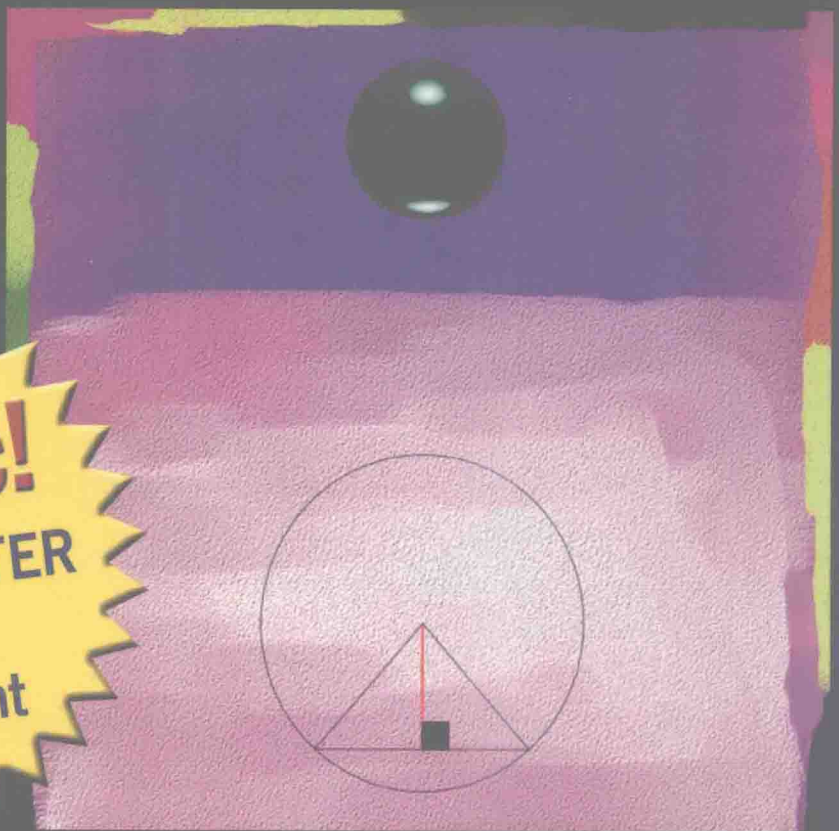
ROGER S. PRESSMAN

SOFTWARE ENGINEERING

A PRACTITIONER'S APPROACH



Hot Topic!
BONUS CHAPTER on Agile Development

FIFTH EDITION

Software Engineering

A PRACTITIONER'S APPROACH

Bonus Chapter:

Agile Development

Roger S. Pressman, Ph.D

Coming Spring 2004--New Edition of Pressman's
Software Engineering: A Practitioner's Approach
containing this material on Agile Development,
extensive new coverage on Web Engineering, and much more.

Mc Graw Hill **Higher Education**

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

The McGraw-Hill Companies

Mc Graw Hill **Higher Education**

BONUS CHAPTER: AGILE DEVELOPMENT, SIXTH EDITION

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 **BKM BKM** 0 9 8 7 6 5 4 3

ISBN 0–07–298627–1

Publisher: *Elizabeth A. Jones*
Managing developmental editor: *Emily J. Lupash*
Marketing manager: *Dawn R. Bercier*
Lead project manager: *Jill R. Peter*
Lead production supervisor: *Sandy Ludovissy*
Senior media project manager: *Jodi K. Banowetz*
Senior media technology producer: *Eric A. Weber*
Senior designer: *David W. Hash*
Cover designer: *Rhiannon Erwin*
Cover illustrator: *Joseph Gilians*
Compositor: *Carlisle Communications, Ltd.*
Typeface: *8.5/13.5 Leawood*

www.mhhe.com

CHAPTER

4

AGILE DEVELOPMENT

KEY CONCEPTS

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

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

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

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

QUICK LOOK

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

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

Why is it important? The modern business environment that spawns computer-based systems and software products is fast-paced and

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

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—customer communication, planning, modeling, construction, delivery and evaluation—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? Customers and software engineers who have adopted the agile philosophy 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.

