Agile and Iterative Development: A Manager's Guide

Table of Contents

1. Introduction.

Software Is New Product Development.
What's Next?
Web Resources.

2. Iterative & Evolutionary.
Iterative Development. Risk-Driven and Client-Driven Iterative Planning. Timeboxed Iterative Development. During the Iteration, No Changes from External Stakeholders. Evolutionary and Adaptive Development. Evolutionary Requirements Analysis. Early Top Ten High-Level Requirements and Skillful Analysis. Evolutionary and Adaptive Planning. Incremental Delivery. Evolutionary Delivery. The Most Common Mistake? Specific Iterative & Evolutionary Methods. What's Next? Recommended Readings.

3. Agile.
Agile Development. Classification of Methods. The Agile Manifesto and Principles. Agile Project Management. Embrace Communication and Feedback. Programming as If People Mattered. Simple Practices and Project Tools. Empirical vs Defined & Prescriptive Process. Principle-Based versus Rule-Based. Sustainable Discipline: The Human Touch. Team as a Complex Adaptive System. Agile Hype? Specific Agile Methods. What's Next? Recommended Readings.

4. Story.
What's Next?

5. Motivation.
The Facts of Change on Software Projects. Key Motivations for Iterative Development. Meeting the Requirements Challenge Iteratively. Problems with the Waterfall. What's Next?

6. Evidence.
Summary. Research Evidence. Early Historical Project Evidence. Standards-Body Evidence. Expert and Thought Leader Evidence. A Business Case for Iterative Development. The Historical Accident of Waterfall Validity? What's Next? Recommended Readings.

7. Scrum.
Method Overview. Lifecycle. Workproducts, Roles, and Practices. Values. Common Mistakes and Misunderstandings. Sample Projects. Process Mixtures. Adoption Strategies. Fact versus Fantasy. Strengths versus Other. History. What's Next? Recommended Readings.

8. Extreme Programming.
Method Overview. Lifecycle. Workproducts, Roles, and Practices. Values. Common Mistakes and Misunderstandings. Sample Projects. Process Mixtures. Adoption Strategies. Fact versus Fantasy. Strengths versus Other. History. What's Next? Recommended Readings.

9. Unified Process.
Method Overview. Lifecycle. Workproducts, Roles, and Practices. Values. Common Mistakes and Misunderstandings. Sample Projects. Process Mixtures. Adoption Strategies. Fact versus Fantasy. Strengths versus Other. History. What's Next? Recommended Readings.

10. Evo.
Method Overview. Lifecycle. Workproducts, Roles, and Practices. Values. Common Mistakes and Misunderstandings. Sample Projects. Process Mixtures. Adoption Strategies. Fact versus Fantasy. Strengths versus Other. History. What's Next? Recommended Readings.

11. Practice Tips.
Project Management. Environment. Requirements. Test.

12. Frequently Asked Questions.
Question List. Questions and Answers.

13. Bibliography.

Top

Back Cover
Agile/iterative methods: From business case to successful implementation

This is the definitive guide for *managers* and *students* to agile and iterative development methods: what they are, how they work, how to implement them*and why you should.*

Using statistically significant research and large-scale case studies, noted methods expert Craig Larman presents the most convincing case ever made for iterative development. Larman offers a concise, information-packed summary of the key ideas that drive all agile and iterative processes, with the details of four noteworthy iterative methods: Scrum, XP, RUP, and Evo. Coverage includes:

- Compelling evidence that iterative methods reduce project risk
- Frequently asked questions
- Agile and iterative values and practices
- Dozens of useful iterative and agile practice tips
- New management skills for agile/iterative project leaders
- Key practices of Scrum, XP, RUP, and Evo

Whether you're an IT executive, project manager, student of software engineering, or developer, Craig Larman will help you understand the promise of agile/iterative development, sell it throughout your organizationaeand transform the promise into *reality*.

Top

Author
CRAIG LARMAN is known throughout the international software community as an expert and passionate advocate for object-oriented technologies and development, and iterative and agile development methods. He serves as Chief Scientist at Valtech, a global consulting and skills transfer company, where he has led the adoption of iterative and agile methods. Larman also authored *Applying UML and Patterns*, the world's best-selling text on object-oriented analysis and design, and iterative development.

Chapter 1. Introduction

Logic is the art of going wrong with confidence.

—Joseph Wood Krutch

OVERVIEW

- What's in this book?
- Predictable versus new product development.

What value will you get from studying this book, an introduction to iterative and agile methods?

First, you will know the key practices of four noteworthy methods, **Scrum**, **Extreme Programming** (**XP**), the **Unified Process** (**UP**), and **Evo** (one of the original iterative methods). This is a "Cliffs Notes" summary, each chapter has something useful to you as a manager, developer, or student of development methods.

Scrum

XP

UP

Evo

Second, your learning curve will be shortened, as this is a distilled learning aid. The four method chapters have the same structure, to speed comprehension and compare-contrast. There's a FAQ chapter, a "tips" chapter of common practices, and plenty of margin pointers to related pages—paper hyperlinks.

FAQ

tips

Third, you will know motivation and evidence. Some organizations accept the value of iterative development, but others are still reluctant. If you need to make a case for an iterative project experiment, you will find in this book the key reasons, research, examples of large projects, standards-body acceptance, a business case, and promotion by well-known thought leaders through the decades. The research and history sections are also of value to students of software engineering methods.

motivation

evidence

Note that agile methods are a subset of iterative methods; this book covers both types.

The chapters may be read in any order; the big picture is this:

1. Introduction, and predictable vs. inventive development.

2. Basic iterative and evolutionary method practices.

3. Summary of agile principles and methods.

4. An agile project story to pull some ideas together.

5–6. Motivation and evidence chapters for iterative and agile methods; useful for some.

7–10. Four method summaries on Scrum, XP, UP, and Evo. Note: practices can be mixed.

11. A tips chapter that expands on some of the method practices, plus others.

12. A frequently asked questions (FAQ) chapter.

Finally, *people trump process*. Every process book should probably include this standard disclaimer:

Process is only a second-order effect.[1] *The unique people, their feelings, qualities, and communication are more influential.*

Some problems are just hard, some people are just difficult. These methods are not salvation.

[1] A quote from the agile methodologist Alistair Cockburn.

Software Is New Product Development

Consider building mobile phones on an assembly line: It *is* possible to unambiguously define the specifications and construction steps. After building some phones and measuring things, it is possible to reliably estimate and schedule the building of future phones.

A different problem: Build a custom house. The owner wants to use new environmentally friendly materials and methods, but isn't exactly sure what they want, and is going to change or clarify their decisions as they see the house, costs, and weeks unfold.

At one end of the spectrum, such as manufacturing phones, there are problems with low degrees of novelty or change, and high rates of repeated identical or near-identical creation—*mass manufacturing* or *predictable manufacturing*.

At the other end, there are problems with high degrees of novelty, creativity, and change, and no previous identical cases from which to derive estimates or schedules. This is the realm of *new product development* or *inventive projects*.

The development process, management values, planning and estimation models appropriately associated with these two domains are different (Table 1.1).

Table 1.1. predictable vs. inventive projects

Predictable Manufacturing	New Product Development
It is possible to first complete specifications, and then build.	Rarely possible to create up-front unchanging and detailed specs.
Near the start, one can reliably estimate effort and cost.	Near the beginning, it is not possible. As empirical data emerge, it becomes increasingly possible to plan and estimate.
It is possible to identify, define, schedule, and order all the detailed activities.	Near the beginning, it is not possible. Adaptive steps driven by build–feedback cycles are required.
Adaptation to unpredictable change is not the norm, and change-rates are relatively low.	Creative adaptation to unpredictable change is the norm. Change rates are high.

Of course, the point is,

Most software is not a predictable or mass manufacturing problem. Software development is new product development.

Plus, many projects use new and buggy technologies that exacerbate the degree of novelty and unpredictability. Note also it is a new product for the inexperienced even if it has been done before.

Since predictable manufacturing is the wrong paradigm for software, practices and values rooted in it are not helpful.

This mismatch lies at the heart of many of the challenges associated with traditional approaches to running a software project.

A "waterfall" lifecycle, big up-front specifications, estimates, and speculative plans applicable to predictable manufacturing have been misapplied to software projects, a domain of inventive, high-change, high-novelty work.

Factors [CP86] preventing reliable up-front specifications include:

- The clients or users are not sure what they want.
- They have difficulty stating all they want and know.
- Many details of what they want will only be revealed during development.
- The details are overwhelmingly complex for people.
- As they see the product develop, they change their minds.
- External forces (such as a competitor's product or service) lead to changes or enhancements in requests.

This deep appreciation—that building software is complex, new product development with high change rates, and not predictable manufacturing—is at the heart of the motivation for agile and iterative methods.

Certainly, another driving force is the desire to compete and win. Iterative and agile methods foster flexibility and maneuverability—a competitive advantage. In *Agile Competitors and Virtual Organizations* [GNP97] the authors examine the limitations of the mass manufacturing model and the need for agility:

Agility […] is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very center of the competitive storms many companies now fear.

What's Next?

The next two chapters summarize basic practices and ideas of iterative, evolutionary, and agile methods. After that, a story chapter illustrates these practices with a concrete scenario.

Web Resources

Related book or journal article suggestions are given in their respective chapters. Web resource suggestions include:

Broad Link or Article Sites

- **www.agilealliance.com** — Collects many articles specifically related to agile methods, plus links.
- **www.cetus-links.org** — The Cetus Links site has specialized for years in object technology (OT). Under "OO Project Management—OOA/D Methods" it has many links to iterative and agile methods, even though they are not directly related to OT.
- **www.bradapp.net** — Brad Appleton maintains a large collection of links on software engineering, including iterative methods.

- **www.iturls.com** — The Chinese front page links to an English version, with a search engine referencing iterative and agile articles.

More Specific Sites

- **c2.com/cgi/wiki?FindPage** — This important, vast Wiki site was the home ground where many of the agile leaders (and design pattern leaders) held their original discussions on XP and other agile methods.
- **www.extremeprogramming.org** — Don Wells' (an early XP leader) introduction to XP.
- **www.xprogramming.com** — Ron Jeffries' (an early XP leader) introduction to XP.
- **www.agilemodeling.com** — Scott Ambler's site contains many articles related to agile modeling practices.
- **sunset.usc.edu** — Associated with the work of Dr. Barry Boehm, a long-time researcher into iterative (e.g., Spiral) methods. Articles related to iterative methods.
- **www.cutter.com** — Cutter's site has an Agile Project Management specialty area.
- **www.martinfowler.com** — Martin Fowler is an early agile methods thought leader (XP method). Articles and links.
- **www.jimhighsmith.com** — Jim Highsmith is an early agile methods thought leader (Adaptive Software Development method). Articles and links.
- **alistair.cockburn.us** — Alistair Cockburn is an early agile methods thought leader (Crystal methods). Articles and links.
- **www.controlchaos.com** — Ken Schwaber is an early agile methods thought leader (Scrum method). Articles and links.
- **jeffsutherland.com** — Jeff Sutherland is an early agile methods thought leader (Scrum method). Articles and links.
- **www.gilb.com** — Tom Gilb is one of the very earliest iterative and evolutionary thought leaders (Evo method). Articles and links.
- **www.craiglarman.com** — My site. Articles and links.
- **www.objectmentor.com** — Company led by Robert C. Martin, an early agile thought leader (XP related). Articles and links.
- **www.nebulon.com** — Company led by Jeff De Luca, an early agile thought leader (Feature-Driven Development method). Articles and links.
- **www.dsdm.org** — Official site for the DSDM method.
- **www.rational.com** — Official site for the Rational Unified Process (RUP) iterative method.
- **name.case.unibz.it** — Network for Agile Methodologies Experience (NAME). A European site that describes research into agile methods, and with links to other sites.

Chapter 2. Iterative & Evolutionary

Experience is that marvelous thing that enables you to recognize a mistake when you make it again.

—F. P. Jones

OVERVIEW

- Basic practices of iterative and evolutionary methods, including timeboxing and adaptive planning.
- A common mistake adopting iterative methods.
- Specific iterative and evolutionary methods, including Evo and UP.

Iterative and evolutionary development is a foundation not only of modern software methods, but—as the history section of the "Evidence" chapter shows—of methods used as far back as the 1960s. Agile methods are a subset of iterative and evolutionary methods. This chapter summarizes key practices:

history

iterative development

evolutionary development

risk-driven and client-driven

evolutionary requirements

timeboxing

adaptive planning

Iterative Development

Iterative development is an approach to building software (or anything) in which the overall lifecycle is composed of several iterations in sequence. Each **iteration** is a self-contained mini-project composed of activities such as requirements analysis, design, programming, and test. The goal for the end of an iteration is an **iteration release**, a stable, integrated and tested *partially* complete system. To be clear: *All* the software across all the teams is integrated into a release each iteration. Most iteration releases are *internal*, a baseline primarily for the benefit of the development team—they are not released externally. The final iteration release is the complete product, released to the market or clients. See Figure 2.1.

Figure 2.1. iterative and incremental development

iterative planning tips

Although an iteration can in theory be only for clean-up or performance tuning, usually the partial system grows incrementally with new features, iteration by iteration; in other words, **incremental development**. The concept of growing a system via iterations has been called **iterative and incremental development** (**IID**), although simply "iterative development" is common. Some older process literature [Wong84] used the term "incremental development" to mean a combination of frozen up-front specifications followed by iterative development of the features, but there is no widespread agreement on usage. In this era, most development methods are IID methods. And, IID is at the core of all the agile methods, including Scrum and XP.

Most projects have at least three iterations before a final public release; I've seen a two-year Valtech project composed of close to 20 iterations averaging around four weeks each, and I know of at least one long project with 45 iterations.

In modern iterative methods, the recommended length of one iteration is between one and six weeks.

Each iteration includes production-quality programming, not just requirements analysis, for example. And the software resulting from each iteration is not a prototype or proof of concept, but a subset of the final system.

More broadly, viewing an iteration as a self-contained mini-project, activities in many disciplines (requirements analysis, testing, and so on) occur within an iteration (see Figure 2.2).

Figure 2.2. disciplines across iterations

Risk-Driven and Client-Driven Iterative Planning

What to do in the next three-week iteration? IID methods promote a combination of risk-driven and client-driven[1] priorities. **Risk-driven iterative development** chooses the riskiest, most difficult elements for the early iterations. For example, maybe the client says "I want the Web pages to be green and the system to handle 5,000 simultaneous transactions." Green can wait. In this way, the highest risks are surfaced and mitigated early rather than late. Risk is a broad concept—maybe you are making a new 3D modeling tool and market research shows that what will capture market interest is a novel, much easier user interface metaphor. The high risk is not getting the UI right.

[1] Throughout this book, *client* or *customer* could mean a proxy, such as a marketing or product manager for a consumer software product, true end-users for an internal application, etc.

risk-driven

ranking risks

first iteration

use cases and iteration planning

Client-driven iterative development implies that the choice of features for the next iteration comes from the client—whatever they perceive as the highest business value to them. In this way, the client steers the project, iteration by iteration, requesting the features that they *currently* think are most valuable. Note that the customer **adaptively plans** the choice for the next iteration, shortly before it starts, based on their latest insight, rather than speculatively at the start of the project. The customer has ongoing control and choice, as fresh information arises.

adaptive and client-driven planning

Apply both schemes. Clients do not always appreciate what is technically hard or risky. Developers do not always appreciate what has high business value.

mixing and ranking iteration goals

Timeboxed Iterative Development

Iteration **timeboxing** is the practice of fixing the iteration end date and not allowing it to change. An overall project may be timeboxed as well. If it eventually appears that the chosen requests (the scope) for the iteration can't be met within the timebox, then rather than slip the iteration end date, the scope is reduced (placing lower priority requests back on the wish-list), so that the partial, growing system always ends in a stable and tested state on the original planned iteration end date. See Figure 2.3.

Figure 2.3. timeboxing

multi-site timeboxed iterations

overlapping activities across timeboxes

It is important that timeboxing is not used to pressure developers to work longer hours to meet the soon-coming deadline. If the normal pace of work is insufficient, do less.

In most IID methods, not all timebox lengths need be equal. The first iteration may be four weeks, the second iteration three weeks, and so forth. On the other hand, the Scrum method recommends that each timebox be exactly 30 calendar days. As mentioned, most IID methods recommend an iteration timebox between one and six weeks.

iteration length

what day to end a timebox?

A three-month or six-month timeboxed "iteration" is extraordinarily long and usually misses the point and value; research shows that shorter steps have lower complexity and risk, better feedback, and higher productivity and success rates. That said, there are extreme cases of projects with hundreds of developers where a three-month iteration is useful because of the overhead.

All the modern IID methods (including Scrum, XP, and so forth) either require or strongly advise timeboxing the iterations.

During the Iteration, No Changes from External Stakeholders

Iterative and agile methods embrace change, but not chaos. In a sea of constant change, a point of stability is necessary. In IID methods this is achieved with the rule:

Once the requests for an iteration have been chosen and it is underway, no external stakeholders may change the work.

One week into a three-week iteration, the product manager should not come along, and ask, "Can you do this too?" They wait for the next iteration. However, the team itself can reduce the scope of an iteration if the timebox deadline cannot otherwise be met.

scope reduction: primary and secondary iteration goals

Evolutionary and Adaptive Development

Evolutionary iterative development implies that the requirements, plan, estimates, and solution evolve or are refined over the course of the iterations, rather than fully defined and "frozen" in a major up-front specification effort before the development iterations begin. Evolutionary methods are consistent with the pattern of unpredictable discovery and change in new product development.

evolutionary requirements

adaptive planning

Adaptive development is a related term. It implies that elements adapt in response to feedback from prior work—feedback from users, tests, developers, and so on. The intent is the same as evolutionary development, but the name suggests more strongly the feedback-response mechanism in evolution.

Some methods or methodologists emphasize the term "iterative" while others use "evolutionary" or "adaptive." The ideas and intent are similar, although strictly speaking, evolutionary and adaptive development does not require the use of timeboxed iterations.

Evolutionary Requirements Analysis

In evolutionary and adaptive development, it is *not* the case that the requirements are forever unbounded or always changing at a high rate. Rather, most requirements discovery and refinement usually occurs during early iterations, and the earliest attention is given to understanding the most architecturally significant or high-business-value requirements. For example, on an ultimately 20-iteration project, it is likely that most requirements will be discovered and refined within the first three or four iterations (that include, in parallel, early software development).

evolutionary requirements tips

In each iteration, there is a one- or two-day requirements workshop in which the specifications expand and refine, in response to further analysis and feedback from the system under development. See Figure 2.4. For example, the first workshop focuses on detailed analysis of 20% of the most architecturally significant and risky requirements; this gives the software architect enough meaningful input to start development and test in short cycles.

Figure 2.4. evolutionary and iterative requirements

Imagine this will ultimately be a 20-iteration project.

In evolutionary iterative development, the requirements evolve over a set of the early iterations, through a series of requirements workshops (for example). Perhaps after four iterations and workshops, 90% of the requirements are defined and refined. Nevertheless, only 10% of the software is built.

workshops

Note as a design comment, that it is not true that 100% of the *functional* requirements need be known to start building an excellent core architecture. The architect needs to know most nonfunctional or quality requirements (e.g., load, internationalization) and a much smaller representative subset of functional requirements.

Early "Top Ten" High-Level Requirements and Skillful Analysis

It is a misunderstanding to equate evolutionary requirements analysis with "no early requirements" or sloppy requirements practices. Modern IID methods encourage the early creation and baselining of vision statements, "top ten" *high-level* requirements lists, and early analysis of architecturally influential factors, such as load, usability, and internationalization. Further, these methods encourage many skillful analysis techniques during early iterations, such as a series of requirements workshops involving both target users and developers, writing use cases, and much more.

vision boxes

product sheets

use cases

various elicitation methods

Evolutionary and Adaptive Planning

As with evolutionary requirements, with evolutionary and adaptive planning it is not the case that estimates and schedules are forever unbounded or unknown. Yet, due to early requirements change and other factors, there is an initial phase of high uncertainty, which drops as time passes and information accumulates. This has been called the **cone of uncertainty** (Figure 2.5) [McConnell98].

Figure 2.5. cone of uncertainty

adaptive planning and related tips

The iterative response to this uncertainty is to defer an expectation of semi-reliable estimates for cost, effort or schedule until a few iterations have passed. Perhaps 10% to 20% into a project.

This is consistent with management practice in other new product development domains, where an initial exploratory phase is common. Further, the practice of **adaptive planning** is encouraged rather than **predictive planning**. That is, a detailed schedule is not created beyond a relatively short time horizon, so that the level of detail and commitment is commensurate with the quality of information.

adaptive and predictive planning

Fixed-Price Contracts

With respect to fixed-price bidding and evolutionary estimates, some IID methods (such as the UP) recommend running projects in two contract phases, each of multiple timeboxed iterations.

The first phase, a relatively short fixed-time and fixed-price contract, has the goal of completing a few iterations, doing early but partial software development and evolutionary requirements analysis. Note the key point that partial software is produced, not merely documents.

The outputs of phase one—including the software base—are then shared with bidders for a phase two fixed-price contract. The evolutionary refinement of specifications and code in phase one provides higher quality data for phase two estimators, and advances the software for the project (Figure 2.6).

Figure 2.6. two contract phases

Incremental Delivery

Incremental delivery is the practice of repeatedly delivering a system into production (or the marketplace) in a series of expanding capabilities (Figure 2.7). The practice is promoted by IID and agile methods. Incremental deliveries are often between three and twelve months.

Figure 2.7. incremental delivery with iterations

Incremental delivery is often confused with iterative development. A six-month delivery cycle could be composed of 10 short iterations. The results of each iteration are not delivered to the marketplace, but the results of an incremental delivery are.

Evolutionary Delivery

Evolutionary delivery is a refinement of the practice of incremental delivery in which there is a vigorous attempt to capture feedback regarding the installed product, and use this to guide the next delivery. Naturally, the evolutionary goal is to best meet some difficult-to-predict need, such as the most frequently requested new features. Uniquely, the Evo method promotes—when possible—very short evolutionary delivery cycles of one or two weeks, so that each iteration delivers something useful to stakeholders.

The Evo method and evolutionary delivery

To contrast "pure" incremental delivery with evolutionary delivery, in the former a plan is defined of several future deliveries—feedback is not driving the delivery plan. In evolutionary delivery, there is no plan (or at least no fixed plan) of future deliveries; each is dynamically created based on emerging information. In practice, a marriage of some future prediction and feedback is obvious and common, and the two terms are used interchangeably.

The Most Common Mistake?

Iterative and agile process coaches often see scenarios like this:

Jill: Sure, we don't apply the waterfall—everyone knows it doesn't work. We've adopted <iterative method X> and are into our first project. We've been at it for two months and have the use case analysis nearly finished, and the plan and schedule of what we'll be doing in each iteration. After review and approval of the final requirements set and iteration schedule, we'll start programming.

This profound misunderstanding, still superimposing waterfall-inspired, big up-front analysis and planning (predictable manufacturing) values onto iterative methods, is one of the most common mistakes that new iterative and agile method adopters make.

Specific Iterative & Evolutionary Methods

Specific *agile* methods are summarized in the next chapter. This section mentions some *iterative* methods (Evo and UP) that predate most agile methods; they may or may not be considered agile.

Of all the methods mentioned in this book (Scrum, XP, Evo, UP, OPEN, DSDM, ...) the UP or its variation the Rational Unified Process (RUP) is perhaps the most widely used. It is found in thousands or tens of thousands of development organizations worldwide. This does not mean it is well applied or well understood.

Evo

Evo was perhaps the first iterative and evolutionary method, starting in the 1960s. Evo recommends short 1–2 week iterations, and uniquely, evolutionary *delivery* each iteration. Evo adaptively plans iterations by highest value-to-cost ratio, and strongly promotes the unambiguous definition of quality requirements (such as load) with quantified and measurable statements.

Evo details

Unified Process

The UP or RUP, first developed in the mid-1990s, brings together the knowledge of many experienced large-system architects and process-leaders at Rational Corp., and their customers, into a well-defined IID method. One key UP theme is risk-driven development in the early iterations, focusing on creation of the core architecture and driving down the high risks. The UP also includes the definition of common project workproducts, such as the *Vision, Software Architecture Document*, and *Risk List*.

UP details

Other Methods

In addition to UP and Evo, other IID methods include:

- The **Microsoft Solutions Framework** process, available from Microsoft Education. It is a description of best practices used by Microsoft.
- The **OPEN** process from Henderson-Sellers, Firesmith, and Graham [FH01].
- **WinWin Spiral Model** and **MBASE Spiral Model** from Barry Boehm (creator in the 1980s of the well-known iterative Spiral Model) and colleagues [BEKPSM98], [BP01].

What's Next?

The next chapter summarizes agile method practices and values. After that, a story chapter illustrates these practices with a concrete scenario.

Recommended Readings

- *Rapid Development*, by Steve McConnell. Examines variations of iterative development, citing plenty of research data.
- *The Mythical Man-Month*, by Frederick Brooks. The silver-anniversary edition of this classic discusses the advantages of IID, in addition to many timeless lessons.

Chapter 3. Agile

Health is merely the slowest possible rate at which one can die.

—anonymous

OVERVIEW

- Basic ideas and principles of agile methods.
- Classification of methods.
- Agile hype?
- Specific agile methods, including Scrum and XP.

Agile Development

Agile development methods apply timeboxed iterative and evolutionary development, adaptive planning, promote evolutionary delivery, and include other values and practices that encourage *agility*—rapid and flexible response to change. If agile methods have a motto, it is *embrace change*.[1] If agile methods have a strategic point, it is *maneuverability*.

[1] The subtitle of the first book on XP, *Extreme Programming Explained: Embrace Change*, by Kent Beck.

It is not possible to exactly define **agile methods**, as specific practices vary. However, short timeboxed iterations with adaptive, evolutionary refinement of plans and goals is a basic practice various methods share. Thus, the book title and opening material includes *iterative development*, which lies at the heart of agile methods.

iterative development

In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-directed teams, programming over documenting, and more.

agile principles

Example practices from Scrum include working in a *common project room* and *self-directed teams* that coordinate through a daily stand-up meeting with special questions each member answers. Example practices from XP include using terse notes on paper *story cards* to summarize requirements, *programming in pairs*, and working in a common project room with *full-time participation by requirement donors* so that detailed written requirements can be replaced with ongoing verbal explanations.

Scrum practices

XP practices

As a distinct software process concept, agile is newer than iterative. Most older IID methods (such as Evo and UP) were not strongly agile in their original definition, although many methods can be applied in an agile spirit with the right understanding.

Although it is possible to imagine truly un-agile IID methods, as a practical matter they are rare or are quickly adopting agile values and practices—it is hard to find someone promoting *un-agility*!

Classification of Methods

Having raised the issue of variations in methods, one classification is their degree of **ceremony**—the amount of **method weight** in terms of documentation, formal steps, review, and so forth. Another classification is their **cycles**—the number and length of iterations. For example, single-pass waterfall has no iterations; at the other end, an Evo project could have very many—one iteration per week. Figure 3.1 illustrates four methods categorized by this scheme.

Figure 3.1. methods by ceremony and cycles

other classification schemes

As hinted in Figure 3.1, XP recommends 1–4 weeks, and the UP 2–6 weeks. Some would say the lower left quadrant (many short iterations, low ceremony) describes the agile methods, but this is not quite accurate. For example, Scrum is silent on the question of ceremony—or more precisely leaves the question to the discretion of the team. A better way to describe agile methods in terms of ceremony is their promotion of *barely sufficient* ceremony. That could be high in some case, such as for a USA Food and Drug Administration (FDA) regulated device.

The Agile Manifesto and Principles

In 2001 a group interested in iterative and agile methods (coining the term) met to find common ground. Out of this came the Agile Alliance (www.agilealliance.com) with a manifesto and statement of principles. Worth study! Most of the many concrete practices described throughout this book reflect these principles. Agile project management is guided by these principles.

The Agile Manifesto

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.

The Agile Principles

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 time scale.

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.

9. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

10. Continuous attention to technical excellence and good design enhances agility.

11. Simplicity—the art of maximizing the amount of work not done—is essential.

12. The best architectures, requirements, and designs emerge from self-organizing teams.

13. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Project Management

The Scrum and XP chapters include specific **agile project management** practices for those methods. Other authors present generalizations with common themes: manager promotes the vision, more communication, avoid command-control, and so on. This section summarizes two well-known descriptions.

Scrum practices

XP practices

Jim Highsmith, an Agile Alliance founder and creator of the Adaptive Software Development method, summarizes nine principles for the agile project manager [Highsmith02]:

1. Deliver something useful to the client; check what they value.

2. Cultivate committed stakeholders.

3. Employ a leadership-collaboration style.

4. Build competent, collaborative teams.

5. Enable team decision making.

6. Use short timeboxed iterations to quickly deliver features.

7. Encourage adaptability.

8. Champion technical excellence.

9. Focus on delivery activities, not process-compliance activities.

Augustine andWoodcock, two managers with experience in XP-oriented projects, recommend six practices [AW02]:

1. **Guiding Vision –** Establish a guiding vision for the project and continuously reinforce it through words and actions.

2. **Teamwork & Collaboration –** Facilitate collaboration and teamwork through relationships and

4. **Open Information –** Provide visible and open access to project management and other information.

5. **Light Touch –** Apply just enough control to foster emergent behavior in a self-directed

1. **Guiding Vision –** Establish a guiding vision for the project and continuously reinforce it through words and actions.

community.

3. **Simple Rules –** Establish and support the team's set of guiding practices, such as Scrum or XP.

4. **Open Information –** Provide visible and open access to project management and other information.

team.

6. **Agile Vigilance –** Reinforce the vision, follow or adapt the rules, listen to the people.

A theme of agile project management in Scrum and XP is the devolution of both control and planning to the entire team, not the manager. The manager does not create a work breakdown structure, schedule, or estimates; this is done as a team. The manager does not (usually) tell people what to do. The manager does not define and assign many detailed team roles and responsibilities.

Rather, the project manager role emphasizes coaching, servant-leadership, providing resources, maintaining the vision, removing impediments, promoting agile principles, etc. Thus, managers more used to control and rule-based methods or project management have some challenge adopting agile methods.

