase.

6. See the section on "White-Box Analysis" in Chapter 6.

A subsystem is a large-scale architecture unit of a system. In a software-oriented development project in which the hardware platform is known and understood, a subsystem is solely a measure for constructing large-scale units of software that appear at runtime. In a systems engineering environment, it includes hardware (e.g., digital electronics, analog electronics, mechanical design, chemical design, etc.) as well as the software.

Good principles of subsystem definition include:

• Elements within a subsystem are tightly coupled in terms of time, functional, or data dependency.

• Elements across a subsystem boundary are loosely coupled in terms of time, functional, or data dependency.

• Elements should collaborate to realize a few narrowly focused capabilities or use cases within the context of the subsystem.

• Subsystems should provide a few well-defined interfaces.

• A subsystem should ideally be constructed by a single colocated team.

• A subsystem model should be kept in a separate model or separate package in a singular system model.

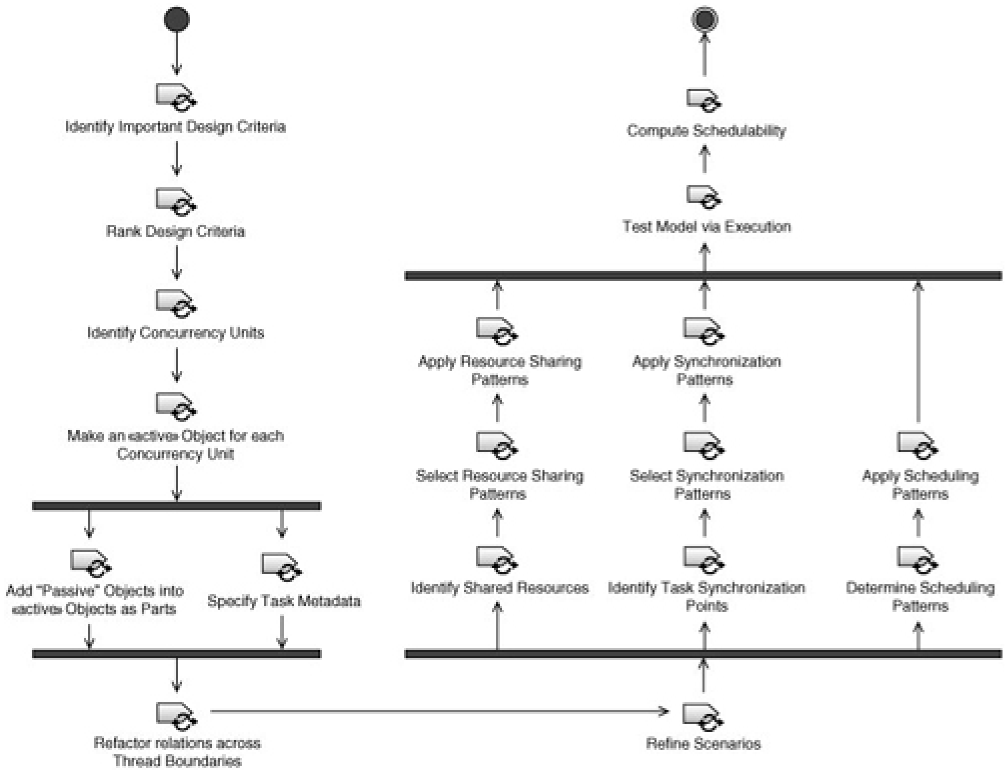
• Classes and types shared between subsystems should be defined within a separate shared model or separate shared package.

This aspect of architecture is usually addressed in the first microcycle (or possibly even prior to that). This lays the foundation for the large-scale pieces of the system design to be handed off to the different teams collaborating on the development.

7.2.4. Optimizing Concurrency and Resource Management Architecture

The concurrency and resource management architecture is of primary concern to real-time and embedded system developers because it has such a profound impact on performance. The workflow is consequently rather more elaborate than that of the other architectural viewpoints, as shown in Figure 7.7.

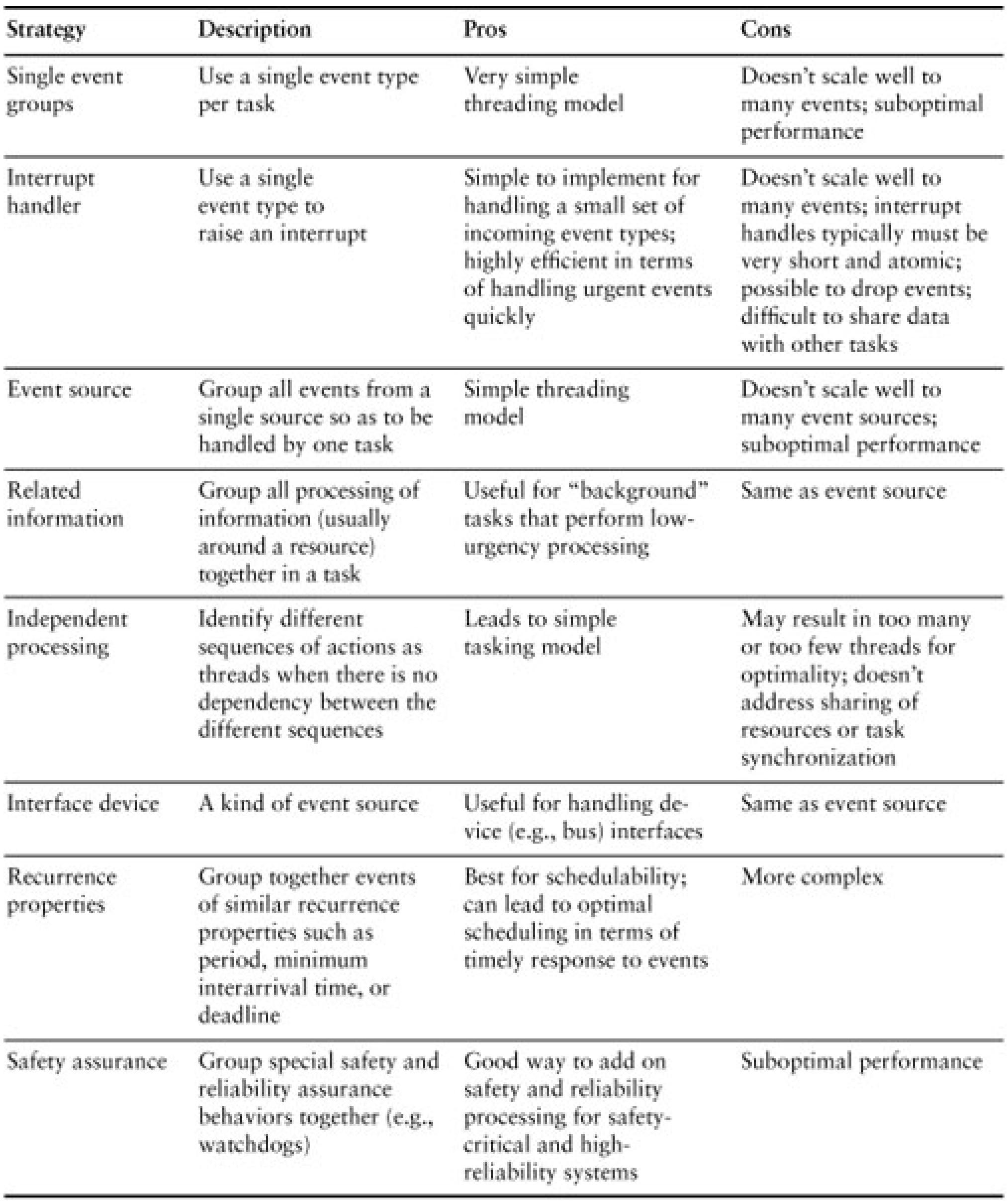
Figure 7.7 *Concurrency and resource workflow*


Identify Important Design Criteria
Rank Design Criteria
Identify Concurrency Units
Make an «active» Object for each Concurrency Unit
Add "Passive" Objects into «active» Objects as Parts
Specify Task Metadata
Refactor relations across Thread Boundaries
Refine Scenarios
Identify Shared Resources
Identify Task Synchronization Points
Determine Scheduling Patterns
Select Resource Sharing Patterns
Select Synchronization Patterns
Apply Scheduling Patterns
Apply Resource Sharing Patterns
Apply Synchronization Patterns
Test Model via Execution
Compute Schedulability

The first couple of steps identify and rank the design criteria you're trying to optimize within the concurrency and resource view. Again, it could be worst-case performance, schedulability (ability to reliably meet deadlines), predictability, simplicity, or some other criterion.

Next we identify the concurrency units (tasks or threads). In the Harmony/ESW process we have a number of strategies that can be used, together or in isolation. These are listed in Table 7.1.

Table 7.1 *Task Identification Strategies*



Strategy	Description	Pros	Cons
Single event groups	Use a single event type per task	Very simple threading model	Doesn't scale well to many events; suboptimal performance
Interrupt handler	Use a single event type to raise an interrupt	Simple to implement for handling a small set of incoming event types; highly efficient in terms of handling urgent events quickly	Doesn't scale well to many events; interrupt handles typically must be very short and atomic; possible to drop events; difficult to share data with other tasks
Event source	Group all events from a single source so as to be handled by one task	Simple threading model	Doesn't scale well to many event sources; suboptimal performance
Related information	Group all processing of information (usually around a resource) together in a task	Useful for "background" tasks that perform low-urgency processing	Same as event source
Independent processing	Identify different sequences of actions as threads when there is no dependency between the different sequences	Leads to simple tasking model	May result in too many or too few threads for optimality; doesn't address sharing of resources or task synchronization
Interface device	A kind of event source	Useful for handling device (e.g., bus) interfaces	Same as event source
Recurrence properties	Group together events of similar recurrence properties such as period, minimum interarrival time, or deadline	Best for schedulability; can lead to optimal scheduling in terms of timely response to events	More complex
Safety assurance	Group special safety and reliability assurance behaviors together (e.g., watchdogs)	Good way to add on safety and reliability processing for safety-critical and high-reliability systems	Suboptimal performance

In the UML, «`active`» classes are the roots of threads; that is, they own their own OS task or thread and they, and their parts, execute in the context of that thread. They also usually manage an event queue shared by all the objects running in the thread, pulling out events when they run and passing them off to the appropriate objects. Once you've identified the desired concurrency units, create an «`active`» class for each. Usually, there is a single instance of each «`active`» class but not always.

Once the tasks are defined, they should be characterized. This is done both by adding the "passive" elements from your analysis model into the «`active`» class as parts[7] and by adding in the concurrency metadata that characterizes the task. This metadata is selected to support the kind of schedulability analysis you want to perform but usually contains metadata such as:

- If the task is periodic:

° Task period

° Task jitter (variance in the period)

7. In the UML specification, parts are roles that instances play in the context of their encapsulating composite owner (in this case, the `«active»` class).

• If the task is aperiodic:

° Initiating event minimum interarrival time

° Initiating event average arrival time

° Initiating event burst length

• Priority

• Worst-case execution time

• Average execution time

• Worst-case blocking time

• Deadline

• Resource access control pattern (e.g., priority inheritance, highest locker, or priority ceiling)

Relations may need to be refactored if they occur across thread boundaries. The scenarios that show the interaction of the passive elements should be elaborated to accurately show the new pattern of interactions necessitated by the concurrency architecture.

Following this, three types of concerns must be addressed:

• How the tasks should be scheduled

• How resources will be shared

• If and how the tasks will synchronize

They are shown in parallel in the right half of Figure 7.7, but that just means that they can be tackled in any desired order.

There are many ways to schedule tasks, such as Priority-Based Preemption, Cyclic Executive, Time-Driven Multitasking, Interrupt-Driven, and others. Among priority-based schemes, the scheduling can be urgency-based or criticality-based, as discussed in Chapter 1, "Introduction to Agile and Real-Time Concepts." Priorities can also be assigned statically (at design time) or dynamically (at runtime). An appropriate set of scheduling patterns must be selected that optimize the weighted set of design criteria you're trying to optimize.

Resources are elements of finite capacity; in this context, we care about resources that must be shared across concurrency boundaries. These resources may be "passive" and run in the context of the caller, or they may be active and process requests asynchronously. It is almost always necessary to serialize access to these resources to prevent invalid data and data corruption. This can be done through the application of a number of patterns, such as Atomic Regions, Queuing, and the use of locks and semaphores.

Concurrency architectures are simple—unless, that is, they need to synchronize or share resources. Unfortunately, this is always the case in real system designs. There are many ways synchronization can be done as well using synchronization points, rendezvous patterns, and the like.

Once these concerns are addressed, you can validate your design via execution, just as we did with the analysis model. It should also be noted that you don't have to do all of this first and then begin to execute the model. It is better to execute at least after every new pattern is introduced, and more frequently is even better. This execution can be used to ensure that the functionality wasn't broken and the desired optimization has been achieved.

Note

In addition to testing, concurrency theory has a rich body of mathematical techniques for analyzing such systems for performance and schedulability. The most common approach is based on the work of Liu and Layland,[8] and the practitioner's bible in such analysis is the reference by Klein et al.[9]

8. C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM,* January 1973.

9. Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Gonzáles Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems* (Norwell, MA: Kluwer Academic Publishers, 1993).

7.2.5. Optimizing Distribution Architecture

Distribution architecture is a concern when the system contains multiple address spaces or CPUs. It focuses on the patterns, techniques, and technologies by which objects can be distributed across those address spaces and by which they can find each other and exchange messages and information. The steps for this architectural view include:

1. Identify and rank the design criteria for distribution.

2. Analyze the message traffic load requirements.

3. Identify the physical media to be used for communications.

4. Select the patterns and technologies:

a. Select design patterns to optimize communications against the design criteria.

b. Select and/or specify the communication protocols to run over the physical media.

c. Select and/or specify middleware.

5. Analyze the bus traffic over the communication media with patterns and protocols in place.

6. Tune the communications protocols.

7. Validate the distribution architecture.

Different approaches to distribution apply to different needs. In some cases, we need good average response but if an occasional message is significantly delayed there is no problem. In those cases, a collision-detection network, such as Ethernet, is highly appropriate. Ethernet works by detecting when collisions occur (multiple senders collide when trying to send a message at the same time) and randomly retrying to send. If the network bandwidth is much higher than the message load, the network works well, but occasionally a message may be significantly delayed. Ethernet also saturates at about 30% load, meaning that at loads greater than 30% almost all of the bus time is spent arbitrating collisions and very little of it is spent actually delivering messages. In contrast, the CAN bus uses a bit-dominance protocol to avoid collisions; when a bit is put on the bus, the sender listens to the actual result on the bus. Since some bits are dominant (i.e., always win), as soon as a collision occurs (a nondominant bit was sent but a dominant bit occurred on the bus), the defeated sender drops off, allowing the other sender to continue. Less time is spent arbitrating collisions, and the 11- or 29-bit header can be used to prioritize messages. On the other hand, the CAN bus has a very limited message size (8 bytes), so it can take many messages to get a lot of data across. The CAN bus works best for short messages.

The physical media usually refers to a bus or network of some kind but is not always so limited. Objects can communicate using other modalities as well, such as IPCs and shared memory. Additionally, cost and power are also concerns that must be addressed. When the nature of the message traffic and the system requirements (good average bandwidth versus schedulability, for example) are understood, a good choice of communication media can be made.

Once the physical media is addressed, some level of protocol (typically known as the **data link protocol**) handles the transmission and reception of network packets. Higher-level protocols can sit on top of these basic protocols to provide more elaborate services, such as session-oriented and sessionless messaging, reliable delivery, automatic transformation of data into a "network format," and application-specific protocol services. These may be purchased as third-party components in some cases, or they may become a part of your design effort.

Middleware is software that connects different architectural elements, such as components, subsystems, and systems. There are a number of commercial middleware packages available, such as CORBA, COM, DCOM, and DDS. They all have pros and cons and work a bit differently. CORBA has the advantage of broad commercial support for the Object Request Broker (ORB) Pattern but can be a bit heavyweight for many real-time and embedded applications.[10] DDS works well for passing data but is less common and has far less commercial support. COM and DCOM are Microsoft-platform-specific. Not all real-time and embedded systems will use middleware, but if you need it, it's nice to be able to buy rather than build!

10. Most people talk about CORBA as if it were a single standard. Actually it is a set of related standards. There's CORBA, High-Performance CORBA, Embedded CORBA, Real-Time CORBA, and so on. Interested parties can find more information at http://www.omg.org/.

There are many patterns that can be used to optimize the distribution architecture. Douglas Schmidt[11] has written extensively on this topic as have others. Once the patterns have been identified and applied, the resulting design must be tested and evaluated. Most commercial protocols and middleware support tuning to improve performance in one way or another as well.

11. A recommended book is vol. 4, *A Pattern Language for Distributed Computing*, of Buschmann et al.'s *Pattern-Oriented Software Architecture* for which Schmidt is one of the authors.

7.2.6. Optimizing Safety and Reliability Architecture

This architectural viewpoint applies only to high-reliability and safety-critical systems and may be ignored if your system doesn't have those special concerns. Safety and reliability architecture attempts to address faults as the system operates in its execution context. This means that the system must be able to detect when things go wrong, identify what to do about it, and then follow through with those actions. Safety seeks to prevent harm, while reliability tries to maximize the percentage of time the system can deliver services. When there is a fallback fault-safe state, these can be opposing concerns; it may very well be safer to stop delivering services if the state of the system cannot be absolutely determined, but this lowers its reliability. When there is no fault-safe state, improving reliability usually improves safety as well.

The steps to optimizing the safety and reliability of the system are the following:

1. Identify and rank the design criteria for this architectural view.

2. Use FTA, FMEA, or other analytic techniques to identify how hazardous conditions can occur.

3. Identify control measures and patterns to improve safety and reliability.

4. Integrate control measures into the architecture.

5. Update the hazard analysis.

6. Validate the safety and reliability architecture.

The hazard analysis, FTA, and FMEA were discussed previously in Chapter 5, "Project Initiation." I won't rehash the basics of those work products here. However, these work products are updated continuously throughout the project. In Chapter 5, I talked about how, during project initiation, we do early safety and reliability analyses to scope and understand the project needs. Later, in analysis (Chapter 6), we update them as we refine the system requirements for the current microcycle. This is still about specifying safe and reliable requirements. Finally, in architectural design we actually add specific design patterns and technologies to address the concerns inside our system implementation. The requirements should not go into which patterns should be used to meet the safety and reliability needs; rather, they should specify those needs only. In this architectural viewpoint, we select the specific patterns we will use to address those needs in an optimal way. The inserted patterns are known as "control measures," and, like all patterns, their selection balances applicability, need, benefit, and cost. Some patterns address only failures (e.g., Homogenous Redundancy Pattern[12]) at a relatively low design cost but high recurring cost, while others address failures and errors (e.g., Heterogeneous Redundancy Pattern). Some patterns allow processing to continue in the presence of faults (e.g., Triple Modular Redundancy), while others require the system to go to a fault-safe state (e.g., Monitor-Actuator Pattern).

12. The patterns mentioned here are from my book *Real-Time Design Patterns*. Interested readers should also check out Robert Hanmer, *Patterns for Fault Tolerant Software* (New York: John Wiley & Sons, 2007).

Once appropriate patterns are selected, they are instantiated in the architecture. For the most part, this means adding redundant channels and fault management logic. The resulting architecture is then analyzed using the FTA and FMEA methods discussed previously, and the results are captured in the hazard analysis. Finally, as always, the architecture is validated. In this case, it often requires **fault seeding,** a technique in which faults are inserted into the system to ensure that they are handled properly. I've been known to yank out chips in running systems, unplug sensors, and otherwise simulate faults to test the safety and reliability control measures of a system under development.

7.2.7. Optimizing Deployment Architecture

The deployment architecture is really all about allocation of responsibilities to elements coming from different engineering disciplines. For projects in which the hardware is COTS or at least already constructed and well understood, the deployment architecture consists of mapping the software to the electronics. For projects in which the hardware and software are codeveloped, the deployment architecture is about allocation of responsibilities to different engineering disciplines, including analog electronics, digital electronics, mechanical engineering, chemical engineering, optics, and software. As discussed in Chapter 2, while UML deployment diagrams suffice for simple cases, they are inadequate for systems engineering projects. In these more complex situations, class diagrams are more powerful for depicting the deployment architecture.

In the context of the microcycle, one of the elements of the microcycle mission statement is the set of target platforms (see Chapter 6). Early prototypes may be targeted only to desktop computers but over time the evolving hardware is incorporated more and more into later prototypes. This means that the deployment architecture does evolve over time; processors change, buses change, and—more commonly—hardware interfaces change. The deployment architecture manages all those concerns.

The steps involved in the optimization of the deployment architecture include:

1. Identify and rank the design criteria.

2. Identify the engineering disciplines involved.

3. Perform trade studies where appropriate.

4. Allocate system responsibilities to the engineering disciplines.

5. Specify the interdisciplinary interfaces.

Trade studies are performed to evaluate different combinations of hardware and software. They are not always crucial when using COTS hardware but are vital when the software and hardware are to be codeveloped. There are a number of approaches to doing a trade study, but most involve the following steps:

1. Identify and rank the decision criteria.

2. Identify alternatives.

3. Model and evaluate each alternative in terms of the quantitative criteria.

4. Select an alternative.

5. Perform sensitivity analysis.

As in design, trade studies are an exercise in selecting a “best” middle path from a set of alternatives. The “best” middle path can be computed as the alternative that does the best job of maximizing the overall set of weighted decision criteria. The best alternative is not necessarily the cheapest, the lightest, the sturdiest, the most reliable, the most available, or the most powerful; it is the combination of all those (and other) criteria, in accordance with their relative importance, that makes an alternative the best.

A format that I like to use for the trade study results in an alternative-versus-criteria matrix, such as Table 7.2. In this table, the relative criticalities of the decision criteria are normalized so that they add up to 100%. The alternatives are evaluated on how well they meet that objective. The weighted results are the sum of the products of the factor criticality and the degree to which that factor is true for the alternative (see Equation 7.1).[13]

Table 7.2 *Example Trade Study Results*

Criticality	Performance 39%	Low Power 13%	Recurring Cost 10%	Development Cost 20%	Reliability 3%	Schedule Risk 15%	Weighted Results 100%
Alternative A	100	60	100	60	100	100	86.80
Alternative B	60	100	100	30	60	100	69.20
Alternative C	60	60	100	30	60	60	58.00

Weights:	Excellent	100
	Acceptable	60
	Marginal	30
	Unacceptable	0

13. I should note, though, that I do use quantitative analysis to drive my optimizations, but I don’t rely exclusively on it. My experience is that while I find the information useful as an input, in cases where my intuition clearly differs from the quantitative analysis, my intuition always turns out to be correct. However, in cases where my intuition (which I call “using the Force”) is indistinct, then quantitative analysis is a profound asset. I believe this is because intuition is unconsciously taking into account factors that didn’t make it into the quantitative analysis, but should have. Your mileage may vary ☺.

The deployment architecture is not necessarily a monolithic outcome. It can change over time because of increasing availability of codeveloped parts, changing requirements, or changing supplier availability. In a systems engineering environment it is most commonly the former.

Equation 7.1 *Alternative Result*

$$Result_{Alternative} = \sum Criteria_j \times Weight_j$$

7.2.8. Optimizing Secondary Architectural Views

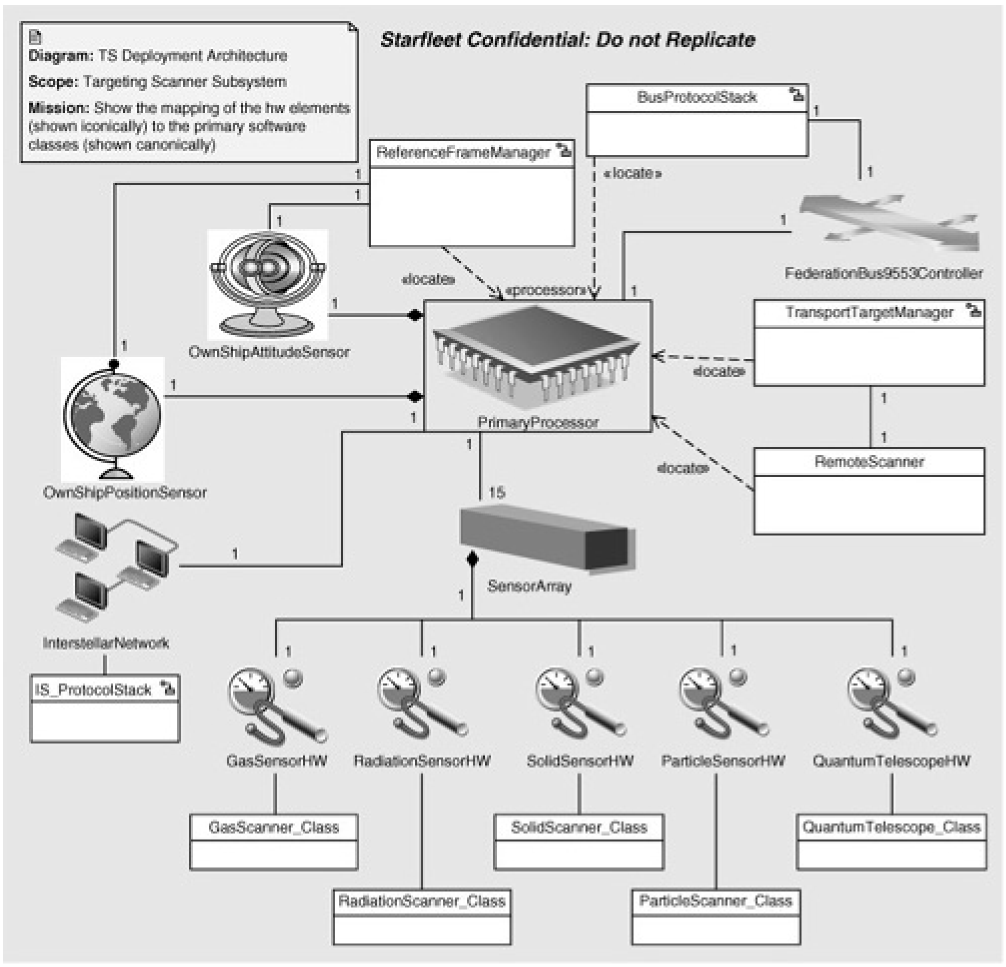
There are a number of secondary architectural views as well. They may not play an important role in your specific system, but even if they do, they typically have less significant impact on the overall structure and behavior of your system than do the five primary architectural views. This is not to say that they are not important, only that their impact tends to be less than that of the previously described views.

7.2.9. Adding Architecture to the Transporter

By way of example, let’s look at adding some architectural aspects to the transporter. Since this is a (short!) book on process and not on design techniques, I’ll show only three architectural views here.

The first of these is the deployment architecture for the Targeting Scanner subsystem (Figure 7.8). This is a class diagram rather than a UML deployment diagram, for the reasons mentioned earlier.[14] All the hardware or physical (i.e., combination of hardware and software) elements are shown in iconic form; the software elements are shown in the standard UML canonical form. The associations between physical elements mean that there is some physical interface, such as a memory, port, bus, or interrupt mapping. The association between the software classes and the hardware elements means that the software accesses the hardware through some physical interface. The `«locate»` dependency means that the instance of the class will execute on the specified CPU known as the `PrimaryProcessor`. Only the main classes are shown; the secondary classes also run on the same processor, but I didn’t show them for readability reasons.

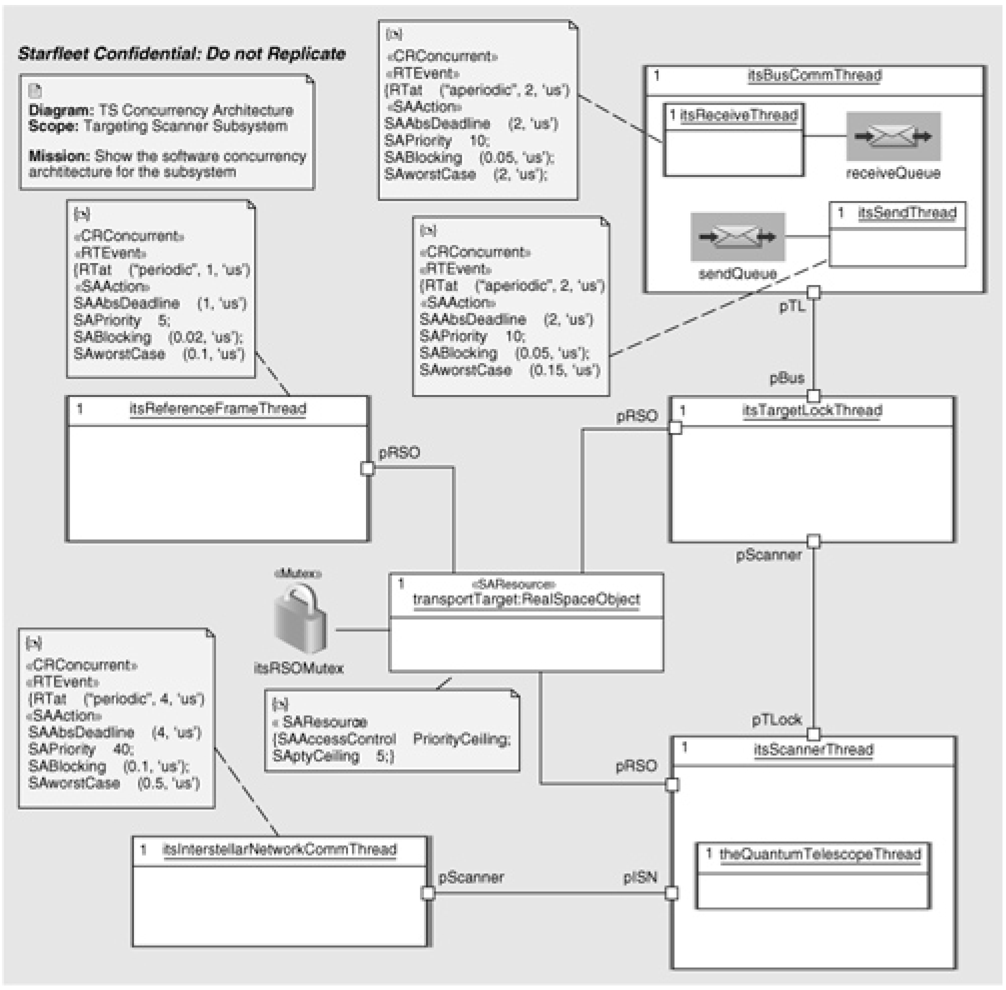
Figure 7.8 *Targeting Scanner deployment architecture*



14. Creating an equivalent diagram using the UML deployment diagram is left as an exercise for the reader.

The second architectural viewpoint is the concurrency viewpoint (see Figure 7.9). The concurrency units are shown as «`active`» objects, and they are characterized with schedulability metadata. Some of these (but not all) are shown in constraints. The `RealSpaceObjects` are resources used by several of the threads. Not shown is the allocation of the passive objects to the threads. This would typically be shown in a separate structure diagram for each «`active`» class.

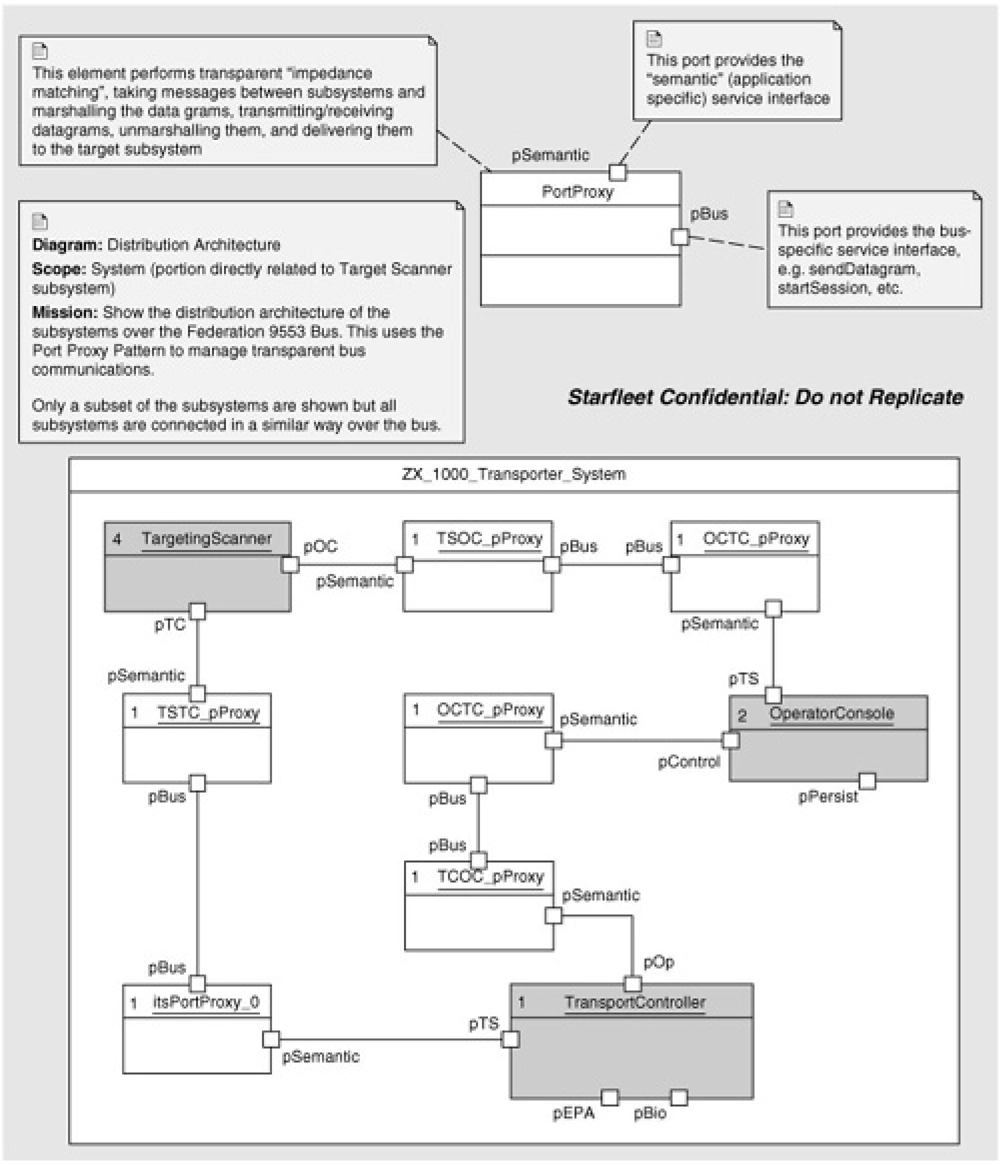
Figure 7.9 *Targeting Scanner concurrency architecture*



Starfleet Confidential: Do not Replicate
Diagram: TS Concurrency Architecture
Scope: Targeting Scanner Subsystem
Mission: Show the software concurrency architecture for the subsystem
«CRConcurrent» «RTEvent» {RTat ("aperiodic", 2, 'us') «SAAction» SAAbsDeadline (2, 'us') SAPriority 10; SABlocking (0.05, 'us'); SAworstCase (2, 'us');
1 itsBusCommThread
1 itsReceiveThread
receiveQueue
1 itsSendThread
sendQueue
«CRConcurrent» «RTEvent» {RTat ("periodic", 1, 'us') «SAAction» SAAbsDeadline (1, 'us') SAPriority 5; SABlocking (0.02, 'us'); SAworstCase (0.1, 'us')
«CRConcurrent» «RTEvent» {RTat ("aperiodic", 2, 'us') «SAAction» SAAbsDeadline (2, 'us') SAPriority 10; SABlocking (0.05, 'us'); SAworstCase (0.15, 'us')
pTL
pBus
1 itsReferenceFrameThread
pRSO
pRSO
1 itsTargetLockThread
pScanner
«Mutex»
itsRSOMutex
1 «SAResource» transportTarget:RealSpaceObject
«CRConcurrent» «RTEvent» {RTat ("periodic", 4, 'us') «SAAction» SAAbsDeadline (4, 'us') SAPriority 40; SABlocking (0.1, 'us'); SAworstCase (0.5, 'us')
« SAResource» {SAAccessControl PriorityCeiling; SAptyCeiling 5;}
pTLock
1 itsScannerThread
pRSO
1 theQuantumTelescopeThread
1 itsInterstellarNetworkCommThread
pScanner
pISN

Last, we see the distribution architecture for the internal bus connecting the subsystems (see Figure 7.10). This system uses the Port Proxy Pattern[15] to transparently connect the subsystems. In this case, the proxy elements sit “between” the subsystems; they translate “semantic” messages from the subsystem to suitable bus messages, transmit them, translate them back into a copy of the original message, and then deliver the message to the target subsystem. Not all subsystems are shown; I limited my focus to those affecting the Targeting Scanner subsystem more or less directly.

Figure 7.10 *Distribution architecture*


This element performs transparent "impedance matching", taking messages between subsystems and marshalling the data grams, transmitting/receiving datagrams, unmarshalling them, and delivering them to the target subsystem

This port provides the "semantic" (application specific) service interface

This port provides the bus-specific service interface, e.g. sendDatagram, startSession, etc.

Diagram: Distribution Architecture
Scope: System (portion directly related to Target Scanner subsystem)
Mission: Show the distribution architecture of the subsystems over the Federation 9553 Bus. This uses the Port Proxy Pattern to manage transparent bus communications.

Only a subset of the subsystems are shown but all subsystems are connected in a similar way over the bus.

