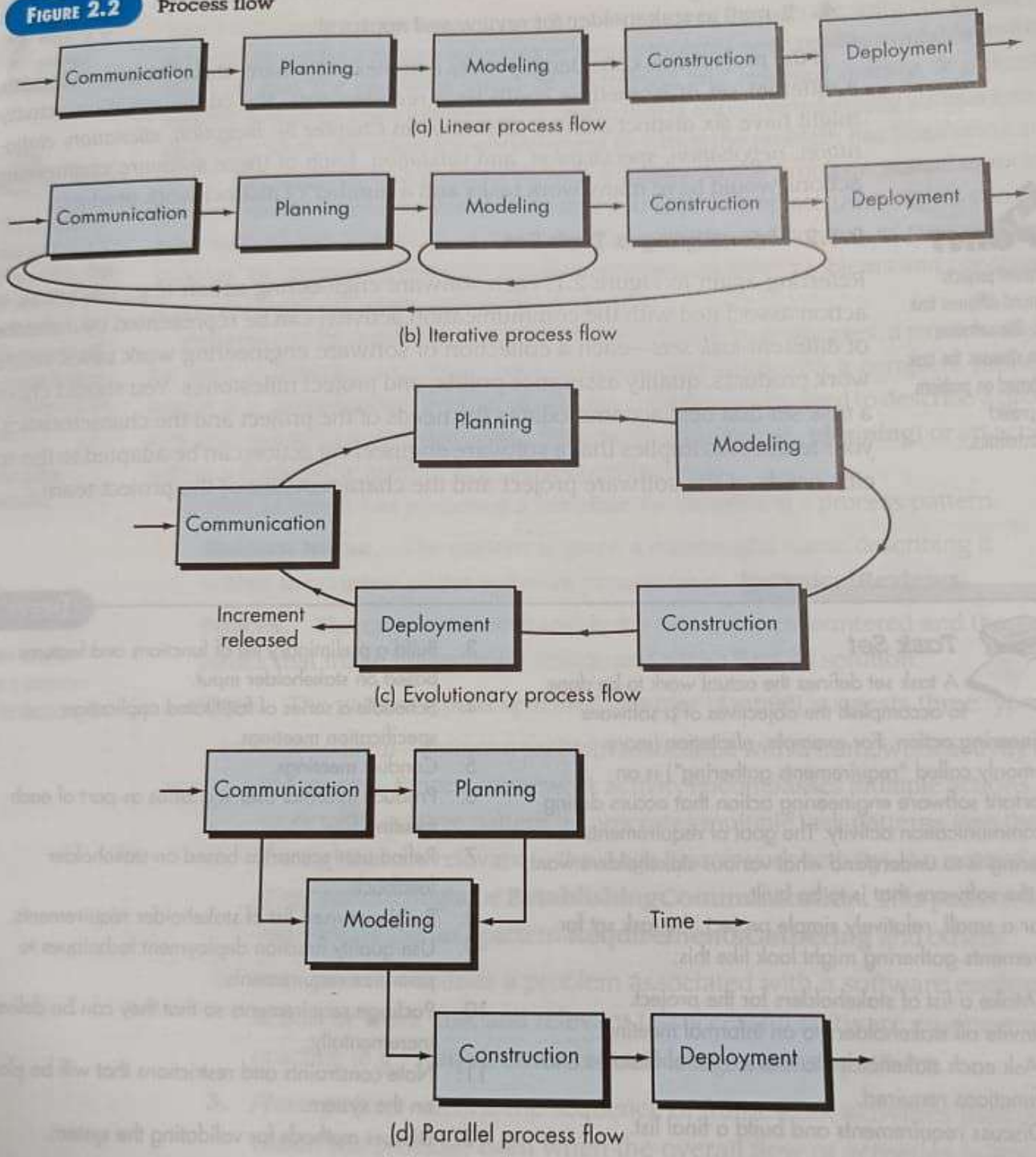
FIGURE 2.2 Process flow



? How does a framework activity change as the nature of the project changes?

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation*, and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and take notes.

after release is very significant. Despite the fact that it is a very important and challenging task; it is routinely the poorly managed headache that nobody wants to face.

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities, and optimization. The purpose of this phase is to preserve the value of the software over time. This phase may span for 5 to 50 years whereas development may be 1 to 3 years.

This model is easy to understand and reinforces the notion of "define before design" and "design before code". This model expects complete and accurate requirements early in the process, which is unrealistic. Working software is not available until relatively late in the process, thus delaying the discovery of serious errors. It also does not incorporate any kind of risk assessment.

Problems of waterfall model

(*i*) It is difficult to define all requirements at the beginning of a project.

(*ii*) This model is not suitable for accomodating any change.

(*iii*) A working version of the system is not seen until late in the project's life.

(*iv*) It does not scale up well to large projects.

(*v*) Real projects are rarely sequential.