Embrace Communication and Feedback

If the agile Prime Directive is *embrace change*, a close second is *embrace communication and feedback*. The manifesto and principles, and agile methods have a theme of working to increase communication, especially face-to-face conversation. This is illustrated in practices such as the daily Scrum meeting and XP's requirement to have onsite customers sitting full-time in the common project room.

daily Scrum

Agile methods are adaptive, requiring feedback to guide direction. Beyond just conversation, they seek out early and frequent feedback through early testing, early demos, and much more.

Programming As If People Mattered

People are more important than any process. Good people with a good process will outperform good people with no process every time. — Grady Booch [Booch96]

The first value in the Agile Manifesto is *Individuals and interactions over processes and tools*. It reminds us that programming is a human activity. For example, XP champions the importance of happy developers for sustainable development. Mindful of the impact of overwork on the ability to program well or maintain a healthy social or family life, XP has the rule of *sustainable pace*—avoiding working overtime. Agile Principle 8 states, *Agile processes promote sustainable development*.

Also, research shows that individual developer contributions vary enormously, with studies illustrating 10 times more productive from best to worst [Boehm81]. This does not imply promotion of (unsustainable) heroic individual programming, but the right knowledge and work habits play a significant productivity role—thus the value of ongoing education and mentoring for developers. XP encourages deep skills transfer through the practice of *pair programming*.

pair programming

The primacy of people and interactions over process is also promoted in agile methods by their emphasis on communication, especially face-to-face conversations. Scrum's *daily meeting* and *common project room,* and XP's *pair programming* and *whole team together* are examples.

daily Scrum

Simple Practices and Project Tools

Agile Principle 11 is *Simplicity is essential*. Most agile methods embrace *do the simplest thing that could possibly work*—an XP aphorism. This applies not only to software design, but to all project practices. Can requirements and tasks be tracked with paper cards in different piles (not done, done)? And so forth.

Most agile methods promote a "low-tech, high-touch" approach to project and management tools. Of course, low-tech is relative: If the simplest thing that could possibly work is a Web tool, then so be it.

> It is a misunderstanding to equate agile methods with sloppiness. Simplicity and lightness does not imply lack of skill or self-discipline. A project applying all the XP practices has plenty of structure and discipline. But—and this is perhaps a key point in agile methods—the "disciplined" practices are very delivery-oriented or quality-code-oriented. Developers quickly see benefit.

Empirical vs. Defined & Prescriptive Process

In general, agile methods promote empirical rather than defined processes, a categorization used by industrial process experts [OR94]. A **defined process** (also known as a **prescriptive process**) has many predefined and ordered activities to be followed during development. Defined processes are suitable for predictable manufacturing domains. **Empirical processes** are used for high-change and unstable domains; rather than many sequenced activities, they are based on frequent measurement and dynamic response to variable events. This approach is reflected in Agile Principles 12 and 13.

> *agile principles*

For example, Scrum is silent on the activities of an iteration, other than the daily Scrum meeting as the measurement and response mechanism. The UP, on the other hand, strikes a middle way; it lists common activities (e.g., *Write Release Notes*), but the team is welcome to ignore or do them in any order.

Agile methodologists understand that the degree of "method weight" and predefinition of ordered activities are functions of the project type. An agile method or project lies on a continuum of more or less empirical, driven by need. A medical device under FDA approval requires more formal, predefined activities.

Principle-Based versus Rule-Based

Similar to empirical versus prescriptive methods, a related way to describe agile methods is as more principle-based than rule-based. Rather than a predefined set of rules regarding the many roles, team organization, responsibilities, relationships, and activities, the team and manager are primarily guided by the principles embodied in the Agile Manifesto and Principles. Agile project management is more than a set of practices—it is a mindset.

Sustainable Discipline: The Human Touch

There is no shortage of stories on attempts to adopt methods requiring significant discipline and effort, only to find compliance drop off dramatically shortly after its introduction. The psychological and social factors necessary for sustained adoption are missing. The agile methodologist Alistair Cockburn asks people and teams, "Would you do these practices again?" This is arguably a more important question than the inherent value of the practice.

The creators of a few agile methods (e.g., XP and Crystal) recognized that human factors such as enjoyment, simplicity, short-term reward, peer pressure, and lots of gain for the pain are important ingredients to create fertile soil for sustainable self-discipline with practices. Not surprisingly, XP practices rate well in response to Cockburn's question [Cockburn02].

For example, test-driven development reveals its payoff quickly to those who try it; developers enjoy the small win of making a test pass and the design clarification that comes from writing the tests before writing the code to be tested.

test-driven development

Team as a Complex Adaptive System

Some agile methods (including Scrum and Adaptive Software Development) speak of a healthy development team as a **complex adaptive system** (CAS). A classic CAS example is a flock of birds. Each bird has relatively local and simple rules of behavior, yet at the macro-scale the flock exhibits order and a collective **emergent behavior**. It is as though there is an overlaying flock-level plan, but there isn't. This is in contrast to a command-control management system where team and individual activities are decided and directed by higher-level managers.

The agile methods promote the value that, for creative inventive projects, a CAS-inspired culture of self-organizing teams is more valuable than control or planning by managers. This is reflected in Agile Principle 12. For example, Scrum teams are self-organizing (no management assignment of roles or tasks); team-level organization and adaptation is enabled by the daily Scrum meeting with its special questions that provide each member with the information to make collective decisions.

agile principles

Scrum meeting questions

Agile Hype?

"Be adaptable, collaborative, delivery driven, people oriented, customer focused, guided by a vision; develop in short iterations; manager is coach" and most other "agile" messages are hardly new. And iterative development had replaced the waterfall on major projects by the 1970s. So is this just hype—recycling old ideas? Yes and no. Many so-called agile messages are repetition from trends of prior decades.

early iterative projects

But, viewing *as a whole* all the principles and practices of Scrum or XP (for example), these methods have a fresh flavor; they push the envelope of embracing changing requirements, communication, self-organizing teams, adaptive planning, and so forth.

adaptive planning

Plus, some practices—such as test-driven development and continuous integration—are relatively new.

test-driven

continuous integration

Agile Methods Did Not Invent Iterative Development

There is misinformation stating that what is revolutionary about XP and other agile methods is their abandonment of the waterfall model and invention of iterative and evolutionary development. Not true—iterative and evolutionary methods that eschewed the waterfall and applied short iterations with adapting plans and specifications have been around since the 1960s.

early methods

Specific Agile Methods

Scrum and XP are widely applied agile methods—the two most common, according to at least one survey [Shine03]. However, there is anecdotal evidence indicating some "XP" adoption is misunderstood as simply "not the waterfall" and the teams are merely practicing iterative and evolutionary development or "programming without documentation" and calling this XP.

see misunderstanding XP:

Scrum

Scrum's distinctive emphasis among the methods is its strong promotion of self-organizing teams, daily team measurement, and avoidance of following predefined steps. Some key practices include a daily stand-up meeting (the Scrum meeting) with special questions, 30-calendar-day iterations, and a demo to external stakeholders at the end of each iteration.

Scrum details

XP

XP is probably the most well known agile method; it emphasizes collaboration, quick and early software creation, and skillful development practices. It is founded on four values: communication, simplicity, feedback, and courage. It includes 12 core practices, including the whole team working together in a common room, pair programming, constant refactoring, and test-driven development.

XP details

Crystal Methods

The Crystal family of agile methods were developed by Alistair Cockburn [Cockburn02]. While acknowledging the necessity of an iterative lifecycle, in this group of methods Cockburn stresses the primacy of "peopleware" issues over process: communication, education, and so on. His definition of software development shows this emphasis: "…a cooperative game of invention and communication."

Different versions of Crystal (Clear, Yellow, …) contain increasing **method weight** (or **process ceremony** in terms of defined and ordered steps, documents, reviews, etc.) as a function of staff size, criticality, and project priority. You choose size and criticality, and this maps to a particular version of Crystal with a recommended method weight. Cockburn has created a scale to illustrate matching method configurations to these factors (Figure 3.2). For example, E6 means a project of 1–6 people, where the worst that can happen from a system failure is loss of essential money. Subsequent chapters refer to this classification model, to classify Scrum, XP, UP, and Evo.

Figure 3.2. Cockburn scale

Agile Modeling

Agile Modeling is not a complete process or agile method, but a set of principles and practices for modeling and requirements analysis that complement most any IID method [Ambler02]. Scott Ambler summarizes best practices (Figure 3.3) that skilled iterative modelers apply. Briefly, Agile Modeling promotes the collaborative "low-tech, high-touch" creation of barely good enough, disposable models to aid understanding and communication. Practices encourage speed, simplicity, and creative flow.

Figure 3.3. some agile modeling practices

Scenario

The project room walls are exposed (no furniture against them) and covered in whiteboards and static-cling-sheet whiteboard material. It's Monday, at the start of a three-week iteration. The team of eight developers has decided to spend two or three hours at the walls to better understand and communicate some ideas. Afterwards, they will start programming. They split into groups. Group 1 explores the object design for the main scenario. On one half of one wall, they sketch UML sequence diagrams. It isn't perfect UML; liberties are taken for sketching. After 15 minutes, they move to the other half of the wall, and sketch a class diagram that complements the sequence diagram. Over three hours, they move back and forth, developing two complementary diagrams. Finally, they take pictures, print them, and erase the boards. When programming later, the printouts provide some inspiration—some design ideas in code are inspired by the prior thought, and some not.

Other Methods and Practices

In addition to Scrum, XP, and the Crystal methods, there are now a host of other agile-oriented practices or methods in use or publication:

- **Adaptive Software Development (ASD)** from Jim Highsmith [Highsmith00]; inspired by the complex adaptive systems viewpoint, Rapid Application Development methods, and more.
- **Dynamic Solutions Delivery Model (DSDM)** from a group of 16 Rapid Application Development (RAD) experts [Stapleton97] (originally called Dynamic Systems Development Method). It continues to be refined by a member consortium.
- **Feature-Driven Development (FDD)** primarily from Jeff De Luca, with contributions by Peter Coad [PF02].
- **Lean Development** from Mary and Tom Poppendieck [Poppendieck03].
- **Pragmatic Programming** from Andy Hunt and Dave Thomas is not a complete method, but contains development practices sympathetic to an agile development approach. See www.pragmaticprogrammer.com.

What's Next?

The next chapter illustrates iterative and agile practices with a concrete story. After that, two related chapters on motivation and evidence. Then, four related chapters on well-known methods: Scrum, XP, UP, and Evo.

Recommended Readings

- *Peopleware*, by DeMarco and Lister. Discusses critical people-side issues in development, a key concern of agile methods.
- *Agile Software Development*, by Alistair Cockburn. Emphasizes the principles and theory underlying agile methods, with a special focus on communication.
- *Agile Software Development Ecosystems*, by Jim Highsmith. Includes interesting interviews with several agile method founders, in addition to a survey of several methods and key principles.

Chapter 4. Story

I find that the harder I work, the more luck I seem to have.

—Thomas Jefferson

OVERVIEW

- Story of an agile project to relate some practices. This example illustrates a variety of Scrum, XP, UP, and Evo principles and practices.

It is Monday, January 2. The government Border Information Group (BIG) needs a biometrics tracking system for non-citizens entering the country, the Biometric Recording Or Tracking Hazardous External Radicals (BROTHER). BIG has a long list of requests, and Martin, the development manager, has convinced BIG management to use timeboxed iterative development and timeboxed evolutionary delivery. Thus, they will deliver the highest priority features possible by October 1. There is a wish list of features for the first release, and everyone has agreed it may vary, but the release date will not vary.

evolutionary delivery

The plan is that after one or two months in operation at two low-volume airports, feedback from the border guards and travelers will be used to help decide refinements and features for the second release six months later, with a wider rollout.

Following a Scrum practice, Martin and his team of seven developers also commit to demo to the Minister of BIG "running, integrated and tested software every three or four weeks, starting next week."

Scrum demo

The project will use a combination of practices from Scrum, XP, and UP that Martin and the developers have decided make sense for the team. They organize their physical space: Rather than separate offices or cubicles with dividers, they take over a large room at one of the test airports, FooBarKhan International, near the border guards' office area. Four cubicles are set up near the project room that people can use when they need private time. The common project room is a Scrum and XP practice.

XP whole team together

caves and common room

All furniture against the walls is removed. Tables for computers are placed near the center. The walls are covered in giant whiteboards, and whiteboard-like static-cling sheets are used as wallpaper elsewhere. This will support the practice of Agile Modeling.

Agile Modeling

The responsible BIG manager, Domina, moves into the project room, along with Itchy, a long-time border guard. Itchy will be dedicated full-time to the BROTHER project, and Domina agrees to be around "most mornings." This is the XP practice of onsite clients.

onsite customer

On Thursday, January 5, they get together and hold a two-day requirements and planning workshop, a UP practice. In addition to receiving a 20-page wish list from senior BIG management, many "agile requirement analysis" techniques are applied the first morning. After lunch (beer and cake, FooBarKhan's national foods), Martin poses this request: *From the high-level requirements we've generated, choose the 20% most architecturally significant, risky, and valuable items*. They use dot-voting to prioritize the items.

workshop

agile requirements tips

dot voting

They spend the remaining afternoon and Friday morning analyzing the 20% items in detail. For functional requirements, they apply the UP practice of writing a few use cases, in the Cockburn style. For the nonfunctional quality requirements, they apply the Evo practice of clearly quantified and measurable goals; the vague "fast response" and "easy to use" goals in the original BIG wish list were not acceptable.

Evo specifications

On Friday after lunch, Martin moves the team on to planning, even though some of the team want to spend more time detailing and clarifying specifications. Martin poses this challenge to the group: *"We start developing next Monday, January 9. By Thursday, January 26, in 13 working days, we need to have a partial running system hooked up to at least one of the biometric readers. There will be a demo that morning to the BIG minister. What should we realistically do in the next 13 days from the 20% list we explored in detail? And, no overtime."* No overtime or "sustainable pace" is an XP practice.

XP sustainable pace

The group picks a common "happy path" scenario from one of the use cases that will force them to touch on many architectural factors and components (a UP practice), some other features, and spends the afternoon analyzing and estimating the related fine-grained tasks in an XP-style Planning Game. Note that the manager does not create the work breakdown structure, schedule or estimates; the team does this. Eventually, they discover that their first set of goals is too much work, so they scale back some of the features until the estimates match their available Ideal Engineering Hours budget (an XP practice). They wrap the meeting. Martin enters the task items into a Scrum Sprint Backlog spreadsheet.

scenarios and iterations

Planning Game

Ideal Engineering Hours

Sprint Backlog

On Monday, January 9, the first iteration starts.

On Monday morning at 9:30, and all subsequent mornings, they hold a 20-minute daily stand-up Scrum meeting. Martin reminds the team of the overall vision, and the specific goals of the iteration. They are standing beside the whiteboard where all the iteration tasks are written. After the Scrum questions, team members start volunteering for tasks, writing their name beside them.

Scrum meeting

Afterwards, the entire group listens to a presentation by Rebecca, the chief architect. Having a chief architect is recommended in UP. Rebecca spent the prior week investigating and considering architectural issues and designs, given the basic information she had. She lays out her vision of the big pieces and problems, to provide a starting point for decomposition of work by large components. Group discussion refines the ideas. Mid-morning, three subgroups head for the walls, doing Agile Modeling for different subcomponents, reviewing the written use case. The team reserved all of Monday as creative "wall time" to explore and coordinate design ideas during this "fuzzy front end" exploratory phase. Rebecca spends time rotating through all the groups, building a synergy of ideas. Digital snapshots of all the wall notes and UML-ish sketches are taken.

Agile Modeling

On Tuesday, January 10, teamwork starts at 9:30 with the Scrum meeting. New tasks and impediments are written on the adjacent whiteboard. Martin reminds the team of the vision and iteration goals. More tasks are volunteered for. As previously agreed, they will start programming this morning, even though many design points and coordination issues are fuzzy.

Most of the developers decided they didn't want to try pair programming, so that XP practice was bypassed, although Martin encouraged pairing by anyone who wanted to try it. However, they all agree to the XP practices of test-driven development and continuous integration. The team is using an IDE with great refactoring tools, and Martin frequently encourages the team to not just cut code, but keep it clean and simple by regularly applying refactorings, another XP practice. The developers occasionally look at the wall sketches or printouts of the snapshots for some inspiration. Some developers head for the walls for 30 minutes to UML-sketch some design ideas, then back to their development stations.

test-driven development

continuous integration

As the day progresses, questions about the features and happy path scenario arise, and Domina and Itchy talk with the developers to clarify and decide the requirements. These two also work with the developer Girija to create acceptance tests—an XP practice.

acceptance testing

As developers complete their classes, they check them and their unit tests in to the version control server. Girija checks in completed acceptance tests. A separate build machine is running these tests within a continuous integration service 24/7, every 15 minutes. Thus, bugs and integration problems are quickly surfaced and resolved. Continuous integration is an XP practice.

Girija also volunteers to be daily tracker, another XP practice. So, each morning, she takes a few minutes to sit with each developer to learn the remaining estimate of effort on their tasks. She updates the Scrum Sprint Backlog spreadsheet with these estimates, and crosses out completed tasks on the whiteboard.

daily tracker

Sprint Backlog

The first few days are rough and confusing. But, by being forced early to develop a very small amount of code and integrate it with the other developers coordination emerges and a small seed of the overall system starts to take shape and be integrated. Hour by hour, more unit and acceptance tests and production code are added to the build.

Fast forward to mid-iteration, Wednesday January 18. The team meets in a sanity check to discuss if they can really meet their original goals by the end of the following Wednesday (in preparation for the Thursday morning demo), or if they need to scale back. Following the timeboxing practices of UP, XP and Scrum, they won't extend the deadline or work longer hours to meet the deadline, but may defer work until a future iteration. However, things have gone well and the Sprint Backlog shows the total remaining effort estimate is within budget. So, no changes.

ending iterations on Wednesday

Next week on Thursday morning, January 26, the BIG minister shows up for the end-of-iteration demo, a Scrum practice. The iris scan demo doesn't do very much, but it runs and doesn't crash. The minister has never seen an iterative project before and is impressed that three weeks into a new project, the team has some software to show. She's used to waiting six months. In addition to encouraging the team with future efforts, the minister shares this: *I just attended a cabinet meeting where the Prime Minister enthused about a TV report she saw on automated face recognition in crowds, being used in Grepland. She wants it, as soon as possible! This is a matter of national pride; we FooBarKhans can't be outdone by the Greplanders!*

Thursday afternoon and Friday are reserved for a second requirements workshop and iteration 2 planning session. Multiple requirements workshops across early iterations is a UP practice. Since the team is applying the Evo, Scrum, and XP practice of adaptive planning, they had not previously decided what to do in iteration 2, although they had some likely ideas based on the iteration 1 planning session. Rather, they deferred the decision until this time, using their latest information to decide what would be most valuable. With the unexpected overhead and novelty of buying, learning, and integrating a third-party face recognition system, the team decides the next iteration needs more slack and should be four weeks rather than three. The team is very clear what should happen at the iteration 2 demo.

adaptive planning

What's Next?

The next two chapters present motivation for adopting iterative and agile methods and evidence. The evidence chapter is a relatively large summary of the research, history, case studies, and more.

Then, four related chapters describe Scrum, XP, UP, and Evo. The book concludes with a chapter on some detailed practice tips and a FAQ chapter.

Chapter 5. Motivation

If you are going through hell, keep going.

—Sir Winston Churchill

OVERVIEW

- The facts of change on software projects.
- Key reasons to adopt iterative and agile development.
- Meeting the requirements challenge iteratively.
- Problems with the waterfall.

Some have no need to justify iterative development. Others need to make a case; this information can help.

If It Ain't Broke, Don't Fix It

If your organization is applying a waterfall-oriented (or any other) process and has high success rates, productivity, and so forth, don't change. Adopting an iterative or agile method should be motivated by a challenge, not *method-du-jour* fads.

The average case within organizations is relatively high failure rates and other undesirable project qualities [Standish00]; it is in this context that IID methods are worth considering.

failure research

Failure has several meanings. For example, at Symantec (now adopting XP on some projects), the company was attempting a waterfall model. Tight project management resulted in delivering a product on time and in budget, but it was the wrong product [Bowers02]. They adopted XP, which led to a better targeted product; productivity and defect rates also improved.

Project failure can mean not only a cancelled or very late project, but one in which the product did not hit the mark.

The Facts of Change on Software Projects

The **Uncertainty Principle** in software [ZR97]: *Uncertainty is inherent and inevitable in software projects and processes.*

The data summarized in Figure 5.1, based on a large study of software projects [Jones97], illustrates that software development is a domain of inventive high-change projects.

Figure 5.1. rates of change on software projects

The X axis plots project size in **function points**, a language-independent measure of system complexity. The Y axis plots the estimate of overall change or creep in requirements.

function points see www.ifpug.org

Even medium sized projects have change rates around 25%; on very large projects it is 35% or more.

These are high change rates and are at the heart of the motivation for agile and iterative methods.

Further, this data is consistent with another study by Boehm and Papaccio showing that a typical software project experienced a 25% change in requirements [BP88].

This data illustrates that a process, management practice, or value based on the assumption of low change and stability—including "reliable" up-front specifications, estimates, and schedules—is inconsistent with the nature of software projects.

Key Motivations for Iterative Development

Iterative development is lower risk; the waterfall is higher risk. Most practically, the motivation to adopt an iterative lifecycle rather than the waterfall is that research now shows the former is associated with lower risk and better success, productivity, and defect rates. It is these results that have led large and experienced software procurement organizations such as the USA Department of Defense (DoD) to promote the use of IID methods rather than the waterfall.

research

DoD evidence

Early risk mitigation and discovery. Risk-driven iterative development forces tackling the hardest, riskiest problems first, such as architecture, integration, and so on. And, early *development* iterations exercise and reveal the true nature of the team and individual skills, the tools, and third-party software. Finally, the truth of the risks emerges: Perceived risks prove not to be, and unsuspected issues are forced into the open.

Accommodates and provokes early change; consistent with new product development. IID methods work with rather than fight against the high-change nature of software projects.

Manageable complexity. Failure rates are higher and productivity lower with high complexity software projects. Iterative development decomposes complex projects or phases into small and bounded mini-projects of manageable complexity.

size research

Confidence and satisfaction from early, repeated success. Short iterations lead to a quick and repeating sense of completion, competency, and closure. These psychological factors are important for *individual satisfaction* and building *team confidence*. This also builds *customer confidence in the team*—they see early visible progress in the direction they care about.

Early partial product. Not only does early visible progress with an integrated and tested partial product increase client confidence, it provides new business opportunities. Earlier demos are possible. And for whatever reason, the product can ship sooner—with fewer features.

Relevant progress tracking; better predictability. Following the waterfall can give a false sense of progress during the early, easier phases, but with low reliability in predicting later phase schedules, which vary widely. A more meaningful progress indicator—tested software—is provided each iteration when using IID methods. That's Agile Principle 7. Further, since work in each iteration exercises most disciplines and each iteration is a similar mini-project, there is earlier and more representative progress data useful for future extrapolation and estimation.

tracking

agile principles

Higher quality; less defects. IID methods require testing early, often, and realistically, in all possible dimensions: load, performance, usability, and so forth. And the tests themselves can be evaluated and refined over the iterations.

defect research

Final product better matches true client desires. Through early evaluation and feedback from clients or prospective users, the product is more likely to hit the mark. This is a refinement of "higher quality."

Early and regular process improvement. A common practice in IID methods is a per-iteration assessment—for example, a 15-minute discussion to discover a couple of concrete actions to take in the next iteration to address a problem or improve the living process. Broad-spectrum process improvement is enabled by IID, since work in many disciplines (programming, requirements, test, etc.) occurs each iteration.

Communication and engagement required. Failure research reveals that lack of client or end-user engagement is a major factor in software project failure. Likewise with lack of coordination and collaboration between members or sub-teams. Developing in iterations forces early integration, coordination, and communication between development team members. A per-iteration demo that requires the presence and feedback of clients increases their engagement, as does their participation

in a per-iteration planning meeting in which they contribute to the choice of requirements for the next iteration.

failure research

IKIWISI required. There's a well-known human-nature related problem in software specifications, especially user-interface oriented: IKIWISI, or I'll Know It When I See It. The complexity, many degrees of freedom in solutions, and intangibility of software seem to demand concrete and cyclic feedback from people evaluating prototypes or partially built systems to clarify and refine their vision.

Timeboxing Benefits

Research shows that timeboxing itself brings benefits in terms of increased productivity. One reason is ***focus***. Steve McConnell summed it up best in his book *Rapid Development*, "It's amazing how much you can get done the day before a vacation." The psychological focus promoted by a fixed end date only three weeks away is very different than if the team's next visible milestone is three months away. Timeboxing may be viewed as an antidote to **Parkinson's Law**: "Work expands so as to fill the time available for its completion" [Parkinson58].

Another value in timeboxing, both of iterations and of the entire project, is a quirk of human nature: ***People remember slipped dates, not slipped features***. Delay a project three months from its original end date to include 100% of the desired feature set, and the "failure" will be remembered. Deliver on time with 75% of the most important features, and it's a success.

Another reason is being ***forced to tackle small levels of complexity***. With a short two-week timeboxed iteration, the team takes on manageable complexity, gets realistic about what they can do, and has the ability to reduce the scope if it appears the deadline can't be met. Data shows that lower-complexity steps are done more productively.

productivity research

A more subtle benefit of timeboxing is its effect on ***early forcing of difficult decisions and trade-offs***. For example, on a Scrum project, you have just committed to a 30-day timeboxed iteration. During the iteration planning meeting, the team has to be very realistic about what gets done and what gets deferred. Since the demo to the client is definitely in 30 days, there is no latitude to be fuzzy about the short-term goals and priorities. Stakeholders are forced to seriously consider priorities, early.

Meeting the Requirements Challenge Iteratively

Some interesting statistics:

- In a study of over 8,000 projects [Standish94], 37% of the factors on challenged projects were related to requirements (see Figure 5.2).

Figure 5.2. factors on challenged projects

- A study of defects by category [SKTYBE92] found that the largest category was in requirements — 41%.
- Boehm [BP88] showed that the cost of fixing a requirement defect increased non-linearly from early to late in the project.
- *Requirements change 25% or more* [Jones97, BP88].

Thus, we have the requirements challenge,

We want the requirements to be stable, but they aren't.

Attempts to face this challenge by early detailed requirements analysis and freeze can rarely succeed, given the high rates of change. In a study of failure factors on over 1,000 software projects [Thomas01], such practices were associated with the largest contributing factor for failure, being cited in 82% of the projects as the number one problem.

It is also instructive to learn how valuable early specified features really are: As mentioned previously, research of many projects [Johnson02] showed that 45% of features were not used—with an additional 19% rarely used (Figure 5.3).

Figure 5.3. actual use of requested features

Even so, some waterfall requirements advocates have used the above-mentioned cost-of-change research by Boehm [BP88] to continue justifying the practice. It is noteworthy that the creator of this cost data, Boehm himself, was an early and active advocate of evolutionary IID methods, rather than the waterfall, to solve this problem [Boehm85, Boehm96].

Thus, another approach, evolutionary requirements, is now recommended to meet the requirements challenge.

The heart of why IID with evolutionary requirements works:

*It **provokes** the inevitable change early.*

Since requirements will change, IID provokes more of the change early on via early development iterations with feedback and practices such as multiple requirements workshops.

The reality of how people handle the requirements challenge is indeed becoming more iterative. In a study of 107 projects [CM95] only 18% of the projects tried to complete the requirements in a single early step; 32% used two cycles of requirements refinement (with programming in between); and in 50% of the projects the requirements analysis was completed over three or more iterations.

Problems with the Waterfall

Although the original description is more iterative than many know (see p. 102), in common usage the waterfall or sequential lifecycle model meant the following development steps:

1. Define up front, in detail, the requirements.
2. Define the "design" (text and diagrammatic descriptions of the software and hardware elements).
3. Implement the system (programming, and so forth).
4. Integrate and test the components.

In the 1970s this was taught as the ideal approach to software development, in response to ad hoc code-and-fix programming in the 1960s. Many books, consulting companies, and teachers promoted the method as ideal, unaware of the accumulating evidence of its problems. Research now shows it is associated with higher risk, failure, and lower productivity.

evidence

The underlying reason for the difficulties of the waterfall is that it requires a low-change, low-novelty, and low-complexity problem. It is unsuitable for complex or inventive projects. Interestingly—and unknown to many—the author of a key waterfall paper (Winston Royce) said the idea was only applicable for the most straightforward un-novel projects; and most interesting, he was himself a proponent of iterative and evolutionary development.

Royce

The waterfall lifecycle pushes many high-risk and difficult elements toward the end of a project, while IID methods, run by risk-driven iterations, surface and resolve the hardest and riskiest elements early (Figure 5.4). For example, programming and testing the core architecture, integrating its major components and clarifying the interfaces, is tackled in the earliest iterations.

Figure 5.4. risk profile: waterfall

Figure 5.5. risk profile: iterative

The waterfall also aggravates complexity overload and **analysis paralysis**. Large steps with overwhelming degrees of complexity are attempted.

At some level, we need to know specifications before programming. On the scale of three or six weeks (an iteration or short project), the waterfall works. The breakdown occurs as complexity grows, change rates increase, and feedback is delayed; it works less and less well as a single step stretches from weeks to months to years.

As a historical footnote, in the 1970s and 1980s the DoD promoted the waterfall. In 1994, in response to failure and evidence of its unsuitability for software projects, the DoD changed its standards to remove the waterfall bias, and promote iterative and evolutionary development.

DoD standards

Senior executives within software organizations often received their software engineering and management education in the 1970s or 1980s, when the waterfall model was usually taught as "ideal." Therefore, it is understandable and common for some to be unaware of more recent evidence that this was not true.

Thus, sometimes the challenge in moving an organization to adopt IID methods is the reluctance of leadership that still believes in the waterfall.

Other problems with the waterfall and the misapplication of its values on IID projects include the following points:

Problem: "Complete" Up-front Specifications with Sign-off

It isn't that IID proponents wouldn't like correct and detailed up-front specifications that do not need to change—if they could be successfully and efficiently created, very good. Yet, research shows it is rarely possible [CM95], and interestingly, a large study showed that 45% of features created from early specifications were never used—with an additional 19% "rarely" used [Johnson02]. A different approach—evolutionary requirements—is needed.

Problem: Late Integration and Test

I recall working with a defense contractor in the mid-1990s; a large project failure had just wound down. The prime reason? After two years of distributed multi-site development involving over 200 developers, the last stage of the project was to integrate all the software components and do system testing. Characteristically, the integration effort overwhelmingly failed; fundamental misunderstandings and wrong assumptions in the communications, components, and how they would collaborate did not surface until this late stage. The problem was so intractable that the project was cancelled.

