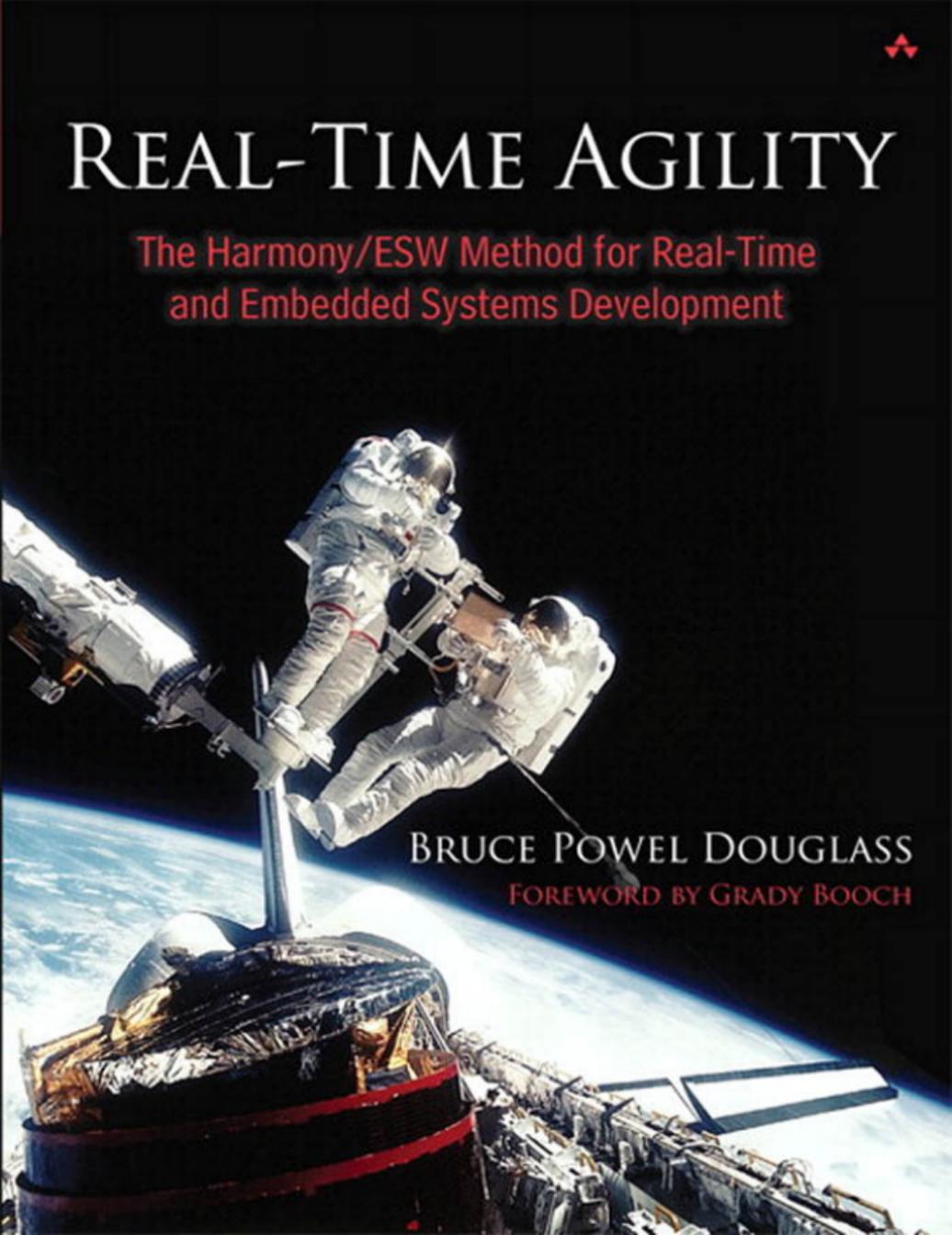




REAL-TIME AGILITY

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



BRUCE POWEL DOUGLASS

FOREWORD BY GRADY BOOCH

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Star Trek and the term “Starfleet Transporter” are TM, ®, and copyright © 2004 Paramount Pictures.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

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

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

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

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 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 OMG¹ standard for developing reusable software assets that is being applied very effectively in aerospace and defense systems.

Applying agile methods to modeling is not new. Scott Ambler's excellent book on agile modeling² 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.

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.

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

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

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.

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.

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.

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:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation

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

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

- *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 Programming³) movement based largely on the work of Kent Beck and Ward Cunningham. Both

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

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, real-time and embedded—and how to combine them with modeling to gain the synergistic benefits of model-driven development (MDD) approaches.⁴

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.

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.

4. A good place for more information about agile modeling is Scott Ambler's agile modeling Web site, www.agilemodeling.com.

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.

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.”

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

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

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 . . .

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.

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.

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.

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*.

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.

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!

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

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

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 real-time, 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 (Find, Fix, Track, Target, Engage, Assess) **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

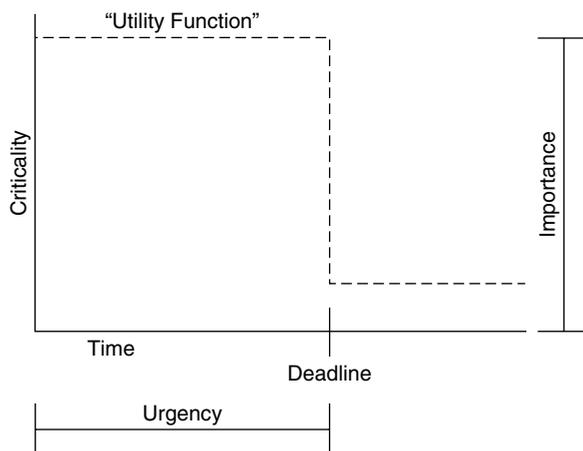


Figure 1.1 *Utility function*

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.

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.

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

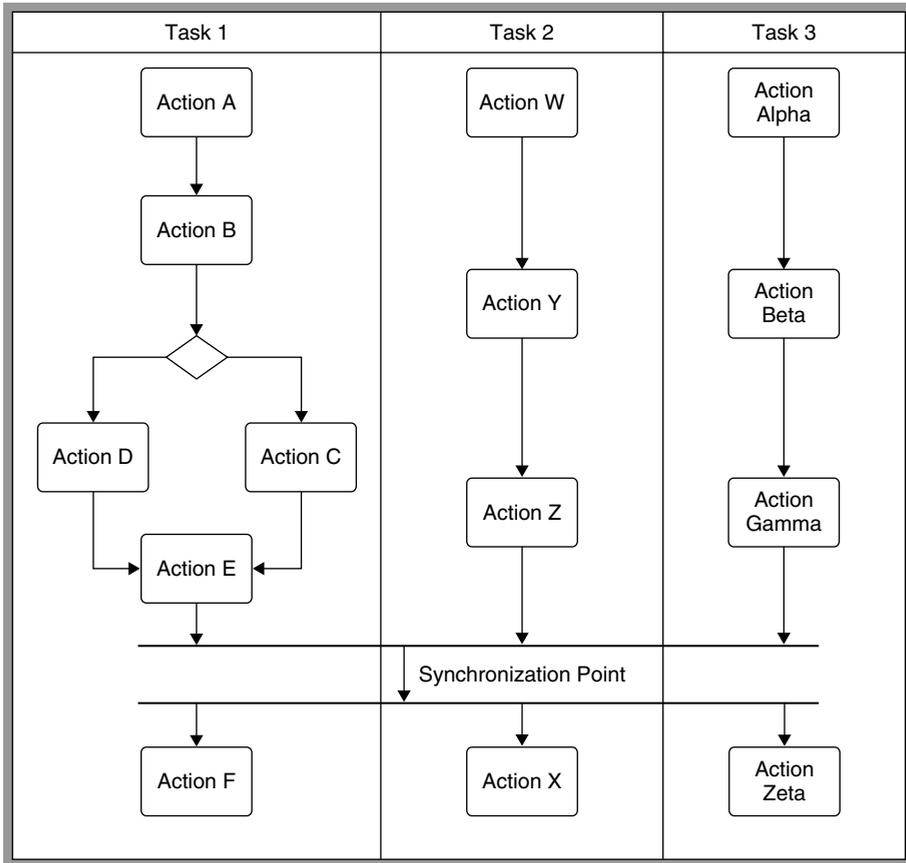


Figure 1.2 *Concurrency units*

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

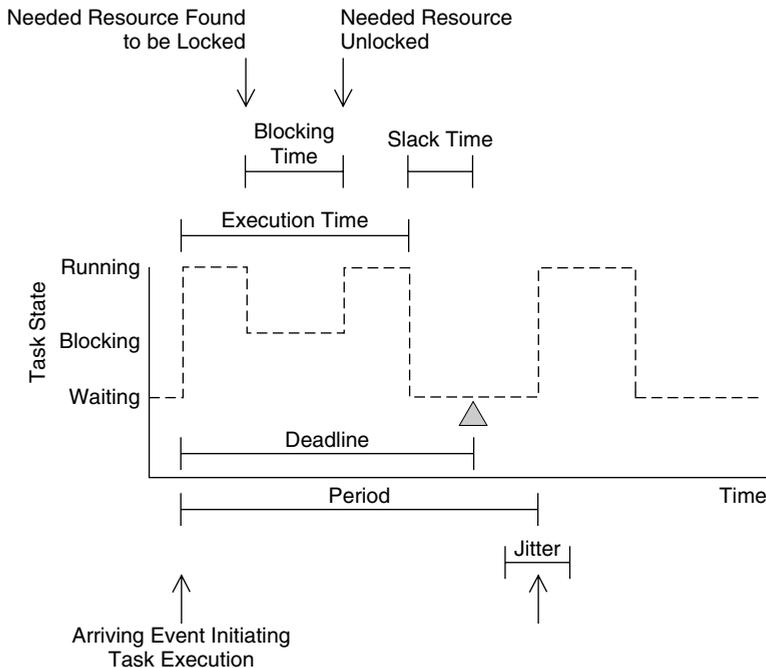


Figure 1.3 *Task time*

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.

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 tools¹¹ 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 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.

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 heavy-weight 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.

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

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

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

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.

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 risk¹⁴
- Efficient high-quality development

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.

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

14. See, for example, www.agileadvice.com.

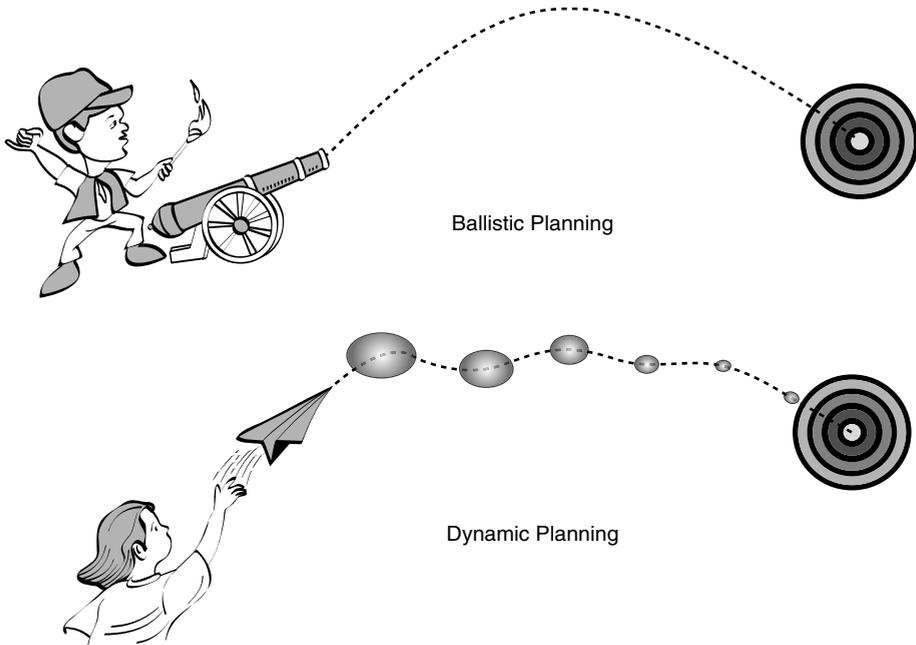


Figure 1.4 *Ballistic versus dynamic planning*

(see Figure 1.4). Agile’s dynamic planning can be thought of as “planning to re-plan.” 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.

