

CHAPTER

4

PROCESS MODELS

KEY CONCEPTS

aspect-oriented software development.....	54
component-based development.....	53
concurrent models...	49
evolutionary process model.....	45
formal methods model.....	53
incremental process models.....	43
Personal Software Process	59
process modeling tools	62
process technology .	61
prototyping	45
spiral model	47
Team Software Process	60
unified process	55
V-model	42
waterfall model ...	41

Process models were originally proposed to bring order to the chaos of software development. History has indicated that these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the products that are produced remain on “the edge of chaos.”

In an intriguing paper on the strange relationship between order and chaos in the software world, Nogueira and his colleagues [Nog00] state

The edge of chaos is defined as “a natural state between order and chaos, a grand compromise between structure and surprise.” [Kau95] The edge of chaos can be visualized as an unstable, partially structured state . . . It is unstable because it is constantly attracted to chaos or to absolute order.

We have the tendency to think that order is the ideal state of nature. This could be a mistake. Research . . . supports the theory that operation away from equilibrium generates creativity, self-organized processes, and increasing returns [Roo96]. Absolute order means the absence of variability, which could be an

QUICK LOOK

What is it? A process model provides a specific roadmap for software engineering work. It defines the flow of all activities, actions and tasks, the degree of iteration, the work products, and the organization of the work that must be done.

Who does it? Software engineers and their managers adapt a process model to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.

Why is it important? Because process provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be “agile.” It must

demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? The process model provides you with the “steps” you’ll need to perform disciplined software engineering work.

What is the work product? From the point of view of a software engineer, the work product is a customized description of the activities and tasks defined by the process.

How do I ensure that I’ve done it right?

There are a number of software process assessment mechanisms that enable organizations to determine the “maturity” of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use.



The purpose of process models is to try to reduce the chaos present in developing new software products.

advantage under unpredictable environments. Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossible. Lack of structure does not always mean disorder.

The philosophical implications of this argument are significant for software engineering. Each process model described in this chapter tries to strike a balance between the need to impart order in a chaotic world and the need to be adaptable when things change constantly.

4.1 PRESCRIPTIVE PROCESS MODELS

WebRef

An award-winning “process simulation game” that includes most important prescriptive process models can be found at:

<http://www.ics.uci.edu/~emilyo/SimSE/downloads.html>.

A *prescriptive process model*¹ strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, we examine the prescriptive process approach in which order and project consistency are dominant issues. We call them “prescriptive” because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities described in Chapters 2 and 3, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

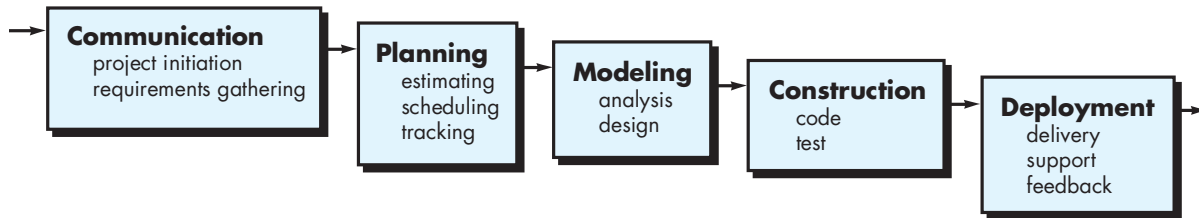
4.1.1 The Waterfall Model

There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.



Prescriptive process models define a prescribed set of process elements and a predictable process work flow.

¹ Prescriptive process models are sometimes referred to as “traditional” process models.

FIGURE 4.1 The waterfall model

KEY POINT

The V-model illustrates how verification and validation actions are associated with earlier engineering actions.

The *waterfall model*, sometimes called the *classic life cycle*, suggests a systematic, sequential approach² to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 4.1).

A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 4.2, the V-model [Buc99] depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moves down the left side.³ In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

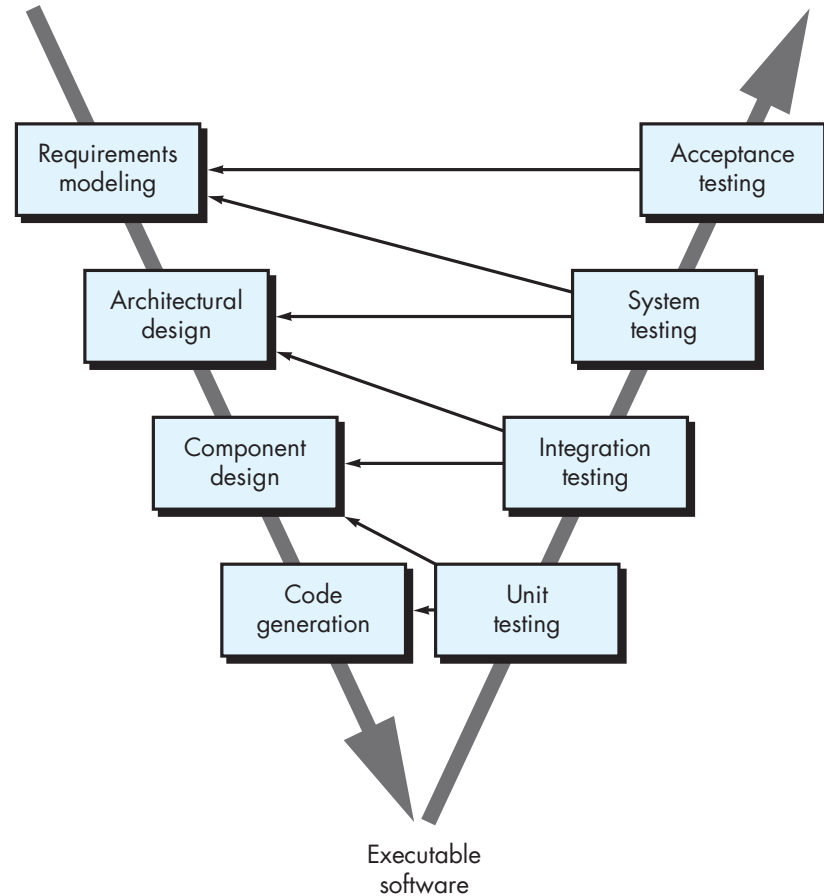
The waterfall model is the oldest paradigm for software engineering. However, over the past four decades, criticism of this process model has caused even ardent supporters to question its efficacy [Han95]. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

? Why does the waterfall model sometimes fail?

² Although the original waterfall model proposed by Winston Royce [Roy70] made provision for “feedback loops,” the vast majority of organizations that apply this process model treat it as if it were strictly linear.

³ A detailed discussion of quality assurance actions is presented in Part 3 of this book.

FIGURE 4.2**The V-model**

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

note:

"Too often, software work follows the first law of bicycling: No matter where you're going, it's uphill and against the wind."