Problem: "Reliable" Up-front Schedules and Estimates

When adopting an iterative or agile method, at the start of a project before programming and the first iteration, do not attempt to create a reliable plan or schedule that lays out all the iterations, what will happen in each and when, or all the *detailed* activities sequenced in a PERT chart. Similarly, do not attempt (at the start) to reliably estimate the overall cost or effort, or milestones of intermediate completion points, or the end date.

Such early attempts at **predictive planning** are more successful on repetitive manufacturing projects of low change and complexity, but not on inventive projects—where the full requirements and risks are not reliably known at the start, and high rates of change are the norm. As with up-front specifications, it isn't that IID practitioners wouldn't welcome reliable up-front estimates and schedules. But in domains of high change, complexity, and novelty, it is premature, risky, and unrealistic. Adaptive risk-driven or client-driven planning implies the scheduling of goals to iterations can and should change as better information and new priorities arise.

predictive planning

risk- and client-driven planning

Still, the demand for premature up-front commitments happens all the time. This doesn't invalidate applying an iterative or agile approach—it's just not ideal. Early estimates can be improved with the iterative estimation technique of **Wideband Delphi**. The iterative and agile practice of early visible progress and client-driven iterations can win the customer's confidence so that they eventually appreciate their dynamic steering control, and adopt **adaptive planning**, ceasing to care if the team is doing, week by week, what they were originally forced to speculate and schedule.

Wideband Delphi

adaptive planning

By developing in risk-driven iterations, the team will earlier discover the depth of their predicament and can react with mitigating actions, such as hiring specialists.

The waterfall has been called a *fail-late* lifecycle; one can have the illusion of an accurate schedule during the early, easier phases. This is because the hard and risky elements (like integration and test) are pushed towards the end. Then, halfway through the schedule—as the real complexity and difficult elements surface—pow! The schedule falls apart. It's like the story of the guy who fell off the cliff:

As he was hurtling down, someone yelled, "How are you doing?" The guy replied, "So far, so good!"

Problem: "Plan the Work, Work the Plan" Values

The old management maxim of "plan the work, work the plan" is suitable advice—for predictable manufacturing. For high-change, novel, inventive domains such as software development, it has limited value except at the highest level of very coarse-grained activities or milestones. The maxim is associated with the values of waterfall development, predictive planning, command-control management, up-front specifications, and so forth. This is inconsistent with the agile method principles of adaptive planning, self-organizing and self-directed teams, and evolutionary development.

Of course, this does not imply that agile methods avoid planning or preparation. But the degree of detail, and commitment to plan are more light and flexible. Plus, the devolution of decision making and task assignment to the team itself rather than the manager changes the tone and goals of management planning.

What's Next?

The next chapter presents evidence illustrating problems with the waterfall, advantages of iterative development, case studies, thought leader advice, and more. Then, four related chapters discuss noteworthy IID methods: Scrum, XP, UP, and Evo.

Chapter 6. Evidence

Q: What are the most exciting, promising software engineering ideas or techniques on the horizon?

A: I don't think that the most promising ideas are on the horizon. They are already here and have been for years, but are not being used properly.

—David L. Parnas

OVERVIEW

- Research, historical, and other evidence related to IID.
- Evidence of waterfall risks.
- Business case for skills transfer to adopt IID.

Some have no need to justify iterative development. Others need to make a case; this material can help.

For students of software engineering methods or those needing to motivate an iterative experiment, this chapter provides some data, in several categories:

research evidence	*expert thought leader advice*
early large projects	*business case*
standards-body advice	*waterfall problems*

IID practices have been entrenched for years in some development markets, yet some government standards still promote single-pass, document-driven, waterfall development, as do some larger IT and consulting organizations, government and military service suppliers, engineering curriculums, and many requirements texts. For example, the Call For Papers in the 2002 *Time-Constrained Requirements Engineering* workshop stated:

In general, requirements engineering literature has been working with the assumption that a system should be clearly specified before its design and implementation can start.

Summary

Research evidence— Data shows that iterative and evolutionary development is correlated with lower risk, higher productivity, and lower defect rates than waterfall projects.

Early large project evidence— Major and life-critical systems have been developed iteratively rather than using the waterfall. Examples include the USA Space Shuttle flight control software, developed in 17 iterations, and the new Canadian air traffic control system. In the 1970s, the IBM Federal Systems Division conceived and widely applied the method *Integration Engineerin*g, an iterative lifecycle process.

Standards-body evidence— In the 1980s the USA Department of Defense promoted a waterfall lifecycle in DOD-STD-2167. It was associated with high failure rates. In 1987 a recommendation was made to prefer iterative and evolutionary methods. This occurred in 1994 with the adoption of MIL-STD-498. NATO, the FDA, and other bodies have similar stories.

Expert thought leader evidence— Many prominent software engineering thought leaders have recommended avoiding the waterfall and adopting iterative development, including Harlan Mills, Frederick Brooks, Barry Boehm, James Martin, Tom DeMarco, Ed Yourdon, and more.

Business case— Iterative development is correlated with lower failure rates; the opposite is true of the waterfall. Each year, 23% of projects, averaging $1.1 million USD, fail. A two-year ROI analysis of investing $100,000 in iterative skills transfer could show an NPV of $700,000 with IRR of 200%.

Waterfall problems— The original "waterfall paper" was misinterpreted and seldom read, its author actually endorsed iterative and evolutionary development, the waterfall was associated with high risks, and the creator of the waterfall DOD-STD-2167 standard retrospectively says he would have promoted an iterative rather than waterfall lifecycle.

Why still waterfall promotion?— There are at least seven reasons why the waterfall continued to be promoted, including lack of awareness of the growing evidence that it was not ideal, its simple definition, and the allure of simple progress tracking (such as "requirements complete").

The remainder of the chapter is detail behind the summaries.

Exhaustive data can make for exhausting reading :-) This chapter is probably best spot-read as a reference.

Research Evidence

Iterative and Evolutionary Research

Evidence on the question of IID and evolutionary delivery comes from several studies by Alan MacCormack and others at Harvard Business School. In the first study [MacCormack01, MVI01] the question, "Does evolutionary development, rather than the waterfall model, result in better success?" was explored in a two-year in-depth analysis of projects. The report's conclusion?

Now there is proof that the evolutionary approach to software development results in a speedier process and higher-quality products. … The iterative process is best captured in the evolutionary delivery model proposed by Tom Gilb.

Tom Gilb's Evo.

And specifically on evolutionary feedback-based requirements and design,

… our research suggests a clear agenda for managers: Get a low-functionality version of the product into customers' hands at the earliest possible stage and thereafter adopt an iterative approach to adding functionality.

*… projects in which most of the functionality was developed and tested prior to releasing a beta version performed uniformly **poorly**. In contrast, the projects that performed **best** were those in which a low-functionality version of the product was distributed to customers at an early stage. [emphasis added]*

The study identified four practices that were statistically correlated with the most successful projects:

1. An iterative lifecycle with early release of the evolving product to stakeholders for review and feedback.
2. Daily incorporation of new software and rapid feedback on design changes (daily builds with regression testing).
3. A team with broad-based experience of shipping multiple projects.
4. Early attention to an overall architecture of modular and loosely coupled components.

Practices 1 and 2 are associated with all modern IID methods. Practice 4 is a key element in the UP.

architecture in UP

In a follow-up study [MKCC03], MacCormack and colleagues examined the effect of eight practices on productivity and defects (reported by customers), including IID and releasing a partial system early for evaluation and evolutionary design. The projects ranged from application software to embedded systems, with median values of nine developers and a 14-month duration; 75% used iterative and evolutionary development, 25% the waterfall. A key conclusion of the study:

In this study, we find that releasing [the result of an iteration] at an earlier stage of development appears to contribute to both a lower defect rate and higher productivity.

Given this measure consistently predicts several different dimensions of performance across different samples of projects we conclude that it represents a software development "best" practice.

In contrast, early detailed design specifications were not particularly valuable:

We find a weak relationship (p = 0.0781) between the completeness of the detailed design specification and a lower defect rate.

And detailed design specs did not improve productivity. However, design reviews with peers did significantly reduce defect rates.

In the multivariate model of defect factors, the following iterative-related practices and their magnitude of impact were significant:

- Releasing a partial system (for evaluation, not operation) when 20% of the functionality is complete as opposed to waiting until 40% (the sample median) is associated with a decrease in the defect rate of 10.

by "10" is meant 10 defects per month per million lines of code

-
- Integration and regression testing at code check-in (the XP practice of continuous integration) is associated with a reduction in the defect rate of 13.

Similarly, in the model of productivity factors, over 50% of the variation in productivity was related to just two factors, both related to iterative practices:

- Releasing the partial product earlier with *less* functionality (early iteration internal release for review) was better than waiting for more functionality. An increase in productivity of eight occurred when released at the 20% rather than 40% complete level.

by "8" is meant 8 more lines of source code per person-day

-
- The use of daily builds with integration and regression testing is associated with a productivity improvement of 17.

In a study of productive software organizations [HC96], researchers at Bell Labs found a consistent pattern on highly successful projects:

- Iterative development with customer evaluation and feedback each iteration.
- Simple organizational structure; fewer roles than average.
- Even distribution of communication among people, more direct involvement of developers with other stakeholders, and more overall communication sharing.

A study published in 2001 summarized the results of research into over 400 projects spanning 15 years [CLW01]. Less than 5% of the code was actually useful or used. This high "software pollution" rate (reflecting un-useful requirements and over-engineering within a waterfall lifecycle) was significantly reduced by adopting iterative, short evolutionary delivery cycles—as in the Evo method—reducing releases from about six months on average to about two weeks.

In a survey of agile method results [Shine03], 88% of organizations cited improved productivity, and 84% improved quality. The most frequently used agile methods were Scrum and XP. Regarding cost of development, 46% stated no change and 49% stated it was less expensive with agile methods. One of the more interesting results—predictable in terms of agile method claims—was the increase in business satisfaction with the new software: Overall 83% claimed higher satisfaction and 26% overall

claimed "significantly better satisfaction." The most frequently cited positive feature of agile methods (48%) was "respond to change rather than follow a predefined plan."

Another large study [Standish98] illustrating the value of iterative-related practices is the Standish Group's CHAOS study of project failure and success factors, analyzing 23,000 projects in the 1998 version. In the CHAOS TEN list of the top ten factors for success, at least four of the top five factors are strongly related to IID practices (Table 6.1).

Table 6.1. top five project success factors

Success Factor	Weight of Influence
User involvement	20
Executive support	15
Clear business objectives	15
Experienced project manager	15
Small milestones	10

High user involvement is central to IID methods; short iterations with demos, reviews, evolutionary requirements refinement, and client-driven iterations are key practices.

Executive support is promoted by these practices and especially through the demonstration of early, tangible results; people like to be associated with projects that show quick value and progress.

Clear business objectives is supported by adaptive client-driven iteration planning. By asking each iteration "What is most valuable?" and building it, the business objectives are clarified and realized, and the project aligned with them.

client-driven planning

Of course, **small milestones** are at the heart of iterative methods.

To quote the study,

We have long been convinced that shorter time frames, with delivery of software components early and often, increase the success rate. Shorter time frames foster an iterative process of design, prototype, develop, test, and deploy small elements. "Growing" (instead of "developing") software engages the user earlier and confers ownership.

Size Research

There is significant size research indicating smaller (and thus, less complex) projects are more successful and productive. This is not direct proof of the value of iterative development, but is very relevant to the IID practice of decomposing large projects into a series of small, short sub-project iterations.

A large study [Thomas01] of failure and success factors in over 1,000 UK IT projects found that 90% of the successful projects were less than 12 months duration; indeed, that 47% were less than 6 months. To quote,

This is not to say that projects over 12 months should not be started but that they should be broken into smaller projects within a programme of change whenever possible.

The trend that the larger the project, the more likely it will fail, has been corroborated in a number of other studies. For example, in a study [Jones96] large sample set data show 48% of 10,000 function point (FP) projects are cancelled, as are 65% of 100,000 FP ones.

Going back to early, fundamental size issues, exploration of general systems theory in the 1950s by von Bertalanfy, Bateson, and others led to this fundamental conclusion [Bertalanfy68]:

The larger the system the harder it is to predict its behavior.

More straightforward evidence that small is beautiful comes from a 23,000 project study [Standish98]. For example, project success versus duration, see Figure 6.1.

Figure 6.1. success vs. duration Success was defined as "The project is completed on time and on budget, with all features and functions as originally specified."

This trend was confirmed in a follow-up study spanning 35,000 projects [Standish00], regarding cost (another size measure) versus success (Table 6.2).

Table 6.2. success vs. cost

Cost (USD)	< $0.5M	0.5M-3M	3M-6M	6M-10M	> 10M
Success	68%	22%	9%	1%	0%

And, to reiterate a portion of the Standish conclusion,

… shorter time frames, with delivery of software components early and often, increase the success rate.

Another interesting research note on size in the Standish research was the declining project failure rates, from 31% in the 1994 study to 23% in the 2000 study. This was correlated with smaller, shorter projects and smaller teams.

Direct smaller-size and evolutionary delivery evidence was presented in a previously cited study [CLW01]. The percentage of developed code that was ultimately found to be useful increased when the delivery cycle was reduced from around six months to about two weeks, as recommended in Evo.

Change Research

For cohesion, this section summarizes and repeats research introduced in an earlier chapter.

A study by Boehm and Papaccio showed that a typical software project experienced a 25% change in requirements [BP88]. This trend is corroborated in another large study; as illustrated in Figure 6.2 [Jones97], software development is a domain of inventive high-change projects.

Figure 6.2. rates of change on software projects

Another measure of change is to investigate how much use is actually made of implemented features defined in early specifications. A large study [Johnson02] showed that 45% of features were never used (Figure 6.3).

Figure 6.3. actual use of requested features

Evolutionary requirements to address change is becoming more widespread. A study of 107 projects [CM95] showed that only 18% of the projects tried to complete the requirements in a single early step; 32% used two cycles of requirements refinement (with programming in between); and in 50% of the projects the requirements analysis was completed over three or more iterations.

The data in this section demonstrates that software development is a high-change domain. Practices or values that encourage early "complete" specifications or schedules are incongruous. Iterative and evolutionary practices that emphasize adaptability and steps to provoke early change are consistent with this research.

Waterfall Failure Research

In a study of failure factors on 1,027 IT projects in the UK [Thomas01] (only 13% didn't fail), scope management related to attempting waterfall practices (including detailed up-front requirements) was the single largest contributing factor for failure, being cited in 82% of the projects as the number one problem, with an overall weighted failure influence of 25%. To quote the study's conclusion,

This suggests that … the approach of full requirements definition followed by a long gap before those requirements are delivered is no longer appropriate.

The high ranking of changing business requirements suggests that any assumption that there will be little significant change to requirements once they have been documented is fundamentally flawed, and that spending significant time and effort defining them to the maximum level is inappropriate.

Other significant evidence of failure applying the waterfall comes from one of its most frequent users in the past, the USA Department of Defense (DoD). Most DoD projects were required—by the standard DOD-STD-2167—to follow a waterfall lifecycle. A report on failure rates in a sample of earlier 2167-era DoD projects concluded that 75% of the projects failed or were never used [Jarzombek99]. Consequently, a task force was convened, chaired by Dr. Frederick Brooks, the well-known software engineering expert. The report recommended replacing the waterfall with IID [DSB87]:

DOD-STD-2167 likewise needs a radical overhaul to reflect modern best practice. … In the decade since the waterfall model was developed, our discipline has come to recognize that [development] requires iteration between the designers and users.

Evolutionary development is best technically, and it saves time and money.

See "Standards-Body Evidence"

In another study of 6,700 projects, it was found that four out of the five key factors contributing to project failure were associated with and aggravated by the waterfall model [Jones95], including inability to deal with changing requirements, and problems with late integration.

In 1996 Barry Boehm published a well-known paper summarizing failures of the waterfall [Boehm96], with advice to use a risk-reducing IID approach combined with three milestone anchor points around which to plan and control; this advice was eventually adopted in the UP.

UP phases

There are several studies (covering thousands of projects) that shed light on the value of large, up-front specifications in a waterfall-oriented lifecycle.

One study [Jarzombek99] cited a 1995 DoD software project study (of over $37 billion USD worth of projects) showing that 46% of the systems so egregiously did not meet the *real* needs (although they met the specifications) that they were never successfully used, and another 20% required extensive rework to meet the true needs (rather than the specifications) before they could be used.

As mentioned earlier, another study [Johnson02] showed that 45% of features were never used—with an additional 19% rarely used.

In the previously cited study of over 400 waterfall-oriented projects [CLW01] averaging six-month cycles, only 10% of the developed code was actually deployed, and of that, only 20% was used. The prime reasons included:

- Users couldn't provide much feedback before delivery.
- Changes in the business since the requirements phase.
- Requirements and business operations were misunderstood.

Productivity Research

There is a productivity motivation to apply short iterations, even if there were up-front requirements.

A study [Solon02] against a sample set (43,700 projects) showed the following productivity differences between IID and waterfall:

Rigorous IID or Evolutionary Prototyping	Rigorous Waterfall
570 function points per full-time equivalent developer	480

Interestingly, the same study showed that among the waterfall projects, those that applied it only "loosely" were significantly *more* productive than those that applied it "rigorously," indicating the negative effect that it has on productivity.

Another relevant study [Jones00] showed that as the size of project decreases (measured in language-independent function points), the monthly productivity of staff increases (Figure 6.4).

Figure 6.4. productivity vs. size

This data illustrates the motivation of organizing a project into small mini-project iterations with low function points per iteration, as the most dramatic productivity drop occurs in the lower function point range (under 1,000).

Timeboxing by itself has been shown to have a productivity effect. Dupont, one of the earliest timebox pioneers, found developer productivity around 80 function points per month with timeboxed iterations, but only 15 to 25 function points for other methods [Martin91].

Note the rate of 80 function points per month at Dupont compared to a high of 12 function points per month in Figure 6.4. This suggests that the combination of a low-complexity step *with* timeboxing has a higher productivity impact than simply a small step without timeboxing.

In another study [Jones00], 47 factors that increase or decrease productivity were identified, including project complexity:

Low complexity	High complexity
+ 13% in productivity	– 35%

productivity was measured in function points per person-month

This indicates a productivity advantage by organizing projects in low-complexity mini-project iterations.

To reiterate the results of a study on productivity and iterative development [MKCC03], their conclusion was,

In this study, we find that releasing [the result of an iteration] at an earlier stage of development appears to contribute to both a lower defect rate and higher productivity.

Quality and Defect Research

Broadly, defect reduction comes from avoiding defects before they occur (Deming's Total Quality Management principle) and from feedback (tests, evaluations, and so forth). IID methods can address both. For example, several methods promote a per-iteration simple process assessment or reflection by the team, to encourage regular process improvement and defect avoidance. Feedback is enabled

by the emphasis on early development of the riskiest elements, per-iteration demos, and a test-early, test-often approach. The association of lower defect rates with iterative development is consistent with Deming's predictions, as IID illustrates the Deming/Shewhart Plan-Do-Study-Act (PDSA) cycle, and supports a culture of continuous improvement by measuring, reflecting, and adjusting each iteration.

Specifically, the study [MKCC03] showed that IID was correlated with lower defects. In other research [MVI01], it was shown that as the time lag between coding and testing decreased, defect rates likewise decreased. A study by Deck [Deck94] also shows a statistically significant reduction in defects using an iterative method. Large case-study research [Jones00] showed that defect rates increase non-linearly as project size grows (Figure 6.5).

Figure 6.5. defects vs. size

Although not statistically reliable, there are several single-case study reports of lower defect densities associated with iterative methods (e.g., [Manzo02]).

Early Historical Project Evidence[1]

[1] Some material is adapted from [LB03].

The usual citation style is replaced with inline citations in this section, as a goal is showing the references in context.

This section emphasizes a history of *early* published iterative projects; there were surely many hundreds or thousands more that applied IID in the 1960s and 1970s. IID projects from the late 1980s forward are not emphasized, as by that time they were common.

Pre-1970

Although IID in software is in the ascendancy as the "modern" approach to replace ad hoc or waterfall development, its practiced and published roots go back surprisingly far—at least as far back as the late 1950s, as a contemporary alternative to the nascent waterfall model, perhaps first described as the "stagewise model" for the USA air defense SAGE project by H. D. Benington in "Production of Large Computer Programs," *Proceedings of the ONR Symposium on Advanced Program Methods for Digital Computers*, June 1956.

See "The Historical Accident of Waterfall Validity?"

Early roots for IID are found in the work of Walter Shewhart, the quality expert at Western Electric who proposed, in the 1930s, a series of PDSA cycles (the **Shewhart Cycle** or **Deming Wheel**) for quality improvement, an idea promoted by quality guru Edward Deming, starting in the 1940s. The application of iterative PDSA to software was explored by Zultner in "The Deming Approach to Quality Software Engineering," *Quality Progress*, 21(11) 1988.

A milestone 1950s project applying IID was the X-15 hypersonic jet (see W. H. Dana, "The X-15 Lessons Learned," NASA Dryden Research Facility, 1993):

A major contributor to the X-15's success over the long run was its emphasis on incremental development.

Although not a software project, the X-15 is mentioned because some personnel (and hence, incremental experience) seeded NASA's 1961–63 Project Mercury, which did apply IID in software—and some of the latter project's personnel seeded the IBM Federal Systems Division (FSD), another early IID proponent.

Project Mercury was run with short half-day iterations. There was a technical review of all changes, and—interestingly—the Extreme Programming practice of test-first development was applied: Tests were planned and written in advance of each micro-increment, and then the code was written to pass the tests. Each mini-iteration required integration of all code and passing of tests.

The Project Mercury description came from the software engineering thought leader, Gerald M. Weinberg, who provided a window onto IID versus waterfall attitudes during this period, especially on the highly visible Project Mercury system [LB03]:

We were doing incremental development as early as 1957, in Los Angeles, under the direction of Bernie Dimsdale [at IBM's Service Bureau Corporation]. He was a colleague of John von Neumann, so perhaps he learned it there, or assumed it as totally natural. I do remember Herb Jacobs (primarily, though we all participated) developing a large simulation for Motorola, where the technique used was, as far as I can tell, indistinguishable from XP.

When much of the same team was reassembled in Washington, DC, in 1958 to develop Project Mercury, we had our own machine and the new Share Operating System, whose symbolic modification and assembly allowed us to build the system incrementally, which we did, with great success. Project Mercury was the seed bed out of which grew the IBM Federal Systems Division. Thus, that division started with a history and tradition of incremental development.

All of us, as far as I can remember, thought waterfalling of a huge project was rather stupid, or at least ignorant of the realities.

Since the early 1960s, Weinberg has been a well-known consultant and author, is an inductee of the Computer Hall of Fame, and has written classics such as *Psychology of Computer Programming* and *Secrets of Consulting*.

1970s

Early practice of IID comes from the leadership of Mike Dyer, Bob McHenry, and Don O'Neill (and many others) during their tenure at the IBM FSD, which was responsible for building many aerospace and defense systems.

What is fascinating about the FSD story is the extent and success of IID use on large, life-critical DoD, space, and avionic systems during the 1970s.

The first major FSD documented application of IID (I could find) was in 1972. This was a major, high-visibility, life-critical system of over one million lines of code: the command and control system for the first USA Trident submarine. The project included Dyer, McHenry, and O'Neill as project manager. O'Neill conceived and planned the use of IID (later called **Integration Engineering** at FSD) on this project; it was a key success factor, and he was awarded an IBM Outstanding Contribution Award for it. Note the visible approval by IBM leadership for IID methods.

To underline this point: IBM FSD created the IID method Integration Engineering, and it was a well-known practice on major FSD projects in the 1970s.

The Trident project could not be late: The system had to be delivered by a certain date; otherwise, FSD faced a $100,000 per day late penalty. The project was organized in four timeboxed iterations of about six months each. There was still a significant up-front specifications effort, and the iteration length was longer than normally recommended today. Although there was some feedback-driven evolution in the requirements, the IID approach was seen also as a way to manage the complexity and risks of development. See O'Neill, "Integration Engineering Perspective," *The Journal of Systems and Software*, No. 3, 1983.

Also in 1972, and outside of the IBM FSD, another major project that applied IID comes from an FSD competitor: TRW.

It applied incremental development on the $100 million TRW/Army Site Defense software project for ballistic missile defense. The project began in February 1972 and developed the system in five iterations. Iteration 1 did tracking of a single object, and by iteration 5, a couple of years later, the system was complete. The iterations were not strictly timeboxed, and there was significant up-front specifications work, but they were also refined in response to each iteration's feedback. See Williams, "Managing the Development of Reliable Software," *Proceedings, International Conference on Reliable Software*, ACM/IEEE, April 1975.

As with IBM FSD, TRW was another early adopter of IID practices. Indeed, Barry Boehm, the originator of the famous IID "spiral model" in the mid-1980s, served at TRW as Chief Scientist. Another key TRW employee relevant to this story was Winston Royce, author of the most frequently cited original paper on the waterfall. As will be explored later in "The Historical Accident of Waterfall Validity?" section on page 102, his message was misunderstood, and he was in fact a proponent of IID rather than the sequential waterfall, as it has come to be known.

Another early (mid-1970s), successful, and large application of IID at FSD was the USA Navy helicopter-ship system LAMPS. A four-year 200 person-year effort involving millions of lines of code, it was evolutionarily delivered in 45 timeboxed iterations (one month per iteration). This is the earliest project example I found where the length of iteration was in the range commonly recommended by today's IID methods. The project was quite successful. To quote,

Every one of those deliveries was on time and under budget.

It is described by the noted 1970s thought leader Harlan Mills in "Principles of Software Engineering," *IBM Systems Journal*, Vol 19(4), 1980.

Although unknown to most software professionals, another early and strikingly impressive example of an IID success is the heart of the NASA Space Shuttle software itself: the Primary Avionics Software System. This was built by FSD from 1977 to 1980, applying IID in a series of 17 iterations over 31 months, averaging around 8 weeks per iteration (see Madden and Rone, "Design, Development, Integration: Space Shuttle Flight Software System," *Communications of the ACM*, Sept. 1984). Their motivation for IID?

[The waterfall lifecycle was not suitable because] the requirements on the Shuttle program evolved during the software development process.

(How unique). Ironically—in hindsight of the research evidence— the authors sound almost apologetic in having to forego the "ideal" waterfall for an IID approach,

Due to the size, complexity, and evolutionary [changing requirements] nature of the program, it was recognized early that the ideal software development lifecycle (the waterfall model) could not be strictly applied. … However, an implementation approach (based on small incremental releases) was devised for STS-1 which met the objectives by applying the ideal cycle to small elements of the overall software package on an iterative basis.

This Shuttle project exhibited classic IID practices: timeboxed iterations in the eight-week range, feedback-driven refinement of specifications, and so forth.

The IBM IID method Integration Engineering was so successful that it was incorporated into the FSD Software Engineering Practices (SEP) in 1977 and disseminated to the 2,500 FSD software engineers (see O'Neill, "The Management of Software Engineering," *IBM Systems Journal*, 19(4), 1980). The idea of IID stimulated substantial interest within IBM's commercial divisions, senior customer ranks, and among competitors, and presentations by SEP representatives on the value of IID, rather than the waterfall to visitors was not uncommon during this period.

Another early story and paper often cited on the subject of iterative development is "A Successful Software Development," by Wong, in *IEEE Transactions on Software Engineering*, No. 3, 1984. It describes an air defense system project that ran from 1977 to 1980 that combined significant up-front specifications followed by iterative development. The project was ostensibly meant to fit within DoD single-pass waterfall standards, with testing and integration in the last phase. However, Wong comments on the unrealism of this approach, and their need for and advantages using IID:

The [waterfall] model was adopted because software development was guided by DoD standards. … In reality, software development is a complex, continuous, iterative, and repetitive process. The [waterfall model] does not reflect this complexity.

1980s and Later

The earliest 1980s reference I could find to a large application successfully built via IID is the 1982 $100 million USD command and control project. See Tamanaha, "An integrated rapid prototyping methodology for command and control systems: Experience and insight," *ACM SIGSOFT Software Engineering Notes*, Dec. 1982.

From 1984 to 1988, Magnavox Electronic Systems built a large field artillery command and control system for the USA Army, ultimately a 1.3 million line Ada project. Under the methodology leadership of Don Firesmith, and with some consulting from Grady Booch, it ran as a successful IID project composed of five iterations. The first iteration emphasized building the core architecture. The project had twice the productivity and three times the usual quality measures compared to other Magnavox projects [Firesmith87].

Dozens or hundreds more large IID projects in the 1980s can be discovered, but I'll close this brief early history of IID for large, risky projects with two related case studies started in the mid-1980s: the attempted next-generation USA and Canadian air traffic control (ATC) systems.

The attempted USA ATC project started in 1983 and ran as a classic and massive big-bang waterfall project: It was a herculean attempt to define the requirements before programming, a consequential lack of stakeholder feedback, analysis paralysis, complexity overload, and so on. The conclusion: In 1994 the government cancelled the project after having spent $2.6 billion USD, with nothing to show for it [GAO98]. The project was restarted in the late 1990s with an IID approach. To quote:

[in the new approach] FAA and the aviation community agreed to (1) use an incremental approach to modernizing the National Airspace System, referred to as the "build a little, test a little" approach; … These practices differ from those of the past in which FAA tried to deploy large, complex projects all at once, known as the "big bang" approach. [GAO99]

Similarly, in 1989, a new Canadian ATC project—CAATS—was started as a waterfall project, following guidance from the USA DoD waterfall-promoting standard 2167A. In 1992, after several hundred million dollars spent, requirements paralysis, and little progress deemed valuable, the Canadian government was considering cancelling the project. But, rather than abandon the effort, the project was restarted under the process leadership of Philippe Kruchten, an IID-experienced architect (and later, chief architect of the Rational Unified Process). Kruchten ran it as an iterative project, averaging six-months per iteration with a staff of several hundred, and CAATS re-surfaced as a successfully evolving system with early visible progress, ultimately with over one million lines of source code, mostly Ada. See Kruchten et al., "Modernizing ATC through Modern Software Methods," *1993 Proceedings of the Air Traffic Control Association*.

2167A, unsuccessful, has since been replaced.