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 Douglass¹⁵ 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.

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

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

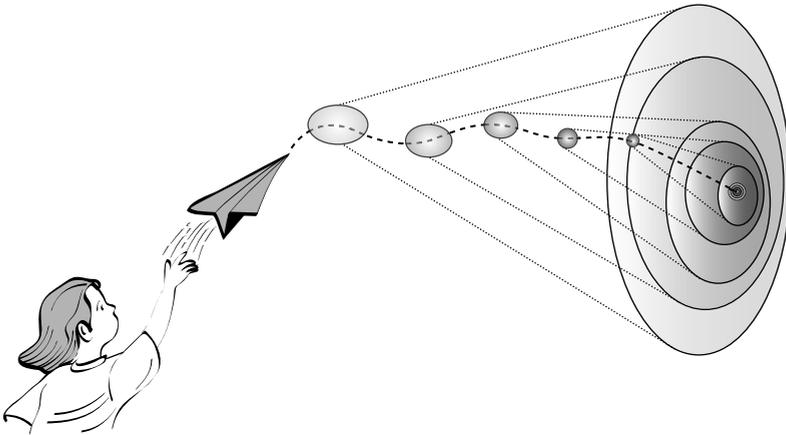


Figure 1.5 *Reduction in uncertainty*

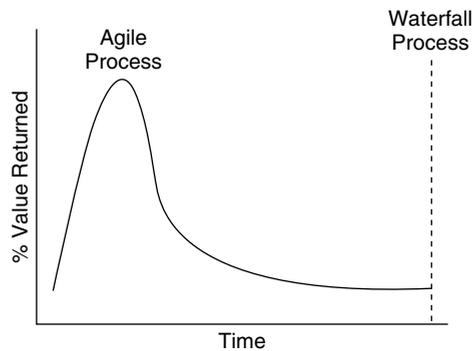


Figure 1.6 *Percent value returned over time*

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.

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.

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

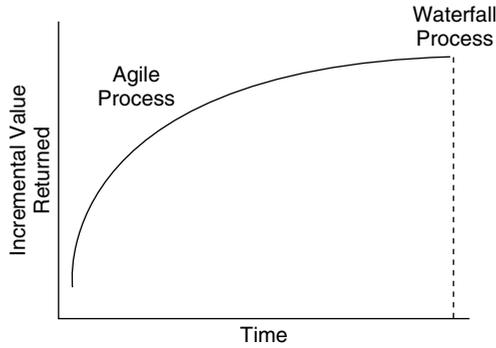


Figure 1.7 *Incremental value*

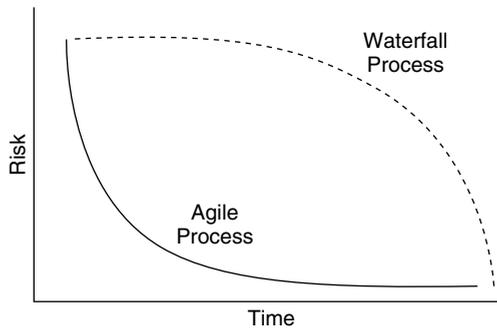


Figure 1.8 *Risk over time*

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.

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/ESW¹⁶ 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

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.

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.

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.

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.

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

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

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.

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 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.

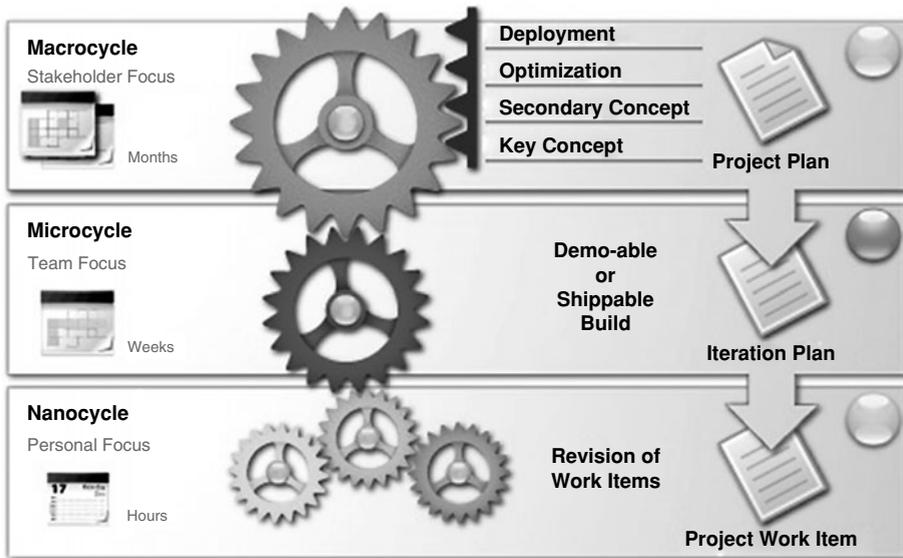


Figure 1.9 *Harmony/ESW timescales*

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 review¹⁸ and in the prototype mission statement that identifies the scope for the microcycle iteration.

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.

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.

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 CORBA¹⁹ is too slow to handle the throughput required, an early prototype²⁰ 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

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.

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

The risk list is a dynamic document that is reviewed at least every iteration (during the party phase²²). 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.

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.

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 Barbosa²³—“more what you’d call a guideline.” Agile development tends to follow a depth-first approach rather

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.

22. See Chapter 9.

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

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.

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.

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 KGB²⁵ agent carefully placed to

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

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

bring down the Western economy by single-handedly intercepting and losing your office memo.

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.

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 *disadvantage* 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.

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.

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.

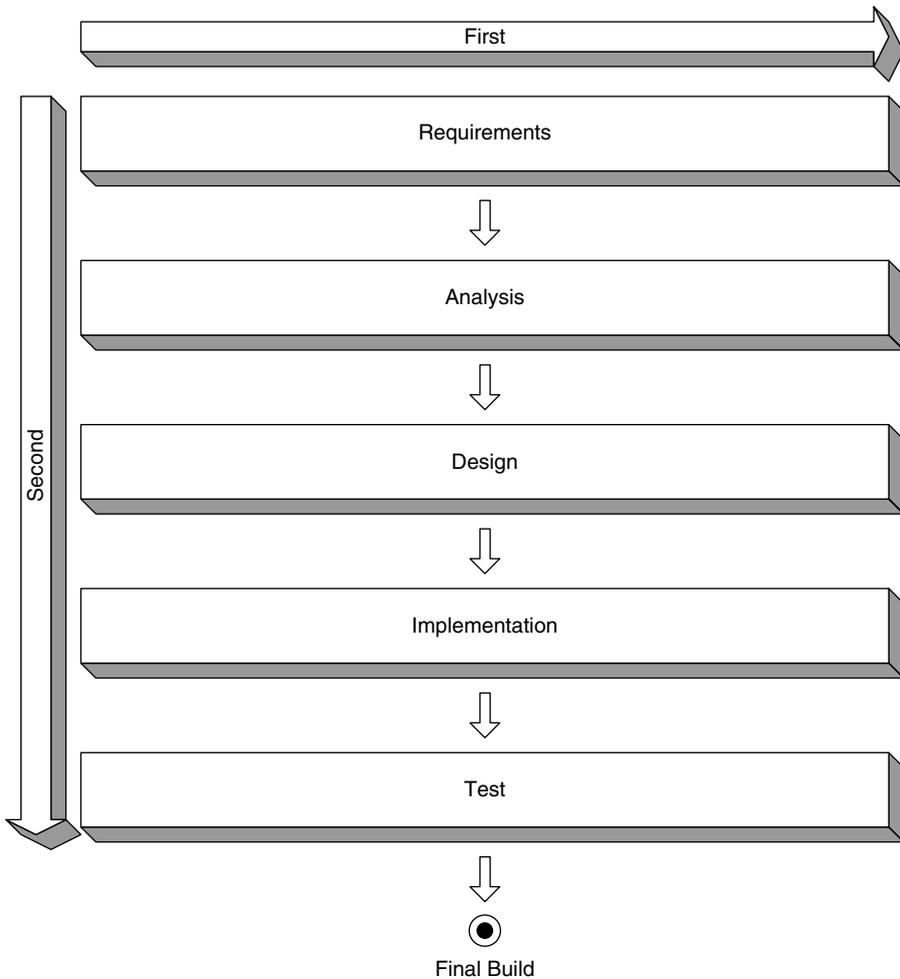


Figure 1.10 *Waterfall lifecycle*

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

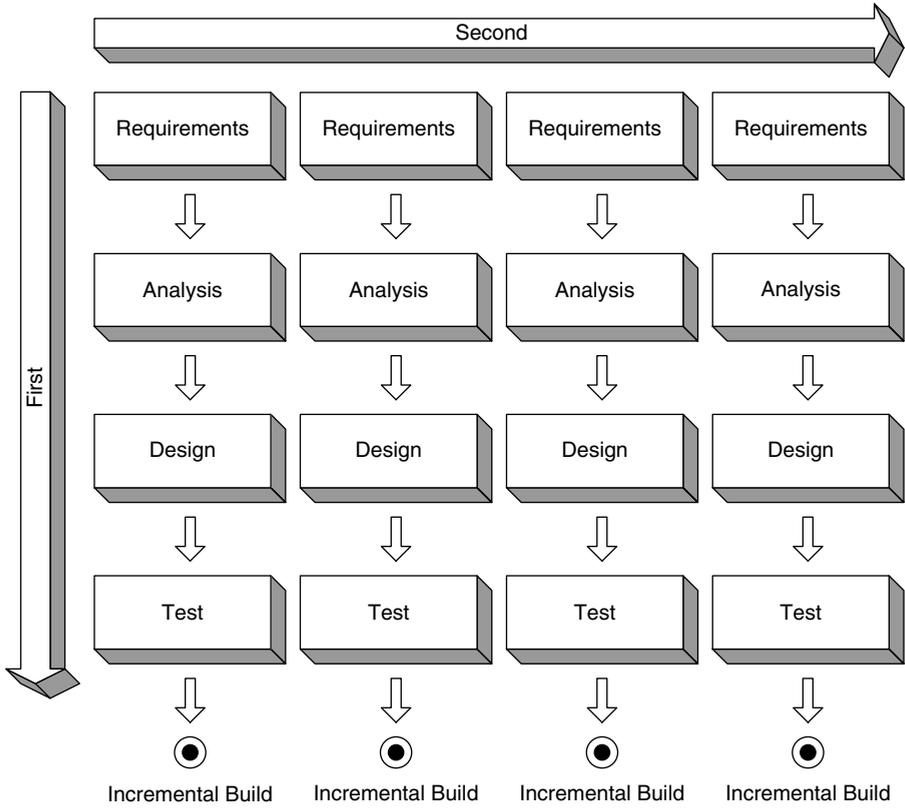


Figure 1.11 *Incremental spiral lifecycle*

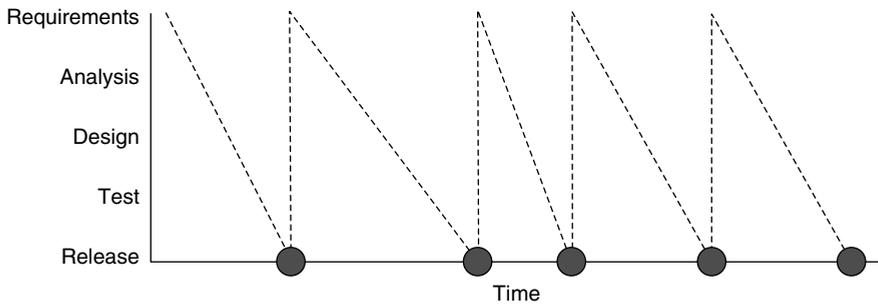


Figure 1.12 *Unrolling the spiral*

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.

