[HES01] Hesse, W., "Dinosaur Meets Archaeopteryx? Seven Theses on Rational's Unified Process (RUP)," *Proc. 8th Intl. Workshop on Evaluation of Modeling Methods in System Analysis and Design,* Ch. VII, Interlaken, 2001.
[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.
[JAC99] Jacobson, I., Booch, G., and J. Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1999.
[JAC99] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1999.
[KAU95] Kauffman, S., *At Home in the Universe,* Oxford, 1995.
[KER94] Kerr, J., and R. Hunter, *Inside RAD,* McGraw-Hill, 1994.
[KIS02] Kiselev, I., *Aspect-Oriented Programming with AspectJ,* Sams Publishers, 2002.
[MAR91] Martin, J., *Rapid Application Development,* Prentice-Hall, 1991.
[McDE93] McDermid, J., and P. Rook, "Software Development Process Models," in *Software Engineer's Reference Book,* CRC Press, 1993, pp. 15/26–15/28.
[MIL87] Mills, H. D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software,* September, 1987, pp. 19–25.
[NIE92] Nierstrasz, O., S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *CACM,* vol. 35, no. 9, September 1992, pp. 160–165.
[NOG00] Nogueira, J., C. Jones, and Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering," Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, June 2000, download from http://www.dodccrp.org/2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.
[REE02] Reed, P., *Developing Applications with Java and UML,* Addison-Wesley, 2002.
[REI95] Reilly, J. P., "Does RAD Live Up to the Hype," *IEEE Software,* September 1995, pp. 24–26.
[ROO96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes," 1996, available at http://www.imd.ch/fac/roos/paper_po.html.
[ROY70] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. WESCON,* August 1970.
[RUM91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.
[STI01] Stiller, E., and C. LeBlanc, *Project-Based Software Engineering: An Object-Oriented Approach,* Addison-Wesley, 2001.
[WIR90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software,* Prentice-Hall, 1990.
[YOU94] Yourdon, E., "Software Reuse," *Application Development Strategies,* vol. 6, no. 12, December, 1994, pp. 1–16.
[YOU95] Yourdon, E., "When Good Enough Is Best," *IEEE Software,* vol. 12, no. 3, May 1995, pp. 79–81.

PROBLEMS AND POINTS TO PONDER

3.1. Provide three examples of software projects that would be amenable to the incremental model. Be specific.

3.2. Provide three examples of software projects that would be amenable to the prototyping model. Be specific.

3.3. What process adaptations are required if the prototype will evolve into a deliverable system or product?

3.4. To achieve rapid development, the RAD model assumes the existence of one thing. What is it, and why is the assumption not always true?

3.5. Provide three examples of software projects that would be amenable to the waterfall model. Be specific.

3.6. Read [NOG00] and write a two- or three-page paper that discusses the impact of "chaos" on software engineering.

3.7. Is it possible to combine process models? If so, provide an example.

3.8. What is the difference between a UP phase and a UP workflow?

3.9. The concurrent process model defines a set of "states." Describe what these states represent in your own words, and then indicate how they come into play within the concurrent process model.

3.10. What are the advantages and disadvantages of developing software in which quality is "good enough"? That is, what happens when we emphasize development speed over product quality?

3.11. It is possible to prove that a software component and even an entire program is correct. So why doesn't everyone do this?

3.12. Provide three examples of software projects that would be amenable to the component-based model. Be specific.

3.13. Discuss the meaning of "cross-cutting concerns" in your own words. The literature of AOP is expanding rapidly. Do some research and write a brief paper on the current state-of-the-art.

3.14. Are the Unified Process and UML the same thing? Explain your answer.

3.15. As you move outward along the spiral process flow, what can you say about the software that is being developed or maintained?

Further Readings and Information Sources

Most software engineering textbooks consider prescriptive process models in some detail. Books by Sommerville (*Software Engineering,* sixth edition, Addison-Wesley, 2000), Pfleeger (*Software Engineering: Theory and Practice,* Prentice-Hall, 2001), and Schach (*Object-Oriented and Classical Software Engineering,* McGraw-Hill, 2001) consider conventional paradigms and discuss their strengths and weaknesses. Although not specifically dedicated to process, Brooks (*The Mythical Man-Month,* second edition, Addison-Wesley, 1995) presents age-old project wisdom that has everything to do with process. Firesmith and Henderson-Sellers (*The OPEN Process Framework: An Introduction,* Addison-Wesley, 2001) present a general template for creating "flexible, yet disciplined software processes" and discuss process attributes and objectives.

Sharpe and McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development,* Artech House, 2001) present tools for modeling both software and business processes. Jacobson, Griss, and Jonsson (*Software Reuse,* Addison-Wesley, 1997) and McClure (*Software Reuse Techniques,* Prentice-Hall, 1997) present much useful information on component-based development. Heineman and Council (*Component-Based Software Engineering,* Addison-Wesley, 2001) describe the process required to implement component-based systems. Kenett and Baker (*Software Process Quality: Management and Control,* Marcel Dekker, 1999) consider how quality management and process design are intimately connected to one another.

Ambriola (*Software Process Technology,* Springer-Verlag, 2001), Derniame and his colleagues (*Software Process: Principles, Methodology, and Technology,* Springer-Verlag, 1999), and Gruhn and Hartmanis (*Software Process Technology,* Springer-Verlag, 1999) present edited conference proceedings that cover many research and theoretical issues that are relevant to the software process.

Jacobson, Booch, and Rumbaugh have written the seminal book on the Unified Process [JAC99]. However, books by Arlow and Neustadt [ARL02] and a three-volume series by Ambler and Constantine [AMB02] provide excellent complementary information. Krutchen (*The Rational Unified Process,* second edition, Addison-Wesley, 2000) has written a worthwhile introduction to the UP. Project management within the context of the UP is described in detail by Royce (*Software Project Management: A Unified Framework,* Addison-Wesley, 1998). The definitive description of the UP has been developed by the Rational Corporation and is available on-line at www.rational.com.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER

4

AN AGILE VIEW OF PROCESS

KEY CONCEPTS

agile manifesto
agile modeling
agile process models
agility
agility principles
ASD
Crystal
DSDM
Extreme Programming
FDD
pair programming
politics
refactoring
Scrum
team characteristics

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [BEC01a] (referred to as the "Agile Alliance") signed the "Manifesto for Agile Software Development." It stated:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> *Individuals and interactions* over processes and tools
>
> *Working software* over comprehensive documentation
>
> *Customer collaboration* over contract negotiation
>
> *Responding to change* over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

A manifesto is normally associated with an emerging political movement—one that attacks the old guard and suggests revolutionary change (hopefully for the better). In some ways, that's exactly what agile development is all about.

Although the underlying ideas that guide agile development have been with us for many years, it has only been during the past decade that these ideas have crystallized into a "movement." In essence, agile[1] methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it is not applicable to all projects, products, people, and situations. It is also *not*

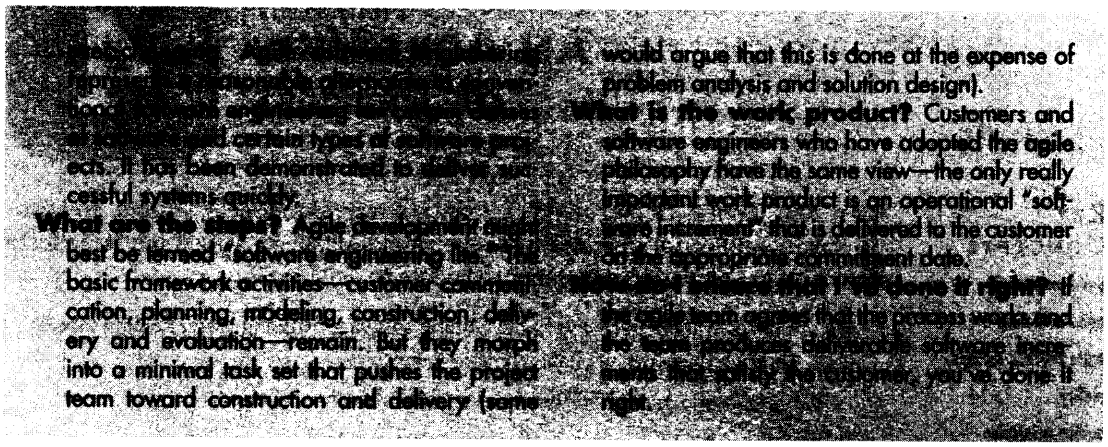
QUICK LOOK

What is it? Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

Who does it? Software engineers and other project stakeholders (managers, customers, end-users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it.

Why is it important? The modern business environment that spawns computer-based system or product needs to adapt to fast-paced and

1 Agile methods are sometimes referred to as *light* or *lean* methods.



antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a Web-based application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, we no longer are able to define requirements fully before the project begins. Software engineers must be agile enough to respond to a fluid business environment.

Does this mean that a recognition of these modern realities causes us to discard valuable software engineering principles, concepts, methods, and tools? Absolutely not! Like all engineering disciplines, software engineering continues to evolve. It can be adapted easily to meet the challenges posed by a demand for agility.

> **"Agility: 1, everything else: 0."**
>
> **Tom DeMarco**

In a thought-provoking book on agile software development, Alistair Cockburn [COC02a] argues that the prescriptive process models introduced in Chapter 3 have a major failing: *they forget the frailties of the people who build computer software.* Software engineers are not robots. They exhibit great variation in working styles and significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not. Cockburn argues that process models can "deal with people's common weaknesses with [either] discipline or tolerance" [COC02a] and that most prescriptive process models choose discipline. He states: "Because consistency in action is a human weakness, high discipline methodologies are fragile" [COC02a].