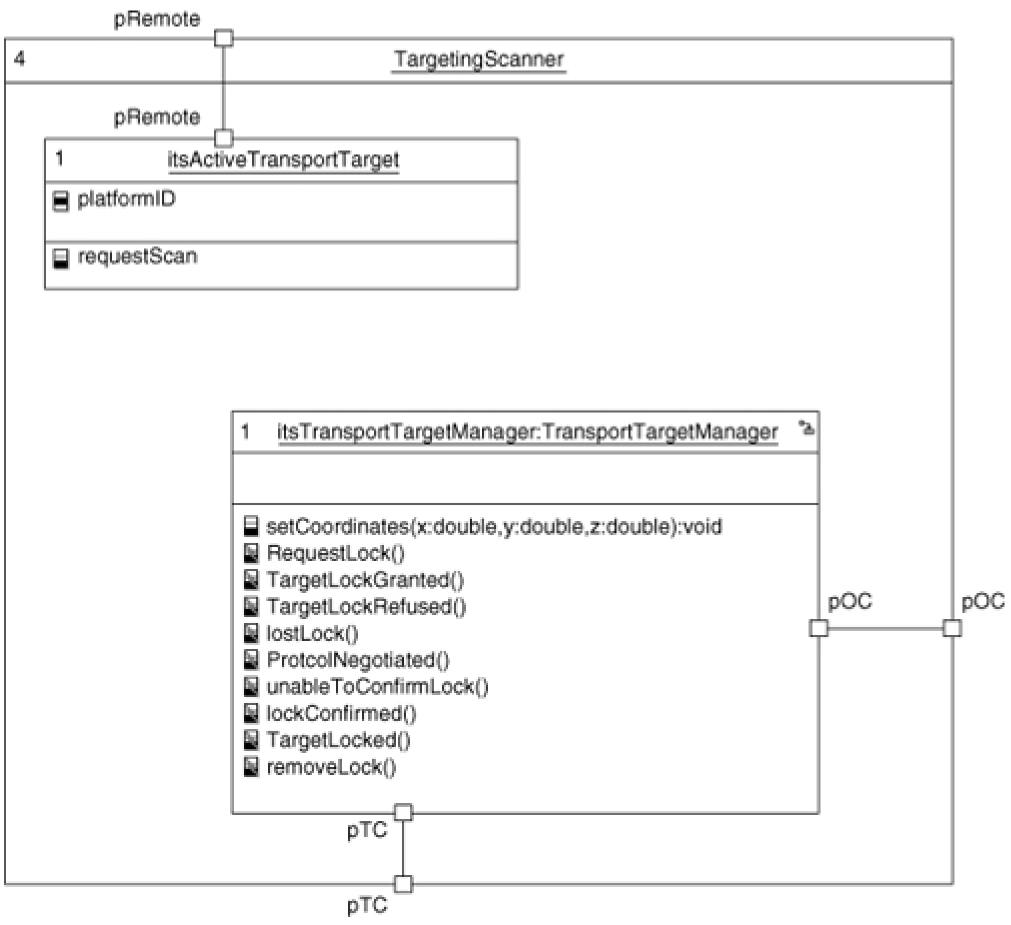
Starfleet Confidential: Do not Replicate

15. See my *Real-Time UML Workshop for Embedded Systems* for a detailed description of the design pattern.

Figure 7.11 shows the objects that connect to the ports defined for the Targeting Scanner subsystem in the subsystem architecture. Specifically, the `ActiveTransport-Target` instance connects to the `pRemote` port; this allows the instance to use the interstellar network of the starship to communicate with the target transporter platform. The `TransportTargetManager` instance needs to communicate with the operator console (via the `pOC` port) and the

`TransportController` (via the `pTC`) port.

Figure 7.11 *Targeting Scanner interface objects*



7.3. Mechanistic Design

Mechanistic design optimizes the system at the scope of the use case collaboration; that is, it applies design patterns and techniques to optimize how a set of elements grouped by collaboration work together. The scope of mechanistic design decisions is generally an order of magnitude smaller than the scope of the decisions made in architectural design, since a system typically consists of one to several dozen use cases. Similarly to architectural design, mechanistic design largely proceeds via the application of design patterns, although the scope of the patterns is much smaller than the scope of those found in architectural design. This is where the classic "Gang of Four" (GoF) patterns[16] and other more fine-grained patterns are applied.

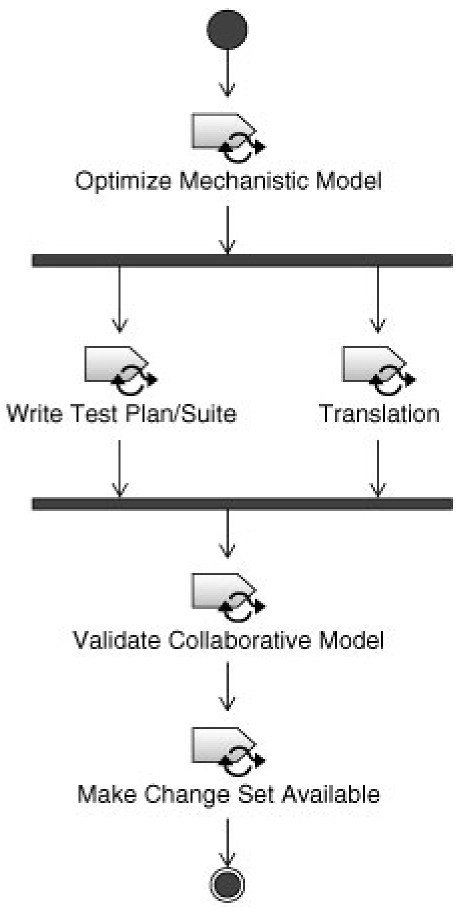
16. Gamma et al., *Design Patterns*.

The mechanistic design view is an elaboration of the object analysis view and uses the same graphical representation: Class and sequence diagrams show collaboration structure and sequence, activity, and state machine diagrams show behavior.

7.3.1. Mechanistic Design Workflow

The workflow for mechanistic design is straightforward, as can be seen in Figure 7.12. Most of the tasks are identical to their occurrence in object analysis, discussed in the previous chapter. The task that is fundamentally different is "Optimize mechanistic model."

Figure 7.12 *Mechanistic design workflow*


Optimize Mechanistic Model
Write Test Plan/Suite
Translation
Validate Collaborative Model
Make Change Set Available

7.3.2. Optimizing the Mechanistic Model

This task optimizes the collaborations from the object analysis model into mechanistic design models. For all its similarity to standard design practice, this task differs from more traditional design in a number of ways:

• In the Harmony/ESW process, the starting point for design is a functionally correct, executing analysis model.

• The Harmony/ESW process encourages designers to explicitly identify and rank the design criteria before any optimization is done.

• Mechanistic design is done explicitly at the collaboration level. This means that if the microcycle realizes four use cases, then this task will occur (at least) four times. These occurrences of the task may take place sequentially if a single small team is responsible for all the collaborations, but usually different teams optimize the collaborations in parallel.

The steps involved in mechanistic design should seem familiar by now:

1. Understand the functionality of the collaboration to be optimized.

2. Identify and rank the design criteria.

3. Select the design patterns.

4. Apply the design patterns.

5. Refine the scenarios.

It is important to keep in mind that the purpose of this level of design is to optimize the collective interaction of the elements of a collaboration. The architectural scope of design will optimize how collaborations interact with other collaborations in the general, strategic sense (focus on system-wide concerns). Detailed design (discussed in the next major section) focuses on optimization of elements in isolation (focus on internal element structure and behavior). This distinction drives the kinds of patterns that are of concern to mechanistic design.

Design patterns contain two kinds of elements. The first are the provided elements and relations; these constitute the “glue” that coordinates the elements of the pattern. When you instantiate the pattern, these “glue” classes are added into the collaboration. The second kind are the formal parameters of the pattern; these elements are subclassed or replaced by appropriate elements from your collaboration. This links the pattern into the collaboration, changing it from an analysis collaboration into a design collaboration.

For mechanistic design patterns, the standard reference is the GoF book. There are many other references on design patterns at this level of concern (a Google search on design patterns returns several million hits); this book is the standard.

The GoF book organizes mechanistic design patterns (although they don't use that designation) into three categories:

• *Creational patterns* generalize and abstract the process of creating collaborating sets of instances. The provided pattern classes encapsulate the responsibilities and techniques of instantiation as well as control the instantiation of the links among the created instances. The Abstract Factory and Prototype patterns are common patterns of this category.

• *Structural patterns* focus on how smaller elements are combined into larger-scale organizational structures. The Composite and Adapter patterns are very common in real-time and embedded systems.

• *Behavioral patterns* control how algorithms are allocated to different elements within the collaboration, how they are extended and specialized, and how they are orchestrated. The Chain of Responsibility, Command, and Observer patterns are all exceedingly common in modern designs.

7.3.3. Practical Example: Optimizing the Collaboration

For the transporter, let's apply a couple of design patterns to optimize the collaborations within the Targeting Scanner subsystem. In this case, let's focus on optimizing how targeting scanning works. We have a number of distinct optimization/design criteria to address:

• We need to construct larger assemblies from more primitive parts to manage the complexity of the subsystem. Note that we are *already* using the Composite Pattern (see Figure 6.32) architecturally for this purpose.

• We want to be able to use different scanning hardware from different vendors. Since this is a Federation project, we'll need to incorporate scanning hardware from humans, Klingons, and Vulcans, at least. We'll use the Adapter Pattern for this.

• We need to scan billions of small spatial regions, and efficient management of that many small objects will be a performance issue. We'll apply the Flyweight Pattern there.

• Finally, for scanning performance, we'll want to use the Quantum Telescope to move from area to area, and we'll want some object to manage all the scanners to work within the same area before the telescope acquires the next region (Mediator Pattern).

To begin, we'll use the Composite Pattern to group the various targeting scanners together. The `RemoteScanner` class (see Figure 7.13 on page 362) will also act as a mediator to sequence the operations. Figure 7.14 (on page 363) shows the updated collaboration where the `RemoteScanner` class now encapsulates the various target scanners.

Figure 7.13 `RemoteScanner` *composite*

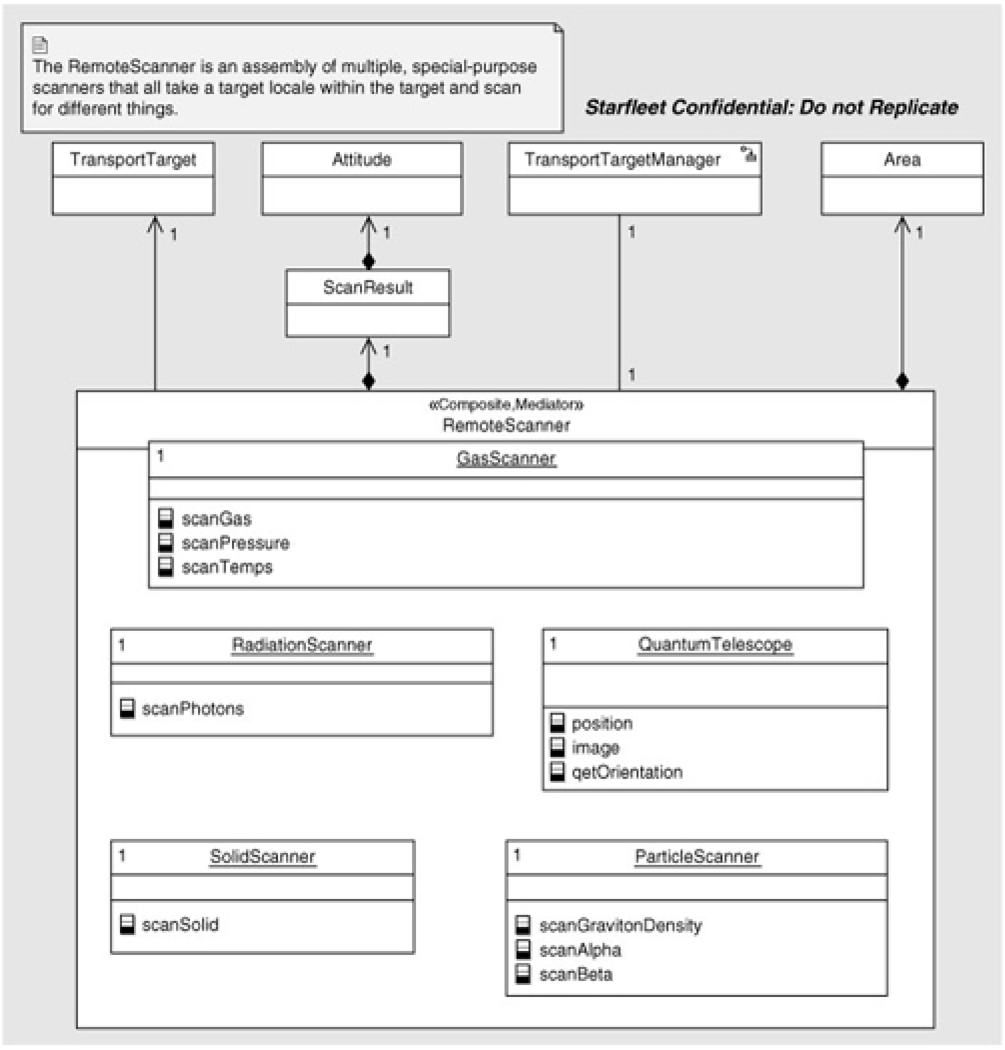
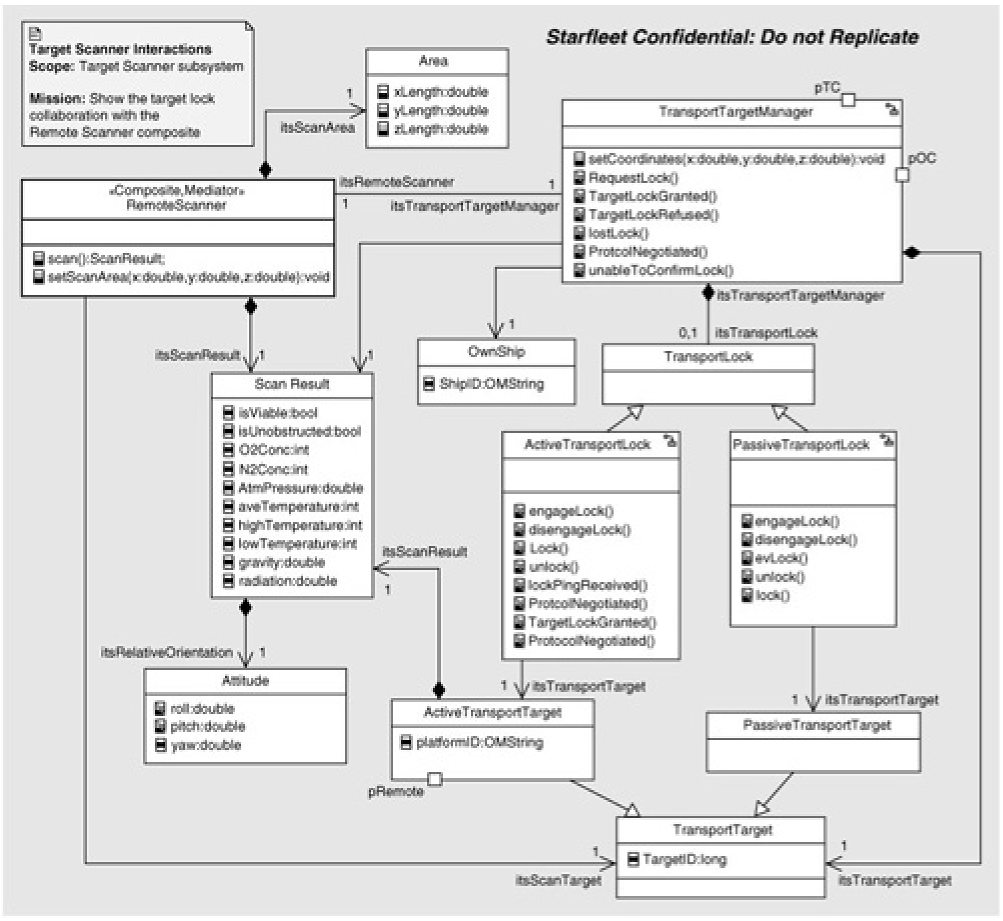
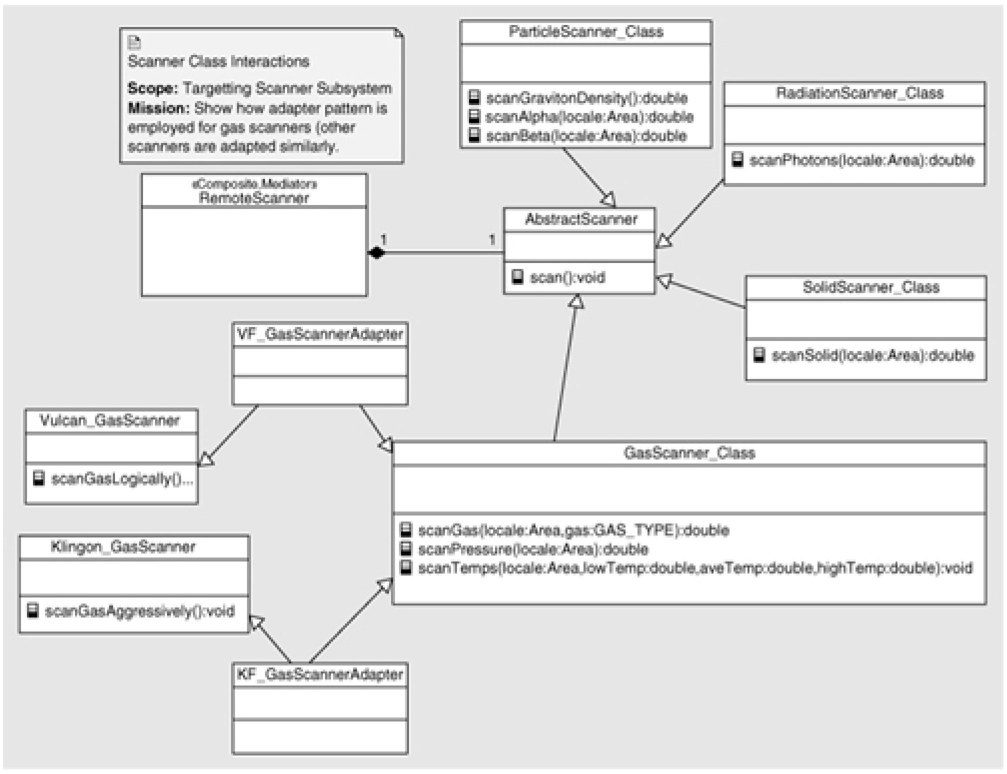


Figure 7.14 `TargetLock` *collaboration with composite*



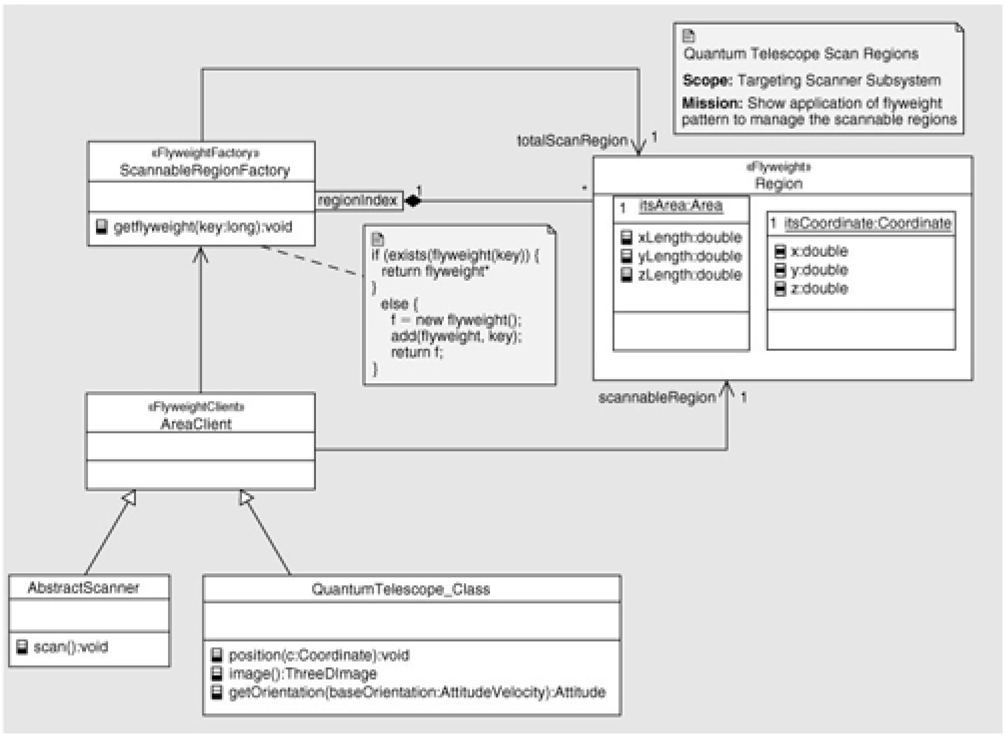
In Figure 7.15 (page 364), we add the Adapter Pattern to isolate the actual interface of the scanners from different vendors from the interface that we actually need to use. In this case the adapters implement the (inherited) required interface in terms of the (inherited) actual interface provided by the vendor.

Figure 7.15 `Scanner_Class` *interactions*


Scanner Class Interactions
Scope: Targetting Scanner Subsystem
Mission: Show how adapter pattern is employed for gas scanners (other scanners are adapted similarly.
«Composite,Mediator» RemoteScanner
ParticleScanner_Class
scanGravitonDensity():double
scanAlpha(locale:Area):double
scanBeta(locale:Area):double
RadiationScanner_Class
scanPhotons(locale:Area):double
AbstractScanner
scan():void
SolidScanner_Class
scanSolid(locale:Area):double
VF_GasScannerAdapter
Vulcan_GasScanner
scanGasLogically()...
GasScanner_Class
scanGas(locale:Area,gas:GAS_TYPE):double
scanPressure(locale:Area):double
scanTemps(locale:Area,lowTemp:double,aveTemp:double,highTemp:double):void
Klingon_GasScanner
scanGasAggressively():void
KF_GasScannerAdapter
1
1

Last, we want to use the Flyweight Pattern to manage the billions and billions of tiny scannable regions of the transport target area, as shown in Figure 7.16 on page 365.

Figure 7.16 `QuantumTelescope` *scan regions*


Quantum Telescope Scan Regions
Scope: Targeting Scanner Subsystem
Mission: Show application of flyweight pattern to manage the scannable regions
«FlyweightFactory» ScannableRegionFactory
getflyweight(key:long):void
regionIndex
totalScanRegion
1
«Flyweight» Region
1 itsArea:Area
xLength:double
yLength:double
zLength:double
1 itsCoordinate:Coordinate
x:double
y:double
z:double
if (exists(flyweight(key)) {
return flyweight*
}
else {
f = new flyweight();
add(flyweight, key);
return f;
}
«FlyweightClient» AreaClient
scannableRegion
1
AbstractScanner
scan():void
QuantumTelescope_Class
position(c:Coordinate):void
image():ThreeDImage
getOrientation(baseOrientation:AttitudeVelocity):Attitude

Mechanistic design optimizes the system at the level of the single collaboration. This allows for different collaborations to be optimized to meet more narrowly focused concerns than global architectural optimization.

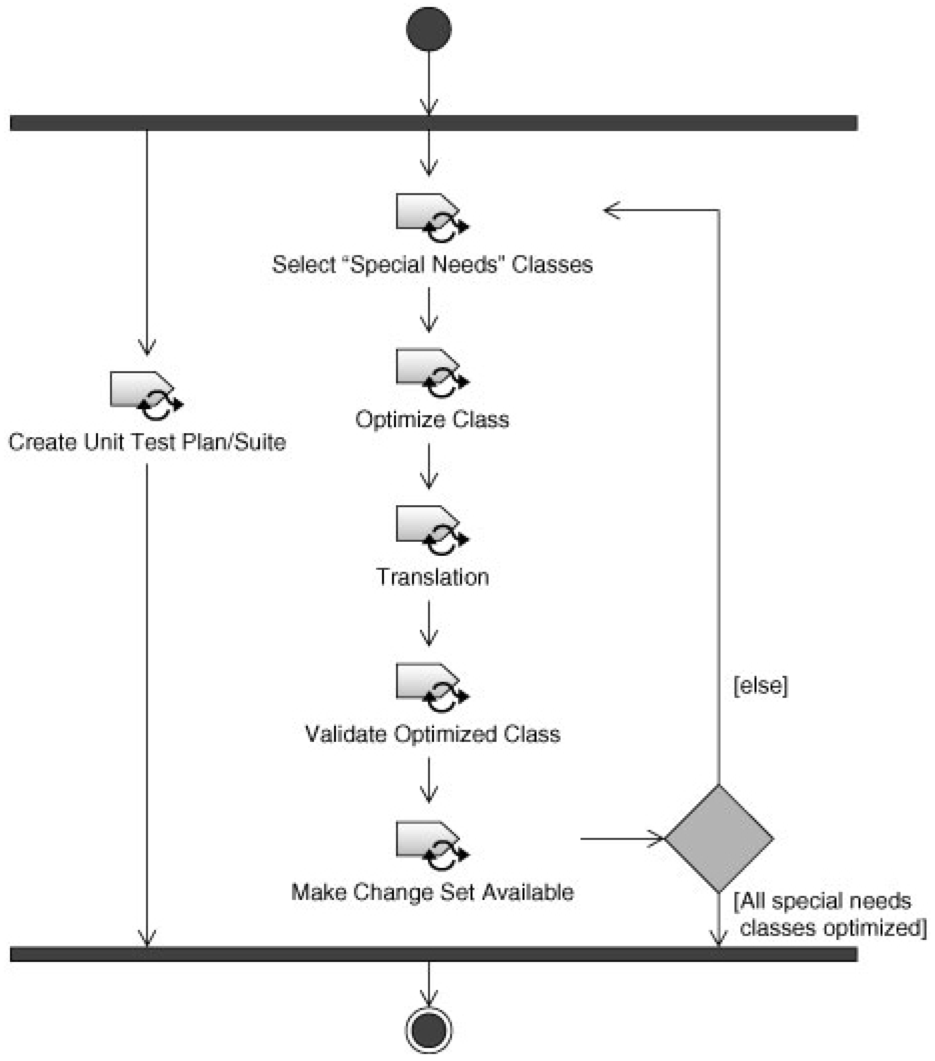
7.4. Detailed Design

Detailed design optimizes the system at the primitive element level; this is the level of a single function, or variable, or noncomposite class. The patterns of organization that appear on this level are commonly referred to as **idioms** instead of patterns, but their purpose and intent is the same as the larger-scale design patterns that we've seen before, that is, to optimize an aspect of the system. In some cases, these detailed design patterns take advantage of specific features of the implementation language (such as using `auto_ptr` from the C++ standard), but most are more or less source-language-independent and try to optimize some small aspect of the element at the expense of less important ones.

7.4.1. Detailed Design Workflow

Figure 7.17 on page 366 shows the workflow for the detailed design activity. Detailed design focuses most of its attention on the small set of primitive elements whose optimization can positively affect the overall system performance. These elements are optimized—again in accordance with the weighted set of design criteria constructed for that purpose—then debugged and formally tested. In parallel with these tasks, the unit test plans are updated to take into account any relevant changes due to the design elaboration. It is also important that QoS and performance testing be done here as well.

Figure 7.17 *Detailed design workflow*


Select "Special Needs" Classes
Create Unit Test Plan/Suite
Optimize Class
Translation
Validate Optimized Class
[else]
Make Change Set Available
[All special needs classes optimized]

7.4.2. Identifying "Special Needs" Classes

In my experience, most of detailed design is trivial; only a few elements (usually in the range of 3% to 5%, but this will vary based on the kind of system and its deployment environment) require special attention because they are highly complex in terms of data or behavior, have special safety or reliability requirements, contribute significantly to the overall system execution time or resource usage, or are part of a high-bandwidth data or control path that must be highly optimized. Detailed design spends most of its effort on the small set of elements that require such special attention. The first task selects those "special needs" classes.

The elements that need optimization can be identified by theory, inspection, or measurement. By "theory" I mean that we predict a priori that optimizing that element should positively impact system performance. By "inspection" I mean that we look at the model or source code and identify elements whose optimization we think will positively affect system performance. By "measurement" I mean that we run the system and *look* at where the system is spending most of its execution time or which elements are hogging the resources. Of the three methods, the last is by far the most useful. Experience has shown that engineers aren't very good at identifying good optimization points in system design, and performance analysis tools can add significant value by revealing the "truth on the ground."

7.4.3. Optimizing Classes

Since detailed design, just like other kinds of design, is about optimization, we have a similar set of steps for its performance:

1. Identify and rank the design criteria

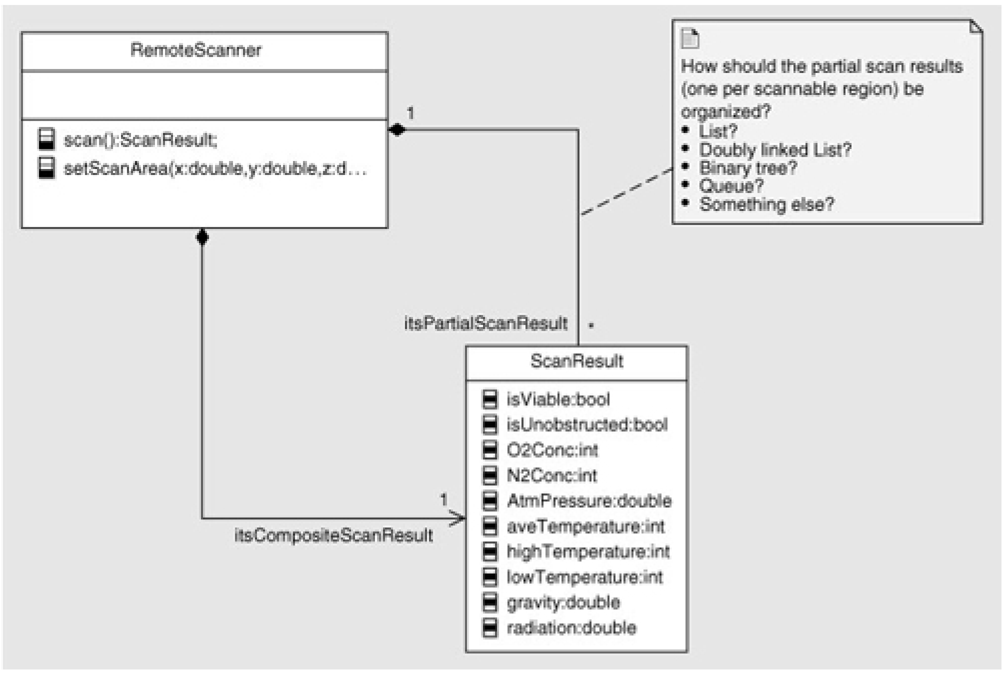
2. Select the design approach (pattern or idiom)

3. Apply the design pattern(s)

4. Refine the scenarios

There are many ways to optimize classes, functions, and attributes. These ways are conflicting because when we optimize one aspect we necessarily deoptimize others. I believe it is best to be explicit about what and why we are optimizing, since in my experience that leads to better designs. For example, consider the simple case of a one-to-many relationship as shown in Figure 7.18. This figure shows that the `RemoteScanner` associates via composition to the `ScanResult` class in two ways. One is the computed total overall result, but in addition to that, each of the billion or so scannable regions has its own partial result.

Figure 7.18 *One-to-many optimization*



What data structure should be used to organize these elements? It depends on precisely what you're trying to optimize. Some options include

• Read access time

• Write access time

• Space complexity (memory usage)

Of course, this varies with the conditions under which the data arrives and what we want to do with it. Is the most recent data the data we'll want to look at first? The oldest data first? Maybe it doesn't matter, but it might. Will we want to write occasionally but read it frequently? The reverse?

Let's look at some alternatives and their performance characteristics

• Linked list

A linked list is a linear data structure. In terms of space complexity, it adds one additional pointer to each data structure, pointing to the next element. You may also want to add a scannable region ID (perhaps of type long) to identify to which region the data refers. Usually, the client of the data begins with a pointer at the first data structure and to find a particular piece of data (in this case, perhaps a scannable region in which the radiation is above a threshold value or with an obstruction), each element must be searched from the start of the list. Thus, it has read access time $O(n)$, where n is the number of elements in the list. With respect to write (insert) performance, if you always insert at the end, the client can keep an end-of-list pointer, in which case write performance is $O(k)$, where k is a constant. For write (delete) performance, the performance is O(n), because the data element must be found before it can be deleted.

• Sorted linked list

A sorted linked list is one that is maintained in a particular order, based on a search key with possibly other secondary search keys. A sorted linked list has better search characteristics than a linked list, although its read access time is also $O(n)$. However, if the list is maintained in sorted order against the search criteria, then you can stop the search once you've reached past the point in this list where the data might appear. For example, the partial scan result list could be ordered by O_2 concentration. Once the value gets below 21%, you can stop. However, what if you want to search on N_2 concentration instead? Then the list is no better than an unsorted list. Or you can maintain multiple lists, sorted by different keys, but that impacts space complexity. As for write access, the search time is $O(n)$, because the list must be searched to find where in the list the new item must be inserted or deleted.

• Array

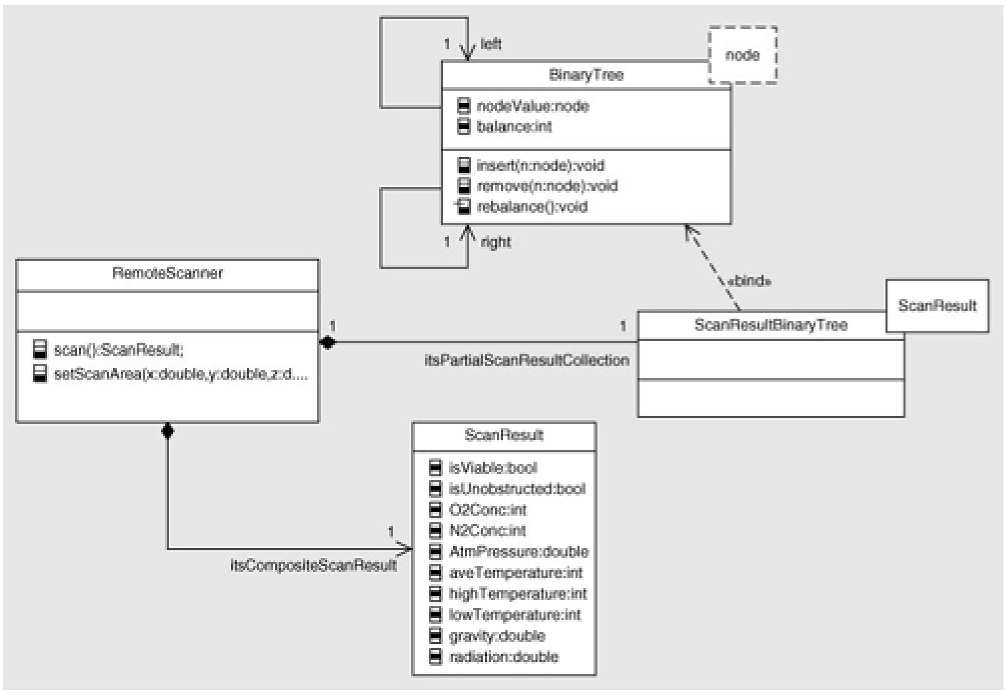
A simple indexed array can also work. It has better space complexity than a linked list because the linkage is implicit (via the index). However, maintaining it in sorted order can be expensive in terms of performance, because the entire block of remaining elements must be moved to make room for a new element in the middle of the list. As long as it is maintained in unsorted order, the time complexity is $O(n)$.

• Balanced binary tree

Binary trees store elements with two additional pointers—a left and a right pointer—which add to the space complexity. Binary trees have $O(\log_n())$ search properties—vastly better for large values of n than lists—provided the tree is balanced. A balanced binary tree is one in which the length of any branch from a node is the same as the other, within ± one node. Balancing requires that each data element add a balance attribute, which can be –1, 0, or +1, upping the space complexity. When a node is inserted, the tree must be reorganized (referred to as left-left, left-right, or right-left rotations) to maintain the balance. This adds a fixed cost to the update time whenever nodes are inserted or deleted. The cost is not incurred when the tree is searched, however. Binary trees have great search performance, but at the cost of higher space complexity and a greater update time fixed cost.

You can see that there are many possibilities, only a few of which are listed here. If the organization contributes significantly to the region-scanning feature of the software, then it makes sense to identify and rank the design criteria so that an optimal design solution can be used. Let's say that memory size isn't an issue, but we have tight time constraints. In this case, let's select a balanced binary tree design solution (see Figure 7.19).

Figure 7.19 *Binary tree for* `ScanResult`



Other concerns for detailed design optimization include:

• Algorithmic optimization

• State machine behavioral optimization

• Explicit statements of pre- and postconditions

• QoS and performance (performance budgets) specification and unit testing

• Defensive design

° Range checking

° Data validity checking

° Unit checking (e.g., kilometers versus miles versus parsecs)

• Association realization

• Class feature visibility

• Internal error and exception handling

• Redundancy-in-the-small for safety and reliability

Defensive development and design were discussed in Chapter 3, "Harmony/ESW Principles and Practices." While there may (or may not) be some defensive aspects of the classes, functions, and variables when identified in the analysis model, there certainly must be those aspects here in detailed design. Low-level redundancy is used to ensure that data corruption can be identified. This topic was also explained in Chapter 3 in the discussion of safety concepts.

State machine design patterns were introduced in one of my earlier books.[17] A state machine design pattern relates to a state machine in the same way that a mechanistic design pattern relates to a collaboration. The state machine design patterns provide a way to structure and organize the state machine to provide some optimization benefits, albeit at some cost. Examples of usage of those state machine design patterns can be seen in another of my books.[18] These patterns are typically applied during detailed design, although they are sometimes applied earlier.

17. *Doing Hard Time.*

18. *Real-Time UML Workshop for Embedded Systems.*

7.5. Coming Up

This chapter focused on the role and implementation of design in the Harmony/ESW process. While analysis establishes the functional correctness of the software (including executable models and code), design is all about optimizing the system. This takes place at three levels of abstraction: architectural, mechanistic, and detailed.

Architectural design focuses on the strategic optimizations that affect most or all of the system and how the system elements collaborate with others on a system-wide scale. Architectural design is split into five primary views. The subsystem and component view identifies and characterizes the large-scale structural elements of the system, including their responsibilities and interfaces. The concurrency and resource view identifies the units of concurrency, how and when they are scheduled, how they synchronize with other concurrency units, and how they share resources. The distribution view identifies the strategies for how elements in, or potentially in, different address spaces find each other and collaborate. The safety and reliability view concentrates on the identification, isolation, and correction of faults at runtime. Finally, the deployment view allocates responsibilities among different engineering disciplines such as software, electronics, and mechanical components. There may be some secondary concerns as well, such as information assurance (security), data management, and so on that have strategic impact on the system architecture.

Mechanistic design works at a much smaller scale than architectural design. It centers on optimizing collections of elements that realize a use case. This is done largely through the application of design patterns at the same level of abstraction as the “Gang of Four” patterns. The goal is to achieve the optimizations required for that use case while allowing the realizations of other use cases to be optimized differently as needed.

Detailed design works at an even smaller scale than mechanistic design. This level optimizes individual elements: classes, functions, data structures, and types. Most of these primitive elements are very simple in a good object-oriented structure, but there is usually a set of elements that require special optimization because of their usage context or because their responsibilities are complex in some way.