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.

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).

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.

Index

- 50th-percentile estimate, 202
 - 7 ± 2 rule, 142
 - abbreviations, CMMI, 449
 - Abstract Factory pattern, 360
 - abstraction
 - clarity and, 35
 - identifying objects and classes, 316
 - abstraction action languages, 56–57
 - access restriction, 249
 - accidents, 82, 483
 - accuracy, 135
 - action languages, 55–57, 483
 - actions
 - defined, 483
 - safety-related faults, 233
 - specifying use case state machines, 292
 - «active» class
 - concurrency and resource management architecture, 343–344
 - defined, 483
 - activity diagrams
 - defined, 48
 - Harmony/ESW process, 171–177
 - model-based testing, 383–385
 - primary scenario, 284–285
 - representing tests, 147
 - Targeting Scanner example, 321
 - activity icons, 171–172
 - activity model, 483
 - actors, 281, 483
 - adaptation principle, 117
 - Adapter pattern, 360–361
 - adjusted use case points, 208–209
 - Agile Alliance, 1
 - agile analysis
 - bottom-up allocation, 303–307
 - change sets, 328
 - collaboration refinement, 321–325
 - creating unit test/suite, 325
 - detailing subsystem-level use cases, 308–309
 - detailing system requirements, 309–310
 - detailing use cases, 279–280
 - establishing traceability, 295–296
 - execution and testing, 327
 - generating system requirements, 296–298
 - iteration plan, 273–278
 - managing safety and reliability requirements, 298–299
 - object analysis, 310–312
 - object and class identification, 313–316
 - overview, 269–272
 - primary scenario activity diagram, 284–285
 - primary scenario identification, 280–284
 - prototype definition, 272–273
 - scenario validation, 292
 - secondary scenario identification, 285–291
 - subsystem architecture, 302–303
 - subsystem-level use case validation, 309
 - Targeting Scanner subsystem example, 317–321
 - top-down allocation, 307–308
 - translating model to code, 325–327
 - use case consistency, 309
 - use case state machine specification, 292–295
 - use case white-box analysis, 299–301
 - user interface specification, 278–279
- agile design
 - architectural. *See* architectural design
 - detailed design, 362–371
 - mechanistic design, 358–362

- agile design (*cont.*)
 - optimization and use of patterns, 331–339
 - overview, 329–331
 - primary and secondary architectural views, 339–340
- agile manifesto, 1–4
- agile methods
 - “Baby Bear” plans, 198–199
 - benefits of, 14–15
 - combining MDA, Harmony and, 194
 - control and, 20–21
 - defined, 483
 - Harmony/ESW process, 159
 - high quality development, 25
 - rapid learning, 15–16
 - real-time embedded systems, 8–14
 - responsiveness to change, 21–23
 - return on investment, 16–19
 - risk reduction, 24–25
 - spiral development, 179
 - stakeholder satisfaction, 19–20
 - vs. traditional processes, 25–30
 - why, 4–8, xx–xxii
- agile process optimization
 - change management, 414–416
 - dynamic planning, 404–406
 - model reviews, 417–420
 - overview, 403
 - party phase, 420–424
 - summary, 424–425
 - tracking and controlling, 407–414
- agile testing
 - concepts, 375–380
 - integration testing, 392–397
 - model-based, 380–385
 - overview, 373–375
 - unit testing, 388–392
 - validation testing, 397–401
 - workflows, 385–388
- Agile@Scale, 3
- Agilista
 - credo, 312
 - defined, 483
- alarms
 - as safety measure, 248
 - safety-critical systems, 231–232
- allocation
 - bottom-up, 303–307
 - top-down, 307–308
- alt (alternative), 281
- Ambler, Scott, i, xxiii
- amounts, 233
- analysis
 - agile. *See* agile analysis
 - CMMI standards, 460–461
 - defined, 483
 - hybrid-V lifecycle, 191–192
 - microcycles, 165–167
 - Newton-Raphson iteration, 133–135
 - patterns, 333
 - risk, 118
 - in spiral development, 177–179
 - spiral development, 180–183
- analysis models
 - defined, 45, 483
 - object analysis, 310
- AND-ing conditions, 245
- animated sequence diagrams, 106–112
- applicability, 60
- appraisals, CMMI, 192–194. *See also* CMMI (Capability Maturity Model Integration)
- architectural design
 - adding to Targeting Scanner, 353–357
 - defined, 483
 - in microcycle, 166–167
 - model organization, 221–223
 - optimization overview, 185–186
 - optimizing concurrency and resource management architecture, 343–347
 - optimizing deployment architecture, 351–353
 - optimizing distribution architecture, 347–349
 - optimizing safety and reliability architecture, 349–351
 - optimizing secondary architectural views, 353
 - optimizing subsystem and component architecture, 341–343
 - principles of, 112
 - workflow, 341

- architecture
 - analysis, 86
 - defined, 160, 483
 - logical. *See* logical architecture
 - model organization patterns, 226–228
 - model-driven. *See* MDA (Model-Driven Architecture)
 - understanding subsystem, 302–303
 - views. *See* Harmony’s architectural views
- arrays
 - performance, 369
 - emitter, 440
- The Art of Software Testing* (Meyers), 375
- assembly languages, 123–124
- associativity, model-code
 - defined, 152–153, 489
 - dynamic, 127
- assumption checking, 97–103
- asymmetric allocation, 86
- asymmetric deployment, 483
- ATG (Automatic Test Generator), 382
- attributes, 367–371
- audience, xxii
- automated transformations, 60–64
- Automatic Test Generator (ATG), 382
- availability
 - baseline, 265, 397
 - requirements, 249–250
 - use case selection criteria, 273–274
- average-case performance, 483
- “Baby Bear” plans, 198–199
- backward transformations
 - defined, 57
 - PSM to PIM, 65
- balanced binary trees, 369–370
- ballistic planning
 - defined, 15–16, 483
 - vs. dynamic planning, 117, 404
- Banerjee, Gautam, 209
- baseline
 - continuous configuration management, 264–265
 - defined, 263
 - validating and accepting changes to, 396–397
- Basic Input/Output System (BIOS), 483
- Beck, Kent, 1–2
 - on feedback, 193
 - on quality, 120
- behavior
 - diagrams, 47–48
 - object and class, 313
 - patterns, 360
- BERT (Bruce’s evaluation and review technique) method
 - defined, 201, 202–203
 - maturity level 4 compliance, 479
- best practices. *See* Harmony/ESW practices
- binary trees, 369–370
- biofilters
 - secondary transporter functions, 442–444
 - submode, 429
 - subsystem, 440
- biomaterials transport mode, 428–429
- BIOS (Basic Input/Output System), 483
- BIT (built-in test), 483
- black-box analysis, 279–280
- black-box testing, 379
- Blash, Ray, xxv
- blocking, 12, 484
- blocks, 484
- Boehm, Barry
 - on spiral development, 157
 - on use case points, 204
- Booch, Grady, xvii
- Boolean logic, 244–247
- bottom-up allocation, 303–307
- boundary tests, 148, 376
- Box, George, 142
- Bruce’s evaluation and review technique (BERT) method
 - defined, 201, 202–203
 - maturity level 4 compliance, 479
- builds
 - defined, 27, 484
 - model organization patterns, 227–228
- built-in test (BIT), 483

- “bulletproof” software, 322
- burn-down view, 411–412
- buses, 348–349
- butterfly effect, 21–22

- C++, 125
- Capability Maturity Model Integration (CMMI). *See* CMMI (Capability Maturity Model Integration)
- Capability Pattern, 484
- capacity
 - cargo transport, 428
 - principle of quality, 122
 - transporter personnel, 429
- CAR (causal analysis and resolution), 480–481
- cargo transport mode, 428
- causal agents, 314
- causal analysis and resolution (CAR), 480–481
- CCB (change control board), 401, 484
- CDR (critical design review)
 - vs. agile methods, 19
 - defined, 485
 - maturity level 3 compliance, 475
- Chain of Responsibility pattern, 360
- change management
 - agile process optimization, 414–416
 - CMMI standards, 460
 - defined, 263, 424, 484
 - dynamic planning and, 405
 - embracing change, 30
 - responding to, 21–23
 - in spiral development, 176–177
- change request, 414–416
- change sets
 - integration testing, 395–396
 - object analysis, 328
 - PI and, 463
 - repairing and rebuilding, 401
- Channel Pattern, 84, 337
- channels
 - defined, 84, 484
 - in Firewall Pattern, 238–242
- chaos theory, 21–22
- checking, 107–112

- checklists
 - hazard analysis, 250
 - logical architecture, 228–229
 - product vision, 252
 - scheduling, 213–214
 - stakeholder requirement, 256–258
- checksums, 240
- CIM (computation-independent model)
 - defined, 45, 484
 - modeling concepts, 41–43
 - transformation to PIM, 64
- CI (configuration items)
 - continuous integration, 151–152
 - model organization, 220–223
- clarity, 35
- class diagrams
 - deployment architecture, 87, 88
 - distribution architecture, 82, 83
 - managing interfaces with, 150–151
 - in MDA, 39
 - PIM models, 47–48
- classes
 - «testBuddy», 148, 390–392
 - defined, 484
 - detailed design, 367–371
 - evolution of, 125
 - identification, 313–316
 - identifying “special needs”, 365–367
 - model organization patterns, 227–228
 - model-based testing, 380
 - object analysis, 182–183
 - optimizing, 187
 - PIM models, 47–49
 - PSM modeling, 52–53
 - SecurityClass, 104–112, 381–383
 - subsystem and component architecture, 76–77
 - subsystem architecture, 302–303
 - subsystem definition, 342
 - taxonomies and structure, 37
 - UML diagrams, 68
- CM (configuration management)
 - CMMI standards, 457–460
 - continuous, 263–266
 - defined, 262–263, 484

- installing and configuring
 - environment, 261–262
 - model organization and, 220
- CMMI (Capability Maturity Model Integration)
 - achieving maturity level 2 compliance, 452–462
 - achieving maturity level 3 compliance, 462–477
 - achieving maturity level 4 compliance, 478–479
 - achieving maturity level 5 compliance, 479–481
 - basics, 447–451
 - defined, 192–194, 484
 - further reading, 482
 - summary, 481–482
- COCOMO (Constructive Cost Model)
 - defined, 135
 - use case points and, 204
- code. *See also* source code
 - defined, 484
 - spiral development and, 179
 - translating models to, 325–327
- code generation
 - defined, 36
 - PSM to PSI, 65–66
 - with Rhapsody, 64
- Cohn, Mike
 - story points, 204
 - on velocity, 135
- collaborations
 - defined, 484
 - in MDA, 37
 - mechanistic design, 186
 - minimizing complexity, 138–141
 - optimizing mechanistic model, 358–362
 - parameterized, 60
 - PIM modeling, 48
 - PSM modeling, 52–53
 - refinement, 321–325
 - in spiral development, 182–183
 - subsystem definition, 342
 - Targeting Scanner example, 317–319
- Command pattern, 360
- commenting on use cases, 281–282
- common mode faults, 237
- Common Object Request Broker Architecture (CORBA)
 - defined, 349, 485
 - MDA and, 33
- Common Warehouse Metamodel (CWM), 70, 485
- communication
 - distribution architecture, 348–349
 - importance to integration testing, 395
 - models and, 38
 - risk management, 119
- compilation, 326
- complexity
 - defined, 4
 - estimation with use case points, 205–210
 - practices to minimize, 138–141
 - “sunny-day” scenarios, 282
- compliance, CMMI. *See* CMMI (Capability Maturity Model Integration)
- component architecture
 - defined, 340
 - in MDA, 76–77
 - optimization, 185, 341–343
 - principles of, 113–114
 - Starfleet ZX-1000 Transporter System, 437–440
- components, 484
- Composer tool, 471
- Composite pattern, 360–361
- computation-independent model (CIM). *See* CIM (computation-independent model)
- concepts, key
 - identifying objects and classes, 316
 - macrophase focus, 164
- conciseness, 121
- concurrency
 - defined, 484
 - principles of, 113–115
- concurrency architecture
 - adding to transporter, 354–355
 - defined, 77–81, 340
 - in MDA, 37
 - optimization, 185, 343–347
 - task diagrams, 54