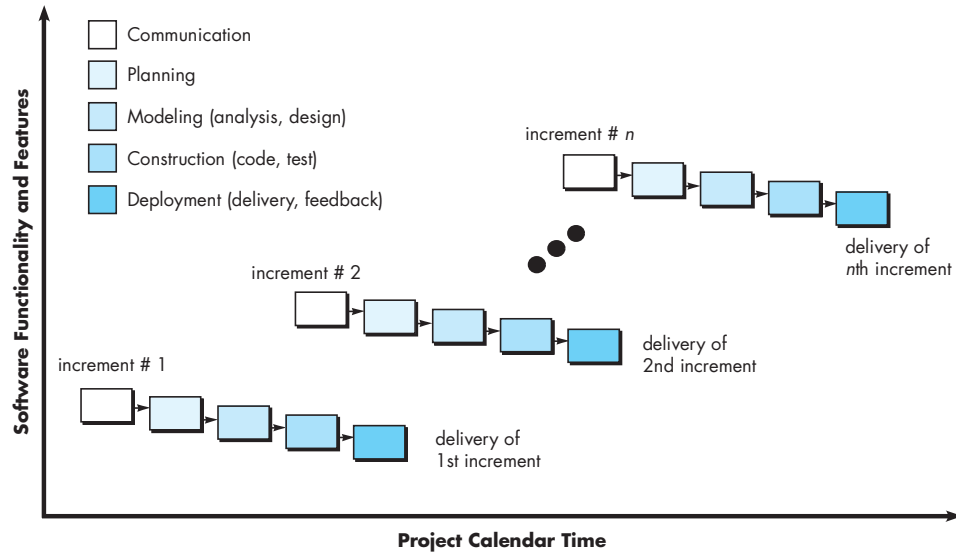
Author unknown

In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

4.1.2 Incremental Process Models

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely

FIGURE 4.3**The incremental model**

linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The incremental model combines the elements' linear and parallel process flows discussed in Chapter 3. Referring to Figure 4.3, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software [McD93].

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm discussed in the next subsection.

When an incremental model is used, the first increment is often a *core product*. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

KEY POINT

The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.

ADVICE

Your customer demands delivery by a date that is impossible to meet. Suggest delivering one or more increments by that date and the rest of the software (additional increments) later.



Evolutionary process models produce an increasingly more complete version of the software with each iteration.

4.1.3 Evolutionary Process Models

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that grows and changes.

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, we present two common evolutionary process models.



"Plan to throw one away. You will do that, anyway. Your only choice is whether to try to sell the throwaway to customers."

Frederick P. Brooks

Prototyping. Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.



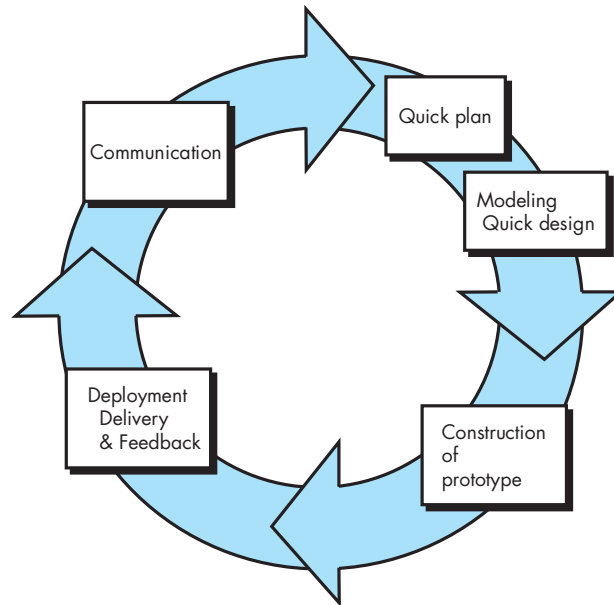
When your customer has a legitimate need, but is clueless about the details, develop a prototype as a first step.

The prototyping paradigm (Figure 4.4) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a “quick design”) occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.

FIGURE 4.4

The prototyping paradigm



But what do you do with the prototype when it has served the purpose described earlier? Brooks [Bro95] provides one answer:

In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as “the first system.” The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as “throwaways,” others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:



Resist pressure to extend a rough prototype into a production product. Quality almost always suffers as a result.

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that “a few fixes” be applied to make the prototype a working product. Too often, software development management relents.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is

SAFEHOME



Selecting a Process Model, Part 1

The scene: Meeting room for the software engineering group at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member; and Ed Robbins, software team member.

The conversation:

Lee: So let's recapitulate. I've spent some time discussing the *SafeHome* product line as we see it at the moment. No doubt, we've got a lot of work to do to simply define the thing, but I'd like you guys to begin thinking about how you're going to approach the software part of this project.

Doug: Seems like we've been pretty disorganized in our approach to software in the past.

Ed: I don't know, Doug, we always got product out the door.

Doug: True, but not without a lot of grief, and this project looks like it's bigger and more complex than anything we've done in the past.

Jamie: Doesn't look that hard, but I agree . . . our ad hoc approach to past projects won't work here, particularly if we have a very tight time line.

Doug (smiling): I want to be a bit more professional in our approach. I went to a short course last week and learned a lot about software engineering . . . good stuff. We need a process here.

Jamie (with a frown): My job is to build computer programs, not push paper around.

Doug: Give it a chance before you go negative on me. Here's what I mean. (Doug proceeds to describe the process framework described in Chapter 3 and the prescriptive process models presented to this point.)

Doug: So anyway, it seems to me that a linear model is not for us . . . assumes we have all requirements up front and, knowing this place, that's not likely.

Vinod: Yeah, and it sounds way too IT-oriented . . . probably good for building an inventory control system or something, but it's just not right for *SafeHome*.

Doug: I agree.

Ed: That prototyping approach seems okay. A lot like what we do here anyway.

Vinod: That's a problem. I'm worried that it doesn't provide us with enough structure.

Doug: Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

The Spiral Model. Originally proposed by Barry Boehm [Boe88], the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more

complete versions of the software. Boehm [Boe01a] describes the model in the following manner:

KEY POINT

The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

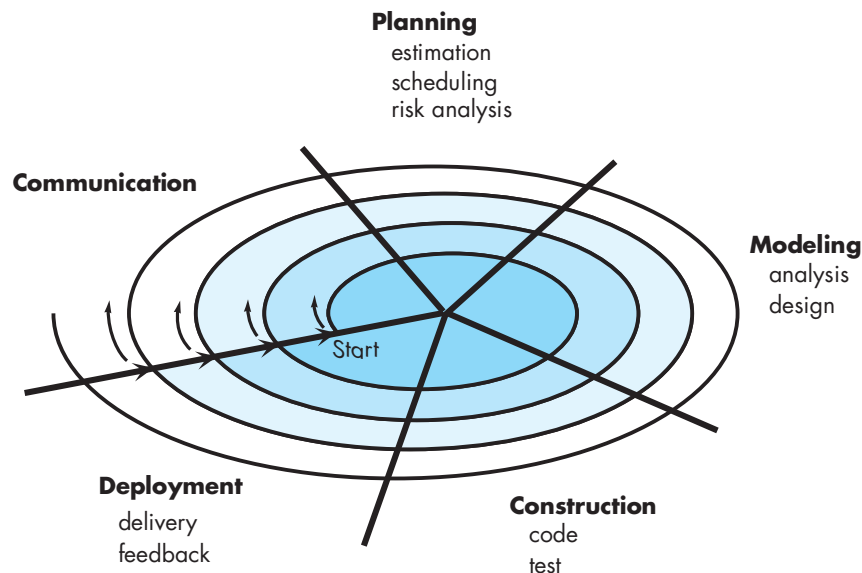
The spiral development model is a *risk-driven process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, we use the generic framework activities discussed earlier.⁴ Each of the framework activities represent one segment of the spiral path illustrated in Figure 4.5. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 35) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

FIGURE 4.5

A typical spiral model



⁴ The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [Boe88]. More recent discussion of Boehm's spiral model can be found in [Boe98].

WebRef

Useful information about the spiral model can be obtained at: www.sei.cmu.edu/publications/documents/00.reports/00sr008.html.



If your management demands fixed-budget development (generally a bad idea), the spiral can be a problem. As each circuit is completed, project cost is revisited and revised.

note:

"I'm only this far and only tomorrow leads my way."