All levels of design abstraction use a similar approach for optimization:

1. Identify the design optimization criteria.

2. Rank the criteria in order of criticality of the optimization.

3. Identify design solutions, patterns, and technologies that optimize the most important of these criteria at the expense of those of lesser importance.

4. Apply the solutions to the model.

5. Validate that the solutions don’t break the existing functionality and also achieve the desired optimization.

The next chapter talks explicitly about testing in the context of an agile model-based development project. We’ve been discussing unit testing in this and the previous chapter, so Chapter 8 will focus on integration and validation testing.

Following that, the last chapter will discuss how agile projects can be tracked, controlled, and improved. This chapter will include discussions on managing change, what goes on during the “party phase” (aka increment review), and how to perform model-based reviews.

Chapter 8
Agile Testing

I remember having a discussion some years ago with my boss, the head of advanced systems development for a company developing embedded system products. I had spent a *couple of years* convincing him that design was, in principle, a good idea. So one day, he came up to me in the hallway and said, “OK, I’m convinced. Thinking about what you’re going to do before doing it is a good idea. But then, since you’ve thought about it, you don’t need to test it.”

Sigh.

The kicker of that story is that the projects we were designing were cardiac pacemakers for implantation in humans.

But the question—had it been phrased as one—is a good one. Why *do* we test software? One of the premises throughout this book is that the best way not to have defects in your software is not to introduce them into the software. A corollary premise is that you can’t effectively test quality into low-quality software. So why do we test?

We test because we screw up. Try as we may, and with the best of intentions of developing great defect-free systems, sometimes we make mistakes, such as

- Implementing requirements that are incorrect, inconsistent, or inadequate
- Misinterpreting requirements
- Writing down the wrong requirements
- Not understanding the context of usage of the system (preconditional invariants)
- Forgetting about conditions that might arise
- Not accounting for faults in hardware or software elsewhere on which we depend, including
  - Originally correct hardware that fails
  - Included hardware or software that contains previously unknown faults

° Runtime errors that occur because of third-party software that is used or included

° Compiler, linker, or OS errors that result in incorrect code or behavior

• Incompletely or incorrectly understanding an algorithm

• Making an algorithmic or data structure mistake

• Not taking into account violations of preconditional invariants at runtime

This book focuses on agile development for real-time and embedded systems. What does this mean with respect to testing? What is "agile testing"?

Agile testing is:

testing the right things, at the right time, to the right level of detail, in the most efficient manner, to prove a software system works and works correctly.[1]

1. Greg Fournier, *Essential Testing: A Use Case Driven Approach* (Charleston, SC: Book-Surge Publishing, 2007). Greg uses this description for what he calls "essential testing," which has, as he describes, an "agile core." See also Elisabeth Hendrickson, "Agility for Testers," Pacific Northwest Software Quality Conference, 2004, available at http://testobsessed.com/wordpress/wp-content/uploads/2007/01/aftpnsqc2004.pdf.

Well, almost, anyway. I assert that testing doesn't *prove* a system is correct but proves that the system behaves as expected for the specified set of conditions and test cases. Proving correctness is theoretically possible but practically impossible for any but the most trivial systems. What we can do, though, is specify the set of test cases and show that the system works for those. We try to select test cases to ensure good coverage of likely and potential conditions that the system will face, but it is simply not possible to test all possible control flows with all possible data sets.

For most software systems, testing is limited to functional testing. This ensures that the required functionality the system is intended to provide is rendered by the system (at least for the cases explicitly tested). For real-time and embedded systems, functional testing is clearly necessary but also clearly insufficient. Real-time systems are characterized by their need to perform within QoS (aka "nonfunctional") constraints. These include memory limitations, worst-case timeliness, average timeliness, reliability, and so forth. One can certainly make the argument (and I have!) that desktop and IT systems should undergo such QoS testing as well, but it is clearly less important in those cases than in a typical real-time and embedded system.

This chapter will introduce you to the various kinds of tests that can be applied to a system and the different levels at which they may be applied (i.e., unit, integration, and validation). After you have that understanding under your belt, I'll get to the heart of the matter: test-driven development (TDD) for model-based designs of real-time and embedded systems. Key in that discussion is the UML Testing Profile, since it defines a standard metamodel supported by different tools. Finally, I'll talk about how testing fits smoothly into the Harmony/ESW process.

8.1. Testing Concepts

My very favorite testing book is *The Art of Software Testing,* first published in 1979 and revised in 2004.[2] The book starts out with a simple self-assessment test: Write the test cases for the following program:

2. Glenford J. Meyers, *The Art of Software Testing, Second Edition,* revised and updated by Tom Badgett and Todd M. Thomas with Corey Sandler (New York: John Wiley & Sons, 2004).

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

The interesting thing is how poorly even experienced software engineers do on this, despite the program's triviality.[3]

3. See the book for a list of the error conditions that have actually occurred in different versions of the program.

It is problematic for software developers that it is impossible to fully test all but the most trivial programs. The number of logic paths in a nontrivial software application is finite but huge. Couple that with different combinations of data values and you have an essentially infinite set of possible test cases. Nevertheless, we test because we want to uncover errors that crept into the logic prior to the release of the system to the customer, or situations that weren't covered explicitly by the requirements but should have been.

To effectively create test plans and test cases, it is important to understand both the different kinds of tests that can be created and applied and the different levels at which testing can be done. TDD is a philosophy of defining tests early, sometimes even preceding the development work, in an effort to ensure that the development work is defect-free. Let's talk about this concern in more detail.

8.1.1. Kinds of Test Cases

Different kinds of tests uncover different kinds of errors. Back in Chapter 3 I introduced the following kinds of tests:

• *Functional*—tests the behavior or functionality of a system or system element

• *QoS*—tests the "performance" of a system or system element, often to measure the performance of the system or element against its performance requirements. Such tests measure the time for an individual action or for a large number of actions in sequence or in parallel. For example, if a device driver is planned to handle 1000 messages/second, then stress testing might consist of testing sustained throughputs of 900 messages/second and 1000 messages/second as well as measuring the time for the processing of a single message.

• *Precondition tests*—tests that the behavior of the system or system element is correct in the case that the preconditional invariants are met *and* in the case that the preconditional invariants are violated

• *Range*—tests values within a data range

• *Statistical*—tests values within a range by selecting them stochastically from a PDF

• *Boundary*—test values just at the edges of, just inside, and just outside a data range

• *Coverage*—tests that all execution paths are executed during a test suite. This can be done at the model level (looking for execution of all transitions in a state machine or all control flows in an activity diagram), at the source code level (looking for all calls, if/then/else, switch/case, and try/catch clauses), or at the assembly language level.

• *Stress*—tests data that exceeds the expected bandwidth of a system or system element. Stress testing tries to break the system or element by giving it a bandwidth range out of its design specification to ensure that the system can gracefully handle overload situations.

• *Volume*, also known as "load testing"—tests the system with large amounts of data that meet or exceed its design load

• *Fault seeding*—tests whether the system properly handles a fault intentionally introduced to the system

• *Regression tests*—normally a subset of previously passed tests; tests that a modification to a system did not introduce errors into previously correctly functioning systems

Coverage testing is a particularly interesting one, especially from the standpoint of safety-critical systems. The civil avionics software standard DO-178B[4] has 66 objectives; the hardest of these to achieve are the coverage objectives. The standard specifies the requirements for five levels of safety criticality, from level E (no impact) to level A (can lead to the loss of the aircraft), as shown in Table 8.1.

Table 8.1 *DO-178B Test Coverage*

Level	Coverage	Explanation
Level A	MC/DC	Level B + 100% modified condition decision coverage
Level B	DC	Level C + 100% decision coverage
Level C	SC	Level D + 100% statement (or line) coverage
Level D		100% requirements coverage
Level E		No coverage requirements

4. Officially known as "Software Considerations in Airborne Systems and Equipment Certification RTCA/DO-178B" (RTCA, Inc., 1992). Note that DO-178C will be released soon and will be more cognizant of advances in software development such as modeling and object-oriented development. Right now, the DO-178B standard doesn't prohibit these advances, but it does make it an extra effort to use them. This is being developed by the SC-205 group within RTCA, of which subgroup SG4-Model-Based Design and Verification is the largest subgroup with about 45 members. Interested readers can go to http://forum.pr.erau.edu/SCAS/.

The coverage requirements differ depending on the safety level:

• No coverage requirements means exactly what it says; for Level E certification, it is not required to demonstrate that all requirements have been tested.

• 100% requirements coverage means that it is necessary to demonstrate that the system is tested against each requirement.

• 100% statement coverage (SC) means that every software statement is covered at least once.

• 100% decision coverage (DC) means that every decision branch in the program has been taken at least once; this means that the same statements may be covered multiple times because there may be more than one path leading to the execution of a software statement.

• Modified condition decision coverage (MC/DC) means not only that the result of the combination of conditions has been tested, but also that all conditions themselves have been independently tested.

For example, consider the statement

```
If (A || B || C) then
        DoStuff();
Else
        DontDoStuff();
```

With statement coverage, all four statements must be executed once. With decision coverage, both the TRUE and FALSE results of the combined decision `(A || B || C)` must be taken. With MC/DC coverage, all independent variations of `A`, `B`, and `C` being TRUE and FALSE must be evaluated, such as:

• `A` is TRUE, `B` is FALSE, and `C` is FALSE

• `A` is FALSE, `B` is FALSE, and `C` is FALSE

• `A` is TRUE, `B` is TRUE, and `C` is FALSE

And so on.

In practice, MC/DC is extremely expensive to achieve.

8.1.2. Levels of Testing

It is also common to identify three kinds of test scope:

• *Unit test*—"white-box" testing of some internal element (class, class collaboration, function, or data element)

• *Integration test*—"gray-box" testing that brings together different architectural-scope software units (components or subsystems typically) as well as possibly elements from engineering disciplines (e.g., electronic, mechanical, chemical, or optical engineering) to ensure that the large-scale components work together properly

• *Validation*—"black-box" testing in which the internal elements of the system are hidden from view and the resulting system (including software running on the target hardware platform) is tested

In the Harmony/ESW process, unit testing is done in parallel with modeling and code generation. It is not a task that is done ex post facto but is an integral part of developing the software.[5] Integration testing is done at least daily because the developers update the baseline (following their unit testing), and the configuration manager performs these tests to ensure that the baseline always works before updating the baseline for the team. Validation testing is done every microcycle against the new requirements added in the current microcycle, and a significant subset of test cases from previous microcycles (known as regression testing) is run as well.

5. I am reminded of a project that I was associated with years ago that performed unit testing *after the product was shipped* to the customers. The resulting unit test identified serious errors and forced the manufacturer to informally recall the devices. I think this is inexcusable, but the prevailing attitude was that unit testing was performed only to “check the box” rather than to actually aid the product quality.

Each of these test scopes contains at least most of the preceding kinds of tests. In fact, I recommend writing test plans that include a placeholder for each kind of test; if there are no test cases within the test suite for that kind of test, then an explicit statement must be made justifying its lack.

8.1.3. Test-Driven Development

An important agile concept is **test-driven development** (TDD). This is a principle that is realized by a number of practices within the Harmony/ESW process. The essence of the principle is that testing is so important that it should not be added as an afterthought but should be incorporated as a part of the analysis or design specification of each element. One of the practices of TDD is that the test cases are specified prior to (or at least simultaneously with) the element(s) to be tested. So in object analysis, before (or at the same time) I add a class, function, or variable, I also add test cases that will ensure that the feature is correctly designed and implemented.

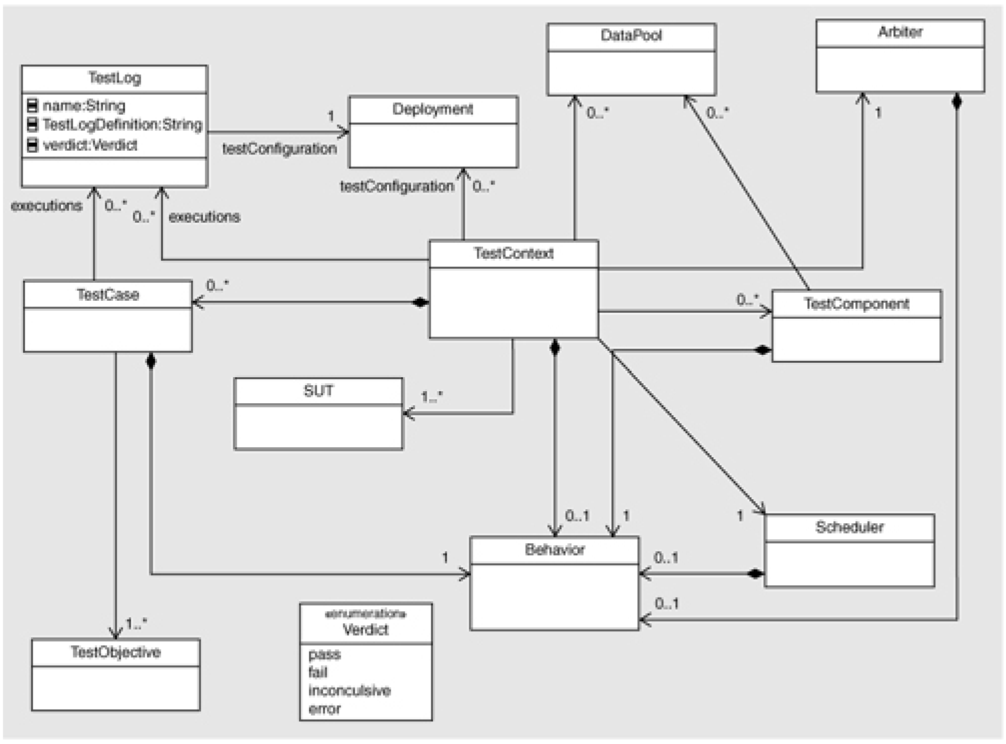
Another practice that directly supports TDD is the continuous execution of these tests throughout the development work. Rather than waiting until the collaboration, subsystem, or system is completely designed and implemented and *then* testing, you apply test cases as an integral part of the development of those elements. This is vastly more effective than the traditional test-at-the-end approach.

8.2. Model-Based Testing

Model-based testing refers to the testing of models, or testing in which the test cases, test environments, or test results that are represented within models. Just as models provide significant advantages over text-based development of requirements and design, so do they provide better ways to think about, create, organize, and execute tests. One way to manually implement model-based testing is through the use of the «testBuddy» class, as discussed in Chapter 3, "Harmony/ESW Principles and Practices." «testBuddy» classes are classes that are added into the model to test specific elements or groups of elements within the model. They are a part of the model and are configuration-managed with those elements that they test, but they are not a part of the shipped, production software.

Going beyond manual approaches, the OMG has a standard called the UML Testing Profile.[6] This profile defines a metamodel of testing concepts and then adds lightweight extensions to the UML (stereotypes, tags, and constraints) to realize this metamodel as a (slightly) customized version of UML. Figure 8.1 shows the primary concepts and their relations in the metamodel.

Figure 8.1 *UML Testing Profile metamodel*



6. UML Testing Profile formal/05-07-07 (OMG, 2007), available at http://www.omg.org/.

Interested readers are encouraged to refer to the standard for more detail, but a few metaclasses stick out. The `SUT` is the system under test; this is the executable artifact to which test cases are being applied. A `TestCase` is a sequence of stimuli and expected behavior and responses. Test cases are owned by the `TestContext` and applied by it to the `SUT`. The `TestContext` also owns the data used by the test cases. The `TestComponent` includes various elements that interact with the `SUT` to realize the test cases defined by the `TestContext`. The `Arbiter` determines the appropriate verdict for the test execution that is put into the `TestLog`. The `TestObjective` has no behavior but provides the rationale for the test case. The profile defines the term *test architecture* to include the `Arbiter`, `Scheduler`, `SUT`, `TestComponent`, and `TestContext`.

The value that the profile brings to the table is that with these standard definitions, it is possible to build standard tools to model the test cases and the testing process. This allows us to realize the same benefits for testing in particular that models bring to software development in general. In this context, I will talk about a specific tool, Rhapsody TestConductor, but because the UML Testing Profile is a standard, other tools can be used to realize the profile as well. In general, I want to apply model-based testing so that I can

• Test my models using the same syntax and semantic environment used to create the models

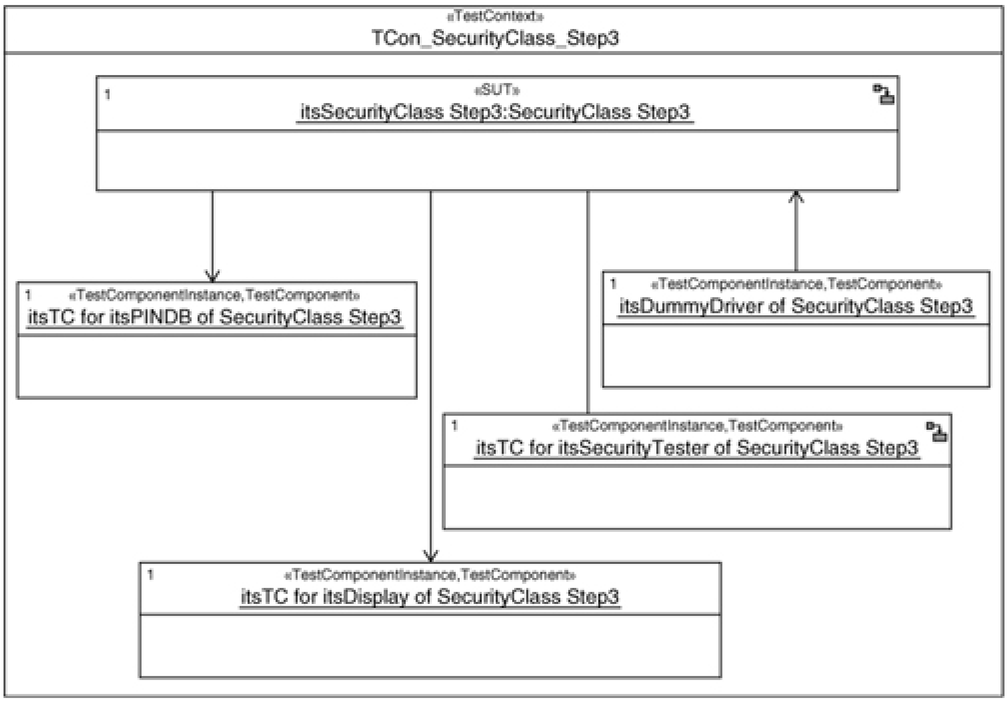
• Improve my ability to get comprehensive, correct test cases into my test suites

• Save time creating and executing test cases

• Automate steps wherever possible, such as creating the test architecture, executing a set of test cases, and evaluating the test results to determine the verdict for each test case

TestConductor (and the profile) allows different realizations of the test cases. The most common way to realize a test case is with a sequence diagram, but also common are activity diagrams and source code. By way of illustration, let's use the `Security` model from Chapter 3. Specifically, TestConductor can automatically create a test architecture for the `SecurityClass_Step3` class from Figure 3.14. In fact, it automagically created the diagram shown in Figure 8.2. The only class not created by TestConductor is the `SUT`, the `SecurityClass_Step3`. The `TestContext` and test components (stubs) are created for the classes that are associated with the `SUT`, the `SecurityTester`, `Display`, and `PINDB` classes. Each testing component subclasses the actual corresponding class from the model and thus supports the same interfaces but also partially realizes the test cases.

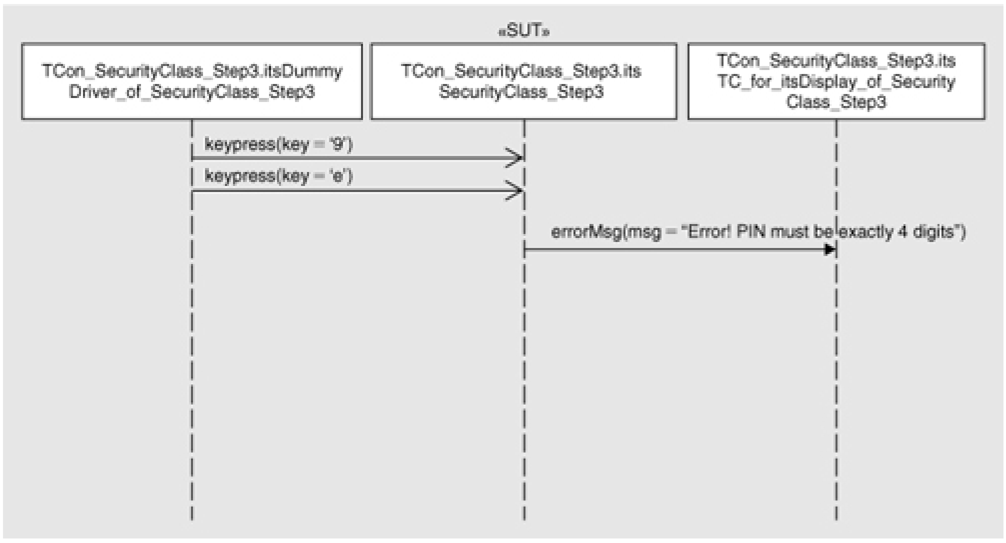
Figure 8.2 *Test architecture for* `Security` *class*



«TestContext»
TCon_SecurityClass_Step3
1
«SUT»
itsSecurityClass Step3:SecurityClass Step3
1 «TestComponentInstance,TestComponent»
itsTC for itsPINDB of SecurityClass Step3
1 «TestComponentInstance,TestComponent»
itsDummyDriver of SecurityClass Step3
1 «TestComponentInstance,TestComponent»
itsTC for itsSecurityTester of SecurityClass Step3
1 «TestComponentInstance,TestComponent»
itsTC for itsDisplay of SecurityClass Step3

Of course, while the test architecture provides the infrastructure necessary to execute the test cases, we still must provide the test cases themselves. Let's do two different test cases, one using a sequence diagram and one using an activity diagram. For the sequence diagram test case, I'll use the Automatic Test Generator (ATG) feature of Rhapsody to generate the sequence diagrams for me, although in this simple case, I could easily have done this myself. ATG, however, will generate all the test cases (eight in this case) to ensure that all the behavior of the SUT is exercised. One of these is shown in Figure 8.3.

Figure 8.3 *Sequence diagram test case*

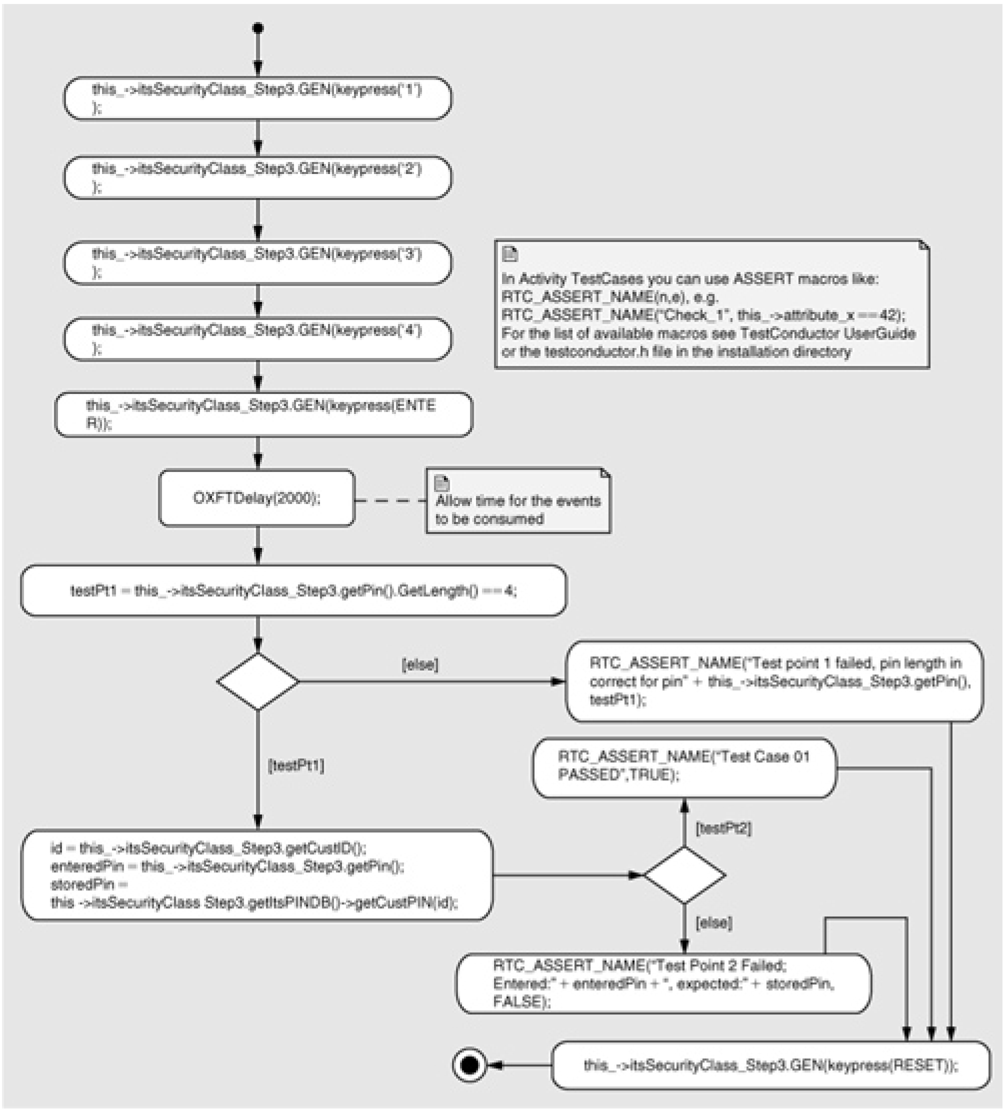


«SUT»
TCon_SecurityClass_Step3.itsDummy Driver_of_SecurityClass_Step3
TCon_SecurityClass_Step3.its SecurityClass_Step3
TCon_SecurityClass_Step3.its TC_for_itsDisplay_of_Security Class_Step3
keypress(key = '9')
keypress(key = 'e')
errorMsg(msg = "Error! PIN must be exactly 4 digits")

One of the biggest advantages of model-based testing is that I can view the test cases in the very same syntax and level of abstraction as the elements I want to test. This allows me to bring all the power of modeling to bear on the difficult and tedious task of creating, applying, and analyzing the results of test cases.

A sequence diagram usually represents a single test case.[7] Activity diagrams commonly represent multiple test cases, using branches, forks, and joins to depict the different flows. Figure 8.4 shows a set of test cases, the different flows depicted as different branches from the branch operator.

Figure 8.4 *Activity diagram test case*



```
this_->itsSecurityClass_Step3.GEN(keypress('1')
);
```

```
this_->itsSecurityClass_Step3.GEN(keypress('2')
);
```

```
this_->itsSecurityClass_Step3.GEN(keypress('3')
);
```

```
this_->itsSecurityClass_Step3.GEN(keypress('4')
);
```

```
this_->itsSecurityClass_Step3.GEN(keypress(ENTE
R));
```

```
OXFTDelay(2000);
```

```
testPt1 = this_->itsSecurityClass_Step3.getPin().GetLength() == 4;
```

```
RTC_ASSERT_NAME("Test point 1 failed, pin length in
correct for pin" + this_->itsSecurityClass_Step3.getPin(),
testPt1);
```

```
id = this_->itsSecurityClass_Step3.getCustID();
enteredPin = this_->itsSecurityClass_Step3.getPin();
storedPin =
this ->itsSecurityClass Step3.getItsPINDB()->getCustPIN(id);
```

```
RTC_ASSERT_NAME("Test Case 01
PASSED",TRUE);
```

```
RTC_ASSERT_NAME("Test Point 2 Failed;
Entered:" + enteredPin + ", expected:" + storedPin,
FALSE);
```

```
this_->itsSecurityClass_Step3.GEN(keypress(RESET));
```

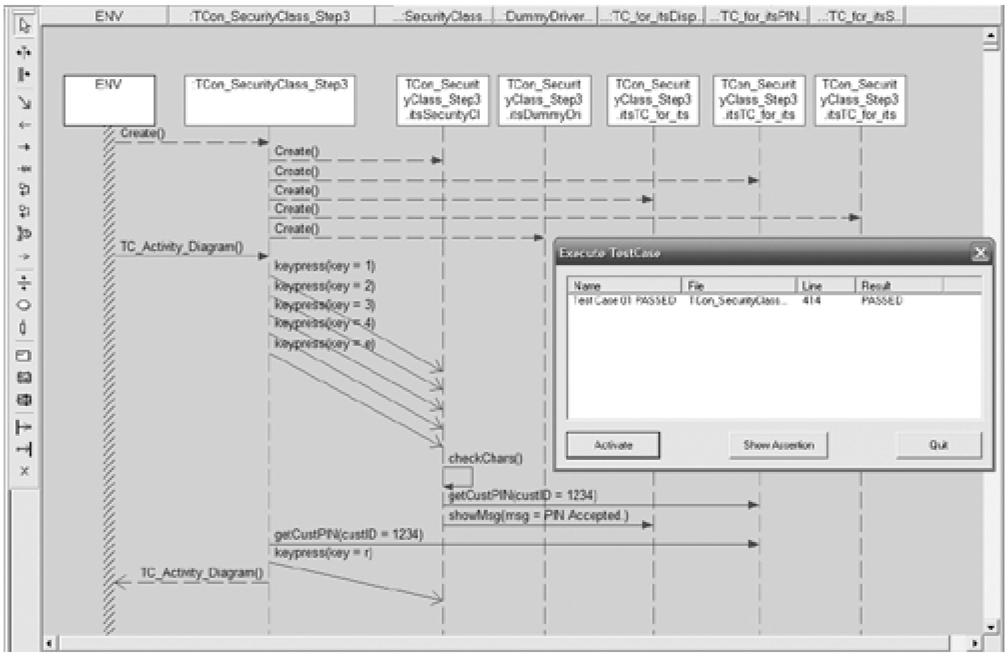
In Activity TestCases you can use ASSERT macros like:
RTC_ASSERT_NAME(n,e), e.g.
RTC_ASSERT_NAME("Check_1", this_->attribute_x == 42);
For the list of available macros see TestConductor UserGuide
or the testconductor.h file in the installation directory

Allow time for the events
to be consumed

7. Although with the sequence diagram operators (*opt*, *alt*, *para*, etc.) it is possible to represent multiple flows within a single sequence diagram.

Figure 8.5 shows the result of the execution of one of the flows from Figure 8.4, including the sequence diagram capturing the test execution. This is a valuable view because the sequence diagram shows the messages exchanged among the elements (including the SUT), and the dialog box in the foreground shows the verdict of the test. If one of the test points had failed, the verdict would have been FALSE and the failed test point would have been identified.

Figure 8.5 *Activity diagram test execution result*

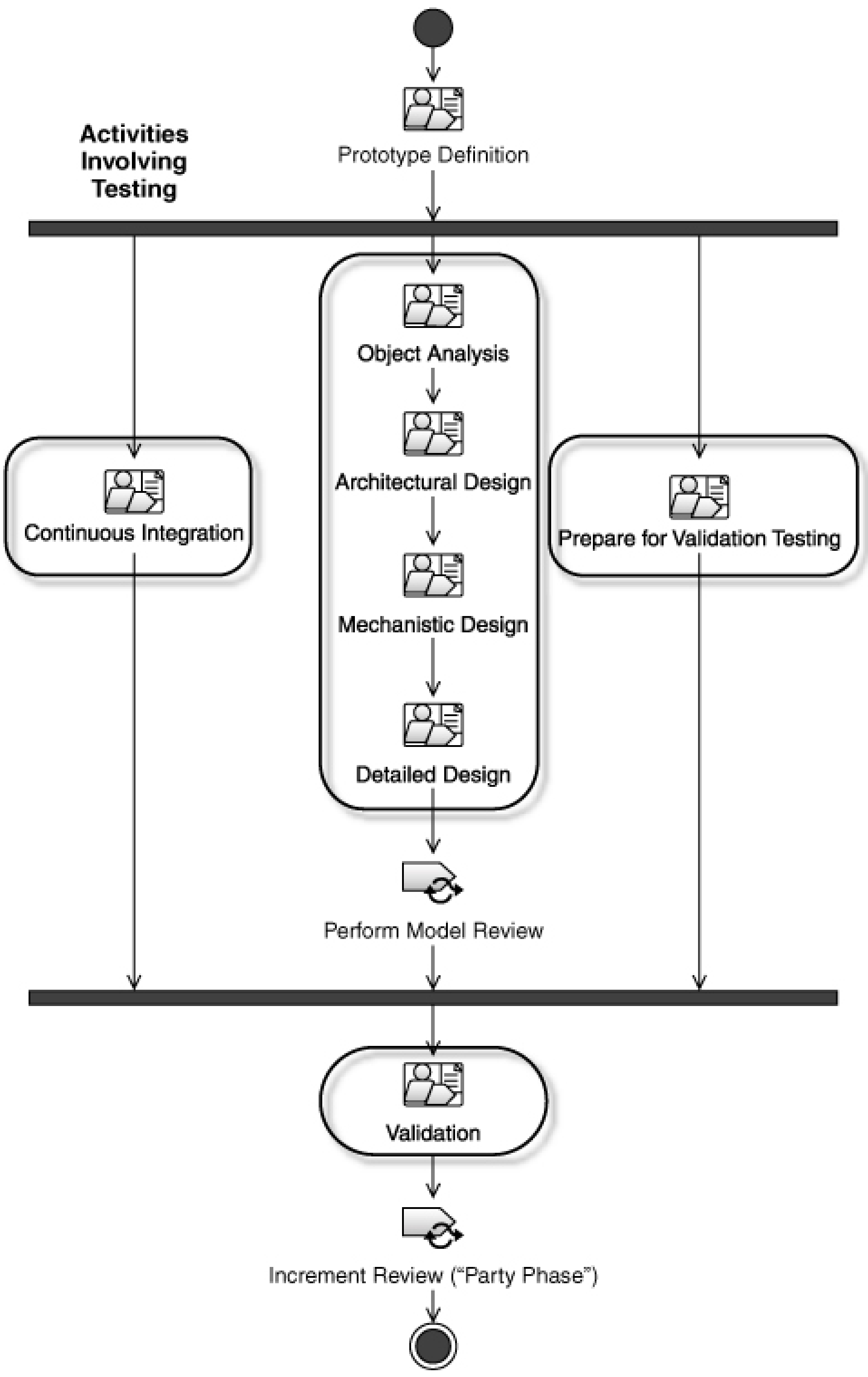


Name	File	Line	Result
Test Case 01 PASSED	TCon_SecurityClass...	414	PASSED

8.3. Testing Workflows

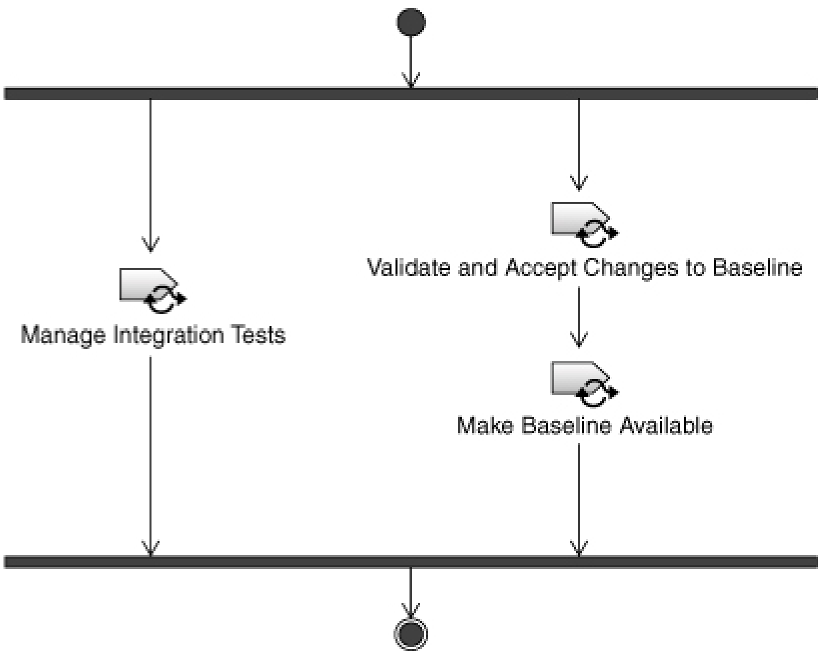
Formal testing appears at several points in the microcycle (see Figure 8.6). Unit testing appears in the object analysis, architectural design, mechanistic design, and detailed design phases. There are two relevant tasks within those phases: “Create the test plan/suite” and either “Execute unit test” (object analysis), “Validate architecture” (architectural design), “Validate collaborative model” (mechanistic design), or “Validate optimized class” (detailed design). These are all more or less equivalent; they all apply unit-level tests in the context of their containing phase.

Figure 8.6 *Testing in the microcycle*



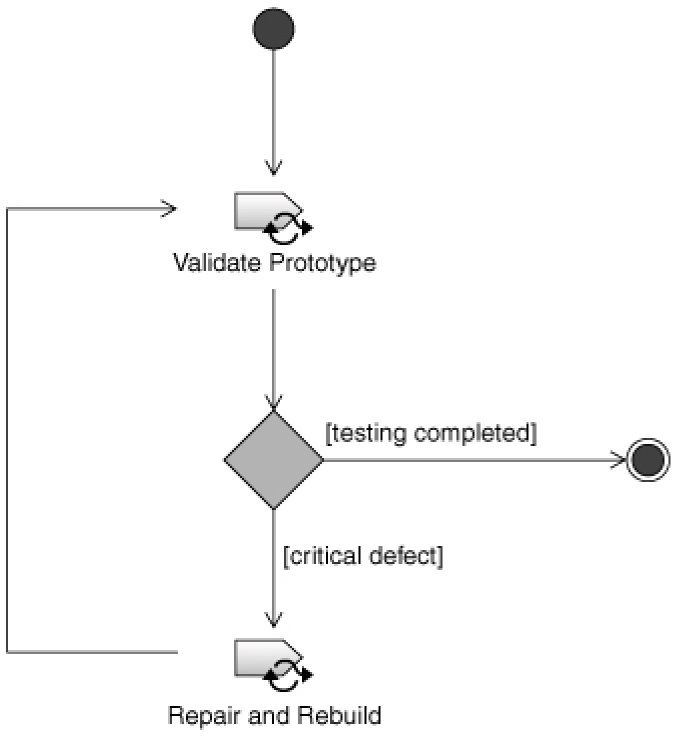
Integration executes in parallel with the development activities in the microcycle (see Figure 8.7). Within the integration activity the task "Manage integration tests" creates and updates integration tests that are applied in the "Validate and accept changes to baseline" task.