If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a manner that shows "tolerance" for the people who do software engineering work. Invariably, tolerant practices are easier for software people to adopt and sustain, but

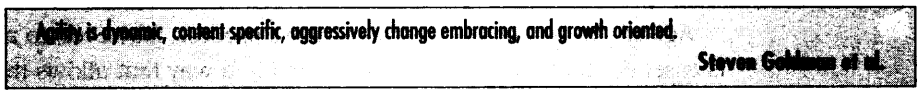
(as Cockburn admits) they may be less productive. Like most things in life, trade-offs must be considered.

4.1 What Is Agility?

Just what is agility in the context of software engineering work? Ivar Jacobson [JAC02] provides a useful discussion:

> *Agility* has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

In Jacobson's view, the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes.



Don't make the mistake of assuming that agility gives you license to hack out solutions. A process is required, and discipline is essential.

But agility is more than an effective response to change. It also encompasses the philosophy espoused in the manifesto noted at the beginning of this chapter. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

The Agile Alliance [AGI03] defines 12 principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

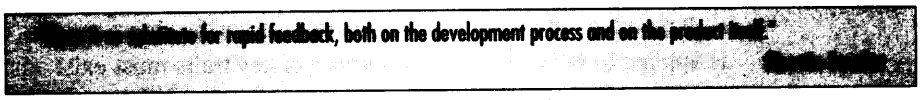
Any *agile software process* is characterized in a manner that addresses three key assumptions [FOW02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as a project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

WebRef
A comprehensive collection of articles on the agile process can be found at www.aanpo.org/articles/index..

Given these three assumptions, an important question arises: How do we create a process that can manage unpredictability? The answer, as we have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable.*

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally.* To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or a portion of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.



4.2.1 The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [HIG02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision-making.

No one is against agility. The real question is: What is the best way to achieve it? As important, how do we build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

You don't have to choose between agility and software engineering. Instead, define a software engineering approach that is agile.

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 4.3), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from conventional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is

much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

The interested reader should see [HIG01], [HIG02a], and [DEM02] for an entertaining summary of the important technical and political issues.

4.2.2 Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors" in successful agile development. As Cockburn and Highsmith [COC01] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that the *process molds to the needs of the people and team,* not the other way around.[2]

> "What results as barely sufficient for one team is either overly sufficient or insufficient for another."
> **Alistair Cockburn**

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

What key traits must exist among the people on an effective software team?

Competence. In an agile development (as well as conventional software engineering) context, "competence" encompasses innate talent, specific software related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help the customer and others understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another, with the customer, and with business managers.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the

2 Most successful software engineering organizations recognize this reality regardless of the process model they choose.

team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers should recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

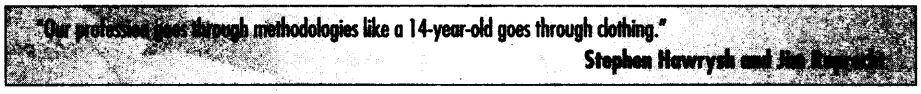
Mutual trust and respect. The agile team must become what DeMarco and Lister [DEM98] call a "jelled" team (see Chapter 21). A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts" [DEM98].

Self-organization. In the context of agile development, *self-organization* implies three things: (1) the agile team organizes itself for the work to be done; (2) the team organizes the process to best accommodate its local environment; (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber [SCH02] addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself."

KEY POINT
A self-organizing team is in control of the work it performs. The team makes its own commitments and defines plans to achieve them.

4.3 AGILE PROCESS MODELS

The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process models—each contending for acceptance within the software development community—the agile movement is following the same historical path.[3]



3 This is not a bad thing. Before one or more models or methods are accepted as a de facto standard, all must contend for the hearts and minds of software engineers. The "winners" evolve into best practice while the "losers" either disappear or merge with the winning models.

In the sections that follow, we present an overview of a number of different *agile process models.* There are many similarities (in philosophy and practice) among these approaches. Our intent will be to emphasize those characteristics of each method that make it unique. It is important to note that *all* agile models conform (to a greater or lesser degree) to the *Manifesto for Agile Software Development* and the principles noted in Section 4.1.

4.3.1 Extreme Programming (XP)

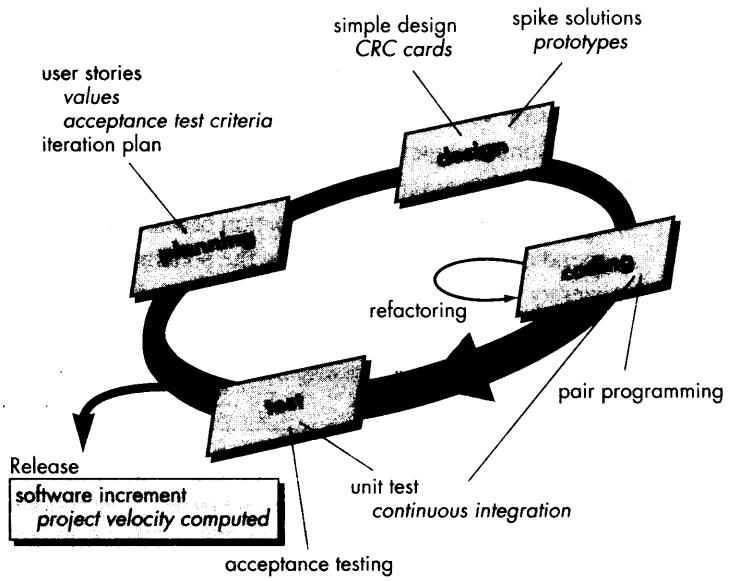
Although early work on the ideas and methods associated with *Extreme Programming (XP)* occurred during the late 1980s, the seminal work on the subject, written by Kent Beck [BEC99] was published in 1999. Subsequent books by Jeffries et al [JEF01] on the technical details of XP, and additional work by Beck and Fowler [BEC01b] on XP planning, flesh out the details of the method.

WebRef
An excellent overview of "rules" for XP can be found at www.extremeprogramming.org/rules.html

XP uses an object-oriented approach (Part 2 of this book) as its preferred development paradigm. XP encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 4.1 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity begins with the creation of a set of *stories* (also called *user stories*) that describe required features and functionality for software to be built. Each story (similar to use-cases described in Chapters 7 and 8) is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a

FIGURE 4.1
The Extreme Programming process



What is an XP "story"?

priority) to the story based on the overall business value of the feature or function.[4] Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story will require more than three development weeks, the customer is asked to split the story into smaller stories, and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

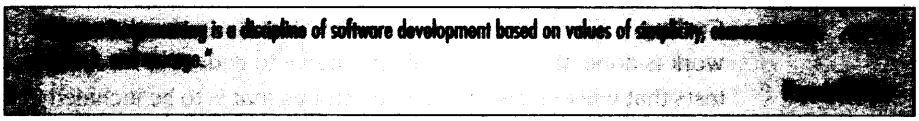
Customers and the XP team work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks); (2) the stories with highest value will be moved up in the schedule and implemented first; or (3) the riskiest stories will be moved up in the schedule and implemented first.

WebRef

A worthwhile XP "planning game" can be found at

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases, and (2) determine whether an over-commitment has been made for all stories across the entire development project. If an over-commitment occurs, the content of releases is modified or end-delivery dates are changed.

As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.



Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.[5]

XP encourages the use of CRC cards (Chapter 8) as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility collaborator) cards identify and organize the object-oriented classes[6] that are relevant to the current software increment. The XP team conducts the design exercise using a

4 The value of a story may also depend on the presence of another story.

5 These design guidelines should be followed in every software engineering method, although there are times when sophisticated design notation and terminology may get in the way of simplicity.

6 Object-oriented classes are discussed in detail in Chapter 8 and throughout Part 2 of this book.

process similar to the one described in Chapter 8 (Section 8.7.4). The CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution,* the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem.

WebRef
Refactoring techniques and tools can be found at www.refactoring.com.

XP encourages *refactoring*—a construction technique that is also a design technique. Fowler [FOW00] describes refactoring in the following manner:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [and modify/simplify the internal design] that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

Because XP design uses virtually no notation and produces few, if any work products other than CRC cards and spike solutions, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes that "can radically improve the design" [FOW00]. It should be noted, however, that effort required for refactoring can grow dramatically as the size of an application grows.

WebRef
Useful information on XP can be obtained at www.xprogramming.com.

A central notion in XP is that design occurs both before *and after* coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

Coding. XP recommends that after stories are developed and preliminary design work is done, the team should not move to code, but rather develop a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).[7] Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the unit test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit tested immediately, thereby providing instantaneous feedback to the developers.

What is pair programming?

A key concept during the coding activity (and one of the most talked about aspects of XP) is *pair programming.* XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance. It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures

7 This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

that coding standards (a required part of XP) are being followed and the code that is generated will "fit" into the broader design for the story.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment (Chapter 13) that helps to uncover errors early.

Testing. We have already noted that the creation of a unit test[8] before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 13) whenever code is modified (which is often, given the XP refactoring philosophy).

As the individual unit tests are organized into a "universal testing suite" [WEL99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things are going awry. Wells [WEL99] states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

XP acceptance tests are derived from user stories.

XP *acceptance tests,* also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

SAFEHOME

Considering Agile Software Development

The scene: Doug Miller's office.

The players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation:

(A knock on the door)

Jamie: Doug, you got a minute?

Doug: Sure Jamie, what's up?

Jamie: We've been thinking about our process discussion yesterday . . . you know, what process we're going to choose for this new *SafeHome* project.

Doug: And?