Due to these weaknesses, the application of waterfall model should be limited to situations where the requirements and their implementation are well understood. For example, if an organisation has experience in developing accounting systems then building a new accounting system based on existing designs could be easily managed with the waterfall model.

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.

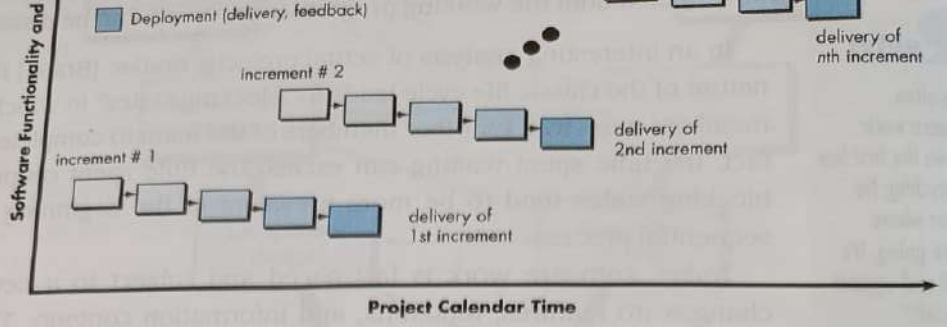
2.3.2 Incremental Process Models

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely 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 elements of linear and parallel process flows discussed in Section 2.1. Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow (Section 2.3.3).

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.

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.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.[8]

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

2.3.3 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

8 It is important to note that an incremental philosophy is also used for all "agile" process models discussed in Chapter 3.

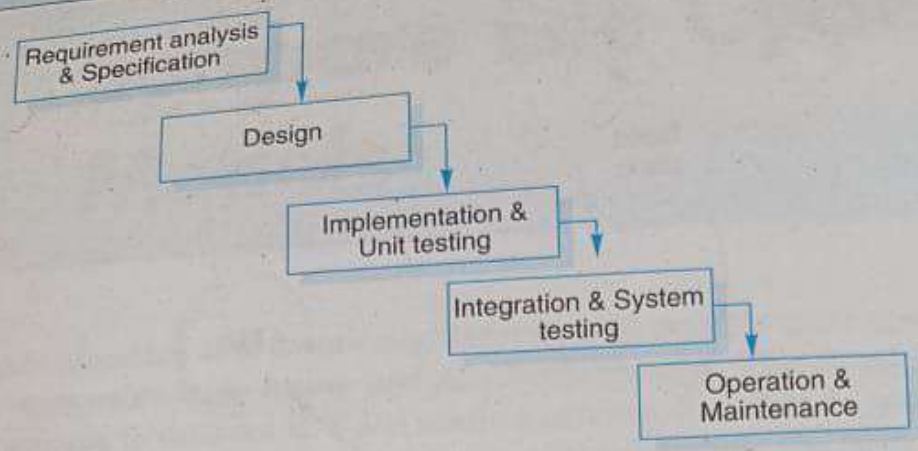


Fig. 2.2: Waterfall model

2. **Design phase:** The SRS document is produced in the previous phase, which contains the exact requirements of the customer. The goal of this phase is to transform the requirements specification into a structure that is suitable for implementation in some programming language. Here, overall software architecture is defined, and the high level and detailed design work is performed. This work is documented and known as software design description (SDD) document. The information contained in the SDD should be sufficient to begin the coding phase.

3. **Implementation and unit testing phase:** During this phase, design is implemented. If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by the software developers is contained in the SDD.

During testing, the major activities are centered around the examination and modification of the code. Initially, small modules are tested in isolation from the rest of the software product. There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? Such problems are solved in this phase and modules are tested after writing some overhead code.

4. **Integration and system testing phase:** This is a very important phase. Effective testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, and more accurate and reliable results. It is a very expensive activity and consumes one-third to one half of the cost of a typical development project.

As we know, the purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing is performed. System testing involves the testing of the entire system, whereas software is a part of the system. This is essential to build confidence in the developers before software is delivered to the customer or released in the market.

5. **Operation and maintenance phase:** Software maintenance is a task that every development group has to face, when the software is delivered to the customer's site, installed and is operational. Therefore, release of software inaugurates the operation and maintenance phase of the life cycle. The time spent and effort required to keep the software operational

unfixable and unenhanceable. There is no room for design. ... to be carried out in a structured or detailed way. The cost of the development using this approach is actually very high as compared to the cost of a properly specified and carefully designed product. In addition, maintenance of the product can be extremely difficult without specification or design documents.

2.2 THE WATERFALL MODEL

The most familiar model is the waterfall model, which is given in Fig. 2.2. This model has five phases: Requirements analysis and specification, design, implementation and unit testing, integration and system testing, and operation and maintenance. The phases always occur in this order and do not overlap. The developer must complete each phase before the next phase begins. This model is named "Waterfall Model", because its diagrammatic representation resembles a cascade of waterfalls.