By the 1990s, IID use was relatively widespread, probably in the tens of thousands of projects worldwide, though varying adoption trends in different domains and cultures could be seen. Predictably, company and national cultures that emphasized hierarchical management and control, and detailed predictive planning, were the slowest adopters.

Standards-Body Evidence

The transition of the USA DoD standards from waterfall to iterative and evolutionary is instructive. In 1980s a DoD standard for software development or procurement was released, DOD-STD-2167, based on a waterfall model and document-driven approach.[2]

[2] It also had an influence on standards set by other governments. The story is explored in the "The Historical Accident of Waterfall Validity?" section on page 102.

As mentioned earlier, the DoD was experiencing failure in the acquisition of software based on a waterfall approach and the 2167 standard. For example, a 1995 report on failure rates in a sample of earlier DoD projects drew grave conclusions: Out of a total cost of $37 billion USD for the sample set, 75% of the projects failed or were never used, and only 2% were used without extensive modification [Jarzombek99].

Thus, there was a motivation to improve the 2167 standard. In the latter 1980s, Firesmith, motivated from the stultifying influence of 2167 on the Magnavox project, helped lead the Ada community's promotion of IID to replace 2167 with a new IID-friendly 2167A standard. In collaboration with Lt. Colin Gylleat (who was deeply involved in producing 2167A), they were able to remove the legal requirements for functional decomposition and the waterfall development cycle. However, the original single-step waterfall *diagrams* remained in the updated 2167A from 2167, because (in the words of Firesmith):

Although the waterfall diagrams did not have any legal impact, I could not get them removed because the military logistics people would not agree with my assessment that they would continue to foster the waterfall mindset.

Magnavox project

His assessment would prove correct. Shortly thereafter, another push to replace the waterfall with evolutionary IID was made in a task force report [DSB87], chaired by Dr. Frederick Brooks. The report recommended to replace the waterfall, disproved on many large DoD projects, with iterative development:

DOD-STD-2167 likewise needs a radical overhaul to reflect modern best practice. Draft 2167A is a step, but it does not go nearly far enough. As drafted, it continues to reinforce exactly the document-driven, specify-then-build approach that lies at the heart of so many DoD software problems.

And on incremental acquisition and development:

In the decade since the waterfall model was developed, our discipline has come to recognize that [development] requires iteration between the designers and users.

Finally, in the section titled *Professional Humility and Evolutionary Development* (humility that it should be accepted that the waterfall goals of getting the specifications or design accurate without evolutionary implementation and feedback are rarely possible):

Experience with confidently specifying and painfully building mammoths has shown it to be simplest, safest, and even fastest to develop a complex software system by building a minimal version, putting it into actual use, and then adding functions [and other qualities] according to the priorities that emerge from actual use.

Evolutionary development is best technically, and it saves time and money.

The updated DOD-STD-2167A (Feb. 1988), which is often viewed by both DoD overseers and contractors as the epitome of a waterfall specification, was ironically actually an amendment promoted by Firesmith and Gylleat (and others) to allow lifecycle neutrality, to encourage IID alternatives to the waterfall:

This standard is not intended to specify or discourage the use of any particular software development method. The contractor is responsible for selecting software development methods (for example, rapid prototyping) that best support the achievement of contract requirements.

Despite this intent, the new standard was interpreted—with justification—by many to still contain an implied preference for the waterfall model, due to its document-driven milestone approach.

Due to ongoing failure with waterfall projects, and to re-emphasize replacing the waterfall with IID for DoD projects, a *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially*, June 1994, was issued that stated:

DoD must manage programs using iterative development.

Apply evolutionary development with rapid deployment of initial functional capability.

As a consequence, 2167A was superseded by MIL-STD-498 in December 1994. In "Changes from DOD-STD-2167A to MIL-STD-498," *Crosstalk: The Journal of Defense Software Engineering*, April 1995, Maj. George Newberry summarizes the changes to encourage "evolutionary acquisition" and IID in the section *Removing the Waterfall Bias*:

MIL-STD-498 describes … incremental builds. Each build implements a specified subset of the planned capabilities. The process steps are repeated for each build, and within each build, steps may be overlapping and iterative.

The MIL-498 standard explains and promotes evolutionary requirements and design; for example:

If a system is developed in multiple builds, its requirements may not be fully defined until the final build. … If a system is designed in multiple builds, its design may not be fully defined until the final build.

In 2000 the DoD 5000.2 acquisition "instruction" was released, that again recommended evolutionary delivery and the use of IID:

There are two approaches, evolutionary and single step [waterfall] to full capability. An evolutionary approach is preferred. … [In this] approach, the ultimate capability delivered to the user is divided into two or more blocks, with increasing increments of capability.

… software development shall follow an iterative spiral development process in which continually expanding software versions are based on learning from earlier development.

See also the scanned memo in Figure 6.6 that clearly communicates the DoD preference for evolutionary and "spiral" (iterative) development.

Figure 6.6. scanned memo

THE UNDER SECRETARY OF DEFENSE

3010 DEFENSE PENTAGON
WASHINGTON, DC 20301-3010

ACQUISITION, TECHNOLOGY AND LOGISTICS

APR 12 2002

MEMORANDUM FOR SECRETARIES OF THE MILITARY DEPARTMENTS
CHAIRMAN OF THE JOINT CHIEFS OF STAFF
UNDER SECRETARIES OF DEFENSE
ASSISTANT SECRETARIES OF DEFENSE
INSPECTOR GENERAL, DEPARTMENT OF DEFENSE
GENERAL COUNSEL, DEPARTMENT OF DEFENSE
DIRECTORS OF THE DEFENSE AGENCIES

SUBJECT: Evolutionary Acquisition and Spiral Development

Since the publication of DoD Directive 5000.1 and DoD Instruction 5000.2, in which the Department established a preference for the use of evolutionary acquisition strategies relying on a spiral development process, there has been some confusion about what these terms mean and how spiral development impacts various processes such as contracting and requirements generation that interface with an evolutionary acquisition strategy. The purpose of this memorandum is to address those questions.

Evolutionary acquisition and spiral development are methods that will allow us to reduce our cycle time and speed the delivery of advanced capability to our warfighters. These approaches are designed to develop and field demonstrated technologies for both hardware and software in manageable pieces. Evolutionary acquisition and spiral development also allow insertion of new technologies and capabilities over time.

Thus, the DoD—perhaps the world's largest and most experienced procurement agency for software—started with the assumption that a waterfall model and up-front specifications was best (codified in 2167), and then—based on the experience of high rates of project failures—adopted iterative and evolutionary methods, demoting the waterfall.

The practice of IID is important to the DoD. For example, there is now an annual conference on the subject, *Evolutionary Acquisition & Spiral Development*, organized by the Institute for Defense and Government Advancement. See www.idga.org.

Unfortunately (for taxpayers worldwide) STD-2167 influenced the definition of standards in other countries, who did not keep up with the fact that the DoD later abandoned 2167 and the waterfall; many of these standard bodies still impose single-pass, document-driven processes.

The DoD is not the only standards group to make this shift. In 2002 the USA Food and Drug Administration (FDA) updated their prior waterfall model requirements [FDA97] for software development of FDA approved devices (e.g., medical devices) with a new standard that promotes iterative development [FDA02]. To quote,

Most software development models will be iterative. This is likely to result in several versions of both the software requirements specification and software design specification.

Although a number of European standards are still waterfall oriented, promising signs of change have emerged, such as the 2002 Bonn Germany *NATO Symposium on Evolutionary Software Development*.

Expert and Thought Leader Evidence

This section concentrates on some of the earliest and most well known software development thought leaders; many other experts reiterate their views. Where possible, the material is organized by date of publication.

Harlan Mills

The earliest iterative-promoting reference I could find from a prominent expert came in 1970 from Harlan Mills, who worked at the IBM FSD. Mills was perhaps the most well-known software engineering thought leader of the 1970s, winner of the DPMA and Warnier Prizes for lifetime contributions to information sciences, an inventor or contributor to structured programming, top-down design and programming, chief programmer teams, the Cleanroom Method, and incremental development.

In his well-known "Top-down Programming in Large Systems" he promotes iterative development. See *Debugging Techniques in Large Systems*, Prentice-Hall (reprinted in Mills' collected papers, *Software Productivity*, Dorset House, 1988). In addition to the paper's primary advice to develop starting from top-level control structures downwards, Mills gives the related lifecycle advice of building the system via "iterated expansions."

… it is possible to generate a sequence of intermediate systems of code and functional subspecifications so that at every step, each [intermediate] system can be verified to be correct, …

In 1976, after seeing the many successes with IID on IBM FSD projects, Mills reiterated and strengthened his IID message. In the widely read "Software Development," *IEEE Transactions on Software Engineering*, December 1976, he writes:

Software development should be done incrementally, in stages with continuous user participation and replanning, and with design-to-cost programming within each stage.

In the context of analyzing a failed three-year inventory control system project, he goes on to challenge the waterfall:

…there are dangers, too, particularly in the conduct of these [waterfall] stages in sequence, and not in iteration—i.e., that development is done in an open loop, rather than a closed loop with user feedback between iterations. The danger in the sequence [waterfall approach] is that the project moves from being grand to being grandiose, and exceeds our human intellectual capabilities for management and control.

Exasperated at the failures applying the waterfall, contrasted with IID at FSD, Mills asks:

…why do enterprises tolerate the frustrations and difficulties of such [waterfall] development?

And this complaint in 1976.

Further, Mills' Cleanroom Method included IID as part of the method.

Tom Gilb

In 1976 there was the (published) arrival of a long-standing and passionate voice promoting iterative and evolutionary methods: Tom Gilb. This was the first clear IID-promoting book reference I could find, in Gilb's *Software Metrics*, Studentlitteratur AB (Sweden). Some examples:

Evolution [or "Evo," the name of Gilb's iterative method] is a technique for producing the appearance of stability.

A complex system will be most successful if it is implemented in small steps and if each step has a clear measure of successful achievement as well as a "retreat" possibility to a previous successful step upon failure.

The advantage is that you cannot have large failures. You have the opportunity of receiving some feedback from the real world before throwing in all resources intended for a system, and you can correct possible design errors before they become costly live systems.

Gilb also wrote Evo articles in the 1978–79 issues of *Computer Weekly*. And in 1985, he published one of a growing number of articles questioning the sequential lifecycle, in "Evolutionary Delivery versus the Waterfall Model" ACM *Sigsoft Software Requirements Engineering Notes*, July 1985.

Evo chapter

Gilb has been developing and consulting since 1960, is the creator of one of the earliest iterative methods, Evo, and author of popular texts such as *Principles of Software Engineering Management*, Addison-Wesley, 1988, in which he expands on Evo in several chapters.

Frederick Brooks

In 1986 there was the release of a popular article in software engineering, one of whose main themes is the demotion of the waterfall and promotion of IID. It was "No Silver Bullet" by Frederick Brooks, published in *Proceedings of the IFIP Tenth World Computing Conference* (and reprinted in *IEEE Computer*, April 1987). In this classic paper, Brooks criticizes the waterfall as undesirable, and extols the advantages of IID:

Nothing in the past decade has so radically changed my own practice, or its effectiveness [as iterative development].

He rejects waterfall advice and the promotion by requirements engineering advocates of detailed up-front specifications, and lays the blame for much software failure on it:

Much of present-day software acquisition procedures rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software acquisition problems spring from that fallacy.

Like Mills, Brooks was another famous software engineering thought leader. He was manager of the landmark IBM OS/360 operating system project, author of arguably the most widely read software engineering text, *The Mythical Man-Month* (Addison-Wesley, 1985), chair of the mid-1980s DoD study into software project failure (and what to change), and recipient of the highest awards in computer science and software engineering (the ACM Turing award, the IEEE John Von Neumann Medal, and the President's National Medal of Technology).

In the Silver Anniversary edition of *The Mythical Man-Month* Brooks tried to make it more plain by simply stating, *The waterfall model is wrong!*

Barry Boehm

Another well-known mid-1980s landmark in IID publications from an experienced expert comes with Barry Boehm's "A Spiral Model of Software Development and Enhancement," *Proceedings of an International Workshop on Software Process and Software Environments*, March 1985 (although more frequently cited in *ACM SIGSOFT Software Engineering Notes*, August 1986). Boehm's spiral model description promotes iterative and evolutionary development with a strong risk-driven emphasis. To quote:

A primary source of difficulty with the waterfall model has been its emphasis on fully-elaborated documents as completion criteria for early requirements and design phases. For some classes of software, such as compilers …, this is the most effective way to proceed. But it does not work well for many classes of software, particularly interactive end-user applications.

Boehm, having ties to the DoD community, was successful in introducing the spiral model and its refinement, MBASE, on DoD projects, with a 90% rate of "highly satisfactory products" over several hundred projects. See, for example, "Balancing Discipline and Flexibility with the Spiral Model and MBASE," *Crosstalk: The Journal of Defense Software Engineering*, Dec. 2001.

Boehm has been involved in software development since the mid-1950s, has served as Chief Scientist for TRW (an experienced large systems contractor), Director of the USA DARPA Technology Office, and

Chair of the Air Force Scientific Advisory Board. He is the creator of the well-known COCOMO estimation model, and has won the Warnier Prize, the NSIA Grace Murray Hopper Award, and the ACM Distinguished Research Award in Software Engineering.

James Martin

James Martin learned of iterative development and timeboxing from Scott Shultz at Dupont, where the practice was first given the name "timebox methodology" or "Rapid Iterative Production Prototyping" (RIPP). It was widely applied at Dupont in the early 1980s under Shultz's guidance.

Seeing its success, Martin started in the late 1980s to also promote development by timeboxed iterations for use both within and outside of the practice of rapid application development (RAD). See Martin, *Information Engineering: Design and Construction*, Prentice-Hall, 1990. Based on its success with his customers, all subsequent method advice from James Martin promoted iterative development, rather than a sequential waterfall. He writes:

It is desirable [to use an iterative lifecycle] because the traditional lifecycle [the waterfall] is very slow. The traditional lifecycle often fails to meet the needs of end users. … The traditional lifecycle prevents exploratory design and programming which is necessary to creative areas. … Evolutionary development is needed in which systems continually grow.

And:

While simple systems can be designed in one step, complex or subtle systems become understood only by attempting to build a [an evolutionary production-grade] prototype, learning from it, [and repeating]. The prototypes are improved a stage at a time until they converge to what the end users agree is a valuable system.

The creators of DSDM, another popular iterative method, were RAD developers who were influenced by Martin's promotion of timeboxing and iterative development.

Martin has been involved in software development since the early 1960s, and was ranked by *Computerworld* among the top ten most influential people in the world of computing. He has served on the DoD Software Scientific Advisory Board, is an inductee of the Computer Hall of Fame, consulted to hundreds of organizations for decades, and is the author of over 100 texts on software development and technology.

Tom DeMarco

Tom DeMarco is a long-time practitioner and student of skillful software development practices, emphasizing people rather than lifecycle issues in his (and Tim Lister's) popular *PeopleWare* (Dorset House, 1987). In response to my question about his views on iterative development, he wrote (personal communication, email),

Craig, I have been a passive advocate of iterative methods since ever, but never wrote on the subject that I remember (my bag is the people side of management). However, my new book ["Waltzing With Bears: Managing Risk on Software Projects," Dorset House, 2003] with co-author Tim Lister has a whole part on iteration as a (read THE) risk mitigation technique.

In their book, DeMarco writes:

The best bang-per-buck risk mitigation strategy we know is incremental delivery.

DeMarco was winner of the 1986 Warnier Prize for "lifetime contribution to the field of computing" and the 1999 Stevens Award for "contribution to the methods of software development."

Ed Yourdon

In the early 1970s, with Larry Constantine and Tom DeMarco, Ed Yourdon founded Structured Analysis and Design, with a waterfall emphasis. But by the mid-1980s, Yourdon (and his colleagues) had shifted course and has since been a promoter of iterative development in many articles and books—for example, "The Future of Software: Best of Times, Worst of Times," *IEEE Software*, January 1998,

It's also easy to be optimistic when we look at the advances we've made in the "discipline" of our field. [New tools] have led many organizations to an iterative development approach that generally leads to more successful results than the waterfall development approach that was partly necessitated by the older generation of batch, mainframe tools.

Yourdon has been in the field since 1964, is the author of dozens of books, was named one of the ten most influential people in the software field in *Crosstalk: The Journal of Defense Software Engineering*, and in 1997 was inducted into the Computer Hall of Fame.

A Business Case for Iterative Development

As the evidence sections show, there is data to support the assertions that the waterfall is more failure-prone for software-intensive projects and that disciplined IID with evolutionary requirements and design is correlated with lower rates of failure and defects, and higher rates of productivity.

Thus, a business case can be made based on several factors, including:

- productivity
- quality (of both the product and the process)
- failure, delay, or cost-overruns
- fit of product to the true demand

This analysis draws special attention to *reduced failure rates*. Projects fail for many reasons, but evidence [e.g., Thomas01] has shown that waterfall practices rather than IID were associated with the most significant failure factors.

Figure 6.7 shows research on failure rates [Standish00] across 35,000 projects. On average, 23% of projects failed and were cancelled before completion in 2000.

Figure 6.7. success rates

In addition, the average cost of these projects is:

Large Company	Medium	Small
$1.2M USD	$1.1M	$0.6M

Taking the average project cost for medium-sized companies, $1.1 million USD (let's round to $1 million), and the average year 2000 failure rate of 23%, if the organization attempts 20 projects in a

year (a $20 million budget), it loses (conservatively rounding down) $4 million USD on four failed projects.

If—modestly—adopting IID leads to a slightly reduced failure rate of 17% (vs. 23%) then for every 20 projects attempted, one more succeeds—and roughly $1 million is saved.

In this case, even a substantial investment in education and consulting expertise to transition to IID pays off handsomely.[3] If a medium sized company with 10 projects per year and an annual $10 million project budget invested $100,000 in IID skills transfer, then—on average—in two years one more $1 million project will be successful. Assuming a two-year analysis and a 10% discount rate, this investment gives a (rounding) NPV of $700,000 with an IRR of 200%, not to mention reclaiming the lost opportunity cost of putting that $1 million project to work.

[3] Using experienced mentors to coach IID pilot projects is an adoption best practice.

In summary, although goals such as productivity and quality improvement are associated with IID, a primary, costly issue is reducing the expensive rate of failure of software projects—and IID is strongly correlated with the major factors to reduce failure and increase success. The opposite is true for waterfall practices.

The Historical Accident of Waterfall Validity?

The waterfall model was a response to ad hoc code-and-fix development in the 1960s. Note—as has been explored in the "Early Historical Project Evidence" section on page 79—disciplined IID did exist as a contemporary alterative. It was not inevitable that the waterfall be widely promoted (rather than IID) starting in the 1970s; it has a bit of an accidental quality to it, as I discovered in my historical research.

Misunderstanding started early, with the article and author most often cited for the waterfall: "Managing the Development of Large Software Systems" by Winston Royce, in *Proceedings of IEEE Westcon*, 1970.

In this article—sometimes (incorrectly) identified as the paragon of single-pass waterfall—Royce actually recommends an approach different than what devolved into the popular notion of waterfall, with its strict single-pass sequence of requirements analysis, design, and development phases. He recommends to "do it twice." To quote:

If the computer program in question is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations areas are concerned.

He goes on to suggest that a 30-month project might have a 10-month throw-away "pilot model" and justifies its necessity when there are novel elements and unknown factors (hardly a unique case).

Winston's son Walker Royce described what his (deceased) father felt about the misinterpretation of the waterfall that was attributed to him, and the widespread promotion of a document-driven single-pass approach [LB03]:

He was always a proponent of iterative, incremental, evolutionary development. His paper described the waterfall as the simplest description, but that it would not work for all but the most straightforward projects. The rest of his paper describes [iterative practices] within the context of the 60s/70s government contracting models (a serious set of constraints).

It is ironic that the author of the seminal waterfall paper was a proponent of iterative and evolutionary development, that his paper was only describing a process for the most straightforward projects, and that Royce did not subscribe to the simplified single-pass waterfall, as it is often described.

Note also that the paper was not based on research into various successful lifecycle choices for large, novel projects. It did not cite any evidence, trends, or other projects. Indeed, the paper opens with this sentence:

I am going to describe my personal views about managing large software development.

Even by the time DOD-STD-2167 was adopted in the 1980s there was a growing body of experience and recommendations by thought leaders to avoid rather than embrace the (misunderstood) waterfall, but this knowledge did not find its way into the 2167 standard. Why? One part of the reason is that these standards were usually combined with and influenced by MIL-STD-1521B, another standard that required a series of document-driven reviews, such as a requirements review. I found another part of the answer in Boston.

In 1996 I visited the Boston area and had lunch with the principal author of 2167. He expressed regret for the creation of the rigid single-pass waterfall standard. He said he was influenced by common knowledge and practice of the time, plus other standards (e.g., 1521B). He was not familiar with the practice of timeboxed iterative development and evolutionary requirements at the time, and in hindsight, said he would have made a strong IID recommendation, rather than what was in 2167.

It is no small irony that 2167 was then used as input to other standards, both within the United States and internationally. For example, the British JSP-188, German V-Model (also the basis of the Austrian and Swiss standards), and the French GAM-T-17 were influenced by 2167, with an emphasis on single-pass waterfall phases, and early large, signed-off specifications before construction.

Although DoD reports started to publicly caution against 2167 and the waterfall in 1987, and 2167 was replaced with an IID-promoting standard in 1994, other governments and standards bodies that had drawn from 2167 did not likewise update their standards, apparently unaware of the changes afoot.

As I uncovered these stories, I was struck by the unintended influence a small number of individuals had on a world of standards and projects. And, how the misunderstanding and speculation that actually lay behind these standards led to a kind of accidental perception of validity for the waterfall, making of it a mirage as the "obvious tried-and-true" best practice for large, novel, complex projects, when in fact this was not the case.

Why Did Waterfall Promotion Continue?

H. L. Mencken quipped, "For every complex problem, there is a solution that is simple, neat, and wrong." In the history of science it is normal that lesser ideas first hold the dominant position, even in the absence of results. Western cosmology's Earth-centric universe dominated Europe for over a millennium until enough evidence and brave souls accumulated to depose the model. Software development is a young field, so it is no surprise the simple formula of "requirements, design, implementation" held sway during the first attempts to create a skillful development process. Other reasons for the early and ongoing adoption of the single-pass waterfall idea include:

- Few actually read Royce's original waterfall paper [Royce70]. Its iterative flavor was lost, and it devolved from the nuanced evolutionary description he gave, to a simple single-step lifecycle. This is seen in the many third-party diagrams supposedly depicting "Royce's waterfall," that do not correctly correspond to the original iterative picture Royce gave.
- Few realized that, as in the words of Royce's son, "[My father] was always a proponent of iterative, incremental, evolutionary development. His paper described the waterfall as the simplest description, but that it would not work for all but the most straightforward projects."
- The single-pass waterfall had simplicity of explanation and recall ("do the requirements, then design, and then implement"); IID is more complex to understand and describe. Note that even Royce's original two-iteration waterfall immediately devolved into a single step model by other adopters and writers.
- As discussed in the opening chapter, software projects have been inappropriately associated with a predictable manufacturing paradigm (such as mobile phones) that can be predictably specified and planned, rather than a new product development paradigm.
- Single-pass waterfall has been favored by some management because it gives the illusion of an orderly, predictable, accountable, and measurable process, with simple document-driven milestones (such as "requirements complete"). There is a special irony in choosing a simple-to-track process that yields higher levels of risk.
- Waterfall values and big up-front specification goals continued to be promoted by requirements engineering (and other groups) as appropriate or even ideal decades after large project experience, research, standards bodies, and the criticism of leading experts advised

against it. Perhaps this was due to unfamiliarity with the evidence or with how disciplined iterative and evolutionary requirements could be made to work.

- The **Capability Maturity Model** (CMM) from the Software Engineering Institute (SEI) influenced some software process engineers in the late 1980s and 1990s towards gated, document-driven, waterfall practices [SEI03]. Although an IID-approach can be certified as CMM-mature, early CMM discussions had a tone of document and plan-driven, phase-oriented, and predictive planning. Many CMM certifiers and consultants had a background in waterfall values and practices and prescriptive process, without experience in iterative and adaptive methods. More recently, SEI CMM thought leaders have promoted IID and agile methods [Paulk01].

  see www.sei.cmu.edu

-

  CMM defines levels of process maturity. Level 1 is chaotic, heroic effort. Level 5 is a reflective, constantly improving system.

-

  prescriptive process

-
- The Project Management Institute (PMI) educates and certifies managers, and influences management values via its Project Management Body of Knowledge (PMBOK). The PMI and PMBOK have valuable contributions, and acknowledge iterative and evolutionary methods. Yet, some early PMBOK content had a tone of "plan the work, work the plan," phases, and predictive planning more consistent with predictable manufacturing projects and the waterfall, than with evolutionary methods for high-change inventive projects.

  see www.pmi.org

-

  problems with "plan the work, work the plan"

What's Next?

Next are four related method chapters that explain the practices of Scrum, XP, UP, and Evo. After that, the book concludes with a practice tips chapter and a FAQ chapter.

Recommended Readings

The many cited papers within this chapter are the place to go for more details, although in some cases I had to contact the original people to help fill in the story. Many older IEEE and ACM journal papers, and some older *IBM Systems Journals*, are now available on the Web.

Chapter 7. Scrum

Ours is too great and too complex a nation for even such as I to direct and lead every action.

—Attila the Hun

OVERVIEW

- Classification of Scrum.
- Workproducts, roles, and practices.
- Common mistakes, adoption and process mixtures, strengths and weaknesses.

Scrum appears simple, yet has practices that deeply influence the work experience and that capture key adaptive and agile qualities. Scrum's distinctive emphasis among the methods is its strong promotion of self-directed teams, daily team measurement, and avoidance of prescriptive process. Some key practices include:

prescriptive process

- self-directed and self-organizing team
- no external addition of work to an iteration, once chosen
- daily stand-up meeting with special questions
- usually 30-calendar day iterations
- demo to external stakeholders at end of each iteration
- each iteration, client-driven adaptive planning

Method Overview

Classification

In terms of cycles and ceremony, Scrum classification is illustrated in Figure 7.1. Scrum is uniquely precise on the length of iterations: usually 30 calendar days, a more-or-less common length compared to other IID methods.[1]

[1] Shorter is legal, but 30-day iterations are encouraged.

Figure 7.1. Scrum on the cycles and ceremony scale

cycles and ceremony

Scrum is flexible on the ceremony scale; discussion of what and how many workproducts is outside its scope, as is how much rigor is required. As a guiding principle, the Scrum founders would say, "as little ceremony as possible." Also on a Scrum project, the whole team—not a manager—will decide how much is appropriate.

High levels for a medical device are acceptable, as are low levels for a casual-information read-only Web site.

In terms of scope on the Cockburn scale, Scrum covers the cells shown in Figure 7.2. Although one Scrum team should be seven or less, multiple teams may form a project. It has been used on both small projects and those involving hundreds of developers. Since Scrum practices include working in a common project room, it scales via a "scrum of scrums" where small teams work together and hold a daily stand-up meeting, and representatives from each those teams likewise meet daily. Scrum is complementary enough to other practices that it may be applied across all domains of software applications, from life-critical to more casual—and it has.

Figure 7.2. Scrum on the Cockburn scale

Cockburn scale

Introduction

Scrum [SB02] is an IID method that emphasizes a set of project management values and practices, rather than those in requirements, implementation, and so on. As such, it is easily combined with or complementary to other methods.

A key Scrum theme is its emphasis on **empirical** rather than **defined** process. Ken Schwaber, one of the Scrum founders, tells a noteworthy story in this context [SB02]:

empirical and defined methods

I wanted to understand why my customers' [waterfall and detailed-defined] processes didn't work for my [software] company, so I brought several methodologies to process theory experts at the DuPont Experimental Station in 1995. These experts, led by Babatunde Ogannaike, are the most highly respected theorists in industrial process control.[2]

They inspected the systems development processes that I brought them. I have rarely provided a group with so much laughter. They were amazed and appalled that my industry [software], was trying to do its work using a completely inappropriate process control model. They said systems development had so much complexity and unpredictability that it had to be managed by a process control model they referred to as "empirical." They said this was nothing new, and all complex processes that weren't completely understood [or had changing inputs] required the empirical model [and not the defined process model].

… [Ogannaike] said my business was an intellectually intensive business that required too much thinking and creativity to be a good candidate for the defined approach. … He was particularly amused that the tasks were linked together with dependencies [in a PERT chart], as though they could predictably start and finish just like a well-defined industrial process.

[2] Ogannaike has written one of the primary university textbooks on industrial process control; see [OR94].

Ogannaike's words echo Deming and Shewhart's industrial process control emphasis on cyclic Plan-Do-Study-Act for complex, changing systems and environments.

Shewhart and Deming

Lifecycle

The Scrum lifecycle is composed of four phases: Planning, Staging, Development, and Release. The following example illustrates a Scrum project and the lifecycle: Jeff Sutherland, one of the Scrum founders, serves at an organization that applies Scrum to build a hand-held medical system. Their evolutionary delivery release cycle is three months, composed of three Scrum Sprints (iterations). As this is a relatively mature and ongoing project, the vision-oriented Planning phase is bypassed (its goals having already been satisfied), and each release begins with Staging—addition and prioritization of items to work on in the first of the three Sprints. During the prior three-month release cycle, some subject matter experts were involved in enough requirements analysis to kick-start the release. In collaboration with the architect and some other team members, the new items are assigned tentative estimates, with no item larger than three person-days of effort. Staging is followed by three iterations of the Development phase. Quality assurance (QA) occurs in each, but the third Sprint has a special focus on QA, with less new development.