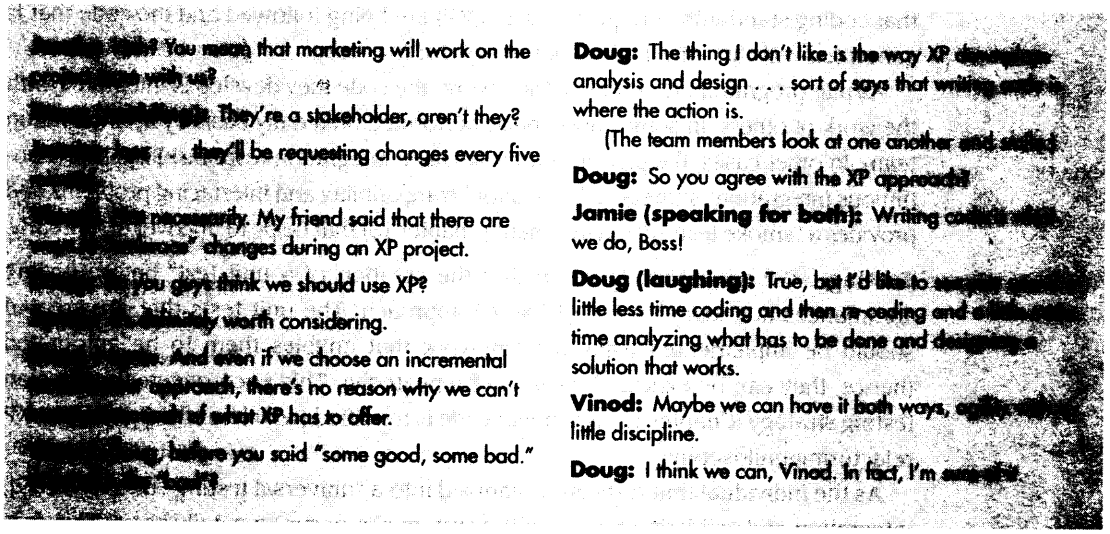
Vinod: I was talking to a friend at another company, and he was telling me about Extreme Programming. It's an agile process model, heard of it?

Doug: Yeah, some good, some bad.

Jamie: Well, it sounds pretty good to us. Lets you develop software really fast, uses something called programming to do real-time quality checks, cool, I think.

Doug: It does have a lot of really good ideas, pair programming concept, for instance, and it's that stakeholders should be part of the team.

8 Unit testing, discussed in detail in Chapter 13, focuses on an individual software component, exercising the component's interface, data structures, and functionality in an effort to uncover errors that are local to the component.



4.3.2 Adaptive Software Development (ASD)

Adaptive Software Development (ASD) has been proposed by Jim Highsmith [HIG00] as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization. Highsmith [HIG98] discusses this when he writes:

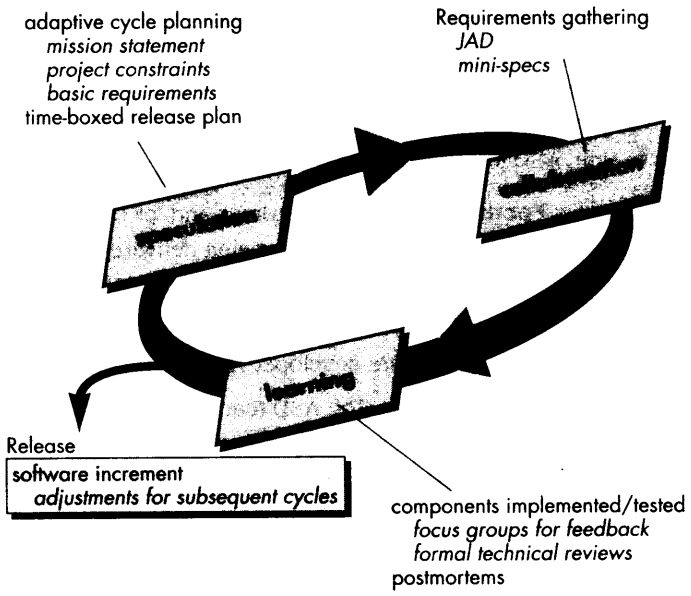
> Self-organization is a property of complex adaptive systems similar to a collective "aha," that moment of creative energy when the solution to some nagging problem emerges. Self-organization arises when individual, independent agents (cells in a body, species in an ecosystem, developers in a feature team) cooperate [collaborate] to create emergent outcomes. An emergent outcome is a property beyond the capability of any individual agent. For example, individual neurons in the brain do not possess consciousness, but collectively the property of consciousness emerges. We tend to view this phenomena of collective emergence as accidental, or at least unruly and undependable. The study of self-organization is proving that view to be wrong.

WebRef

Highsmith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" (Figure 4.2) that incorporates three phases: speculation, collaboration, and learning.

Speculation. During *speculation,* the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descrip-

FIGURE 4.2
Adaptive software development



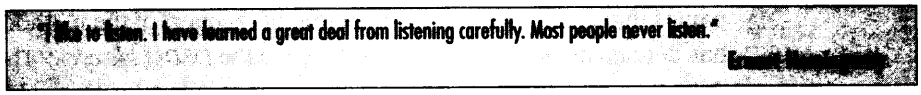
What are the characteristics of ASD adaptive cycles?

tions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.[9]

Collaboration. Motivated people work together in a way that multiplies their talent and creative output beyond their absolute numbers. This collaborative approach is a recurring theme in all agile methods. But collaboration is not easy. It is not simply communication, although communication is a part of it. It is not only a matter of teamwork, although a "jelled" team (Chapter 21) is essential for real collaboration to occur. It is not a rejection of individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity; (2) assist without resentment; (3) work as hard or harder as they do; (4) have the skill set to contribute to the work at hand; and (5) communicate problems or concerns in a way that leads to effective action.

ADVICE

Effective collaboration with your customer will only occur if you jettison any "us and them" attitudes.



9 Note that the adaptive cycle plan can and probably will be adapted to changing project and business conditions.

Learning. As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on learning as much as it is on progress toward a completed cycle. In fact, Highsmith [HIG00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways:

1. **Focus groups.** The customer and/or end-users provide feedback on software increments that are being delivered. This provides a direct indication of whether or not the product is satisfying business needs.
2. **Formal technical reviews.** ASD team members review the software components that are developed, improving quality and learning as they proceed.
3. **Postmortems.** The ASD team becomes introspective, addressing its own performance and process (with the intent of learning and then improving its approach).

It is important to note that the ASD philosophy has merit regardless of the process model that is used. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

4.3.3 Dynamic Systems Development Method (DSDM)

WebRef
Useful resources for DSDM can be found at www.dsdm.org.

The *Dynamic Systems Development Method* (DSDM) [STA97] is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" [CCS02]. Similar in some respects the RAD process discussed in Chapter 3, DSDM suggests a philosophy that is borrowed from a modified version of the Pareto principle. In this case, 80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

Like XP and ASD, DSDM suggests an iterative software process. However, the DSDM approach to each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The DSDM Consortium (www.dsdm.org) is a worldwide group of member companies that collectively take on the role of "keeper" of the method. The consortium has defined an agile process model, called the *DSDM life cycle*. The DSDM life cycle defines three different iterative cycles, preceded by two additional life cycle activities:

WebRef
A useful overview of DSDM can be found at www.cs3inc.com/DSDM.htm.

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer (note: all DSDM prototypes are intended to evolve into the deliverable application). The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration—revisits prototypes built during the functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end-users. In some cases, the functional model iteration and the design and build iteration occur concurrently.

Implementation—places the latest software increment (an "operationalized" prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the function model iteration activity.

DSDM can be combined with XP to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments. In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

4.3.4 Scrum

Scrum (the name derived from an activity[10] that occurs during a rugby match) is an agile process model that was developed by Jeff Sutherland and his team in the early 1990s. In recent years, further development of the Scrum methods has been performed by Schwaber and Beedle [SCH01]. Scrum principles [ADM96] are consistent with the agile manifesto:

- Small working teams are organized to "maximize communication, minimize overhead, and maximize sharing of tacit, informal knowledge."
- The process must be adaptable to both technical and business changes "to ensure the best possible product is produced."
- The process yields frequent software increments "that can be inspected, adjusted, tested, documented, and built on."
- Development work and the people who perform it are partitioned "into clean, low coupling partitions, or packets."
- Constant testing and documentation is performed as the product is built.

---

10 A group of players forms around the ball and the teammates work together (sometimes violently!) to move the ball downfield.

- The Scrum process provides the "ability to declare a product 'done' whenever required (because the competition just shipped, because the company needs the cash, because the user/customer needs the functions, because that was when it was promised. . . ." [ADM96].