1. **Requirement analysis and specification phase:** The goal of this phase is to understand the exact requirements of the customer and to document them properly. This activity is usually executed together with the customer, as the goal is to document all functions, performance and interfacing requirements for the software. The requirements describe the "what" of a system, not the "how". This phase produces a large document, written in a natural language, contains a description of what the system will do without describing how it will be done. The resultant document is known as software requirement specification (SRS) document.

The SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure to implement the contracted system.

onal) company that makes consumer products for and commercial use.

layers: Lee Warren, engineering manager; Doug software engineering manager; Jamie Lazar, re team member; Vinod Raman, software team er; and Ed Robbins, software team member.

onversation:

o let's recapitulate. I've spent some time discussing eHome product line as we see it at the moment. bt, we've got a lot of work to do to simply define , but I'd like you guys to begin thinking about're going to approach the software part of this

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

't know, Doug, we always got product out

ue, but not without a lot of grief, and this ks like it's bigger and more complex than ve've done in the past.

oesn't look that hard, but I agree . . . our proach to past projects won't work here, if we have a very tight time line.

learned a lot about software engineering . . . go 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 this chapter 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 OK. 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.

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:

> 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.

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.

VICE

nagement fixed-budget ent a bad idea), an be a s each mpleted, is revisited

te:

his far morrow ay."

thews

AFEHOME

Selecting a Process Model, Part 2

The scene: Meeting room for the engineering group at CPI Corporation, a that makes consumer products for home and ial use.

yers: Lee Warren, engineering manager; Doug ftware engineering manager; Vinod and Jamie, of the software engineering team.

versation: [Doug describes evolutionary ptions.]

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

In this model, the flow of activity
out properly, the chances of reverting to an earlier
remote. In a system environment where there is clarity and unanimity on the requirements and specifications, LSM works very efficiently.

✓ ***The Prototyping Model (PRM)***

This model is proposed when users, customers, or stakeholders are not sure about their requirements. Also, sometimes, the software engineer is not sure what kind of design solution would be the most appropriate. In all such situations, the prototype model is built to ascertain the requirement specifications, solution and, in some cases, the most appropriate technology. The advantage of using the prototype model is that with a lesser investment of finances and time, the customers' requirements are confirmed. The risk of losing time and a large investment in hardware and development of software is eliminated. Figure 1.8 shows the Prototype Model (PRM).

The success of PRM depends on how good the prototype is and its ability to confirm the requirements of the customer.

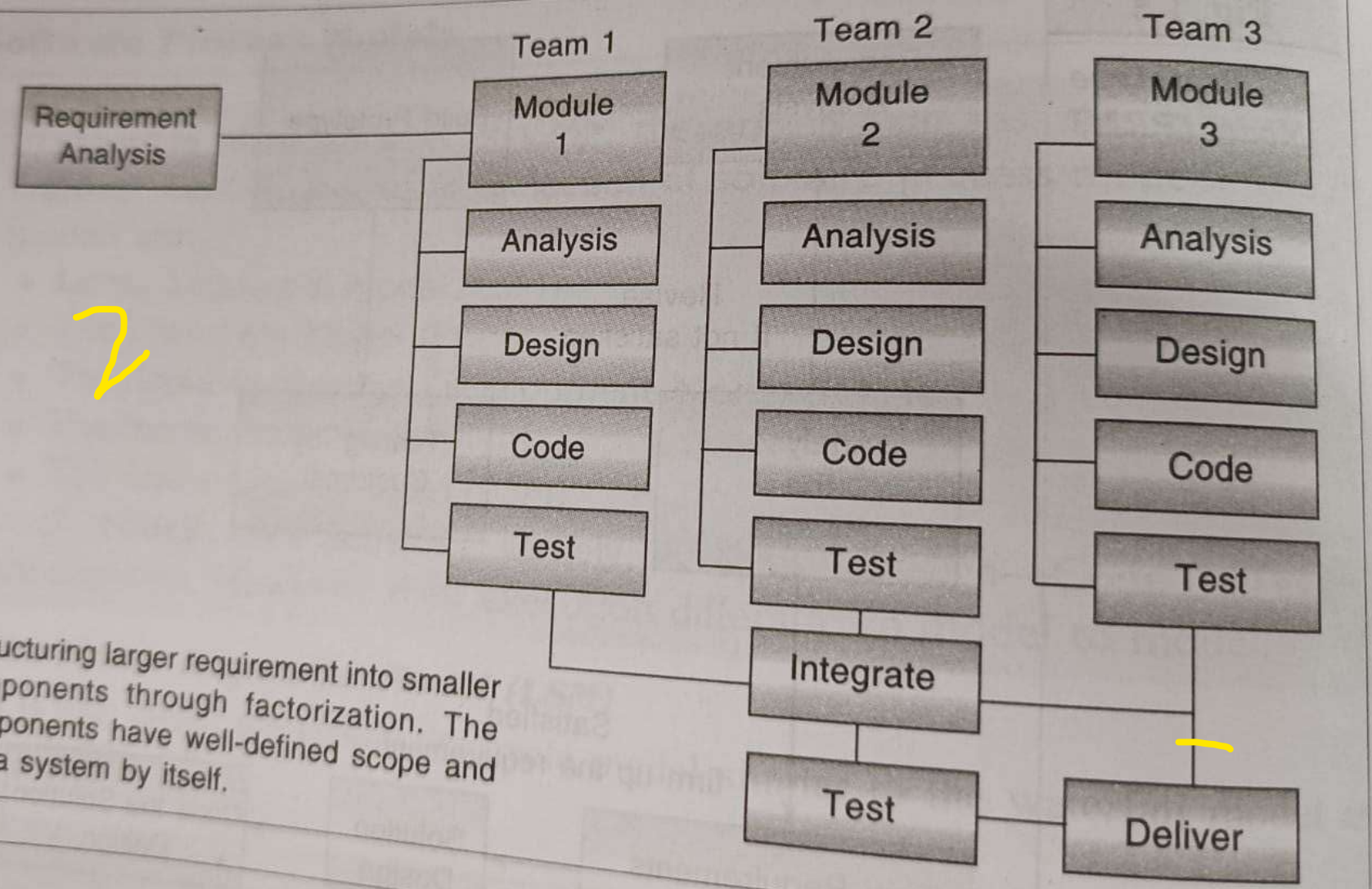
✓ ***The Rapid Application Development Model (RAD)***