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

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

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

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

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

If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a man-

ner that shows "tolerance" for the people who do software engineering work. Invariably, tolerant practices are easier for software people to adopt and sustain, but (as Cockburn admits) they may be less productive. Like most things in life, trade-offs must be considered.

4.1 What Is Agility?

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

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

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

Agility is dynamic, content specific, aggressively change embracing, and growth oriented.
Steven Goldman et al.

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.

4.2 What Is an Agile Process?

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

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

WebRef

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

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

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

4.2.1 Agility Principles

A set of 12 *agility principles* has been developed by the Agile Alliance (see [AGI03], [FOW01]) to establish a basis for any agile process model. Glen Alleman [ALL02] presents each of these principles (noted in italics) and then considers the domain in

which the principles are applicable and issues (including critique) associated with their applicability within that domain.[2]

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.* The concept of *customer value for work performed* is well developed in any business domain. However definitions of *value, early,* and *satisfaction* are not provided, so domain specific definitions need to be developed before this principle can be of practical use in a specific circumstance.

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

2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.* Agile process contributes to success in these situations. This might be considered the *operational* definition of agility. Early and repetitive feedback on product or project design is good practice in many engineering disciplines.

   But changing requirements may be an indication of changing business values, unstable requirements, or a lack of understanding of the desired business outcome. Without stable business success metrics, the creation of software to address unstable requirements is not good business strategy.

   A close examination of *why* these requirements are changing is an important risk assessment step in determining if agile process will be successful. Late detail binding and separation of concerns can support changing requirements; however, decisions must still be made to identify the areas that require flexibility to deal with changing requirements.

The agile philosophy encourages incremental delivery, but to be effective, increments must be planned.

3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.* Agility helps here, but in general, iterative and incremental methods exhibit this as a behavior, including *spiral* methods—without the necessary relabeling of *agile.* The concept of rapid prototyping is standard practice in many manufacturing and engineering processes. The granularity of the deliverables is the issue here. The question is—what is the *appropriate* absorption rate of the software iteration for a specific domain?

4. *Business people and developers must work together daily throughout the project.* This is common business practice in successful organizations. The definition of the *customer* is restrictive in many of the *agile* process methods, especially when building products rather than projects. The granularity of the interaction is the issue. If the *customer* is co-located, direct daily interaction is possible. If not, then some other form of communication is necessary. Documentation then plays a more significant role.

---

2 The remainder of this section is reprinted from http://www.niwotridge.com/BookReview/AAPrinciples.htm with the permission of Glen Alleman of Niwot Ridge Consulting.

5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.* This is common business practice in successful organizations—it's the people that make a project successful. From Jack Welch [ex-CEO of General Electric Co.] down to the local coffee shop, all business managers understand this principle. Practicing this principle, however, is much more difficult.

6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.* Although this is the basis of agile processes, this is neither unique nor many times practical and in many cases may not even be desirable. Written specifications are useful in many instances, and in others instances they are imposed by contract, geography, regulatory, or safety requirements. This principle is a tautology, but provides no suggestions for alternatives.

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

7. *Working software is the primary measure of progress.* Although working software is an *outcome* of development, there are other critical deliverables and measures of progress as well that are not addressed by many of the agile processes. The focus on software alone misses many opportunities for process improvement prior to and after the generation of code.

8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.* Although a goal, the agile process has little to say on how to achieve this in practice for a specific environment. As well, the statement on sustainability is a conjecture not yet supported by field evidence.

9. *Continuous attention to technical excellence and good design enhances agility.* This is the basis of many good engineering practices. The metrics of technical excellence and good design are not stated, leaving them open to interpretation.

10. *Simplicity—the art of maximizing the amount of work not done—is essential.* Without a context, the term *simplicity* has no meaning. What is simple in one domain may appear complex when viewed from another. This principle fails to address nonfunctional and extra-functional requirements of product and project-based processes which are the sources of much of the complexity in large scale systems.

11. *The best architectures, requirements, and designs emerge from self-organizing teams.* This is conjecture and is not based on analytical measurements. This principle does not state the domains in which it is applicable. The science of *systems engineering* has much to say here, but no recognition to this previous work is provided.

12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.* This is good team development

practice independent of the software environment. No metrics are provided by which to assess past behavior or adjust future behavior.