- concurrency units, 11–12, 484
- configuration
 - development tool, 258–259, 261
 - transporter system, 436
- configuration items (CIs)
 - continuous integration, 151–152
 - model organization, 220–223
- configuration management (CM). *See* CM (configuration management)
- consequences, design pattern, 60, 333–334
- consistency
 - vs. accuracy, 135
 - use case, 309
- constraints
 - defined, 72, 484
 - design hints, 58–59
 - developing stakeholder requirements, 255
 - pre-conditional variants, 98
 - real-time embedded systems, 8
 - reuse plan, 215–216
- construction
 - practices, 130–133
 - schedule, 210–211
 - subsystem definition, 342
- Constructive Cost Model (COCOMO)
 - defined, 135
 - use case points and, 204
- continuous configuration management, 263–266
- continuous integration
 - agile testing, 394–395
 - CM and, 458
 - defined, 151–152, 484
 - PI and, 463
 - project initiation, 262–266
 - spiral development, 180
- control
 - with agile methods, 20–21
 - agile process optimization, 407–414
 - CMMI standards of, 454–455, 461
 - project, 176
 - risk, 118–119
 - transporter operator, 434
- control measures
 - defined, 233
 - defining, 248–249
 - optimizing architecture, 350
- CORBA (Common Object Request Broker Architecture)
 - defined, 349, 485
 - MDA and, 33
- core practices. *See* Harmony/ESW practices
- core principles. *See* Harmony/ESW principles
- correctness
 - defined, 10
 - object analysis and, 312
 - proving system, 146–147
 - quality and, 120
 - testing for, 374
- COTS, 351
- coverage tests, 148, 376–378
- CRC (Class, Responsibility, and Collaboration) cards
 - achieving safety with, 240
 - defined, 485
- creational patterns, 360
- creator role, 263–264
- critical design review (CDR)
 - vs. agile methods, 19
 - defined, 485
 - maturity level 3 compliance, 475
- criticality
 - coverage testing, 377–378
 - critical regions, 55
 - defined, 485
 - design pattern criteria, 335–337
 - use case selection criteria, 273–274
- cross-compilation, 9
- Cunningham, Ward, 2
- custom hardware, 9
- customers. *See also* stakeholders
 - agile methods and, xxi
 - agile priorities, 2
 - judge of quality, 120
 - satisfying with agile methods, 19–20
 - scheduling and, 210–211
- CWM (Common Warehouse Metamodel), 70, 485
- Cyclic Executive Scheduling Pattern, 338

- Data Distribution Service (DDS), 485
- data link protocols, 349
- data management
 - corruption, 240
 - identifying objects and classes, 315–316
 - optimizing structures, 187
 - principles of, 116
 - views, 88–89
- DC (decision coverage), 378
- DDS (Data Distribution Service), 485
- Deadline Monotonic Analysis (DMA), 485
- deadlines, 11
- debugging
 - defined, 154, 485
 - model execution, 327
 - PIM to PSI transformation, 66
 - real-time embedded systems, 9
- decision coverage (DC), 378
- decomposition, lifeline
 - bottom-up allocation, 303–304
 - top-down allocation, 307–308
- defects
 - CAR, 480–481
 - principles of minimizing, 95–96
 - proving system correctness, 146–147
 - reviewing list and rates, 423
 - safety-related faults, 233–242
 - spiral development, 170
- defensive development
 - defined, 96
 - design, 235
 - detailed design, 370–371
 - principle of, 97–103
 - refining collaborations, 322
- defined CMMI processes, 193
- delivered functionality, 123
- delivery process, 485
- DeMarco, Tom
 - approach compared to BERT method, 203
 - on measurement, 117
 - on tracking, 211
- dematerialization, 432
- Department of Defense Architecture Framework (DoDAF), 486
- deployment
 - development environment, 258–262, 408–410
 - macrophase focus, 164
- deployment architecture
 - adding to transporter, 353–354
 - defined, 84–87, 340, 485
 - in MDA, 37
 - optimization, 185
 - optimizing, 351–353
 - principles of, 113–115
- depth-first development, 27–30
- design. *See also* agile design
 - constraints, 215–216
 - defined, 485
 - Harmony/ESW process. *See* Harmony/ESW process
 - hints, 58–59
 - hybrid-V lifecycle, 191–192
 - maturity level 3 compliance, 465–466
 - MDA. *See* MDA (Model-Driven Architecture)
 - microcycles, 165–166
 - optimization, 128–129
 - PSM modeling, 49–55
 - source-code-based systems vs. MDA, 36–40
 - in spiral development, 177–179
 - spiral development, 183–187
 - testing workflows, 385–388
- design cost, 485
- design idioms
 - agile design, 362–363
 - defined, 485
 - PSM modeling, 53
- design models, 485
- design patterns
 - applying intelligently, 149–150
 - defined, 49, 485
 - for Harmony architectural views, 114–115
 - maturity level 3 compliance, 465–466
 - mechanistic, 360–362
 - metamodel transformations, 59–64
 - optimization and use of, 331–339
- detailed design
 - agile design, 362–371

- detailed design (*cont.*)
 - defined, 485
 - in microcycle, 166–167
 - optimization, 187
 - principles of, 112
- detectability
 - hazard analysis, 247
 - optimizing, 349–350
- deterministic, 485
- detoxygenation submode, 429
- development
 - CMMI, 192–194
 - defensive. *See* defensive development
 - evolution of, 123–128
 - with frameworks, 144–146
 - high quality with agile methods, 25
 - principle of working software, 92–94
 - spiral. *See* spiral development
- development environment
 - defining and deploying, 171, 174
 - maturity level 5 compliance, 480
 - OPD, 471
 - project initiation, 258–262
 - refining and deploying, 408–410
- device drivers, 8–9
- device identification, 316
- diagrams
 - agile analysis. *See* agile analysis
 - defined, 486
 - evolution of, 125–128
 - Harmony/ESW process, 171–177
 - Harmony’s architectural views and. *See* Harmony’s architectural views
 - in MDA, 37–39
 - MDA technologies, 68–73
 - model-based testing, 380–385
 - modeling and, 42
 - modeling with purpose, 142–144
- distribution architecture
 - adding to transporter, 355–357
 - defined, 81–82, 340, 486
 - deployment architecture and, 86
 - in MDA, 37
 - optimization, 185
 - optimizing, 347–349
 - principles of, 113–115
- diverse redundancy, 239–241
- DMA (Deadline Monotonic Analysis), 485
- DMCA (dynamic model-code associativity), 127
- DO-178B
 - coverage testing, 377
 - defined, 295
- documentation
 - agile methods and, 14
 - agile priorities, 1
 - CMMI, 450
 - principle of quality, 121
- DoDAF (Department of Defense Architecture Framework), 73, 486
- Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Douglass), xix
- domains, 222
- DOORS, 253
 - traceability, 295–296
- Doppler compensations, 433
- drawing tools, 65–66
- dual maintenance, 255
- dynamic, 159
- Dynamic Memory Pattern, 338
- dynamic model-code associativity (DMCA), 127
- dynamic planning
 - benefits of, 15–16
 - defined, 26–27, 486
 - practices, 133–138
 - principles of, 117
 - understanding, 404–406
- Eclipse, 486
- Eclipse Process Framework (EPF), 171
- Eclipse Process Framework (EPF) Composer, 471, 486
- education as safety measure, 248
- effect review for nanocycle iteration
 - estimation (ERNIE) method
 - defined, 201, 211–213
 - maturity level 4 compliance, 479
- efficiency
 - with agile methods, 25
 - defined, 486

- estimation with use case points, 209
- principle of quality, 121–122
- embedded software. *See* Harmony/ESW (Embedded Software)
- embedded system, 486. *See also* real-time and embedded systems
- emitter array, 440
- energizing coils, 439
- engineered software
 - defined, 6
 - systems engineering, 191–192
- engineering, reverse. *See* reverse engineering
- engineering, round-trip
 - defined, 66–67, 491
 - evolution of UML, 127
- enterprise architecture, 34–35
- environments
 - estimation with use case points, 208
 - installing and configuring CM, 261–262
 - objects, 205
- environments, development
 - defining and deploying, 171, 174
 - project initiation, 258–262
- EPF (Eclipse Process Framework), 171
- EPF (Eclipse Process Framework) Composer, 471, 486
- equipment safety, 248
- ERNIE (effect review for nanocycle iteration estimation) method
 - defined, 201, 211–213
 - maturity level 4 compliance, 479
- errors
 - agile planning and, 26–27
 - defined, 486
 - principles of handling, 116
 - safety and reliability architecture, 82–84
 - safety-related faults, 233–242
 - testing. *See* testing
 - views, 88–89
- essential, 179
- essential models, 45
- estimable work units (EWUs), 202
- estimation
 - agile planning, 26–27
 - dynamic planning, 133–136
 - OPP, 478–479
 - scheduling. *See* scheduling software, 4–5
- Ethernet, 348
- events
 - concurrency and resource management architecture, 345
 - identifying objects and classes, 315
 - specifying use case state machines, 292
- evolution of software development, 123–128
- EWUs (estimable work units), 202
- exceptions
 - defined, 486
 - detailing scenarios, 285–291
 - handling views, 88–89
 - principles of handling, 116
 - sets, 285
- execution
 - dynamic planning and, 405
 - evolution of UML, 127
 - against expectations, 97
 - feedback principle, 103–112
 - identifying objects and classes, 316
 - measuring progress, 95
 - object analysis, 327
 - PIM, 48
 - principle of frequent, 96–97
 - TDD, 379–380
 - unit test, 390–392
 - validation test, 400–401
- expectations
 - execution against, 97
 - getting feedback, 103–112
 - increment review, 190
- extensions
 - SysML, 69
 - UML Profile, 70–73
- external expectations, 97
- external quality, 120
- extracting tests, 397–399
- Extreme Programming (XP), 1, 486
- F2T2EA (Find, Fix, Track, Target, Engage, Assess) Kill Chain, 10, 486
- Fagan inspections, 189, 418