The RAD model is proposed when requirements and solutions can be modularized as independent system or software components, each of which can be developed

1

Fig. 1.9

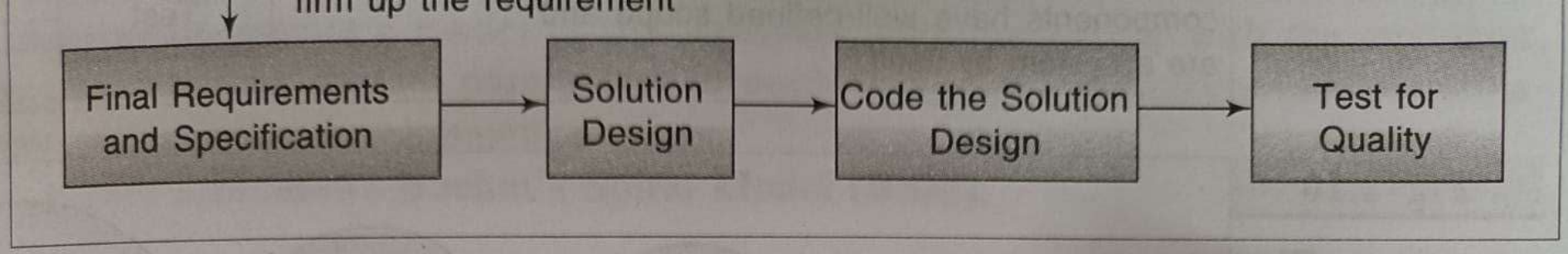
Rapid Application (RAD) Model



*Structuring larger requirement into smaller components through factorization. The components have well-defined scope and are a system by itself.

Fig. 1.10

The Incremental Model (INM)



by different teams. After these smaller system components are developed, they are integrated to produce the larger software system solution. The modularization could be on a functional, technology or architectural basis, such as front-end-back-end, client side-server side and so on.

RAD becomes faster if the software engineer uses component technology such that the components are really available for reuse. Since the development is distributed into component development teams, the teams work in tandem and total development is completed in a short period.