Workproducts, Roles, and Practices

WORKPRODUCTS
(non-software)
Requirements
Design
Product & Release Backlog
All items for the product. The Release Backlog is a subset.
Includes features, use cases, enhancements, defects, technologies.
Implementation
Test & Verification
Project Management
Configuration & Change Management Environment
Backlog Graph
Estimate of work remaining versus days.
Sprint Backlog
Tasks for the iteration. Granularity 4-16 hours.
Product & Release Backlog

Roles

ROLES

Customer

Product Owner

- *one* person who is responsible for creating and prioritizing the Product Backlog

- chooses the goals (from the Product Backlog) for the next Sprint

- along with other stakeholders, reviews the system at the end of each Sprint

Development

Scrum Team

- work on the Sprint (iteration) Backlog

- there is explicitly no other title than "team member"

Management

Scrum Master

-50% developer, not just management

-knows and reinforces the project and iteration vision and goals

- ensures Scrum values and practices followed

- mediates between Management and Scrum Team

- listens to progress and removes impediments

- conducts the Daily Scrum

- conducts the Sprint Review (demo)

Other

Chickens

- everyone else can observe, but not interfere or speak during an iteration

Practices

Core Practices

Practice	Description
Pre-game planning and staging	During Pre-game Planning, all stakeholders can contribute to creating a list of features, use cases, enhancements, defects, and so forth, recorded in the Product Backlog. One Product Owner is designated its owner, and requests are mediated through her. During this session, at least enough work for the first iteration is generated, and likely much more. Starting at these meeting and evolving over time, is identification of the Release Backlog, the subset of the Product Backlog that will make the next operational or product release.
Sprint planning	Before the start of each iteration—or Sprint—two consecutive meetings are held. In the first, stakeholders meet to refine and re-prioritize the Product Backlog and Release Backlog, and to choose goals for the next iteration, usually driven by highest business value and risk. In the second meeting, the Scrum Team and Product Owner meet to consider how to achieve the requests, and to create a Sprint Backlog of tasks (in the 4–16 hour range) to meet the goals. If estimated effort exceeds resources, another planning cycle occurs. As the iteration proceeds, the Sprint Backlog is updated, often daily during the early part of the iteration, as new tasks are discovered. As a history of many Sprint Backlogs grows, the team improves their creation of new ones.

Practice	Description
Sprint	Work is usually organized in 30-calendar-day iterations; each is called a Sprint.
Self-directed and self-organizing teams	During an iteration, management and the Scrum Master do not guide the team in how to fulfill the iteration goals, solve its problems (other than to make decisions when requested, and remove reported blocks), nor plan the order of work. The team is empowered with the authority and resources to find their own way, and solve their own problems. This hands-off approach for 30 days, except to provide resources and remove blocks, is perhaps the most personally challenging aspect for management adopting Scrum.
Scrum meeting	Each workday at the same time and place, hold a meeting with the team members in a circle, at which the same special Scrum questions are answered by each team member. Details are described in the next section.
Don't add to iteration	During an iteration, management does not add work to the team or individuals. Uninterrupted focus is maintained. In the rare case something has to be added, something else should ideally be removed. But, before each new iteration, the Product Owner and Management have the right and responsibility to re-prioritize the Product Backlog, and indicate what to do in the next iteration, as long as the work request estimates don't exceed the resources.
Scrum master firewall	The Scrum Master looks out to ensure the team is not interrupted by work requests from other external parties, and if they occur, removes them and deals with all political and external management issues. The Scrum Master also works to ensure Scrum is applied, removes reported blocks, provides resources, and makes decisions when requested. She also has to take initiative when she sees during the meeting that someone isn't completing work, if the team doesn't speak up.
Decision in one hour	Blocks reported at the Scrum Meeting that require decisions by the Scrum Master are ideally decided immediately, or within one hour. The value of "bad decisions are better than no decisions, and they can be reversed" is promoted.
Blocks gone in one day	Blocks reported at the Scrum Meeting are ideally removed before the next meeting.
Chickens and pigs	During the Scrum Meeting, only the Scrum Team can talk (the pigs). Anyone else can attend, but should remain silent (the chickens), even the CEO. An exception is management (e.g., CEO) feedback on survival points or explanation of the business relevance of the team's work. The Scrum needs to be a vehicle for communicating the product vision and organization goals. From this story: A pig and chicken discussed the name of their new restaurant. The chicken suggested *Ham n' Eggs*. "No thanks," said the pig, "I'd be committed, but you'd only be involved!"
Teams of seven	Scrum can scale to large projects, but recommends one team have a maximum of seven members. Larger projects are multi-team.
Common room (preferred)	Ideally, the team work together in a common project room, rather than separate offices or cubes. Separate, private space is still available for other activities. However, teams composed of geographically spread members, participating by speakerphone in the Daily Scrum, have reported success.

Practice	Description
Daily build	At least one daily integration and regression test across all checked-in code for the project. The XP practice of Continuous Integration is even better.
Sprint review	At the end of each iteration, there is a review meeting (maximum of four hours) hosted by the Scrum Master. The team, Product Owner, and other stakeholders attend. There is a demo of the product. Goals include informing stakeholders of the system functions, design, strengths, weaknesses, effort of the team, and future trouble spots. Feedback and brainstorming on future directions is encouraged, but no commitments are made during the meeting. Later at the next Sprint Planning meeting, stakeholders and the team make commitments. "Power Point" presentations are forbidden. Preparation emphasis is on showing the product.

The Scrum Meeting: Details

The Scrum Meeting—or scrum—is the heartbeat of Scrum and the project. Each workday at the same time and place, hold a meeting with the team members *standing* in a circle, at which time the same special questions are answered by each member:

1. What have you done since the last Scrum?
2. What will you do between now and the next Scrum?
3. What is getting in the way (blocks) of meeting the iteration goals?

I've added two more questions—since shortly after starting to apply Scrum in 1998—that have been useful:[3]

[3] These additional questions have been reviewed and approved by Schwaber and Sutherland, the Scrum founders.

4. Any tasks to add to the Sprint Backlog? (missed tasks, not new requirements)
5. Have you learned or decided anything new, of relevance to some of the team members? (technical, requirements, …)

The last question provides an efficient forum for a continuously improving and learning group—vital to agile development—and is often an interesting way to end the reports, increasing their perceived value.

Other points:

- The meeting is ideally held in a stand-up circle to encourage brevity.
- On average, 15 or 20 minutes for 7–10 people. Longer meetings are common near the start of an iteration.
- Non-team members (chickens) are outside the circle.
- It is held next to a whiteboard at which all the tasks and blocks are written when reported.
- The Scrum Master erases blocks only once they've been removed.
- There is a speaker-phone for offsite member participation—which is required.
- The Scrum Master ensures the rules are followed and prepares the location for an efficient meeting.
- Must start on time. Late fines collected by the Scrum Master and donated to charity are a popular rule.
- Chickens and Pigs rule enforced: Non-team members don't talk or ask questions. An exception is management feedback on survival points or explanation of the business relevance of the team's work. The Scrum needs to be a vehicle for communicating the product vision and organization goals.

- No other discussion is allowed beyond the three (or five) questions. The Scrum Master has authority to refocus the discussion.
- If other issues need discussion, secondary meetings immediately after the Scrum Meeting occur, usually with subsets of the team. For example, during the Scrum meeting, I may say to Jill, during her answer report, "We need to talk about that. Let's meet after the Scrum."

The Value of the Scrum Meeting

Value: Since there is a self-directing and organizing team, with no manager directing workers or solving problems (unless asked) during an iteration, the Scrum Meeting creates the daily mechanism to quickly inform the team about the state of the project and people. Then, people can take action. External people can observe the daily Scrums to get an accurate, timely, and information-rich measure of progress and issues. It supports openness and allows resolution of dependencies and conflicts in real time to maximize throughput.

Value: When a person reports on what they are doing for the next day, they are expressing a kind of social promise to the team. This increases responsibility and follow-through.

Value: Scrum is based on the insight that software development is creative and unpredictable new product development, and therefore empirical rather than defined methods are needed. The Scrum Meeting provides the frequent measuring and adaptive response mechanism that empirical methods require.

Value: Project risks include not accounting for all tasks, poor estimates, and not quickly resolving blocks. The Scrum Meeting provides a daily forum to update tasks, and surface and remove impediments.

Value: It is important to have people (and teams) that are continuously improving and learning. The Scrum Meeting supports this, especially with the addition of Question 5. Unspoken (or *tacit*) information and knowledge becomes spoken and shared.

Value: Shared language, values, and practices help a development team. This is created and reinforced in the daily Scrum.

Workproducts

In addition to the workproducts illustrated on p. 114, Scrum allows any other workproducts of value to the project. For example, it can be combined with some UP practices, and one can create a Vision or Risk List, using UP terminology.

Product Backlog— A sample, partial Product Backlog is shown in Figure 7.3. Note that all conceivable items go in the backlog and are prioritized by the Product Owner. The estimates (in person-hours of effort) start as rough guidelines, refined once the team commits to an item.

Figure 7.3. sample Product Backlog

	A	B	C	D	E	F
1	Product Backlog					
2						
3	Requirement	Num	Category	Status	Pri	Estimate
4	log credit payments to AR	17	feature	underway	5	2
5	process sale-simple cash scenario	232	use case	underway	5	60
6	slow credit payment approval	12	issue	not started	4	10
7	sales commission calculation	43	defect	complete	4	2
8	lay-away plan payments	321	enhance	not started	3	20
9	PDA sale capture	53	technology	not started	1	100
10	process sale-credit pmt scenario	235	use case	underway	5	30

Sprint Backlog— A sample Sprint Backlog is shown in Figure 7.4. Note the daily estimate of work remaining for each task; these columns also show the date (e.g., 6 of Jan) and total hours remaining on each day (e.g., Jan 6, 362 hours). It is updated daily by the responsible members or by a daily tracker who visits each member. New estimates are allowed to increase above the original estimate. The simplest (and thus preferred) tool is a spreadsheet; Sutherland uses a customized version of the open-source GNU GNATS tracking tool, with a Web interface.

Figure 7.4. sample Sprint Backlog

	A	B	C	D	E	F	G	H	I
1	Sprint Backlog								
2	Task Description	Originator	Responsible	Status	Hours of work remaining				
3					6	7	8	9	10
4					362	322	317	317	306
5	Meet to discuss the goals and	JM	JM/SR	Completed	20	10	0	0	0
6	Move Calculations out of	TL	AW	Not Started	8	8	8	8	8
7	Get GEK Data		TN	Completed	12	0	0	0	0
8	Analyse GEK Data - Title		GP	In Progress	24	20	30	25	20
9	Analyse GEK Data - Parcel		TK	Completed	12	12	12	12	12
10	Define & build Database		BR/DS	In Progress	80	80	75	60	52

Sprint Backlog Graph— A Sprint Backlog Graph is shown in Figure 7.5. It is a visual summary of estimated task hours remaining in the Sprint Backlog. In Scrum, this is considered the most critical project data to track. Recommended: Post an updated version of this each day on the wall by the Scrum meeting.

Figure 7.5. sample Backlog Graph

Other Practices and Values

- **Workers daily update the Sprint Backlog**— Once tasks are underway, individuals are responsible for daily updates estimating the time remaining for their tasks.
- **No PERT charts allowed**— A PERT chart is built on the assumption that the tasks of a project can be identified, ordered, and reliably estimated, that there is minimal change and noise in the system, and in general that a defined process can be applied. This is inconsistent with the recognition in iterative and agile methods that software is semi-chaotic new product development with high degrees of change and noise, and defined processes can't apply.
- **Scrum Master reinforces vision**— She needs to daily share and clarify the overall project vision, and goals of the Sprint, perhaps at the start of the Scrum meeting.
- **Replace ineffective Scrum Master**— The manager/Scrum Master is the servant of the developers, not vice versa. If Scrum Master is not removing blocks promptly, acting as a firewall, and providing resources, the Scrum founders encourage replacing the Scrum Master.

Values

The Scrum values are described in [SB02]:

- **Commitment**— The Scrum Team commits to a defined goal for an iteration, and is given the authority and autonomy to decide themselves how best to meet it. Management and the Scrum Master commits to not introduce new work during an iteration, avoid directing the team, and work to provide the resources and quickly remove blocks that the team reports in their daily Scrum meeting. The Product Owner commits to define and prioritize the Product Backlog, guide choice of the next iteration's goals, and review and provide feedback on the result of each iteration.
- **Focus**— The Scrum Team has to be able to focus on the stated goals of the iteration, without distraction. Thus, management and the Scrum Master focus on providing the team with resources, removing blocks, and avoiding interrupting the team with additional work requests.

- **Openness**— The openly accessible Product Backlog makes visible the work and priorities. The Daily Scrums make visible the overall and individual status and commitments. Work trend and velocity are made visible with the Backlog Graph.
- **Respect**— Or, team responsibility rather than scapegoating. The individual members on a team are respected for their different strengths and weaknesses, and not singled out for iteration failures. The whole team rather than a manager, through self-organization and direction, adopts the attitude of solving "individual" problems through group exploration of solutions, and is given the authority and resources to react to challenges, such as hiring a specialist consultant to compensate for missing expertise.
- **Courage**— Management has the courage to plan and guide adaptively and to trust individuals and the team by avoiding telling them how to get the iteration done. The team has the courage to take responsibility for self-direction and self-management.

Error: Not a self-directed team; managers or Scrum Master direct or organize the team— The urge may be strong during an iteration to tell or suggest to team members how to work, or solve a problem. Many managers are used to an emphasis on directing and planning, rather than their role in Scrum: To quickly remove blocks, provide resources, act as a firewall to the rest of the organization, and otherwise stay out of the way. This is especially true for the Scrum Master during the Scrum Meeting, when there is a natural tendency for the team to look to a leader for direction and solutions.

Error: No daily update of the Sprint Backlog by members or daily tracker— Self-explanatory.

Error: New work added to iteration or individual— In a sea of constant change, some stability is required. Not changing the requirements for an iteration, once begun, is Scrum's point of control.

Error: Product Owner isn't involved or doesn't decide— Scrum is customer driven; the Product Owner needs to decide what the Product Backlog priorities are and choose the requirements for the next iteration.

Error: No Sprint Review— Feedback and adaptation drive Scrum; the demo and review are needed to inform the customers, so they can steer the next iteration.

Error: Many masters— Scrum requires one voice on the Product Backlog requirements, priorities, and work for the next iteration: the Product Owner.

Error: Documentation is bad— Scrum isn't anti-documentation; discussion of project workproducts is simply outside the scope of its definition. As with all agile methods, non-code workproducts are expected to add real value, rather than be created for the sake of following a process formula.

Error: Design or diagramming is bad— Scrum is pragmatic rather than doctrinaire on the team's approach to design: If they find value in some pre-programming design or diagramming work during an iteration, it's done.

Error: Full team (including customers and management) not briefed in Scrum and its values— Self-explanatory.

Error: Scrum Meeting too long or unfocused— Keep it below 20 minutes, and focused on the Scrum questions.

Error: Iteration doesn't end in an integrated and tested partial product— An iteration doesn't just finish on the end date. The goal is that all the software has been integrated, tested, and baselined.

Error: Each iteration ends in a production release— Although a Scrum iteration *may* end in a production release, it is not a requirement. It may require many iterations before readiness.

Error: Predictive planning; PERT chart planning— As with all IID methods, it is a misunderstanding to create a plan laying out exactly how many iterations there will be for a long project and what will occur in each, or to create a PERT chart identifying many tasks, their order and estimated duration.

You Know You Didn't Understand Scrum When...

Some of the key misunderstandings expressed as a checklist:

- You think a manager or Scrum Master should tell the team what to do, or how to solve its problems.
- Customers are not involved in each iteration, not prioritizing requirements, not attending each demo, and are not choosing the highest business value set for the next iteration.
- New requirements or extra-project tasks are added to team members during an iteration.
- You create a plan laying out how many iterations there will be for the project, and what will occur in each, and think you can enforce it.
- You create a PERT chart or a plan of dependent, ordered tasks, with estimated durations.

Sample Projects

The following projects had significant Scrum influence:

Large— IDX Web-enabled benefits suite

- One year, 330 people across multiple related projects, an E300 project on the Cockburn scale, [SB02]

- A suite of 15 related applications were developed within one year of adopting Scrum, after three years of struggle to deliver one application.

- Prime developer: IDX.

Medium— Caremark

- Four months, 20 people, an E20 project, [SB02]

- After two years of struggle, 160 staff at its height, and no delivery, Scrum was introduced with a reduced team size of 20 developers (10 new hires). In four months, a successful production release was created.

- Prime developer: Caremark and consultants.

Small— Individual Personal NewsPage

- One month, eight people, a C20 project [SB02]

- After nine months without delivery, Scrum was adopted, and a usable production release emerged after one 30-day iteration. After five months of releases, most of the original goals were achieved.

- Prime developer: Individual Inc.

Process Mixtures

Scrum + Evo

Some Evo practices are compatible with Scrum. Scrum does not discuss specific specification methods, and thus Evo's Planguage is applicable. Evo's measurement emphasis is compatible; indeed, Sutherland, one of the Scrum creators, takes a strong interest in measurement when applying Scrum.

Evo

Planguage

Scrum's 30-day iteration length is not consistent with Evo—too long.

Scrum + UP

The Scrum practices are either equal to or specializations of UP practices, or are consistent additions. If some workproducts are required on a Scrum project, using the UP versions is reasonable. The Product Backlog is an acceptable portion of the UP Project Plan, and the Sprint Backlog is an acceptable version of the UP Iteration Plan.

UP

One area of different emphasis is the presence in the UP of optional but predefined activities; the UP describes a set of possible activities related to requirements analysis, testing, and so forth. And, the UP indicates some dependent ordering of these optional activities; for example, that a project vision is created before a detailed requirement is described. Scrum's rejection of defined process and predictable steps is inconsistent with this structure, if the UP activities are viewed as a required formula. But, if the activities are treated as optional advice, performable in any order, and without attempt to schedule their order and duration on a project, it is within Scrum.

See "UP as a Heavy, Defined Process versus an Agile UP Approach"

Scrum + XP

Most Scrum practices are compatible with XP or refinements, such as the Scrum Meeting. Indeed, Kent Beck borrowed the XP stand-up meeting idea from Scrum.

XP

The Scrum practice of a demo to external stakeholders at the end of each iteration enhances XP's feedback and communication goals. The Scrum Backlog and progress tracking approaches are minor variations of XP practices, and so simple that they are well within the XP spirit of "do the simplest thing that could possibly work."

Scrum's 30-day timeboxed iteration length is not completely consistent with XP, which prefers shorter—even one-week—iterations.

Mike Beedle, one of the original Scrum contributors, has developed **XBreed**—a combination of Scrum and XP practices applied (at least originally) to the creation of reusable components in the context of a concurrent multi-project development.

see www.xbreed.net

Adoption Strategies

As always, coaching by an experienced method expert on the first project is recommended.

1. In contrast to the recommended gentle, pilot-project adoption strategy of UP (for example), the Scrum creators encourage organizations to first adopt it on their single most difficult and critical project. This brave advice underscores the Scrum creators' view that it is strong medicine with a high success rate. They feel the crucible of a critical project best spurs real change to the new values and practices of Scrum.

2. After the first project is underway, but not before the second iteration (so that all the practices have been practiced), extra-project management and potential customers may be invited to observe Scrum Meetings, Sprint Planning, and Sprint Reviews.
3. Second-generation Scrum projects can start before the completion of the first, although the first should be given some time to "ripen," such as three completed iterations. The Product Owner, Scrum Master, and some team members of the new projects will benefit from attending some of the first project's meetings (daily, planning, review) shortly before embarking on their new project. Coaching is useful—during the first iteration—for the new Product Owner and Scrum Master by those on the first project.
4. Jeff Sutherland, one of the Scrum creators and a VP Engineering or CTO at several organizations, recommends ultimately expanding Scrum practices to the highest levels of the development organization. Every level is team based. Projects hold daily Scrums, including a daily Scrum of Scrums among Scrum Masters of subsystem teams. Project representatives in a product line family meet weekly for a Scrum, and upper management holds monthly Scrums.

Fact versus Fantasy

First, a standard disclaimer:

Process is only a second-order effect. The unique people, their feelings and qualities, are more influential.

Scrum practitioners do not report significant variation from the ideals of Scrum compared to its concrete use, presumably due to the relatively small and unambiguous set of practices. The most commonly reported reality checks are the encroachment of non-iteration work on to team members, and attempts by management to direct or organize the team, or solve—unasked—its problems.

Scrum iterations have also failed when the Scrum Master does not regularly reinforce the project vision and Sprint goals, and the team drifts.

Strengths versus "Other"

Strengths

- Simple practices and management workproducts.
- Individual and team problem solving and self-management.
- Evolutionary and incremental requirements and development, and adaptive behavior.
- Customer participation and steering.
- Focus.
- Openness and visibility.
- Easily combined with other methods.
- Team communication, learning, and value-building.
- Team building via the daily Scrum, even if not in common project room.

Other[4]

[4] Could be viewed as a weakness, strength, or deliberate desirable exclusion, depending on point of view.

- Minimal guidance within disciplines other than project management (e.g., programming). That is, Scrum's emphasis is the lifecycle and project management aspects of development, rather than—for example—software or requirements engineering techniques.

  - For example, the Product Owner has the domain knowledge and requirements vision. But, they will describe a function in only brief terms in the Product Backlog. How to transfer and expand this domain knowledge or requirement? Scrum does not address such issues related to workproducts, requirements analysis, and so forth.

- Many projects will need some documents. Scrum does not define what these may be, and thus each project may create ones with similar intent, but varying names and content. In other

words, no common, standard workproducts that are shareable with common names across projects. This impedes reuse of workproducts and impedes a common workproduct vocabulary in larger organizations.

History

The roots of Scrum are found in a well-known article summarizing common best practices in 10 innovative Japanese companies, "The New New Product Development Game," *Harvard Business Review*, Jan 1986, by Takeuchi and Nonaka. It introduced the terms *Sashimi* (slices) for IID, and *Scrum* for the adaptive and self-directed team practices. The name was taken from the game of rugby, for the adaptive team behavior moving a ball up the field.

Jeff Sutherland is one of the Scrum creators and was VP at Easel Corporation in 1994 when he introduced some of its practices; he had read the article. He was also influenced by a report on a hyper-productive project at Borland Corporation that effectively used structured daily meetings [Coplien94]. In 1995 Ken Schwaber worked with Sutherland at Easel on the formalization of Scrum. Their results were described in a workshop paper [Schwaber95]. In 1996 Sutherland joined Individual Inc., and asked Ken Schwaber to assist in the adoption of Scrum ideas. Schwaber refined and extended Scrum, in collaboration with Sutherland, into the versions ultimately described in [BDSSS98] and [SB02].

What's Next?

The next chapter covers XP, another popular agile method. That's followed by chapters on UP and Evo. Finally, there are chapters on more practice tips and a FAQ.

Recommended Readings

The bible of Scrum is *Agile Software Development with Scrum*, primarily by Schwaber, with contributions from Beedle and Sutherland.

Several books and articles influenced the Scrum creators, and are worth mention as recommended readings because of their widespread or seminal influence:

- "The New New Product Development Game," *Harvard Business Review*, Jan 1986, by Takeuchi and Nonaka. Discusses common best practices of innovative Japanese companies.
- *Wicked Problems, Righteous Solutions*, by DeGrace. Discusses why the waterfall is unsuitable for most software projects, and explores various iterative feedback-based practices for development.
- *Hidden Order: How Adaptation Builds Complexity*, 1995, and *Emergence: From Chaos to Order*, 1998, by Holland. Both discuss the dynamics and value of adaptive, self-organizing groups to solve problems.

Chapter 8. Extreme Programming

It's easy to have a complicated idea.

It's very very hard to have a simple idea.

—Carver Mead

OVERVIEW

- Classification of XP.
- Workproducts, roles, and practices.
- Common mistakes, adoption and process mixtures, strengths and weaknesses.

Extreme Programming (**XP**) is a well-known agile method; it emphasizes collaboration, quick and early software creation, and skillful development practices. It is founded on four values: communication, simplicity, feedback, and courage. In addition to IID, it recommends 12 core practices:

1. Planning Game

2. small, frequent releases

3. system metaphors

4. simple design

5. testing

6. frequent refactoring

7. pair programming

8. team code ownership

9. continuous integration

10. sustainable pace

11. whole team together

12. coding standards

Method Overview

Classification

In terms of cycles and ceremony, XP classification is illustrated in Figure 8.1. For average projects, the recommended length of a timeboxed iteration is between one and three weeks—somewhat shorter than for UP or Scrum.

Figure 8.1. XP on the cycle and ceremony scale.

cycles and ceremony

XP is low on the ceremony scale; it has only a small set of a predefined, informal workproducts, such as paper index cards for summarizing feature requests, called **story cards**.

A refreshing quality of the original XP description was the statement of known applicability: It had been proven on projects involving roughly 10 developers or fewer, and not proven for safety-critical systems. Nevertheless, it has been more recently applied with larger teams. Consequently, in terms of the Cockburn scale, XP perhaps covers the cells shown in Figure 8.2.

Figure 8.2. XP on the Cockburn scale

Cockburn scale

Introduction

XP[1] [Beck00], created by Kent Beck, is an IID method that stresses customer satisfaction through rapid creation of high-value software, skillful and sustainable software development techniques, and flexible response to change. It is aimed at relatively small team projects, usually with delivery dates under one year. Iterations are short—usually one to three weeks.

[1] Some writers capitalize the full name as "eXtreme Programming" but Beck does not.

As the word *programming* suggests, it provides explicit methods for programmers, so they can more confidently respond to changing requirements, even late in the project, and still produce quality code. These include test-driven development, refactoring, pair programming, and continuous integration, among others. In contrast to most methods, some XP practices truly are adopted by developers—they sense its practical programmer-relevant techniques.

XP is very communication- and team-oriented; customers, developers, and managers form a team working in a common project room to quickly deliver software with high business value.

XP is distinctive in not *requiring* detailed workproducts except for program code and tests. However, it doesn't disallow other detailed workproducts. Although all evolutionary methods avoid detailed up-front specifications and plans that span the entire release cycle, most of these methods encourage writing down details for at least the next iteration. In contrast, XP emphasizes *oral communication* for requirements and design. For example, a feature is summarized "Find lowest fare" on a handwritten paper index **story card**. Then, when work starts on the feature, the programmers learn details by talking with the customers working full-time in the project room. This may sound disorganized or naive, but Beck is experienced and well aware of the implications of sloppy requirements. Instead, XP is posing this interesting question:

Is there a sane and disciplined way to quickly succeed—on typical small projects—by focusing on code and tests, while avoiding most other documentation overhead?

XP's premise isn't hacker code-and-fix programming; rather, its premise is that there is a new, structured, and sustainable way to succeed with a focus on rapid code production and oral communication, while avoiding most other overhead. To reiterate, XP is not hacking. Quite the contrary, an XP project involves constant practice of highly disciplined—yet agile—software development practices and values.

XP consultant Don Wells explains the influence of the XP values [Wells01]:

XP improves a software project in four essential ways; communication, simplicity, feedback, and courage. XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology.

There is a considerable set of practices in XP: 12 core practices and many ancillary ones. Speaking of these, Wells writes [Wells01]:

It is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen.

Many of these practices work *in synergy*, and thus *it is risky to customize XP by removing some elements.* For example, XP aims to produce software quickly by—in part—avoiding detailed requirements documentation. But, this is compensated by the practice of onsite customers sitting in the project room to fill in the details.

The word *extreme* in XP comes from Beck's conviction that it is hard to have too much of a good thing in software development. That is, take known good practices and "turn the dial up to 10," or to extreme levels. For example:

- Testing is good, so write unit tests for (almost) all code, and acceptance tests for all features.
- Code reviews are good—even better close to creation date—so do code reviews in real time and all the time via pair programming.
- Frequent integration of code across all team members is good, so do it 24/7 with an automated, continuous integration process on a dedicated build machine.
- Short iterations and early feedback are good, so make iterations one or two weeks long, if possible.
- More customer involvement is good, so bring customers into the project full-time, sitting in the common project room.
- Communication is good, so have everyone sit together, pair program, include onsite customers, and involve the customer frequently in planning, steering, and evaluation.

Lifecycle

EXPLORATION	PLANNING	ITERATIONS TO FIRST RELEASE	PRODUCTIONIZING	MAINTENANCE
Purpose: - Enough well-estimated story cards for first release. - Feasibility ensured.	**Purpose**: - Agree on date and stories for first release.	**Purpose**: - Implement a tested system ready for release.	**Purpose**: - Operational deployment	**Purpose**: - Enhance, fix. - Build major releases.
Activities: - prototypes - exploratory proof of technology programming - story card writing and estimating	**Activities**: - Release Planning Game - story card writing and estimating	**Activities**: - testing and programming - Iteration Planning Game - task writing and estimating	**Activities**: - documentation - training - marketing - . . .	**Activities**: - May include these phases again, for incremental releases.

Some comments on the XP lifecycle phases defined by Beck:

1. Like many projects, XP can start with exploration. Some story cards (features) may be written, with rough estimates.
2. In the Release Planning Game, the customers and developers complete the story cards and rough estimates, and then decide what to do for the next release.
3. For the next iteration, in the Iteration Planning Game, customers pick stories to implement. They choose stories—and thus steer the project—based on current status, and their latest priorities for the release. Developers then break the stories into many short, estimated tasks. Finally, a review of the total estimated task-level effort may lead to readjustment of the chosen stories, as XP does not allow overworking the developers with more than they can do based on "family-friendly" work days, such as an eight-hour day. Overtime is seriously discouraged in XP; it is viewed as a sign of a dysfunctional project, increasingly unhappy people, and dropping productivity and quality.

4. Developers implement the stories within the agreed timeboxed period, continually collaborating with customers (in the common project room) on tests and requirement details.
5. If not finished for release, return to step 3 for the next iteration.

Workproducts, Roles, and Practices

Roles

ROLES
Customer
Customer
- writes stories and acceptance tests
- picks stories for release and for iteration
Development
Programmer
- writes tests, designs, code
- refactors
- identifies tasks and estimates
Tester
- helps customer write and develop tests
Management
Coach
- process conscience
- process customizing
- intervention
- teaching
Tracker
- collect metrics
- tell progress
- feedback on poor estimates
Other
Consultant
- technical consulting
- coaching

Practices

Core Practices

Practice	Description
Whole team, or Onsite customers	The whole team—programmers and customers—work together in a common project room. One or more customers sit more-or-less full time with the team; they are expected to be subject matter experts and empowered to make decisions regarding requirements and their priority. The customer contribution includes detailed explanation—to the programmers—of the features briefly summarized on story cards, Planning Game participation, clarification, and writing acceptance tests in collaboration with a programmer. The purpose is in response to the consistent project failure research indicating that more customer involvement is paramount to successful projects. The first release of XP spoke of only one onsite customer; this has been recently revised to emphasize a group of customers.
Small, frequent releases	Evolutionary delivery. Not applicable to all projects. Not to be confused with organizing one release cycle into many short iterations.