Figure 8.7 *Continuous integration workflow*



The last occurrence of testing in the microcycle is validation testing. The development of the test cases and test context is done in the microcycle activity "Prepare for validation testing" in Figure 8.6. The basic workflow for validation testing itself at the end of the microcycle is shown in Figure 8.8. All the testing work is done in the "Validate prototype" task. Note that in validation testing, critical defects are fixed before the project progresses, but minor defects are identified (via change requests) and scheduled for downstream repair.

Figure 8.8 *Validation workflow*



Now that we understand where testing occurs in the workflows, let’s discuss the three levels of testing themselves.

8.4. Unit Test

As mentioned previously, unit testing is applied within multiple phases of the microcycle and is broken up into tasks for both planning the unit test and executing that plan. Unit tests are inherently white-box and applied to all models that produce source code (and to manually generated source code).

Unit-testing tasks appear in the workflows of object analysis, architectural design, mechanistic design, and detailed design—anytime the software analysis or design is being modeled and code is being generated. For example, if you look at Figure 6.28 back in the chapter on analysis, you see the tasks “Create unit test plan/suite” and “Execute unit test.” Because unit testing is tightly integrated into modeling and code generation, it doesn’t have a separate workflow except in those contexts.

8.4.1. Unit Test Planning

The planning for unit testing is done in the “Create unit test plan/suite” task. The purpose of unit test planning is to identify the scope of testing, what will be tested, and the set of test cases necessary to achieve this task. It should be noted that because unit testing is so integral to the different activities related to generating code, the test metadata (plans, test cases, and test fixtures) are also created incrementally. As a feature or aspect is added to a set of elements, tests are added *at the same time*. This is significantly different from the way unit test plans are usually created. Most commonly, the entire test plan is defined at once (often long after the code has been written). In an incremental and agile process, it is important that the testing be likewise incremental and agile.

This task has the following steps:

1. Understand the expected behavior of the model or package under development.

2. Create the test plan organization.

3. Capture relevant test cases and data in the test plan.

4. Perform coverage analysis to ensure that test case coverage is adequate.

5. Specify the tester's steps required for test execution.

6. Examine the test plan for completeness, accuracy, consistency, and correctness.

The scope of a unit test plan is usually a set of elements related by having a common author and working closely together within a context, such as within a package or a use case collaboration. The purpose of the unit test plan is to provide test cases, taken from the kinds of tests discussed earlier in the chapter, that when executed ensure that the elements do the right thing and do it correctly. In object analysis, the focus is primarily on functionality. The initial tests usually focus exclusively on computing the correct outputs based on the inputs and states of the relevant elements. Within the scope of object analysis, it is also appropriate to add robustness tests such as precondition, range, boundary, coverage, and statistical data tests. Later in design, the scope of the tests evolves to include various aspects of performance and qualities of service, including stress, volume, and fault-seeding tests, as well as functionality tests for elements added in the design phase.

A typical test plan includes the following sections:

• Scope and objectives

• Schedule

• Responsibilities

• Regression tests

• Test case libraries and standards

• Tools and test fixtures

• Test environment(s)

• Completion criteria

• For each feature or use case:

° Feature/use case description

° Risk level

° Criticality

° Test case

• Steps

• Data

• Completion criteria

The main purpose of the test plan is to ensure the adequacy of the test suite and to identify when, where, who, and how. The unit test plan need not be completely written before unit testing is begun. Instead, it is best developed incrementally along with the test cases and the SUT. In addition, if the same person is writing the test plan as well as executing it (often the case), then some of the information can be abbreviated. However, care should be taken that for future maintenance, enough information is provided that the unit test suite can be successfully repeated.

8.4.2. Unit Test Execution

The test plan is executed in the unit test tasks mentioned previously. They all consist of essentially the same steps:

1. Review the work items completed.

2. Select the test cases.

3. Set up the test environment.

4. Create the build.

5. Execute the test cases against the build.

6. Gather the results.

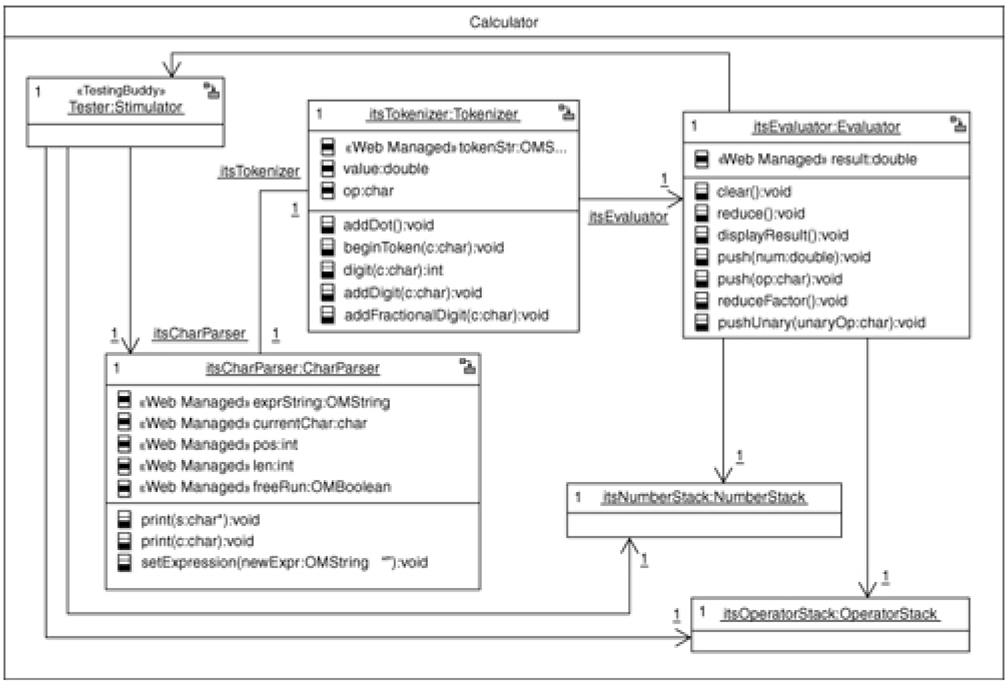
7. Analyze and communicate the test results.

The important thing about unit testing is that it is executed incrementally and repetitively as the analysis and design models are created. It is not, as is traditional, performed at the end of the implementation phase.

One of the approaches to testing is to create “friends” (in the C++ sense) of the classes to be tested. Generically, these are called **test buddies.** Test buddies are custom testing elements that can be added or removed from the constructed unit and exist solely to support unit testing of those elements. For the most part, the test buddies I create are classes that are friends of the classes under test; this allows access to the private and protected members of the class and facilitates testing. «testBuddy» classes usually have one-way associations to the classes that they test so that the classes being tested have no knowledge or dependency on the testing fixtures.

One of the reasons I like «testBuddy» classes is that it is easy to create them in parallel with the classes under test. They are usually stored within the same package (or a nested package) as the classes that they test. In addition, when I check out the package for work, I automatically have access to the test set. Figure 8.9 shows an example that I often use when teaching people how to effectively use state machines.[8] The model takes a string of characters, such as 2.3 × (1.01 - 6.2/7), the `CharParser` class parses it by identifying the characters (digits, dots, operators, and whitespace), then the `Tokenize` class builds up the tokens (i.e., constructs an operator or number token with the parsed value), and then the `Evaluator` class applies the operator-precedence rules of arithmetic. The `NumberStack` and `OperatorStack` are realizations of a template `Stack`, specialized for the type of element stored (double for the `NumberStack` and char for the `OperatorStack`).

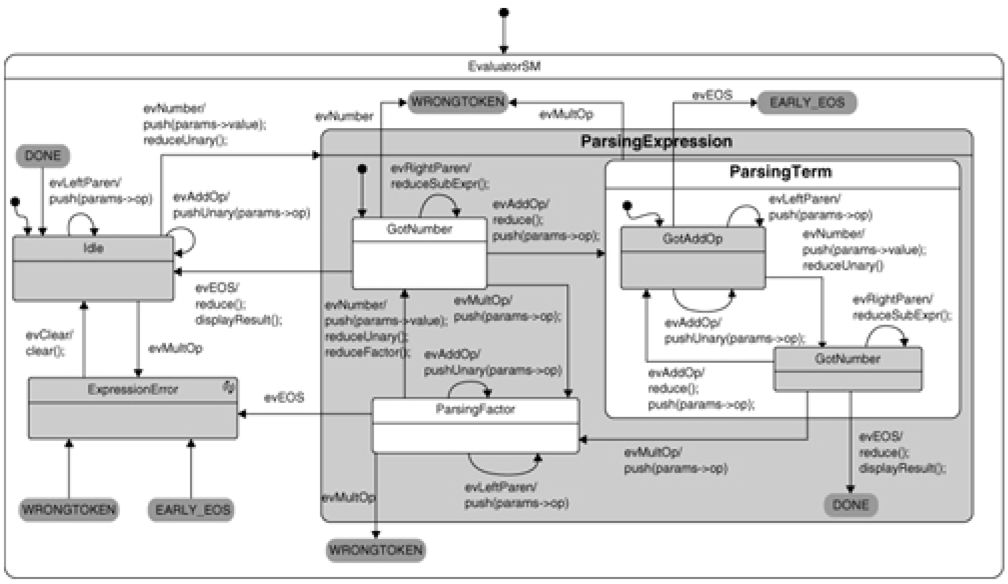
Figure 8.9 `Calculator` *example*



8. A complete description of the example is available in my book *Real-Time UML, Third Edition*.

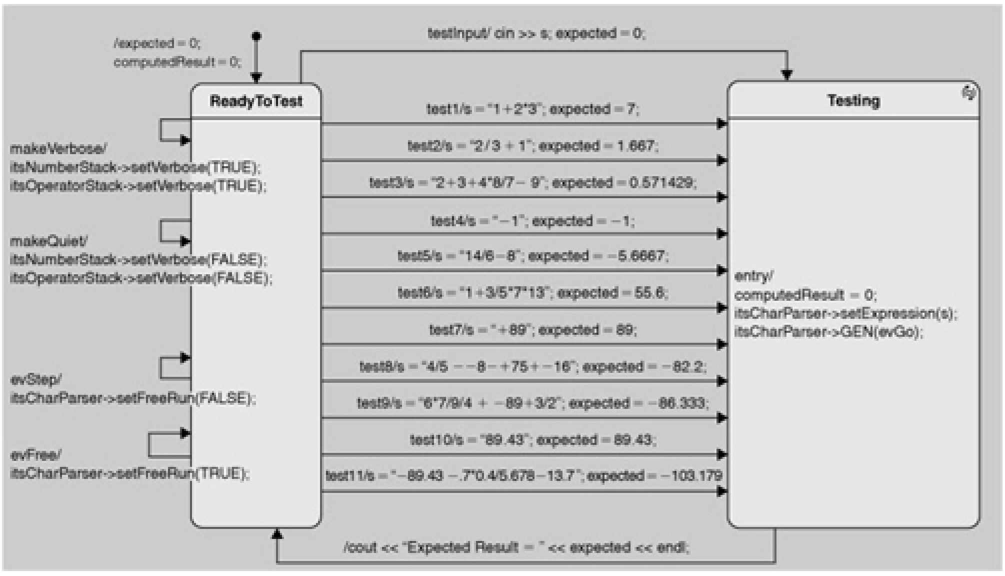
The state machines can be rather complex, as you can see in the `Evaluator` state machine shown in Figure 8.10. This state machine not only performs the computations for the evaluation of the tokenized expression but also determines when it is expecting different kinds of tokens. For example, in the `Idle` state, the machine expects the next token to be a number, an additive operator (+ or – , to be treated as a leading sign), or a left parenthesis. If a multiplicative operator (* or /) is seen instead, this is clearly an error, and the state machine transitions to the error state. In the state `GotNumber`, the `Evaluator` is now expecting an operator of some kind; another number would be a syntactic error.

Figure 8.10 `Evaluator` *state machine*



There are many paths in this state machine, let alone in combination with the state machines for the other classes. To assist, the model includes a `Stimulator` class (stereotyped `«TestingBuddy»`) to provide test cases (see Figure 8.11). To run a test, you simply run the generated application, then send the appropriate event (`test1`, `test2`, etc.) to the `Stimulator`.

Figure 8.11 `Stimulator` *state machine*


/expected = 0;
computedResult = 0;
testInput/ cin >> s; expected = 0;
ReadyToTest
Testing
makeVerbose/
itsNumberStack->setVerbose(TRUE);
itsOperatorStack->setVerbose(TRUE);
makeQuiet/
itsNumberStack->setVerbose(FALSE);
itsOperatorStack->setVerbose(FALSE);
evStep/
itsCharParser->setFreeRun(FALSE);
evFree/
itsCharParser->setFreeRun(TRUE);
test1/s = "1+2*3"; expected = 7;
test2/s = "2 / 3 + 1"; expected = 1.667;
test3/s = "2+3+4*8/7− 9"; expected = 0.571429;
test4/s = "−1"; expected = −1;
test5/s = "14/6−8"; expected = −5.6667;
test6/s = "1+3/5*7*13"; expected = 55.6;
test7/s = "+89"; expected = 89;
test8/s = "4/5 − −8− +75+ −16"; expected = −82.2;
test9/s = "6*7/9/4 + −89+3/2"; expected = −86.333;
test10/s = "89.43"; expected = 89.43;
test11/s = "−89.43 −.7*0.4/5.678−13.7"; expected = −103.179
entry/
computedResult = 0;
itsCharParser->setExpression(s);
itsCharParser->GEN(evGo);
/cout << "Expected Result = " << expected << endl;

The `Stimulator` state machine, just like the classes under test, didn't emerge fully formed. It started with a single test case, and more were added over time. Once the basic functionality worked (initial tests just parsed whole numbers, fractional numbers, and white space), these were removed since the more advanced tests included parsing of these aspects.

8.5. Integration Test

The purpose of the integration test is to ensure that the units (modules, subsystems, components, etc.) work together properly. Integration testing is "gray-box" in that most of the internal details of the units are hidden, but the overall architecture is exposed to the integrator. In the Harmony/ESW process, integration tests are performed by the configuration manager, but in some project environments, the integrator may be a different role.

The tasks involved in integration testing include performing the integration per se, managing the integration test suite, validating the submitted changes in the context of the baseline, and making the updated baseline available. The workflow for these tasks is shown in Figure 8.7.

8.5.1. Continuous Integration

Integration is frequently a thorn in the side of systems developed traditionally. It is not uncommon for an integration team for a larger-scale embedded system to spend several months trying to get the subsystems to work together properly. Harmony/ESW continuous integration takes another tack. If the subsystems are integrated daily, then as soon as an incompatibility is introduced, it is discovered and fixed. It's true that early prototypes don't do much, but in my experience it is easy to remove a defect from a system that was demonstrably correct yesterday and has undergone a minor change today. In fact, it is far easier, more reliable, and lower-cost to do that every day of the project than it is to take a set of complete subsystems that have never been integrated before and get them to work together.

Figure 8.7 shows the workflow for integration. Within the microcycle, all the development activities contain a task known as "Make change set available" following successful execution of the unit tests.[9] This task releases the unit-tested model and code to the configuration manager for integration into the evolving software baseline and is done at least daily. The configuration manager collects the set of developer submissions (known as "change sets") and creates a testable build, which he or she will evaluate prior to accepting the baseline changes and republishing the baseline.

9. That is, we never try to integrate code that hasn't been successfully unit-tested. Tell your friends.

In addition to the execution of the integration tests, the configuration manager must also update the tests as the system becomes more and more complete. The integration test is sometimes known as a "smoke test" and is meant to ensure that the new changes don't break the baseline in any obvious way. It is common to use a subset of the validation and system regression tests for this purpose. This means that as functionality is added to the baseline, the integration tests must be updated to include some measures of correctness for the new functionality.

8.5.2. Managing Integration Tests

The "Manage integration tests" task adds new tests to the change set. The steps for this task include:

1. Add new tests from a change set.

As a change set comes in, the developer must indicate what new functionality he or she is providing so that the appropriate integration tests can be created.

2. Add new tests from a previous validation.

Integration tests for a microcycle typically include a subset of the validation tests from previous spirals. This provides a check that the new changes don't break functionality that was known to be correct in previous microcycles.

3. Modify tests as interfaces and functionality change.

This often requires the configuration manager to coordinate with multiple developers so that related functionality shows up at about the same time.

4. Add new tests as defects are identified.

Defects that are missed by the integration tests but found later may indicate inadequacy of the integration test suite and result in additional tests.

This task requires good communication between the configuration manager and the developers. As mentioned above, these tests are added “just in time,” so that the tests always reflect the expectations of the current baseline plus the day’s change sets. This agile integration-testing approach removes the need for long delays while the test suites and documentation are created and aligns the integration with the software development.

8.5.3. Validating and Accepting Changes to the Baseline

The “Validate and accept changes to baseline” task creates the build from the developer’s collective change sets and evaluates them with the integration tests. The steps for this task are:

1. Update the workspace.

The configuration manager updates his or her workspace to include the submitted developer change sets. This may require spinning off a new configuration for the purpose of integration.

2. Validate the configuration.

Any conflicts must be resolved.

3. Create the build.

The details of this step vary depending on modeling tool, source language, and development environment. The purpose of this step is to build a version of the system that incorporates the previous functionality along with the new developer changes. This includes code generation, compilation, and linking of the compiled elements.

4. Run the integration tests.

The integration tests are run and the results must be compared to expectations. Automation is preferred when available and cost-effective. If defects or discrepancies are identified, the offending change sets are rejected and the developer must correct them as soon as possible. In some environments this can be done informally, while in others this is done by submitting change requests through the change management system, such as Change from IBM Rational.

5. Create the baseline.

Once the integration tests complete successfully, the developer changes are accepted. The configuration manager then updates the baseline with the changes.

This task is typically done daily. Because the developers are using a common definition of the interfaces and the developers' work must pass unit tests prior to the changes being submitted to the configuration manager, this task usually proceeds smoothly. If it does not, the root cause is usually that integration isn't happening often enough, the developers are not using a common definition for the interfaces, or the submitted software quality is too low. In the latter case, an improvement in the unit testing is the recommended cure.

8.5.4. Making the Baseline Available

The "Make baseline available" task makes the baseline available to the project developers. The baseline includes not only the source code, but the models as well. In an MDD environment, the model represents the analysis or design of the system and is the most important element in the baseline. Source code can be derived from the model and is therefore recommended to be included, but it is not essential (except for manually written source code, which must be included in the baseline). In addition, any unit test documentation and test cases should be included in the evolving baseline.

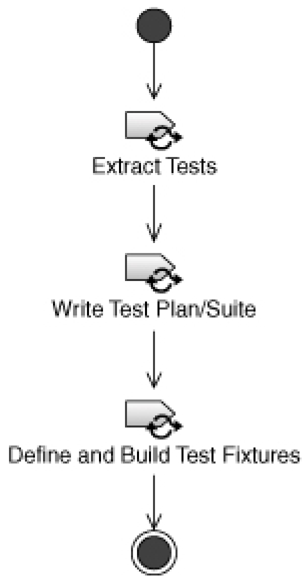
8.6. Validation Testing

The purpose of validation testing is to test the constructed system against the requirements. In the Harmony/ESW microcycle context, validation testing is done against the new requirements realized within the current microcycle plus the set of requirements realized in previous microcycles. Validation tests are created incrementally in a fashion similar to unit and integration tests. The two relevant activities from Figure 8.6 are "Prepare for validation testing" and *"Validation."*

8.6.1. Preparing for Validation

As you can see in Figure 8.6, this activity begins after the prototype definition activity and executes in parallel with the development activities. It consists of three tasks (see Figure 8.12). The purpose of this activity is to create the test cases for validation of the current microcycle prototype and create/update the testing environment necessary for that.

Figure 8.12 *Prepare for validation*



Extracting Tests

The "Extract tests" task creates the test cases to be applied at the end of the microcycle, as well as possibly to be handed off to the configuration manager for addition into the integration tests. This task starts after the use cases are detailed within the prototype definition activity. Most of the functional tests come from the use case scenarios, but the validation tester usually adds other tests as well.

The steps involved in this task are as follows:

1. Review the specification document and model.

2. Extract the functional flows.

3. Determine the data sets for the test cases.

4. Determine the test preconditions.

5. Specify the expected results.

The use case scenarios, as mentioned, form the basis of most of the functional tests, but they are often not specified fully enough to support repeatable testing. The use case scenarios and state machines must be augmented with precise descriptions of preconditions and use specific data values for the tests. It is common for a single use case scenario to result in a dozen or more tests with different data values and slightly different preconditions.

The scenarios provide functional flows in an obvious way. The scenarios may not represent all functional flows, however. This is especially true for error handling and exceptional (rainy-day) cases. The normative use case state machine should contain these exceptional flows, however, and these should be represented as test cases as well. At a minimum, every transition and action in the use case state machine should be represented in at least one test case. Besides the use-case-derived test cases, the tester may add other kinds of tests—especially performance- and safety-related tests. For validation testing, not all kinds of tests may be represented; some, such as coverage testing, are fundamentally white-box and are therefore inappropriate for validation testing.[10]

10. It should be noted that validation testing is *mostly* black-box but may contain fault seeding for reliability and safety testing, and such tests are white-box. Fault seeding is the intentional introduction of a fault into the running system to ensure that the corrective measures correctly identify and handle the fault. I've been known to rip out chips, boards, and cables of a running system in the testing lab for this purpose.

Writing the Test Plan/Suite

The validation test plan is not significantly different from the other test plans, except that because the tests are usually performed by people other than the developers, more documentation is needed. The same basic organization can be used for the validation test plan as for the unit test plan. The test plan provides a means by which the adequacy of the test suite can be assessed and describes the methods, techniques, and tools used to execute the tests. Of particular interest are the steps the tester must take to set up and actually perform the test.

Defining and Building the Test Fixtures

Because the validation tests are black-box, they often require different testing tools from unit testing. It is often necessary to create simulators for various input and output devices (actors) and to orchestrate their behavior to execute the test cases. Test fixtures are special-purpose devices or applications that facilitate the testing process. They fall into two broad categories:

• A test fixture may simulate a system or application with which the system must interact. The simulation allows for faults and special difficult-to-replicate event sequences to be created.

• A test fixture may provide additional monitoring of the system for the determination of the success or failure of the test case execution.

Test fixtures can be either purchased or constructed. Purchased test fixtures include devices such as in-circuit emulators, ROM emulators,[11] oscilloscopes, and JTAG[12]-compliant devices.

11. I have fond memories of my Orion Instruments ROM emulator for the Z80 and 6502 . . . back when we had only 0s and 1s—and some days we didn't even have 1s! Oh, those were the days! ☺

12. Joint Test Action Group, the common name for IEEE 1149.1-1990, "Standard Test Access Port and Boundary-Scan Architecture." For details, see http://standards.ieee.org/reading/ieee/std_public/description/testtech/1149.1-1990_desc.html.

8.6.2. Validation

The validation activity applies the validation test plan to the prototype at the end of the microcycle and results in two work products. The first is a validated prototype and supporting model/code baseline. This is a version of the system that supports a subset of the requirements (except for the last prototype, of course) but contains the real code that will ship to the customer. This validated prototype has no detected critical defects. The other work product is a list of minor defects to be fixed in downstream microcycles, most commonly captured in change requests or defect reports.

Validating the Prototype

The execution of the validation test suite applies the test plan to the completed system. As mentioned, this is done at the end of every microcycle, typically every four to six weeks. The execution requires the environment for each test to be set up with the appropriate test fixtures and preconditions, the test case steps to be performed by the tester (or test fixture automation), and the gathering and analysis of the test results. The steps for this task are:

1. Review the work items completed in the system for the prototype.

2. Create the build from the baseline.

3. Select the test cases.

4. For each test case:

a. Set up the test environment.

b. Execute the test case.

c. Gather the results.

d. Analyze the results.

5. Communicate the test results.

6. Issue change requests as necessary.

Repairing and Rebuilding

When critical defects are identified, they must be identified and isolated as much as possible and passed back to the appropriate developers. This is often done by issuing a change request with a "critical" status. The change control board (CCB) allocates the critical request to one or more responsible parties for repair. The steps involved in this task are:

1. Analyze the change request (defect).

2. Allocate the change request to the appropriate developers.

3. The developers fix the defect.

4. The developers repeat the unit tests for the changed units.

5. The developers submit the change set to the configuration manager.

6. The configuration manager repeats the integration test and build.

7. The configuration manager passes the built prototype back to the validation team for retest.

Minor defects do not go through the "Repair and rebuild" task. Instead, they are logged and added to the work items list for downstream microcycles.

8.7. Coming Up

This chapter showed how testing is an essential aspect of the Harmony/ESW process. Unit-level testing takes place continuously during software development in the object analysis, architectural design, mechanistic design, and detailed design activities. It is a key part of the nanocycle whose steps include:

1. Design the element/define the test.

2. Update the collaboration.

3. Debug.

4. Test.

5. Make changes available to the baseline.

Integration testing also happens continuously. As unit-tested elements are submitted to the build manager (aka the configuration manager), integration tests are applied to ensure that the new elements don't "break the build." As more features are added to the evolving baseline, the integration tests must be elaborated as well.

Validation testing occurs periodically; in fact, it takes place at the end of every microcycle. Validation testing ensures that the constructed build meets the requirements introduced in this spiral and also ensures that previously validated functionality hasn't been broken in the current microcycle.

The last chapter of this book focuses on process optimization within the Harmony/ESW process. This involves tracking "truth on the ground," comparing the actual situation with the plan, identifying what actions are appropriate to bring the plans into line with that truth, and executing those actions. Appropriate actions may be simply adjusting the plans, but ideally the actions include changing how the work is being done to improve our ability to effectively produce the high-quality systems and software that are our charter.

Chapter 9
Agile Process Optimization

Not long ago I read a news report about the Vermont state patrol arresting a teenager for speeding at 104 mph. The teenager explained that she hadn't noticed that police officers were trying to pull her over because *she was busy talking on her cell phone at the time*.[1] It struck me that most software projects are a lot like that. Project managers and project teams engage in risky behavior that they're not even actually aware of because they're not paying attention. And then they are surprised that they don't end up where and when they had originally planned.

1. I swear I don't make this stuff up! www.daveanddarren.com/dumbass.htm, June 19, 2008.

This chapter addresses that issue (not the talking on your cell phone while speeding issue; the other one). I have repeatedly stressed the absolute essentialness of dynamic planning. We simply don't have all the information at the start of a project, and the information at the start of the project isn't known with infinite fidelity. Beyond that, things that you know now can change, and sometimes they were never true to begin with. For all these reasons, it is necessary to adjust plans to reflect the "facts on the ground."

A key aspect of agile processes is the notion of dynamic planning and using information gathered during current work to update the plans for future work. In this chapter, I'll discuss what dynamic planning means and how to track and control projects. This will involve:

• Tracking project progress

• Updating plans

• Refining the development environment

• Managing things that are changing or must change

• Reviewing work products (especially models, since this is a model-driven approach)

• Performing periodic project assessments

9.1. Understanding Dynamic Planning

Dynamic planning simply means that if we understand that plans are by their very nature made with incomplete knowledge, then we must, as a part of that planning, plan to update the plans. Or, more simply, we must “plan to replan.” This is in stark contrast to the far more common “ballistic planning” approach in which infinite depth and fidelity of knowledge are assumed.

Taking changes into account while dynamically planning has a number of steps:

1. Identify the deviation from what you planned or expected and recognize what the circumstance actually is

2. Analyze the impact of this deviation on your plans

3. Determine if there are more optimal ways to perform the work

4. Adjust your plans to minimize the impact of the variation from the plan (or possibly take advantage of it) and to improve how the work is performed

5. Execute your adjusted plans

The first step—identification—occurs in a number of ways in the Harmony/ESW process. The most important of these is the tracking of the execution of the project against the schedule, the topic of the next section. Further, testing identifies necessary changes in the system analysis, design, or implementation, as discussed in the previous chapter. In addition, the party phase (aka the increment review) is a short (half- to full-day) task performed at the end of every microcycle to capture and act on “lessons learned” during the microcycle. Last, model reviews allow developers to learn technical approaches and lessons from others as well as to ensure that the models adhere to architectural intent.

Replanning, the fourth step, is done primarily at the microcycle level in the prototype definition activity. While the primary focus of this activity is to address the scope and content of the work in the current microcycle, higher-level overarching planning and concerns are usually adjusted in this phase as well.

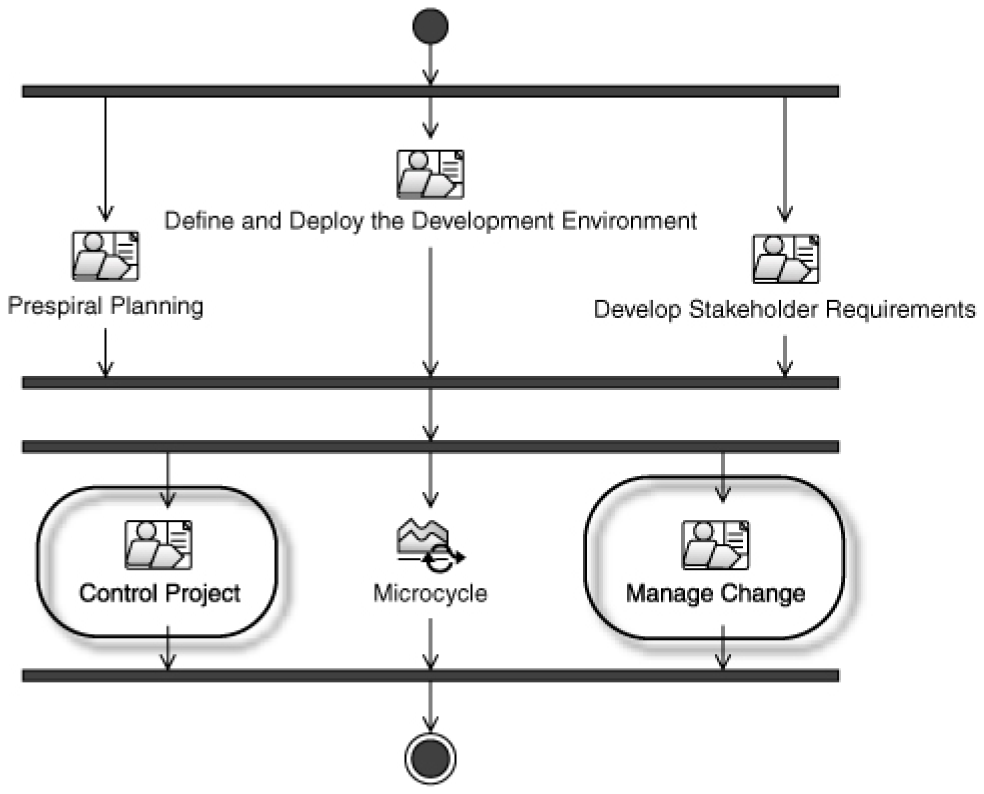
Analysis of the work style, approach, techniques, and tools allows us to identify process inefficiencies that can be improved by changing how we work. This is a critical part of process improvement and can be done “in-line” during the execution of the project. This is primarily done in the party phase, performed at the end of each microcycle.

Change management provides an infrastructure for controlling changes to work products, including project plans. Change management is normally a more or less structured activity for the process of requesting, planning, analyzing, and implementing changes to a system or project work product. The most common changes to be managed are changing requirements, identified defects, development plans, and schedules.

The fifth step, executing the adjusted plans, is performed in the daily nanocycle of development, a topic on which I've spent a lot of space in this book. The adjustments are realized in new or updated work items that are scheduled within the daily work to be done by the project staff.

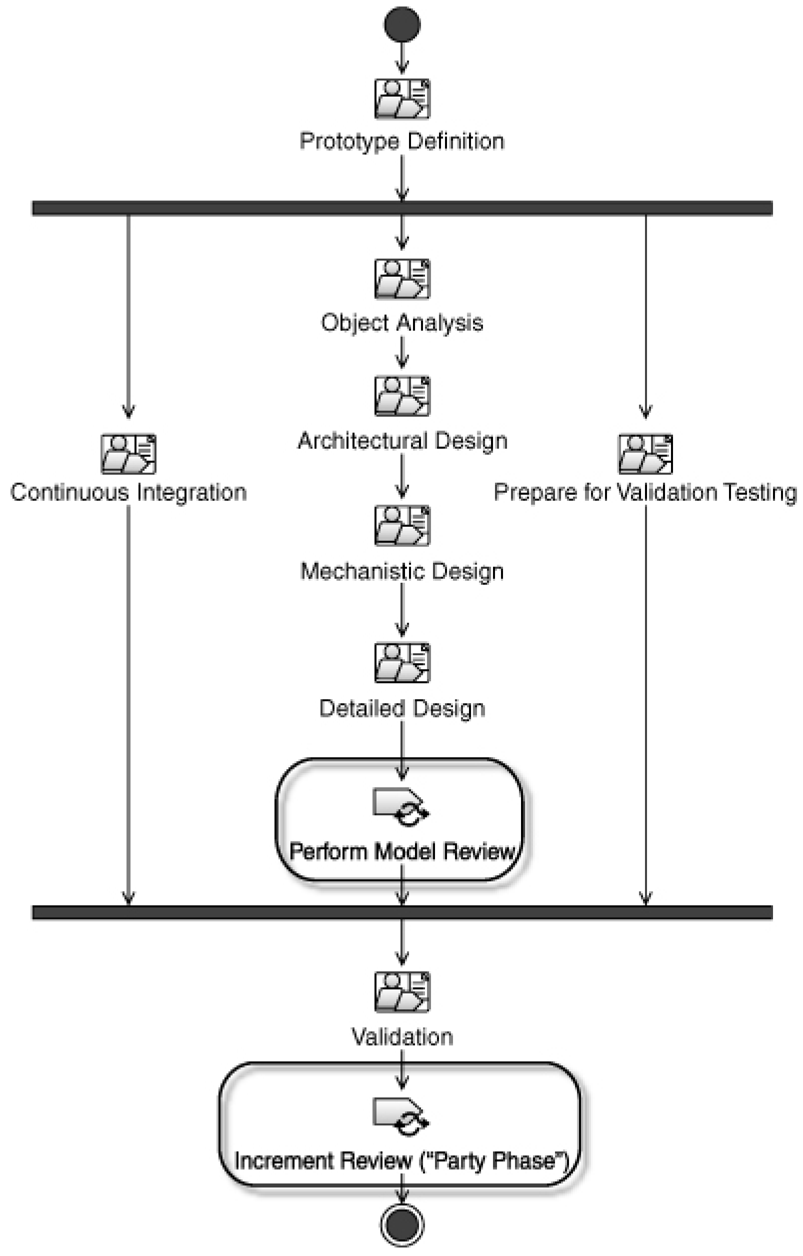
The high-level activities concerned with process and project optimization can be seen in Figure 9.1. These activities are executed in parallel with the sequence of microcycles that follow the project initiation activities. These activities will be discussed in the relevant sections later in this chapter.

Figure 9.1 *Tracking, controlling, and change management activities in the overall Harmony/ESW process*


Define and Deploy the Development Environment
Prespiral Planning
Develop Stakeholder Requirements
Control Project
Microcycle
Manage Change

Besides those overall activities, the microcycle itself includes a couple of activities that address these concerns. Testing, as discussed in the previous chapter, occurs in the object analysis, architectural design, mechanistic design, detailed design, continuous integration, and validation activities. This chapter will focus on the activities highlighted in Figure 9.2.

Figure 9.2 *Process optimization activities in the microcycle*



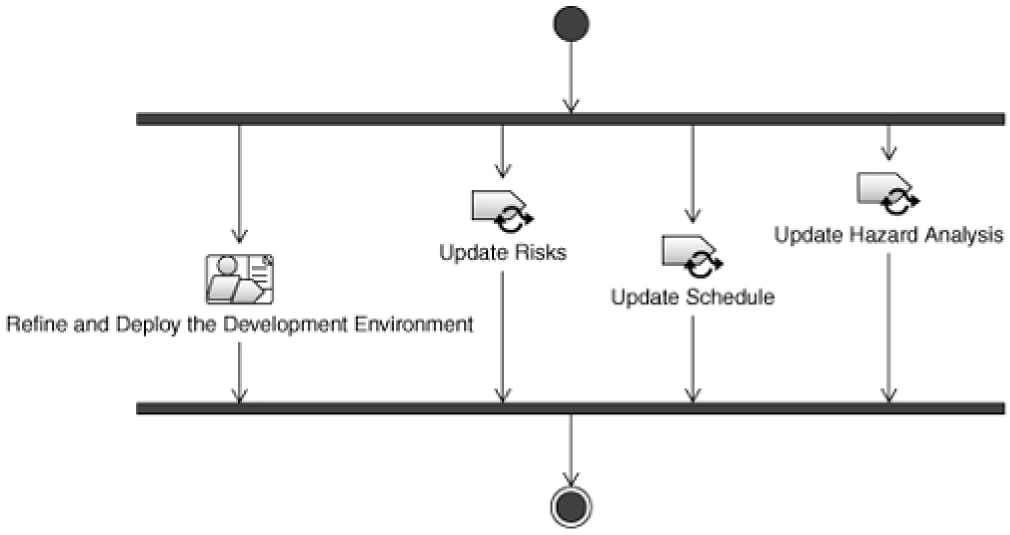
9.2. Tracking and Controlling

As I've mentioned a time or two before, software projects are very dynamic. They involve invention on a daily basis and so traditional industrial processes cannot really be applied successfully most of the time. Such "ballistic" planning approaches assume infinite accuracy and fidelity of information, but that's not how software projects actually are. Once we accept this, we are left with a need for dynamic planning—plans that must be actively and frequently modified as we learn more and as things change. A crucial aspect of dynamic planning is uncovering the actual condition and progress of the project so that we have enough information to adjust our plans. This is known as **project tracking.** Changing plans on the basis of information thus garnered is called **controlling the project.** Tracking will be done via change requests, model reviews, and the periodic project assessment (known as the increment review, or the party phase). Controlling is the actual updating of the various plans, directions, and development environment based on the results of the tracking tasks.

9.2.1. Controlling Project Workflow

In the Harmony/ESW process, tracking and dynamic plan adjustment are performed primarily in the "Control project" activity that runs in parallel with all the development work. The "Control project" workflow is shown in Figure 9.3. The individual activities are detailed in the following sections.