- fail-safe state, 238
- failure mode, effects, and criticality analysis (FMECA), 486
- failure modes and effects analysis (FMEA). *See* FMEA (failure modes and effects analysis)
- failures
 - cost of ignoring risks, 118–119
 - defined, 486
 - detailing “rainy-day” scenarios, 285–291
 - vs. errors, 234–235
 - modes, 238
 - safety and reliability architecture, 82–84
- fairness, 486
- fault Seeding tests, 149, 377
- fault tolerance time
 - defined, 14
 - single-point faults, 236
- fault tree analysis (FTA). *See* FTA (fault tree analysis)
- faults
 - detailing “rainy-day” scenarios, 285–291
 - leading to hazards, 244–248
 - safety and reliability architecture, 84
 - safety-related, 233–242
- FDA (Food and Drug Administration), 248
- featurecide, 413
- feedback
 - CMMI, 193
 - dynamic planning, 15–16
 - error detection, 241
 - principles, 103–112
- feed-forward error detection, 241–242
- Find, Fix, Track, Target, Engage, Assess (F2T2EA) Kill Chain, 10, 486
- Firewall Pattern, 238–242
- five architectural views. *See* Harmony’s architectural views
- Fixed Block Memory Allocation Pattern, 337
- fixtures, test, 399–400
- Flyweight pattern, 361
- FMEA (failure modes and effects analysis)
 - defined, 247, 486
 - managing safety and reliability requirements, 298–299
 - optimizing safety and reliability architecture, 350–351
- FMECA (failure mode, effects, and criticality analysis), 486
- Food and Drug Administration (FDA), 248
- formal, 35
- Fowler, Martin, 1, 333
- frameworks
 - defined, 486
 - model organization patterns, 225
 - practices using, 144–146
- FTA (fault tree analysis)
 - defined, 244–247, 486
 - managing safety and reliability requirements, 298–299
 - optimizing safety and reliability architecture, 350–351
- functional tests
 - agile testing, 374–375
 - defined, 148–149, 376
- functionality
 - incremental value, 16–19
 - MDA, 35
 - optimization and design patterns, 331–339
 - optimization vs., 128–129
- functions
 - detailed design, 367–371
 - optimizing, 187
- fuzzy logic, 6–7, 486
- Gamma, Erich, 333
- “Gang of Four” (GoF) patterns, 357, 360
- Gantt charts, 486
- Gaussian distribution curve, 203
- generation, code. *See* code generation
- generative tools, 65
- global positioning system (GPS), 486
- goals
 - agile, 14–15

- agile development, 4–8
- agile manifesto, 1–3
- author's, xxii
- MDA, 36–40
- measuring progress, 94–95
- mission statements. *See* mission statements
- planning by, 136
- scheduling, 210–211
- specific CMMI, 448
- working software development, 92–94
- GoF (“Gang of Four”) patterns, 357, 360
- GPS (global positioning system), 486
- graphical models
 - evolution of, 125–128
 - visualization and, 37
- graphical user interface (GUI), 487
- guards, 292
- GUI (graphical user interface), 487
- guidelines, 157–158

- hacking, 120
- Hamming codes, 240
- hard real-time systems, 10
- hardware
 - adding architecture to transporter, 353–357
 - random faults vs. systematic faults, 234–235
- Harmony, 487
- Harmony/ESW (Embedded Software)
 - achieving maturity level 2 compliance, 452–462
 - achieving maturity level 3 compliance, 462–477
 - achieving maturity level 4 compliance, 478–479
 - achieving maturity level 5 compliance, 479–481
 - CMMI basics, 447–451
 - introduction, xxii
- Harmony/ESW practices
 - applying patterns intelligently, 149–150
 - creating software and tests, 147–149
 - dynamic planning, 133–138
 - incremental construction, 130–133
 - interfaces and integration, 150–152
 - minimizing complexity, 138–141
 - modeling with purpose, 142–144
 - overview, 130
 - proving system correctness, 146–147
 - using frameworks, 144–146
 - using model-code associativity, 152–153
- Harmony/ESW principles
 - attention to quality, 120–123
 - defensive development, 97–103
 - developing working software, 92–94
 - executing against expectations, 97
 - executing frequently, 96–97
 - five architectural views, 112–115
 - getting feedback, 103–112
 - measuring progress, 94–95
 - minimizing defects, 95–96
 - modeling, 123–128
 - optimization, 128–129
 - overview, 91–92
 - plan, track, and adapt, 117
 - project failure and ignoring risks, 118–119
 - secondary architectural views, 116
- Harmony/ESW process, 487
 - accessing content, xxiv
 - agile design. *See* agile design
 - agile optimization. *See* agile process optimization
 - analysis, 180–183
 - CMMI, 192–194
 - combining agile, MDA and, 194
 - continuous integration, 180
 - defined, 22–23
 - design, 183–187
 - increment review, 190–191
 - introduction, 158–163
 - macrocycle process view, 171–177
 - model review, 188–189
 - prototype-based spiral development, 168–170
 - PSM modeling, 52–53
 - spiral in depth, 177–179
 - systems engineering, 191–192
 - time frames, 163–168

- Harmony/ESW process (*cont.*)
 - validation, 189–190
 - validation preparation, 188
 - why, 155–158
- Harmony/Hybrid V, 487
- Harmony’s architectural views
 - agile design, 339–340
 - concurrency and resource management, 77–81
 - deployment, 84–87
 - distribution, 81–82
 - modeling with purpose, 142–143
 - optimization, 185–186
 - overview, 74–76
 - safety and reliability, 82–84
 - secondary, 87–89
 - subsystem and component, 76–77
 - writing mission statement, 277–278
- Harmony/SE, 487
- hatching patterns, 331–332
- hazard analysis
 - agile analysis, 298–299
 - checklist, 250
 - defined, 14
 - identifying hazards, 242–244
 - safety and reliability architecture, 84
 - updating, 414
- hazardous materials filter (HMF), 444–445
- hazards
 - costs of ignoring, 118–119
 - defined, 82, 231–232, 487
 - faults leading to, 244–248
 - identification, 242–244
 - planning for risk reduction, 217–219
 - safety-related faults, 233–242
- heterogeneous redundancy, 84–85, 487
- Heterogeneous Redundancy Pattern, 350
- high-quality development, 25
- high-reliability systems, 14
- HMF (hazardous materials filter), 444–445
- Hoffman, Hans Peter, 487, xix
- homogenous redundancy
 - achieving safety with, 239–240
 - defined, 84, 487
- Homogenous Redundancy Pattern, 350
- human operational nodes, 72
- Humphrey, Watts, 193
- hybrid-V lifecycle, 191–192
- IABM (Integrated Architecture Behavioral Model), 55
- identification
 - dynamic planning, 404
 - risk, 118
- idioms, design
 - agile design, 362–363
 - defined, 485
 - PSM modeling, 53
- IDL (interface description language), 487
- IEC 651508, 248
- IEEE, 252–253
- implementation
 - action languages and, 56–57
 - dynamic planning, 136
 - measuring progress, 94–95
 - PSM modeling, 54
- incentives, 211
- incidental complexity, 139
- increment, 131
- increment reviews
 - agile process optimization, 420–424
 - CMMI standards, 461
 - defined, 190–191, 487
 - maturity level 5 compliance, 480–481
 - in microcycle, 166, 168
- incremental construction, 130–133
- incremental development, 487. *See also* spiral development
- incremental implementation, 103–112
- incremental prototypes, 165
- incremental value
 - agile benefits, 16–19
 - depth-first development, 27–30
- industrial software development
 - vs. agile, 3–8
 - process, 156
- information assurance
 - defined, 230, 487
 - principles of, 116
 - views, 88–89
- information technology (IT) systems
 - defined, 487
 - vs. real-time embedded systems, 8–9

- infrastructure, 273–274
- inherent complexity, 139
- initial processes, 192
- initiation, project. *See* project initiation
- inspections
 - model review, 417–420
 - model reviews, 189
 - optimization by, 366
- installation
 - CM environment, 261–262
 - development tools, 258–259, 261
- instance models, 44, 487
- instances, 37
- instantiation, pattern, 336
- Integrated Architecture Behavioral Model (IABM), 55
- integrated project management (IPM), 474–475
- integration
 - continuous. *See* continuous integration
 - with MDA, 34–35
 - practices, 150–152
- integration testing
 - continuous configuration management, 265–266
 - model organization, 223
 - overview, 392–397
 - scope, 378–379
 - workflows, 385–387
- interactions, 1
- interarrival time, 11
- interface description language (IDL), 487
- interfaces
 - concurrency and resource management architecture, 345
 - defined, 487
 - practices, 150–152
 - subsystem architecture, 79–80
- interlocks
 - life-scanning safety, 428
 - safety measures, 248
- internal checking, 248
- internal expectations, 97
- internal quality, 120
- interoperability
 - defined, 487
 - with MDA, 34–35
- interrupt handlers, 345
- invariants, pre-conditional, 97–103
- invention, 6
- inversion, priority, 12, 490
- investment returns, 16–19
- IPM (integrated project management), 474–475
- isolation
 - Firewall Pattern, 238–242
 - of MDA, 74
- issue questionnaire, 421–422
- IT (information technology) systems
 - defined, 487
 - vs. real-time embedded systems, 8–9
- iterations
 - dynamic planning, 133–136
 - prototype definition, 273–278
- Java, 125
- Jazz, 487
- Jeffries, Ron, 1
- jitter
 - defined, 11, 487
 - periods, 344
- JSSEO (Joint SIAP System Engineering Organization), 55
- Karner, Gustav, 209
- key architectural views. *See* Harmony’s architectural views
- key concepts
 - identifying objects and classes, 316
 - macrophase focus, 164
- key risks, 218
- Klein, Mark H., 347
- Klingon, 327
- labeling, 249
- languages
 - action, 55–57
 - evolution of development, 123–128
 - MDA technologies, 67–73
 - MDA vs. source-code, 39–40
 - Var-aq, 327
- latent faults, 233, 237

- launching development environment, 259, 261
- Law of Douglass
 - defined, 487
 - dynamic planning, 16
 - estimation, 212
 - review process, 422
- Layland, James W., 347
- learning, 15–16
- levels in metamodel, 44
- Leveson, Nancy
 - on fail-safe state, 238
 - on risk, 231
- life sign scanner, 442
- lifeline decomposition
 - bottom-up allocation, 303–304
 - top-down allocation, 307–308
- life-scanning safety interlock, 428
- life-support systems, 231–232
- LightController, 98–103
- lightweight, 159
- likelihood
 - defined, 83
 - hazard analysis, 247
- linked lists, 368
- links
 - in MDA, 37
 - translating model to code, 326
- Lister, Timothy
 - approach compared to BERT method, 203
 - on measurement, 117
 - on tracking, 211
- lists, detailed design, 368–369
- Liu, C. L., 347
- load testing, 149, 377
- logical architecture
 - checklist, 228–229
 - model organization, 219–223
 - model organization patterns, 223–228
- loop operator, 281
- MA (measurement and analysis), 460–461
- macrocycles
 - defined, 23, 488
 - Harmony time frames, 163–164
 - process view, 171–177
- maintenance
 - dual, 255
 - principle of quality, 121
- managed processes, 193
- management
 - change, 176–177
 - configuration. *See* CM (configuration management)
 - integration testing, 395–396
 - risk, 24–25
 - scheduling. *See* scheduling
 - team structure and, 214–215
- manager role, 263–264
- manifest faults, 233
- manifesto, agile, 1–4
- mapping rules, 57–58
- MARTE (Modeling and Analysis of Real-Time and Embedded Systems), 73, 488
- Martin, Robert, 1
- matter stream transmission, 433–434
- maturity levels
 - CMMI, 450–452
 - level 1 compliance, 452
 - level 2 compliance, 452–462
 - level 3 compliance, 462–477
 - level 4 compliance, 478–479
 - level 5 compliance, 479–481
- MC/DC (modified condition decision coverage), 378
- MDA (Model-Driven Architecture)
 - benefits of, 73–74
 - CIM, 45
 - combining agile, Harmony and, 194
 - common model transformations, 64–67
 - concurrency and resource management architecture, 77–81
 - defined, 33–36, 488
 - deployment architecture, 84–87
 - distribution architecture, 81–82
 - Harmony’s five architectural views, 74–76
 - introduction, xxiii
 - maturity level 3 compliance, 465–466