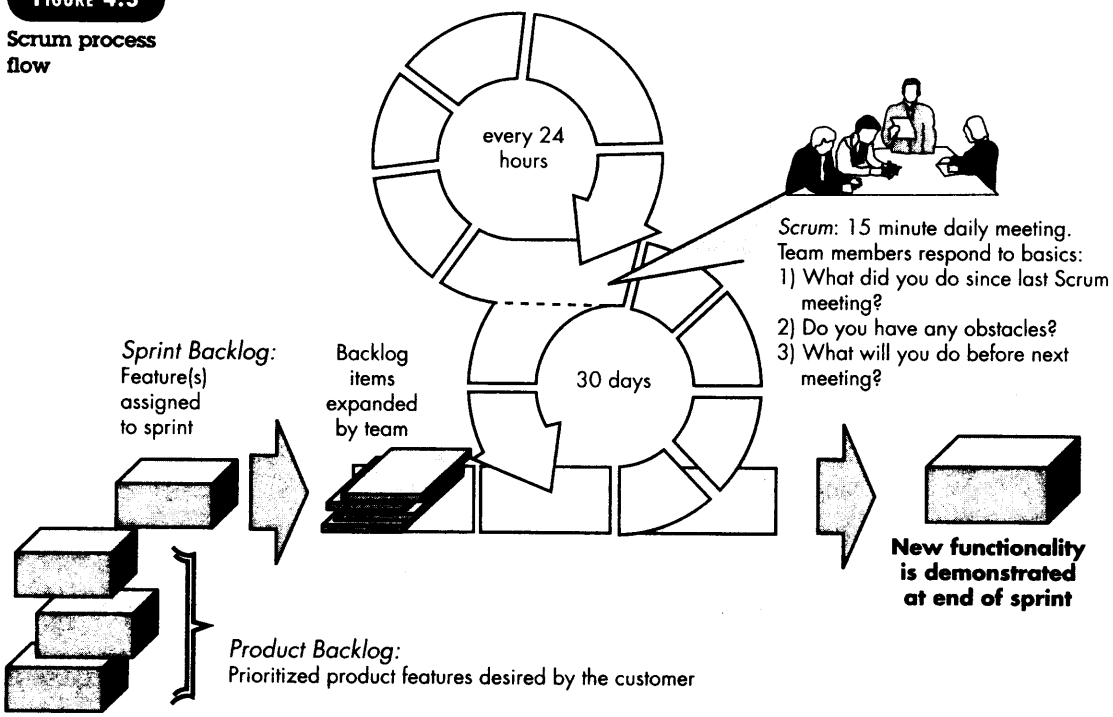
WebRef
Useful Scrum information and resources can be found at www.controlchaos.com.

Scrum principles are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern (discussed in the following paragraph) called a *sprint*. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real-time by the Scrum team. The overall flow of the Scrum process is illustrated in Figure 4.3.

"Scrum allows us to build softer software."
Mike Smith et al.

Scrum emphasizes the use of a set of "software process patterns" [NOY02] that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development activities:

FIGURE 4.3
Scrum process flow



Scrum incorporates a set of process patterns that emphasize project priorities, compartmentalized work units, communication, and frequent customer feedback.

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days). During the sprint, the backlog items that the sprint work units address are frozen (i.e., changes are not introduced during the sprint). Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members [NOY02]:

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

A team leader, called a "Scrum master," leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" [BEE99] and thereby promote a self-organizing team structure.

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Beedle and his colleagues [BEE99] present a comprehensive discussion of these patterns in which they state: "SCRUM assumes up-front the existence of chaos. . . ." The Scrum process patterns enable a software development team to work successfully in a world where the elimination of uncertainty is impossible.

4.3.5 Crystal

Alistair Cockburn [COC02a] and Jim Highsmith [HIG02b] created the *Crystal family of agile methods*[11] in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource-limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game" [COC02b].

To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process

11 The name "crystal" is derived from the characteristics of geological crystals, each with its own color, shape, and hardness.

WebRef
A comprehensive discussion of Crystal can be found at www.crystalmethodologies.org.

patterns, work products, and practice that are unique to each. The Crystal family is actually a set of agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

4.3.6 Feature Driven Development (FDD)

WebRef
A wide variety of articles and presentations on FDD can be found at www.thecoadletter.com.

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues [COA99] as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing [PAL02] have extended and enhanced Coad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" [COA99]. The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily, understand how they relate to one another more readily, and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues [COA99] suggest the following template for defining a feature:

<action> the **<result>** **<by | for | of | to>** a(n) **<object>**

where an **<object>** is "a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)." Examples of features for an e-commerce application might be:

Add the product to a shopping cart.

Display the technical-specifications of a product.

Store the shipping-information for a customer.

A feature set groups related features into business-related categories and is defined [COA99] as:

<action><-ing> a(n) **<object>**

For example: *Making a product sale* is a feature set that would encompass the features noted earlier and others.

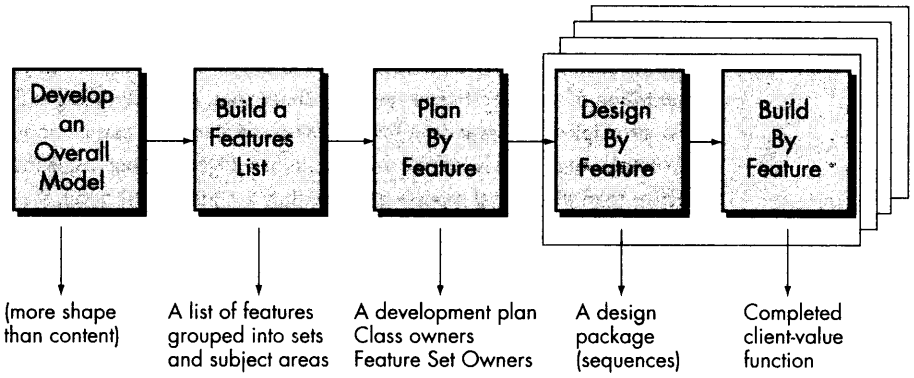


FIGURE 4.4
Feature Driven Development [COA99] (used with permission)

The FDD approach defines five "collaborating" [COA99] framework activities (in FDD these are called "processes") as shown in Figure 4.4.

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and the customer to understand project status—what accomplishments have been made and problems have been encountered. If deadline pressure is significant, it is critical to determine if software increments (features) are properly scheduled. To accomplish this, FDD defines six milestones during the design and implementation of a feature: "design walkthrough, design, design inspection, code, code inspection, promote to build" [COA99].

4.3.7 Agile Modeling (AM)

WebRef
Comprehensive information on agile modeling can be found at **www.agilemodeling.com.**

There are many situations in which software engineers must build large, business-critical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished; (2) the problem can be partitioned effectively among the people who must solve it; and (3) quality can be assessed at every step as the system is being engineered and built.

Over the past 30 years, a wide variety of software engineering modeling methods and notation have been proposed for analysis and design (both architectural and component-level). These methods have significant merit, but they have proven difficult to apply and challenging to sustain (over many projects). Part of the problem is the "weight" of these modeling methods. By this we mean the volume of notation required, the degree of formalism suggested, the size of the models for large projects, and the difficulty in maintaining the model as changes occur. Yet analysis and design modeling have substantial benefit for large projects—if for no other reason than to make these projects intellectually manageable. Is there an agile approach to software engineering modeling that might provide an alternative?

At "The Official Agile Modeling Site," Scott Ambler [AMB02] describes *Agile Modeling* (AM) in the following manner:

> Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.

In addition to the values that are consistent with the agile manifesto, Ambler suggests *courage* and *humility*. An agile team must have the courage to make decisions that may cause it to reject a design and refactor. It must have the humility to recognize that technologists do not have all the answers, that business experts and other stakeholders should be respected and embraced.

Although AM suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are [AMB02]:

Model with a purpose. A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

Use multiple models. There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

"Traveling light" is an appropriate philosophy for all software engineering work. Build only those models that provide value—no more, no less.

Travel light. As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler [AMB02] notes that "every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner (hence potentially enhancing communication within your team as well as with project stakeholders)."

Content is more important than representation. Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.

Know the models and the tools you use to create them. Understand the strengths and weaknesses of each model and the tools that are used to create it.

Adapt locally. The modeling approach should be adapted to the needs of the agile team.

SOFTWARE TOOLS

Agile Development

Objective: The objective of agile development tools is to assist in one or more aspects of agile development with an emphasis on facilitating the rapid generation of operational software. These tools can also be used when prescriptive process models (Chapter 3) are applied.

Mechanics: Tool mechanics vary. In general, agile tool sets encompass automated support for project planning, use-case development and requirements gathering, rapid design, code generation, and testing.

Representative Tools:[12]

Note: Because agile development is a hot topic, most software tools vendors purport to sell tools that support the agile approach. The tools noted below have characteristics that make them particularly useful for agile projects.

Actif Extreme, developed by Microtool (www.microtool.com), provides agile process management support for various technical activities within the process.

Ideogramic UML, developed by Ideogramic (www.ideogramic.com), is a UML toolset specifically developed for use within an agile process.

Together Tool Set, distributed by Borland (www.borland.com or www.togethersoft.com), provides a tools suite that supports many technical activities within XP and other agile processes.

4.4 SUMMARY

An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform; communication and collaboration between team members and between practitioners and their customers; a recognition that change represents an opportunity; and an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

Extreme Programming (XP) is the most widely used agile process. Organized as four framework activities—planning, design, coding, and testing—XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases delivering features and functionality that have been described and then prioritized by the customer.

Adaptive Software Development (ASD) stresses human collaboration and team self-organization. Organized as three framework activities—speculation, collaboration, and learning—ASD uses an iterative process that incorporates adaptive cycle planning, relatively rigorous requirements gathering methods, and an iterative development cycle that incorporates customer focus groups and formal technical reviews as real-time feedback mechanisms. The Dynamic Systems Development Method (DSDM) defines three different iterative cycles—functional model iteration, design and build iteration, and implementation—preceded by two additional life cycle activities—feasibility study and business study. DSDM advocates the use of time-

12 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

box scheduling and suggests that only enough work is required for each software increment to facilitate movement to the next increment.

Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the Scrum team to construct a process that is adapted to the needs of the project.

Crystal is a family of agile process models that can be adopted to the specific characteristics of a project. Like other agile approaches, Crystal adopts an iterative strategy but adjusts the rigor of the process to accommodate projects of different sizes and complexities.

Feature Driven Development (FDD) is somewhat more "formal" than other agile methods, but still maintains agility by focusing the project team on the development of features—client-valued functions that can be implemented in two weeks or less. FDD provides greater emphasis on project and quality management than other agile approaches. Agile Modeling (AM) suggests that modeling is essential for all systems, but that the complexity, type, and size of the model must be tuned to the software to be built. By proposing a set of core and supplementary modeling principles, AM provides useful guidance for the practitioner during analysis and design tasks.

