CHAPTER

AGILITY AND PROCESS

3

In 2001, a group of noted software developers, writers, and consultants [Bec01] signed the "Manifesto for Agile Software Development" in which they argued in favor of "individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan."

KEY CONCEPTS

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).

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? Modern business environments that spawn computer-based systems and software products are fast paced and ever changing. Agile software engineering represents a reasonable alternative to conventional software engineering. 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.

What is the work product? The most important work product is an operational "software increment" that is delivered to the customer on the appropriate commitment date. The most important documents created are the use stories and their associated test cases.

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.

The underlying ideas that guide agile development led to the development of agile[1] methods designed to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it may not be 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 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.

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 and significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not. 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.

3.1 What Is Agility?

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

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, and 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

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

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.

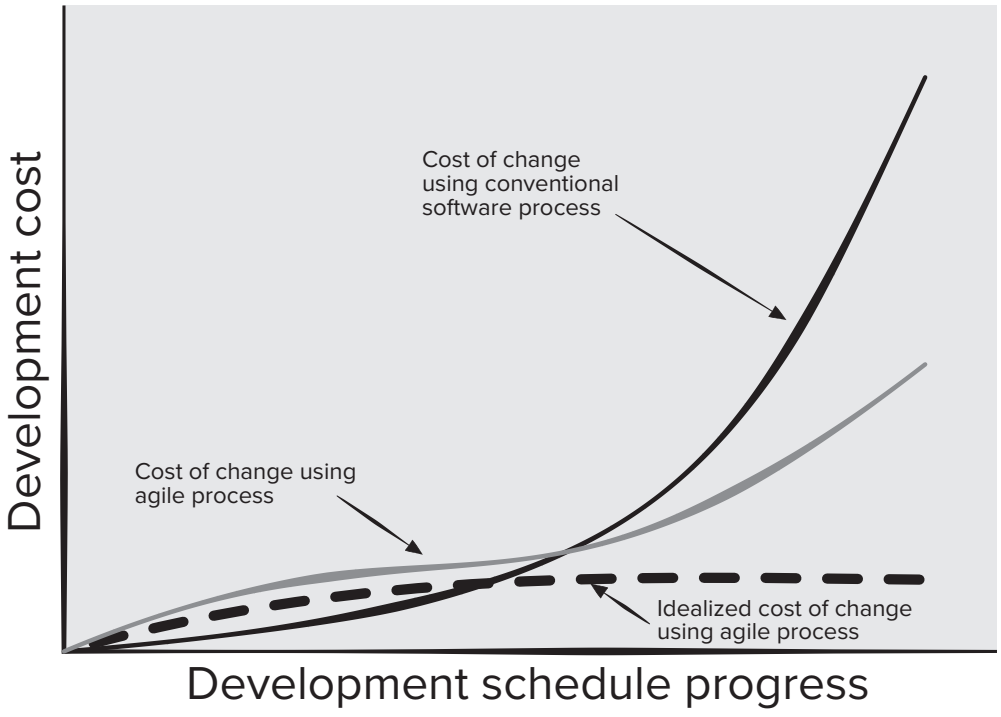
3.2 AGILITY AND THE COST OF CHANGE

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 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 effort required to ensure that the change is made without unintended side effects are nontrivial.

Proponents of agility (e.g., [Bec99], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 3.1, shaded, solid curve), allowing

FIGURE 3.1

Change costs as a function of time in development



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 briefly in Section 3.5.1 and in more detail in Chapter 20), 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.
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.

3.3.1 Agility Principles

The Agile Alliance [Agi17][2] defines 12 principles for those software organizations that want to achieve agility. These principles are summarized in the paragraphs that follow.

Customer satisfaction is achieved by providing value through software that is delivered to the customer as rapidly as possible. To achieve this, agile developers

2 The Agile Alliance home page contains much useful information: https://www.agilealliance.org/.

recognize that requirements will change. They deliver software increments frequently and work together with all stakeholders so that feedback on their deliveries is rapid and meaningful.

An agile team is populated by motivated individuals, who communicate face-to face and work in an environment that is conducive to high quality software development. The team follows a process that encourages technical excellence and good design, emphasizing simplicity—"the art of maximized the amount of work not done" [Agi17]. Working software that meets customer needs is their primary goal, and the pace and direction of the team's work must be "sustainable," enabling them to work effectively for long periods of time.