- metamodel transformations, 57–64
- metamodels, 43–44
- model concept, 41–43
- PIM, 45–49
- PSI, 55–57
- PSM, 49–55
- safety and reliability architecture, 82–84
- secondary architectural views, 87–89
- subsystem and component architecture, 76–77
- technologies, 67–73
- why, 36–40
- MDD (model-driven development)
 - defined, 3, 488
 - introduction, xxiii
 - maturity level 3 compliance, 465–466
- mean time between failure (MTBF), 83, 229–230
- measurement and analysis (MA), 460–461
- measurements
 - optimization, 366
 - principle of progress, 94–95
- measures of effectiveness (MOE), 488
- mechanism, 488
- mechanistic design
 - agile design, 358–362
 - defined, 52–53, 488
 - in microcycle, 166–167
 - optimization, 186
 - principles of, 112
- metaclasses, 488
- meta-metamodel, 44, 488
- metamodels
 - defined, 43–44, 488
 - transformations, 57–64
- MetaObject Facility (MOF), 44, 67–68, 489
- methodology, 156
- methods, agile. *See* agile methods
- metrics
 - defined, 488
 - principles of good, 117
 - tracking project deviance, 21
- MFC (Microsoft Foundation Classes), 146
- microcycles
 - agile analysis. *See* agile analysis
 - agile process optimization. *See* agile process optimization
 - defined, 18, 23, 488
 - Harmony time frames, 163–168
 - incremental construction, 131–132
 - project initiation. *See* project initiation
 - initiation
 - spiral development, 177–179
 - testing workflows, 385–388
- Microsoft Foundation Classes (MFC), 146
- middleware, 349
- minimum interarrival time, 11, 488
- mining, pattern, 332
- Ministry of Defense Architecture Framework (MoDAF), 488
- mission statements
 - defined, 488
 - macrocycle, 163–164
 - microcycle, 165–166
 - modeling with purpose, 142–144
 - writing microcycle, 276–278
- MoDAF (Ministry of Defense Architecture Framework), 73, 488
- model organization
 - defined, 219–223
 - patterns, 223–228
 - team structure and, 214
- model repositories, 489
- model reviews
 - agile process optimization, 417–420
 - maturity level 5 compliance, 481
 - in microcycle, 166–167
 - PPQA, 461–462
- model-code associativity
 - defined, 489
 - dynamic, 127
 - using, 152–153
- Model-Driven Architecture (MDA). *See* MDA (Model-Driven Architecture)
- model-driven development (MDD). *See* MDD (model-driven development)
- modeling
 - Harmony/ESW process, 158–160
 - practices. *See* Harmony/ESW practices

- modeling (*cont.*)
 - principles, 123–128. *See also*
 - Harmony/ESW principles
 - with purpose, 142–144
 - reviews, 188–189
- Modeling and Analysis of Real-Time and Embedded Systems (MARTE), 73, 488
- models
 - agile testing, 380–385
 - compiling, 65–66
 - defined, 220, 488
 - libraries, 72
 - reuse plan, 215–216
 - transformations. *See* transformations, model
 - translating to code, 325–327
- modified condition decision coverage (MC/DC), 378
- MOE (measures of effectiveness), 488
- MOF (MetaObject Facility), 44, 67–68, 489
- Monitor-Actuator Pattern, 350
- monitoring
 - CMMI standards, 454–455
 - defining and building test fixtures, 399
 - safety-critical systems, 232
- motivational schedules, 210
- MTBF (mean time between failure), 83, 229–230
- mutual exclusion, 55

- naming design patterns, 333
- nanocycles
 - basic workflow, 96
 - defined, 23, 489
 - Harmony time frames, 163–168
 - incremental construction, 130
- near-warp transport, 441
- needlines, 72
- networks, 348–349
- neural networks, 7, 489
- Newton-Raphson iteration, 133–135
- nodes, 86–87
- nongenerative tools, 65
- nonlinear, 22
- nouns, underlining, 314

- OATM (Open Architecture Track Manager), 55
- object analysis
 - change sets, 328
 - CMMI standards, 459
 - collaboration refinement, 321–325
 - creating unit test/suite, 325
 - defined, 489
 - execution and testing, 327
 - in microcycle, 166–167
 - object and class identification, 313–316
 - overview, 310–312
 - in spiral development, 181–183
 - Targeting Scanner subsystem example, 317–321
 - translating model to code, 325–327
- object identification strategies, 64
- Object Management Group (OMG). *See* OMG (Object Management Group)
- Object Windows Library (OWL), 146
- object-oriented programming (OOP), 125–126
- Observer pattern, 360
- obviation, 248
- OID (organizational innovation and development), 480
- OMG (Object Management Group)
 - defined, 33–34, 489
 - introduction, xxiii
 - modeling concepts, 41–43
 - PSI modeling, 55–57
- one-to-many optimization, 367
- OOP (object-oriented programming), 125–126
- OPD (organizational process definition), 470–471
- Open Architecture Track Manager (OATM), 55
- operational modes, 427–429
- operational nodes, 72
- operators
 - Boolean logic, 244–247
 - detailing use cases, 281
 - transporter console, 437–438
 - transporter control, 434

- OPF (organizational process focus), 471–474
- OPP (organizational process performance), 478–479
- opt (optional), 281
- optimization
 - agile design. *See* agile design
 - agile process. *See* agile process optimization
 - applying intelligently, 149–150
 - defined, 489
 - design patterns and, 331–339
 - in Harmony/ESW process, 162
 - macrophase focus, 164
 - maturity level 3 compliance, 465–466
 - principles, 128–129
 - PSM modeling, 50, 53–54
 - real-time embedded systems, 8
 - in spiral development, 183–187
- optimization criteria, 489
- optimizing processes, 193
- organization, model. *See* model organization
- organizational innovation and development (OID), 480
- organizational process definition (OPD), 470–471
- organizational process focus (OPF), 471–474
- organizational process performance (OPP), 478–479
- orthogonal use cases, 309
- OT (organizational training), 474
- outcomes, 410
- OWL (Object Windows Library), 146

- packages
 - model organization, 220–222
 - patterns, 224–228
- Parable of the Cave, 4–5
- parameterized collaborations, 60
- parity, 240
- “party” phase
 - agile process optimization, 420–424
 - defined, 23, 487
 - in microcycle, 166, 168
 - spiral development, 190–191

- PAs (process areas), 448–450
- pattern buffers
 - defined, 439–440
 - storage, 441
- pattern hatching, 331–332
- pattern instantiation, 336
- pattern mining, 332
- patterns
 - in distribution architecture, 81–82
 - model organization, 223–228
 - transporter storage, 433
- PDF (probability density function), 489
- PDR (preliminary design review)
 - vs. agile methods, 19
 - defined, 490
 - maturity level 3 compliance, 475
- performance
 - defined, 489
 - OPP, 478–479
 - optimization. *See* optimization
- periods, 344, 489
- persistent data, 316
- personnel transport mode, 429
- PERT (Project Evaluation and Review Technique) chart, 489
- phase transition coils, 439
- phrases, noun, 314
- physical architecture
 - adding to transporter, 353–357
 - model organization patterns, 226–227
- physical devices, 316
- PI (product integration), 462–463
- PIM (platform-independent model)
 - common transformations, 64–67
 - defined, 45–49, 489
 - metamodel transformations, 57–64
 - modeling concepts, 41–43
 - transformation to PSM, 50–55
- planning
 - agile, 26–27
 - “Baby Bear”, 198–199
 - CMMI standards, 456–457
 - dynamic, 15–16. *See also* dynamic planning
 - prespiral planning. *See* prespiral planning
 - principle of, 117

- planning (*cont.*)
 - for reusability, 215–216
 - risk, 118
 - for risk reduction, 217–219
 - unit test, 388–390
 - writing test plan/suite, 399
- platform-independent model (PIM). *See* PIM (platform-independent model)
- platforms
 - PSM modeling, 49–55
 - real-time embedded systems, 9
- platform-specific implementation (PSI). *See* PSI (platform-specific implementation)
- platform-specific model (PSM). *See* PSM (platform-specific model)
- PMC (project monitoring and control), 454–455
- Port Proxy Pattern, 355–357
- portability
 - defined, 490
 - of MDA, 73
 - principle of quality, 121
- POST (power-on self-test), 490
- postconditions, 490
- power requirements, 431–432
- power-on self-test (POST), 490
- PP (project planning), 456–457
- PPQA (process and product quality assurance), 461–462
- practices
 - defined, 91, 490
 - Harmony/ESW. *See* Harmony/ESW practices
 - specific CMMI, 448–449
- precision, 26
- precondition tests
 - defined, 376
 - validation testing, 398
- preconditions, 97–103, 490
- predictability
 - defined, 490
 - real-time embedded systems, 9
 - scheduling, 210
 - “The Parable of the Cave”, 4–5
- preliminary design review (PDR)
 - vs. agile methods, 19
 - defined, 490
 - maturity level 3 compliance, 475
- prespiral planning
 - BERT method, 202–203
 - constructing schedule, 210–211
 - creating schedule, 200–202
 - defined, 166, 171
 - ERNIE method, 211–213
 - in macrocycle process view, 173–174
 - PP and, 456–457
 - project initiation, 199–200
 - scheduling checklist, 213–214
 - story points, 204
 - use case points, 204–210
- primary architectural views. *See* Harmony’s architectural views
- primary energizing coils, 439
- primary scenarios
 - activity diagram, 284–285
 - identification, 280–284
- principles
 - agile manifesto, 1–3
 - defined, 91, 490
 - Harmony/ESW. *See* Harmony/ESW principles
- prioritizing
 - agile manifesto, 1–3
 - concurrency and resource management architecture, 346–347
 - defined, 490
 - design pattern criteria, 335–339
 - work items list elements, 275–276
 - work items lists, 137
- priority inversion, 12, 490
- probability density function (PDF), 489
- process
 - agile methods and, xx–xxii
 - Harmony/ESW. *See* Harmony/ESW process
 - optimization. *See* agile process optimization
 - steps, 247
 - tailoring, 258–260
- process and product quality assurance (PPQA), 461–462
- process areas (PAs), 448–450
- product integration (PI), 462–463

- product vision, 250–252
- profiles
 - defined, 490
 - model-based testing, 380–385
 - representing tests, 147–148
 - SysML, 69
 - UML, 70–73
- progress measurement, 94–95
- project deviance, 20–21
- Project Evaluation and Review Technique (PERT) chart, 489
- project initiation
 - “Baby Bear” plans, 198–199
 - BERT method, 202–203
 - checklist for hazard analysis, 250
 - checklist for logical architecture, 228–229
 - constructing schedule, 210–211
 - continuous integration, 262–266
 - creating schedule, 200–202
 - creating team structure, 214–215
 - defining and deploying development environment, 258–262
 - defining safety measures, 248–249
 - developing stakeholder requirements, 250–258
 - ERNIE method, 211–213
 - faults leading to hazards, 244–248
 - hazard identification, 242–244
 - model organization, 219–223
 - model organization patterns, 223–228
 - overview, 197–198
 - planning for reuse, 215–216
 - planning for risk reduction, 217–219
 - prespiral planning, 199–200
 - safe requirements, 249–250
 - safety and reliability analysis, 229
 - safety-related faults, 233–242
 - scheduling checklist, 213–214
 - specifying for logical architecture, 219
 - story points, 204
 - terms, definitions and basic concepts, 229–232
 - use case points, 204–210
- project monitoring and control (PMC), 454–455
- project planning (PP), 456–457
- projects
 - controlling, 176
 - failure do to ignoring risk, 118–119
 - Harmony/ESW process. *See* Harmony/ESW process
 - tracking and controlling, 407–414
 - velocity, 204
- properties
 - analysis, 180–181
 - PIM architecture, 47
 - of real-time and embedded systems, 8–14
 - tools based on, 65–66
- protocols
 - data link, 349
 - defined, 490
- prototype definition
 - bottom-up allocation, 303–307
 - defined, 490
 - detailing subsystem-level use cases, 308–309
 - detailing system requirements, 309–310
 - detailing use cases, 279–280
 - establishing traceability, 295–296
 - generating system requirements, 296–298
 - iteration plan, 273–278
 - managing safety and reliability requirements, 298–299
 - in microcycle, 166–167
 - overview, 272–273
 - primary scenario activity diagram, 284–285
 - primary scenario identification, 280–284
 - scenario validation, 292
 - secondary scenario identification, 285–291
 - in spiral development, 181–182
 - subsystem architecture, 302–303
 - subsystem-level use case validation, 309
 - top-down allocation, 307–308
 - use case consistency, 309
 - use case state machine specification, 292–295
 - use case white-box analysis, 299–301
 - user interface specification, 278–279