REFERENCES

[ADM96] Advanced Development Methods, Inc., "Origins of Scrum," 1996, http://www.controlchaos.com/.

[AGI03] The Agile Alliance Home Page, http://www.agilealliance.org/home.

[AMB02] Ambler, S., "What Is Agile Modeling (AM)?" 2002, http://www.agilemodeling.com/index.htm.

[BEC99] Beck, K., *Extreme Programming Explained: Embrace Change,* Addison-Wesley, 1999.

[BEC01a] Beck, K., et al., "Manifesto for Agile Software Development," http://www.agilemanifesto. org/.

[BEC01b] Beck, K., and M. Fowler, *Planning Extreme Programming,* Addison-Wesley, 2001.

[BEE99] Beedle, M., et al., "SCRUM: An extension pattern language for hyperproductive software development," included in: *Pattern Languages of Program Design 4,* Addison-Wesley Longman, Reading, MA, 1999. Download at http://jeffsutherland.com/scrum/scrum_plop.pdf.

[BUS00] Buschmann, F., et al., *Pattern-Oriented Software Architecture,* 2 volumes, Wiley, 1996, 2000.

[COA99] Coad, P., E. Lefebvre, and J. DeLuca, *Java Modeling in Color with UML,* Prentice-Hall, 1999.

[COC01] Cockburn, A., and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer,* vol. 34, no. 11, November 2001, pp. 131–133.

[COC02a] Cockburn, A., *Agile Software Development,* Addison-Wesley, 2002.

[COC02b] Cockburn, A., "What Is Agile and What Does It Imply?" presented at the Agile Development Summit at Westminster College in Salt Lake City, March 2002, http://crystalmethodologies.org/.

[CCS02] CS3 Consulting Services, 2002, http://www.cs3inc.com/DSDM.htm.

[DEM98] DeMarco, T., and T. Lister, *Peopleware,* 2nd ed., Dorset House, 1998.

[DEM02] DeMarco, T., and B. Boehm, "The Agile Methods Fray," *IEEE Computer,* vol. 35, no. 6, June 2002, pp. 90–92.

[FOW00] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 2000.

[FOW01] Fowler M., and J. Highsmith, "The Agile Manifesto," *Software Development Magazine,* August 2001, http://www.sdmagazine.com/documents/s=844/ sdm0108a/0108a.htm.

[FOW02] Fowler. M., "The New Methodology," June 2002, http://www.martinfowler.com/articles/newMethodology.html#N8B.
[HIG98] Highsmith, J., "Life—The Artificial and the Real," *Software Development,* 1998, at http://www.adaptivesd.com/articles/order.html.
[HIG00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems,* Dorset House Publishing, 1998.
[HIG01] Highsmith, J., ed., "The Great Methodologies Debate: Part 1," *Cutter IT Journal,* vol. 14, no. 12, December 2001.
[HIG02a] Highsmith, J., ed., "The Great Methodologies Debate: Part 2," *Cutter IT Journal,* vol. 15, no. 1, January 2002.
[HIG02b] Highsmith, J., *Agile Software Development Ecosystems,* Addison-Wesley, 2002.
[JAC02] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More," *Cutter IT Journal,* vol. 15, no. 1., January 2002, pp. 18–24.
[JEF01] Jeffries, R., et al., *Extreme Programming Installed,* Addison-Wesley, 2001.
[NOY02] Noyes, B., "Rugby, Anyone?" *Managing Development* (an on-line publication of Fawcette Technical Publications), June 2002, http://www.fawcette.com/resources/managingdev/methodologies/scrum/.
[PAL02] Palmer, S., and J. Felsing, *A Practical Guide to Feature Driven Development,* Prentice-Hall, 2002.
[SCH01] Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM,* Prentice-Hall, 2001.
[SCH02] Schwaber, K., "Agile Processes and Self-Organization," Agile Alliance, 2002, http://www.aanpo.org/articles/index.
[STA97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice,* Addison-Wesley, 1997.
[WEL99] Wells, D., "XP—Unit Tests," 1999, http://www.extremeprogramming.org/ rules/unittests.html.

Problems and Points to Ponder

4.1. Select one agility principle noted in Section 4.1 and try to determine whether each of the process models presented in this chapter exhibits the principle.

4.2. Try to come up with one more "agility principle" that would help a software engineering team become even more maneuverable.

4.3. Could each of the agile processes be described using the generic framework activities noted in Chapter 2? Build a table that maps the generic activities into the activities defined for each agile process.

4.4. Why does an iterative process make it easier to manage change? Is every agile process discussed in this chapter iterative? Is it possible to complete a project in just one iteration and still be agile? Explain your answers.

4.5. Describe agility (for software projects) in your own words.

4.6. Reread "The Manifesto for Agile Software Development" at the beginning of this chapter. Can you think of a situation in which one or more of the four "values" could get a software team into trouble?

4.7. Why do requirements change so much? After all, don't people know what they want?

4.8. Most agile process models recommend face-to-face communication. Yet today, members of a software team and their customers may be geographically separated from one another. Do you think this implies that geographical separation is something to avoid? Can you think of ways to overcome this problem?

4.9. Consider the seven traits noted in Section 4.2.2. Order the traits based on your perception of which is most important to which is least important.

4.10. Write an XP user story that describes the "favorite places" or "favorites" feature available on most Web browsers.

4.11. Visit the Official Agile Modeling Site and make a complete list of all core and supplementary AM principles.

4.12. Describe the XP concepts of *refactoring* and *pair programming* in your own words.

4.13. Why is Crystal called a *family of agile methods*?

4.14. Using the process pattern template presented in Chapter 2, develop a process patterns for any one of the Scrum patterns presented in Section 4.3.4.

4.15. Using the FDD feature template described in Section 4.3.6, define a feature set for a Web browser. Now develop a set of features for the feature set.

4.16. What is a spike solution in XP?

Further Readings and Information Sources

The overall philosophy and underlying principles of agile software development are considered in depth in books by Ambler (*Agile Modeling*, Wiley, 2002), Beck [BEC99], Cockburn [COC02], and Highsmith [HIG02b].

Books by Beck [BEC99], Jeffries and his colleagues (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi and Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk and Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001), and Auer and his colleagues (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) provide a nuts and bolts discussion of XP along with guidance on how best to apply it. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) takes a critical look at XP, defining when and where it is appropriate. An in-depth consideration of pair programming is presented by McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

Fowler and his colleagues (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) address the important XP concept of refactoring in considerable detail. McBreen (*Software Craftsmanship: The New Imperative*, Addison-Wesley, 2001) discusses software craftsmanship and argues for agile alternatives to traditional software engineering

ASD is addressed in depth by Highsmith [HIG00]. A worthwhile treatment of DSDM has been written by Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Palmer and Felsing [PAL02] present a detailed treatment of FDD. Carmichael and Haywood (*Better Software Faster*, Prentice-Hall, 2002) presents another useful treatment of FDD that includes a step-by-step journey through the mechanics of the process. Schwaber and his colleagues (*Agile Software Development with SCRUM*, Prentice-Hall, 2001) present a detailed treatment of Scrum.

Martin (*Agile Software Development*, Prentice-Hall, 2003) discusses agile principles, patterns, and practices with an emphasis on XP. Poppendieck and Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) provide guidelines for managing and controlling agile projects. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) presents a worthwhile survey of agile principles, processes, and practices.

A wide variety of information sources on agile software development are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the agile process can be found at the SEPA Web site:
http://www.mhhe.com/ pressman.

Two

Software Engineering Practice

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the principles, concepts, and methods that comprise software engineering practice. These questions are addressed in the chapters that follow:

- What concepts and principles guide software engineering practice?
- How does system engineering lead to effective software engineering?
- What is requirements engineering, and what are the underlying concepts that lead to good requirements analysis?
- How is the analysis model created, and what are its elements?
- What is design engineering, and what are the underlying concepts that lead to good design?
- What concepts, models, and methods are used to create architectural, interface, and component-level designs?
- What strategies are applicable to software testing?
- What methods are used to design effective test cases?
- What measures and metrics can be used to assess the quality of analysis and design models, source code, and test cases?

Once these questions are answered you'll be better prepared to apply software engineering practice.

CHAPTER

5 SOFTWARE ENGINEERING PRACTICE

KEY CONCEPTS

principles of:
- **agile modeling**
- **analysis**
- **coding**
- **communication**
- **deployment**
- **design**
- **planning**
- **software engineering**
- **testing**

problem solving

W^{5}HH questions

In a book that explores the lives and thoughts of software engineers, Ellen Ullman [ULL97] depicts a slice of life as she relates the thoughts of practitioner under pressure:

> I have no idea what time it is. There are no windows in this office and no clock, only the blinking red LED display of a microwave, which flashes 12:00, 12:00, 12:00, 12:00. Joel and I have been programming for days. We have a bug, a stubborn demon of a bug. So the red pulse no-time feels right, like a read-out of our brains, which have somehow synchronized themselves at the same blink rate. . . .
>
> What are we working on? . . . The details escape me just now. We may be helping poor sick people or tuning a set of low-level routines to verify bits on a distributed database protocol—I don't care. I should care; in another part of my being—later, perhaps when we emerge from this room full of computers—I will care very much why and for whom and for what purpose I am writing software. But just now: no. I have passed through a membrane where the real world and its uses no longer matter. I am a software engineer. . . .

A dark image of software engineering practice to be sure, but upon reflection, many of the readers of this book will be able to relate to it.

QUICK LOOK

What is it? The practice of software engineering is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed. It represents the details—the technical considerations and how-to's—that are below the surface of the software process: the things that you'll need to actually build high-quality computer software.