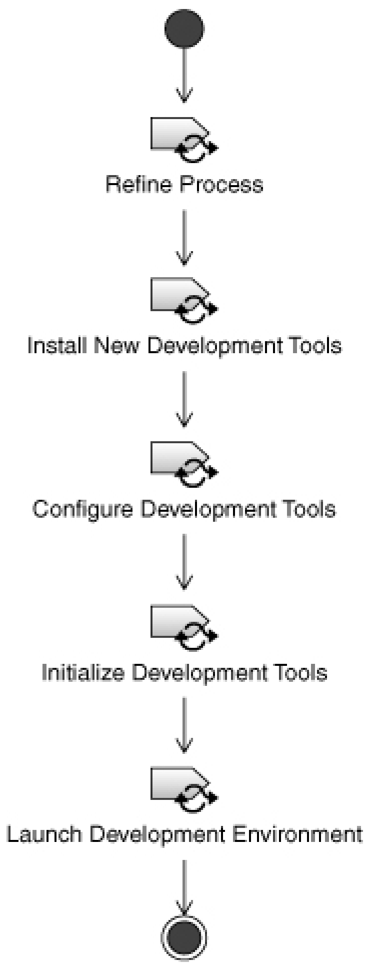
Figure 9.3 *"Control project" workflow*



9.2.2. Refining and Deploying the Development Environment

The "Refine and deploy the development environment" activity contains a set of tasks (see Figure 9.4) focused on both maintaining the IT infrastructure of the project development team and updating the process. The changes to the process are, for the most part, identified during the party phase (discussed later), but they are implemented in this task. For example, it may be discovered that the build process for performing continuous unit testing and continuous integration just takes too long, and that providing a bank of servers for building the system would be an improvement to the team's efficiency. In this activity, the servers would be installed, configured, and initialized with the build tools (CM tools, compilers, linkers, etc.), and the process guidance for how developers should use that environment would be updated.

Figure 9.4 *"Refine and deploy the development environment" workflow*


Refine Process
Install New Development Tools
Configure Development Tools
Initialize Development Tools
Launch Development Environment

Of these, the most interesting is the task "Refine process." The other tasks are rather more obvious, and the steps for installing and configuring tools are highly tool- and environment-dependent.

Refining the Process

The purpose of this task[2] is to address identified process inefficiencies and provide updated guidance to project team members to guide them through their work. This usually results in updated process documentation and possibly retraining, as appropriate. Processes can be documented in a number of ways, from Word and PowerPoint descriptions of the workflow, work products, roles, and guidance to more formal representations in tools such as the open-source EPF Composer or the more powerful IBM RMC.

2. IBM Rational defines a process for authoring and managing processes known as MAM (Method Authoring Method). It is provided as a part of the RMC process content. See the IBM Rational Web site for more information.

The steps involved in this task are the following:

1. Assess the project needs

Review previous team issue questionnaires (see the “Party Phase” section later in this chapter), microcycle mission statements, schedules, progress reports, and other relevant data

2. Review the current process guidance

Understand the current approach and analyze deficiencies

3. Define the scope of the tailoring effort

As a result of the identified process deficiencies, identify the related portions of the processes (workflows, tasks, work products, checklists, and other guidance) that require modification

4. Tailor the processes

Update the appropriate process information with the improved process guidance

5. Integrate the disciplines

Ensure that the updated process guidance works well with the other aspects of the process, for example, that the updated outputs from one task reflect the input needs of subsequent tasks

6. Maintain the process

Put the updated process guidance under CM

7. Disseminate the process guidance to project team members

As process updates occur, the team members need to be made aware of the changes so that they can adapt their work.

The usual impetus for updating the process is either developer pain or a deviation from plan. In either case, the fix may be to either change the plan or change how the plan is implemented, or both.

9.2.3. Updating Risks

Back in Chapter 5, "Project Initiation," I talked about the importance of project risk management and creating the risk management plan. This task updates the risk management plan as existing risks are resolved or new risks are identified. The task consists of the following steps:

1. Analyze the results of the RMAs

2. Review the risk details

3. Identify and clarify new risks

4. Add or modify risk details

RMAs[3] end up as work items that are performed during the microcycles. The result of the RMA, known as an **outcome**, provides valuable information that can be used to appropriately update plans and technical approaches. Is CORBA fast enough for the product needs? Take a performance critical flow from the system design, design it with CORBA, and measure the message delivery speed, throughput, or some other appropriate criterion. Does the real-time framework result in a sufficient reduction in memory usage? Incorporate the framework and measure the reduction in user-developed code and apply estimation metrics to scale it up for the project. The point of RMAs is to actively look at potential problems and determine whether they are real problems or not, and their actual severity if they are. This information is then used to adjust plans when potential problems are discovered to be real. The details of the risks —such as their likelihood, severity, and so on—are contained within the risk management plan and must be updated throughout the project as RMAs are performed (as well as when new risks are identified).

3. Some agile authors refer to these as *spikes*.

9.2.4. Updating the Schedule

The project schedule is a timed sequence of work activities allocated to human resources. It is used to plan what needs to be done, when it needs to be done, and who needs to do it. It is a plan against which the project must be tracked to determine whether the plan is being met and whether adjustments in the plan or the work implementation must be made. The schedule should be tracked daily or weekly and updated no less than once per microcycle. The schedule can be updated more frequently than that depending on the project needs and the variance from plan.

The steps involved in updating the schedule are:

1. Track actual project progress against the plan

2. Determine the variance between actual and planned progress

3. Determine the actions to manage schedule variance

4. Update the schedule to reflect replanning

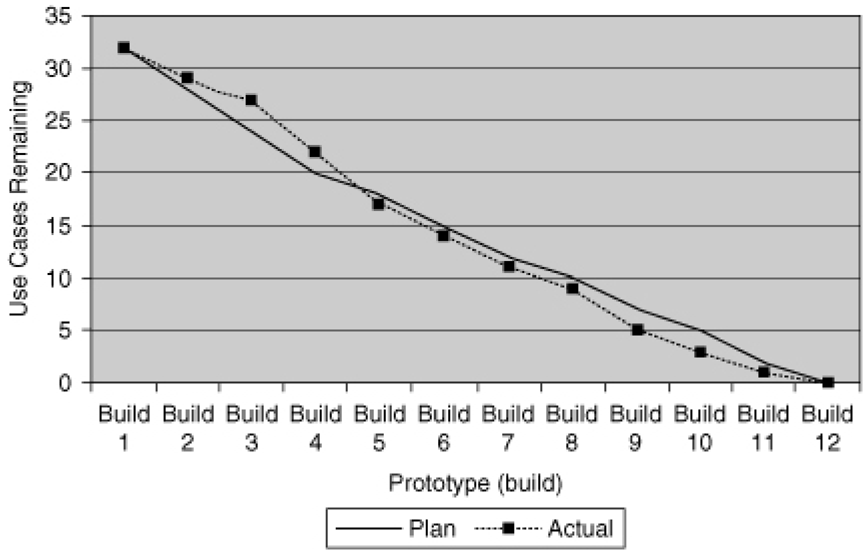
Tracking is the most interesting and complex step, so let's consider that in some detail. Schedules are captured in various forms such as Gantt charts, PERT charts, work breakdown structures (WBSs), and so on. Those same views can be, and frequently are, used to show progress. There are a couple of other views that I have found useful as a supplement to these more traditional views.

For the graphically inclined, burn-down[4] (or burn-up) views provide a good view at a glance of overall progress. These are goal-versus-time views in which the completion of a set of goals, such as work items, use cases, or features, is shown as a function of time. The "burn-down" list shows remaining goals along the ordinate axis while the "burn-up" (completion) view shows the number of such goals completed as a function of time. In addition, I also like to show the planned project goal trajectory on the same graph. Different such views can be used for different timescales, but usually only the microcycle (work item) and macrocycle (use cases, product features, story points, or use case points) are shown. Of course, the project velocity is the slope (first derivative) of that curve.[5]Figure 9.5 shows a typical burn-down chart for a system with 32 use cases. Figure 9.6, on the other hand, shows a work item completion chart within a specific microcycle that plans to implement about 90 work items.

4. See, for example, Cohn, *Agile Estimation and Planning*, for a discussion of the work item burn-down graph.

5. The concept of project velocity was discussed in Chapter 3.

Figure 9.5 *Use case burn-down chart*



Both of these views provide an "overall progress-at-a-glance" perspective. Often, you'd like a more detailed view. For this, I use either the same views, but different charts for each worker; or, more often, I use a tabular view. Table 9.1 shows a section of such a view, in which the empty cells indicate that the data is not available, such as the actual end dates and work efforts for tasks that are still ongoing. The table can be sorted according to different criteria, such as by worker, due date, completion status, lateness, or criticality. In the table shown, the tasks are identified by a change request (CR) number, but a description can be entered instead if desired.

Table 9.1 *Work Item Summary*

Owner	Task	Criticality	Planned Start	Planned End	Last Best Estimate End	Actual Start	Actual End	Planned Effort	Actual Effort
Bruce	CR 105	10	Jan. 4	Jan. 6	Jan. 7	Jan. 5	Jan. 7	9 hr	8 hr
Sarah	CR 92	4	Jan. 4	Jan. 4	—	Jan. 4	Jan. 4	2 hr	2 hr
Sarah	CR 14	7	Jan. 7	Jan. 11	Jan. 11	Jan. 7	Jan. 10	19 hr	14 hr
Lauren	CR 77	9	Jan. 7	Jan. 8	Jan. 10	Jan. 7		6 hr	
Bruce	CR 78	5	Jan. 5	Jan. 9	Jan. 10	Jan. 7	Jan. 10	8 hr	8.5 hr

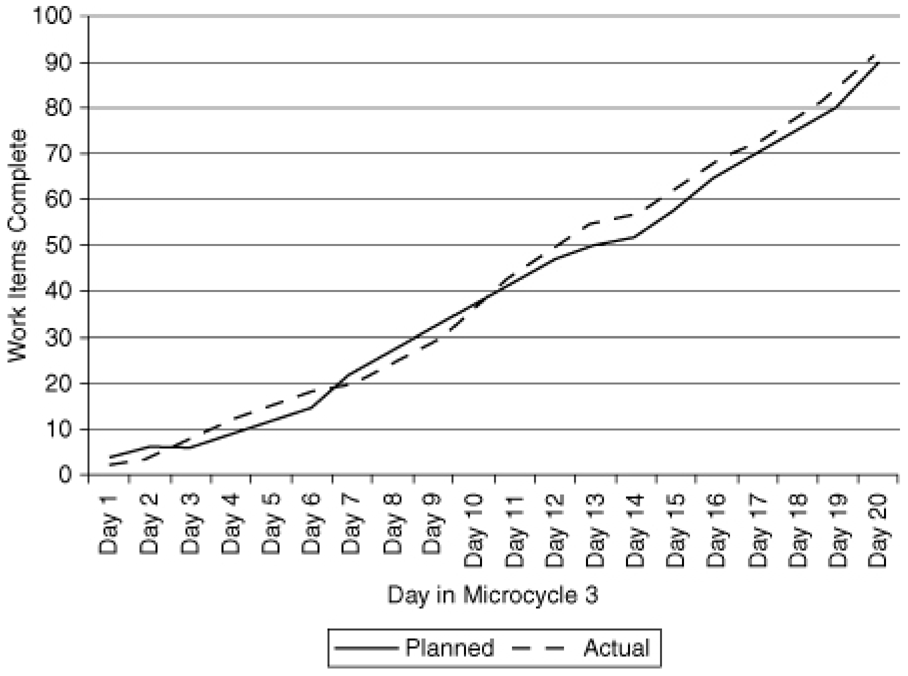
You should never forget that the reason for tracking the actual project performance is so that you can adjust the project when it is necessary or appropriate. The earlier any variance between planned and actual performance is identified, the easier and cheaper it is to correct.

The schedule can be adjusted in many ways to account for or correct schedule variance. One way is to simply lengthen the schedule when a critical-path task[6] is late. That may not always be desirable, but it at least adjusts expectations to reality. Alternatively, the feature can be removed[7] to enable the team to meet the existing schedule. More workers can be added to the task, or the task can be outsourced. The important thing is that the schedule reflects, as accurately as is necessary for project management, the actual work progress; this information is crucial for good management and business decision making.

6. The critical path is the sequence of work tasks such that if any of them is delayed, the project end date is pushed out.

7. An approach I call *featurecide*.

Figure 9.6 *Work items completion chart*



9.2.5. Updating the Hazard Analysis

The hazard analysis was also discussed in Chapter 5. Like the schedule and risk management plan, it is a dynamic document that is continually updated as more information becomes available. In the case of the hazard analysis, new information comes primarily in the form of existing hazards being managed by the design or realization of control measures, and new hazards can be introduced for management based on new information or the evolving design and technology decisions being made. For new hazards, new FTA or FMEA analysis is done and the outcome of the analysis is added to the hazard analysis document.

9.3. Change Management

The change management workflow captures the manner in which changes—such as changes in requirements from the customer, changes in the design resulting from testing or user feedback, or changes in the process resulting from developer feedback during the party phase —are captured, represented, assigned to workers, reviewed, and closed. This is important because projects experience change in many different ways throughout their lifecycles. It is necessary to manage those change requests so that none are lost in the myriad of details with which every project must contend and so that each change request results in an appropriate action, such as a change to a work product or a rejection of the change request.

9.3.1. Change Management Workflow

The Harmony/ESW process provides a workflow for managing change requests, as shown in Figure 9.7.

A change request assumes a number of states as it is processed in the tasks of the change management workflow, such as:

• New

• Accepted

• Rejected

• Assigned

• In progress

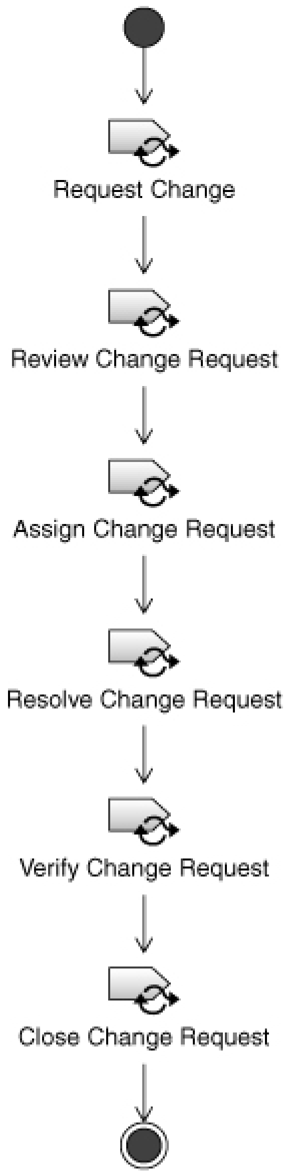
• Awaiting review

• Closed

When a change request is submitted, it is assigned the state "new." Any role (e.g., developer, tester, configuration manager, etc.) in the process can typically submit a change request against any work product (e.g., requirement, use case, diagram, class, source code file, test plan, schedule, etc.). A CCB[8] reviews the request and decides to either accept or reject the change (it may be rejected if the CCB decides it is inappropriate, too expensive, or a duplicate). If the change is accepted, the project manager typically assigns the change request to one or more workers.

8. The change control board may consist of a single person, such as the project manager, in some cases.

Figure 9.7 *Change management workflow*

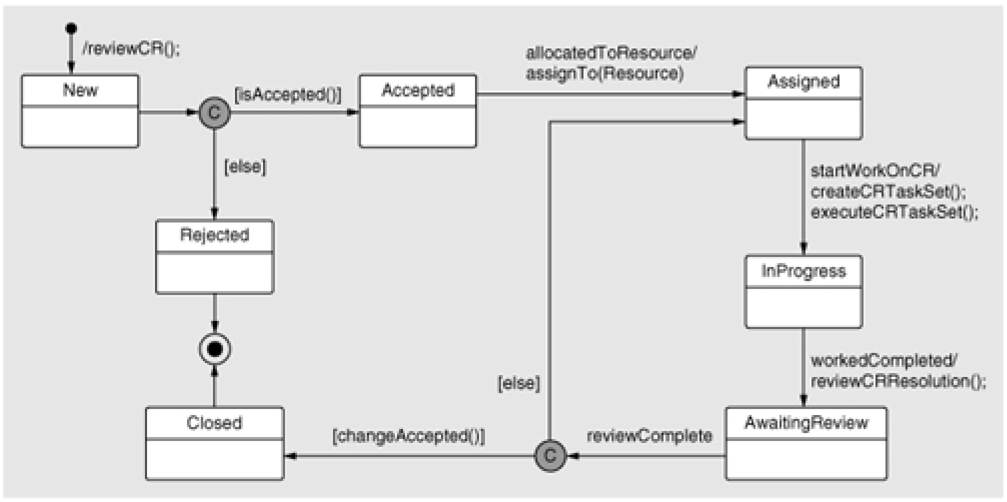


The workers create work items to resolve the request; while the work items are being performed, the change request is in the "in progress" state. Usually, this task requires the creation of one or more work items (work tasks) to resolve the change request. Once the work tasks are complete, the modified work products are submitted to an authority for acceptance (and are in the "awaiting review" state); different work products often have different authorities for this purpose. For software defects, it may be that the software now passes unit, integration, or validation tests. For document defects, the changes are reviewed manually.

Once the changed work products are accepted, the request assumes the “closed” state. If the changes to the work products are rejected, the request reassumes the “assigned” state and the artifacts are once again worked on. The state flow for a change request is captured in Figure 9.8.

The change request workflow is fairly simple but requires some rigor to ensure that all changes are properly addressed.

Figure 9.8 *Change request state machine*



9.4. Model Reviews

Model reviews (aka “model inspections”) are an optional task in the Harmony/ESW process. The purposes of a model review are:

• Improve the content and/or organization of a model

• Disseminate analysis and design information among a set of stakeholders

• Ensure adherence to quality assurance standards and guidelines

The ultimate purpose of a model review is to improve the product or decrease project time. Some people believe that this is the primary means by which quality is ensured within the model. These people would be *wrong*.[9] The *primary* way to ensure that a model or system has no defects is not to put any defects into the model in the first place. We achieve and demonstrate that with continual execution and integration. Model reviews *do* add value but at a high cost. In a review, you may have a dozen people in the room and 24 eyes looking over a work product. It is far cheaper to find errors by running and testing the model than to find them by looking at the model or source code.

9. Not that I have an opinion about that or anything ☺.

For this reason, I strongly recommend that you review only models that have been debugged and formally unit-tested. I don't want to spend a dozen people's time and effort to find defects such as "The statement is missing a semicolon—it won't even compile!" That is a hugely expensive way to find those kinds of defects. I want to find problems such as:

• The model doesn't meet naming standards (identified by the QA role)

• It isn't consistent with the architectural intent (architect role)

• It doesn't meet the requirements properly (use case analyst role)

• It doesn't meet the technical need (SME role)

Of course, another reason to review the model is to share analysis and design approaches that work well to improve this and other models.

Any work product can be the subject of a review, but because of the expense of performing reviews, the Harmony/ESW process recommends it only for key project artifacts such as the system requirements (and associated use case model) and the analysis or design models. Many *Agilistas* actively discourage the use of reviews because they can lead to person-weeks of wasted time if performed inefficiently. For this reason, the Harmony/ESW process specifies a particular, efficient way of performing reviews based on Fagan inspections.[10]

10. Wikipedia has a good discussion of Fagan inspections. See http://en.wikipedia.org/wiki/Fagan_inspection. The original reference is Fagan's paper "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* 15, no. 3 (1976), 182–211, also available from the Wikipedia article.

In a code-based development effort, the code is the primary artifact type to be reviewed. In a model-based development effort, the code review is optional and infrequent. The model is the primary subject of review. In the Harmony/ESW process, reviews can be performed at any time, but generally they are performed at the end of prototype definition (CIM review—optional) and object analysis (PIM review—optional), and at the end of detailed design (PSM review—recommended).

In general, a model review is focused on a single portion of a model and has a narrow purpose, such as:

• To evaluate the requirements, use cases, and their details

• To evaluate an object analysis (platform-independent) model

• To evaluate a design (platform-specific) model

• To evaluate the adequacy and coverage of a test plan and its details

• To evaluate architectural decisions/models

A review should last no more than two hours and should contain multiple diagrams relevant to the mission. To keep the meeting focused, the review moderator requires reviewers to come in with review comments, and only those comments are discussed. Solutions are not identified during the meeting but are added only after the meeting. If a reviewer has a solution in mind, he or she and the product owner can get together after the meeting to discuss it and not waste the time of the other 10 people in the room.

Performing a review requires several distinct roles.[11] The product owner answers questions and addresses comments during the review. The **Moderator** runs the meeting. The **scribe** takes down the meeting minutes and captures the action items. The **QA** role examines the work product for adherence to standards and guidelines. The **architect** looks for adherence to architectural concepts. The **reviewer** identifies concerns and issues with the work products.

11. The roles are somewhat different from those identified in Fagan's original 1976 paper. This reflects the nature of model-based versus code-based development.

The steps involved in reviewing a work product include:

1. Determine the purpose of the review

This step identifies the goals of the review to the attendees, product owner (presenter), and moderator.

2. Prepare the materials for the review

The materials necessary to achieve the review's goal need to be disseminated. For model reviews, this is usually a set of diagrams related to the purpose. For example, a use case review would include the portion of the stakeholder and/or system requirements relevant to the use case, the use case diagram, the use case description, the use case scenarios, and the use case state. A PIM review is usually performed for a use case at a time, and the work products reviewed include one or more class diagrams showing the software elements realizing the use case, their state or activity diagrams, sequence diagrams showing the execution of the collaboration realizing the use case, the use case sequence diagrams, and possibly the source code generated from those classes.

3. Disseminate the materials to reviewers

The review materials must be delivered to the reviewers early enough to allow them to perform the review prior to the meeting but not so early that they put off reading them or forget them entirely. This means at least two days but not more than two weeks

4. Schedule the review

The review must be scheduled so that action items resulting from the review can be addressed in a timely way. I remember another project in which the code was reviewed after the product was shipped to the customer, and then only to get the “check mark” for meeting the letter of the process.

5. Reviewers inspect the material individually

This is an absolutely crucial step for efficient reviews; the reviewers *must* show up at the review meeting having read the materials so that they arrive with questions and comments. It is inefficient to spend the meeting teaching the attendees the content. The reviewers should come to the meeting with prepared comments and questions to be discussed.

6. Collectively discuss the review contents in the review proper

The moderator has the responsibility of keeping the meeting on track and progressing. The moderator systematically ensures that everyone’s comments and questions are raised and addressed. Further, the moderator limits the discussion to clarification and identification of the action items (captured by the scribe role) but prevents discussions of possible solutions. Again, because reviews are very expensive, it is important to make them as efficient as possible. To this end, any discussion of possible solutions is done outside the meeting with a limited subset of stakeholders.

7. Collect the work items resulting from the review

The scribe captures the work items from the review and publishes the list to the meeting attendees. This list should include unresolved work items (some comments or questions may be resolved by clarification and not require additional resolution).

8. Perform the work items resulting from the review

The product owner has the responsibility for addressing the unresolved work items resulting from the review meeting. If the changes are trivial, then no further meetings may be necessary. If the changes are substantial, then a subsequent review may be required. In this case, only the work items from the previous review are discussed in the follow-on review.

9.5. The “Party Phase”

The formal name for this activity is the increment review; it is often called a project or microcycle postmortem. However, a postmortem is a procedure performed by a forensic physician to determine why someone died. I view activity as a “celebration of ongoing success” rather than a postmortem, hence the informal name “party phase.” The point of the party phase is to improve the efficiency and productivity of the project team during the execution of the project. We do this by paying attention to project progress, identifying inefficiencies and problems, and then addressing them. This phase is where the Harmony/ESW process addresses the concerns of CMMI levels 4 and 5, which require a process that self-optimizes.

9.5.1. Party Phase Workflow

The party phase is short—usually less than a day, but certainly no more than two days. It focuses on specific measures of efficiency plus a “catchall” issue questionnaire to be completed by the team to identify concerns not otherwise addressed.

The basic workflow is a short set of steps:

1. Distribute the issue questionnaire to the project team

2. Review the issue questionnaire feedback

3. Review the schedule progress against the plan

4. Review the defect list and defect rates

5. Review architecture scalability

6. Review risk reductions against the risk management plan

7. Review reuse against the reuse plan

8. Review the process workflow concerns

The result of the party phase is a set of work items that must be scheduled into the subsequent microcycles.

9.5.2. Issue Questionnaire

The issue questionnaire is not meant as a “gripe forum” but rather a vehicle for eliciting useful feedback from the project team about their issues and concerns. Ideally, both positive and negative feedback is provided by the team so it is clear to the project manager what appears to be working well and what appears to be inefficient or broken. A typical issue questionnaire will include questions such as these:

• Is progress on your work items meeting the plan in terms of schedule and quality?

° If not, why not?

° If so, to what do you attribute that success?

• What appears to be working well in terms of the project approach?

• Is the project tool environment working well?

° What is the best aspect of the tool environment?

° What is the worst aspect of the tool environment?

• Is the architecture providing a good infrastructure for the product?

• Are the process workflows aiding your success?

° If you were charged with improving them, what changes would you make?

• What are the barriers or potential barriers to your success?

• What do you perceive as the three highest risks to your success in this project?

• Do you have any other concerns or comments?

9.5.3. Reviewing Process

As mentioned in Chapter 1, "Introduction to Agile and Real-Time Concepts," the reason we define and deploy a process is to improve our ability to reliably and efficiently produce high-quality software. Before we run the project, what we have is a theory (however existentially based) about how it will work. We need to adapt the process details based on "facts on the ground." It is common during the execution of a process on real projects to find little inefficiencies that result from differences in IT environment, tool sets, team personalities, management styles, customer interference,[12] and so on. Further, far too many process workflows are created by people who don't have to live with them,[13] and the real world often differs significantly from the ideal. This is, in fact, so common that there's even a Law of Douglass about it:

The difference between theory and practice is greater in practice than it is in theory.

12. Oops, I mean "interaction." I'm kidding. Really!

13. Having to implement your own decisions is commonly termed "eating your own dog food" and is a practice that I highly recommend!

For this reason, while we have an idea about what the process should be and how it should work in theory, it is important to actually look at how it is working in actual practice and tweak it as necessary. Process optimization means that we learn from our mistakes and improve our efficiency as the project progresses.

9.5.4. Reviewing the Schedule

The schedule is the single most important metric of project progress. The schedule provides a plan for what needs to be done (the work items), who needs to do it (the personnel resources), and when it needs to be started and finished. Reviewing the schedule means comparing the actual progress against the plan to identify discrepancies. The actual progress is determined by tracking the work items. While the formal schedule review occurs once every microcycle, tracking must take place far more frequently—usually daily—as discussed earlier in the chapter. The point of this schedule review is to evaluate the velocity of the project, evaluate the feasibility of the schedule, and determine if more sweeping changes to the project are necessary or appropriate.

9.5.5. Reviewing the Defect List and Defect Rates

The other side of the software coin is the quality of the software. By following an agile process that provides constant feedback about system quality, we expect few defects to be identified and result in work items. However, it is important to determine the actual truth of the matter. This step is done to see if the actual defect rates are in line with expectations. If defects are too high and too much time is spent in either integration or validation because of this, the process is clearly allowing too many defects to be entered. It may very well be necessary to modify the process to eliminate or at least reduce the introduction of defects.

Are the defects internal to components? Then changing how unit testing is performed or the frequency of model execution may be the right actions. Are the defects occurring because the component teams aren't clear on their responsibilities and/or interfaces? If so, then the architectural intent is unclear or not being properly communicated or managed. Are the defects primarily about not implementing the requirements properly? Then perhaps the use case sequence diagrams are not adequate or they are incorrect in detail. Are the defects coming from the customer, indicating that the system passing validation isn't meeting the need? Then the requirements need improvement.

9.5.6. Reviewing the Reuse Plan

The reuse plan is optional and relevant in the party phase only if the project plans call for reusing existing work products or for the creation of reusable work products. These work products may be requirements, use case models, analysis models, design models, source code, test fixtures, and so on. Reuse can be "in the large" (such as frameworks, platforms, components, and architectures), of medium scale (such as design patterns), or small-scale (classes, functions, algorithms, or data structures). If there is a reuse plan, then this review compares the contents of the plan against the plan's execution.

9.5.7. Reviewing the Risk Management Plan

The risk management plan (also known as the "risk list") is a dynamic work product documenting the identified risks, their metadata,[14] any associated RMAs, and the results of execution of those activities. The primary concerns of this review are to analyze the results of the RMAs (and take corrective action if appropriate) and to look ahead for new risks.

14. Including severity, likelihood, and risk.

9.6. Summary

This last chapter wrapped up the process with a discussion about project tracking and controlling. By “tracking” I mean identifying project status and comparing it to planned status for the current time. Some measures are qualitative (such as risk assessments), but many of the best ones are quantitative (such as number of requirements validated). The Harmony/ESW process performs tracking at several points within the microcycle. Continuous testing (unit and integration) and periodic testing (validation) identify defects and so give a measure of software quality. Model reviews assess the analysis and design artifacts against company and project standards and against project and design intentions. The increment review, executed at the end of each microcycle, explicitly looks for feedback about project progress and efficiency and can result in change requests.

The other side of the coin is controlling the project. Most of this is done via the process of change management. Change management takes place fully asynchronously to and in parallel with the development of the software; when plans or other work products require updates, the change management process ensures that they get done properly. These changes may be to the analysis, design or code (especially identified defects), or to plans, such as the schedule, risk management plan, reuse plan, hazard analysis, and even the development process and environment.

Applying project tracking and control can make ongoing projects more efficient, lower costs, and improve quality. Because information is gathered and disseminated, subsequent projects can learn from the mistakes and successes of previous projects.

For more information, remember that Google is your friend. The IBM Rational Web site contains links to many white papers and references of interest in addition to a wide variety of development tools, such as Rhapsody, Change, Synergy, ClearCase, and RMC, just to name a few. The EPF Composer is a freely available open-source environment for authoring process content and comes with some simple “exemplary” processes; go to www.eclipse.org/epf. The OMG Web site (http://www.omg.org/) has many standards that may be of interest, such as SPEM (Software Process Engineering Metamodel) and the UML Testing Profile. I do a considerable amount of training and consulting in this area; I can be reached at bruce.douglass@us.ibm.com.

Appendix A[1]
Starfleet ZX-1000 Transporter System Requirements Specification

1. The purpose of this appendix is to provide a requirements specification for various examples shown throughout the book. It is written in the style of a typical specification and no doubt has all the defects usually found in one. Keep this in mind if you actually plan on implementing the design on your spacecraft.

1 Overview

The ZX-1000 Transporter System (ZX-1000TS) is a multipurpose twenty-fifth-century next-generation material transporter for crew and cargo. It is meant to be installed in different configurations for different purposes, but its primary intent is to provide starship- (Intrepid class and above—see Section 3.2) mounted transportation for ship-to-ship, site-to-site, ship-to-ground, and ground-to-ship transportation. For cargo transport up to 20,000 kg, it can be installed in its Cargo2Go™ form with redundant scanner and beamers. For safety-critical transport, such as for transporting crew, it can also be installed in the CrewBeGone™ form with redundant pattern buffers in addition to redundant scanners and beamers, with a total personnel weight limit of 2000 kg, split across up to 12 crew transporter platforms within a single transport chamber.

2 Operational Modes

The ZX-1000 has several different operational modes, described in this section: Cargo, Biomaterials, and Personnel modes. Each primary mode may be optionally operated in Detoxification and Biofilter submodes. The submodes may be enabled or disabled independently from each other. It should be noted that all modes work bidirectionally, that is, both local source to remote target and remote source to local target.

2.1 Cargo Transport Mode

This section defines the requirements for transporting nonliving cargo.

2.1.1 Cargo Capacity

Cargo transport shall be provided by a low-resolution (molecular-resolution) matter transporter capable of 10^{15} calories (4184 petajoules [PJ]) burst energy production, resulting in scanning, reduction, quantum binding, transmission, and reassembly of up to 10^{32} molecules at a clear range (in vacuum) of 150,000 km, or 75,000 in a gaseous medium at 1 atm pressure.

2.1.2 Resolution

The cargo transmitter shall normally be operated at molecular resolution. This is done to conserve power and therefore boost the mass that can be transported. At this setting, a single transporter can transmit 10^{32} molecules.

2.1.3 Life Scanning Safety Interlock

Prior to a cargo transmission, the system shall perform a scan for life signs in the transporter chamber. When set to Cargo mode, the system shall not permit transport if life signs are detected above single-cellular levels (e.g., bacteria), as this would remove life from the transmitted cargo. For more details on life sign scanning requirements, see Section 5.6.

If necessary, the cargo transport installation can be set to the quantum level, required for reliable life-system transport. However, the energy requirements at quantum-level scanning go up, so a correspondingly smaller mass can be transmitted. At this resolution, 12×10^{28} molecules can be reliably transmitted at 150,000 km (in vacuum) or 75,000 in a gaseous medium at 1 atm pressure.

2.2 Biomaterials Transport Mode

Biomaterials Transport mode works identically to Cargo mode except that it operates at atomic resolution, with a corresponding reduction in capacity to 10^{30} molecules at a clear range (in vacuum) of 150,000 km, or 75,000 in a gaseous medium at 1 atm pressure. Note that the same life sign scanning takes place as in Cargo mode; thus, this transport mode cannot be used for live biomaterial transport. For live biomaterial transport, Personnel mode should be used. However, it can be used to transport viruses and complex nanotechnology that does not rely on quantum states.

2.3 Personnel Transport Mode

This section defines the requirements for transporting personnel.

2.3.1 Personnel Capacity

A single person (defined, for the purpose of this document, to be a sentient being of a member race of the United Federation of Planets) ranges in mass from about 2 kg to 200 kg and is composed of up to roughly 10^{28} molecules. The ZX-1000 transporter shall provide from 1 (one) to 12 (twelve) transporter platforms (one per personnel unit), each transporting 10^{28} molecules at a clear range (in vacuum) of 150,000 km, or 75,000 in a gaseous medium at 1 atm pressure.

2.4 Operational Submodes

The ZX-1000 shall provide two operational submodes: Detoxification and Biofiltering. The submodes may be applied in any primary operational mode and may be independently enabled or disabled. The default behavior for each shall be disabled.

2.5 Detoxification Submode

This operational submode modifies the primary mode to scan for, and remove from the pattern buffer, molecules in the currently active hazardous materials list. For detailed requirements on detoxification, see Section 5.8.

2.6 Biofilter Submode

This operational submode modifies the primary mode to scan for and remove live biomaterials that do not match transport criteria. For detailed biofilter requirements, see Section 5.7.

3 General System Requirements

This section defines requirements that apply in all transporter operational modes.

3.1 Timeliness Requirements

The entire transport process shall require no more than 6 s from the transportation inertial reference frame if no precondition violations are identified during the scanning, targeting, initiation, transportation, reassembly, and verification phases. If the computed time for transport exceeds 10 s, the transportation process shall require additional user verification prior to initiation.

Once transportation is complete and confirmed to be successful, the original matter shall be destroyed within an additional 2 s.

> Warning
>
> It is illegal under Federation interstellar law to use the transporter to make multiple copies of sentient beings. For this reason, the Federation Transport Standard (IFS 99833-1105a) requires that the system shall provide verifiable proof that the original matter copy of the sentient being shall be destroyed within 2 s of successful transport. Long-term persistent storage of scanned patterns is permitted by the same standard; however, automatic or user-controlled reassembly shall be permitted only if the transport process of that pattern can be demonstrated to have failed. In the event that it cannot be automatically demonstrated, reassembly from a stored pattern requires a level-1 override and shall result in the generation of an Unstructured Reassembly Event document (IFS99833-1105a-15) to be filed with the Sector Governor's office within 10 standard Terran days of occurrence.

Scanning shall take no more than 0.001 s for a target of 10^{28} molecules (Personnel mode) or 10^{32} molecules (Cargo mode).

> Note
>
> The continuation of consciousness requires fast scanning. Since transported personnel are actually matter constructs, quantum-bound to the original matter, transportation of consciousness should appear instantaneous to the transportee.

3.2 Power Requirements

Matter transportation is an energy-intensive operation. The energy required is a complex function of mass to be transported, distance between transport points, resolution, density between transmission points (of energy, mass, magnetic flux, quantum distortion, and gravity), and momentum displacement. Prior to engaging transportation, the system shall determine all relevant factors and ensure that adequate energy stores exist to complete transportation.

Standard space transport conditions (SSTC) are defined to be

• Transport distance less than 1.5×10^5 km

• Momentum difference less than 2×10^4 m/s

• Absence of significant presence between transport points of

° Matter density (vacuum)

° Energy (less than 10^{10} ergs/cm^2)

° Magnetic flux (less than 10^2 Gauss)

° Quantum distortion (5×10^7 qdf/cm^2)

° Gravity (less than 10 km/s^2)

Standard atmospheric transport conditions (SATC) are defined to be

• Transport distance less than 7.5×10^4 km

• Momentum difference less than 10^2 m/s

• Absence of significant presence between transport points of

° Matter density (oxy-nitrogen atmosphere at 1 atm pressure)

° Energy (less than 10^8 ergs/cm^2)

° Magnetic flux (less than 10^3 Gauss)

° Quantum distortion (5×10^7 qdf/cm^2)

° Gravity (less than 100 m/s^2)

2. Vacuum is defined to be a situation in which the density of matter is $<10^4$ elementary particles per cubic centimeter.

Energy requirements for transportation under SSTC or SATC shall be less than or equal to 10 calories (4184 petajoules [PJ]) burst energy production, for scanning, reduction, quantum binding, transmission, and reassembly of up to 10^{32} molecules.

To meet these energy requirements, the ZX-1000 shall contain a quantum singularity of mass 10^{35} g housed within a warp bubble. This power system shall be able to provide transportation burst energy requirements. The power system shall reabsorb the source matter into energy storage and so shall be demonstrably 92% or more efficient, as required by IFS A879-198.3. Loss energy during a single transport cycle shall not result in increase in ambient temperature of more than 0.5 K.

3.3 Target Scan and Coordinate Lock

The term *target* is defined to be the element to be remotely transported to the location of the transporter chamber. The term *target area* is defined to be the space around the target to a distance of 1 m, or 5% of the linear dimensional measure in three orthogonal space coordinates.

For transport of a remote target to the transporter chamber, the system shall be able to localize the target within the target area to a resolution of 10^{-12} Å. Once the target is localized, transport shall not commence before the target lock is engaged. Target lock shall be considered to be engaged if and only if the location of the target can be retained within 10^{-12} Å for 1 s or longer. The system shall be able to provide target lock under SSTC and SATC.