Practice	Description
Testing: Acceptance testing & Customer tests	Testing practices in XP are very important. All features must have **automated acceptance (functional) tests**. All tests (acceptance and unit) must run with a binary pass/fail result, so that no human inspection of individual test results is required. The acceptance tests are written with collaboration of the customer—they define a testable statement of what acceptance means. This is called **Customer Tests** in XP.
Testing: Test-driven development and unit testing	Unit tests are written for most code, and the practice of **test-driven development** (and **test-first development**) is followed. This includes the practice that the unit test is written by the programmer *before* the code to be tested. It is a cycle of test Õ code, rather than code Õ test. Usually, the open-source XUnit testing framework family (such as JUnit) is applied (see www.junit.org). All acceptance and unit tests are automatically run repeatedly in a 24/7 continuous integration build and test cycle. See p. 292 for a detailed example.
Release planning game	The **Release Planning Game** goal is to define the scope of the next operational release, with maximum value (to the customer) software. Typically a half-day one-day session, customer(s) write **story cards** to describe features, and developers estimate them. There may also exist story cards from prior exploration phase work. The customer then chooses what's in the next release by either 1) setting a date and adding cards until the estimate total matches the time available, or 2) choosing the cards and calculating the release date based on their estimates.
Iteration planning game	The **Iteration Planning Game** goal is to choose the stories to implement, and plan and allocate tasks for the iteration. It happens shortly before each new iteration (1–3 weeks in length). Customer(s) choose the story cards for the iteration. For each, programmers create a **task list** (on cards or whiteboard) that fulfill the stories. This is followed by a volunteering step in which the programmers choose a set of tasks. They then estimate their task lengths. If tasks are not estimated in the half-day to two-day range, they are refactored.
Simple design	Avoid speculative design for possible future changes. Avoid creating generalized components that are not immediately required. The design should avoid duplicate code, have a relatively minimal set of classes and methods, and be easily comprehensible.
Pair programming	All production code is created by two programmers at one computer; they rotate using the input devices periodically. Pairs may change frequently, for different tasks. The observer is doing a real-time code review, and perhaps thinking more broadly than the typist, considering tests and so forth. Certainly, team productivity is not simply a function of the number of hands typing—it is more nuanced. The XP claim is that the combination of cross learning, the peer pressure of more disciplined practice observance and more hours actually programming than procrastinating, defect reduction due to real-time code review, and the stamina and insight to carry on when one programmer is stuck, all add up to an overall team improvement.
Frequent	Refactoring in the XP context is the continual effort to simplify the fine-

Practice	Description
refactoring	grained code and larger design elements, while still ensuring all tests pass. That is, cleaning the code and design, without changing functionality. There is supposed to be "lots" of refactoring in XP. This practice is also known as **continuous design improvement**. The goal is minimal, simple, comprehensible code. It is achieved by small change steps, verifying tests after each, and ideally the use of refactoring tools, now available in some IDEs.
Team code ownership	Any pair of programmers can improve any code, and the XP value system is that the entire team is collectively responsible for all the code. The value of "it's her code, and her problem" is not endorsed; rather, if a problem or chance to improve is spotted, it's the spotter's responsibility. A related goal is faster development by removing the bottleneck associated with change requests in an individual code ownership model. The obvious danger of modifying code one did not originally write is ameliorated in XP by some of the other practices: The guaranteed-present unit and acceptance tests running within an automated continuous integration build process inform you if you broke the code, your pairing partner brings another set of eyes to the change, and common adherence to coding standards ensures all the code looks the same.
Continuous integration	All checked-in code is continuously re-integrated and tested on a separate build machine, in an automated 24/7 process loop of compiling, running all unit tests and all or most acceptance tests. There are several open-source tools for this, built on the ubiquitous Ant technology, including CruiseControl and Anthill.
Sustainable pace	Frequent overtime is rightly considered a symptom of deeper problems, and doesn't lead to happy, creative developers, healthy families, or quality, maintainable code. XP doesn't support it—rather, it promotes "no overtime."
Coding standards	With collective code ownership, frequent refactoring, and regular swapping of pair programming partners, everyone needs to follow the same coding style.
System metaphors	To aid design communication, capture the overall system or each subsystem with memorable metaphors to describe the key architectural themes. For example, the C3 payroll system was described in terms of an assembly line of checks with posting rule "machines" operating on them, extracting money from different "bins." Many have reported this the least necessary practice.

Workproducts

Story Cards— Figure 8.3 shows a simple story card: A handwritten note on a paper index card. During the Planning Game, many of these are written. This spartan example was chosen to emphasize the minimalist approach to recorded requirements that XP encourages.[2]

[2] In fact, when XP expert Ron Jeffries reviewed this chapter, he felt even the number "3" on this real-example card was excessive.

Figure 8.3. sample story card

The story cards record **user stories**: features, fixes, or nonfunctional requirements that the user wants. There can even be a story card to create documentation. Stories are usually in the one-day to three-week range of estimated duration. Contrary to some misunderstanding, XP stories are *not* use cases or scenarios. They usually represent features. Note that XP prefers a feature-driven approach to describing requirements rather than the use-case-driven approach that UP promotes.

In XP, oral communication is preferred, and the story card purpose is *not* to detail the user story, but to jot a summary, make references to other documents, and in general, to view the card as "a promise to talk" (in Alistair Cockburn's words) with the customer who wrote it, by the developers implementing it. Since *whole team together* is an XP practice, the card donor should be readily available. XP coaches vary on their advice regarding granularity for estimation. Some say stories can be in the two-day to two-week range of effort, others recommend stories be estimated in units of one, two, or three weeks, but not in finer person-day units.

Task List— During an Iteration Planning Game, the team convenes around a whiteboard, and generates a list of tasks for all the stories chosen for the iteration. Another popular alternative is to generate individual task cards. Once a task is chosen by a volunteer, they enter an effort estimate (in *ideal engineering hours*)—tasks should be in the 1–2 day range.

Visible Graphs— The idea is to easily communicate—to the team—something they find useful to measure. Measure at least one thing. XP doesn't mandate what that should be, though known examples include acceptance tests defined and passing, story progress, and task progress.

Other Practices and Values

- **Onsite customer proxies**— Many groups wishing to apply XP cannot find full-time "ultimate" customers to work in the project room. For example, consider a new internal system for (very busy) commodity traders. And this problem is common for commercial products for an external market. The common solution is to designate **customer proxies** that do join the team in the project room, have good (though not as ideal as true customer) knowledge of the domain and requirements, and that represent the ultimate customers. If proxies are used, it is important that the true customers at least participate in end-of-iteration demos, and preferably, in the Planning Games.
- **Customer on call**— When the onsite customer is not present, arrange matters so that the customer representative is committed to fast access, such as via a mobile phone.
- **Embrace change**— The overarching attitude that XP promotes is to embrace rather than fight change, in the requirements, design, and code, and be able to move quickly in response to change.

- **Only by volunteering (accepted responsibility)**— Tasks are not assigned to people. Rather, during the Iteration Planning Game, people choose or volunteer for tasks. This leads to a higher degree of commitment and satisfaction in the self-accepted responsibility, as has been explored in [DL99].
- **Very light modeling**— XP encourages programming very early and does take to "extreme" the avoidance of up-front design work. Any more than 10 or 20 minutes of design thinking (e.g., at the whiteboard with sketches or notes) before programming is considered excessive. Contrast this with Scrum or the UP, for example, where a half-day of design thought near the start of an iteration is acceptable.
- **Minimal or "just enough" documentation**— With the goal of getting to code fast, XP discourages writing *unnecessary* requirements, design, or management documents. The use of small paper index cards is preferred for jotting brief descriptions, as are verbal communication and elaboration. Note that the practice of "avoiding documentation" is compensated by the presence of an onsite customer. XP is not anti-documentation, but notes it has a cost, perhaps better spent on programming.
- **Metrics**— XP recommends daily measurement of progress and quality. It doesn't mandate the exact metrics, but to "use the simplest ones that could work." Examples include numbers of completed tasks and stories, and number and success rate of running tests.
- **Visible wall graphs**— The collected metrics are daily updated on wall graphs for all to easily see.
- **Tracking and Daily Tracker**— The regular collection of task and story progress metrics is the responsibility of a tracker. This is done with a walk-about to all the programmers, rather than email; commenting on this—and very telling of the XP attitude—Ron Jeffries (one of the XP founders) said, "XP is about people, not computers." Test metrics can be automatically collected by software.
- **Incremental infrastructure**— XP recommends (as do the other iterative processes) that the back-end infrastructure (for example, a persistence layer) not be the main focus of implementation in the early iterations, but rather, only enough is implemented to satisfy the user-functional requirements of each iteration.
- **Common project room**— XP projects are run in a common project room rather than separate offices. Pair programming tables are in the center of the room, and the walls are clear for whiteboard and poster work. Of course, people may have a private space for private time, but production software development is a team sport in XP.
- **Daily stand-up meeting**— As in Scrum, there is a daily short stand-up (to keep it short) meeting of status.
- **Ideal engineering hours (IEH)**— Task estimates—and possibly story estimates—are done in terms of IEH, or uninterrupted, dedicated, focused time to complete a task.
- **Story estimates**— To estimate larger stories, some XP practitioners recommend using only coarse values of one, two, or three week durations rather than IEH or day-level estimates.

Values

Beck's XP description is noteworthy in the process world for being perhaps the first to explicitly state the values that underly the attitude and practices on a healthy XP project. To quote Beck on the relationship of values and practices:

The [practices] are what you do. The values are how you decide if you are doing it right.

They are:

- **Communication**— XP accepts the widely appreciated observation that problems in communication underly most project difficulties. Communication between programmers is promoted through pair programming, the daily stand-up meeting,

and the Planning Game. Communication is promoted through customer involvement in writing acceptance tests and the Planning Game.

- **Simplicity**— or, "Do the simplest thing that could possibly work." This applies not only to the design of software, but to other things such as requirements and project management "tools." For example, XP encourages the use of simple paper index cards to write a brief description of feature and task requests, if more formal artifacts can be avoided. In terms of software design, XP avoids speculative design for possible change ("future proofing") or the creation of more generalized components that aren't immediately justified by current requirements.
- **Feedback**— This value drives quality and adaptation. Feedback in the short term is driven by the XP practice of test-first development with unit tests. It also comes from the practice of continuous integration; a broken build tells the story. When a customer writes a story card (a feature description), programmers immediately estimate it, so the customers know the effort. The practice of daily tracker provides feedback to the team and customer on progress for the iteration. On a longer scale, the customer written acceptance tests provide feedback. Short iterations give the customer the chance to see (and perhaps operate) an incrementally evolved partial system, and clarify or redirect the requirements. And the practice of frequent operational releases generates feedback from production users.
- **Courage**— The courage to develop fast, and make changes fast emerges from the support of the other values and practices, and modern technologies. For example, without a massive set of unit tests, acceptance tests, and continuous integration, making deep "architectural" changes in the code base is tricky business—difficult to tell what will break. But the presence of these, combined with a simple design, very clean code refined from frequent refactoring, and modern automated refactoring tools provided by many IDEs enables more rapid and radical change.

Common Mistakes and Misunderstandings

or, How to Fail with Extreme Programming

Error: No onsite customer; rather, use specifications written for the next iteration— It is normal and acceptable to create written specifications for the iteration (for example, two use cases) if the adopted method is Scrum, UP, Evo, etc. An iterative or agile project that takes this approach can work well, but is better characterized as being based on another method (e.g., the UP) that allows written evolutionary requirements. It is a cornerstone of XP to avoid detailed specifications, use oral communication of the requirements, and onsite customers.

Error: Applying a subset of uncompensated practices; customizing before trying— Many XP practices work as a synergistic whole, and it is a mistake to remove one that compensates or supports another. For example, collective code ownership isn't feasible without testing, continuous integration, and coding standards. Minimal requirements documentation isn't possible without an onsite customer. Frequent refactoring doesn't work without the tests. That is why Beck, while not wishing to be rigid, in general recommends adoption of all or most of the core practices. To quote Beck, "Do all of XP before trying to customize it."

Error: XP is just iterative development + minimal documentation + unit testing— Although there is some flexibility in what practices must be present to define an XP project, it is more than this—common to several IID methods. One could run a Scrum, UP, DSDM, or other evolutionary methods primarily with just these practices. XP is characterized by a larger set of practices, including pair programming, onsite customers, customer written acceptance tests, and more.

Error: Not writing the unit tests first— Test-first development has a more subtle dimension than first glance, and is an important XP practice. Writing the tests *first* influences how one conceives, clarifies, and simplifies the design. Test-first has an

interesting psychological quality of satisfaction: I write the test, and then I make it succeed. There is a feeling of accomplishment that sustains the practice.

Error: Customer doesn't decide— XP is customer driven; they need to decide what the acceptance criteria are (via tests), and what stories go in a release and iteration.

Error: No customer-owned tests— Ron Jeffries has said, "The failure to have customer-owned acceptance tests [in each iteration] is one of the most common mistakes in XP."

Error: Minimal refactoring— The XP avoidance of design thought before programming is meant to be compensated by a relatively large refactoring effort, such as 25% of total effort applied to refactoring. Beck's point is that one can't avoid both design thought before programming and refactoring; it's either/or.

Error: Must have only one onsite customer— The original XP books talked of *one* onsite customer in the project room. Beck and the XP leaders have since refined this to emphasize that the customer team needs to be considered as a whole, with requirements coming from perhaps many customers participating in the Planning Games. Thus, they have replaced the practice "onsite customer" with "whole team together."

Error: Many fine-grained task cards— Most task cards should be in the one or two-day range of effort. Most in the "few hours" range creates unnecessary information management.

Error: Pairing with one partner too long— XP pairing changes frequently, often in two days or less. Variation also helps spread the learning.

Error: Customer or manager is tracker— Programmers will feel awkward reporting slow progress.

Error: Not integrating the QA team— Many organizations have a separate Quality Assurance team, used to having a completed system "thrown over the wall" to them. One or more dedicated QA people need to be brought onto the project full-time—the whole team practice—usually to write the acceptance tests in collaboration with the customer.

Error: Post-development design documentation is wrong— XP isn't anti-documentation, but prefers programming if that's sufficient to succeed. XP can support the creation of design documentation to reflect the existing code base, once the program is complete. As always, the simplest approach that can work is the goal, such as video recording an explanation.

Error: Diagramming is bad— Although XP advice is minimal modeling or diagramming, such as 15 minutes before programming, a very little is acceptable.

Error: Only young pair programmers— Some XP projects have suffered in the pair programming practice when most of the developers were quite young, without the patience or maturity to handle working closely with others.

Error: Pairing newbies— One of the two partners should have pair programmed before.

Error: One partner going too fast— When pair programming, the quicker or more experienced partner needs to be sensitive to the speed or comprehension differential of their partner, and slow down their activities and explanations.

Error: Observer can't easily see the monitor— Self-explanatory, but a surprisingly frequent problem.

Error: Not willing to learn; not willing to explain— For successful pair programming, an attitude of openness to learning and of explaining yourself is required.

Error: Full team (including customers) not briefed in XP and its motivations— Self-explanatory.

Error: Dissenter on team— XP is about communication and collaboration, and a culture of development; lone programmers who don't wish to accept it can impair the team culture and project progress.

Error: Stand-up meeting too long or unfocused— Keep it below 20 minutes, and on status of tasks, not a discussion of design and requirements.

Error: Lumping into one big "bug fix" story— As defects accumulate, don't group them into one task or story card; keep them with their original related cards.

Error: No dedicated acceptance tester— One is needed to work with the customer on transforming their acceptance criteria into runnable tests. An exception is very small projects—the tester may fulfill that role part-time.

Error: Onsite customer and Big Boss aren't aligned— XP talks of the Big Boss, or the person ultimately owning or responsible for the project goals and milestones. These two stakeholders need to be in agreement.

Error: Customer writing acceptance tests isn't the reviewer of their execution— A classic problem of serving different masters.

Error: Iterations too long— XP iterations should be 1–3 weeks.

Error: Iterations aren't timeboxed— It is a misunderstanding to let the iteration length expand when it appears the goals can't be met within the original time frame. Rather, the preferred strategy is usually to remove or simplify goals for the iteration. And, analyze why the estimates were off.

Error: Iteration doesn't end in an integrated and tested baseline— An iteration doesn't just successfully end on the end date. The goal is that all the software has been integrated, tested, and baselined.

Error: Each iteration ends in a production release— The baselined software produced at the end of an iteration is an *internal* release rather than shippable code. It represents a subset of the final production release, which may only be ready after a dozen or more short iterations. It is true that in iterative development one goal is that each iteration release is stable enough to potentiality release to production, if necessary. However, this is not the normal intent of an iteration release.

Error: Predictive planning— It is a misunderstanding to create, *at the start of the project*, a believable plan laying out exactly how many iterations there will be for a long project, their lengths, and what will occur in each. This is contrasted with the agile approach: adaptive planning. The XP team and customer plans the next iteration, and then planning adapts iteration by iteration, based on current feedback.

You Know You Didn't Understand XP When...

Some of the key misunderstandings expressed as a checklist:

- You think you should customize the choice of practices without having first applied them all.

- You think "doing XP" means to avoid the waterfall model and develop iteratively, or just to avoid documentation, or just to write some unit tests.
- Customers are not involved in the Planning Game, creating acceptance tests, or reviewing the iteration results.
- You create a plan laying out how many iterations there will be for the project, their lengths, and what will occur in each.

Sample Projects

The following projects had significant XP influence:

Large— Atlas leasing system

- Three years, 60+ people, Java technologies, an E100 project, [Schuh01]

- *Fully adopted practices*: simple design, testing, frequent refactoring, collective code ownership, continuous integration

- Pair programming was attempted, but did not stick. There was no onsite customer.

- Prime developer: ThoughtWorks.

Medium— Orca security incident-response

- One year, 25 people, a D40 project, [Morales02]

- *Fully adopted practices*: most practices, with the exception of small, frequent releases as this was a commercial product.

- Prime developer: Symantec.

Small— C3 payroll

- One year, 10+ people, an E20 project [C3Team98]

- *Fully adopted practices*: This was the original project that defined XP, coached by Kent Beck and Ron Jeffries. All practices were adopted.

- Prime developer: Chrysler.

Process Mixtures

XP + Evo

XP values and spirit regarding specifications is not compatible with Evo. XP's value of avoiding written or precise requirements, and preferring oral communication between developers and requirement donors is very different than Evo's emphasis that when a specification is required, it be done so with clarity and measurable qualities.

Evo

Evo specifications

On the other hand, many XP development practices may be consistently applied with Evo, such as test-driven development, pair programming, and so forth.

XP's client-driven adaptive planning is also consistent with Evo. The XP stand-up meeting, common project room, and whole team together supports Evo's feedback goals.

adaptive planning

XP's 1–3 week iteration length is relatively consistent with Evo, which prefers 1–2 week iterations.

XP + Scrum

Most Scrum practices are compatible with XP. The Scrum meeting is a refinement of the XP stand-up meeting (in fact, Beck got the idea from Scrum), using special questions. Both recommend a common project room. The Scrum practice of a demo to external stakeholders at the end of each iteration enhances XP's feedback and communication goals. The Scrum Backlog and progress tracking approaches are minor variations of XP practices.

Scrum

Scrum meeting

Scrum's 30-day iteration length is not consistent with XP—too long.

A Scrum practice is to have only one customer representative, the Product Owner, who is ultimately responsible for the requirements and priorities. But in recent updates to XP, there is an emphasis on collaborating with a *group* of customers—avoiding just a single person. It is nice to have a single customer voice, and it is useful to know and resolve multiple people's goals. XP and Scrum tackle this tension differently, shifting whether development or business is responsible for resolving the conflict.

Mike Beedle, one of the early Scrum practitioners, has explored combinations of XP and Scrum under the name "XBreed."

see www.xbreed.net

XP + UP

Most XP practices are either equal to or specializations of UP practices, and many XP practices can be applied in the context of an overarching UP project. For example, test-first development is a specialization of the UP *continuously verify quality* best practice. The UP does not require or promote unnecessary document creation—all artifacts are optional—and so it is a misunderstanding to assume the methods are fundamentally incompatible. Although speaking of some XP within UP can have conceptual integrity, the opposite is not true, as there are some differences in style and emphasis.

UP

UP practices

One area of difference is in the accepted degree of up-front modeling (diagramming, etc.). For example, within a UP project and a two-week iteration, it is considered acceptable to spend a half-day near the start to consider design ideas "at the whiteboard" before programming. In XP, no more than 10 or 20 minutes before programming is considered suitable.

Another difference is in the goal of the early iterations. In the UP the goal is to identify and drive down the high risks: technical, political, satisfying the customer, and so forth. Although this may happen in the XP, it is not an explicit guiding principle.

A third difference is in requirements specifications. The UP allows and supports the creation of relatively detailed specifications (evolutionarily, over a series of iterations), assuming that an onsite customer is not going to be present. These will usually take the form of use cases and an associated nonfunctional specifications document, created in a series of timeboxed requirements workshops. The idea in the UP is, during the early programming iterations, to have a parallel track of requirements analysis where the majority of requirements are being written, while the development team is also programming something critical. The programming work is meant, in part, to help clarify the requirements work.

XP stories are normally features, rather than use cases. Thus, XP promotes a feature-driven approach to requirements. On the other hand, the UP promotes a use-case-driven approach, although the UP accepts and allows features rather than use cases.

Adoption Strategies

As always, coaching by an experienced method expert on the first project is recommended.

Similar to Scrum, but in contrast to the recommended gentle, pilot-project adoption strategy of UP (for example), XP recommends adoption like this:

1. Pick the worst project or problem.

2. Apply XP until solved.

3. Repeat.

If all the XP practices can't be swallowed at once, Beck recommends starting with:

- whole team together in a common project room
- test-first development
- acceptance tests written/owned by customers
- Planning Game
- pair programming

That said, there are dangers in only adopting a few of the practices, especially if one does not understand how they support each other. Avoidance of up-front design (even on a per-iteration basis) is compensated by frequent refactoring. Frequent refactoring is supported by continuous integration, and test-first development. And so forth.

Introducing customers to this new engaged approach is a challenge. The key is to help them see the early, tangible business value that they want, and emphasize that they will be

steering the team to meet their needs in short cycles. An XP goal is to so delight the customer with this new-found control and responsiveness, that after a few iterations they will love the process.

On the common problem of customers wanting more and more in an iteration, one technique is to exploit the physical nature of story cards: Once the cards have been estimated and chosen for an iteration—and therefore consuming all available development resources—they are laid out on a table. Clearly, with the "no overtime" rule of XP, adding a new card means an existing one must be physically removed. This visual and tangible impact teaches a clear lesson.

If you can't get a customer into the room, what to do? First, look for another related representative, such as a product manager. If no luck, establish the closest possible communications. Visit them frequently, use a mobile phone, spend time at their job, have them use an instant messenger service to simulate the feel of close communication, have them attend the Planning Games, and show lots of demos.

Programmers will adopt most of the practices without concern, except pair programming. XP recommends not inviting programmers opposed to the idea to an XP project. Do not put only young programmers together; the maturity and patience of some older developers is necessary to make pairing work. Ensure the pairs mix regularly, the typing developer rotates frequently, and that people are learning from each other; i.e., sharing with their partners what they know, and what they are thinking. If pairs aren't frequently talking together, something isn't working.

For XP, the physical environment must change: open common room with development stations near the center, and the walls exposed for visible graphs, sketching, and so forth. And, the stations need to support pair programming. Ensure the non-typing partner can easily see the monitor.

Since pair programming implies lots of talk, a culture of quiet talking needs to be encouraged.

Fact versus Fantasy

Reading the XP method practices can create the impression that they are a silver bullet, but of course, they are not. As always,

Process is only a second-order effect. The unique people, their feelings and qualities, are more influential.

One fantasy regarding XP adoption is found in groups that believe by just adopting iterative and evolutionary development and avoiding up-front specifications, they are "doing XP." Likewise with unit testing, working in a common project room, and so forth. Although data is still sketchy, it seems that many of the projects claiming to be doing XP are simply applying some iterative and evolutionary practices common to many IID methods (such as short iterations), and the group mistakenly believes these are unique XP ideas.

Probably the most common XP fantasy is getting onsite customers. It seems to be rare as hen's teeth to achieve this. Also, there is no shortage of so-called "XP" projects one investigates that could not arrange pair programming, which Beck considers one of the basics of an XP project.

Resistance to pair programming is perhaps the most common issue among developers. Some just don't want to do it.

It is also rare to find a common project room, or enough whiteboards.

Test-first development, early acceptance tests defined with customers, constant refactoring, and continuous integration are all widely confirmed as sustainable, excellent practices.

Strengths versus "Other"

Strengths

- Practical, high-impact development techniques, many of which are easily and sustainably adopted by developers (e.g., continuous integration, test-driven development).
- Emphasizes customer participation and steering.
- Evolutionary and incremental requirements and development, and adaptive behavior.
- Programmers estimate the tasks they have chosen, and the schedule follows this, not vice versa (i.e., scheduling is rational).
- Emphasizes communication between all stakeholders.
- Emphasizes quality through many practices. Test-first development, continuous integration, and team code ownership are examples.
- Clarifies what is an acceptable system by requiring the customer to define the acceptance tests.
- Daily measurement, and developers are involved in measuring and defining what to measure.
- Every iteration, developers get practice (during the Planning Game) identifying tasks and estimating them, leading to improvement in these vital skills.
- Frequent, detailed reviews and inspections, as all significant work is done in pairs. Inspection is strongly correlated with reduced defect levels.

Other[3]

[3] Could be viewed as a weakness, strength, or deliberate desirable exclusion, depending on point of view.

- Requires the presence of onsite customers (or proxies). This is often not possible, and their absence makes difficult or impossible the practice of "oral requirements" and using short story cards. XP has no standard solution for written requirements. That takes us back to other methods, such as the UP, which have a mechanism for iteratively recording detailed requirements.
- Relies on oral history for knowing the design and requirements details. This has limitations related to quickly helping new members, or scaling to larger projects.
- The XP practices are interdependent and mutually supporting. It isn't really a pick-and-choose process; most need to be done. Yet, people avoid some in the urge to avoid "unnecessary" steps, and thus failure ensues. Then, XP is unfairly criticized.
- No standard way to describe or document the software design as a learning aid.
- Some developers do not want to do pair programming.
- Many projects will need a set of documents other than code. XP does not define what these may be, and thus each project may create ones with similar intent, but varying names and content. In other words, no common, standard workproducts that are shareable with common names across projects. This impedes reuse of workproducts and impedes a common workproduct vocabulary in larger organizations.
- Lack of architecture-oriented emphasis in the early iterations. Lack of architectural design methods. XP advocates claim simple design and refactoring lead to the same goal.

History

In the mid-1980s, Kent Beck and Ward Cunningham worked together in a research group at Tektronix. They founded the idea of CRC cards and (the seminal contribution of) design patterns, while building many Smalltalk systems. The roots of XP come from this collaboration. Eventually, Beck branched into private consulting, slowly (re-)discovering the

various XP practices, such as the value of working in a common room. Cunningham went on to create the popular and unique Web concept of Wiki Webs.

In the mid-1990s, Beck was retained by Chrysler to help with a new Smalltalk-based payroll system, the C3 project. One aim of the project was the education of the staff in object-technology skills; a successful payroll system was desirable, but not the only goal. Beck introduced the majority of practices that became XP, and brought in Ron Jeffries to daily lead and coach the team. Martin Fowler was also invited for some consulting. Primarily led by the vision of Beck, the XP practices coalesced on this project.

Beck says that at its heart, XP is expressing what he learned with and from Cunningham.

There has been some mis-information that the C3 project "failed." In fact, management felt that the team received good object-technology education, and the C3 payroll system did successfully go into production for several thousand employees, but was eventually phased out and the team reassigned, as management—with direction from the new Daimler owners—developed different ideas for handling payroll at the new-found DaimlerChrysler company.

What's Next?

The next chapter presents the practices of the UP, an iterative method with a somewhat different emphasis than XP. Following that, Evo is introduced—one of the first evolutionary methods.

Recommended Readings

There are many XP books but only a few essentials. *Extreme Programming Explained* by Kent Beck is required reading. A good practical companion by three members of the original C3 team is *Extreme Programming Installed*.

Supporting or related texts that are recommended include:

- *Test-Driven Development: By Example*, by Kent Beck. Teaches the essentials of this key XP practice.
- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler. The bible on refactoring skills.
- *Peopleware*, by Tom DeMarco and Tim Lister. Discusses some of the people-side issues that inspired Beck in XP.
- *The Deming Management Method*, by W. Edwards Deming and Mary Walton. Discusses the critical role of personal pride in workmanship. This also influenced Beck and XP.
- *Toyota Production System: Beyond Large-Scale Production*, by Taiichi Ohno. This book—by the creator of the Toyota method—on "lean manufacturing" is something of the physical-goods equivalent to XP for software. Although Beck did not read this work until after creating XP, he has since highly praised it for capturing many of his goals and values in XP.
- "Episodes: A Pattern Language for Competitive Development" in *Pattern Languages of Program Design 2*. Article by Ward Cunningham, edited by John Vlissides. Cunningham presents some of the key ideas that became XP.

Chapter 9. Unified Process

The true measure of a man is how he treats someone who can do him absolutely no good.

—Samuel Johnson

OVERVIEW

- Classification of UP.
- Workproducts, roles, and practices.
- Common mistakes, adoption and process mixtures, strengths and weaknesses.

The **Unified Process** (**UP**) is a popular iterative process framework, particularly its refinement in the **Rational Unified Process** or **RUP**. Some of the UP key practices and guidelines illustrate its spirit:

UP versus RUP

- Develop in short timeboxed iterations.

practice details

-
- Develop the high-risk and high-value elements (for example, the core architecture) in early iterations, preferring re-use of existing components.
- Ensure that you deliver value to your customer.
- Accommodate change early in the project.
- Work together as one team.

The UP organizes iterations within four phases. The **elaboration phase** iterations emphasize programming the risky, core architecture, **construction phase** iterations build the remainder.

Method Overview

Classification

In terms of cycles and ceremony, UP classification is illustrated in Figure 9.1. For average projects, the recommended length of a timeboxed iteration is between two and six weeks—somewhat longer than XP, for example.

Figure 9.1. UP on the cycles and ceremony scale.

cycles and ceremony

Perhaps the most noteworthy quality of the UP, in comparison with other popular IID methods, is its ability to scale up and down on the ceremony spectrum, with *optional* support for higher degrees of formality and documentation. Contrary to some misunderstanding, the UP encourages a relatively light footprint in terms of ceremony, although in general it recommends more documentation and modeling than in XP. It offers a set of around 50 optional workproducts for many contingencies.