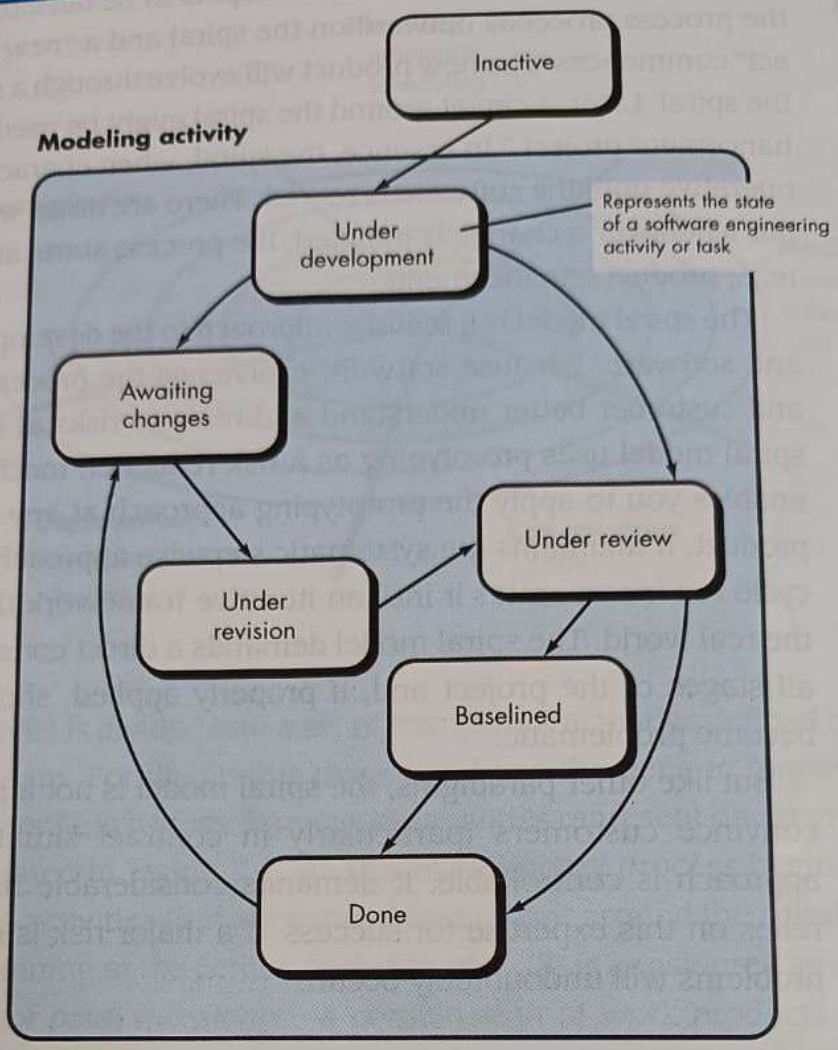
Figure 1.9 shows the RAD model.

RAD is also recommended when system components have already been developed by the organisation in the context of other software systems, and these components can be used in the new system with minor or no modification. The model is called RAD due to the reusability of the system components and the possibility of splitting the requirement into smaller components that can then be assigned to different development teams

✓ *The Incremental Model (INM)*

It has been observed that often the nature of the requirement is complex and that visualizing it in precise terms is difficult. Such a situation requires a lot of experimentation through trial and error in order to finalise the requirement analysis and specifications. Often, requirement needs to be built up in parts to avoid the risk of

FIGURE 2.8
One element of the concurrent process model



2.3.4 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 defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.[11]

ADVICE

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

Figure 2.8 provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—**modeling**—may be in any one of the states[12] 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.

---

11 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.

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

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 **inactive** 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 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.

Quote:

"ry process in organization a customer, without a omer a process no purpose."

Daniel Hunt

2.3.5 A Final Word on Evolutionary Processes

I 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.[13]

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. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.
>
> 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 produc-

GDPA, ... Bremen University in Germany (**www.informatik** ... **.processwave.net/Links/tool_links.htm**.

2.5 The Unified Process

In their seminal book on the *Unified Process,* 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 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

17 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.

WebRef
An interesting discussion of the UP in the context of agile development can be found at **www.ambysoft.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, requirements 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 tests[21] 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.

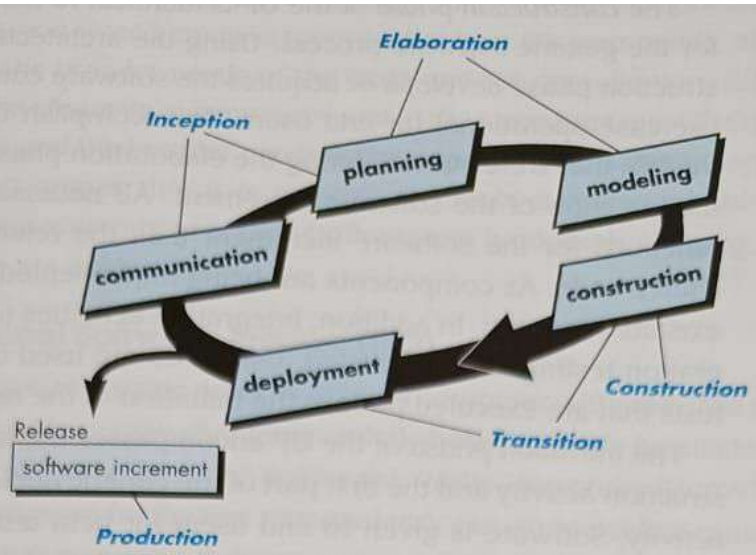
It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set (described earlier in this chapter). 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.

2.6 Personal and Team Process Models

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 signif

FIGURE 2.9
The Unified Process



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 5) 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 function 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 2.9). 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 requirements 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.[20] The architectural baseline demonstrates the viability of the 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.

---

20 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.

(Chapter 3). The Unified Process recognizes the importance of cus... ...unic-
cation and streamlined methods for describing the customer's view of a system
(the use case[18]). 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.

2.5.1 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 unfa-
miliar 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.

UML provided the necessary technology to support object-oriented software engi-
neering practice, but it did not provide the process framework to guide project teams
in their application of the technology. Over the next few years, Jacobson, Rumbaugh,
and Booch developed the *Unified Process,* a framework for object-oriented software
engineering using UML. Today, the Unified Process (UP) and UML are widely used on
object-oriented projects of all kinds. The iterative, incremental model proposed by the
UP can and should be adapted to meet specific project needs.