An agile team is a "self-organizing team"—one that can develop well-structured architectures that lead to solid designs and customer satisfaction. Part of the team culture is to consider its work introspectively, always with the intent of improving the manner in which it addresses its primary goal.

Not every agile process model applies characteristics described in this section with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more agile principles. However, these 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? Keep in mind that working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability. 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?

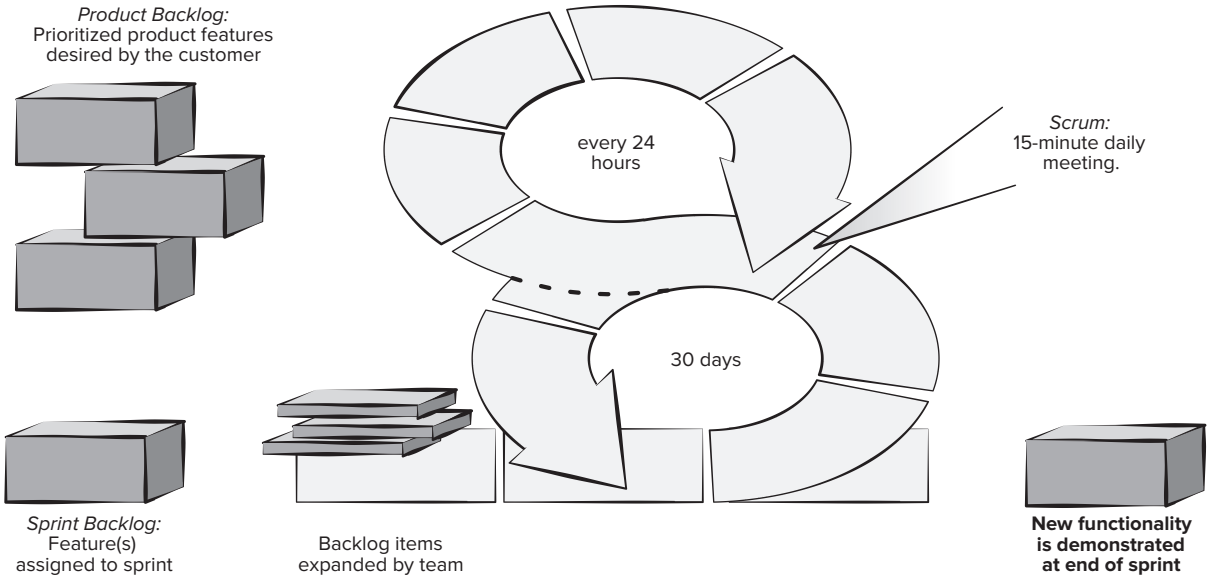
There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed framework models (Sections 3.4 and 3.5), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. The bottom line is there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

3.4 SCRUM

Scrum (the name is derived from an activity that occurs during a rugby match)[3] is a very popular agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Further development on the Scrum methods was 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 take place in a relatively short time-boxed[4] period called a *sprint*. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on size of the product and its complexity) 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 3.2. Much of our description of the Scrum framework appears in Fowler and Sutherland [Fow16].[5]

FIGURE 3.2 **Scrum process flow**



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

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

5 The Scrum Guide is available at: https://www.Scrum.org/resources/what-is-Scrum.

SAFEHOME

Considering Agile Software Development

The scene: Doug Miller's office.

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

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

Jamie: Doug, you got a minute?

Doug: Sure Jamie, what's up?

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

Doug: And?

Vinod: I was talking to a friend at another company, and he was telling me about Scrum. 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 sprints to deliver software increments when the team decides the product is done . . . it's pretty cool, I think.

Doug: It does have a lot of really good ideas. I like the sprint concept, the emphasis on early test case creation, and the idea that the process owner should be part of the team.

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

Doug (nodding): They're stakeholders but not really the product owner. That would be Marg.

Jamie: Good. She will filter the changes marketing will want to send every 5 minutes.

Vinod: Even so, my friend said that there are ways to "embrace" changes during an agile project.

Doug: So you guys think we should use Scrum?

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 Scrum 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 Scrum 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 Scrum approach?

Jamie (speaking for both): It can be adapted to fit our needs. Besides, 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 needs 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.