- Prototype pattern, 360
- prototypes
 - defined, 24, 76, 270, 490
 - incremental, 27, 165–166
 - incremental construction, 131–133
 - model organization patterns, 227–228
 - spiral development based on, 168–170
 - validation, 400–401
- PSI (platform-specific implementation)
 - common transformations, 65–66
 - defined, 55–57, 489
 - modeling concepts, 41–43
- PSM (platform-specific model)
 - common transformations, 65–66
 - defined, 49–55, 489
 - modeling concepts, 41–43
 - transformations, 57–64
- purpose
 - design pattern, 333–334
 - integration test, 392
 - model review, 419
 - modeling with, 142–144
 - unit test, 389
- QoS (quality of service)
 - applying design patterns, 334–339
 - defined, 490
 - Harmony/ESW process, 159–162
 - object analysis and, 312
 - optimization, 128–129
 - principles of, 116
 - views, 88–89
- QoS (quality of service) tests
 - agile, 374–376
 - defined, 148
- QPM (quantitative project management), 193, 479
- quality
 - agile development, 25
 - attention to, 120–123
 - defined, 490
 - in process, 159–160
- quantitative project management (QPM), 193, 479
- queuing requests, 55
- “rainy-day” behavior
 - defined, 98
 - identification, 285–291
- random faults, 234–235
- range tests, 148, 376
- Rapid Object-Oriented Process for Embedded Systems (ROPES), xix
- rate monotonic analysis (RMA), 490
- Rational Method Composer (RMC), 171, xxiv
- Rational Team Concert (RTC), 491
- Rational Unified Process (RUP), 491, xxiii
- RD (requirements development)
 - CMMI, 448–450
 - maturity level 3 compliance, 463–464
- real-time, 491
- real-time and embedded systems
 - applying design patterns, 335–339
 - Harmony/ESW process. *See* Harmony/ESW process
 - importance of, 8–14
 - importance of concurrency and resource management architecture, 343
 - PSM modeling, 51
- Real-Time Design Patterns: Robust Scalable Architectures for Real-Time Systems* (Douglass), xx
- real-time operating system (RTOS), 491
- Real-Time UML* (Douglass), xx
- Real-Time UML Workshop for Embedded Systems* (Douglass), xx
- reasonableness checks, 241
- rebuilding, 401
- recurrence properties, 345
- recurring costs, 491
- Recursive Containment Pattern, 338
- redundancy
 - achieving safety with, 238–242
 - defined, 230, 491
 - detailed design, 371
 - in Harmony/ESW process, 162
 - safety and reliability architecture, 84
- refactoring
 - defined, 132
 - spiral development, 168–169

- reference frames, 317–318
- refining development environment, 408–410
- regression tests, 149, 377
- regulating safety-critical systems, 14
- relations, 182–183
- release of energy, 231
- releases, 9
- reliability
 - analysis, 229. *See also* safety analysis
 - defined, 14, 229–230, 491
 - in Harmony/ESW process, 162
 - managing requirements, 298–299
 - principle of quality, 121
 - PSM modeling, 53
- reliability architecture
 - defined, 340
 - in MDA, 82–84
 - optimization, 185
 - optimizing, 349–351
 - principles of, 113–115
- rematerialization, 434
- repairing, 401
- replanning, 138, 404
- repositories, model, 489
- representations
 - object and class, 315
 - test, 147–149
- REQM (requirements management), 452–454, 491
- requests, change, 414–416
- requirements
 - agile testing. *See* agile testing
 - defined, 491
 - detailing system, 309–310
 - developing stakeholder, 171, 175–176, 250–258
 - establishing traceability to, 295–296
 - generating system, 296–298
 - Harmony/ESW process, 160–161
 - managing safety and reliability, 298–299
 - minimizing complexity, 138–141
 - model organization, 221–222
 - model review, 189
 - Starfleet ZX-1000 Transporter System, 430–436
 - system, 198
 - traceability, 295–296
 - validation, 189–190
 - validation testing, 397–401
- requirements development (RD)
 - CMMI, 448–450
 - maturity level 3 compliance, 463–464
- requirements management (REQM), 452–454, 491
- resolution
 - cargo transport, 428
 - hazard analysis, 247
- resource management architecture
 - in MDA, 77–81
 - optimizing, 343–347
 - principles of, 113–115
- resources
 - defined, 491
 - identifying objects and classes, 315–316
- responsibilities, 351–353
- responsiveness to change, 21–23
- restricting access, 249
- return on investment (ROI), 16–19
- reusability
 - defined, 59, 491
 - design pattern, 332
 - of MDA, 74
 - review of, 423
- reuse plan
 - defined, 215–216, 491
 - project initiation, 215–216
- reverse engineering
 - defined, 36, 491
 - PSM to PIM, 65
 - vs. round-trip engineering, 66–67
- reviews
 - BERT method, 202–203
 - hazard analysis, 236–237
 - increment. *See* increment reviews
 - in MDA, 38–39
 - model. *See* model reviews
 - stakeholder requirements, 256–258
 - use case consistency analysis, 309
- Rhapsody
 - defined, 491
 - design patterns, 60–64
 - frequent execution with, 96–97
 - PIM to PSI transformation, 66

- Rhapsody (*cont.*)
 - TestConductor, 380–385
 - translating model to code, 326
- risk management plan
 - defined, 119, 219, 491
 - review of, 424
 - updating, 275
- risk management (RSKM), 476–477
- risks
 - cost of ignoring, 118–119
 - defined, 83, 230–232, 491
 - hazard analysis, 247
 - planning for, 217–219
 - reducing with agile methods, 24–25
 - updating, 410–411
 - use case selection criteria, 273–274
- RMA (rate monotonic analysis), 490
- RMAs (risk mitigation activities)
 - defined, 24–25, 218–219, 491
 - updating risk management plan, 275
 - updating risks, 410–411
- RMC (Rational Method Composer), 171, 491, xxiv
- robustness
 - defined, 491
 - quality and, 120
- ROI (return on investment), 16–19
- roles
 - continuous configuration management, 263–264
 - defined, 157–158
 - model review, 417–418
- ROPES (Rapid Object-Oriented Process for Embedded Systems), xix
- round-trip engineering
 - defined, 66–67, 491
 - evolution of UML, 127
- RSKM (risk management), 476–477
- RTC (Rational Team Concert), 491
- RTOS (real-time operating system), 491
- rule-based tools, 65–66
- RUP (Rational Unified Process), 491, xxiii–xxiv
- safety
 - concurrency and resource management architecture, 345
 - control measures, 233
 - defined, 491
 - in Harmony/ESW process, 162
 - managing requirements, 298–299
 - vs. security, 89
- safety analysis
 - analysis, 229
 - checklist for hazard analysis, 250
 - defining measures, 248–249
 - faults leading to hazards, 244–248
 - hazard identification, 242–244
 - related faults, 233–242
 - requirements, 249–250
 - terms, definitions and basic concepts, 229–232
- safety architecture
 - defined, 82–84, 340
 - optimization, 185
 - optimizing, 349–351
 - principles of, 113–115
 - PSM modeling, 53
- safety integrity levels (SILs), 248
- safety-critical systems
 - coverage testing, 377–378
 - defined, 14
 - hazards, 231–232
- SAM (supplier agreement management), 457
- SC (statement coverage), 378
- scalability
 - of agile development, 3
 - estimation with use case points, 209
 - model organization patterns, 224, 228
 - process, 156
- scanners, targeting, 440
- scenarios
 - identifying objects and classes, 316
 - “rainy-day”, 285–291
 - “sunny-day”, 280–284
 - validation, 292
- schedulability, 13, 492
- Schedulability, Performance, and Time (SPT), 72–73, 492
- scheduling
 - BERT method, 202–203
 - checklist, 213–214

- concurrency and resource management
 - architecture, 346–347
- constructing, 210–211
- creating, 200–202
- defined, 492
- detailing for current iteration, 276
- dynamic planning, 134–136
- ERNIE method, 211–213
- model review, 419
- in process, 157–158
- real-time embedded systems, 9
- review of, 422–423
- RMAs, 219
- story points, 204
- updating, 411–413
- use case points, 204–210
- work items, 137
- Schmidt, Douglas, 349
- scope
 - testing, 378–379
 - unit test, 389
- Scoville, Mark, i
- Scrum, 492
- secondary architectural views
 - agile design, 339–340
 - defined, 87–89
 - optimizing, 353
 - principles, 116
 - writing mission statements and, 277–278
- secondary concepts, 164
- secondary energizing coils, 439
- secondary functions, 440–445
- secondary scenarios, 285–291
- security
 - defined, 230, 492
 - principle of quality, 122
 - vs. safety, 89
- SecurityClass
 - feedback principle, 104–112
 - model-based testing, 381–383
- SEI (Software Engineering Institute), 447, 492
- self-assessment tests, 375
- sequence diagrams
 - defined, 68
 - detailing use cases, 281–284
- feedback principle, 106–112
- model-based testing, 381–383
- representing tests, 147
- validating scenarios with stakeholders, 292
- sequencing faults, 233
- sequencing transportation, 434–436
- service quality. *See* QoS (quality of service)
- service-oriented architecture (SOA), 88–89
- severity
 - defined, 492
 - hazard analysis, 247
- SGs (specific goals), 448
- Shafer, Don, i
- shared models, 227
- shutdown state
 - achieving safety, 241
 - fail-safe state, 238
- SIAP (Single Integrated Air Picture), 55
- sicherheit, 89
- SILs (safety integrity levels), 248
- simplicity
 - model, 64
 - practices, 138–141
 - use case selection criteria, 274
- simulators, 399
- single event groups, 345
- Single Integrated Air Picture (SIAP), 55
- single-bit errors, 240
- single-point faults, 235–237
- site-to-site transport, 441
- slack time, 492
- SME (subject matter expert), 492
- smoke tests
 - continuous integration, 395
 - defined, 265
- SOA (service-oriented architecture), 88–89
- soft real-time systems, 10, 492
- software
 - agile design. *See* agile design
 - agile development. *See* agile methods
 - agile testing. *See* agile testing
 - creation practices, 147–149
 - evolution of development, 123–128
 - random faults vs. systematic faults, 234–235