In terms of the Cockburn classification, the UP covers all cells. See Figure 9.2.

Figure 9.2. UP covers all cells on the Cockburn scale

Cockburn scale

The UP can be applied to small three-person projects with no more than loss of comfort, and up to hundreds of developers working on life-critical systems. For example, the core UP practices were applied on the Canadian Air Traffic Control System project, involving many hundreds of developers.

Introduction

The UP [JBR99] is an **iterative process framework**—a general process description that can and should be refined into a more detailed process description for an organization or project, such as the RUP [Kruchten00].[1] A UP specialization may itself be a more detailed process *framework* (as is the RUP) or a concrete process description for one particular project.

[1] IBM bought Rational in 2003, maintaining the "RUP" branding.

The UP is more of a defined process, and more broad and ambitious than the other iterative processes described in this text. However, all activities and workproducts, and their ordering, are optional and arbitrary.

See "UP as a Heavy, Defined Process versus an Agile UP Approach"

The RUP refinement of the UP is both a process framework (for creating specific processes), and a licensed product. As a product, it is a set of around 100 core Web pages of process description—with several thousand detailed supporting pages—and with templates for its artifacts. A customizing "RUP Builder" tool can configure the set of Web pages to describe your organization's specific tailoring of the RUP. Many organizations purchase and install a version on their intranet as a learning aid, quality assurance aid, and template resource. See Figure 9.3. It is not required to own the RUP product to adopt and apply the UP's general ideas or practices.

Figure 9.3. sample Web page from the RUP product

The UP defines a set of approximately 50 optional (non-software) **artifacts** (workproducts), such as the **Vision**. A particular project may create zero or more, and "less is better" is a guiding rule, although a couple of workproducts are usually recommended, including a **Vision** and **Risk List**. Note that workproducts in the UP are *information abstractions* rather than necessarily computer documents; for example, the Risk List could be realized with a poster on the wall of the project room. Nevertheless, the RUP product includes document templates (for example, in HTML) for those wishing to use them.

The workproducts are organized within **disciplines**—such as the Requirements discipline—that define major areas of concern and activity on a software project. See Table 9.1 for a sample.

Table 9.1. sample UP disciplines and workproducts

Discipline	Workproduct	Comment
Requirements	Vision	Summary of stakeholders' key needs and features.
	Use-Case Model	The set of use cases describing the intended functions and environment.
Design	Design Model	An object model describing the hardware and software realization of the use cases in terms of collaborating objects.
	Software Architecture Document	A system overview or learning aid that includes several architectural views.

Table 9.1. sample UP disciplines and workproducts		
Discipline	**Workproduct**	**Comment**
Project Management	Iteration Plan	The goals and tasks for the current or next iteration.
	Risk List	A list of prioritized risks with associated mitigation plans.

In terms of disciplines and iterations, the UP envisions a project approximating Figure 9.4. In each iteration, activity occurs in most disciplines, though the relative efforts vary over time.

Figure 9.4. sample UP disciplines and iterations

The UP requires that it be **tailored** for each project; that is, choosing the set of practices and UP workproducts to create, from the large but optional set available. This unique tailoring is called the **Development Case** of the project; a simple example is shown in Table 9.2. In general, "less is better" is the guideline, and a Development Case is encouraged to contain the minimal set of workproducts needed to address the risks and goals of the project.

Table 9.2. Sample partial UP Development Case						
Discipline	**Techniques**	**Artifact**	**Incep.**	**Elab.**	**Const.**	**Trans.**
		Iteration→	**I1**	**E1..En**	**C1..Cn**	**T1..Tn**
Requirements	one-day timeboxed requirements workshops, prototypes, paper-based UI mock-ups.	Vision	S[a]	r		
		Supplementary Specification	S	r		
Design	Pair designing doing whiteboard sketches captured with camera, test-first design, reverse engineering.	Design Model		s	r	
		SW Architecture Document		s		
Project Mgmt	All Scrum management practices	Risk List	S	r	r	r
Implementation	Pair programming, test-first development, continuous integration	code, graphics, etc.	S	r	r	r

Table 9.2. Sample partial UP Development Case						
Discipline	Techniques	Artifact	Incep.	Elab.	Const.	Trans.
		Iteration →	I1	E1..En	C1..Cn	T1..Tn
...	...					

[a] s = start. r = refine.

Lifecycle

1. **Inception** is ideally short, such as a few days. Iterations are possible, but rare. Activities could include a short requirements workshop, 10% of the requirements captured in detail (the most architecturally influential ones), a "top ten" high-level requirements list, and a first draft of the vision and business case for the project. If this phase is long, it is usually a sign of excessive up-front specifications or planning.
2. In **elaboration**, the core, architecturally significant elements are programmed and tested in a series of short timeboxed iterations, and by its end a semi-reliable plan and estimate is possible. This phase includes *programming* work, not only requirements or design modeling. In addition to development, with feedback from the growing system, there may be a series of short requirements workshops (one per iteration) to refine most of the requirements. This is a step of discovery and creativity; when complete, the core of the system and most requirements have stabilized through an iterative, evolutionary process.
3. In **construction**, the remainder of the system is built—in short iterations—on top of the solid foundation laid in elaboration. Requirements may still change, but ideally the big surprises were provoked and discovered earlier in elaboration. Other

activities include alpha testing, performance tuning, and document creation (for example, user aids).

4. In **transition**, the system is ultimately deployed. First, a release candidate is exposed for review and feedback. This may occur in several iterations. Finally, there is deployment, that may include distribution over various channels, education, parallel run with an older system, data conversion, and so on.

The UP identifies **milestone objectives** in a project that define the boundaries of these phases. These come from the work of the Barry Boehm, in [Boehm96].[2] Boehm called the end of inception the Life Cycle Objectives (LCO) milestone. The end of elaboration was called the Life Cycle Architecture (LCA) milestone.

[2] Boehm and the UP creators have a long history of collaboration.

Each phase may contain multiple iterations. The milestone goals of each phase are described in Table 9.3, and an example of phases and iterations is shown in Figure 9.5.

Figure 9.5. UP phases; the size of the relative effort in each discipline is suggestive, not literal

Table 9.3. UP phases

Phase	Milestone Goals	Comments
Inception	Agreement on scope, vision, and priorities. Some risks identified. A plan to start elaboration exists.	*Establish a common vision.* Typically a very short phase, such as a few days or weeks. A first requirements workshop might be held.
Elaboration	The vision, requirements, and architecture are stabilized. The core executable architecture is implemented; major risks are mitigated. The majority of requirements are defined. Estimates and coarse-grained plan are defined.	*Build and test the risky core.* This phase contains significant production programming and testing, combined with evolutionary requirements and design work. Usually composed of several iterations. In addition to programming, perhaps a series of requirements workshops; one per iteration. Semi-reliable plans and estimates at *end* of elaboration.

Table 9.3. UP phases		
Phase	**Milestone Goals**	**Comments**
Construction	System is believed ready to be deployed. Stakeholders are ready for deployment.	*Build and test the rest.* Typically the largest set of iterations. As in elaboration, major testing occurs in each iteration.
Transition	System is deployed. Users are satisfied.	*Deploy.* Beta testing, release candidate evaluation, training.

Figure 9.5 shows an *example* of iterations across the phases: three iterations in elaboration and 10 in construction. As usual, iterations are not necessarily the same length. Often, early elaboration iterations are longer (e.g., three weeks) due to the demands of creative and unpredictable discovery. Later construction iterations could be one or two weeks long. The resources or staffing also varies. Ideally, the elaboration phase is staffed by a small, cohesive, co-located team who shape the core. During construction, larger teams and more parallel development may occur. See Figure 9.6.

Figure 9.6. resources across phases; the size of the relative resource use is suggestive, not literal

Workproducts, Roles, and Practices

WORKPRODUCTS
(non-software)
Requirements
Vision
Summary of objectives, features, business case.
Supplementary Spec.
All detailed requirements not elsewhere. Features, rules, constraints, non-functional, etc.
Use-case Model
Stories or scenarios of using the system. Emphasizes functional requirements.
Design
Design Model
Object model (static and dynamic) of the packages and objects. May be reverse engineered from code.
Data Model
SW Arch. Document
Short learning aid to understand the system.
Implementation
Test & Verification
Test Plan
Summary of the goals and methods of testing.
Project Management
SW Dev. Plan
Overall and composite plan. Milestones, resources, etc.
Iteration Plan
The detailed plan for the next iteration, not a plan of all iterations.
Risk List
Risks with severity and mitigations
Configuration & Change Management Environment
Dev. Case
Customized process description. The workproducts and practices for the project.
Change Request
A uniform way to track all requests for work (enhancements, defects, etc.)

Roles

ROLES
Customer
Stakeholder
- customers, product manager, ...
Development
Implementer
- writes tests, designs, code
- refactors
Tester
- write and develop system tests
SW Architect
- establishes and maintains architectural vision
System Analyst
- requirements analysis
DB Designer
UI Designer
Management
Project Manager
Process Engineer
- leads the definition and refinement of the Development Case
Other
Graphic Artist
Technical Writer

Practices

Core Practices

To be called a UP project, it arguably should illustrate at least the following:

- Follow the UP guidelines and best practices.
- Create at least a couple of UP artifacts, such as the Vision and Risk List.

  - Conform to the UP workproduct names rather than use other or prior names, to establish a common vocabulary.

- Organize the iterations and milestone goals according the goals of inception, elaboration, construction, and transition.

UP Guidelines

Although the original UP creators had in mind a preference for a relatively light approach to process, this was not well communicated in their early learning aids. In their more recent material, this has improved. For example, [KK03] presents the "spirit of the RUP" guidelines for success—core attitudes to hold on a UP/RUP project:

- Attack risks early and continuously, or they will attack you.
- Deliver value to your customer—early and often.

- Stay focused on developing executable software in early iterations, not specifications or other documents.
- Accommodate change early in the project. Provoke and managing change via early development, multiple requirements workshops, change management tools, and so on.
- Baseline an executable architecture early.
- Prefer component-oriented architectures and the reuse of existing components.
- Work together as one team (e.g., cross-functional teams)
- Quality is a way of life, not an afterthought.

The Six Best Practices

The UP is not really limited to only six best practices, but these are an important minimal set to focus on. UP adopters should understand these practices, and most or all should be applied on UP projects:

1. The most important of the UP practices is to develop using ***timeboxed iterations***, recommended between two and six weeks. In other words, do not apply a waterfall lifecycle or attempt to do thorough requirements analysis first. Rather—as with all iterative lifecycle processes—start programming early, when only a portion of the most significant requirements are understood in detail. Refine the requirements and design based on feedback and adaptation from the programming effort instead of attempting large, up-front requirements analysis or speculative "Power Point architectures."
2. Emphasize programming the high-risk and high-value elements, and a ***cohesive architecture*** in early iterations, and strive to re-use ***existing components***—large and small—to reduce new code and defects. For large projects, ideally the requirements analysis and core architecture is developed by a co-located small team; later, the early team members divide into subproject leaders coaching subteams doing parallel development.
3. ***Continuously verify quality***. Test early, often, and realistically by integrating and testing all the software each iteration—unit, system, and load testing. This best practice includes techniques such as test-driven development and continuous integration, both promoted in XP. Plus, it extends beyond code to include early verification of usability, the quality of non-code artifacts (such as requirements), and of the process itself via regular team meetings to reflect on the value or lack thereof in different activities.
4. Before starting programming in an iteration, do at least a little ***visual modeling***, such as sketching at the whiteboard for an hour, to explore and communicate creative design ideas while ignoring low-level code details. Oftentimes, sketches will be loosely based on the industry-standard Unified Modeling Language (UML). Practices promoted by Agile Modeling, such as creating multiple visual models in parallel, using digital cameras to capture sketches, and so on, are applicable to this best practice. Sometimes, CASE tools for visual modeling can add value, to regularly reverse engineer the growing code base into UML diagrams that offer a big-picture view of the software.
5. ***Manage requirements*** through skillful means to *find, organize*, and *track* them. *Find* and refine requirements iteratively and incrementally rather than via major up-front analysis—for example, through a series (once per iteration) of short timeboxed one-day requirements workshops in the early phases of the project. Where suitable, *find* functional requirements by writing use cases to record functional requirements, rather than older-style function lists. Using a tool, *organize* requirements with attributes, such as risk, priority, and status (e.g., new, assigned), and with traceability to other dependent requirements, so that they can be analyzed; using a tool for this task is especially valuable on large projects where it is difficult for any one person to envision and reason about all the requirements. And using a tool, *track* the status of requirements so that we know what's finished, underway, and so forth.

6. ***Manage change*** through disciplined configuration management and version control, a change request protocol, and baselined releases at the end of each iteration.

Workproducts

The workproduct diagram (p. 184) shows a subset of common UP workproducts (artifacts). There are approximately 50 in total. The details are outside the scope of this introduction; see [Larman01] for many examples, or the RUP product.

Important overarching points regarding the UP workproducts:

- **Unfair criticism?**— Some have criticized the UP as having too many workproducts in comparison with some other IID methods. Yet, when you examine the full UP list, it contains workproducts such as *Release Notes, Bill of Materials, Training Materials*, and so forth. These are workproducts that a team makes for many products, regardless of method. It is just that the UP has identified them, whereas XP and Scrum, for example, have not. And, all are optional.
- **Common vocabulary**— For larger organizations, especially those with hundreds of developers or multiple offices, it is useful to have a common vocabulary of workproducts (*Vision, Software Architecture Document*, etc.). It helps communication between people and offices. It helps during scavenging for reusable artifacts from prior projects.
- **Information abstractions**— The UP artifacts are information abstractions rather than concrete computer documents, although the RUP product provides sample document templates. The *Risk List* can be a poster on the wall; the *Software Architecture Document* can be a video; the *Design Model* can be UML-ish whiteboard sketches captured on a digital camera.

Other Practices and Values

Absorbing practices— The UP is a broad and general IID process definition. It defines or endorses many detailed practices, both "local" to the UP or adopted from other methods, such as XP testing or Scrum meetings. Note that many practices, such as XP's *test-driven development*, are specializations or variations of the UP practices, such as *continuously verify quality*. If the RUP product is used, the "RUP Builder" can configure the process definition to include documentation on absorbed practices.

Use-case driven— The UP does not require the application of use cases to capture functional requirements; feature lists and feature-driven development are possible. However, when use cases are suitable for the problem domain, their use is encouraged in the UP. In that case, the UP recommends organizing iterations by scenarios of use cases, tackling the most architecturally significant, risky, and highest business value scenarios early. Then, one designs and implements to fulfill use case scenarios. Further, the UP recommends basing system-level acceptance tests on these scenarios. Thus, use cases drive the iterative development and acceptance testing.

iteration planning with use cases and scenarios

Values

In contrast to XP and Scrum, the UP does not make explicit a set of underlying values, although some can be inferred. And whereas the XP values emphasize human and communication qualities, the UP values emphasize a project-oriented rather than people-oriented focus. These include:

- It is important to apply the UP guidelines and best practices. Some UP values logically follow from this, such as iterative is better than waterfall, having a cohesive architecture is good, change control must be formalized, and so forth.
- Be risk- and value-driven. That is, in early iterations, identify and drive down the high risks and build elements deemed of highest value to the stakeholders.
- It is important to define a clear vision (in the UP Vision workproduct) for the project that summarizes the stakeholders' real needs.
- It is critical that the UP process be tailored to the unique needs of each project while still observing the best practices. Plus, be tailored to the minimal set of workproducts and activities that add value.
- It is useful to have a well-defined process that provides guidance on activities, on what artifacts to create, and on the tasks of individuals and the team.

UP as a Heavy, Defined Process versus an Agile UP Approach

It is this last value that is a point of contention in the UP compared to the classic agile methods. Some in the agile process community have dismissed off-handedly the idea that a defined process such as the UP is useful, and that it is too "heavyweight," but the issue is more nuanced.

for a discussion of defined vs. empirical methods

First, the UP creators did not intend the process to be applied or adopted in a rigid or heavyweight manner; they are experienced and practical software developers who also appreciate simplicity and agility. They view the UP as a suite of options to pick from, within the constraints of adopting its spirit and best practices.

Another perspective on this issue is offered by Cockburn, who describes three levels of behavior and listening as people mature in learning a subject [Cockburn02]: 1) following, 2) detaching, and 3) fluent. He relates this to literature and guidance for software methods by noting that Level-3 advice such as terse "work in a common project room and deliver usable software every four weeks" may be suitable for the *fluent* master, but not for the novice *following* developer. The detailed defined process aspects of the UP are aimed especially at a Level-1 audience, and in that context are of some value for learning, and as quality assurance checklists.

Level-2 and Level-3 developers may ignore the prescriptive and defined aspects of the UP, such as what tasks to do in what order, and instead focus more simply on choosing a set of UP workproducts to create, applying the best practices, and realizing these according to the creative judgment of the team in a more empirical process spirit.

A second issue in the value of defined processes is repeatability and avoiding re-invention. There *are* at least a few predictable, repeatable, useful steps in deploying a system to production, such as writing release notes. The UP makes explicit this advice (which is especially useful for the Level-1 audience), whereas the agile methods do not.

A third point in the value of defined processes is having a common vocabulary for artifacts, such as the UP Vision and Design Model. The UP provides such a vocabulary, which removes the need for each project to recreate one, and promotes communication across projects and organizations.

The issue of defined versus empirical processes for software projects boils down to a matter of balance and moderation. Certainly, rigid command-control sequencing of fine-grained, predefined activities is not skillful on most software projects, but on the other hand, some aspects of defined processes add value and can be applied in an agile and empirical spirit, as some activities are predictable—for example, many related to deployment.

Given a set of iteration goals, a self-directed team doing daily Scrum meetings can be characterized as applying the UP if it creates a few UP workproducts and follows its best practices, but is driven by the dynamic and creative judgment of the team, rather than by following a UP recipe of ordered activities.

In summary, the UP—although a semi-defined process—can be and is applied by some in an agile style.

Common Mistakes and Misunderstandings

or, How to Fail with the Unified Process

There are internal IT organizations, books, Web pages, articles, consulting organizations, and speakers that demonstrate these misunderstandings. Be cautious in receiving UP advice or hiring consultants, and apply the tests in the "Signs the UP "Expert" Is Not an Iterative Expert" section on page 198.

Superimposing Waterfall Ideas

The most common of the significant UP misunderstandings, and a quick sign that the UP "expert" actually is not, is to describe the four phases akin to the waterfall phases (requirements, design, implementation, test):

1. **Inception**— requirements analysis, detailed specifications.

   > *THIS IS INCORRECT; it is a sign of misinformation by those not truly understanding the UP*

2.
3. **Elaboration**— more detailed requirements analysis, modeling, and design work.
4. **Construction**— programming the design.
5. **Transition**— testing.

This *incorrect* description of the UP phases illustrates the most common misunderstanding: superimposition of waterfall phases onto the UP. A corollary of misapplying the waterfall phases includes other common waterfall mistakes in UP adoption:

- Attempting to do most requirements analysis or design before programming.
- Deferring major testing or QA till near the end of the project.

Other Common Misunderstandings

Error: Iterations too long— It is usually a misunderstanding to define iterations of several months long. The UP recommends an iteration length between two and six weeks, excluding massive projects involving hundreds of people and many subteams. Skilled iterative project leaders strive towards shorter timeboxed iterations in the two to four-week range, all other things being equal. One project may be composed of dozens of short iterations.

Error: Iterations aren't timeboxed— It is a misunderstanding to let the iteration length expand when it appears the goals can't be met within the original timeframe. Rather, the usual expert strategy is to remove or simplify goals for the iteration.

Error: Iteration doesn't end in an integrated and tested baseline— An iteration is not properly complete unless all the software, across all or most subteams, has been integrated, tested, and baselined. It is a misunderstanding to think an iteration simply ends arbitrarily on the end date; the goal is to pull everything together.

Error: Each iteration ends in a production release— It is a misunderstanding to think each iteration must end in a production release. Although this is possible (especially during maintenance) it is less common than requiring many iterations before a production release.

Error: Elaboration phase goal is to create a throwaway prototype— Prototypes are perfectly acceptable in the UP (usually during inception), but the goal of elaboration is not a throwaway prototype but rather a production subset of the final system. Some have misunderstood this (and some UP books have misadvised on this point) because the original UP and RUP literature itself used the unfortunate choice of phrase "architectural prototype" in a few places to describe the output of elaboration. This confusing term was meant to imply "architectural subset of the final production system" but many interpreted "prototype" to invariably mean throwaway code.

Error: Development Case too complex; too many workproducts— It is a misunderstanding to define a Development Case with dozens of UP workproducts, when fewer will suffice. The guideline is "less is better." This is not advice to avoid demonstrably useful documentation, or the forethought that accompanies its creation; the UP recommends creating workproducts that really add value, and abandon make-work or low-value document or model creation.

Error: Predictive planning— It is a misunderstanding to create, *near the start of the project*, a believable plan laying out exactly how many iterations there will be for a long project, their lengths, and what will occur in each.

Error: The team should do lots of modeling and UML diagrams, and use a CASE tool— The UP contains several optional models, with many opportunities to apply the UML for diagramming speculative designs before programming. However, these are optional, and if a team can successfully and easily develop software with little or no prior diagramming, they may. Modeling and UML diagramming in the UP are aids to help with complexity and creativity, no more. And it is certainly not necessary to use a CASE or UML drawing tool while modeling on a UP project. The Agile Modeling approach that recommends the simplest possible tool—perhaps whiteboard hand sketches and digital cameras—is perfectly suitable.

Error: Need many tools— It is a misunderstanding to think many software tools need to be applied on an UP project. Rather, it can be run as low-tech, high-touch as paper cards, wall posters, and whiteboards, combined with a sprinkling of CVS, Anthill (for continuous integration), and Bugzilla (for issue and defect tracking).

Error: Software Architecture Document (SAD) "finished" before end of elaboration— The UP SAD is a learning aid that summarizes the big ideas and motivation in the architecture. It is a misunderstanding to create the final SAD before the end of elaboration, as that would imply major up-front design, and speculative definition of the architecture without programming. In the UP, the architecture evolves iteration by iteration through an interplay of some educated guesses combined with programming and testing. The architecture is not stabilized until the end of elaboration, after significant programming to build and prove it. Thus, the SAD, which summarizes the architecture, cannot be finished until elaboration is over.

Error: Not conforming to the official UP workproduct names or phase names— One purpose of the UP is to establish a common vocabulary, both within an organization and globally across UP-conforming teams, for workproducts and major lifecycle phases. For an organization that is adopting the UP to replace a prior process, it is a misunderstanding to rename the UP workproduct to the older familiar names, rather than surrender to the new terms.

You Know You Didn't Understand the UP When...

Some of the key misunderstandings expressed as a checklist:

- You think a sequence of requirements design implementation express the way to run a UP project.
- You think inception is like requirements analysis, elaboration is like design and detailed requirements analysis, construction is like programming after the design, and transition is a testing phase.
- You want to do most requirements analysis or design before programming.
- You defer the major testing until near the end of the project.
- You define iterations months long rather than weeks long.
- You create the SAD before the end of elaboration.

Signs the UP "Expert" Is Not an Iterative Expert

UP consultants, authors, and speakers from both major well-known consulting organizations and small shops may exhibit a misunderstanding of the UP and iterative development. Here are some of the key ones illustrated in the behavior of a UP "expert":

- Describes the UP phases similar to waterfall phases; recommends doing most requirements or design before programming; encourages creation of more and more detailed specifications and plans, before starting development work.
- Suggests iterations more than six weeks long. Does not strive towards short iterations.
- Recommends an inception phase that is weeks (or worse, months) long, involving major planning and analysis; is not encouraging a rapid transition to early programming in an elaboration phase.
- Is not stressing the importance of early programming starting in the first elaboration iteration, well before all requirements, plans, and design are known.
- Recommends creating many UP artifacts, rather than seeking ways to reduce the number. Some UP consultants do this out of ignorance; others out of a justification for their presence and guidance in the creation of these and handling a more complex process.
- Builds only a throwaway prototype in the elaboration phase.
- Defers unit, load, usability, or acceptance testing until near the end of the project.
- Insists that the team follow the UP activities "by the book."
- Insists on the team using UML CASE tools to draw many diagrams, whereas hand sketches on whiteboards, or simply bypassing drawing and just programming, would suffice.
- Creates the "final" SAD before the end of elaboration and before significant programming has occurred.
- Near the start, defines a "believable" plan laying out how many iterations there will be for the project, their lengths, and what will occur in each (i.e., predictive planning).
- If hired as an UP auditor to verify the team is skillfully applying the UP, encourages more or more thorough workproduct creation or more detailed planning, rather than early programming, evolutionary requirements, and adaptive planning.

Sample Projects

The following projects applied many of the values and practices that characterize a UP project:

Large— Canadian Automated Air Traffic Control System

- Ten years, 400 people, Ada and C++, an L400 life-critical project, [PKT93].

- This was a large test bed for the practices that were later refined as the RUP. The chief architect was Dr. Kruchten (who drove the adoption of these practices), also the lead architect of the RUP. It was first attempted as a waterfall project, failed, and then was restarted under the new process direction of Kruchten as an iterative

project. As a successful iterative ATC project, it is an instructional contrast to the failed attempt (after 11 years and $2.6 billion USD) to replace the USA ATC, which was unsuccessfully run as a waterfall project [GAO98].

- Prime developer: Raytheon Systems Canada. Originally, Hughes Canada Systems Division.

Medium— Ogre Nextgen Economic Modeling System

- Two years, 15 people, Java technologies, an E20 project.

- A Java technology decision support system used by oil/gas asset holders (e.g., potential oil fields). Development involved 19 iterations that on average were four weeks. Included the Scrum practice of a demonstration to external stakeholders at the end of each iteration.

- Prime developer: Valtech USA.

Small— QUICKcheck point-of-sale

- One year, six people, Java technologies, [Evans01].

- A self-checkout POS system for grocery stores.

- Prime developer: Kyrus.

Process Mixtures

UP + Evo

The UP is especially for software development, and usually for projects involving multiple iterations before production delivery. Consequently, the UP could be applied to Evo backroom development work. However, Evo's very frequent evolutionary delivery and project management style is not exactly in the same spirit as the UP, although both share an interest in early identification and mitigation of risks. The UP has its own set of workproducts and approach to requirements capture: the Use-Case Model (and thus, use cases), and Supplementary Specification for description of functions, features, and nonfunctional requirements. Evo Planguage elements, such as the Performance Requirement Specification, may be used within the UP Supplementary Specification. Evo's measurement emphasis is compatible or acceptable with the UP. The upper bound of UP's 2–6 week iteration length is not consistent with Evo—too long.

Evo

backroom

Planguage

UP + Scrum

The Scrum practices are consistent with UP practices. The Scrum Product Backlog is an acceptable portion of the UP Project Plan, and the Sprint Backlog is an acceptable version of the UP Iteration Plan. One area of potential conflict is the presence in the UP of optional but predefined activities; the UP indicates some dependent ordering of these optional

activities—for example, that a project vision is created before a detailed requirement is described. Scrum's rejection of defined process and predictable steps is inconsistent with this structure if the UP activities are viewed as a required formula. But, if the activities are treated as optional advice, performable in any order, and without attempt to schedule their order and duration on a project, and without assignment of tasks to individuals by a manager, it is not in conflict with Scrum.

Scrum

Product and Sprint Backlog

See "UP as a Heavy, Defined Process versus an Agile UP Approach"

UP + XP

Most XP practices are consistent with UP practices, and many XP practices can be applied in the context of an overarching UP project. For example, test-driven development is a specialization of the UP *continuously verify quality* best practice. Since all UP workproducts are optional it is a misunderstanding to assume the methods are fundamentally incompatible, given XP's promotion of minimal modeling and documentation.

XP

test-driven development

However, although speaking of some XP practices within a UP project can have conceptual integrity, the opposite is not true, as there are some differences in style and emphasis. One area of difference is in the accepted degree of up-front modeling (diagramming, etc.). For example, within a UP project and a two-week iteration, it is considered acceptable to spend a halfday near the start to sketch and explore design ideas "at the whiteboard" (visual modeling) before programming. In XP, no more than 10 or 20 minutes before programming is considered suitable.

Another difference is in the goal of the early iterations. In the UP the goal is to identify and drive down the high risks: technical, political, satisfying the customer, and so forth. Although this may happen in the XP, it is not an explicit guiding principle.

A third difference is in requirements specification. The UP allows and supports the creation of relatively detailed specifications (incrementally, over a series of iterations), assuming that an onsite customer is not going to be present. These will usually take the form of the Use-Case Model and Supplementary Specification.

Adoption Strategies

As always, coaching by an experienced method expert on the first project is recommended. UP adoption is meant to be iterative, incremental, and adaptive. "Big bang" process adoption, where many people are trained near the same time, and/or many UP projects start at once, should be avoided.

First, executive management will benefit from education in some UP concepts (for example, through a short seminar). An executive sponsor should be identified, and discussion of a suitable pilot project initiated.

Then, a pilot project starts. The project should be large enough to be meaningful and interesting, yet not bet-the-bank risky. For example, a 10-person, six-month project.

A UP coach should be responsible for leading the definition of the Development Case, and its refinement during the elaboration phase. Ideally, the coach should be a hands-on developer during the pilot, so that *eating their own dogfood* is strongly experienced, and the process is realistically refined.

See "Signs the UP "Expert" Is Not an Iterative Expert"

To contrast that, a *worst practice* is if a "Methods and Practices" group within the organization defines how the organization or a project should adopt the UP. "Armchair process engineers" are not helpful, should be avoided, and often superimpose waterfall values on to the UP, or promote excessive workproduct creation. Take advice on what to do or adopt in the UP from hands-on developers and managers doing early UP projects, in collaboration with a UP coach participating and developing as well. If corporate process engineers are involved in UP adoption recommendations, they should discover and refine recommendations by serving as hands-on developers during pilot projects, rather than by speculation.

A related worst practice is to attempt to fit the UP into the organization's existing concepts and workproduct vocabulary. For example, taking the organization's current lifecycle phases, and believing the UP phases fit within them, or are equal to them. They won't be, and it usually leads to waterfall superimposition. It is a similar mistake to rename the UP artifacts (Vision, etc.) to the old names used by the organization. In short, UP adoption is best done by incremental surrender to a new set of ideas and terms, not by dressing up an old horse in new clothes.