Who does it? The practice of software engineering is applied by software engineers and their managers.

Why is it important? The software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a destination successfully. Practice provides you with the detail you'll need to drive along the road. It tells you where the bridges, the roadblocks, and the forks are located. It helps you understand the concepts and principles that must be understood and followed to drive safely and rapidly. It instructs you on how to drive, where to slow down, and where to speed up. In the context of software engineering, practice is what you do day in and day out as software evolves from an idea to a reality.

What are the steps? Three elements of practice apply regardless of the process model that is chosen. They are: concepts, principles, and methods. A fourth element of practice—tools—supports the application of methods.

What is the work product? Practice encompasses the thoughts, decisions, and models that produce all work products created by the software engineering team.

How do I ensure that I've done it right?
First, have a firm understanding of the concepts and principles that apply to the work (e.g., design) that you're doing at the moment. Then, be certain that you've chosen an appropriate method for the work, be sure that you understand how to apply the method and use automated tools when they're appropriate for the task, and be adamant about the need for techniques to ensure the quality of work products that are produced.

People who create computer software practice the art or craft or discipline[1] that is software engineering. But what is software engineering "practice"? In a generic sense, *practice* is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis. Practice allows managers to manage software projects and software engineers to build computer programs. Practice populates a software process model with the necessary technical and management how-to's to get the job done. Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

5.1 Software Engineering Practice

WebRef
A variety of thought-provoking quotes on the practice of software engineering can be found at **www.literateprogramming.com.**

In Chapter 2, we introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—communication, planning, modeling, construction, and deployment—and umbrella activities establish a skeleton architecture for software engineering work. All of the software process models presented in Chapters 3 and 4 can be mapped into this skeleton architecture. But how does the practice of *software engineering* fit in? In the sections that follow, we consider the generic concepts and principles that apply to framework activities.[2]

5.1.1 The Essence of Practice

You might argue that Polya's approach is simply common sense. True. But it's amazing how often common sense is uncommon in the software world.

In a classic book, *How to Solve It,* written before modern computers existed, George Polya [POL45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

---

1 Some writers argue for one of these terms to the exclusion of the others. In reality, software engineering is all three.

2 The reader is encouraged to revisit relevant sections within this chapter as specific software engineering methods and umbrella activities are discussed later in this book.

In the context of software engineering, these common sense steps lead to a series of essential questions [adapted from POL45]:

Understand the problem.

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, features, and behavior are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the solution.

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, features, and behavior that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry out the plan.

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution probably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, features, and behavior that are required?* Has the software been validated against all stakeholder requirements?

"There is a grain of discovery in the solution of any problem."
George Polya.

5.1.2 Core Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book we discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., customer communication), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help us establish a mind set for solid software engineering practice. They are important for that reason.

David Hooker [HOO96] has proposed seven core principles that focus on software engineering practice as a whole. They are reproduced below:[3]

Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users.* All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: Does this add real value to the system? If the answer is no, don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler.* This facilitates having a more easily understood, and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The pay-off is software that is more maintainable and less error-prone.

> **"There is a certain majesty in simplicity which is far above all the quaintness of wit."**
>
> **Alexander Pope (1688–1744)**

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . .

Compromising the architectural vision of a software system weakens and will eventually break even a well-designed system. Having an empowered architect

---

3 Reproduced with permission of the author [HOO96]. Hooker defines patterns for these principles at: http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.

who can hold the vision and enforce compliance helps ensure a very successful software project.

KEY POINT

If software has value, it will change over its useful life. For that reason, software must be built to be maintainable.

The Fourth Principle: *What You Produce, Others Will Consume*

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing.* The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those who must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: *Be Open to the Future*

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete after just a few months, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner.* Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.[4] This could very possibly lead to the reuse of an entire system.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[5] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. Those at the detailed design and code level are well known and documented. New literature is addressing the reuse of design in the form of software patterns. However, this is just part of the battle.

---

4 Author's note: This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can require more project effort.

5 Author's note: Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justified.

Communicating opportunities for reuse to others in the organization is paramount. How can you reuse something that you don't know exists? *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh Principle: *Think!*

This last Principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results.* When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six Principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

5.2 Communication Practices

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a *communication* (also called *requirements elicitation*) activity. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help. Communication has begun. But the road from communication to understanding is often full of potholes.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront a software engineer. In this context, we discuss communication principles and concepts as they apply to customer communication. However, many of the principles apply equally to all forms of communication that occur within a software project.

Before communicating be sure you understand the point of view of the other party, know a bit about his or her needs, and then listen.

Principle #1: *Listen.* Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. Never become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle #2: *Prepare before you communicate.* Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle #3: ***Someone should facilitate the activity.*** Every communication meeting should have a leader (facilitator) to keep the conversation moving in a productive direction; (2) to mediate any conflict that does occur; (3) to ensure than other principles are followed.

Principle #4: ***Face-to-face communication is best.*** But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

> "Plain questions and plain answers make the shortest road to most perplexities."
>
> **Mark Twain**

Principle #5: ***Take notes and document decisions.*** Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Principle #6: ***Strive for collaboration.*** Collaboration and consensus occur when the collective knowledge of members of the team is combined to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Principle #7: ***Stay focused, modularize your discussion.*** The more people involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved (however, see Principle #9)

INFO

The Difference Between Customers and End-Users

Software engineers communicate with many different stakeholders, but customers and end-users have the most significant impact on the technical work that follows. In some cases the customer and the end-user are one in the same, but for many projects, the customer and the end-user are different people, working for different managers in different business organizations.

A *customer* is the person or group who: (1) originally requested the software to be built; (2) defines overall business objectives for the software; (3) provides basic product requirements; and (4) coordinates funding for the project. In a product or system business, the customer is often the marketing department. In an IT environment, the customer might be a business component or department.

An *end-user* is the person or group who: (1) will actually use the software that is built to achieve some business purpose, and (2) will define operational details of the software so the business purpose can be achieved.

Principle #8: ***If something is unclear, draw a picture.*** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

Principle #9: ***(a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and can-***

What happens if I can't come to an agreement with the customer on some project-related issue?

not be clarified at the moment, move on. Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle #2) and that "moving on" is sometimes the best way to achieve communication agility.

Principle #10: ***Negotiation is not a contest or a game. It works best when both parties win.*** There are many instances in which the software engineer and the customer must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Therefore, negotiation will demand compromise from all parties.

SAFEHOME

Communication Mistakes

The scene: Software engineering team workspace.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins software team member.

The conversation:

Ed: What have you heard about this *SafeHome* project?

Vinod: The kick-off meeting is scheduled for next week.

Jamie: I've already done a little bit of investigation, but it didn't go well."

Ed: What do you mean?

Jamie: Well, I gave Lisa Perez a call. She's the marketing honcho on this thing."

Vinod: And . . . ?

Jamie: I wanted her to tell me about *SafeHome* features and functions . . . that sort of thing. Instead, she began asking me questions about security systems, surveillance systems . . . I'm no expert.

Vinod: What does that tell you?

(Jamie shrugs.)

Vinod: That marketing will need us to act as consultants and that we'd better do some homework on this product area before our kick-off meeting. Doug said that he wanted us to "collaborate" with our customer, so we'd better learn how to do that.

Ed: Probably would have been better to stop by her office. Phone calls just don't work as well for this sort of thing.

Jamie: You're both right. We've got to get our act together or our early communications will be a struggle.

Vinod: I saw Doug reading a book on "requirements engineering." I'll bet that lists some principles of good communication. I'm going to borrow it from him.

Jamie: Good idea . . . then you can teach us.

Vinod (smiling): Yeah, right.

TASK SET

Generic Task Set for Communication

1. Identify primary customer and other stakeholders (Section 7.3.1).
2. Meet with primary customer to address "context free questions" (Section 7.3.4) that define:
   - Business need and business values.
   - End-users' characteristics/needs.
   - Required user-visible outputs.
   - Business constraints.

3. Develop a one-page written statement of project scope that is subject to revision (Sections 7.4.1 and 21.3.1).
4. Review statement of scope with stakeholders and amend as required.
5. Collaborate with customer/end-users to define:
   - Customer visible usage scenarios using standard format[6] (Section 7.5).
     - Resulting outputs and inputs.
     - Important software features, functions, and behavior.
     - Customer-defined business risks (Section 25.3).
6. Develop a brief written description (e.g., a set of lists) of scenarios, output/inputs, features/functions and risks.
7. Iterate with customer to refine scenarios, output/inputs, features/functions and risks.
8. Assign customer-defined priorities to each user scenario, feature, function, and behavior. (Section 7.4.2).
9. Review all information gathered during the communication activity with the customer and other stakeholders and amend as required.
10. Prepare for planning activity (Chapters 23 and 24).

5.3 PLANNING PRACTICES

The communication activity helps a software team to define its overall goals and objectives (subject, of course, to change as time passes). However, understanding these goals and objectives is not the same as defining a plan for getting there. The *planning* activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives.

> "In preparing for battle I have always found that plans are useless, but planning is indispensable."
> **Dwight D. Eisenhower**

There are many different planning philosophies. Some people are "minimalists," arguing that change often obviates the need for a detailed plan. Others are "traditionalists," arguing that the plan provides an effective road map, and the more detail it has, the less likely the team will become lost. Still others are "agilists," arguing that a quick "planning game" may be necessary, but that the road map will emerge as "real work" on the software begins.

What to do? On many projects, overplanning is time consuming and fruitless (too many things change), but underplanning is a recipe for chaos. Like most things in life, planning should be conducted in moderation, enough to provide useful guidance for the team—no more, no less.

WebRef
An excellent repository of planning and project management information can be found at **www.4pm.com/repository.htm.**

Regardless of the rigor with which planning is conducted, the following principles always apply.