- software development plan, 259–260, 492
- Software Engineering Institute (SEI), 447, 492
- solutions, 60. *See also* design patterns
- sorted linked lists, 368–369
- source code
 - importance of modeling systems based on, 123–128
 - vs. MDD, 36–40
 - model-code associativity, 152–153
 - translating models to, 325–327
- “special needs” classes, 365–367
- specific goals (SGs), 448
- specific practices (SPs), 448–449
- specifications
 - CMMI. *See* CMMI (Capability Maturity Model Integration)
 - defined, 492
 - for logical architecture, 219
 - use case state machine, 292–295
 - user interface, 278–279
 - validation testing, 398
- spikes, 25, 492
- spiral development
 - analysis, 180–183
 - based on prototypes, 168–170
 - continuous integration, 180
 - defined, 27–30, 492
 - in depth, 177–179
 - design, 183–187
 - increment review, 190–191
 - microcycles, 164–165
 - model review, 188–189
 - validation, 189–190
 - validation preparation, 188
- SPs (specific practices), 448–449
- SPT (Schedulability, Performance, and Time), 72–73, 492
- stakeholder requirements
 - achieving maturity level 2 compliance, 452–454
 - detailing, 309–310
 - detailing use cases, 279–280
 - developing, 171, 175–176
 - establishing traceability to, 295–296
 - maturity level 3 compliance, 463–464
 - project initiation, 250–258
 - selecting use cases, 273–275
- stakeholders
 - defined, 492
 - managing interfaces and, 150
 - satisfying with agile methods, 19–20
 - validating scenarios with, 292
 - validating subsystem-level use cases, 309
- Standard Template Library (STL), 60–61, 493
- standards, CMMI. *See* CMMI (Capability Maturity Model Integration)
- standards, MDA, 67–73
- Starfleet ZX-1000 Transporter System
 - defined, 282
 - general system requirements, 430–436
 - operational modes, 427–429
 - overview, 427
 - secondary functions, 440–445
 - system components, 437–440
- state
 - change request, 414–416
 - UML diagrams, 68
- state machines
 - «testBuddy» class, 326
 - change request, 416
 - defined, 492
 - detailed design, 371
 - feedback principle, 103–112
 - minimizing complexity, 139–141
 - PIM models and, 48
 - Targeting Scanner example, 322–324
 - unit test execution, 391–394
 - use case specification, 292–295
 - validation testing, 398–399
- statement coverage (SC), 378
- Static Priority Pattern, 337
- statistical tests, 148, 376
- stereotypes
 - defined, 71–72, 492
 - design hints, 58–59
- STL (Standard Template Library), 60–61, 493
- storage
 - pattern buffers, 441
 - redundancy, 239–240
 - transporter pattern, 433

- story points
 - creating schedule, 204
 - defined, 134
- stress tests, 148, 376
- stretch goals, 210
- structural patterns, 360
- studies, trade, 351–352, 493
- subject matter expert (SME), 492
- submachines, 293–295
- subsystem architecture
 - defined, 76–77, 340
 - in MDA, 37
 - model organization patterns, 227–228
 - optimization, 185
 - optimizing, 341–343
 - principles of, 113–114
 - Targeting Scanner example, 317–321
 - understanding, 302–303
- subsystem-level use cases
 - bottom-up allocation, 303–307
 - detailing, 308–309
 - top-down allocation, 307–308
 - validation, 309
 - white-box analysis, 299–301
- “sunny-day” behavior
 - activity diagram, 284–285
 - defined, 98
 - identification, 280–284
- supplier agreement management (SAM), 457
- symmetric allocation, 86
- symmetric deployment, 493
- synchronization
 - evolution of development, 126
 - points, 493
 - real-time embedded system, 11–12
- SysML (Systems Modeling Language)
 - blocks, 484
 - defined, 69, 493
- system requirements
 - defined, 198
 - detailing, 309–310
 - generating, 296–298
 - vs. stakeholder requirements, 254
 - Starfleet ZX-1000 Transporter System, 430–436
- systematic faults, 234–235
- systems
 - engineering, 191–192, 493
 - modeling. *See* MDA (Model-Driven Architecture)
 - proving correctness, 146–147
 - safety issues, 233
- Systems Modeling Language (SysML)
 - blocks, 484
 - defined, 69, 493
- tags
 - defined, 493
 - design hints, 58–59
 - stereotypes, 72
- tailoring, 258–260
- Talbott, Chris, i
- target platforms
 - deployment architecture, 351
 - writing mission statement, 277
- target scan and coordinate lock, 432
- Targeting Scanner subsystem example
 - adding architecture to, 353–357
 - defined, 317–321
 - optimizing mechanistic model, 361–362
- targeting scanners, 440
- task diagrams
 - concurrency architecture, 77, 81
 - defined, 493
 - PSM modeling, 54
- tasks
 - concurrency and resource management architecture, 343–347
 - defined, 157–158, 493
 - defining and deploying development environment, 258–259
 - execution time, 12
 - icon, 172
 - prespiral planning, 199–200
- TDD (test-driven development)
 - agile testing, 379–380
 - defined, 26, 30, 493
- Team Software Process (TSP), 193
- team structure
 - model organization and, 219–220
 - project initiation, 214–215
- technical factors in estimation, 207

- technical solution (TS), 464–466
- technologies
 - MDA, 67–73
 - PSM modeling, 49–55
- templates, 157–158
- terms
 - glossary, 483–494
 - project initiation, 229–232
- test buddies, 390–392
- testability, 121
- «testBuddy» class
 - creating unit test/suite, 325
 - model-based testing, 380
 - representing tests, 148
 - unit test execution, 390–392
- TestConductor, 380–385
- test-driven development (TDD)
 - agile testing, 379–380
 - defined, 26, 30, 493
- testing. *See also* agile testing
 - attention to quality, 122–123
 - continuous configuration management, 265–266
 - defined, 493
 - dynamic planning, 404
 - feedback principle, 103–112
 - hybrid-V lifecycle, 191–192
 - maturity level 5 compliance, 481
 - in MDA, 40
 - microcycles, 165–166
 - model organization, 221–223
 - object analysis, 327
 - PIM to PSI transformation, 66
 - practices, 147–149
 - spiral development, 170
 - UML Testing Profile, 73
- theory and optimization, 366
- threads, 493
- throughput, 122
- throwaway prototypes
 - defined, 24
 - incremental construction, 131
 - user interface specification, 279
- time, fault tolerance
 - defined, 14
 - single-point faults, 236
- time frames, 163–168
- timeliness
 - defined, 13, 493
 - real-time embedded system, 9–10
 - transporter requirements, 430
- timing diagrams, 80, 82
- timing faults, 233
- TMR (Triple Modular Redundancy), 337, 350
- tools
 - agile process optimization, 424–425
 - agile testing. *See* agile testing
 - creating unit test/suite, 325
 - defining and building test fixtures, 399–400
 - development, 258–260
 - MDA. *See* MDA (Model-Driven Architecture)
 - in process, 157–158
 - traceability, 296
 - translating model to code, 326
- top-down allocation, 307–308
- traceability, 295–296
- tracking
 - agile process optimization, 407–414
 - change sets, 328
 - defined, 424, 493
 - ERNIE method, 211–213
 - improved control with, 20–21
 - measuring progress, 94–95
 - modeling concepts, 42–43
 - principle of, 117
 - risks, 118
 - safety analysis. *See* safety analysis
- trade studies, 351–352, 493
- trade-offs, 335
- traditional processes
 - vs. agile methods, 25–30
 - defined, 7
- training, organizational (OT), 474
- transactions, 315
- transformations, model
 - common, 64–67
 - defined, 489
 - metamodel, 57–64
 - modeling concepts, 42–43
- translation, 325–327
- transport chamber, 439

- transport sequencer, 439
- transportation sequencing, 434–436
- transporter. *See* Starfleet ZX-1000 Transporter System
- Triple Modular Redundancy (TMR), 337, 350
- TS (technical solution), 464–466
- TSP (Team Software Process), 193
- TUV, 236
- types, 182–183

- UML (Unified Modeling Language)
 - defined, 68–69, 493
 - detailing use cases, 281
 - evolution of, 126–128
 - Harmony/ESW process diagrams, 171–177
 - MDA and, 33–34
 - metamodel concepts, 43–44
 - model organization, 220
 - model-based testing, 380–385
 - modeling concepts, 42–43
 - Testing Profile, 40, 493
- UML Profile for DoDAF and MoDAF (UPDM), 73
- UML Profile for Schedulability, Performance, and Time, 72–73, 492
- unadjusted actor weight, 205
- uncertainty, 137–138
- underlining nouns and noun phrases, 314
- understandability, 120–121
- Unified Modeling Language (UML). *See* UML (Unified Modeling Language)
- unit testing
 - creating test/suite, 325
 - defined, 154
 - maturity level 5 compliance, 481
 - model execution, 327
 - model organization, 223
 - overview, 388–392
 - scope, 378–379
 - workflows, 385–386
- updating
 - baseline change, 396
 - hazard analysis, 414
 - maturity level 5 compliance, 480
 - risk management plan, 275
 - risks, 410–411
 - schedules, 411–413
- UPDM (UML Profile for DoDAF and MoDAF), 73
- urgency
 - defined, 493
 - use case selection criteria, 274
- use case diagrams
 - CIM models, 45–46
 - in MDA, 39
- use case points, 204–210
- use cases
 - collaborations, 182
 - defined, 493
 - detailing, 279–280
 - developing stakeholder requirements, 253–254
 - dynamic planning, 134–136
 - establishing traceability, 295–296
 - executing against expectations, 97
 - model organization, 221–222
 - model organization patterns, 224–225
 - PIM models and, 47–48
 - primary scenario activity diagram, 284–285
 - primary scenario identification, 280–284
 - reviewing stakeholder requirements, 256–258
 - scenario validation, 292
 - secondary scenario identification, 285–291
 - selecting, 273–275
 - spiral development, 170
 - state machine specification, 292–295
- user interfaces, 278–279
- user models, 44, 493
- user stories, 493
- utility functions, 10–11, 494

- VAL (validation), 465–469
- validation
 - baseline change, 264–265, 396–397
 - CMMI verification and, 469–470
 - defined, 494
 - design pattern criteria, 336
 - feedback principle, 107–112
 - in microcycle, 165–168

- validation (*cont.*)
 - model organization, 223
 - preparing for, 188
 - proving system correctness, 146–147
 - spiral development, 169–170
 - in spiral development, 178–179
 - spiral development, 189–190
 - subsystem-level use case, 309
- validation (VAL), 465–469
- validation testing
 - overview, 397–401
 - scope, 379
 - workflows, 385–388
- Vaporizer, 139–141
- Var’aq programming language, 327
- velocity
 - defined, 494
 - dynamic planning, 134–136
 - project, 204
- VER (verification), 469–470
- verification, 494
- verification (VER), 469–470
- views
 - Harmony/ESW process, 171–177
 - Harmony’s architectural. *See* Harmony’s architectural views
 - models and, 38
 - schedule updating, 411–413
- Virtual Machine Pattern, 338
- visual modeling, 125–128
- visualization, 37
- Vlissides, John, 331–332
- volume tests, 149, 377

- warp transport, 441
- waterfall approach
 - defined, 494
 - vs. depth-first approach, 27–30
 - vs. incremental approach, 16–19
 - to process, 156
- weight
 - defined, 129
 - unadjusted actor, 205
 - use case, 206

- well-formed, 494
- Westfall, Scott, 4
- white-box analysis
 - bottom-up allocation, 303–307
 - detailing subsystem-level use cases, 308–309
 - overview, 299–301
 - subsystem architecture, 302–303
 - subsystem-level use case validation, 309
 - top-down allocation, 307–308
- Windows, 146
- work items lists
 - BERT method, 202
 - completion charts, 411–413
 - defined, 136–137
 - generated by model review, 419
 - prioritizing elements, 275–276
- work products, 157–158
- workflows
 - agile analysis. *See* agile analysis
 - agile design. *See* agile design
 - agile process optimization. *See* agile process optimization
 - agile testing, 385–388
 - architectural design, 341
 - basic nanocycle, 96
 - continuous integration, 152
 - defined, 157–158
 - feedback principle, 103–112
 - Harmony/ESW process. *See* Harmony/ESW process
 - Harmony/ESW process
 - from PIM to PSM, 50–55
 - working schedules, 210–211
 - working software
 - agile priorities, 1–2
 - attention to quality, 122
 - principles of developing, 92–94
- worst-case performance, 12, 494

- XMI (XML Model Interchange), 70, 494
- XP (Extreme Programming), 1, 486