3.4.1 Scrum Teams and Artifacts

The Scrum team is a self-organizing interdisciplinary team consisting of a *product owner*, a *Scrum master,* and a small (three to six people) *development team.* The principle Scrum artifacts are the *product backlog*, the *sprint backlog*, and the code *increment*. Development proceeds by breaking the project into a series of incremental prototype development periods 2 to 4 weeks in length called *sprints*.

The product backlog is a prioritized list of product requirements or features that provide business value for the customer. Items can be added to the backlog at any time with the approval of the product owner and the consent of the development team. The product owner orders the items in the product backlog to meet the most important goals of all stakeholders. The product backlog is never complete while the product is evolving to meet stakeholder needs. The product owner is the only person who decides whether to end a sprint prematurely or extend the sprint if the increment is not accepted.

The sprint backlog is the subset of product backlog items selected by the product team to be completed as the code increment during the current active sprint. The increment is the union of all product backlog items completed in previous sprints and all backlog items to be completed in the current sprints. The development team creates a plan for delivering a software increment containing the selected features intended to meet an important goal as negotiated with the product owner in the current sprint. Most sprints are time-boxed to be completed in 3 to 4 weeks. How the development team completes the increment is left up to the team to decide. The development team also decides when the increment is done and ready to demonstrate to the product owner. No new features can be added to the sprint backlog unless the sprint is cancelled and restarted.

The Scrum master serves as facilitator to all members of the Scrum team. She runs the daily Scrum meeting and is responsible for removing obstacles identified by team members during the meeting. She coaches the development team members to help each other complete sprint tasks when they have time available. She helps the product owner find techniques for managing the product backlog items and helps ensure that backlog items are stated in clear and concise terms.

3.4.2 Sprint Planning Meeting

Prior to beginning, any development team works with the product owner and all other stakeholders to develop the items in the product backlog. Techniques for gathering these requirements are discussed in Chapter 7. The product owner and the development team rank the items in the product backlog by the importance of the owner's business needs and the complexity of the software engineering tasks (programming and testing) required to complete each of them. Sometimes this results in the identification of missing features needed to deliver the required functionality to the end users.

Prior to starting each sprint, the product owner states her development goal for the increment to be completed in the upcoming sprint. The Scrum master and the development team select the items to move from to the sprint backlog. The development team determines what can be delivered in the increment within the constraints of the time-box allocated for the sprint and, with the Scrum master, what work will be needed to deliver the increment. The development team decides which roles are needed and how they will need to be filled.

3.4.3 Daily Scrum Meeting

The daily Scrum meeting is a 15-minute event scheduled at the start of each workday to allow team members to synchronize their activities and make plans for the next

24 hours. The Scrum master and the development team always attend the daily Scrum. Some teams allow the product owner to attend occasionally.

Three key questions are asked and answered by all team members:

- 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?

The 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. It is the Scrum master's task to clear obstacles presented before the next Scrum meeting if possible. These are not problem-solving meetings, those occur off-line and only involve the affected parties. Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure.

Some teams use these meetings to declare sprint backlog items complete or done. When the team considers all sprint backlog items complete, the team may decide to schedule a demo and review of the completed increment with the product owner.

3.4.4 Sprint Review Meeting

The sprint review occurs at the end of the sprint when the development team has judged the increment complete. The sprint review is often time-boxed as a 4-hour meeting for a 4-week sprint. The Scrum master, the development team, the product owner, and selected stakeholders attend this review. The primary activity is a *demo* of the software increment completed during the sprint. It is important to note that the demo may not contain all planned functionality, but rather those functions that were to be delivered within the time-box defined for the sprint.

The product owner may accept the increment as complete or not. If it is not accepted, the product owner and the stakeholders provide feedback to allow a new round of sprint planning to take place. This is the time when new features may be added or removed from the product backlog. The new features may affect the nature of the increment developed in the next sprint.

3.4.5 Sprint Retrospective

Ideally, before beginning another sprint planning meeting, the Scrum master will schedule a 3-hour meeting (for a 4-week sprint) with the development team called a *sprint retrospective*. During this meeting the team discusses:

- What went well in the sprint
- What could be improved
- What the team will commit to improving in the next sprint

The Scrum master leads the meeting and encourages the team to improve its development practices to become more effective for the next sprint. The team plans ways to improve product quality by adapting its definition of "done." At the end of this meeting, the team should have a good idea about the improvements needed in the next sprint and be ready to plan the increment at the next sprint planning meeting.