Principle #1: ***Understand the scope of the project.*** It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

6 Formats for usage scenarios are discussed in Chapter 8.

Principle #2: ***Involve the customer in the planning activity.*** The customer defines priorities and establishes project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, timelines, and other project related issues.

Principle #3: ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate replanning (after the delivery of each software increment) based on feedback received from users.

Principle #4: ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

Principle #5: ***Consider risk as you define the plan.*** If the team has defined risks that have high impact and high probability, contingency planning is necessary. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

Principle #6: ***Be realistic.*** People don't work 100 percent of every day. Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

> **"Success is more a function of consistent common sense than it is of genius."**
>
> **An Wang**

The term *granularity* refers to the detail with which some element of planning is represented or conducted.

Principle #7: ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed. A "fine granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A "coarse granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from fine to coarse as the project timeline moves away from the current date. Over the next few weeks or months, the project can be planned in significant detail. Activities that won't occur for many months do not require fine granularity (too much can change).

Principle #8: ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality. If formal technical reviews[7] are to be conducted, they should be scheduled. If pair programming (Chapter 4) is to be used during construction, it should be explicitly defined within the plan.

---

7 Formal technical reviews are discussed in Chapter 26.

Principle #9: ***Describe how you intend to accommodate change.*** Even the best planning can be obviated by uncontrolled change. The software team should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

Principle #10: ***Track the plan frequently and make adjustments as required.*** Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

To be most effective, everyone on the software team should participate in the planning activity. Only then will team members "sign up" to the plan.

In an excellent paper on software process and projects, Barry Boehm [BOE96] states: "You need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the *W*^{5}*HH principle*, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

What questions must be asked and answered to develop a realistic project plan?

Why is the system being developed? All parties should assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

What will be done? Identify the functionality to be built, and by implication, the tasks required to get the job done.

When will it be accomplished? Establish a workflow and timeline for key project tasks and identify the milestones required by the customer.

Who is responsible for a function? The role and responsibility of each member of the software team must be defined.

Where are they organizationally located? Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 23) based on answers to earlier questions.

The answers to Boehm's W^{5}HH questions are important regardless of the size or complexity of a software project. But how does the planning process begin?

> "We think that software developers are missing a vital truth: most organizations don't know what they do. They think they know, but they don't know."
>
> Tom DeMarco

TASK SET

Generic Task Set for Planning

1. Reevaluate project scope (Sections 7.4 and 21.3).
2. Assess risks (Section 25.4).
3. Develop and/or refine user scenarios (Sections 7.5 and 8.5).
4. Extract functions and features from the scenarios (Section 8.5).
5. Define technical functions and features that enable software infrastructure .
6. Group functions and features (scenarios) by customer priority
7. Create a coarse granularity project plan (Chapters 23 and 24).
   - Define the number of projected software increments.
   - Establish an overall project schedule (Chapter 24).
   - Establish projected delivery dates for each increment.
8. Create a fine granularity plan for the current iteration (Chapters 23 and 24).
   - Define work tasks for each function feature (Section 23.6).
   - Estimate effort for each work task (Section 23.6).
   - Assign responsibility for each work task (Section 23.4).
   - Define work products to be produced.
   - Identify quality assurance methods to be used (Chapter 26).
   - Describe methods for managing change (Chapter 27).
9. Track progress regularly (Section 24.5.2).
   - Note problem areas (e.g., schedule slippage).
   - Make adjustments as required.

5.4 MODELING PRACTICE

We create models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (e.g., a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, when the entity is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that the users desires, and the behavior of the system as the transformation is taking place. Models must accomplish these objectives at different levels of abstraction—first depicting the software from the customer's viewpoint and later representing the software at a more technical level.

Analysis models represent customer requirements. Design models provide a concrete specification for the construction of the software.

In software engineering work, two classes of models are created: analysis models and design models. *Analysis models* represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain. *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture (Chapter 10), the user interface (Chapter 12), and component-level detail (Chapter 11).

In the sections that follow we present basic principles and concepts that are relevant to analysis and design modeling. The technical methods and notation that allow software engineers to create analysis and design models are presented in later chapters.

> "The engineer's first problem in any design situation is to discover what the problem really is."
>
> **Author unknown**

5.4.1 Analysis Modeling Principles

Over the past three decades, a large number of analysis modeling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

Principle #1: ***The information domain of a problem must be represented and understood.*** The *information domain* encompasses the data that flow into the system (from end-users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means) and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).

Analysis modeling focuses on three attributes of software: information to be processed, function to be delivered, and behavior to be exhibited.

Principle #2: ***The functions that the software performs must be defined.*** Software functions provide direct benefit to end-users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Principle #3: ***The behavior of the software (as a consequence of external events) must be represented.*** The behavior of computer software is driven by its interaction with the external environment. Input provided by end-users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle #4: ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.*** Analysis modeling is the first step in software engineering problem solving. It allows the practitioner to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, we use a divide and conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called *partitioning*, and it is a key strategy in analysis modeling.

Principle #5: ***The analysis task should move from essential information toward implementation detail.*** Analysis modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively, a keyboard command might be typed or a joystick (or mouse) might be pointed in a specific direction.

TASK SET

Generic Task Set for Analysis Modeling

1. Review business requirements, end-users' characteristics/needs, user-visible outputs, business constraints, and other technical requirements that were determined during the customer communication and planning activities.
2. Expand and refine user scenarios (Section 8.5).
   - Define all actors.
   - Represent how actors interact with the software.
   - Extract functions and features from the user scenarios.
   - Review the user scenarios for completeness and accuracy (Section 26.4).
3. Model the information domain (Section 8.3).
   - Represent all major information objects.
   - Define attributes for each information object.
   - Represent the relationships between information objects.
4. Model the functional domain (Section 8.6).
   - Show how functions modify data objects.
   - Refine functions to provide elaborative detail.
   - Write a processing narrative that describes each function and subfunction.
   - Review the functional models (Section 26.4).
5. Model the behavioral domain (Section 8.8).
   - Identify external events that cause behavioral changes within the system.
   - Identify states that represent each externally observable mode of behavior.
   - Depict how an event causes the system to move from one state to another.
   - Review the behavioral models (Section 26.4).
6. Analyze and model the user interface (Chapter 12).
   - Conduct task analysis.
   - Create screen image prototypes.
7. Review all models for completeness, consistency and omissions.

5.4.2 Design Modeling Principles

The software design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the system.

> **"See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect."**
>
> **William Shakespeare**

There is no shortage of methods for deriving the various elements of a software design. Some methods are data-driven, allowing the data structure to dictate the program architecture and the resultant processing components. Others are pattern-driven, using information about the problem domain (the analysis model) to develop architectural styles and processing patterns. Still others are object-oriented, using problem domain objects as the driver for the creation of data structures and the methods that manipulate them. Yet all embrace a set of design principles that can be applied regardless of the method that is used:

Principle #1: ***Design should be traceable to the analysis model.*** The analysis model describes the information domain of the problem, user visible functions, system behavior, and a set of analysis classes that package business objects with the methods that service them. The design model translates this information into an architecture: a set of subsystems that implement major functions, and a set of component-level designs that are the realization of analysis classes. With the exception of design associated with the software infrastructure, the elements of the design model should be traceable to the analysis model.

WebRef
Insightful comments on the design process, along with a discussion of design aesthetics, can be found at **cs.wwc.edu/~aabyan/Design/.**

Principle #2: ***Always consider the architecture of the system to be built.*** Software architecture (Chapter 10) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle #3: ***Design of data is as important as design of processing functions.*** Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle #4: ***Interfaces (both internal and external) must be designed with care.*** The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle #5: ***User interface design should be tuned to the needs of the end-user.*** *However, in every case, it should stress ease of use.* The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well-designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle #6: *Component-level design should be functionally independent.* Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be *cohesive*—that is, it should focus on one and only one function or subfunction.[8]

Principle #7: *Components should be loosely coupled to one another and to the external environment.* *Coupling* is achieved in many ways—via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood or error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

Principle #8: *Design representations (models) should be easily understandable.* The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle #9: *The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.* Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability). *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts (Chapter 9).

INFO

Agile Modeling

In his book on agile modeling, Scott Ambler [AMB02] defines a set of principles[9] that are applicable when analysis and design are conducted within the context of the agile software development philosophy (Chapter 4):

Principle #1: The primary goal of the software team is to build software, not create models.

Principle #2: Travel light—don't create more models than you need.

Principle #3: Strive to produce the simplest model that will describe the problem or the software.

Principle #4: Build models in a way that makes them amenable to change.

Principle #5: Be able to state an explicit purpose for each model that is created.

8 Additional discussion of cohesion can be found in Chapter 9.

9 The principles noted in this section have been abbreviated and rephrased for the purposes of this book.

Principle #6: Adapt the models you develop to the system at hand.
Principle #7: Try to build useful models, but forget about building perfect models.
Principle #8: Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.
Principle #9: If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.
Principle #10: Get feedback as soon as you can.

Regardless of the process model that is chosen or the specific software engineering practices that are applied, every software team wants to be agile. Therefore, these principles can and should be applied regardless of the software process model that is chosen.

TASK SET

Generic Task Set for Design

1. Using the analysis model, select an architectural style (pattern) that is appropriate for the software (Chapter 10).
2. Partition the analysis model into design subsystems and allocate these subsystems within the architecture (Chapter 10).
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.
   - Using the information domain model, design appropriate data structures.
3. Design the user interface (Chapter 12).
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create behavioral model of the interface.
   - Define interface objects, control mechanisms.
   - Review the interface design and revise as required (Section 26.4).
4. Conduct component-level design (Chapter 11).
   - Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component level data structures.
   - Review the component level design (Section 26.4).
5. Develop a deployment model (Section 9.4.5).