**Dave
Matthews Band**

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations⁵ until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

4.1.4 Concurrent Models

The *concurrent development model*, sometimes called *concurrent engineering*, allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modeling activity

⁵ The arrows pointing inward along the axis separating the *deployment* region from the *communication* region indicate a potential for local iteration along the same spiral path.

SAFEHOME



Selecting a Process Model, Part 2

The scene: Meeting room for the software engineering group at CPI

Corporation, a company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Vinod and Jamie, members of the software engineering team.

The conversation: (Doug describes evolutionary process options.)

Jamie: Now I see something I like. An incremental approach makes sense, and I really like the flow of that spiral model thing. That's keepin' it real.

Vinod: I agree. We deliver an increment, learn from customer feedback, re-plan, and then deliver another increment. It also fits into the nature of the product. We

can have something on the market fast and then add functionality with each version, er, increment.

Lee: Wait a minute. Did you say that we regenerate the plan with each tour around the spiral, Doug? That's not so great; we need one plan, one schedule, and we've got to stick to it.

Doug: That's old-school thinking, Lee. Like the guys said, we've got to keep it real. I submit that it's better to tweak the plan as we learn more and as changes are requested. It's way more realistic. What's the point of a plan if it doesn't reflect reality?

Lee (frowning): I suppose so, but . . . senior management's not going to like this . . . they want a fixed plan.

Doug (smiling): Then you'll have to reeducate them, buddy.

KEY POINT

Project plans must be viewed as living documents; progress must be assessed often and revised to take changes into account.

ADVICE

The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

note:

"Every process in your organization has a customer, and without a customer a process has no purpose."

V. Daniel Hunt

defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.⁶

Figure 4.6 provides an example of the concurrent modeling approach. An activity—**modeling**—may be in any one of the states⁷ noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **none** state while initial communication was completed) now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

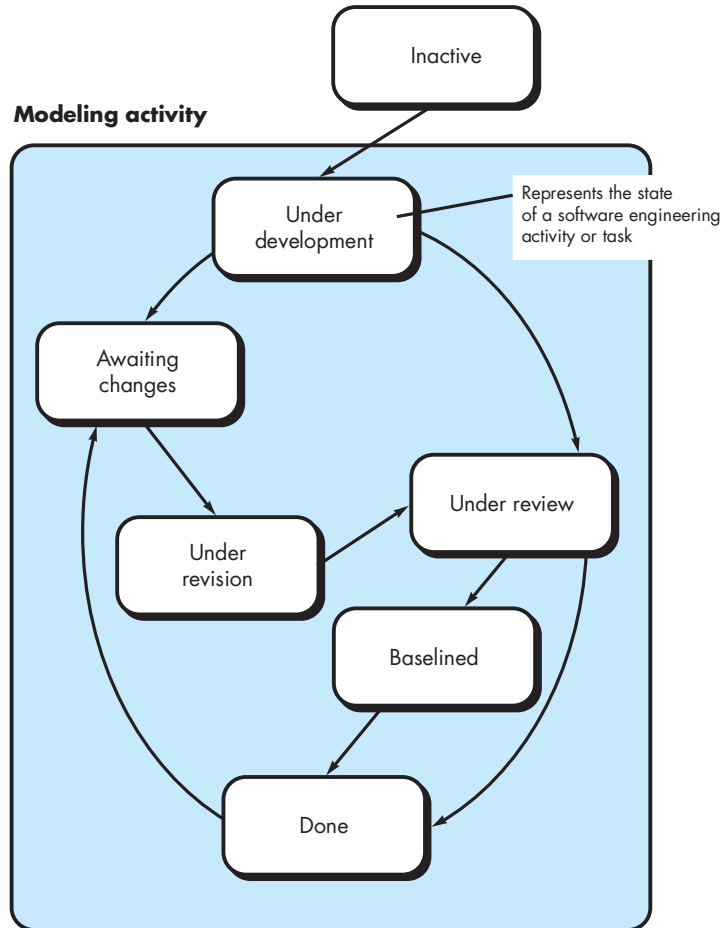
Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements

⁶ It should be noted that analysis and design are complex tasks that require substantial discussion. Part 2 of this book considers these topics in detail.

⁷ A *state* is some externally observable mode of behavior.

FIGURE 4.6

One element of
the concurrent
process model



model is uncovered. This generates the event *analysis model correction*, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states associated with each activity.

4.1.5 A Final Word on Evolutionary Processes

We have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction. In many cases, time-to-market is the most important management

requirement. If a market window is missed, the software project itself may be meaningless.⁸

Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses. These are summarized by Nogueira and his colleagues [Nog00]:

Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product . . .

Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected . . .

Third, [evolutionary] software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary.

? What are the potential weaknesses of evolutionary process models?

Indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality does sound scary. And yet, this idea has been proposed by a number of well-respected software engineering experts (e.g., [You95], [Bac97]).

The intent of evolutionary models is to develop high-quality software⁹ in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

4.2 SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.¹⁰

⁸ It is important to note, however, that being the first to reach a market is no guarantee of success. In fact, many very successful software products have been second or even third to reach the market (learning from the mistakes of their predecessors).

⁹ In this context software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Part 2 of this book.

¹⁰ In some cases, these specialized process models might better be characterized as a collection of techniques or a “methodology” for accomplishing a specific software development goal. However, they do imply a process.

WebRef

Useful information on component-based development can be obtained at:
www.cbd-hq.com.

4.2.1 Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the component-based development model comprises applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages¹¹ of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits including a reduction in development cycle time and a reduction in project cost if component reuse becomes part of your organization's culture. Component-based development is discussed in more detail in Chapter 14.

4.2.2 The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* [Mil87, Dye92], is currently applied by some software development organizations.

¹¹ Object-oriented concepts are discussed in Appendix 2 and are used throughout Part 2 of this book. In this context, a class encompasses a set of data and the procedures that process the data. A package of classes is a collection of related classes that work together to achieve some end result.

When formal methods (Appendix 3) are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

? If formal methods can demonstrate software correctness, why is it they are not widely used?

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

4.2.3 Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object-oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns*. *Aspectual requirements* define those crosscutting concerns that have an impact across the software architecture. *Aspect-oriented software development* (AOSD), often referred to as *aspect-oriented programming* (AOP) or *aspect-oriented component engineering* (AOCE) [Gru02], is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and

WebRef

A wide array of resources and information on AOP can be found at: aosd.net.



AOSD defines “aspects” that express customer concerns that cut across multiple system functions, features, and information.

constructing *aspects*—“mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern” [Elr01].

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

A detailed discussion of aspect-oriented software development is best left to books dedicated to the subject. If you have further interest, see [Ras11], [Saf08], [Cla05], [Fil05], [Jac04], and [Gra03].

SOFTWARE TOOLS



Process Management

Objective: To assist in the definition, execution, and management of prescriptive process models.

Mechanics: Process management tools allow a software organization or team to define a complete software process model (framework activities, actions, tasks, QA checkpoints, milestones, and work products). In addition, the tools provide a road map as software engineers do technical work and a template for managers who must track and control the software process.

Representative tools:¹²

GDPA, a research process definition tool suite, developed at Bremen University in Germany

(www.informatik.uni-bremen.de/uniform/gdpa/home.htm), provides a wide array of process modeling and management functions.

ALM Studio, developed by Kovair Corporation (<http://www.kovair.com/>) encompasses a suite of tools for process definition, requirements management, issue resolution, project planning, and tracking.