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 agile process models presented in this chapter.

> **"There is no substitute for rapid feedback, both on the development process and on the product itself."**
> **Martin Fowler**

4.2.2 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 we build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

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

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 4.3), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from conventional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

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

4.2.3 Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors" in successful agile development. As Cockburn and

Highsmith [COC01] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that the *process molds to the needs of the people and team,* not the other way around.[3]

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

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

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

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

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

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

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

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

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

ing those that solve the wrong problem) may be of benefit to the team later in the project.

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

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

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

4.3 AGILE PROCESS MODELS

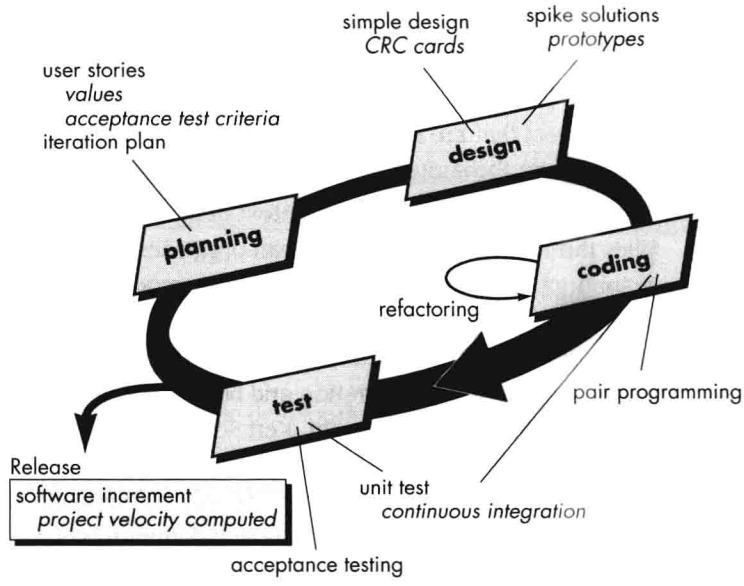
The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process models—each contending for acceptance within the software development community—the agile movement is following the same historical path.[4]

> **"Our profession goes through methodologies like a 14-year-old goes through clothing."**
> **Stephen Hawrysh and Jim Ruprecht**

In the sections that follow, we present an overview of a number of different *agile process models*. There are many similarities (in philosophy and practice) among these approaches. Our intent will be to emphasize those characteristics of each method that make it unique. It is important to note that *all* agile models conform (to a greater or lesser degree) to the *Manifesto for Agile Software Development* and the principles noted in Section 4.2.1.

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

FIGURE 4.1
The Extreme Programming Process



4.3.1 Extreme Programming (XP)

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

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

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

What is an XP "story"?

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

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

the customer is asked to split the story into smaller stories, and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

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

WebRef

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

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

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

> **"Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage."**
>
> **Ron Jeffries**

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]

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[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 (Section 8.7.4). The CRC cards are the only design work product produced as part of the XP process.

---

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 detail in Chapter 8 and throughout Part 2 of this book.

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

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

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

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

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

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

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

Coding. XP recommends that after stories are developed and preliminary design work is done, the team should not move to code, but rather develop a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).[8] Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the unit test. Nothing extraneous it added (KIS). Once the code is complete, it can be unit tested immediately, thereby providing instantaneous feedback to the developers.

What is pair programming?

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

---

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.

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

Testing. We have already noted that the creation of a unit test[9] before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 13) whenever code is modified (which is often, given the XP refactoring philosophy).

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

XP acceptance tests are derived from user stories.

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

SAFE HOME

Considering Agile Software Development

The scene: Doug Miller's office.

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

The conversation:

(A knock on the door)

Jamie: Doug, you got a minute?

Doug: Sure Jamie, what's up?

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

Doug: And?

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

Doug: Yeah, some good, some bad.

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

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

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

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

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

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

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

Doug: So you guys think we should use XP?

Jamie: It's definitely worth considering.

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

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

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

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

Doug: So you agree with the XP approach?

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

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

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

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

4.3.2 Adaptive Software Development (ASD)

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

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

WebRef

Useful resources for ASD can be found at **www.adaptivesd.com.**

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

Speculation. During *speculation,* the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software