3.5 Other Agile Frameworks

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 frameworks—each contending for acceptance within the software development community—the agile movement has followed the same historical path.[6]

As we noted in the last section, one of the most widely used of all agile frameworks is Scrum. But many other agile frameworks have been proposed and are in use across the industry. In this section, we present a brief overview of three popular agile methods: Extreme Programming (XP), Kanban, and DevOps.

3.5.1 The XP Framework

In this section we provide a brief overview of *Extreme Programming* (XP), one of the most widely used approaches to agile software development. Kent Beck [Bec04a] wrote the seminal work on XP.

Extreme Programming encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 3.3 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. The key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity (also called the *planning game*) begins with a requirements activity called *listening*. 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 *user story* (described in Chapter 7) 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.[7] Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. It is important to note that new stories can be written at any time.

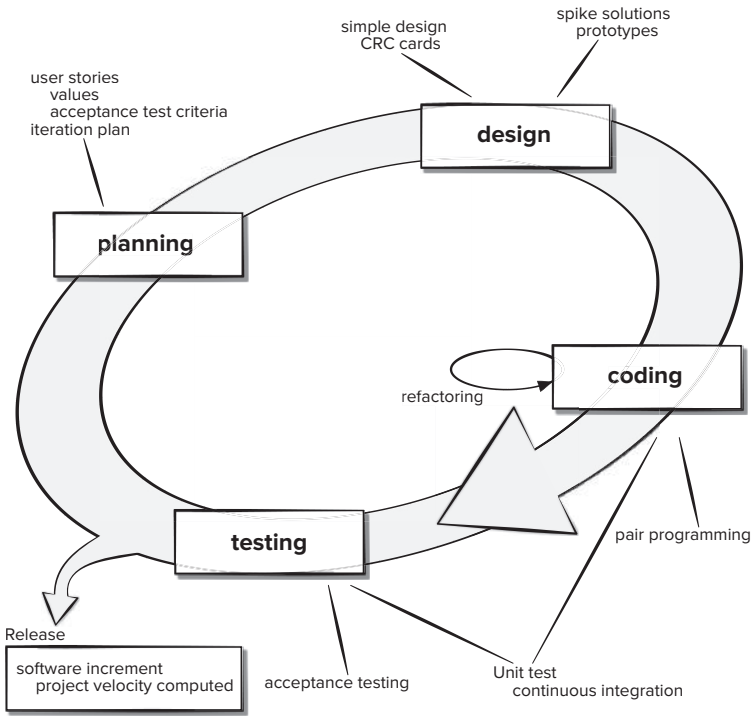
Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders 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.

6 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.

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

FIGURE 3.3

The Extreme Programming process



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 help estimate delivery dates and schedule for subsequent releases. The XP team modifies its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. The design of extra functionality (because the developer assumes it will be required later) is discouraged.[8]

XP encourages the use of CRC cards (Chapter 8) as an effective mechanism for thinking about the software in an object-oriented context. CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes[9] that are relevant to the current software increment. 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

8 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.

9 Object-oriented classes are discussed throughout Part Two of this book.

design. A central notion in XP is that design occurs both before *and after* coding commences. Refactoring—modifying/optimizing the code in a way that does not change the external behavior of the software [Fow00]—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 user 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).[10] Once the unit test[11] has been created, the developer is better able to focus on what must be implemented to pass the test. Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

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 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).[12]

As pair programmers complete their work, the code they develop is integrated with the work of others. This "continuous integration" strategy helps uncover compatibility and interfacing errors early.

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 implementing a regression testing strategy (Chapter 20) whenever code is modified (which is often, given the XP refactoring philosophy). 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. They are derived from user stories that have been implemented as part of a software release.

3.5.2 Kanban

The *Kanban* method [And16] is a lean methodology that describes methods for improving any process or workflow. Kanban is focused on change management and service delivery. Change management defines the process through which a requested change is integrated into a software-based system. Service delivery is encouraged by focusing on understanding customer needs and expectations. The team members manage the work and are given the freedom to organize themselves to complete it. Policies evolve as needed to improve outcomes.