For transportation platforms traveling at warp, the system shall support transport, provided that both the source and target destinations are traveling within the same warp factor and other conditions required for safe and reliable transport are met.

3.4 Dematerialization

Dematerialization is defined to be the unbinding of quantum states of the original matter target from the remote target material construct, and the destruction of the source matter. To meet Federation “Green Energy” requirements (IFS A879-198.3), the source matter shall be converted to energy and stored in the system power source (see Section 3.2). Dematerialization shall take place within 2 s of successful transportation determination.

3.5 Pattern Storage

The ZX-1000 provides enough storage capacity for 20 personnel patterns at quantum resolution or 50 cargo patterns at molecular resolution, assuming maximum mass for each element. Pattern types can be freely mixed, up to storage capacity. Each pattern shall be stored holographically with enough redundancy to reconstruct an intact pattern with up to 2% bit corruption. Each pattern shall be stored with unique identifier information and pattern demographics. For personnel, this shall include, at minimum, name, 1024-digit Federation ID, race, address, and any special conditions, including medical and legal data.

Prior to transport scanning, the system shall determine whether pattern storage is possible. Storage for at least one pattern is required prior to initiating transport. This local storage is in addition to the normal pattern buffer.

The optional ZX-1000 Storaway™ module allows for additional storage of up to 100 personnel patterns and 250 cargo patterns with the same capacity and redundancy.

3.6 Doppler Compensations

The transporter shall be able to determine the differences in translation and rotational dimensions between the source and target locations and adjust to within 10^{-5} m/s so that transmissions are smooth and do not impart noticeable translative or rotational momentum to transmitted objects or crew across all valid transportation distances.

3.7 Matter Stream Transmission

Matter stream transmission refers to the transmission of quantum and/or molecular data from the source location to the destination. This transmission shall be done as a triphase energy-encoded data burst no more than 0.5 s wide. Transmission shall be done at light speed until the quantum binding has completed. Once that is completed, interaction between the quantum elements is instantaneous at the distances concerned.

Accuracy of target lock (see Section 3.3) is required for focusing the triphased matter stream to coalesce on the proper location. This enables the stream to “pass through” the spaces between the molecules of material between the transmission source and the target while still converging adequately for rematerialization.

The triphased energy-encoded data requires molecular separation found in “normal” matter. Matter in unusual states, such as is found in neutron stars, will effectively block transportation at any distance. For similar reasons, energy shields will normally block transportation unless the energy shield is actively phased orthogonally to the transport signal.

3.8 Rematerialization

Rematerialization refers to the creation of particle mass corresponding to the orginal source material at either the quantum or the molecular level (depending on transportation mode) at the remote location. Rematerialization is more accurately termed "replication" since no actual matter is transported; however, this term has been abandoned by the marketing department due to a loss of customer confidence in private marketing polls on this issue. The system menus and documentation shall at no time use the term *replication* to refer to the assembly of the copy of the source at the target location.

Rematerialization is done by injecting a concentrated energy stream that encodes the source scanned data, followed by the binding of quantum states of the target with those of the source. Rematerialization is complete when all quantum states have been bound.

Following rematerialization, assessment of rematerialization shall take place in no more than 10^{-5} s. Rematerialization is considered successful for personnel if there are fewer than 0.01% errors (Personnel mode) or 0.1% (Cargo mode).

Destruction of the original source shall not occur unless successful rematerialization can be assured.

3.9 Operator Control

The system shall automate tasks that require precise timing (accuracy to within <2 s); however, initiation, safety monitoring, and safety override remain within user control. In addition, the operator shall be able to configure the system, select the transport mode, designate the target location, and store or reassemble stored patterns (under restrictions previously noted).

3.10 Transportation Sequencing

The system requires precise sequencing achievable only through significant automation. The system shall be configured prior to allowing transport. Once the system is configured, the normal sequence of events for chamber-to-target transport, assuming no errors, fault conditions, violation of safety or operational parameters, or operator overrides, shall be:

1. Position source within transport chamber.

2. Select outgoing operation.

3. Select operational mode.

4. Select storage mode (volatile or persistent).

5. Select target location.

6. Scan and assess target location.

7. Initiate transport:

1. Acquire target lock.

2. Make initial scan of source.

3. Validate scanned pattern against original.

4. Store pattern with required redundancy.

8. Perform transport:

1. Execute burst transmission.

2. Rematerialize at target location.

3. Bind quantum states of target to source.

9. Validate transport:

1. Pulse quantum parameters for all transported quanta.

2. Assess error rate.

10. Perform dematerialization.

11. Clear pattern buffer.

Transport sequencing is slightly different if the source initiates at the target site rather than in the transport chamber:

1. Select incoming operation.

2. Select operational mode.

3. Select storage mode (volatile or persistent).

4. Select target location.

5. Scan and assess target location.

6. Initiate transport:

1. Acquire target lock.

2. Make initial scan of source.

3. Validate scanned pattern against original.

4. Store pattern with required redundancy.

7. Perform transport:

1. Execute burst transmission.

2. Rematerialize in transport chamber.

3. Bind quantum states of target to source.

8. Validate transport:

1. Pulse quantum parameters for all transported quanta.

2. Assess error rate.

9. Perform dematerialization.

10. Clear pattern buffer.

3.11 System Configuration

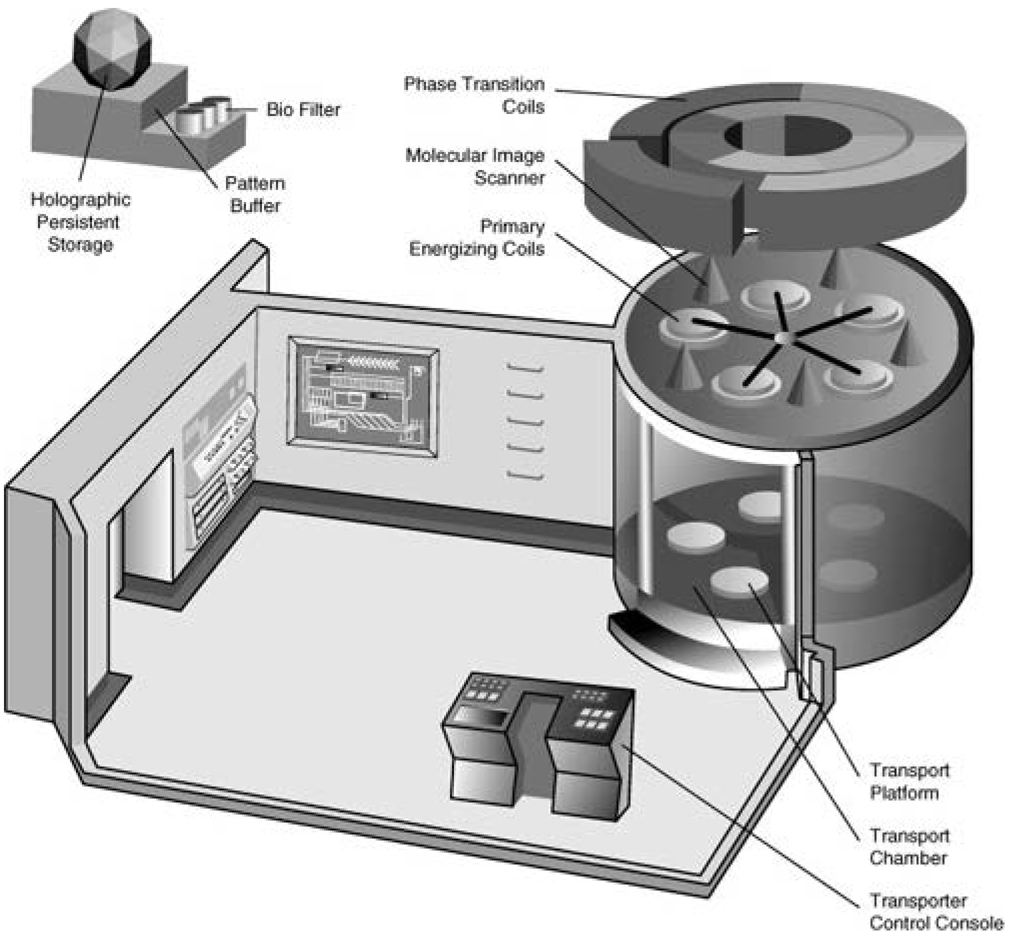
Configuration of the system includes initial setup of the power supply and energy conduits, precise alignment of imaging scanners and transport antennas, initialization of primary and secondary energizing coils and phase transition coils, and performing a series of transport tests of test material to ensure proper operation.

The system shall perform a power-on self-test (POST) whenever reinitialized. In addition, a continual built-in test (CBIT) shall ensure that each component is working within specification. Special care must be taken with the scanner, coil, and emitter array alignment. This may not deviate more than 10^{-8} cm from alignment. The system shall be able to adjust to misalignment of these elements within that error range. The pattern buffer and persistent holographic storage shall also be tested for errors no less often than every 10 transport cycles or one standard Terran day, whichever occurs first.

4 Major System Components

The ZX-1000 system consists of a number of interacting components, each with a specialized purpose. These include the operator console, energizing coils, phase transition coils, quantum and molecular scanners, pattern buffers, biofilters, and the emitter array. (See Figure A.1.)

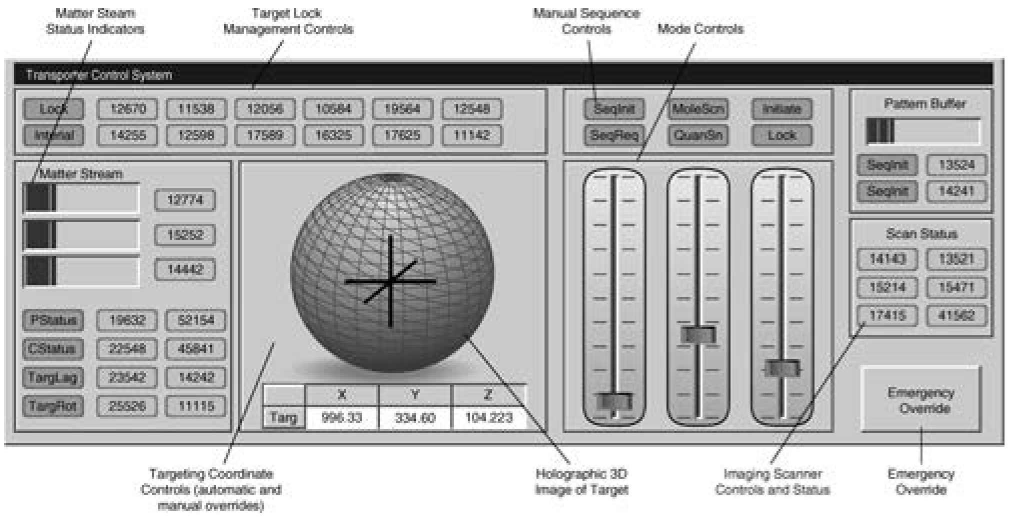
Figure A.1 *ZX-1000 major components*



4.1 Operator Console

The operator console (see Figure A.2) allows user control of system configuration, transportation tasks, and entry of data. The console also provides real-time data for monitoring of the transport cycle and status of POST and CBIT results.

Figure A.2 *ZX-1000 console*



4.2 Transport Chamber

The transport chamber is either an open or an enclosed space of no less than 4 m^2 and no more than 200 m^2.

4.3 Transport Sequencer

The transport sequencer shall provide automated, semiautomated, or manual control over the transport process in accordance with the requirements stated in Section 3.10.

4.4 Primary and Secondary Energizing Coils

The primary energizing coils store and manage the controlled release of the energy required for transport functions. The primary energizing coils provide adequate power for safe transport. The secondary coils shall provide backup. In the case of a primary energizer coil failure, identification and switch to secondary coils shall occur in no more than 10^{-4} s. This timeliness requirement is necessary for smooth and uninterrupted transport. The mean time between failure (MTBF) for the energizing coils shall be 1000 hr of use. Standard routine maintenance shall occur between 200 and 250 hr of use or every six months.

4.5 Phase Transition Coils

The phase transition coils perform the actual quantum or molecular binding through the pinprick singularity created and managed by the emitter array. The phase transition coils shall have a MTBF of 1000 hr. Standard routine maintenance shall occur between 200 and 250 hr of use or every six months.

4.6 Quantum Molecular Imaging Scanners

The quantum imaging scanners scan the local or remote cargo and personnel. The scanners are redundant in case of error. The MTBF for scanners shall be 2000 hr of use. The scanners include parts to store the quanta and molecular metadata, and neutrino emitters and detectors for imaging the contents of the transport chamber. A minimum of three scanners, interlocking with a 60-degree phase shift, is required for scanning.

4.7 Pattern Buffer

The pattern buffer shall be automatically purged 5 s after a confirmed successful transportation unless either overridden by the operator (see Section 4.1) or the pattern buffer is set to provide long-term storage. In the latter case, the long-term buffer storage shall provide pattern storage indefinitely in a quantum holographic medium as defined in Section 3.5. The long-term pattern buffer storage can be used in conjunction with the biofilter to remove pathogens and alien parasites from cargo and crew upon transport retrieval.

4.8 Biofilter

The Biofilter subsystem is responsible for performing hazardous materials filtering as required by Section 5.7. The biofilter has direct access to the pattern buffer and imaging scanners.

4.9 Emitter Array

The emitter array is responsible for creating, stabilizing, and destroying the pinprick singularity through which the transport occurs, as well as supporting the phase transition coils in their role of quantum binding. The singularity binds together the transport chamber and the remote transport site. The emitter array can be grossly aimed through rotation of the ship (fixed mount) and through mounting gimbals. When mounting gimbals are used, the emitter array may be rotated through 120 degrees in two orthogonal axes and maintained in that position with an accuracy of 0.01 degree. Synthetic aperture phasing allows precise alignment with 10 degrees of accuracy. The pinprick singularity shall be maintainable to at least 20 s within standard transporter conditions.

4.10 Targeting Scanners

Targeting scanners are essentially imaging scanners that work on the remote transport site through the pinprick singularity created by the emitter array. In addition to normal imaging functionality, they also employ a phase lock loop to maintain precise alignment prior to and during transport.

5 Secondary Functions

The ZX-1000 provides a number of secondary functions. These are functions that do not constitute more than 5% of intended system usage over its life span.

5.1 Site-to-Site Transport

Site-to-site transport is a special function that is, in reality, the sequencing of two transportations of the same cargo and personnel, first from a remote source to the transporter system pattern buffer, and then, following validation of success, immediately from the pattern buffer to the second remote location. This is done with materialization in the transport chamber. The process shall require either redundant emitters (one locking on the remote source and the other locking on the remote target) or a single emitter to be retargeted following the first phase of beaming the cargo or personnel to the pattern buffer. In the latter case, the entire process shall take no more than 60 s (the majority of the time being required for realigning the emitter array). In either case, the cargo or personnel shall have no awareness of time passing since the quantum binding is to the point in space-time at which the transport cycle began.

5.2 Pattern Buffer Storage

Long-term storage of patterns is desirable for a number of reasons, including biofiltering. However, pattern storage and reconstruction are closely regulated by interstellar law. See the discussion in Section 3.1. This secondary function shall comply with the requirements in Section 3.5.

5.3 Hazardous Material Disposal

In some cases, it may be necessary to dispose of hazardous material. This can be accomplished through the simple means of scanning the material, destroying the original, and discarding the pattern.

5.4 Near-Warp Transport

In near-warp transport, the emitter array shall be able to maintain a relative position lock in accordance with the requirements in Section 3.3.

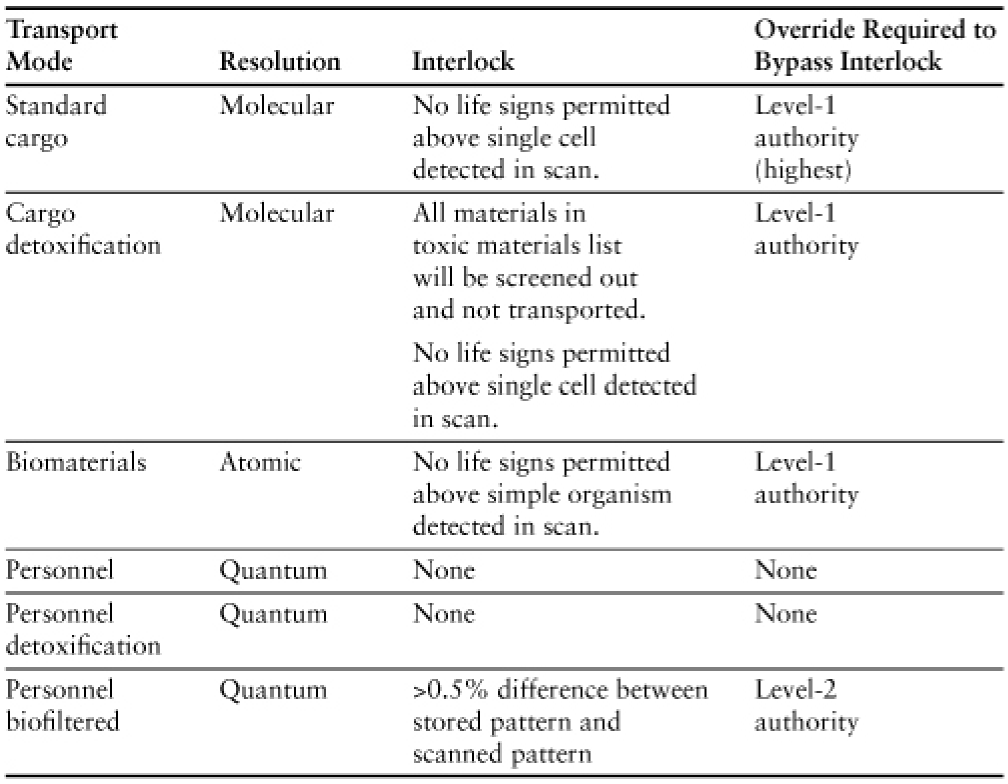
5.5 Warp Transport

For transport between locations, both locations must be within the same warp index (warp I, warp II, and so on). If the two locations are at different warp indexes, then transport between the locations shall be prohibited.

5.6 Life Sign Scanner

The life sign scanner provides a crucial safety interlock for the system. Prior to any transport, the system shall perform a life sign scan. This scan shall cover the entire transport chamber. Depending on the settings for the transport, if life signs are detected, transportation may be prevented, without an authorized priority override, since this may result in the transportation of a structure of a living organism at molecular resolution, resulting in its death. The available mode settings are shown in Table A.1.

Table A.1 *Life Sign Scanner Mode Settings*



Transport Mode	Resolution	Interlock	Override Required to Bypass Interlock
Standard cargo	Molecular	No life signs permitted above single cell detected in scan.	Level-1 authority (highest)
Cargo detoxification	Molecular	All materials in toxic materials list will be screened out and not transported. No life signs permitted above single cell detected in scan.	Level-1 authority
Biomaterials	Atomic	No life signs permitted above simple organism detected in scan.	Level-1 authority
Personnel	Quantum	None	None
Personnel detoxification	Quantum	None	None
Personnel biofiltered	Quantum	>0.5% difference between stored pattern and scanned pattern	Level-2 authority

5.7 Biofilter

The biofilter works by identifying and removing live biomaterial from the transport pattern buffer that does not match that found in a stored pattern or that is on the currently active biofilter list.

5.7.1 Biofiltering against a Stored Pattern

In this configuration, the biofilter compares the pattern stored in the buffer against the reference pattern in persistent storage. The biofilter removes biological organisms from the transport pattern buffer that were not present in the stored reference pattern. Note that this does not include nonliving materials, so inert materials (e.g., clothing and ingested food and drugs) are not affected by this scan.

5.7.2 Biofiltering against the Active Biofilter List

A biofilter list contains a list of biological organisms to screen out when the Biofilter submode is enabled and the biofilter is configured to use the biofilter list. Each list entry shall contain

• A list of common names for the organism

• The scientific name for the organism

• A transporter signature for the organism (up to 10^6 points of identification, commonly unique DNA pair strands for carbon DNA-based life)

• A pattern-match threshold for triggering removal (default 99.9%), settable from 75% to 100% (exact match)

5.7.3 Default Biofilter List

• Earth-based life form infectious organisms

• Orion-based life form infectious organisms

• Klingon-based life form infectious organisms

• Vulcan-based life form infectious organisms

• Silicon-based life form infectious organisms

• Energy-based life form infectious organisms

5.7.4 Managing Biofilter Lists

Users shall be able to

• Create their own named lists of biological organisms to screen out

• Delete existing lists

• Clone an existing list to a new list

• Add new entries to existing lists

• Delete entries from existing lists

• Clone an existing list entry to a new entry in the same or a different list

• Edit each field of an existing entry in an existing list

5.8 Hazardous Materials Filter

The hazardous materials filter (HMF) works by identifying and removing atoms and molecules identified as hazardous on the current hazardous materials list from the pattern buffer. The system shall allow, during configuration, the selection of different sets of hazardous materials. Each list can contain up to 10^6 entries. Entries contain

• List of common names for this material

• Scientific name for this material

• Molecular structure for this material

The system normally uses a single list (default: Earth-based life forms), but up to 20 hazardous materials lists may be simultaneously active. The current list (or lists) will be employed only in Biofilter and Detoxification modes.

5.8.1 Default HMF Lists

At installation, the system shall provide lists for the following life forms:

• Earth-based life forms

• Orion-based life forms

• Klingon-based life forms

• Vulcan-based life forms

• Silicon-based life forms

• Energy-based life forms

5.8.2 Managing Hazardous Materials Lists

Users shall be able to

• Create their own named lists of hazardous materials

• Delete existing lists

• Clone an existing list to a new list

• Add new entries to existing lists

• Delete entries from existing lists

• Clone an existing list entry to a new entry in the same or a different list

• Edit each field of an existing entry in an existing list

5.9 Phase/Frequency/Packet Compensation

This capability is specifically provided for transporting through deflector screens. For rotating phase deflector screens, transport can be done a packet at a time, taking 15 s instead of the normal 4 to 5 s.

Appendix B
Harmony/ESW and CMMI: Achieving Compliance

Abstract

Harmony/ESW (Embedded Software) is a development process for real-time and embedded software. It is guided by a set of principles and realizing practices for model-based development of high-quality systems using an integration of UML and agile methods. Harmony/ESW provides special guidance on safety-critical and high-reliability systems for project managers, team leads, architects, testers, safety and reliability personnel, and quality assurance staff.

The Capability Maturity Model Integration for Development[1] (CMMI-DEV) is a process improvement approach published by the Software Engineering Institute (SEI) to provide companies with the important elements of effective development processes. The primary benefits of mature processes are repeatable quality of products under development and a framework in which processes can improve over time.

1. Carnegie Mellon University Software Engineering Institute, CMMI-DEV (Version 1.2, August 2006). www.sei.cmu.edu/publications/documents/06.reports/06tr008.html.

This appendix discusses how Harmony/ESW not only supports process improvement but can also assist compliance with the CMMI standard.

CMMI Basics

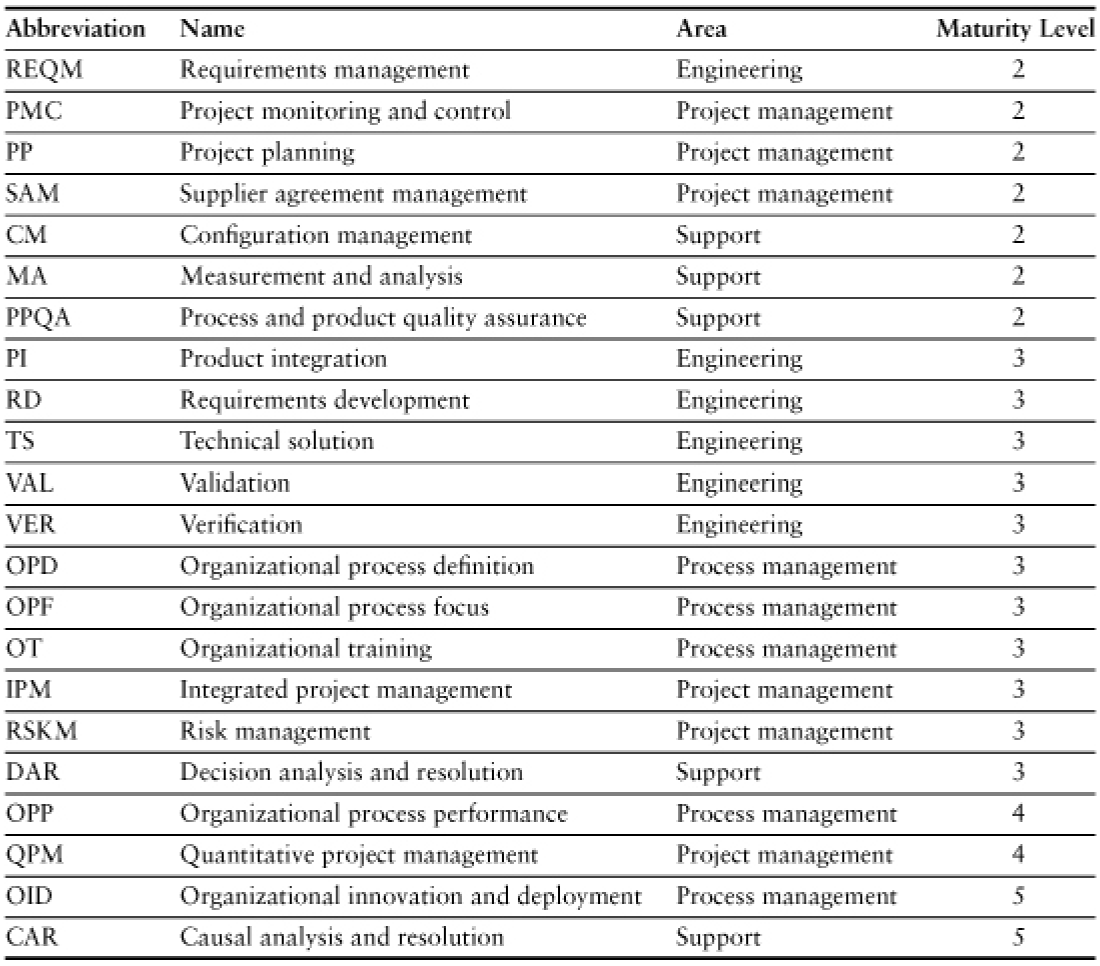
The SEI released version 1.2 of the CMMI standard in August 2006. The CMMI standard is based on the SEI's previous work defining the CMM process model[2] but takes its focus a step further, integrating it with overall business processes across entire organizations.

2. Carnegie Mellon University Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process* (Reading, MA: Addison-Wesley, 1995).

Process Areas

CMMI-DEV defines 22 different process areas (PAs) with which it concerns itself. Each process area includes a set of goals that must be achieved for the organization to consider itself having addressed that process area. Third-party companies exist to assess organizations as to their level of maturity. It is not uncommon for process areas from a higher level of maturity to be addressed even though not all process areas from a lower level of maturity are addressed. In that case, the organization is assessed to be at the lowest level of maturity completely addressed by its processes. Table B.1 lists the process areas and the level of organizational process maturity at which they are addressed.

Table B.1 *CMMI-DEV Process Areas*



Abbreviation	Name	Area	Maturity Level
REQM	Requirements management	Engineering	2
PMC	Project monitoring and control	Project management	2
PP	Project planning	Project management	2
SAM	Supplier agreement management	Project management	2
CM	Configuration management	Support	2
MA	Measurement and analysis	Support	2
PPQA	Process and product quality assurance	Support	2
PI	Product integration	Engineering	3
RD	Requirements development	Engineering	3
TS	Technical solution	Engineering	3
VAL	Validation	Engineering	3
VER	Verification	Engineering	3
OPD	Organizational process definition	Process management	3
OPF	Organizational process focus	Process management	3
OT	Organizational training	Process management	3
IPM	Integrated project management	Project management	3
RSKM	Risk management	Project management	3
DAR	Decision analysis and resolution	Support	3
OPP	Organizational process performance	Process management	4
QPM	Quantitative project management	Project management	4
OID	Organizational innovation and deployment	Process management	5
CAR	Causal analysis and resolution	Support	5

As mentioned, each process area contains a purpose, notes, and a specific set of goals, outlined in the standard. For example, the standard lists the following specific goals (SGs) and specific practices (SPs) for the requirements development (RD) process area:

• SG 1 Develop customer requirements:

° SP 1.1 Elicit needs.

° SP 1.2 Develop the customer requirements.

• SG 2 Develop product requirements:

° SP 2.1 Establish product and product component requirements.

° SP 2.2 Allocate product component requirements.

° SP 2.3 Identify interface requirements.

• SG 3 Analyze and validate requirements:

° SP 3.1 Establish operational concepts and scenarios.

° SP 3.2 Establish a definition of required functionality.

° SP 3.3 Analyze requirements.

° SP 3.4 Analyze requirements to achieve balance.

° SP 3.5 Validate requirements.

Each specific goal has a typical set of work products and specific practices. It is crucial to remember that these are typical, often *recommended* work products and practices but not *required* ones. As long as the goals are demonstrably met, the process area can be assessed to be addressed.[3] This gives the organization freedom to address the process area using different work products and practices as long as they meet the goals. This is important because CMMI does not define a development process. Indeed, any process can be used to achieve a specified maturity level as long as it enables the organization to meet the goals required of that maturity level. It is possible to achieve CMMI compliance with a waterfall, V-cycle, hybrid V-cycle, spiral, or agile process.

3. SEI, CMMI-DEV. See also David Anderson, "Stretching Agile to Fit CMMI Level 3," AgileManagement.Net (2005), at www.agilemanagement.net/Articles/Papers/StretchingAgiletoFitCMMIL.html; and Christine Davis, Margaret Glover, John Manzo, and Donald Opperthauser, "An Agile Approach to Achieving CMMI: Having Your Cake and Eating It Too," *AgileTek*, www.agiletek.com/thoughtleadership/whitepapers.

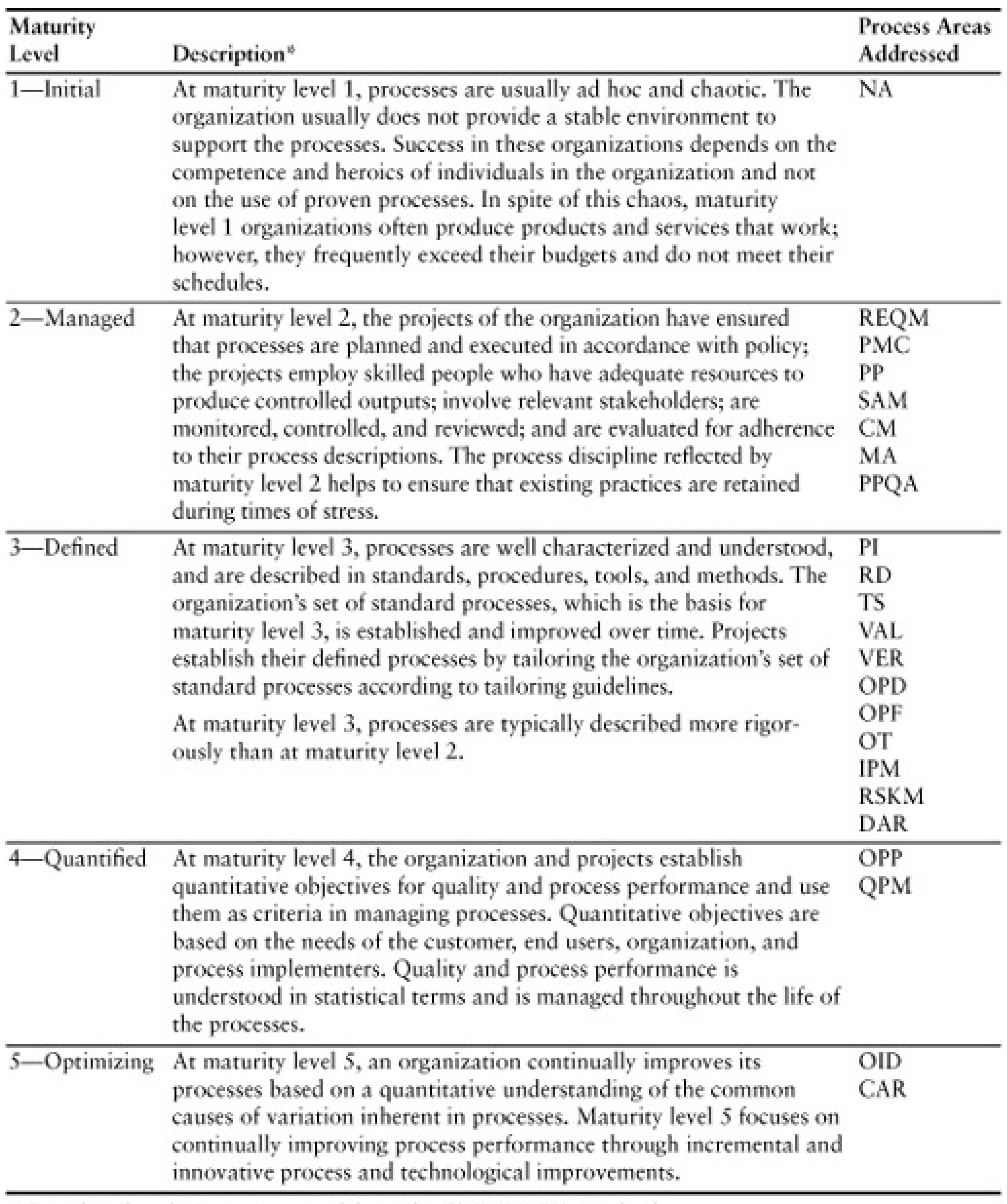
The full set of goals and practices for the various levels of CMMI compliance will not be listed here. The interested reader is referred to the defining specification.[4]

4. SEI, CMMI-DEV.

Maturity Levels

CMMI-DEV identifies five levels of organizational maturity with respect to process (see Table B.2). The maturity level is defined by the organization having achieved all of the process goals for the specified set of process areas. Each subsequent process area includes the process areas of its predecessors. Thus, maturity level 4 includes not only the process areas defined for itself but also those for maturity levels 2 and 3. (Level 1 has no process areas defined.)

Table B.2 *CMMI-DEV Maturity Levels*



Maturity Level	Description*	Process Areas Addressed
1—Initial	At maturity level 1, processes are usually ad hoc and chaotic. The organization usually does not provide a stable environment to support the processes. Success in these organizations depends on the competence and heroics of individuals in the organization and not on the use of proven processes. In spite of this chaos, maturity level 1 organizations often produce products and services that work; however, they frequently exceed their budgets and do not meet their schedules.	NA
2—Managed	At maturity level 2, the projects of the organization have ensured that processes are planned and executed in accordance with policy; the projects employ skilled people who have adequate resources to produce controlled outputs; involve relevant stakeholders; are monitored, controlled, and reviewed; and are evaluated for adherence to their process descriptions. The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress.	REQM PMC PP SAM CM MA PPQA
3—Defined	At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods. The organization's set of standard processes, which is the basis for maturity level 3, is established and improved over time. Projects establish their defined processes by tailoring the organization's set of standard processes according to tailoring guidelines. At maturity level 3, processes are typically described more rigorously than at maturity level 2.	PI RD TS VAL VER OPD OPF OT IPM RSKM DAR
4—Quantified	At maturity level 4, the organization and projects establish quantitative objectives for quality and process performance and use them as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance is understood in statistical terms and is managed throughout the life of the processes.	OPP QPM
5—Optimizing	At maturity level 5, an organization continually improves its processes based on a quantitative understanding of the common causes of variation inherent in processes. Maturity level 5 focuses on continually improving process performance through incremental and innovative process and technological improvements.	OID CAR

*Note that descriptions are quoted from the CMMI-DEV standard.

CMMI has a reputation for being very heavyweight and expensive to implement. This means that it often requires a great deal of *additional* paperwork and distracting activities to achieve compliance. Indeed, some authors report that as many as 400 document types and 1000 artifacts are required to support an appraisal.[5]

5. Dennis M. Ahern, Jim Armstrong, Aaron Clouse, Jack R. Ferguson, Will Hayes, and Kenneth Nidiffer, *CMMI SCAMPI Distilled: Appraisals for Process Improvement* (Boston: Addison-Wesley, 2005).

But it doesn't have to be like this. CMMI does not define or require a particular process. You can—and people have—used much lighter-weight processes to achieve CMMI compliance.[6] And that is exactly what I propose with the Harmony/ESW process.

6. Anderson, “Stretching Agile to Fit CMMI Level 3.”

Achieving CMMI Compliance with Harmony/ESW

The Harmony/ESW process directly addresses many of the process areas in all CMMI maturity levels. There are a few of the process areas whose goals are not addressed directly by Harmony/ESW, but those can be addressed using other means in conjunction with the Harmony/ESW process.

Maturity Level 1

Maturity level 1 does not systematically address any process areas. It is usually characterized as “chaotic” and “ad hoc.” There is no certification for this level of maturity, and we won’t otherwise discuss it here other than to say, “Let’s do better than *that*.”

Maturity Level 2

At maturity level 2, projects are consistently managed using processes. Seven key areas are addressed: REQM, PMC, PP, SAM, CM, MA, and PPQA. Harmony/ESW addresses all but SAM, which it considers to be out of its main focus.

Requirements Management (REQM)

REQM has a single specific goal, “Manage requirements,” and five specific practices:

• SG 1 Manage requirements:

° SP 1.1 Obtain an understanding of requirements.

° SP 1.2 Obtain commitment to requirements.

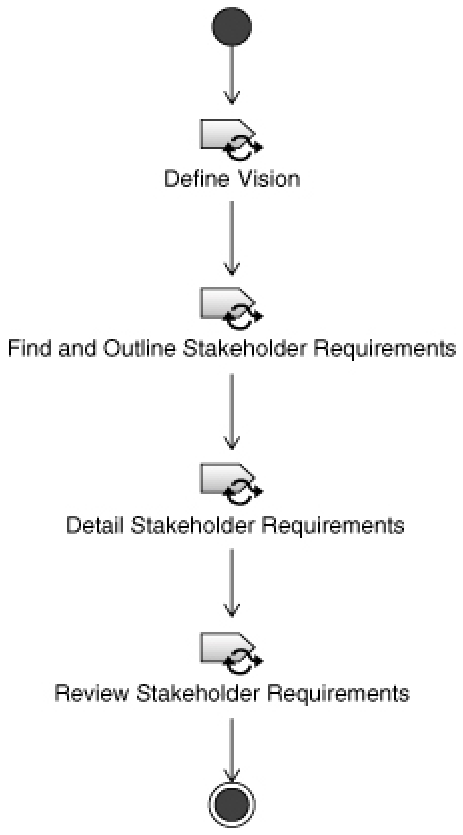
° SP 1.3 Manage requirements changes.