2.5.2 Phases of the Unified Process[19]

Earlier in this chapter, I discussed five generic framework activities and argued that
they may be used to describe any software process model. The Unified Process is no
exception. Figure 2.9 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.

---

18 A *use case* (Chapter 5) 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 requirements model.

19 The Unified Process is sometimes called the *Rational Unified Process* (RUP) after the Rational Cor-
poration (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.

KEY POINT

elf-organizing team
n control of the
rk it performs. The
m makes its own
mitments and
es plans to
ve them.

Self-organization. ... implies three things: (1) the agile team ... envi-
done, (2) the team organizes the process to best accommodate its local envi-
ronment, (3) the team organizes the work schedule to best achieve delivery
of the software increment. Self-organization has a number of technical bene-
fits, but more importantly, it serves to improve collaboration and boost team
morale. In essence, the team serves as its own management. Ken Schwaber
[Sch02] addresses these issues when he writes: "The team selects how much
work it believes it can perform within the iteration, and the team commits to
the work. Nothing demotivates a team as much as someone else making
commitments for it. Nothing motivates a team as much as accepting the
responsibility for fulfilling commitments that it made itself."

3.4 Extreme Programming (XP)

In order to illustrate an agile process in a bit more detail, I'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]. More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed [Ker05]. IXP refines XP and targets the agile process specifically for use within large organizations.

3.4.1 XP Values

Beck [Bec04a] defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

In order to achieve effective *communication* between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors[3] for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

3 In the XP context, a *metaphor* is "a story that everyone—customers, programmers, and managers—can tell about how the system works" [Bec04a].

ADVICE

Keep it simple whenever you can, but recognize that continual "refactoring" can absorb significant time and resources.

To achieve *simplicity*, XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored*[4] at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy (Chapters 17 through 20), the software (via test results) provides the agile team with feedback. XP makes use of the *unit test* as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the *user stories* or *use cases* (Chapter 5) that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.

Quote:

"XP is the answer to the question, 'How little can we do and still build great software?'"

Anonymous

Beck [Bec04a] argues that strict adherence to certain XP practices demands *courage*. A better word might be *discipline*. For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates *respect* among it members, between other stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

3.4.2 The XP Process

WebRef

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

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 3.2 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 of the XP team to understand the business context for the software and to get a broad

4 Refactoring allows a software engineer to improve the internal structure of a design (or source code) without changing its external functionality or behavior. In essence, refactoring can be used to improve the efficiency, readability, or performance of a design or the code that implements a design.

KEY POINT

Project velocity is a subtle measure of team productivity.

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.[6]

ADVICE

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

XP encourages the use of CRC cards (Chapter 7) as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes[7] 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 8. The CRC cards are the only design work product produced as part of the XP process.

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

WebRef

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

In the preceding section, we noted that XP encourages *refactoring*—a construction technique that is also a method for design optimization. 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

---

6 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.

7 Object-oriented classes are discussed in Appendix 2, in Chapter 8, and throughout Part 2 of this book.

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.

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

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

Coding. 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).[8] Once the unit test[9] has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

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

? 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.

ADVICE
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 17) that helps to uncover errors early.

? How are unit tests used in XP?

Testing. I have already noted that the creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 17) whenever code is modified (which is often, given the XP refactoring philosophy).

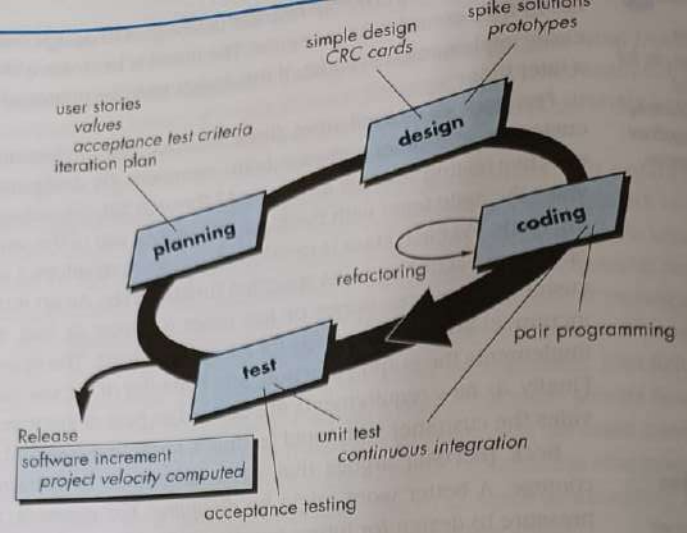
---

8 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.

9 Unit testing, discussed in detail in Chapter 17, 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.

FIGURE 3.2

The Extreme Programming process



? What is an XP "story"?

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 5) 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.[5] 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.