ProVision BPMx, developed by OpenText (<http://bps.opentext.com/>), is representative of many tools that assist in process definition and workflow automation.

A worthwhile listing of many different tools associated with the software process can be found at www.computer.org/portal/web/swebok/html/ch10.

4.3 THE UNIFIED PROCESS

In their seminal book on the *Unified Process (UP)*, Ivar Jacobson, Grady Booch, and James Rumbaugh [Jac99] discuss the need for a “use case driven, architecture-centric, iterative and incremental” software process when they state:

Today, the trend in software is toward bigger, more complex systems. That is due in part to the fact that computers become more powerful every year, leading users to expect more from them. This trend has also been influenced by the expanding use of

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.

the Internet for exchanging all kinds of information . . . Our appetite for ever-more sophisticated software grows as we learn from one product release to the next how the product could be improved. We want software that is better adapted to our needs, but that, in turn, merely makes the software more complex. In short, we want more.

In some ways the Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development (Chapter 5). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case).¹³ It emphasizes the important role of software architecture and “helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse” [Jac99]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

4.3.1 A Brief History

During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a “unified method” that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts (e.g., [Wir90]) in object-oriented modeling. The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems. By 1997, UML became a de facto industry standard for object-oriented software development.

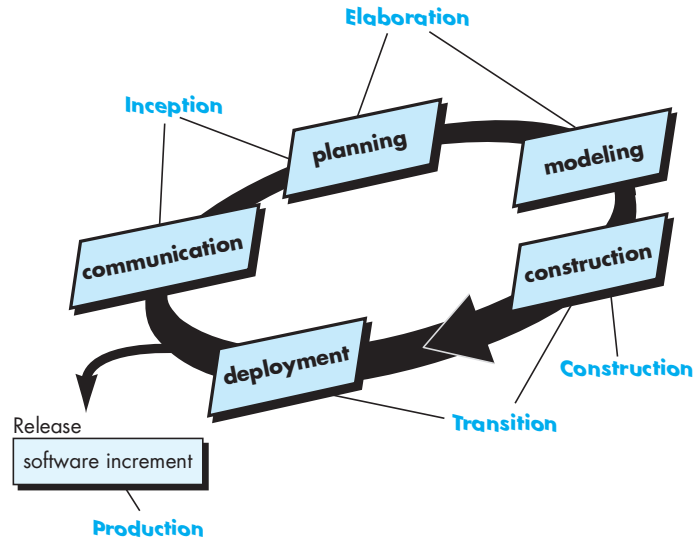
UML is used throughout Part 2 of this book to represent both requirements and design models. Appendix 1 presents an introductory tutorial for those who are unfamiliar with basic UML notation and modeling rules. A comprehensive presentation of UML is best left to textbooks dedicated to the subject. Recommended books are listed in Appendix 1.

4.3.2 Phases of the Unified Process¹⁴

In Chapter 3, we discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process

¹³ A *use case* (Chapter 8) is a text narrative or template that describes a system function or feature from the user's point of view. A use case is written by the user and serves as a basis for the creation of a more comprehensive analysis model.

¹⁴ The Unified Process is sometimes called the *Rational Unified Process* (RUP) after the Rational Corporation (subsequently acquired by IBM), an early contributor to the development and refinement of the UP and a builder of complete environments (tools and technology) that support the process.

FIGURE 4.7**The Unified Process**

is no exception. Figure 4.7 depicts the “phases” of the UP and relates them to the generic activities that have been discussed in Chapter 1 and earlier in this chapter.

KEY POINT

UP phases are similar in intent to the generic framework activities defined in this book.

The *inception phase* of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases (Chapter 8) that describe which features and functions each major class of users desires. Architecture at this point is nothing more than a tentative outline of major subsystems and the functions and features that populate them. Later, the architecture will be refined and expanded into a set of models that will represent different views of the system. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

The *elaboration phase* encompasses the communication and modeling activities of the generic process model (Figure 4.7). Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an “executable architectural baseline” [Arl02] that represents a “first cut” executable system.¹⁵ The architectural baseline demonstrates the viability of the

¹⁵ It is important to note that the architectural baseline is not a prototype in that it is not thrown away. Rather, the baseline is fleshed out during the next UP phase.

architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

WebRef

An interesting discussion of the UP in the context of agile development can be found at www.ambyssoft.com/unifiedprocess/agileUP.html.

The *construction phase* of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests¹⁶ are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

The *transition phase* of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing, and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release.

The *production phase* of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set (described in Chapter 3). That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

16 A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 22 through 26).

4.4 PERSONAL AND TEAM PROCESS MODELS



quote:

“A person who is successful has simply formed the habit of doing things that unsuccessful people will not do.”

Dexter Yager

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization. Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization. Watts Humphrey ([Hum05] and [Hum00]) argues that it is possible to create a “personal software process” and/or a “team software process.” Both require hard work, training, and coordination, but both are achievable.¹⁷

4.4.1 Personal Software Process




WebRef

A wide array of resources for PSP can be found at <http://www.sei.cmu.edu/tsp/tools/academic/>.

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a “process” does exist. Watts Humphrey [Hum05] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The *Personal Software Process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.



What framework activities are used during PSP?

¹⁷ It's worth noting the proponents of agile software development (Chapter 5) also argue that the process should remain close to the team. They propose an alternative method for achieving this.

High-level design review. Formal verification methods (Appendix 3) are applied to uncover errors in the design. Metrics are maintained for important tasks and work results.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for important tasks and work results.

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.



PSP emphasizes the need to record and analyze the types of errors you make, so that you can develop strategies to eliminate them.

PSP stresses the need for you to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.

PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [Hum96], the resulting improvement in software engineering productivity and software quality are significant [Fer97]. However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach. PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high. The required level of measurement is culturally difficult for many software people.

Can PSP be used as an effective software process at a personal level? The answer is an unequivocal “yes.” But even if PSP is not adopted in its entirety, many of the personal process improvement concepts that it introduces are well worth learning.

4.4.2 Team Software Process

WebRef

Information on building high-performance teams using TSP and PSP can be obtained at www.sei.cmu.edu/tsp/.

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a *Team Software Process* (TSP). The goal of TSP is to build a “self-directed” project team that organizes itself to produce high-quality software. Humphrey [Hum98] defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.



To form a self-directed team, you must collaborate well internally and communicate well externally.

- Accelerate software process improvement by making CMM¹⁸ level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: *project launch*, *high-level design*, *implementation*, *integration and test*, and *postmortem*. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. "Scripts" define specific process activities (i.e., project launch, design, implementation, integration and system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.

TSP recognizes that the best software teams are self-directed.¹⁹ Team members set project objectives, adapt the process to meet their needs, control the project schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering.

Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality. The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.



TSP scripts define elements of the team process and activities that occur within the process.

4.5 PROCESS TECHNOLOGY

One or more of the process models discussed in the preceding sections must be adapted for use by a software team. To accomplish this, *process technology tools* have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality.

¹⁸ The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 37.

¹⁹ In Chapter 5 we discuss the importance of "self-organizing" teams as a key element in agile software development.

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities discussed in Chapter 3. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model. Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

SOFTWARE TOOLS



Process Modeling Tools

Objective: If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called *process technology* or *process management* tools) are used to represent the key elements of a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the actions and work tasks that are required to perform it. Process modeling tools provide links to other tools that provide support to defined process activities.

Mechanics: Tools in this category allow a team to define the elements of a unique process model (actions, tasks, work products, QA points), provide

detailed guidance on the content or description of each process element, and then manage the process as it is conducted. In some cases, the process technology tools incorporate standard project management tasks such as estimating, scheduling, tracking, and control.