5.5 CONSTRUCTION PRACTICE

The *construction* activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end-user. In modern software engineering work, coding may be: (1) the direct creation of programming language source code; (2) the automatic generation of source code using an intermediate design-like representation of the component to be built; (3) the automatic generation of executable code using a fourth generation programming language (e.g., Visual C++).

"For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change."

David Parnas

The initial focus of testing is at the component level, often called *unit testing*. Other levels of testing include: (1) *integration testing* (conducted as the system is constructed); (2) *validation testing* that assesses whether requirements have been met for the complete system (or software increment); and (3) *acceptance testing* that is conducted by the customer in an effort to exercise all required features and functions.

A set of fundamental principles and concepts are applicable to coding and testing. They are considered in the sections that follow.

5.5.1 Coding Principles and Concepts

The principles and concepts that guide the coding task are closely aligned programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

Avoid developing an elegant program that solves the wrong problem. Pay particular attention to the first preparation principle.

Preparation principles: *Before you write one line of code, be sure you:*

1. Understand the problem you're trying to solve.
2. Understand basic design principles and concepts.
3. Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
4. Select a programming environment that provides tools that will make your work easier.
5. Create a set of unit tests that will be applied once the component you code is completed.

WebRef

A wide variety of links to coding standards can be found at **www.literateprogramming.com/fpstyle.html.**

Coding principles: *As you begin writing code, be sure you:*

1. Constrain your algorithms by following structured programming [BOH00] practice.
2. Select data structures that will meet the needs of the design.
3. Understand the software architecture and create interfaces that are consistent with it.
4. Keep conditional logic as simple as possible.
5. Create nested loops in a way that makes them easily testable.
6. Select meaningful variable names and follow other local coding standards.
7. Write code that is self-documenting.
8. Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation principles: *After you've completed your first coding pass, be sure you:*

1. Conduct a code walkthrough when appropriate.

2. Perform unit tests and correct errors you've uncovered.
3. Refactor the code.

Books on coding and the principles that guide it include early works on programming style [KER78], practical software construction [MCC93], programming pearls [BEN99], the art of programming [KNU99], pragmatic programming issues [HUN99], and many, many others.

TASK SET

Generic Task Set for Construction

1. Build architectural infrastructure (Chapter 10).
   - Review the architectural design.
   - Code and test the components that enable architectural infrastructure.
   - Acquire reusable architectural patterns.
   - Test the infrastructure to ensure interface integrity.
2. Build a software component (Chapter 11).
   - Review the component-level design.
   - Create a set of unit tests for the component (Sections 13.3.1 and 14.7).
   - Code component data structures and interface.
   - Code internal algorithms and related processing functions.
   - Review code as it is written (Section 26.4).
   - Look for correctness.
   - Ensure that coding standards have been maintained.
   - Ensure that the code is self-documenting.
3. Unit test the component.
   - Conduct all unit tests.
   - Correct errors uncovered.
   - Reapply unit tests.
4. Integrate completed component into the architectural infrastructure.

5.5.2 Testing Principles

In a classic book on software *testing*, Glen Myers [MYE79] states a number of rules that can serve well as testing objectives:

What are the objectives of software testing?

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a dramatic change in viewpoint for some software developers. They move counter to the commonly held view that a successful test is one in which no errors are found. Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

Davis [DAV95] suggests a set of testing principles[10] that have been adapted for use in this book:

10 Only a small subset of Davis's testing principles are noted here. For more information, see [DAV95].

Principle #1: ***All tests should be traceable to customer requirements.***[11] The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle #2: ***Tests should be planned long before testing begins.*** Test planning (Chapter 13) can begin as soon as the analysis model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

In a broader software design context, recall that we begin "in the large" by focusing on software architecture and end "in the small" focusing on components. For testing, we simply reverse the focus and test our way out.

Principle #3: ***The Pareto principle applies to software testing.*** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle #4: ***Testing should begin "in the small" and progress toward testing "in the large."*** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Principle #5: ***Exhaustive testing is not possible.*** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised (Chapter 14).

TASK SET

Generic Task Set for Testing

1. Design unit tests for each software component (Section 13.3.1).
   Review each unit test to ensure proper coverage.
   Conduct the unit test.
   Correct errors uncovered.
   Reapply unit tests.
2. Develop an integration strategy (Section 13.3.2).
   Establish order of and strategy to be used for integration.
   Define "builds" and the tests required to exercise them.
   Conduct smoke testing on a daily basis.
   Conduct regression tests as required.
3. Develop validation strategy (Section 13.5).
   Establish validation criteria.
   Define tests required to validate software.
4. Conduct integration and validation tests.
   Correct errors uncovered.
   Reapply tests as required.
5. Conduct high-order tests.
   Perform recovery testing (Section 13.6.1).
   Perform security testing (Section 13.6.2).
   Perform stress testing (Section 13.6.3).
   Perform performance testing (Section 13.6.4).
6. Coordinate acceptance tests with customer (Section 13.5.3).

---

11 This principle refers to *functional* tests, i.e., tests that focus on requirements. *Structural* tests (tests that focus on architectural or logical detail) may not address specific requirements directly.

5.6 DEPLOYMENT

As we noted in Chapter 2, the deployment activity encompasses three actions: delivery, support, and feedback. Because modern software process models are evolutionary in nature, deployment happens not once, but a number of times as software moves toward completion. Each delivery cycle provides the customer and end-users with an operational software increment that provides usable functions and features. Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to date. Each feedback cycle provides the software team with important guidance that results in modifications to the functions, features, and approach taken for the next increment.

The delivery of a software increment represents an important milestone for any software project. A number of key principles should be followed as the team prepares to deliver an increment:

Be sure that your customer knows what to expect before a software increment is delivered. Otherwise, you can bet the customer will expect more than you deliver.

Principle #1: ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver and disappointment occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten [KAR94] states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle #2: ***A complete delivery package should be assembled and tested.*** A CD-ROM or other media containing all executable software, support data files, support documents, and other relevant information must be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in all possible computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements).

Principle #3: ***A support regime must be established before the software is delivered.*** An end-user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support material should be prepared, and appropriate record keeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

Principle #4: ***Appropriate instructional materials must be provided to end-users.*** The software team delivers more than the software itself. Appropriate

training aids (if required) should be developed, trouble-shooting guidelines should be provided, and a "what's-different-about-this-software-increment" description should be published.[12]

Principle #5: *Buggy software should be fixed first, delivered later.* Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day."

The delivered software provides benefit for the end-user, but it also provides useful *feedback* for the software team. As the increment is put into use, the end-users should be encouraged to comment on features and functions, ease of use, reliability, and any other characteristics that are appropriate. Feedback should be collected and recorded by the software team and used to (1) make immediate modifications to the delivered increment (if required); (2) define changes to be incorporated into the next planned increment; (3) make necessary design modifications to accommodate changes; and (4) revise the plan (including delivery schedule) for the next increment to reflect the changes.

TASK SET

Generic Task Set for Deployment

1. Create delivery media.
   - Assemble and test all executable files.
   - Assemble and test all data files.
   - Create and test all user documentation.
   - Implement electronic (e.g., pdf) versions.
   - Implement hypertext "help" files.
   - Implement a troubleshooting guide.
   - Test delivery media with a small group of representative users.
2. Establish human support person or group.
   - Create documentation and/or computer support tools.
   - Establish contact mechanisms (e.g., Web site, phone, e-mail).
   - Establish problem-logging mechanisms.
   - Establish problem-reporting mechanisms.
   - Establish problem/error reporting database.
3. Establish user feedback mechanisms.
   - Define feedback process.
   - Define feedback forms (paper and electronic).
   - Establish feedback database.
   - Define feedback assessment process.
4. Disseminate delivery media to all users.
5. Conduct on-going support functions.
   - Provide installation and start-up assistance.
   - Provide continuing troubleshooting assistance.
6. Collect user feedback.
   - Log feedback.
   - Assess feedback.
   - Communicate with users on feedback.

12 During the communication activity, the software team should determine what types of help materials users want.

5.7 SUMMARY

Software engineering practice encompasses concepts, principles, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different, yet a set of generic principles and tasks apply to each process framework activity regardless of the project or the product.

A set of technical and management essentials are necessary if good software engineering practice is to be conducted. Technical essentials include the need to understand requirements and prototype areas of uncertainty, and the need to explicitly define software architecture and plan component integration. Management essentials include the need to define priorities and define a realistic schedule that reflects them, the need to actively manage risk, and the need to define appropriate project control measures for quality and change.

Customer communication principles focus on the need to reduce noise and improve bandwidth as the conversation between developer and customer progresses. Both parties must collaborate for the best communication to occur.

Planning principles all focus on guidelines for constructing the best map for the journey to a completed system or product. The plan may be designed solely for a single software increment, or it may be defined for the entire project. Regardless, it must address what will be done, who will do it, and when the work will be completed.

Modeling encompasses both analysis and design, describing representations of the software that progressively become more detailed. The intent of the models is to solidify understanding of the work to be done and to provide technical guidance to those who will implement the software.

Construction incorporates a coding and testing cycle in which source code for a component is generated and tested to uncover errors. Integration combines individual components and involves a series of tests that focus on overall function and local interfacing issues. Coding principles define generic actions that should occur before code is written, while it is being created, and after it has been completed. Although there are many testing principles, only one is dominant: testing is a process of executing a program with the intent of finding an error.

During evolutionary software development, deployment happens for each software increment that is presented to the customer. Key principles for delivery consider managing customer expectations and providing the customer with appropriate support information for the software. Support demands advance preparations. Feedback allows the customer to suggest changes that have business value and provide the developer with input for the next iterative software engineering cycle.