° SP 1.4 Maintain bidirectional traceability of requirements.

° SP 1.5 Identify inconsistencies between project work and requirements.

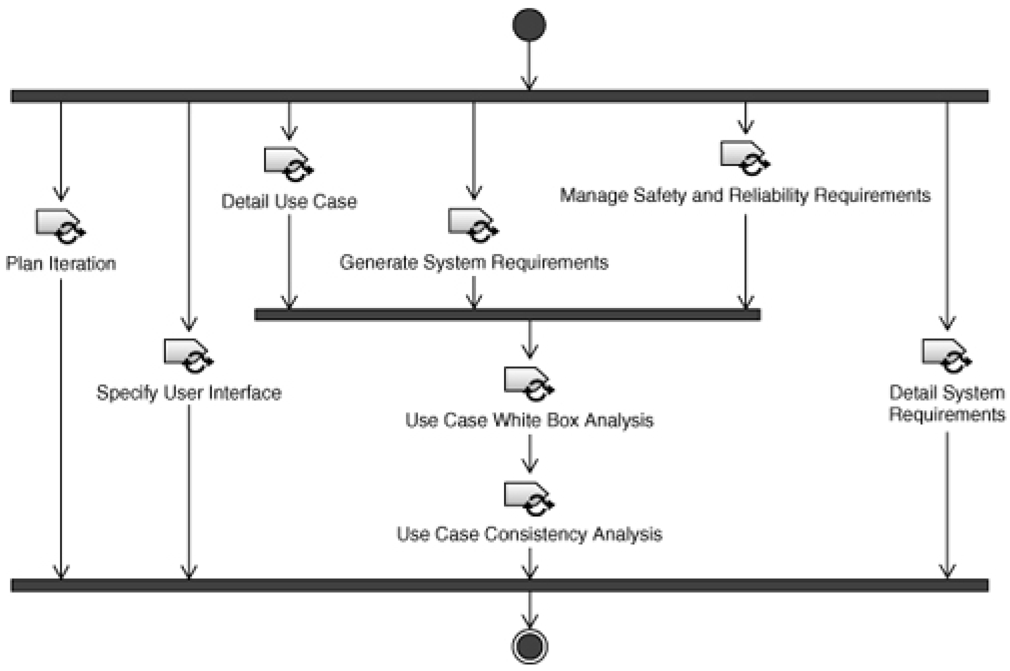
Harmony/ESW fully meets this goal in a number of ways. Prior to the start of the spiral increment, the process defines an activity "Develop stakeholder requirements" (Chapter 5). This activity contains a set of tasks that define and produce both a vision and a stakeholder requirements document. The workflow for this activity is shown in Figure B.1. Each task has a purpose, description, assigned roles, input and output work products, task steps, and associated guidance on how to create and manage such work products that includes all of the specific practices called out for the process area, including achieving traceability between the vision and the stakeholder requirements. More detailed traceability is added later in the process, during the incremental spiral.

Figure B.1 *Tasks within the "Develop stakeholder requirements" activity*


Define Vision
Find and Outline Stakeholder Requirements
Detail Stakeholder Requirements
Review Stakeholder Requirements

Once within the incremental spiral, the prototype definition activity (Chapter 6) incrementally creates the system requirements specification and corresponding use case model. As Figure B.2 shows, prototype definition consists of a sequenced set of tasks that also achieve the goals of the process area, but in a more detailed fashion as downstream engineering continues.

Figure B.2 *Prototype definition*


Plan Iteration
Detail Use Case
Generate System Requirements
Manage Safety and Reliability Requirements
Specify User Interface
Use Case White Box Analysis
Use Case Consistency Analysis
Detail System Requirements

It is important to note that the Harmony/ESW process does not define all the system requirements up front before development work is performed. Rather, it defines an incremental "depth-first" approach in which some requirements are implemented and validated first, before other system requirements are elucidated. This means that the high-criticality or high-urgency requirements can be addressed early, and the resulting design and implementation can be created and tested, before lower-priority requirements are considered. This leads to higher-quality systems because practice has clearly demonstrated that it is extremely difficult to create correct, accurate, and consistent requirements in the absence of design and implementation.

In addition, during subsequent phases of the spiral development, as design and implementation work products are added, traceability to requirements is added as well.

Project Monitoring and Control (PMC)

The purpose of PMC is to provide managers with the ability to identify what corrective measures are appropriate to take when project performance deviates from the plan. This is distinct from the creation of those plans (done in project planning [PP]). There are 2 specific goals for this process area and 10 specific practices:

• SG 1 Monitor project against plan:

° SP 1.1 Monitor project planning parameters.

° SP 1.2 Monitor commitments.

° SP 1.3 Monitor project risks.

° SP 1.4 Monitor data management.

° SP 1.5 Monitor stakeholder involvement.

° SP 1.6 Conduct progress reviews.

° SP 1.7 Conduct milestone reviews.

• SG 2 Manage corrective action to closure:

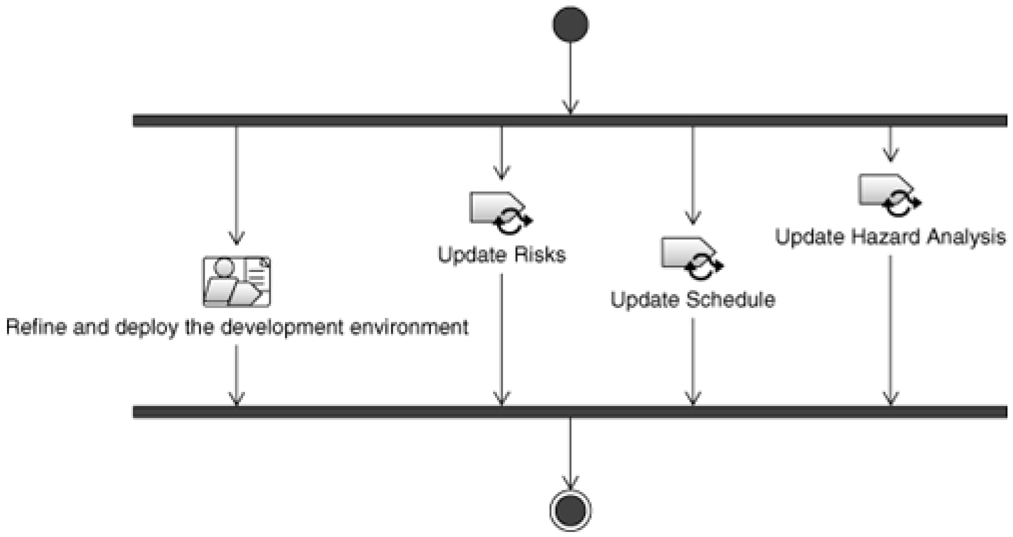
° SP 2.1 Analyze issues.

° SP 2.2 Take corrective action.

° SP 2.3 Manage corrective action.

The Harmony/ESW process addresses these goals in a number of different activities. First, project tracking is performed in an activity known as “Control project” (Chapter 9; see Figure B.3). This activity includes tasks for refining the development environment (which includes updating the process steps and guidance when appropriate), identifying new risks and updating the risk management plan, tracking and updating the schedule, and managing the hazard analysis (for safety-critical projects only).

Figure B.3 *“Control project” activity*



In addition, the microcycle contains a task called the increment review, colloquially known as the party phase (Chapter 9). This task, performed at the end of each microcycle, examines project performance and issues change requests on that basis.

Project Planning (PP)

The purpose of the PP process area is to create and maintain plans that define project activities. Usually this includes developing the project plans, working with appropriate stakeholders, getting plan commitment, and updating the plan when necessary. This process area has 3 specific goals and 14 specific practices:

• SG 1 Establish estimates:

° SP 1.1 Estimate the scope of the project.

° SP 1.2 Establish estimates of work product and task attributes.

° SP 1.3 Define project lifecycle.

° SP 1.4 Determine estimates of effort and cost.

• SG 2 Develop a project plan:

° SP 2.1 Establish the budget and schedule.

° SP 2.2 Identify project risks.

° SP 2.3 Plan for data management.

° SP 2.4 Plan for project resources.

° SP 2.5 Plan for needed knowledge and skills.

° SP 2.6 Plan stakeholder involvement.

° SP 2.7 Establish the project plan.

• SG 3 Obtain commitment to the plan:

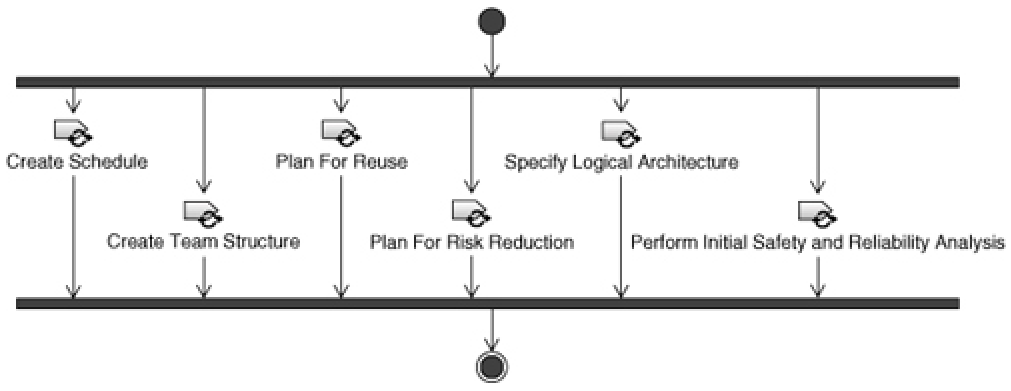
° SP 3.1 Review plans that affect the project.

° SP 3.2 Reconcile work and resource levels.

° SP 3.3 Obtain plan commitment.

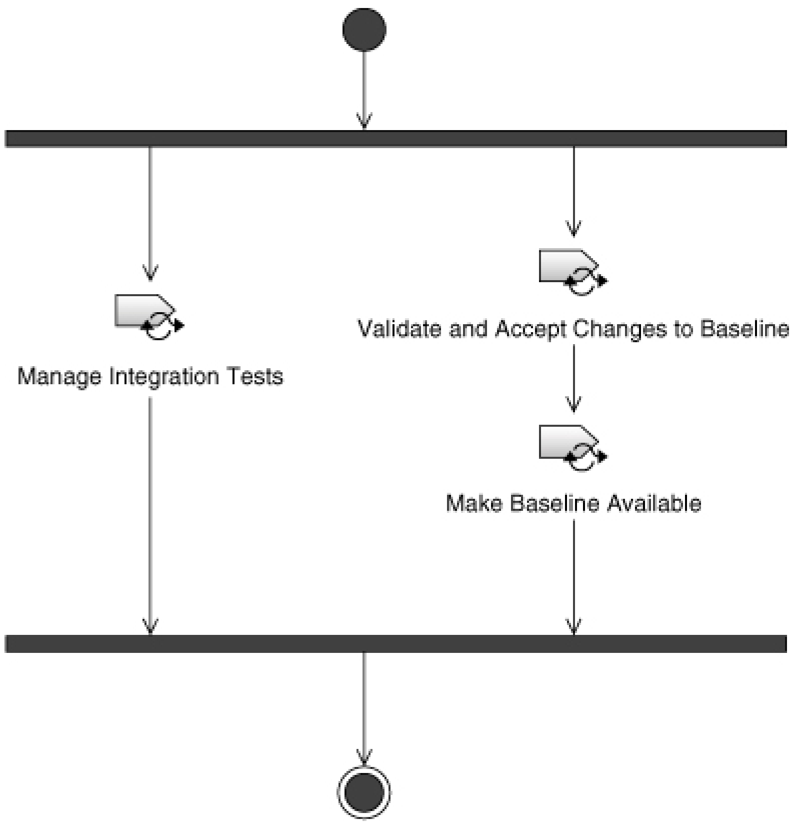
The Harmony/ESW process performs process planning in an activity called *prespiral planning* (Chapter 5) that predates the start of software development. Figure B.4 shows the set of tasks performed within prespiral planning. The creation of the schedule and team organization links the project tasks with the available workforces, estimates the effort and time required, and determines when work products and milestones will be achieved. The reuse planning task identifies how existing work products will be reused and also drives the design of to-be-created work products for the purpose of reuse. The risk reduction task identifies, quantifies, and analyzes project risks to identify risk mitigation activities that are inserted into the schedule and allocated to personnel. The initial safety and reliability analysis task identifies safety- and reliability-related requirements for the system.

Figure B.4 *Prespiral planning*


Create Schedule
Plan For Reuse
Specify Logical Architecture
Create Team Structure
Plan For Risk Reduction
Perform Initial Safety and Reliability Analysis

In addition, replanning is done frequently; the “Control project” activity (Chapter 9; see Figure B.3) monitors some important project metrics (e.g., performance against schedule and risk management plan), and the party phase (Chapter 9) has a review and replanning activity at the end of every microcycle, typically every four to six weeks.

Figure B.5 *Continuous integration tasks*



Supplier Agreement Management (SAM)

Harmony/ESW does not directly address this process area. If supplier management is relevant to the project, additional measures must be added, such as those outlined in the CMMI-DEV specification.

Configuration Management (CM)

The purpose of this process area is to establish and manage the integrity and consistency of a set of work products, using configuration identification, control, status accounting, and audits. This process area has three specific goals and seven specific practices:

• SG 1 Establish baselines:

° SP 1.1 Identify configuration items.

° SP 1.2 Establish a configuration management system.

° SP 1.3 Create or release baseline.

• SG 2 Track and control changes:

° SP 2.1 Track change requests.

° SP 2.2 Control configuration items.

• SG 3 Establish integrity:

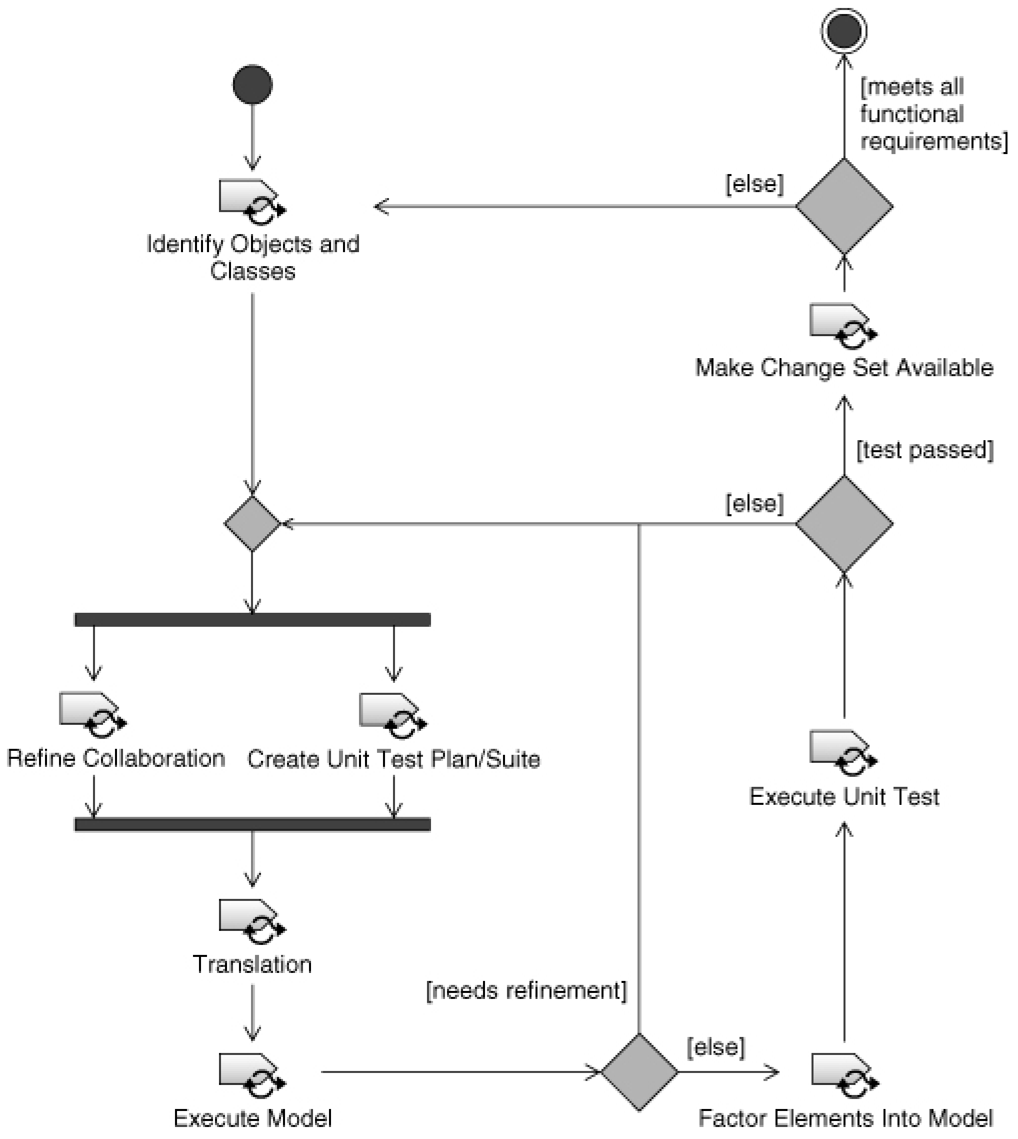
° SP 3.1 Establish configuration management records.

° SP 3.2 Perform configuration audits.

The Harmony/ESW process realizes the specific goals and practices of this process area through continuous integration (Chapter 5). Figure B.7 shows how the continuous integration activity executes in parallel with the analysis and design activities. This activity receives updates to work products on a highly frequent basis, evaluates them for quality, and ensures that the new products do not “break the build” of the software. If these tests are passed, the baseline is updated and made available to the project team as a whole. The details of this activity are shown in Figure B.5.

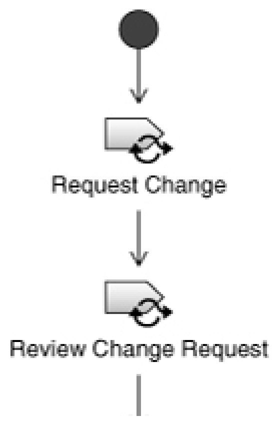
In parallel with the continuous integration activity, the development activities perform frequent (more than hourly) unit testing and submit updated and tested changes to the baseline at least once per day. For example, in the object analysis activity (Chapter 6; see Figure B.6), you can see the “Make change set available” task that releases unit-tested versions of the software to the configuration manager for evaluation and possible inclusion in the baseline. The inner loops in the object analysis workflow are referred to as the *nanocycle* of development and typically execute in 20 to 60 minutes.

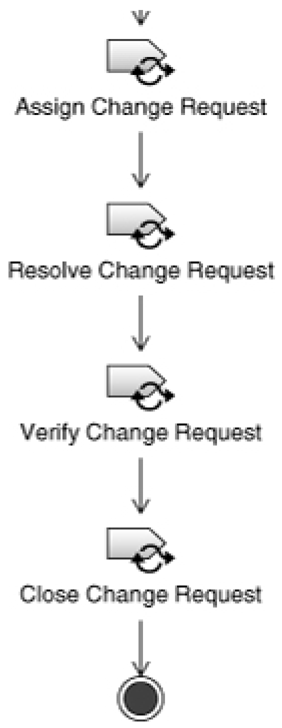
Figure B.6 *Object analysis tasks*


[meets all functional requirements]
[else]
Identify Objects and Classes
Make Change Set Available
[test passed]
[else]
Refine Collaboration
Create Unit Test Plan/Suite
Execute Unit Test
Translation
[needs refinement]
[else]
Execute Model
Factor Elements Into Model

Change requests are managed through a "Manage change" activity (Chapter 9). This activity executes in parallel with the development activities (see Figure B.7). The detailed task breakdown of this activity is shown in Figure B.7.

Figure B.7 *"Manage change" tasks*





Measurement and Analysis (MA)

The purpose of this process area is to measure appropriate indicators of project status to support project management. This activity has two specific goals and eight specific practices:

• SG 1 Align measurement and analysis activities:

° SP 1.1 Establish measurement objectives.

° SP 1.2 Specify measures.

° SP 1.3 Specify data collection and storage procedures.

° SP 1.4 Specify analysis procedures.

• SG 2 Provide measurement results:

° SP 2.1 Collect measurement data.

° SP 2.2 Analyze measurement data.

° SP 2.3 Store data and results.

° SP 2.4 Communicate results.

The Harmony/ESW process has two primary activities that address these process goals. The first is the "Control project" activity (Chapter 9; see Figure B.3) in which progress against schedule and risk is constantly monitored. The second is in the increment review (party phase) (Chapter 9) at the end of each spiral in which analysis is performed to support management replanning.

Process and Product Quality Assurance (PPQA)

This process area provides personnel, including management, with objective information related to the quality of various work products. It has two specific goals and four specific practices:

• SG 1 Objectively evaluate processes and work products:

° SP 1.1 Objectively evaluate processes.

° SP 1.2 Objectively evaluate work products and services.

• SG 2 Provide objective insight:

° SP 2.1 Communicate and ensure resolution of noncompliance issues.

° SP 2.2 Establish records.

The most visible point at which PPQA is addressed is within the "Perform model review" activity (Chapter 9). In this task, significant software work products—notably aspects of the UML or SysML model, the source code, unit test plans, procedures, tests, and documentation —are reviewed and assessed. It is common in high-reliability environments to include reviews:

• after prototype definition (Chapter 6), to assess the requirements artifacts (e.g., use case model, use cases, scenarios, use case state machines, constraints, requirements elements).

• after object analysis (Chapter 6), to assess the quality and reusability of the object analysis model (aka the PIM), including the relevant work products (class diagrams, classes, scenarios, state machines, unit test plans, cases, and procedures).

• after architectural design (Chapter 7), to assess the quality, performance, scalability, and applicability of the architecture, including architectural work products (e.g., class diagrams and related elements), around the five primary views of architecture (subsystem and component view, concurrency and resource view, distribution view, safety and reliability view, and deployment view).

• after the “Prepare for validation testing” activity (Chapter 8), to assess the quality, completeness, and thoroughness of the validation test set.

Process evaluation occurs within the increment review (party phase) (Chapter 9) as an explicit step in the task.

The PPQA activities can result in changes being put into the “Manage control” activity (Chapter 9; see Figure B.7).

Maturity Level 3

At maturity level 3, the process by which development occurs is well defined. In addition to the process areas addressed by maturity level 2, the key areas addressed by maturity level 3 are PI, RD, TS, VAL, VER, OPD, OPF, OT, IPM, RSKM, and DAR. Harmony/ESW provides guidance for all of these process areas.

Product Integration (PI)

The purpose of this process area is to assemble the product from its various pieces (normally configuration items), ensure that the integrated product functions properly, and deliver the product. This process area has three specific goals and nine specific practices:

• SG 1 Prepare for product integration:

° SP 1.1 Determine integration sequence.

° SP 1.2 Establish the product integration environment.

° SP 1.3 Establish product integration procedures and criteria.

• SG 2 Ensure interface compatibility:

° SP 2.1 Review interface descriptions for completeness.

° SP 2.2 Manage interfaces.

• SG 3 Assemble product components and deliver the product:

° SP 3.1 Confirm readiness of product components for integration.

° SP 3.2 Assemble product components.

° SP 3.3 Evaluate assembled product components.

° SP 3.4 Package and deliver the product or product components.

The Harmony/ESW process addresses these concerns with *continuous integration* (Chapter 5). During development work, the developers continually evolve and elaborate their work products (notably models and code, but this can also include other work products such as hazard analyses). Frequently (at least daily), the developers unit-test and deliver the work products to integration in a task called "Make change set available" (Chapters 6 and 7). Once these have been delivered to the configuration manager (who also functions as the integrator), he or she tests them in conjunction with the other parts of the system to ensure that the system integrates and works properly. If the tests are passed, then the set of integrated items updates the evolving product baseline and the baseline is made available to the project team. In parallel with these activities, the configuration manager adds new integration tests as functionality of the system evolves and grows (see Figure B.5).

This approach is superior to traditional "big bang" integration-at-the-end approaches because it uncovers integration defects far earlier, when they are much less expensive to fix.

Requirements Development (RD)

This process area focuses on the production and analysis of requirements at three levels of concern: customer, product, and product component. It has 3 specific goals and 10 specific practices.

• SG 1 Develop customer requirements:

° SP 1.1 Elicit needs.

° SP 1.2 Develop the customer requirements.

• SG 2 Develop product requirements:

° SP 2.1 Establish product and product component requirements.

° SP 2.2 Allocate product component requirements.

° SP 2.3 Identify interface requirements.

• SG 3 Analyze and validate requirements:

° SP 3.1 Establish operational concepts and scenarios.

° SP 3.2 Establish a definition of required functionality.

° SP 3.3 Analyze requirements.

° SP 3.4 Analyze requirements to achieve balance.

° SP 3.5 Validate requirements.

The Harmony/ESW process provides very strong guidance in the development of requirements at all three levels of concern. Requirements are developed and analyzed with a combination of textual and model (e.g., use case) specifications. Prior to beginning the software development, and in parallel with prespiral planning, the activity “Develop stakeholder requirements” (Chapter 5) is performed. This activity focuses on customer requirements and creates both vision and stakeholder requirements documents. In the microcycle, the first activity is the prototype definition activity (Chapter 6); in this activity, the system requirements specification (along with subsystem specifications, as necessary) is produced along with use cases, sequence diagrams, state machines, and constraints (refer to Figure B.2). Because the microcycle is an iterated activity, the system requirements are developed incrementally, organized around the system use cases.

Technical Solution (TS)

This process area’s purpose is to design, develop, and implement solutions to the requirements. There are three specific goals and eight specific practices:

• SG 1 Select product component solutions:

° SP 1.1 Develop alternative solutions and selection criteria.

° SP 1.2 Select product component solutions.

• SG 2 Develop the design:

° SP 2.1 Design the product or product component.

° SP 2.2 Establish a technical data package.

° SP 2.3 Design interfaces using criteria.

° SP 2.4 Perform make, buy, or reuse analyses.

• SG 3 Implement the product design:

° SP 3.1 Implement the design.

° SP 3.2 Develop product support documentation.

This is, if anything, Harmony/ESW's strongest aspect—the analysis and design of software and systems. Using a combination of model-driven development (MDD) and agile methods, Harmony has very strong guidance on how and when to develop various work products, resulting in extremely high-quality products.

The Harmony/ESW process does this work in the microcycle workflow and breaks up the activities into object analysis (Chapter 6; refer to Figure B.6) for the creation of an "analysis model"—aka PIM in the Model-Driven Architecture (MDA) standard of the Object Management Group (OMG). This analysis model meets all of the functional requirements allocated to its current (and previous) microcycles. Design optimizations occur in the following three activities: architectural design (large-scale, or strategic focus; see Chapter 7 and Figure B.8), mechanistic design (middle-scale, or collaboration focus; see Chapter 7 and Figure B.9), and detailed design (small-scale or class focus; see Chapter 7 and Figure B.10). In all cases, design is a design-pattern-centric approach in which optimization is achieved through five key steps:

1. Identify the design optimization criteria of concern.

2. Rank the criteria in order of criticality.

3. Identify design patterns and technical solutions that optimize the most important criteria at the expense of the least important.

4. Apply the design patterns and technical solutions.

5. Validate the solution:

a. Ensure that the previously working functionality isn't broken.

b. Ensure that you have achieved your optimization goals.

Figure B.8 *Architectural design tasks*

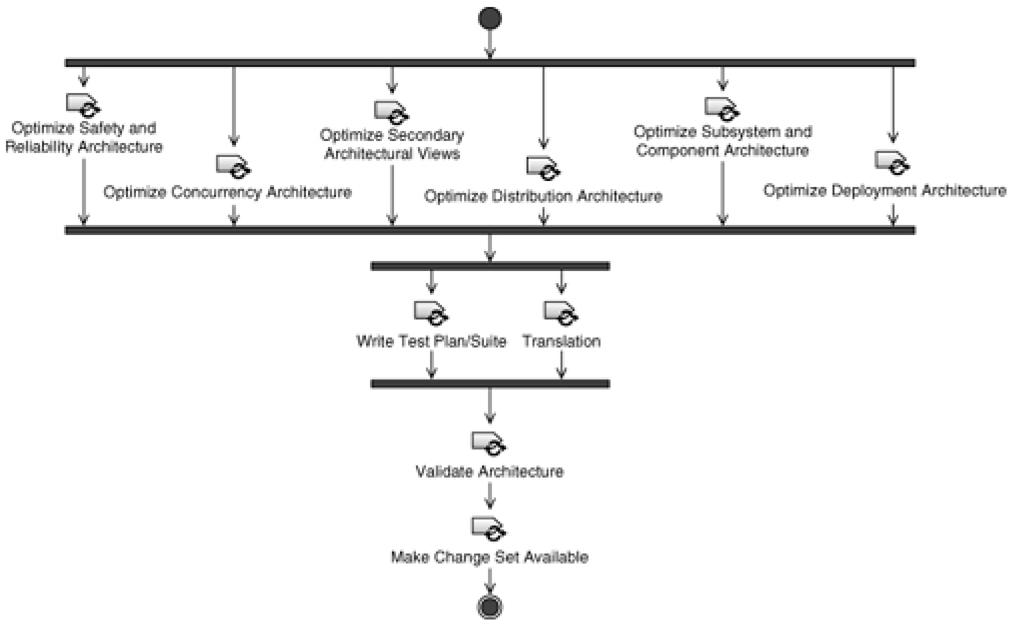

Optimize Safety and Reliability Architecture
Optimize Concurrency Architecture
Optimize Secondary Architectural Views
Optimize Distribution Architecture
Optimize Subsystem and Component Architecture
Optimize Deployment Architecture
Write Test Plan/Suite
Translation
Validate Architecture
Make Change Set Available

Figure B.9 *Mechanistic design tasks*

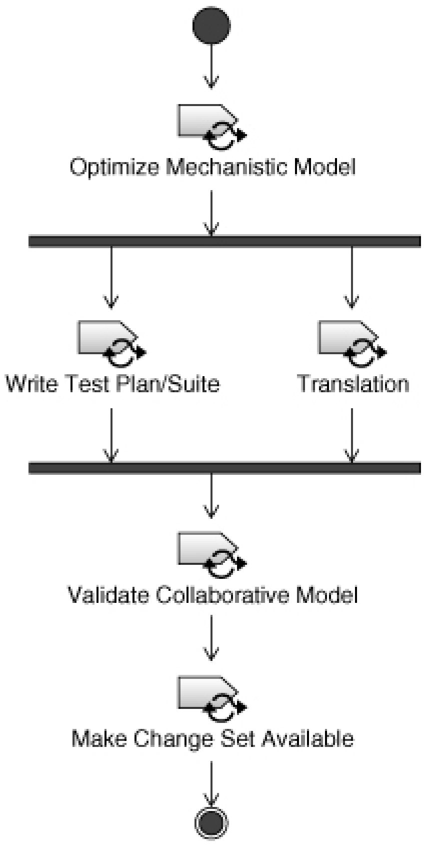
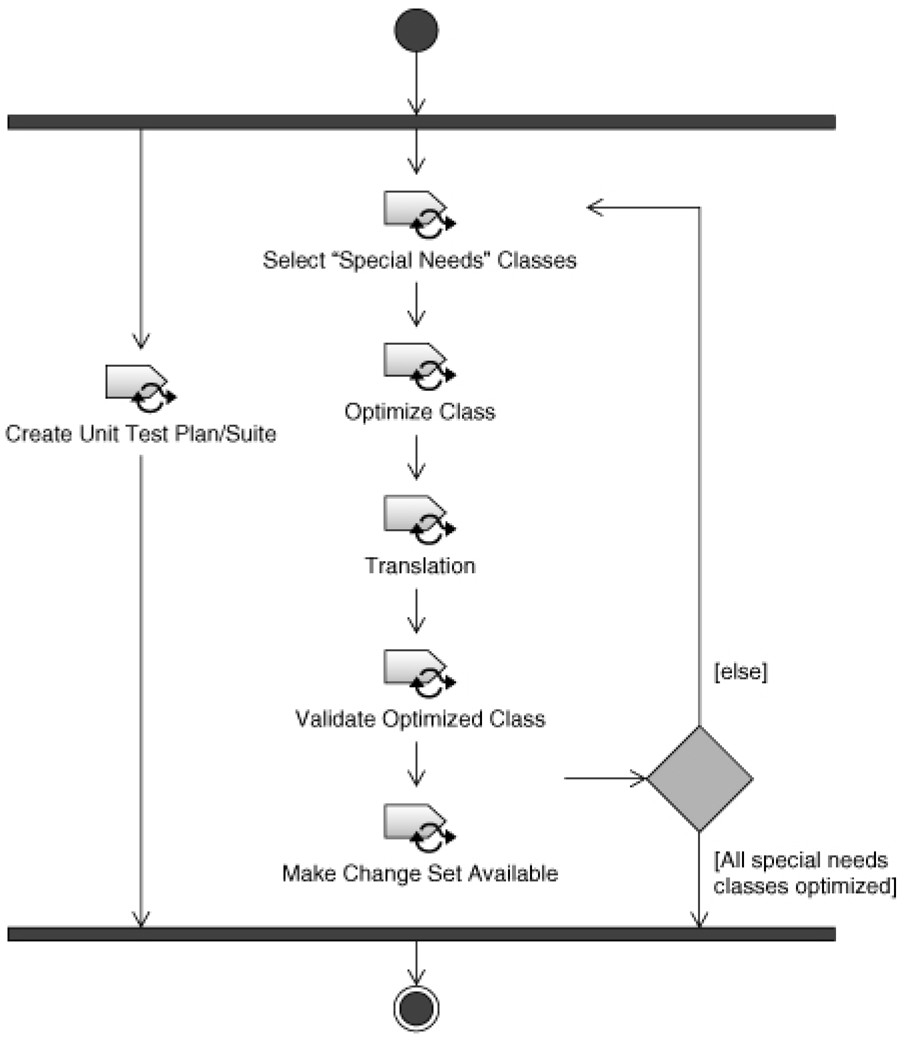

Optimize Mechanistic Model
Write Test Plan/Suite
Translation
Validate Collaborative Model
Make Change Set Available

Figure B.10 *Detailed design tasks*


Select "Special Needs" Classes
Create Unit Test Plan/Suite
Optimize Class
Translation
Validate Optimized Class
Make Change Set Available
[else]
[All special needs classes optimized]

In all activities, from object analysis to detailed design, source code is generated and unit-tested, so there is no downstream "implementation" activity. This ensures that the analysis and design are correct, because the code is immediately generated and tested.

Validation (VAL)

The purpose of this process area is to demonstrate that the product meets the user need (stating that need is the purpose of the stakeholder requirements) for use once placed into its intended environment. This process area has two specific goals and five specific practices:

• SG 1 Prepare for validation:

° SP 1.1 Select products for validation.

° SP 1.2 Establish the validation environment.

° SP 1.3 Establish validation procedures and criteria.

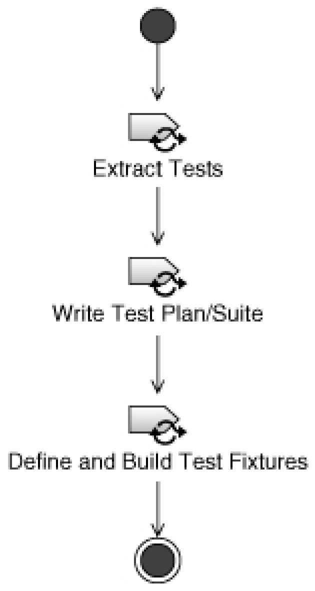
• SG 2 Validate product or product components:

° SP 2.1 Perform validation.

° SP 2.2 Analyze validation results.

The Harmony/ESW process achieves these goals primarily through two activities (see Figure B.11). It is important to note that the process includes creating traceability links between the vision, stakeholder requirements, system (or software) requirements, and various use case models.

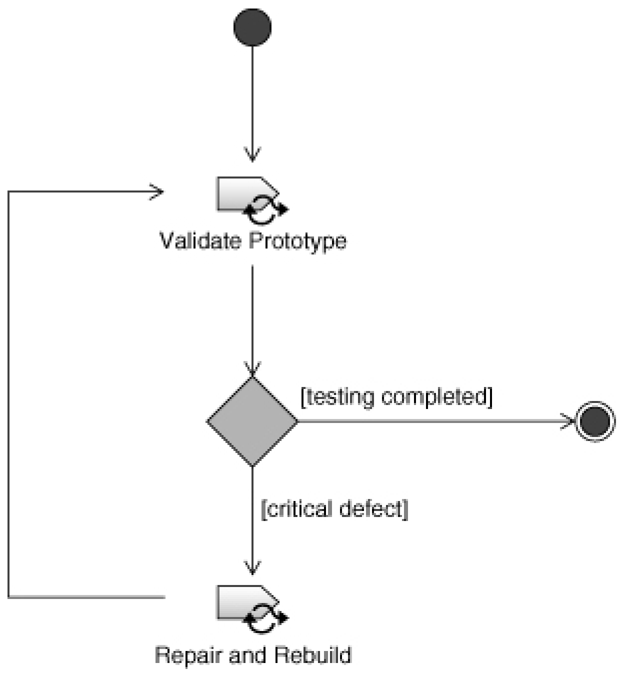
Figure B.11 *"Prepare for validation" tasks*



The "Prepare for validation" activity (Chapter 8) directly addresses specific goal 1 (see Figure B.11). The test case identification is done in the first task; this is greatly simplified because the use case scenarios identified earlier in the prototype definition activity (Chapter 6; see Figure B.2) form the specification of the functional and quality-of-service test cases. The test cases define the validation criteria for the product. The tester often defines additional test cases as well. The last task, "Define and build test fixtures," defines and constructs the validation environment.

The second specific goal is addressed through the validation activity (Chapter 8; see Figure B.12). In this activity, test cases are applied. If a critical defect is found, that defect is repaired immediately. Noncritical defects are logged into the change management system and are scheduled for repair in subsequent microcycles.

Figure B.12 *Validation*



Verification[7] (VER)

7. Note: The Harmony/ESW process performs verification and validation in the same set of tasks.

The purpose of this process area is to ensure that the selected work products meet their specific requirements (e.g., system or software requirements). This process area has three specific goals and eight specific practices:

• SG 1 Prepare for verification:

° SP 1.1 Select work products for verification.