Representative tools:²⁰

Igrafx Process Tools—tools that enable a team to map, measure, and model the software process (<http://www.igrafx.com/>)

Adeptia BPM Server—designed to manage, automate, and optimize business processes (www.adeptia.com)

ALM Studio Suite—a collection of tools with a heavy emphasis on the management of communication and modeling activities (<http://www.kovair.com/>)

4.6 PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer. But an obsessive overreliance on process is also dangerous. In a brief essay written many years ago, Margaret Davis [Dav95a] makes timeless comments on the duality of product and process:

About every ten years give or take five, the software community redefines “the problem” by shifting its focus from product issues to process issues. Thus, we have

²⁰ 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.

embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process) followed by object-oriented methods, followed by agile software development.

While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community's focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means to perform the job let alone perform it well. The swings also do not solve "the problem" for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.

There is precedence in the scientific community to advance notions of duality when contradictions in observations cannot be fully explained by one competing theory or another. The dual nature of light, which seems to be simultaneously particle and wave, has been accepted since the 1920s when Louis de Broglie proposed it. I believe that the observations we can make on the artifacts of software and its development demonstrate a fundamental duality between product and process. You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product.

All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result in a representation or instance that can be used or appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.

Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity. Taking one view over the other dramatically reduces the opportunities for reuse and, hence, loses the opportunity for increasing job satisfaction.

People derive as much (or more) satisfaction from the creative process as they do from the end product. An artist enjoys the brush strokes as much as the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. As creative software professional, you should also derive as much satisfaction from the process as the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

4.7 SUMMARY

Prescriptive process models have been applied for many years in an effort to bring order and structure to software development. Each of these models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment.

Sequential process models, such as the waterfall and V-models, are the oldest software engineering paradigms. They suggest a linear process flow that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight time lines) in the software world. They do, however, have applicability in situations where requirements are well defined and stable.

Incremental process models are iterative in nature and produce working versions of software quite rapidly. Evolutionary process models recognize the iterative, incremental nature of most software engineering projects and are designed to accommodate change. Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly. These models can be adopted to apply across all software engineering activities—from concept development to long-term system maintenance.

The concurrent process model allows a software team to represent iterative and concurrent elements of any process model. Specialized models include the component-based model that emphasizes component reuse and assembly; the formal methods model that encourages a mathematically based approach to software development and verification; and the aspect-oriented model that accommodates crosscutting concerns spanning the entire system architecture. The Unified Process is a “use case driven, architecture-centric, iterative and incremental” software process designed as a framework for UML methods and tools.

Personal and team models for the software process have been proposed. Both emphasize measurement, planning, and self-direction as key ingredients for a successful software process.

PROBLEMS AND POINTS TO PONDER

- 4.1. Provide three examples of software projects that would be amenable to the waterfall model. Be specific.
- 4.2. Provide three examples of software projects that would be amenable to the prototyping model. Be specific.
- 4.3. What process adaptations are required if the prototype will evolve into a delivery system or product?
- 4.4. Provide three examples of software projects that would be amenable to the incremental model. Be specific.
- 4.5. As you move outward along the spiral process flow, what can you say about the software that is being developed or maintained?

- 4.6. Is it possible to combine process models? If so, provide an example.
- 4.7. 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.
- 4.8. 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?
- 4.9. Provide three examples of software projects that would be amenable to the component-based model. Be specific.
- 4.10. It is possible to prove that a software component and even an entire program is correct. So why doesn’t everyone do this?
- 4.11. Are the Unified Process and UML the same thing? Explain your answer.

FURTHER READINGS AND INFORMATION SOURCES

Most of the software engineering books discussed in the *Further Readings* section of Chapter 2 address prescriptive process models in some detail.

Cynkovic and Larsson (*Building Reliable Component-Based Systems*, Addison-Wesley, 2002) and Heineman and Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) describe the process required to implement component-based systems. Jacobson and Ng (*Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2005) and Filman and his colleagues (*Aspect-Oriented Software Development*, Addison-Wesley, 2004) discuss the unique nature of the aspect-oriented process. Monin and Hinchey (*Understanding Formal Methods*, Springer, 2003) present a worthwhile introduction, and Boca and his colleagues (*Formal Methods*, Springer, 2009) discuss the state of the art and new directions.

Books by Kenett and Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) and Chrissis, Konrad, and Shrum (*CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed., Addison-Wesley, 2011) consider how quality management and process design are intimately connected to one another.

In addition to Jacobson, Rumbaugh, and Booch’s seminal book on the Unified Process [Jac99], books by Shuja and Krebs (*IBM Rational Unified Process Reference and Certification Guide*, IBM Press, 2008), Arlow and Neustadt (*UML 2 and the Unified Process*, Addison-Wesley, 2005), Kroll and Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003), and Farve (*UML and the Unified Process*, IRM Press, 2003) provide excellent complementary information. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) discusses project management within the context of the UP. Dennis, Wixom, and Tegarden (*Systems Analysis and Design with UML*, 4th ed., Wiley, 2012) tackles programming and business process modeling as it relates to UP.

A wide variety of information sources on software process models 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 website: www.mhhe.com/pressman.

CHAPTER

5

AGILE DEVELOPMENT

KEY CONCEPTS

acceptance tests ... 75
agile alliance..... 70
agile process..... 69
Agile Unified
Process 82
agility 68
agility principles ... 70
cost of change..... 68
Dynamic Systems
Development Method
(DSDM) 79

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01] (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.

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 systems and software products is fast-paced and ever-changing. Agile software engineering represents a reasonable alternative to

conventional software engineering for certain classes of software and certain types of software projects. It has been demonstrated to deliver successful systems quickly.

What are the steps? Agile development might best be termed “software engineering lite.” The basic framework activities—communication, planning, modeling, construction, and deployment—remain. But they morph into a minimal task set that pushes the project team toward construction and delivery (some would argue that this is done at the expense of problem analysis and solution design).

What is the work product? Both the customer and the software engineer have the same view—the only really important work product is an operational “software increment” that is delivered to the customer on the appropriate commitment date.

How do I ensure that I’ve done it right? If the agile team agrees that the process works, and the team produces deliverable software increments that satisfy the customer, you’ve done it right.

Extreme Programming (XP).....	72
Industrial XP.....	72
pair programming	75
politics of agile development.....	71
project velocity.....	73
refactoring	74
Scrum.....	78
spike solution.....	74
XP story.....	72

KEY POINT

Agile development does not mean no documents are created, it means only creating documents that will be referred to later in the development process.

Quote:

"Agility: 1, everything else: 0."

Tom DeMarco

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 been less than two decades since these ideas have crystallized into a "movement." In essence, agile¹ 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, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an over-riding 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 mobile application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, you won't be able to define requirements fully before the project begins. You must be agile enough to respond to a fluid business environment.

Fluidity implies change, and change is expensive—particularly if it is uncontrolled or poorly managed. One of the most compelling characteristics of the agile approach is its ability to reduce the costs of change through the software process.

Does this mean that a recognition of challenges posed by modern realities causes you 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.

In a thought-provoking book on agile software development, Alistair Cockburn [Coc02] argues that the prescriptive process models introduced in Chapter 4 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; 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 leitherl discipline or tolerance" and that most prescriptive process models choose discipline. He states: "Because consistency in action is a human weakness, high discipline methodologies are fragile."

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.

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

5.1 WHAT IS AGILITY?

Just what is agility in the context of software engineering work? Ivar Jacobson [Jac02a] 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 deemphasizes 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.