WebRef

A worthwhile XP "planning game" can be found at: **c2.com/cgi/wiki?planningGame.**

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

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

KEY POINT

ASD emphasizes learning as a key element in achieving a "self-organizing" team.

adaptive cycle, the emphasis is on "learning" as much as it is on progress toward a completed cycle. In fact, Highsmith [Hig00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups (Chapter 5), technical reviews (Chapter 14), and project postmortems.

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

3.5.2 Scrum

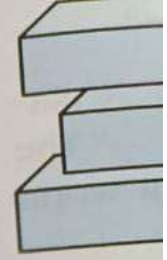
WebRef

Scrum ation and es can be found w lchaos.com.

Scrum (the name is derived from an activity that occurs during a rugby match[13]) 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 [Sch01a].

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

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



KEY PO

Scrum inc set of pro that emp priorities compart work un commu frequen feedbac

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 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members [Noy02]:

3

- 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.

3.5.3 Dynamic Systems Development Method (DSDM)

Ref

urces for
e found at
m.org.

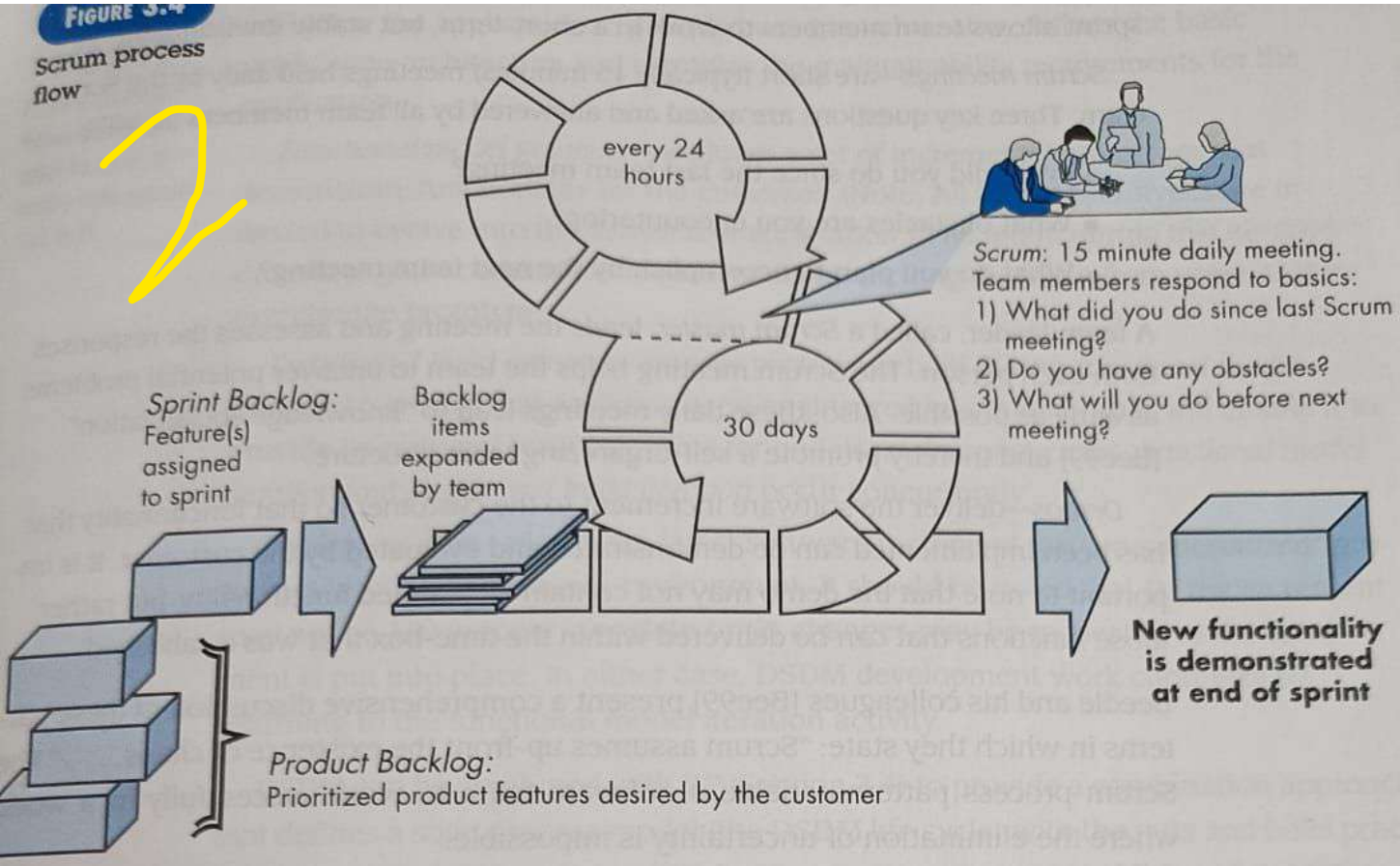
The *Dynamic Systems Development Method* (DSDM) [Sta97] is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" [CCS02]. The DSDM philosophy 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 has defined an agile process model, called the *DSDM life cycle*, ... ferent iterative cycles ...