° SP 1.2 Establish the verification environment.

° SP 1.3 Establish verification procedures and criteria.

• SG 2 Perform peer reviews:

° SP 2.1 Prepare for peer reviews.

° SP 2.2 Conduct peer reviews.

° SP 2.3 Analyze peer review data.

• SG 3 Verify selected work products:

° SP 3.1 Perform verification.

° SP 3.2 Analyze verification results.

The Harmony/ESW process achieves these goals through a combination of its validation procedures (discussed in the previous section), model and requirements document reviews, and well-defined traceability between the various requirements and development artifacts.

Organizational Process Definition (OPD)

This process area seeks to establish and maintain a usable set of organization process assets and work environment standards. This process has one specific goal (two for integrated product [IP] teams) and six specific practices (nine for integrated product teams):

• SG 1 Establish organizational process assets:

° SP 1.1 Establish standard processes.

° SP 1.2 Establish lifecycle model descriptions.

° SP 1.3 Establish tailoring criteria and guidelines.

° SP 1.4 Establish the organization’s measurement repository.

° SP 1.5 Establish the organization’s process asset library.

° SP 1.6 Establish work environment standards.

• SG 2 Enable IPPD management (IPPD addition):

° SP 2.1 Establish empowerment mechanisms.

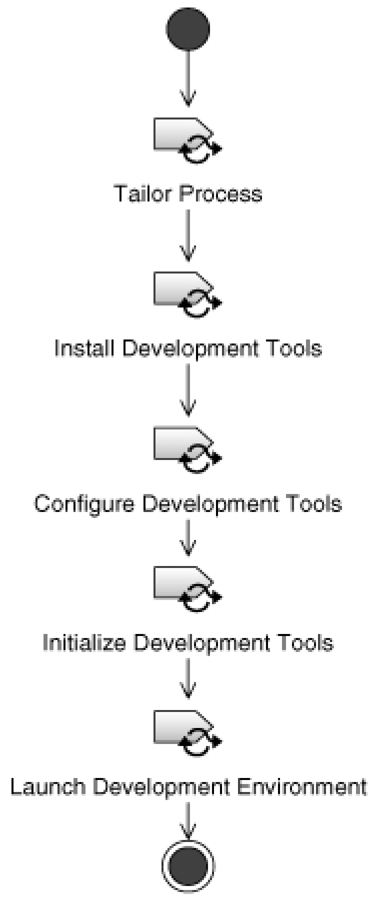
° SP 2.2 Establish rules and guidelines for integrated teams.

° SP 2.3 Balance team and home organization responsibilities.

The Harmony/ESW process supports these by providing customizable process assets in a format compatible with the Rational Method Composer (RMC) commercial tool as well as the Eclipse Process Framework (EPF) Composer tool, an open-source process authoring and tailoring environment available at www.eclipse.com/epf. The figures in this appendix are screen shots taken from the process documentation resulting from publishing that process content.

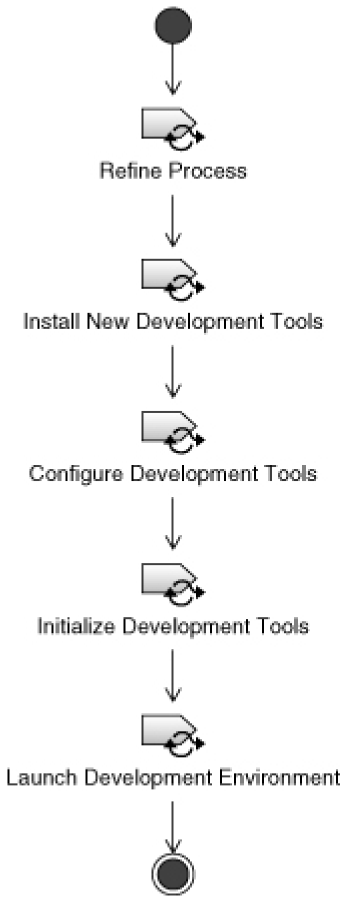
In addition, the process itself contains activities and tasks directly addressing the tailoring of the development environment, including processes. The "Define and deploy the development environment" activity (Chapter 5) occurs before development activities (see Figure B.13) and includes process tailoring (see Figure B.13).

Figure B.13 *"Define and deploy the development environment" activity*


Tailor Process
Install Development Tools
Configure Development Tools
Initialize Development Tools
Launch Development Environment

Additionally, the "Control project" activity (Chapter 9; refer to Figure B.3) includes a "Refine and deploy the development environment" activity (Chapter 5) to continuously optimize the development environment, including the processes (see Figure B.14). Last, the increment review (Chapter 9) periodically analyzes the project, including the development processes, for optimality and enters change requests when improvements to the process are necessary or desirable.

Figure B.14 *“Refine and deploy the development environment” activity*


Refine Process
Install New Development Tools
Configure Development Tools
Initialize Development Tools
Launch Development Environment

Organizational Process Focus (OPF)

This process area’s purpose is to plan, implement, and deploy organization process improvements based on an understanding of the strengths and weaknesses of the current processes. This process area has three specific goals and nine specific practices:

• SG 1 Determine process improvement opportunities:

° SP 1.1 Establish organizational process needs.

° SP 1.2 Appraise the organization’s processes.

° SP 1.3 Identify the organization’s process improvements.

• SG 2 Plan and implement process improvements:

° SP 2.1 Establish process action plans.

° SP 2.2 Implement process action plans.

• SG 3 Deploy organizational process assets and incorporate lessons learned:

° SP 3.1 Deploy organizational process assets.

° SP 3.2 Deploy standard processes.

° SP 3.3 Monitor implementation.

° SP 3.4 Incorporate process-related experiences into the organizational process assets.

The Harmony/ESW process is primarily focused on project needs and has a project-oriented point of view. However, Harmony/ESW has been deployed as an organizational standard and has been extended to broaden the use of its process assets to multiple projects within an organization. The same techniques and methods identified in the previous process area (organizational process definition) apply to this process area as well.

Organizational Training (OT)

The purpose of this process area is to develop skills and knowledge of personnel so that they can perform their roles effectively. This process area has two specific goals and seven specific practices:

• SG 1 Establish an organizational training capability:

° SP 1.1 Establish the strategic training needs.

° SP 1.2 Determine which training needs are the responsibility of the organization.

° SP 1.3 Establish an organizational training tactical plan.

° SP 1.4 Establish training capability.

• SG 2 Provide necessary training:

° SP 2.1 Deliver training.

° SP 2.2 Establish training records.

° SP 2.3 Assess training effectiveness.

Currently, the Harmony/ESW process does not include tasks that address these specific goals, but later revisions of the process will include them. As a practical matter, during the prespiral planning activities (Chapter 5; refer to Figure B.4), the identification of training needs and planning to address them always occurs. In addition, as a result of the project analysis in the increment review task, training needs may be identified so that during the "Control project" activity (Chapter 9), training can be planned for and scheduled. It should be noted that the "Launch development environment" task of the "Define and deploy the development environment" activity (Chapter 5; see Figure B.13) specifically calls for process training.

Integrated Project Management (IPM)

The purpose of this process area is to establish and manage the project and the involvement of stakeholders. This process area has 2 specific goals (3 for integrated product teams) and 9 specific practices (14 for integrated product teams):

• SG 1 Use the project's defined process:

° SP 1.1 Establish the project's defined process.

° SP 1.2 Use organizational process assets for planning project activities.

° SP 1.3 Establish the project's work environment.

° SP 1.4 Integrate plans.

° SP 1.5 Manage the project using the integrated plans.

° SP 1.6 Contribute to the organizational process assets.

• SG 2 Coordinate and collaborate with relevant stakeholders:

° SP 2.1 Manage stakeholder involvement.

° SP 2.2 Manage dependencies.

° SP 2.3 Resolve coordination issues.

• SG 3 Apply IPPD principles (IPPD addition):

° SP 3.1 Establish the project's shared vision.

° SP 3.2 Establish the integrated team structure.

° SP 3.3 Allocate requirements to integrated teams.

° SP 3.4 Establish integrated teams.

° SP 3.5 Ensure collaboration among interfacing teams.

This process area is another of the strengths of the Harmony/ESW process. Harmony/ESW has significant guidance on how to plan, schedule, and track projects. For example, the prespiral planning activity (Chapter 5; refer to Figure B.4) details how to create schedules, project teams, model/system organizational units, and risk and reuse plans. The "Control project" activity (Chapter 9; refer to Figure B.3) tracks project performance against plan and updates the development environment, schedule, and risk plan. Coordination with the stakeholders is done largely through the vision statement and stakeholder requirements documents and through reviews such as a preliminary design review (PDR) and critical design review (CDR), which occur at the end of specified microcycles. Further, because the product is incrementally developed, early releases of the system can be deployed to the customer for early feedback.

Risk Management (RSKM)

The purpose of this process area is to identify and address potential project problems before they manifest themselves. This includes the identification of risk mitigation activities to resolve potential problems as early as possible. This process area includes three specific goals and seven specific practices:

• SG 1 Prepare for risk management:

° SP 1.1 Determine risk sources and categories.

° SP 1.2 Define risk parameters.

° SP 1.3 Establish a risk management strategy.

• SG 2 Identify and analyze risks:

° SP 2.1 Identify risks.

° SP 2.2 Evaluate, categorize, and prioritize risks.

• SG 3 Mitigate risks:

° SP 3.1 Develop risk mitigation plans.

° SP 3.2 Implement risk mitigation plans.

Harmony/ESW addresses the concerns of this process area through explicit risk management tasks that appear in three primary activities. Prespiral planning (Chapter 9; refer to Figure B.4) includes a “Plan for risk reduction” task that includes the following steps:

1. Identify key project hazards.

2. Determine the likelihood of key project hazards.

3. Compute key project risks.

4. Rank project risks.

5. Specify risk mitigation activities for key project risks.

6. Write the risk mitigation plan.

These risk mitigation activities are allocated to appropriate microcycles and resources in the schedule.

While the product development is under way, the “Control project” activity (Chapter 9; refer to Figure B.3) runs in parallel; this activity includes a specific “Update risks” task that analyzes the results of the risk mitigation activities and updates the risk management and other plans appropriately. In addition, the increment review (Chapter 9), which is performed at the end of each microcycle, examines the risk mitigation activities and risk management plan.

Decision Analysis and Resolution (DAR)

The purpose of this process area is to analyze decisions using a formal evaluation process that evaluates alternatives using established criteria. This process area has one specific goal and six specific practices:

• SG 1 Evaluate alternatives:

° SP 1.1 Establish guidelines for decision analysis.

° SP 1.2 Establish evaluation criteria.

° SP 1.3 Identify alternative solutions.

° SP 1.4 Select evaluation methods.

° SP 1.5 Evaluate alternatives.

° SP 1.6 Select solutions.

The Harmony/ESW process uses a specific workflow for evaluating alternatives. This process is primarily used within the three design phases (Chapter 7; refer to Figure B.9 through Figure B.11) to select an optimal design approach but can be used elsewhere for trade-off analysis. The steps in trade-off analysis are:

1. Identify the design optimization criteria of concern.

2. Rank the criteria in order of criticality.

3. Identify design patterns and technical solutions that optimize the most important criteria at the expense of the least important.

4. Apply the design patterns and technical solutions.

5. Validate the solution:

a. Ensure that the previously working functionality isn't broken.

b. Ensure that you have achieved your optimization goals.

Maturity Level 4

This level of maturity is characterized by quantitative management. It includes the process areas, goals, and practices of maturity level 3 and adds the gathering of quantitative measures which are then analyzed using quantitative techniques such as (but not limited to) statistical analysis. This level of maturity includes two additional process areas: OPP and QPM.

Organizational Process Performance (OPP)

This process area's purpose is to create and maintain a quantitative measure of the performance of the processes, especially in relation to quality, repeatability, and schedule accuracy. This process area has one specific goal and five specific practices:

• SG 1 Establish performance baselines and models:

° SP 1.1 Select processes.

° SP 1.2 Establish process-performance measures.

° SP 1.3 Establish quality and process-performance objectives.

° SP 1.4 Establish process-performance baselines.

° SP 1.5 Establish process-performance models.

The Harmony/ESW process provides guidance on creating and tracking projects, primarily through the work product known as the schedule. This work product identifies when worker tasks are to start, their estimated effort and completion dates, and their allocation to personnel. The tasks that create, update, and monitor the project work occur in the prespiral planning (Chapter 5; refer to Figure B.4), "Control project" (Chapter 9; refer to Figure B.3), and increment review (Chapter 9) activities. In addition, defects are tracked via the "Manage change" activity (Chapter 9; refer to Figure B.7).

For estimation, the Harmony/ESW process provides guidance on using use case points (Chapter 5), a volume-times-complexity measure somewhat similar to COCOMO and VSSES (Very Simple Software Estimation System[8]). Guidance is provided on tuning the approach for scheduling and estimation based on measured efficiency. The process includes a BERT (Bruce's Evaluation and Review Technique) workflow for constructing estimates of work tasks, and an ERNIE (Effect Review of Nanocycle Iteration Estimation) workflow for adjusting tuning criteria based on actual results against plan.

8. Lawrence H. Putnam and Ware Meyers, *Measures for Excellence: Reliable Software On Time, Within Budget* (Englewood Cliffs, NJ: Prentice Hall, 1991).

Quantitative Project Management (QPM)

The purpose of this process area is to use quantitative data to manage the project. This process area has two specific goals and eight specific practices:

• SG 1 Quantitatively manage the project:

° SP 1.1 Establish the project’s objectives.

° SP 1.2 Compose the defined process.

° SP 1.3 Select the subprocesses that will be statistically managed.

° SP 1.4 Manage project performance.

• SG 2 Statistically manage subprocess performance:

° SP 2.1 Select measures and analytic techniques.

° SP 2.2 Apply statistical methods to understand variation.

° SP 2.3 Monitor performance of the selected subprocesses.

° SP 2.4 Record statistical management data.

While the Harmony/ESW process does not define specific tasks to perform statistical quantitative management, it does gather the quantitative information necessary to perform such tasks. This includes performance measures such as effort expended and project velocity (a measure of work items performed per unit time) as well as quality measures such as defect counts.

Maturity Level 5

This level of maturity is characterized as an “optimizing process.” This means that it is quantitatively managed (since it includes the process areas, goals, and practices of maturity level 4) and uses the results of this analysis to continually improve the process over time. This maturity level adds the process areas OID and CAR.

Organizational Innovation and Deployment (OID)

The purpose of this process area is to select and deploy incremental process improvements to support the organization’s quality and process-performance objectives. This process area includes two specific goals and seven specific practices:

• SG 1 Select improvements:

° SP 1.1 Collect and analyze improvement proposals.

° SP 1.2 Identify and analyze innovations.

° SP 1.3 Pilot improvements.

° SP 1.4 Select improvements for deployment.

• SG 2 Deploy improvements:

° SP 2.1 Plan the deployment.

° SP 2.2 Manage the deployment.

° SP 2.3 Measure improvement effects.

The Harmony/ESW process supports this process area in three primary activities. The increment review (Chapter 9) is performed each microcycle. One of its responsibilities is to identify process gaps and areas in which process improvement is required, and another is to measure how effective such changes have been. Based on this, the “Define and deploy the development environment” (Chapter 5; refer to Figure B.13) and the “Refine and deploy the development environment” (Chapter 5; refer to Figure B.14) activities update the process content and specify how the updated processes are deployed to the project team and any training needs for the team.

Causal Analysis and Resolution (CAR)

The purposes of this process area are to identify the root causes of defects and other project concerns and to specify methods and techniques to prevent their recurrence. This process area includes two specific goals and five specific practices:

• SG 1 Determine causes of defects:

° SP 1.1 Select defect data for analysis.

° SP 1.2 Analyze causes.

• SG 2 Address causes of defects:

° SP 2.1 Implement the action proposals.

° SP 2.2 Evaluate the effect of changes.

◦ SP 2.3 Record data.

The primary activity in the Harmony/ESW process that addresses the concerns of this process area is the increment review (Chapter 9) that occurs at the end of each microcycle. This task has the responsibility to analyze project aspects—including product quality, process efficiency, risk reduction, and architectural scalability—and recommend changes in process aspects (role responsibilities, workflows, task steps, and work products) to improve the process efficiency and product quality. In addition, the model review task (Chapter 9) allows quality assurance personnel, architects, developers, and other stakeholders to assess quality and adherence to standards. Also, testing is applied more or less continuously throughout the process. Unit testing is performed in the object analysis (Chapter 6), architectural design (Chapter 7), mechanistic design (Chapter 7), and detailed design (Chapter 7) activities. Integration testing is performed continuously as the product is being developed. Validation testing (Chapter 8) occurs at the end of each microcycle, typically every four to six weeks.

Summary

In this appendix, I have introduced the basics of both the Harmony/ESW process and the CMMI standard. The CMMI standard identifies five levels of maturity (see Table B.2): initial, managed, defined, quantified, and optimizing. Each level has a set of specific goals it addresses and specific practices that it recommends to achieve those goals. The CMMI standard does not, however, specify a development process to be used. Each level includes the goals and practices of the levels below it. To achieve compliance at a specific level of maturity, a process must demonstrate that it adequately addresses the goals of that level.

The bulk of the appendix addressed, in some detail, how each specific goal for each of the process areas is addressed by the Harmony/ESW process. The Harmony/ESW process is a model-driven, agile development approach that emphasizes quality, adherence to requirements, well-structured and specified architectures, and constant execution and integration.

It is possible to achieve any level of maturity with the Harmony/ESW process. The process itself provides the means for tailoring the process to enable it to be used in different organizations and markets. The process assets are stored in an open-source tool—EPF Composer—that provides a powerful process authoring and customization environment. Once tailored, the process can be published as a Web site hosted in the project team environment, providing strong and immediate guidance on process workflows, tasks, roles, and work products.

Further Reading

Ahern, Dennis M., Aaron Clouse, and Richard Turner. *CMMI Distilled: A Practical Introduction to Integrated Process Improvement*. Boston: Addison-Wesley, 2003.

Douglass, Bruce Powel, Ph.D. *Real-Time UML: Advances in the UML for Embedded Systems, Third Edition*. Boston: Addison-Wesley, 2004.

———. *Real-Time UML Workshop for Embedded Systems*. Burlington, MA: Elsevier Press, 2006.

Konrad, Mike, and James W. Over. “Agile CMMI: No Oxymoron,” *Dr. Dobbs* (2005). www.ddj.com/architect/184415287.

Software Process Engineering Meta-Model, Version 2.0, Object Management Group, 2008. www.omg.org/technology/documents/formal/spem.htm.

Glossary

accident—a loss of some kind, such as personal injury, death, damage to equipment, or loss of money.

action—a behavioral primitive statement that a system executes.

action language—language in which actions are written. This may be a concrete language (such as C, Java, or Var'aq) or an abstract action language that requires translation to a concrete language for execution.

`<<active>>` **class**—the primary unit of concurrency in the UML. An `<<active>>` class owns an OS thread in which it and its nested parts execute.

activity model—a model of the flow of control behavior of an element.

actor—an object outside the scope of a system that has interesting interactions with that system.

agile methods—a lightweight, minimal-ceremony approach to software and system development emphasizing product quality, meeting customer needs, team collaboration, and responsiveness to change.

Agilista—a practitioner of agile methods.

analysis—the process of identifying the essential characteristics of a system or element.

analysis model—the coherent collection of essential elements of a system or element.

architectural design—the process of defining a system architecture.

architecture—the specification of the largest-scale design optimization decisions for a system. This is divided into five primary views: subsystem and component architecture, concurrency and resource architecture, distribution architecture, safety and reliability architecture, and deployment architecture.

asymmetric deployment—a deployment architecture in which the location of software is determined dynamically at runtime.

average-case performance—a specification or measurement of mean execution time.

ballistic planning—planning that assumes no changes to the plan will be necessary in the future.

BIOS—Basic Input/Output System, usually the most basic portion of an OS or OS loader.

BIT—built-in test; normally a set of tests that are executed either under user command or periodically as a system runs.

block (SysML)—a stereotype of class in SysML that has the same semantics as class; a notational convenience.

blocking—a period of time during which a higher-priority concurrency unit that is ready to run is prevented from executing by a lower-priority task, because the lower-priority task owns a necessary resource. See **priority inversion.**

build—an operational version of a compiled system that demonstrates some subset of the system functionality.

Capability Pattern—a reusable process element; also known as a workflow.

change control board (CCB)—a central authority for the management of change requests.

change management—the process of managing and controlling change requests during system development or maintenance.

channel—a kind of "end-to-end" subsystem that takes input from sensors and processes it through a series of data transformational steps into control of actuators; used often in safety-critical and high-reliability architectures, such as triple modular redundancy (TMR).

class—the specification of an object instance; an encapsulation boundary including data (attributes) and behavior (operations that manipulate that data).

CMMI—Capability Maturity Model Integration, a standard owned by the Software Engineering Institute (SEI).

code—source code, usually as expressed in a concrete action language.

collaboration—a set of object roles working together to realize a use case.

component—a metasubtype of class; a large-scale class that contains internal parts and provides services via a well-defined interface. See **subsystem.**

computation-independent model (CIM)—in the Harmony/ESW process, the use case model containing use cases, scenarios, activity models, and state machine specifying black-box functionality and performance.

concurrency—the simultaneous or pseudo-simultaneous execution of multiple action sequences.

concurrency unit—an action sequence in which the sequence is fully known; implemented by operating systems as a task, thread, or process.

configuration management (CM)—the process of managing versions of configuration items and their coherent consistent sets, in order to control their modification and release, and to ensure their consistency, completeness, and accuracy.

constraint—a user-defined "well-formedness" rule that applies to one or more elements. For example, a deadline is usually expressed as a constraint.

continuous integration—the process of integrating different configuration items into a coherent baseline on a continuous or highly frequent basis.

CORBA—Common Object Request Broker Architecture, an OMG middleware standard.

CRC cards—**C**lass, **R**esponsibility, and **C**ollaboration cards, an approach for class identification championed by Rebecca Wirfs-Brock.

critical design review (CDR)—in a traditional waterfall process, a review of the critical architectural elements, meant to ensure the validity of subsequent work. In the Harmony/ESW process, the CDR is optional but can be performed at the end of a microcycle in which the primary architectural concerns have stabilized.

criticality—importance against some measure.

CWM—Common Warehouse Metamodel, an OMG standard primarily concerned with metadata management and data transformations.

DDS—Data Distribution Service, a middleware standard.

deadline—the point in time at which the execution of an action becomes irrelevant or incorrect.

Deadline Monotonic Analysis (DMA)—a schedulability analysis technique that analyzes whether a set of tasks can be guaranteed to meet their deadlines.

debugging—the informal process of execution of software for the purpose of probing for errors.

delivery process—the collection of capability patterns and practices into an end-to-end process.

deployment architecture—the identification of the responsibilities and interfaces between different engineering disciplines of a system, such as software, digital electronics, analog electronics, mechanical engineering, optics, and chemical engineering.

design—the process of optimizing an analysis model through the selection of design technologies, decisions, and patterns.

design cost—the cost of developing a system design.

design idioms—small-scale design patterns, usually applied at the class, function, and data structure level.

design model—the result of evolving the analysis model with specific design solutions, technologies, and patterns to achieve functionally correct and optimal execution.

design pattern—a generalized solution to a commonly occurring problem. Design patterns are characterized by their name, applicability, scope, structure, behavior, and consequences.

detailed design—the process of optimizing a system at the small scale, i.e., individual classes, functions, and data structures.

deterministic—a computation situation in which the execution time of an action sequence is known precisely.

diagram—a graphical view; in this context, a coherent view of some set of semantic elements within a model.

distribution architecture—the architectural view focused on the location of runtime instances into multiple address spaces and the rules by which they interact.

DoDAF—Department of Defense Architecture Framework, a DoD standard. The UML representation of DoDAF is specified by the UML Profile for DoDAF and MoDAF (UPDM), an OMG standard.

dynamic planning—the process of creating plans with the expectation and plan to frequently update those plans on the basis of measured success.

Eclipse—a Java-based development and execution environment.

efficiency—a measure of the cost per time or cost per effort.

embedded system—a computer-based system that has a set of dedicated purposes, one of which is *not* to function as a general-purpose computing environment.

EPF Composer—Eclipse Process Framework Composer, an open-source version of the Rational Method Composer used to create and maintain process content. See www.eclipse.org/epf.

error—a systematic fault or mistake.

exception—a condition that violates one or more preconditional invariants; a kind of event sent to indicate such a violation.

execution time—the duration of the execution of an action.

Extreme Programming (XP)—an agile software development approach championed by Kent Beck and others.

F2T2EA Kill Chain—**F**ind, **F**ix, **T**rack, **T**arget, **E**ngage, **A**ssess; a standard approach to characterizing combat system operation.

failure—a stochastic fault.

fairness—a concurrency scheduling doctrine meant to ensure that all concurrency units have an equal or proportional amount of execution time.

fault—a nonconformance of a system to its requirements.

FMEA—failure modes and effects analysis; an approach to analyzing the effect of faults on system reliability.

FMECA—failure mode, effects, and criticality analysis; an approach to analyzing the effects of faults on system reliability and safety.

framework—a coherent architecture that provides an incomplete template for systems within a specific domain; a coherent set of design patterns.

FTA—fault tree analysis; a graphic-based technique for safety analysis.

fuzzy logic—a logic using fuzzy sets, that is, in which elements can have partial set membership.

Gantt chart—a graphical technique used to represent task scheduling, particularly for project management.

GPS—global positioning system.

GUI—graphical user interface.

hard real-time—a system in which correctness is determined by the correct functionality implemented by actions that all meet their deadlines.

Harmony—a family of processes owned by IBM Rational.

Harmony/ESW—the member of the Harmony process family that addresses real-time and embedded software development. Harmony/ESW is an evolution of the Rapid Object-Oriented Process for Embedded Systems (ROPES) authored by Bruce Powel Douglass.

Harmony/Hybrid V—the member of the Harmony process family that addresses both systems engineering and embedded software engineering as a coherent delivery process. It defines a well-specified model-based handoff from systems engineering to software development.

Harmony/SE—the member of the Harmony process family that addresses systems engineering. It is based heavily on the work of Dr. Hans-Peter Hoffmann.

hazard—a condition that leads to an accident.

heterogeneous redundancy—a set of elements with similar functionality but achieved using different means, such as different algorithms, design approaches, or development teams; also known as diverse redundancy. See **channel**.

homogeneous redundancy—a set of elements with identical functionality achieved by replicating the same design elements multiple times. See **channel.**

IDL—interface description language.

increment review (party phase)—a capability pattern in which the work and work products produced during a microcycle are reviewed for the purpose of assessment and process improvement.

incremental development—a process of developing a system as a series of incremental steps.

information assurance—the management of information-related risks.

instance model—a system running in some location at some instant in time.

interface—the specification of the means by which services can be invoked and data can be manipulated across an encapsulation boundary (e.g., class).

interoperability—the ability of diverse systems and organizations to operate together.

IT—information technologies; the infrastructure organization responsible for managing software development and deployment environments.

Jazz—an Eclipse-based technology platform created by IBM to enhance collaborative software development and delivery.

jitter—variation in period, usually given as a range.

Law of Douglass—one of a large set of aphorisms created by, or glommed onto by, Bruce Powel Douglass.

macrocycle—the largest-scale view of time in the Harmony/ESW process; the software project scope timescale.

MARTE—Modeling and Analysis of Real-Time and Embedded Systems, an OMG standard meant to replace the SPT for UML 2.x.

MDA—Model-Driven Architecture, an OMG standard identifying approaches and technologies for large-scale interoperability of systems, emphasizing the use of models for system development.

MDD—model-driven development, the genericized MDA.

measures of effectiveness (MOE)—quantifiable measure of the “goodness” of an approach or technology.

mechanism—the implementation of a collaboration.

mechanistic design—in the Harmony/ESW process, the “middle” level of design which focuses on optimizing collaborations.

metaclass—the “type” of a UML model element. The UML is defined in terms of a metamodel, in which the UML elements are typed by metaclasses, such as class, event, state, and so on.

meta-metamodel—the model used to define the metamodel. In the case of the UML, this is known as the MetaObject Facility (MOF).

metamodel—the model of a language used to develop systems. In the case of UML, the definition of UML itself is the metamodel.

metric—a measurement of some parameter, usually used in the assessment of a technology, approach, or design.

microcycle—the spiral part of the Harmony/ESW process. The microcycle consists of a sequence of activities—prototype definition, object analysis, architectural design, mechanistic design, detailed design, model review, and increment review—that result in the creation of a validated build of a system. A microcycle is usually four to six weeks in duration but can range from one week to several months.

minimum interarrival time—the minimum time between their arrival of aperiodic events of the same type.

mission—a stated purpose, intent, or goal. In the Harmony/ESW process, several work activities have specific missions, including diagram creation and the microcycle.

MoDAF—Ministry of Defence Architecture Framework, a standard similar in purpose and scope to DoDAF but owned by the UK Ministry of Defence.

model—a formal representation of different aspects of a system being constructed in which inessential details are abstracted away. Models in the book are usually captured in UML. See **UML.**

model, executable—a model that is sufficiently well specified so as to allow execution of its semantics.

model-code associativity—a dynamic relation between source code and a model, such that if a change is made to the model the source code automatically changes, and if the source code is changed, then the model is automatically updated.

model repository—the coherent set of data that defines the semantic elements of a user model.

model transformation—the process of applying transformational rules to a model to convert it to another model or form.

MOF—MetaObject Facility, an OMG standard used to define the UML and other standards.

nanocycle—the smallest time frame of consideration in the Harmony/ESW process. This is the time during which analysis or design elements are elaborated, debugged, and unit-tested.

neural network—a collaboration of simple, primitive processing elements that self-organize and self-optimize to achieve computation goals. While these occur in biological systems, in this context we usually mean artificial neural networks such as might be used in optical character recognition applications.

object analysis—in the Harmony/ESW process, the activity by which the analysis model is created.

OMG—Object Management Group, a standards organization. Also, an abbreviation for "Oh, my God," an exclamation frequently used by software developers.

optimization—the process of improving the performance of some element with respect to a set of optimization criteria.

optimization criteria—a set of metrics that measure the performance of an element or set of elements, such as worst-case execution time, reliability, safety, or portability.

PDF—probability density function.

performance—quantitative measures of the execution of some element or set of elements, such as average execution time or memory usage.

period—for regularly recurring events, the length of time between event occurrences. Also see **jitter.**

PERT chart—Project Evaluation and Review Technique; a graphical representation of work tasks and their predecessor and successor relations.

platform-independent model (PIM)—a model of a system that focuses on essential functionality and elides specific solution technologies. See **analysis model**.

platform-specific implementation (PSI)—the source code generated from the PSM. See **code.**

platform-specific model (PSM)—a model of a system that contains essential properties mixed in with optimizing design solutions. See **design model.**

portability—the ability to execute a software system on a different technology platform.

POST—power-on self-test, a set of tests performed when power is applied.

postcondition—a condition guaranteed to be true after the execution of an action.

practice—a set of work tasks performed to realize a specific development intent.

precondition—a condition required to be true before the execution of an action.

preconditional invariant—a precondition that does not vary depending on system dynamics or history.

predictable—the ability to know, in advance, the execution performance of an action, such as its execution duration. In extreme cases, an action is said to be unpredictable (nothing can be stated in advance) or deterministic (the precise performance can be stated in advance).

preliminary design review (PDR)—in a traditional waterfall process, a review of the preliminary architectural concepts, meant to ensure the validity of subsequent work. In the Harmony/ESW process, the PDR is optional but can be performed at the end of a microcycle in which the primary architectural concerns have been addressed.

principle—a basic statement of truth, especially with respect to a goal or intent. The Harmony/ESW process is guided by a set of core principles that are realized by a set of practices.

priority—a scalar value used by an operating system to determine which of a set of ready-to-run tasks should run. Priority can be based on urgency or criticality, or a combination of the two.

priority inversion—a condition in which a lower-priority task is running even though a higher-priority task is waiting to run. This occurs during multitasking preemption scheduling when the lower-priority task owns a resource needed by the higher-priority task. See **blocking.**

profile—a specialized version of a model; in this context, a specialized version of UML used to address a particular market or set of concerns, such as SysML, MARTE, or SPT.

protocol—the forms and ceremony used to manage the interaction of elements.

prototype—a validated build of a system produced at the end of an iteration microcycle. See **build.**

prototype definition—a development activity in the microcycle that elucidates the system requirements, use case model, and microcycle mission statement.

quality—a measure of the acceptability or “goodness” of a system or element.

quality of service (QoS)—a criterion of performance of a service or element, such as the worst-case execution time for an operation.

rate monotonic analysis (RMA)—a set of mathematical analytic techniques used for schedulability analysis of systems.

Rational Unified Process (RUP)—a process owned and popularized by IBM Rational.

real-time—a condition in which timeliness is essential to correctness.

recurring cost—the “cost per shipped item.”

redundancy—the replication of elements used in the identification, isolation, and correction of faults.

reliability—a stochastic measure of the likelihood that a system will be able to deliver a service.

requirement—a statement of a property that is necessary for correctness.

requirements management—the practice of controlling the development, traceability, and evolution of requirements. Normally this practice involves the use of a tool specialized for this purpose such as IBM Rational’s DOORS.

resource—an element characterized by a finite capacity.

reusability—the ability to use an element or work product in a different circumstance or environment.

reuse plan—a plan stating how existing elements will be reused within a project and/or how elements created in the project will be reused in subsequent projects.

reverse engineering—the construction of a model from a set of source code files.

Rhapsody—a UML-based executable modeling tool from IBM Rational.

risk—a quantitative measure of the possibility of an accident, the product of the likelihood of the accident and its severity; in project management, an unwelcome impact on cost, effort, functionality, or schedule.

risk management plan (aka risk list)—a plan that identifies a set of project risks, ranked in order of criticality, along with risk mitigation activities meant to address them.

risk mitigation activity (RMA)—a project work task used to characterize a potential risk and to identify a change to reduce that risk by lessening either its likelihood or its severity.

RMC—Rational Method Composer, a process authoring tool from IBM Rational. RMC is a more capable version of EPF Composer that comes with a large set of predefined process content.

robustness—the ability of an element to perform its intended function in an adverse environment, such as when preconditions are violated.

round-trip engineering—the processing of source code changes into an existing model.

RTC—Rational Team Concert, a commercial Jazz-based technology environment available from IBM Rational.

RTOS—real-time operating system.

safety—freedom from accidents or losses.

schedulable—the characteristic of a set of timely tasks, that is, a set of tasks, each of which can be guaranteed to meet its timeliness requirements.

schedule—an ordered set of tasks characterized by duration and loading onto resources. For project management, this is the plan that specifies which people should perform what tasks when. For a concurrent system, this is a statement of what tasks should be executed by the scheduler and when.

Scrum—an agile process based on a football analogy, using small teams working in an intensive, independent manner.

security—an aspect of information assurance concerned with the protection of information from espionage or inappropriate access.

SEI—Software Engineering Institute; see http://www.sei.cmu.edu/.

severity—the degree of potential harm of a situation or hazard.

slack time—the duration of time during which an action may be delayed before its ability to complete prior to its deadline is affected.

SME—subject matter expert.

soft real-time—a real-time system whose timeliness requirements are stochastically characterized.

software development plan—a plan that outlines how a project will be run, including the tasks to be performed, the work products to be produced, and the sequence of the tasks and products.

specification—a detailed coherent statement of particulars. A requirements specification details requirements whereas a design specification details the design of a system or element.

spike—an experiment that must be completed before project progress can be made. See **risk mitigation activity.**

spiral development—the process of incremental development. See **incremental development.**

SPT—the UML Profile for Schedulability, Performance, and Time, an OMG standard used to characterize models with timeliness and performance metadata to support performance and schedulability analysis. See **MARTE.**

stakeholder—a role that is concerned with the quality and content of a work product.

state machine—a formal specification of behavior in terms of conditions (states), events of interest that cause transitions among states, and actions that are executed as a result of event reception.

stereotype—in the UML, a user-defined "special kind" of metaclass. This is often associated with metadata stored in tags associated with the stereotype. Stereotypes often have special graphical images but are most commonly indicated by showing the base metaclass with the stereotype name in guillemets, such as in `«RTAction»`.

STL—Standard Template Library.

subsystem—a large-scale structured class that organizes and orchestrates internal parts. In the UML, it is a metasubtype of class. See **component**.

symmetric deployment—the allocation of functionality to processing nodes during design.

synchronization point—a point in the execution of a concurrent action sequence where all participants must wait for a condition to occur.

SysML—a UML profile addressing the needs of systems engineering.

systems engineering—an interdisciplinary field of engineering, focusing on the requirements and architecture for complex systems that will be realized by a combination of different engineering disciplines, such as software, electronics, mechanical, and chemical systems.

tag—a placeholder for metadata, usually associated with a stereotype.

task—a kind of concurrency unit supported by most operating systems, typically “heavier-weight” than a thread.

task diagram—a class diagram with the mission of showing the concurrency architecture.

test-driven development (TDD)—the practice of using testing to drive development forward, characterized by early specification of test cases, continual testing throughout development, and heavy use of regression testing.

testing—the application of test cases against a build to ensure that a system performs correctly in those cases.

thread—a kind of concurrency unit supported by most operating systems, typically “lighter-weight” than a task.

timeliness—the ability of a task to repeatedly meet its timeliness requirements.

tracking (project)—the process of measuring the progress of a project and comparing it to the planned progress.

trade study—an activity used to identify the most balanced technical solution among a set of proposed alternatives.

UML—the Unified Modeling Language, an OMG standard.

UML Testing Profile—an OMG standard that defines a metamodel for testing constructs, used to define a standard way to approach testing.

urgency—a measure of the required timeliness of an action; the shorter the time, the more urgent the action.

use case—a concrete usage of a system, characterized by a set of scenarios, a set of requirements to which it traces, and a specification state machine.

user model—the model of a system created by the developer.

user story—a requirement formulated into a small number of sentences in the user's natural language. User stories are common in Extreme Programming (XP).

utility function—the value of the completion of an action stated as a function of time. Actions whose timeliness is modeled with deadlines show a step function for their utility function.

validation—the process of checking that a system meets the user needs.

velocity—in agile methods, the rate at which goals are being achieved.

verification—the process of ensuring that a system meets its specifications.

waterfall lifecycle—a development approach based in industrial automation characterized by intensive, ballistic planning.

well-formed—in accordance with the syntax and semantics of an element.

worst-case performance—the longest amount of time an action can take during its execution.

XMI—XML Model Interchange, an OMG standard for model interchange.

Index