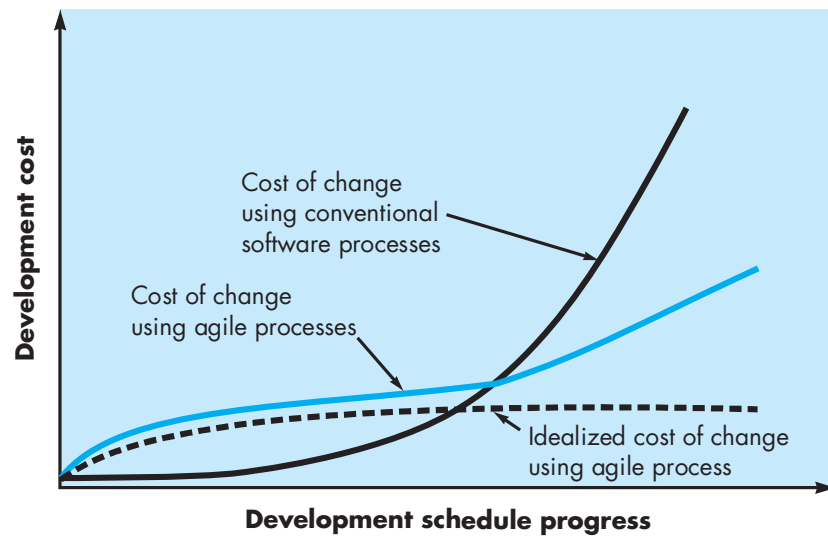
Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all 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.

5.2 AGILITY AND THE COST OF CHANGE

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses (Figure 5.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written

FIGURE 5.1

Change costs
as a function
of time in
development

**note:**

"Agility is dynamic, content specific, aggressively change embracing, and growth oriented."

Steven Goldman
et al.

KEY POINT

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 5.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. You've already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming (discussed later in this chapter), the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.

5.3 WHAT IS AN AGILE PROCESS?

Any agile software process is characterized in a manner that addresses a number of 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 the project proceeds.

WebRef

A comprehensive collection of articles on the agile process can be found at <http://www.agilemodeling.com/>.

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.

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 portions 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.

5.3.1 Agility Principles

The Agile Alliance (see [Agi03], [Fow01]) defines 12 agility 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.



Although agile processes embrace change, it is still important to examine the reasons for change.



Working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability.

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.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models presented in this chapter.

5.3.2 The Politics of Agile Development



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

There has been 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 you 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?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 5.4), 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 traditional 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.

If you have further interest, see [Hig01], [Hig02a], and [DeM02] for an entertaining summary of other important technical and political issues.

5.4 EXTREME PROGRAMMING

WebRef

An award-winning “process simulation game” that includes an XP process module can be found at <http://www.ics.uci.edu/~emilyo/SimSE/downloads.html>.

? What is an XP “story”?

In order to illustrate an agile process in a bit more detail, we’ll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck [Bec04a]. A variant of XP, called *Industrial XP* (IXP), refines XP and targets the agile process specifically for use within large organizations [Ker05].

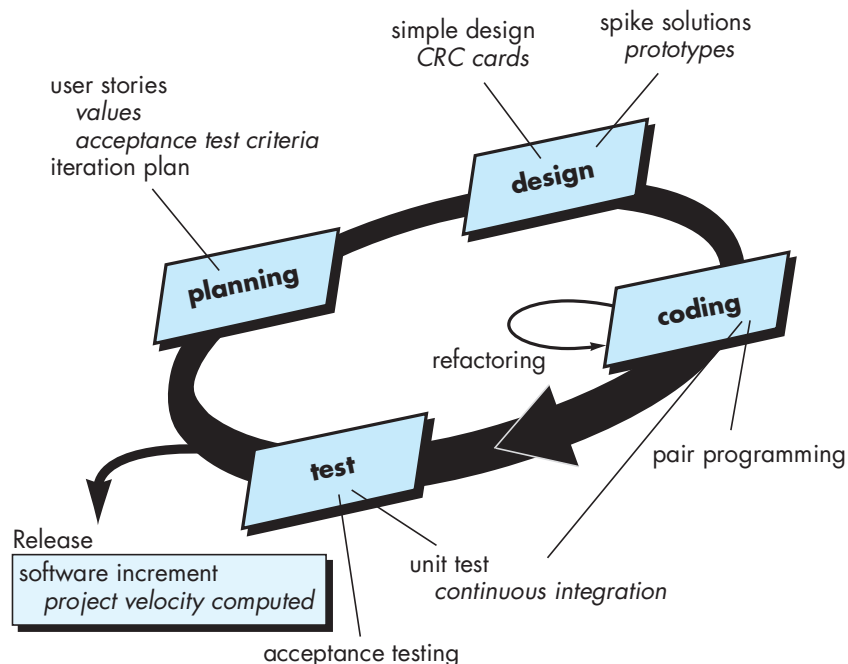
5.4.1 The XP Process

Extreme Programming uses an object-oriented approach (Appendix 2) as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 5.2 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 (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members

FIGURE 5.2

The Extreme Programming process



WebRef

A worthwhile XP “planning game” can be found at:
<http://csis.pace.edu/~bergin/xp/planninggame.html>.

of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of “stories” (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *story* (similar to use cases described in Chapter 8) is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.² Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story is estimated to 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 developers 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.

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 overcommitment has been made for all stories across the entire development project. If an overcommitment 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.³

XP encourages the use of CRC cards (Chapter 10) as an effective mechanism for thinking about the software in an object-oriented context. CRC



Project velocity is a subtle measure of team productivity.



XP deemphasizes the importance of design. Not everyone agrees. In fact, there are times when design should be emphasized.

² The value of a story may also be dependent on the presence of another story.

³ 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.

WebRef

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



Refactoring improves the internal structure of a design (or source code) without changing its external functionality or behavior.

(class-responsibility-collaborator) cards identify and organize the object-oriented classes⁴ that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one described in Chapter 10. 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.

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 the effort required for refactoring can grow dramatically as the size of an application grows.

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.

WebRef

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

Coding. After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).⁵ Once the unit test⁶ has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added

4 Object-oriented classes are discussed in Appendix 2, in Chapter 10, and throughout Part 2 of this book.

5 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.

6 Unit testing, discussed in detail in Chapter 22, 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.

(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 (the code is reviewed as it is created). 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 that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.⁷



Many software teams are populated by individualists. You'll have to work to change that culture if pair programming is to work effectively.

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 22) that helps to uncover errors early.

 **How are unit tests used in XP?**

Testing. 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 22) 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 go 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.

5.4.2 Industrial XP

 **What new practices are appended to XP to create IXP?**

Joshua Kerievsky [Ker05] describes *Industrial Extreme Programming* (IXP) in the following manner: “IXP is an organic evolution of XP. It is imbued with XP’s minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices.” IXP incorporates six new practices that

⁷ Pair programming has become so widespread throughout the software community that *The Wall Street Journal* [Wal12] ran a front-page story about the subject.

are designed to help ensure that an XP project works successfully for significant projects within a large organization:

Readiness assessment. The IXP team ascertains whether all members of the project community (e.g., stakeholders, developers, management) are on board, have the proper environment established, and understand the skill levels involved.

Project community. The IXP team determines whether the right people, with the right skills and training have been staged for the project. The “community” encompasses technologists and other stakeholders.

Project chartering. The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization.

Test-driven management. An IXP team establishes a series of measurable “destinations” [Ker05] that assess progress to date and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives. An IXP team conducts a specialized technical review (Chapter 20) after a software increment is delivered. Called a *retrospective*, the review examines “issues, events, and lessons-learned” [Ker05] across a software increment and/or the entire software release.