---

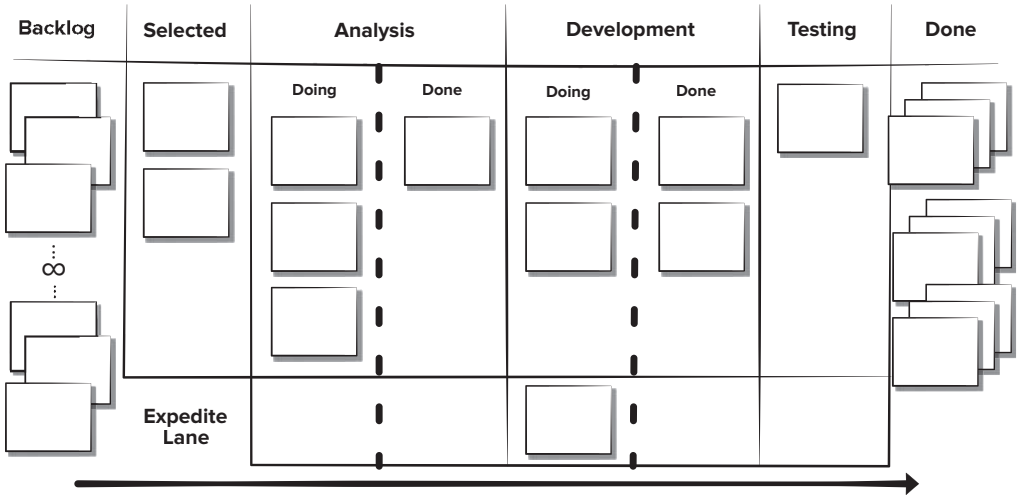
10 This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

11 Unit testing, discussed in detail in Chapter 20, 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.

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

FIGURE 3.4

Kanban board



Kanban originated at Toyota as a set of industrial engineering practices and was adapted to software development by David Anderson [And16]. Kanban itself depends on six core practices:

1. Visualizing workflow using a Kanban board (an example is shown in Figure 3.4). The Kanban board is organized into columns representing the development stage for each element of software functionality. The cards on the board might contain single user stories or recently discovered defects on sticky notes and the team would advance them from "to do," to "doing," and "done" as the project progresses.
2. Limiting the amount of *work in progress* (WIP) at any given time. Developers are encouraged to complete their current task before starting another. This reduces lead time, improves work quality, and increases the team's ability to deliver software functionality frequently to their stakeholders.
3. Managing workflow to reduce waste by understanding the current value flow, analyzing places where it is stalled, defining changes, and then implementing the changes.
4. Making process policies explicit (e.g., write down your reasons for selecting items to work on and the criteria used to define "done").
5. Focusing on continuous improvement by creating feedback loops where changes are introduced based on process data and the effects of the change on the process are measured after the changes are made.[13]
6. Make process changes collaboratively and involve all team members and other stakeholders as needed.

13 The use of process metrics is discussed in Chapter 23.

The team meetings for Kanban are like those in the Scrum framework. If Kanban is being introduced to an existing project, not all items will start in the backlog column. Developers need to place their cards in the team process column by asking themselves: Where are they now? Where did they come from? Where are they going?

The basis of the daily Kanban standup meeting is a task called "walking the board." Leadership of this meeting rotates daily. The team members identify any items missing from the board that are being worked on and add them to the board. The team tries to advance any items they can to "done." The goal is to advance the high business value items first. The team looks at the flow and tries to identify any impediments to completion by looking at workload and risks.

During the weekly retrospective meeting, process measurements are examined. The team considers where process improvements may be needed and proposes changes to be implemented. Kanban can easily be combined with other agile development practices to add a little more process discipline.