FIGURE 3.4

Scrum process flow



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 3.4.

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 actions:

KEY POINT

m incorporates a f process patterns emphasize project ties, artmentalized units, unication, and ent customer ack.

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

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box[14] (typically 30 days).

14 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.

AGILE DEVELOPMENT

3

KEY CONCEPTS

Adaptive Software Development ...81
agile process ...68
Agile Unified Process 89
agility 67
Crystal 85
DSDM 84
Extreme Programming ...72

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01a] (referred to as the "Agile Alliance") signed the "Manifesto for Agile Software Development." It stated:

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

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.

3.1 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.

ADVICE

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

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

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.

3.2 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 3.1, 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 specification can be edited. The costs of doing this work are minimal, and the time required will

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.

3.3.2 The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

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

No one is against agility. The real question is: What is the best way to achieve it? As important, how do 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 3.4), 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

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

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 I 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.

3.3.1 Agility Principles

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

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

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

.....86
trial XP ...77
Software
lopment ...87
ramming ...76
ect
city 74
ctoring 75
m 82
ries 74
process 73

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, agile[1] methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it is not applicable to all projects, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a Web-based application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, 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 throughout 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.

Quote:

"Agility: 1, everything else: 0."

Tom DeMarco

In a thought-provoking book on agile software development, Alistair Cockburn [Coc02] argues that the prescriptive process models introduced in Chapter 2 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 [either] 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.

---

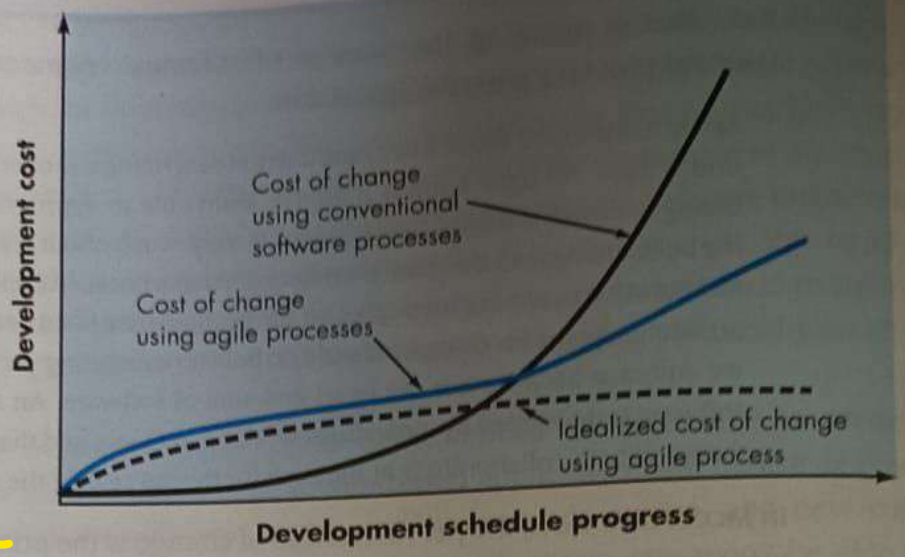
1 Agile methods are sometimes referred to as *light methods* or *lean methods*.

ADVICE

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

FIGURE 3.1

Change costs as a function of time in development



Quote:

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

—Steven Goldman et al.

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.

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.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 3.1, 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.

3.3 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.

The project manager is responsible for managing these knowledge areas and integrating them with the assistance of tools and techniques to meet the project success goals. Let us elaborate these knowledge areas further.

- Scope: Scope management defines the requirements that, when delivered, will be the successful conclusion of the project.
- Time: Time management means estimating the time for project completion, and time for each activity in the project; scheduling the activities of the project using activity time; and monitoring the activity progress to complete the project on time as agreed with the customer.
- Cost: Cost management means estimating the effort and resources required, costing them for budgeting and controlling them throughout the project duration. Cost management also includes control of cost of outsourcing and cost of procurement items (H & S). All these costs need to be managed to ensure that costs are within the budget to keep project's profit margin intact.
- Quality: Quality management means producing the quality of all deliverables as expected by the customer and stakeholders.
- Human resource: HR Management means managing people and their skills effectively as estimated to deliver quality products within cost and time.
- Communications: Communications Management means capturing all aspects of the data on a project and its activities , processing the data, and distributing it to the concerned people so as to respond effectively to meet the commitments.
- Risk: Risk management means identification, analysis, build-monitoring and mitigation plans to control the risk.
- Procurement: Procurement management means identifying procurement requirements, such as development and engineering services, hardware and software as a support to the project.
- Integration: Integration management means integrating various tasks in the scope to produce project deliverables as per the expectations of the customer within cost and time budgets.

Project management software is a tool that can facilitate the management processes described