Continuous learning. The IXP team is encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher-quality product.

In addition to the six new practices discussed, IXP modifies a number of existing XP practices and redefines certain roles and responsibilities to make them more amenable to significant projects for large organizations. For further discussion of IXP, visit <http://industrialxp.org>.

Quote:

“Ability is what you're capable of doing. Motivation determines what you do. Attitude determines how well you do it.”

Lou Holtz

SAFEHOME



Considering Agile Software Development

The scene: Doug Miller's office.

Doug: Sure Jamie, what's up?

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

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

The conversation:

(A knock on the door, Jamie and Vinod enter Doug's office.)

Doug: And?

Jamie: Doug, you got a minute?

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 pair programming to do real-time quality checks . . . it's pretty cool, I think.

Doug: It does have a lot of really good ideas. I like the pair-programming concept, for instance, and the idea that stakeholders should be part of the team.

Jamie: Huh? You mean that marketing will work on the project team with us?

Doug (nodding): They're a stakeholder, aren't they?

Jamie: Jeez . . . they'll be requesting changes every five minutes.

Vinod: Not necessarily. My friend said that there are ways to "embrace" changes during an XP project.

Doug: So you guys think we should use XP?

Jamie: It's definitely worth considering.

Doug: I agree. And even if we choose an incremental model as our approach, there's no reason why we can't incorporate much of what XP has to offer.

Vinod: Doug, before you said "some good, some bad." What was the bad?

Doug: The thing I don't like is the way XP downplays analysis and design . . . sort of says that writing code is where the action is . . .

(The team members look at one another and smile.)

Doug: So you agree with the XP approach?

Jamie (speaking for both): Writing code is what we do, Boss!

Doug (laughing): True, but I'd like to see you spend a little less time coding and then recoding and a little more time analyzing what has to be done and designing a solution that works.

Vinod: Maybe we can have it both ways, agility with a little discipline.

Doug: I think we can, Vinod. In fact, I'm sure of it.

5.5 OTHER AGILE PROCESS MODELS

note:

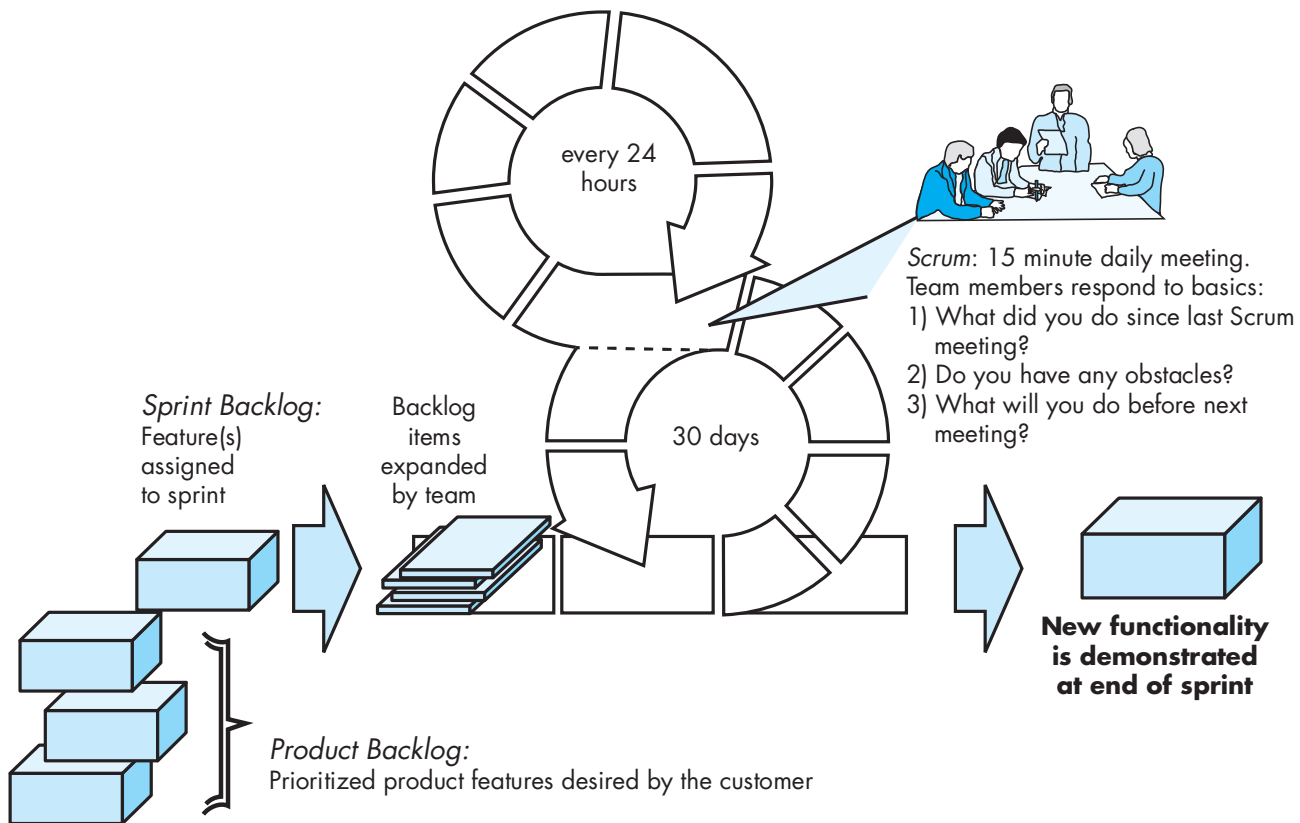
"Our profession goes through methodologies like a 14-year-old goes through clothing."

Stephen
Hawrysh and
Jim Ruprecht

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.⁸

As we noted in the last section, the most widely used of all agile process models is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. In this section, we present a brief overview of four common agile methods: Scrum, DSSD, Agile Modeling (AM), and Agile Unified Process (AUP).

⁸ 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.

FIGURE 5.3 Scrum process flow

5.5.1 Scrum

WebRef

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

Scrum (the name is derived from an activity that occurs during a rugby match)⁹ is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle [Sch01b].

Scrum principles are consistent with the agile manifesto and 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 5.3.

⁹ A group of players forms around the ball and the teammates work together (sometimes violently!) to move the ball downfield.

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:

KEY POINT

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-box¹⁰ (typically 30 days). Changes (e.g., backlog work items) 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-minute) 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 team to work successfully in a world where the elimination of uncertainty is impossible.

5.5.2 Dynamic Systems Development Method

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

WebRef

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

¹⁰ A *time-box* is a project management term (see Part 4 of this book) that indicates a period of time that has been allocated to accomplish some task.

prototyping in a controlled project environment” [CCS02]. The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

DSDM is an iterative software process in which 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*, that begins with a *feasibility study* that establishes basic business requirements and constraints and is followed by a *business study* that identifies functional and information requirements. DSDM then defines three different iterative cycles:

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 functional model iteration activity.

DSDM can be combined with XP (Section 5.4) 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.

5.5.3 Agile Modeling

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 as the system is being engineered and built. But in some cases, it can be daunting to manage the volume of notation

KEY POINT

DSDM is a process framework that can adopt the tactics of another agile approach such as XP.

WebRef

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

required, the degree of formalism suggested, the sheer size of the models for large projects, and the difficulty in maintaining the model(s) as changes occur. Is there an agile approach to software engineering modeling that might provide some relief?

At “The Official Agile Modeling Site,” Scott Ambler [Amb02a] 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 (AM) 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.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and 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 [Amb02a]:

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.