3.5.3 DevOps

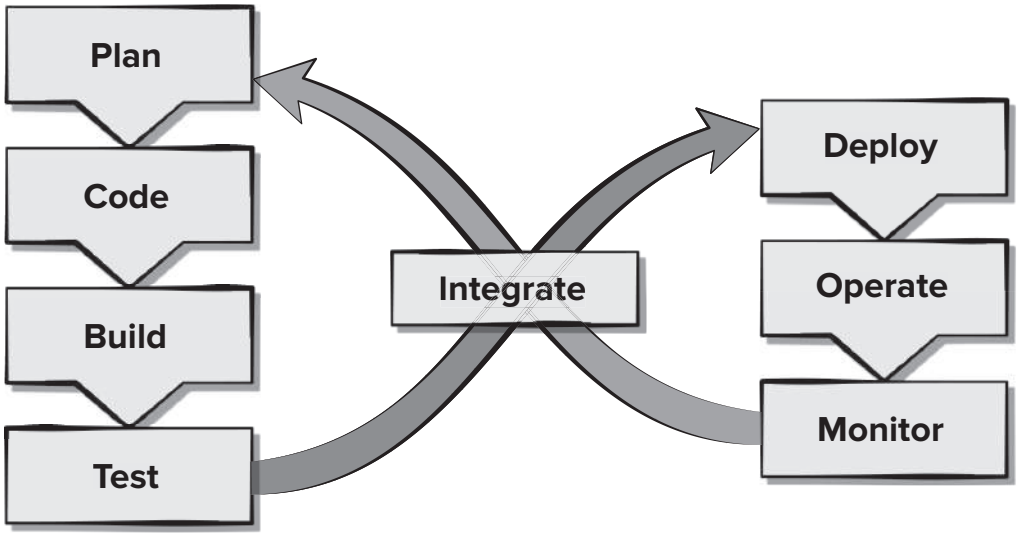
DevOps was created by Patrick DeBois [Kim16a] to combine Development and Operations. DevOps attempts to apply agile and lean development principles across the entire software supply chain. Figure 3.5 presents an overview of the DevOps workflow. The DevOps approach involves several stages that loop continuously until the desired product exists:

- **Continuous development.** Software deliverables are broken down and developed in multiple sprints with the increments delivered to the quality assurance[14] members of the development team for testing
- **Continuous testing.** Automated testing tools[15] are used to help team members test multiple code increments at the same time to ensure they are free of defects prior to integration.
- **Continuous integration.** The code pieces with new functionality are added to the existing code and to the run-time environment and then checked to ensure there are no errors after deployment.
- **Continuous deployment.** At this stage the integrated code is deployed (installed) to the production environment, which might include multiple sites globally that need to be prepared to receive the new functionality.
- **Continuous monitoring.** Operations staff who are members of the development team help to improve software quality by monitoring its performance in the production environment and proactively looking for possible problems before users find them.

DevOps enhances customers' experiences by reacting quickly to changes in their needs or desires. This can increase brand loyalty and increase market share. Lean approaches like DevOps can provide organizations with increased capacity to innovate

14 The quality assurance is discussed in Chapter 17.
15 Automated testing tools are discussed in Chapter 19.

FIGURE 3.5
DevOps



by reducing rework and allowing shifts to higher business value activities. Products do not make money until consumers have access to them, and DevOps can provide faster deployment time to production platforms [Sha17].

3.6 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.

Some of the strengths and weaknesses of the agile methods we discussed are summarized in Table 3.1. In previous editions of this book we have discussed many others. The reality is that no agile method is perfect for every project. Agile developers work on self-directed teams and are empowered to create their own process models.

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. There is no reason why a Scrum team could not adopt the use of a Kanban chart to help organize its daily planning meeting.

TABLE 3.1
Comparing agile techniques

Scrum pros	The product owner sets priorities. The team owns decision making. Documentation is lightweight. It supports frequent updating.
Scrum cons	It is difficult to control the cost of changes. It may not be suitable for large teams. It requires expert team members.
XP pros	It emphasizes customer involvement. It establishes rational plans and schedules. There is high developer commitment to the project. There is reduced likelihood of product rejection.
XP cons	There is temptation to "ship" a prototype. It requires frequent meetings about increasing costs. It may allow for excessive changes. There is a dependence on highly skilled team members.
Kanban pros	It has lower budget and time requirements. It allows for early product delivery. Process policies are written down. There is continuous process improvement.
Kanban cons	Team collaboration skills determine success. Poor business analysis can doom the project. Flexibility can cause developers to lose focus. Developer reluctance to use measurement.
DevOps pros	There is reduced time to code deployment. The team has developers and operations staff. The team has end-to-end project ownership. There is proactive monitoring of deployed product.
DevOps cons	There is pressure to work on both old and new code. There is heavy reliance on automated tools to be effective. Deployment may affect the production environment. It requires an expert development team.

Extreme programming (XP) is organized around 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. There is nothing preventing them from using DevOps techniques to decrease their time to deployment.

PROBLEMS AND POINTS TO PONDER

3.1. Read the "Manifesto of Agile Software Development" [Bec01] noted 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?

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

3.3. 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.

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

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

3.6. 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?

3.7. Write a user story that describes the "favorite places" or "favorites" feature available on most Web browsers.

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