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 [Amb02a] 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.

note:

“I was in the drugstore the other day trying to get a cold medication . . . Not easy. There’s an entire wall of products you need. You stand there going, Well, this one is quick acting but this is long lasting . . . Which is more important, the present or the future?”

Jerry Seinfeld



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

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.

A major segment of the software engineering community has adopted the Unified Modeling Language (UML)¹¹ as the preferred method for representing analysis and design models. The Unified Process (Chapter 4) has been developed to provide a framework for the application of UML. Scott Ambler [Amb06] has developed a simplified version of the UP that integrates his agile modeling philosophy.

5.5.4 Agile Unified Process

The *Agile Unified Process* (AUP) adopts a “serial in the large” and “iterative in the small” [Amb06] philosophy for building computer-based systems. By adopting the classic UP phased activities—*inception*, *elaboration*, *construction*, and *transition*—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities [Amb06]:

- *Modeling.* UML representations of the business and problem domains are created. However, to stay agile, these models should be “just barely good enough” [Amb06] to allow the team to proceed.
- *Implementation.* Models are translated into source code.
- *Testing.* Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- *Deployment.* Like the generic process activity discussed in Chapters 3, deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- *Configuration and project management.* In the context of AUP, configuration management (Chapter 29) addresses change management, risk management, and the control of any persistent work products¹² that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

11 A brief tutorial on UML is presented in Appendix 1.

12 A *persistent work product* is a model or document or test case produced by the team that will be kept for an indeterminate period of time. It will *not* be discarded once the software increment is delivered.

- *Environment management.* Environmental management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Although the AUP has historical and technical connections to the Unified Modeling Language, it is important to note that UML modeling can be used in conjunction with any of the agile process models described in this chapter.

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 4) 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:¹³

Note: Because agile development is a hot topic, most software tools vendors purport to sell tools that

support the agile approach. The tools noted here have characteristics that make them particularly useful for agile projects.

OnTime, developed by Axosoft (www.axosoft.com), provides agile process management support for various technical activities within the process.

Ideogramic UML, developed by Ideogramic (<http://ideogramic-uml.software.informer.com/>) is a UML tool set specifically developed for use within an agile process.

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

5.6 A TOOL SET FOR THE AGILE PROCESS

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not at all pivotal to the success of the team. However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that "agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops."

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04] or modern social networking techniques) that provides information and coordination among agile developers. Active communication is achieved via the team dynamics (e.g., pair programming), while

KEY POINT

The "tool set" that supports agile processes focuses more on people issues than it does on technology issues.

¹³ 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.

passive communication is achieved by “information radiators” (e.g., a flat panel display that presents the overall status of different components of an increment). Project management tools deemphasize the Gantt chart and replace it with earned value charts or “graphs of tests created versus passed . . . other agile tools are using to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)” [Coc04].

Are any of these things really tools? They are, if they facilitate the work performed by an agile team member and enhance the quality of the end product.

5.7 SUMMARY

In a modern economy, market conditions change rapidly, customer and end-user needs evolve, and new competitive threats emerge without warning. Practitioners must approach software engineering in a manner that allows them to remain agile—to define maneuverable, adaptive, lean processes that can accommodate the needs of modern business.

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 that deliver features and functionality that have been described and then prioritized by stakeholders.

Other agile process models also stress human collaboration and team self-organization, but define their own framework activities and select different points of emphasis. For example, Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight time lines, 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. The Dynamic Systems Development Method (DSDM) advocates the use of time-box scheduling and suggests that only enough work is required for each software increment to facilitate movement to the next increment. 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. The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building software.

PROBLEMS AND POINTS TO PONDER

- 5.1. 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?
- 5.2. Describe agility (for software projects) in your own words.
- 5.3. 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.
- 5.4. Could each of the agile processes be described using the generic framework activities noted in Chapter 3? Build a table that maps the generic activities into the activities defined for each agile process.
- 5.5. Try to come up with one more “agility principle” that would help a software engineering team become even more maneuverable.
- 5.6. Select one agility principle noted in Section 5.3.1 and try to determine whether each of the process models presented in this chapter exhibits the principle. [Note: We have presented an overview of these process models only, so it may not be possible to determine whether a principle has been addressed by one or more of the models, unless you do additional research (which is not required for this problem).]
- 5.7. Why do requirements change so much? After all, don’t people know what they want?
- 5.8. 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?
- 5.9. Write an XP user story that describes the “favorite places” or “favorites” feature available on most Web browsers.
- 5.10. What is a spike solution in XP?
- 5.11. Describe the XP concepts of refactoring and pair programming in your own words.
- 5.12. Using the process pattern template presented in Chapter 3, develop a process pattern for any one of the Scrum patterns presented in Section 5.5.1.
- 5.13. Visit the Official Agile Modeling site and make a complete list of all core and supplementary AM principles.
- 5.14. The tool set proposed in Section 5.6 supports many of the “soft” aspects of agile methods. Since communication is so important, recommend an actual tool set that might be used to enhance communication among stakeholders on an agile team.

FURTHER READINGS AND INFORMATION SOURCES

The overall philosophy and underlying principles of agile software development are considered in-depth in many of the books referenced in the body of this chapter. In addition, books by Pichler (*Agile Project Management with Scrum: Creating Products that Customers Love*, Addison-Wesley, 2010), Highsmith (*Agile Project Management: Creating Innovative Products*, 2nd ed. Addison-Wesley, 2009), Shore and Chromatic (*The Art of Agile Development*, O’Reilly Media, 2008), Hunt (*Agile Software Construction*, Springer, 2005), and Carmichael and Haywood (*Better Software Faster*, Prentice Hall, 2002) present useful discussions of the subject. Aguanno (*Managing Agile Projects*, Multi-Media Publications, 2005), and Larman (*Agile and*

Iterative Development: A Manager's Guide, Addison-Wesley, 2003) present a management overview and consider project management issues. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) presents a survey of agile principles, processes, and practices. A worthwhile discussion of the delicate balance between agility and discipline is presented by Booch and his colleagues (*Balancing Agility and Discipline*, Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2009) presents the principles, patterns, and practices required to develop “clean code” in an agile software engineering environment. Leffingwell (*Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, Addison-Wesley, 2011) and (*Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) discusses strategies for scaling up agile practices for large projects. Lippert and Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley, 2006) discuss the use of refactoring when applied in large, complex systems. Stamelos and Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) discuss SQA techniques that conform to the agile philosophy.

Dozens of books have been written about Extreme Programming over the past decade. Beck (*Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, 2004) remains the definitive treatment of the subject. In addition, 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).

Kohut (*Professional Agile Development Process: Real World Development Using SCRUM*, Wrox, 2013), Rubin (*Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, 2012), Larman and Vodde (*Scaling Lean and Agile Development: Thinking and Organizational Tools for Large Scale Scrum*, Addison-Wesley, 2008), and Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) discuss the use of Scrum for projects that have a major business impact. The nuts and bolts of Scrum are discussed by Cohn (*Succeeding with Agile*, Addison-Wesley, 2009), and Schwaber and Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Worthwhile treatments of DSDM have been written by the DSDM Consortium (*DSDM: Business Focused Development*, 2nd ed., Pearson Education, 2003) and Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997).

Books by Ambler and Lines (*Disciplined Agile Delivery: A Practitioner's Guide to Agile Delivery in the Enterprise*, IBM Press, 2012) and Poppendieck and Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) provide guidelines for managing and controlling agile projects. Ambler and Jeffries (*Agile Modeling*, Wiley, 2002) discuss AM in some depth.

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 website: www.mhhe.com/pressman.