There are many possible practices in the UP. Thus, it is useful in the early iterations—assuming a prior culture of low software engineering maturity—to start simple and add some practices as the iterations proceed.

After project completion, assuming it was successful and positive, "in-selling" of the UP practices by the project members or customers themselves to others in the organization is better than promotion by outside consultants or management. For example, a lunch-time session where team members share their experiences, pros and cons, with a wider audience.

Assuming there will be a second generation of UP projects, some of the members from the first generation project will ideally move into process engineering roles, and help coach these new projects. The Development Case from the prior projects should be carried forward as a starting point. Therefore, the adoption spreads incrementally.

Another useful activity is a project retrospective that looks at the refined Development Case and ensures it accurately reflects what worked well, and what didn't.

Fact versus Fantasy

As implied earlier, perhaps the most common fantasy regarding UP adoption is that it is widely being applied in the iterative and adaptive manner its creators intended. Rather, a non-trivial percentage of organizations and consulting companies are incorrectly applying it in a predictive and waterfall spirit, with up-front specifications and planning.

Another fantasy is when the organization purchases many copies of the RUP product for their development staff, and continue with the mistaken belief that the presence of the installed product implies true iterative and evolutionary development is underway.

Another common and related fantasy is when the organization decides to adopt the RUP, thinking it is simply a more detailed, well-defined, and well-documented version of what they already expect in development (e.g., the waterfall), and they do not themselves have to make deep changes in their expectations and behavior. For example, in the RUP, reliable estimates for the entire project are not to be expected until the *end* of elaboration.

Instead, this happens:

> *Memo: Congratulations! We've adopted the UP. Please ensure the requirements are finished for our upcoming UP projects so we can estimate them before moving on to the programming phase.*

Strengths versus "Other"

Strengths

- Focus on risk- and value-driven priorities.
- Emphasizes building a cohesive architecture early, preferring existing components.
- Evolutionary and incremental requirements and development, and adaptive behavior.
- Well-defined workproducts, offering a common vocabulary.
- The guidelines and best practices are a good foundation.
- Offers guidance on many disciplines, from requirements to configuration management.
- Easily combined with techniques from other methods.
- Easily customized; encourages minimal "light" versions.
- RUP product has easily accessible useful advice, and standard project templates.
- Proven to scale to large or small projects.
- Encourages use cases where appropriate.
- Widely adopted; hence, learning and consulting resources.
- Created with many customers, refined on many projects; not a speculative process.

Other[3]

[3] Could be viewed as a weakness, strength, or deliberate desirable exclusion depending on point of view.

- Many details.
- UP phases are often incorrectly applied in a waterfall style.
- Minimal attention to the social dynamics and communication aspects of successful and sustainable development practices. Contrast to Scrum and XP.
- The RUP product can give the (unintended) impression of promoting a very defined and predictable development process, as though software was mass manufacturing rather than new product development.

History

Some of the roots of the UP/RUP exist in the work of Barry Boehm and his spiral model. Its risk-driven and iterative approach influenced key UP/RUP contributors at Rational Corporation, including Philippe Kruchten, Grady Booch, Mike Devlin, Rich Reitman, and Walker Royce. Indeed, Royce collaborated with Boehm on research, writing, and projects. Boehm's well-known "anchor point" milestones [Boehm96] became the milestones that defined the boundaries between inception, elaboration, and construction. The Rational team created the UP/RUP in collaboration with customers, taking input from the "Rational Approach" they had developed through the 1980s and 1990s, and from the Objectory process they acquired from Ivar Jacobson. The core initial development was around 1995–98. The chief architect of the UP/RUP was Philippe Kruchten, an experienced architect or process leader on major applications such as the new Canadian air traffic control system,

and a well-known architecture thought leader, e.g., [Kruchten95]. Although Rational had *Rational* Unified Process and a commercial product in mind from the start, they also wanted to communicate and promote the idea of a process more public domain and open—a generalized *Unified Process*. This was consistent with their open Unified Modeling Language initiative. Hence, Ivar Jacobson wrote the first book to present this view, *The Unified Software Development Process* (1999) working from a draft of the RUP specification and product being developed by Kruchten's team. Since then, many books have been written under the appellation of simply "Unified Process" to signify similarity to the RUP and adoption of its major ideas (the best practices, the phases, the disciplines, and so forth), while not necessarily being strictly the Rational process.

What's Next?

The next chapter presents Evo, a very early iterative and evolutionary method. After that are two final chapters on practice tips and a FAQ.

Recommended Readings

- An excellent introduction is *The Rational Unified Process Made Easy* by Per Kroll and Philippe Kruchten.
- Another very readable introduction to the RUP is *The Rational Unified Process—An Introduction* by Philippe Kruchten, its lead architect.
- *Applying UML and Patterns*: *An Introduction to OOA/D and the Unified Process* [Larman01] takes a case-study approach to introducing the UP and OOA/D. The concepts and workproducts are introduced incrementally over a series of case-study iterations.
- As a caution, there are a number of so-called UP and RUP texts that give incorrect descriptions.

Worthwhile readings that influenced the UP creators:

- "A Spiral Model of Software Development and Enhancement" by Barry Boehm, *ACM SIGSOFT Software Engineering Notes*, August 1986 (the reprint in *IEEE Computer*, May 1988 is often cited). A widely cited paper that describes the spiral model of iterative and risk-driven development.
- Tom Gilb's iterative Evo method, described in *Principles of Software Engineering Management*, 1988.
- Grady Booch's *Object Solutions—Managing the OO Project*, 1995.
- "Anchoring the Software Process" by Barry Boehm, *IEEE Software*, July 1996. Describes the milestones from which the inception, elaboration, and construction phases were inspired.
- The PMBOK (www.pmi.org). Some of the Project Management discipline was influenced by the PMBOK.

Chapter 10. Evo

There is only one move that really counts: the next one.

—Chess master Jose Capablanca

OVERVIEW

- Classification of UP.
- Workproducts, roles, and practices.
- Demonstrate Planguage for Evo specifications.
- Common mistakes, adoption and process mixtures, strengths and weaknesses.

Evo (short for **Evolutionary Project Management**) is perhaps the oldest IID method with a significant agile and adaptive quality, first taking shape in the 1960s and then published in 1976. Evo emphasizes:

- short iterations, with evolutionary delivery each iteration
- evolutionary requirements and design
- adaptive client-driven or value-driven planning
- quantifiable measurements of value and progress
- defining all quality requirements with numeric measures
- optional use of a language, Planguage, for specifications

Method Overview

Classification

In terms of cycle and ceremony, Evo classification is illustrated in Figure 10.1. For average projects, a common length of a timeboxed iteration is one or two weeks.

Figure 10.1. Evo on the cycles and ceremony scale.

Evo recommends some initial work to define a "critical top ten" list of measurable project objectives, and when specifications are written, Evo encourages unambiguous precision. It also encourages brevity, promoting one page summaries. Evo avoids big up-front

specifications, although evolving specs—that could be part of a small or large set—are acceptable if shown to be valuable.

When describing high-level requirements, a structured language call **Planguage**[1] is possible; it encourages clarity, precision, and measurement. If used, it raises Evo on the ceremony scale.

[1] Rhymes with "language."

Similar to Scrum, it has only a small set of predefined workproducts, such as an impact estimation table. Others may be adopted from different methods as needed.

impact estimation table

In terms of the Cockburn scale, Evo covers the cells shown in Figure 10.2. Since the 1970s, it has been applied on a wide range of projects of many sizes.

Figure 10.2. Evo on the Cockburn scale

Introduction

Evo [Gilb76, Gilb88] was created by Tom Gilb, a pioneer of iterative and evolutionary development.

I'm including this chapter on Evo—less well known than Scrum, XP, and UP—not only because of its inherent interest, but to balance the historical oversight of this pioneering iterative method and to show that some agile method principles have long been part of Evo, such as a adaptive, client-driven planning of iterations.

Agile Principles

Gilb has been an advocate for an iterative, light, and adaptive approach to systems development since the 1960s; he first wrote about this in 1976, and his 1988 *Principles of Software Engineering Management* is a milestone early book presenting an evolutionary and iterative process.[2]

[2] Gilb also wrote the first book on software metrics, coining the term in [Gilb76], and continues to refine Evo, e.g., [Gilb03].

Evo's evolutionary emphasis is consistent with the Shewhart/Deming cycle of Plan-Do-Study-Act (PDSA), and makes reference to PDSA as an underlying conceptual model.

Evo is not just for software. It is applicable in a larger systems engineering context—new software is just one solution to fulfill project objectives. For example, if more education (on the existing software) or operational change has a better value-to-cost ratio than new software, the former approaches are preferred.

It emphasizes—short iteration by iteration—making maximum progress towards the client's current highest-priority requirements, for the lowest cost. And each iteration, delivering into the hands of some stakeholders some useful results, so that early benefit and feedback is achieved. This is the practice of client-driven adaptive planning and evolutionary delivery.

Evo is pragmatic, has some qualities similar to newer agile methods, is customer focused and results oriented—in the spirit of the Agile Manifesto and Principles. Anything necessary can change (based on the PDSA model) to reach the requirements (function or performance) within the project constraints.

Agile Manifesto

One of Evo's distinguishing ideas is its emphasis on clearly defining, quantifying, estimating, and measuring the **performance requirements** that need improvement over time.

these bold terms are official Evo terms

Performance includes **quality requirements** such as reliability, **workload capacity requirements** such as throughput, and **resource savings requirements** such as money. The impact of Evo steps on budgeted resource consumption is monitored both in design activity and iteration project management activity.

example requirements

Evo requires evaluating proposed solutions for their impact on the state of these requirements, and then actually measuring the impact of those introduced.

This structured approach and emphasis on improving the *performance* characteristics, rather than just on delivering functionality, is a key part of Evo's unique flavor.

Thus, note that in Evo there is explicit recognition that the requirements delivered may be either functions *or* performance objectives (quality, workload capacity, or resource saving).

Evo expects that each iteration there is a re-evaluation of solutions which yield the highest value to cost ratio, guided by feedback and estimates. As such, Evo requires active stakeholder participation to steer the project each iteration—client-driven adaptive planning. These practices are part of **evolutionary project management**.

adaptive planning

Measurable progress is a key principle of Evo, which takes seriously Drucker's maxim: *If you can't measure it, you can't manage it*. Quantifiable measures for performance requirements, and their regular measurement, is required. Unproven improvements, and vague quality goals such as "usable" are discouraged.

In Evo, the value system is that management doesn't schedule the details of the entire project, but they must be able to measure, control, and steer a dynamically evolving project. In other words, adaptive planning.

Evo encourages *precision* and (where relevant) *quantification* in specifications. It does so by encouraging (but not requiring) the use of a compact, structured specification language called Planguage to record requirements—iteratively and incrementally.

Planguage

It is a misunderstanding to interpret Evo's promotion of high-quality, low-volume critical specifications as an attempt at large up-front analysis. Evo promotes avoiding unnecessary analysis and detail—until it is needed.

Inspection— especially of these specifications—is encouraged in Evo as an economical method to improve quality. Indeed, research verifies this [Russell91], and Gilb has been an active promoter of inspections for decades, including co-authoring the text *Software Inspections*.

inspection

Evo also encourages a risk-driven approach, as does the Unified Process. As Gilb has aptly said,

If you do not actively attack the risks in your project, they will actively attack you.

Lifecycle

1. In the **Strategic Management** cycle, stakeholders decide which solutions ready for delivery (perhaps from the backroom activities) will actually be delivered, usually based on highest value-to-cost and risk. This activity also includes approving changes to objectives and solutions, analyzing feedback measurements, and obtaining resources.
2. These cycles may be concurrent. Ideally, each week something is delivered to stakeholders for use and feedback. In parallel, timeboxed development and production cycles work on incrementally building solutions ready for delivery, although it may be weeks (or longer) before they are eligible for delivery. The analogy Evo offers is a business with the following organization:

   - **Backroom**— products are prepared, and when ready, are "placed on a delivery shelf" available for delivery.

   - **Frontroom**— some eligible products are taken off the shelf and delivered to stakeholders (see Figure 10.3).

Figure 10.3. backroom and frontroom delivery

Projects carry on, driven by the goal of maximizing stakeholder value at lowest cost, until there are no more profitable requirements to fulfill.

Niels Malotaux, another Evo consultant, describes the lifecycle of Evo projects from his experience working with clients [Malotaux03]:

1. A project kick-off "Evo Day" that includes the project manager, architect, and all other development team members. Activities include presenting an overview of Evo ideas and practices, explaining the product vision and architectural ideas, identifying and estimating tasks for the first two-week iteration, and prioritization. Finally, people choose and commit to a set of individual tasks for the next week.
2. Execution of the two-week iteration.[3]

   [3] Malotaux has found that two-week delivery iterations are more sustainable than one-week delivery iterations.

3. On the last day of the iteration:

   - First, the project manager visits each developer and discusses the task results and completion. If things were not completed, there is reflection on the causes.

   - Second, the project manager discusses the project status with stakeholders (e.g., the product manager). Requirements are revisited and re-prioritized. Those chosen for the next iteration are analyzed and specified in greater detail, with measurements and so forth.

   - Third, the project manager and development team generate a new set of tasks. Again, developers choose and commit to the highest-priority tasks for the next week. In a team meeting, experiences of the last iteration may be discussed for process improvement ideas, and the product vision and evolving architecture may again be summarized or refined, to promote a common team goal.

Workproducts, Roles, and Practices

WORKPRODUCTS
(non-software)
Requirements
Requirement Specification
Function Requirement Spec
What the system does
Performance Requirement Spec
How well it does it:
- qualities
- workload
- savings, ...
Design
Design Spec
Solutions to the requirements
Each idea explains how it contributes to the performance requirements
Implementation
Test & Verification
Performance Requirement Spec
Each spec includes "meter" or test methods
Project Management
Evo Plan
A tentative description of the frontroom and backroom iterations
An adaptive plan--the next iteration is definite, but later ones may be unspecified or change
Impact Estimation Table
Shows analysis of impact of design ideas on the performance and cost requirements
Configuration & Change Management Environment

Roles

ROLES
We see only relatively generic roles in Evo, as it is a general systems engineering method.
Customer
Owner
- responsible for the specifications for the next iteration
Stakeholder
- those who receive results of next iteration
- any other party with an interest in the system
Development
Implementer
- responsible for implementing the iteration
Systems Architect
- responsible for identifying and advising on architectural choices
Management
Project Manager
- responsible for measuring the results of solutions, coordination, etc.
Other

Practices

Core Practices

Evo applies to systems engineering in general—not only software development—although software projects are a common domain of application.

As with the other IID methods covered, Evo promotes evolutionary requirements analysis. Yet, when requirements and design ideas are written, Evo requires analysis with respect to a measurable evaluation of the value and impact of requirements and designs. Evo is infused with the practice and value of *measurable, measuring*, and *adaptive response* to the results.

Requirements Practices

Practice	Description
Find stakeholders	Both internal and external, friendly and foe, and across the lifecycle of the system.
Define "top 10" key reqs	Evo, as with other IID methods, encourages an early definition (in Planguage) of "critical top ten" *high-level* requirements. They need not all be decomposed into fine details, although those facing early

Practice	Description
	implementation may be. Each iteration, they are reviewed and refined.
Define function specs	Evo functions describe what the system does. Evo does not promote major up-front detailed functional requirements analysis, but it does require at least clear definitions for the next iteration, *optionally* described in the **Function Requirement Specification**, using Planguage. *example p. 232*
Define performance specs	Evo promotes describing system performance—how well the system works, its benefits, and how it affects the environment. Written and refined incrementally. Performance attributes are attached to functions. Specifically, Evo performance attributes fall into three categories: 1) quality—how well it performs (usability, reliability, …), 2) workload capacity, and 3) resource savings. The Performance Requirement Specification captures this information using Planguage. *example p. 233*
Define clear, and (where possible) measurable, specs	When specifications are written, do so in a manner and language which exposes and minimizes misunderstanding or ambiguity. The Evo Requirement Specification examples illustrate this. Evo promotes a balance between too little and too much detail in requirements. It wants clarity and detail for the key specifications you have chosen to implement in the next short iteration. Other more speculative or unassigned requirements can wait. Evo's performance specifications should have measurable impact, which should be identified. *examples p. 231*
Use Planguage for specs	Planguage is Evo's structured language for specifications in both requirements and design. It is optional, but encouraged. Evo includes **Planguage templates** for its requirements and design specifications. *notation p. 231, examples p. 231*

Project Management Practices

Practice	Description
Evolutionary project management	Key ideas include: - evolutionary delivery to stakeholders for real use and feedback - small steps (ideally bi-weekly, or between 2–5% of total project financial cost and time) - steps with highest quality-to-cost ratios given highest priority for delivery - the existing system is preferred as the initial system base - feedback modifies future plans and requirements; adaptive

Practice	Description
	planning and evolving specifications - total systems approach; do anything that helps - early results-orientation *discussion p. 227*
Evolutionary delivery	Evolutionary delivery emphasizes delivering a partial solution into production early, in order to obtain early business value, and feedback to guide and evolve future deliverables. A common delivery frequency in Evo is weekly, or more specifically, every 2–5% of duration and budget. In Evo, the solution chosen for delivery in the next iteration is based on highest value-to-cost ratio and early risk reduction. Each iteration's solution can be of a different type. For example, within a project to replace an older mainframe payroll application, early deliverables could be quick-fix operational changes in the existing system, or adding a Web-based front end to the old system, while work on the new system is underway in the backroom. *discussion p. 227*
Measure impact of delivered solutions	Evo embraces Shewhart and Deming's core principle of improvement: PDSA. Plus, Drucker's maxim that you can't manage what you can't measure. The *study* step requires measuring, each iteration, the effect of the solution on the objectives. This data is used to help drive evolutionary project management (*act* in response to *study*), iteration by iteration. *Evo plan table p. 229*

Design Practices

Practice	Description
Define design specs	Design ideas are also recorded in Planguage, in the **Design Specification**, and are incrementally evolved, as with requirements specs. *example p. 234*
Impact estimation	A method to numerically analyze and compare the effectiveness of design ideas to meet cost and performance requirements—the qualities, workload capacity, and resource savings. The results are expressed in an ch10lev3sec7. *example p. 235*
Describe how design ideas meet reqs	The design specifications in Evo should explain why and to what degree they fulfill the requirements. This information is used in impact estimation, and discourages "resume-driven design" in which over-engineered or unfocused designs arise that are not really pertinent to business goals and value. *example p. 234*

Test and Verification Practices

Practice	Description
Specify tests and measures in the reqs	The *study* step in Plan-Do-Study-Act step requires measurements or **meters**, in Evo terms. Although new meters can always be adopted, Evo recommends that during performance analysis, the meters for that

Practice	Description
	performance attribute be defined, within the Performance Requirement Specification. *example p. 233*
Specification quality control through early inspection	When goals or specifications are written, research shows that defects and misunderstanding are likely. Research also shows that early inspection is a powerful, cheap tool to reduce those defects. Note that **specification defects** have a precise meaning in Evo: failure to observe a formal, written, required specification rule. Gilb is an expert in the effective use of inspection— which is *not* the same as informal review. The Evo quality control practice includes *sampling*, and application of the **Defect Detection Process**, and **Defect Prevention Process**. *details p. 230*

Configuration & Change Management

Practice	Description
Specification relationships	The Planguage specification templates contain relationship sections to support requirements traceability. *example p. 234*

Evolutionary Project Management

As with Scrum, XP and UP, Evo's project management philosophy is adaptive rather than predictive planning. And, as with the other methods, there is still attention to the long-term vision, objectives, and a robust architecture. Some controlling principles:

adaptive planning

- **Financial Control**— An iteration should be between 2–5% of the total initial financial budget before delivering some measurable results. This excludes larger capital costs that must be incurred in an iteration, such as buying a server, as these are "backroom" expenses.
- **Deadline Control**— A delivery (or frontroom) iteration should be between 2–5% of total project time, with a lower-bound of one or two weeks. This leads to the official Evo rule of thumb of one-week iterations for a one-year project. The Evo consultant Niels Malotaux has found two-week delivery iterations are more sustainable than one week.
- **Value Control**— Choose design ideas for the next iteration that deliver the best stakeholder value for costs.

With these control guidelines, the next iteration is chosen in response to the latest measurements and evolving understanding of the requirements. A misstep that doesn't deliver expected value consumes no more than (say) 2% of resources.

Future iterations may be tentatively assigned to specific design ideas, and ordered with respect to dependencies, but Evo encourages only very light investment in this kind of predictive planning, as it is central to Evo to adapt the plan at each step.

Unless there is a specific stakeholder request for the next iteration, Evo recommends the use of impact estimation table analysis to choose design ideas for the next step.

impact estimation table

For tracking and adapting, Evo also recommends the use of an impact table to record the results of delivered solutions, and to indicate the steps of the Evo plan. See Table 10.1 for a simplified example after the first iteration.

Table 10.1. simplified Evo plan and results table the capitalization in Evo implies these are terms defined in Planguage elsewhere

Target Requirements	Iteration 1 (plan, actual)	Iteration 2	Cumulative to date
Responsive Browsing	5%, 2%[a]	10%, __	2%
System Reliability	10%, 5%	20%, __	5%
Capital Costs	0%, 0%	5%, __	0%
Development Costs	2%, 2%	2%, __	2%

[a] the percentage of the final target

Regarding evolutionary delivery: A common Evo project management question is, "If I'm making a new plane (for example), how can I deliver it for use by stakeholders in weekly increments?" Although evolutionary delivery of software is often possible—such as bi-weekly refinement to a Web site, or new updates which can be downloaded—this of course will not apply to new products with long development lead times. In this case, Evo's approach is to work on their development in the *backroom*. It could be months before something from the backroom is available for delivery. Meanwhile, Evo still requires that *something* of measurable value be delivered to stakeholders each *frontroom* iteration (e.g., every two weeks). For example, early documentation samples, improvements to the existing system or operational environment, and so forth.

backroom/frontroom

The last point underlines Evo's total systems approach: Do anything that helps. It is not limited to new software or hardware constuction. Gilb believes there is an expensive and risky tendency to avoid looking at the existing system (when there is one) for the desired improvements—sometimes due to technologists' delight in new technologies—and thus he promotes in Evo a preference for considering the existing system as the base for improvement.

Specification Quality Control (SQC) Through Early Inspection

When specifications are created (iteratively), Evo recommends the use of classic systems engineering process control through sampling and inspection. Evo promotes defect removal in specs, done with agility, through its Defect Detection Process and Defect Prevention Process.

Evo's SQC draws from IBM's research and practice [e.g., MJHS90], and Gilb's experience; he is co-author of *Software Inspection* (which emphasizes specification inspection).

A key idea in SQC is that specifications are not informally inspected for any kind of fault; rather, there is only a search for defects—meaning a violation of a written rule from a rule set or checklist that the "checker" is working against. Here's a simplified defect rule set[4] :

[4] Adapted from [Gilb03].

- **Clear**— They must be unambiguously clear to the intended readers.
- **Scale**— Performance and cost requirements must specify a scale of measure to define the concept.

Other key practices in SQC include:

- Two to five checkers for an inspection.
- Specification pages are *sampled* for inspection; the entire document is not checked. If the sampled defect level is above a threshold, the specification is not released for use.
- The checkers do not volunteer solution or correction advice to the author. They only note issues. It is up to the author to determine solutions or take the initiative to ask the checkers for suggestions.

Defect *prevention* in Evo is a process improvement activity that comes from collecting inspection data, reflecting on the results, and experimenting with changes in source workproduct creation.

Planguage

Planguage is Evo's compact specification language. Figure 10.4 shows common notation for one partial specification.

Figure 10.4. Planguage

examples: See "Workproducts"

Workproducts

Full description of Evo's workproducts and how they can be expressed in Planguage is beyond the scope of this introduction. Nevertheless, the following examples provide a sample of Evo's flavor. More detailed examples are given than for the Scrum, XP, and UP chapters, as Evo examples are less well-known and less widely available.

Planguage specifications are incrementally developed over the iterations, and only to the extent that doing so adds value.

Function Requirement Specification

Individual functions are recorded in an Evo Function Specification, using Planguage. These could be a high-level top-ten list of functions, or detailed and decomposed functions. The following example illustrates standard parameters (e.g., "Gist") from the Evo Planguage template. Some statements are purposefully undefined, both for brevity and to emphasize the normal process of partial and evolving specifications in Evo. All capitalized tag elements (e.g., Call Center) refer to other specifications previously defined, probably hyperlinked and clickable. Observe that opinions or "facts" in a specification are sourced to a party; Evo expects claims to have some substantiation, or at least explicit acknowledgment that they are wild guesses.

Tag: FLF:

Type: Function Specification

Planguage

======= Basic Information ==========

"version, status, owner, stakeholders are elided"

Gist: Find lowest fare for air travel.

Description: <input: dates, airports, carriers. output: flights sorted by cost>

========== Relationships ==========

Relationships: Evo supports requirements traceability in this section.

Supra-functions: Res.Search

Sub-functions: none

Is Impacted By: { Call Center, Web Front End }

Linked To: Supports: Res.Booking

========== Measurement ==========

Measurement: Goals in Evo should be testable and measurable.

Test: T1: <correctness test 1>

========== Priority and Risk Management ==

Rationale: <Our competitors have it> <- Marketing Director

Assumptions:

A1 [before end of next year]: Competitor X doesn't upgrade

A2: < ?? >

Dependencies: Res.DB

Risks: R2, R6

Priority: Must be in first public release <- Marketing Director

========== Specific Budgets ==========

Financial Budget: < ?? >

Performance Requirement Specification

Individual performance requirements (quality, workload capacity, resource saving) are recorded in the Planguage form shown in this example.

Tag: Responsive Browsing:

Type: Workload Capacity Requirement: Response:

Budget: < ?? >

========== Basic Information ========

"version, status, owner, stakeholders are elided"

Ambition: <Many> Res.Users with <acceptable> response time.

========== Measurement ==========

Measurement: Illustrating the quantifiable emphasis in Evo.

Scale: Average HTTP response time in seconds

Meter: Automated HTTP server monitor

========== Targets ==========

Goal

[First Release]: response under 3 seconds for up to 1,000 requests per second <- Marketing,

[Second Release]: response under 2 seconds for up to 1,000 requests per second <- Marketing

========== Constraints ==========

Fail [First Release]: response over 6 seconds <- Marketing

========== Benchmarks ==========

Past [Old System, last year]: response under 5 seconds for up to 1,000 requests per second <- ABC Research Report

Record [CompetitorY, this year]: response under 1 second for up to 3,000 requests per second <- ABC Research Report

========== Relationships ==========

Relationships: Illustrates the performance requirement relates to other performance requirements (or perhaps, directly to functions).

Is Impacted By: Res.DB.Response <- DBA

Impacts: Usability

========== Priority and Risk Management ==

Value <this level will retain 95% of first-time users> <-Marketing "assumptions, dependencies, etc."

Design Specification

Design ideas are expressed in the Planguage template form demonstrated in this next example.

Tag: Server Cluster:

Type: Design Idea

========== Basic Information ==========

"version, status, owner, stakeholders are elided"

Gist: Cluster of 10 application servers with an IP sprayer.

Description: < ?? >

========== Design Relationships ==========

Design Constraints: { Use Moon Spark 5000s, Use Java Technologies, Use Open Source }

Sub-Designs: < replication, fail over >

========== Impacts Relationships ==========

Impacts: design ideas must be connected to functions and/or performance requirements

Impacts [Functions]: { Res.Search, Res.Transaction, Res.Browse }

Impacts [Intended]: { [Good] Responsive Browsing, [Good] System Reliability, <more> }

Impacts [Cost]: { Operations Budget, [if not open source] Development Budget }

Impacts [Other Designs]: { Deployment Model, Data Model }

Value: < meeting responsiveness and reliability goals will maintain customer retention at 95% <- Marketing Director >

== Impact Estimation of Design on Selected Requirements ==

Impact Estimation: a design idea should contribute to performance objectives. Its impact on each is analyzed.

Tag: Responsive Browsing

Type: Performance Requirement Cross Reference

Scale: Average HTTP response time in seconds

Scale Impact: under 3 seconds for up to 1,000 requests per second

Scale Uncertainty: ± 1 second <- Jill Jones

Note that claims are sourced, and uncertainty and credibility estimated.

Percentage Impact: [if Use Moon Spark 5000s] 100%[5]

[5] From some baseline (such as "Past") in the requirement.

Percentage Uncertainty: ± 33%

Evidence: CompetitorX has this configuration and response

Source: Jill Jones (Chief Architect)

Credibility: 0.5 as Jill worked for CompetitorX on similar project

Tag: System Reliability

"repeat analysis using the above set of parameters"

========== Priority and Risk Management ====

"assumptions, dependencies, risks, priority, issues are elided"

Impact Estimation Table

This tool is used in Evo to analyze the impact of alternative (or complementary) design ideas on performance requirements. Barring "obvious" priorities for the next iteration as indicated by stakeholders, this table is used to rationally choose the set of design ideas to implement next, based on the best benefit-to-cost ratio. Note that the horizontal and

vertical summing of impact percentages do not always accurately predict a result; they may or may not provide a sense of aggregate impact. For example, can one sum the *Responsive Browsing* impact of both a server cluster and high-performance hardware? Perhaps…

Table 10.2. simplified impact estimation table

Design Ideas -> Requirements	**Server Cluster**	**High-performance hardware**	Sum of Impact[a]
Responsive Browsing Baseline: 5 sec. Goal: 3 sec.			
Scale and % impact[b]	3 ± 1 sec. 100% ± 50	4 ± 1 sec. 50% ± 50	150% ± 100
Evidence and Credibility	CompetitorX has this configuration and response <- Jill Jones 0.2	Moon Microsystems has customers achieving this <- Moon Sys Eng. 0.1	
System Reliability Baseline: 3000 hours MTBF. Goal: 3500			
Scale and % impact	3200 ± 200. 40% ± 40	3100 ± 200. 20% ± 40	60% ± 80
Evidence and Credibility	CompetitorX has this config and "suspected" reliability <- Jill Jones 0.2	Moon Microsystems has customers achieving this <- Moon Sys Eng. 0.1	
Sum of Impact[c]	140%	70%	
Capital/Dev Cost Baseline: $0 USD. Budget: $200K			
Amount and %	$20K ± 10K. 10% ± 5	$100K ± 10K. 50% ± 5	60% ± 10
Evidence and Credibility	Bob's friend guesses this cost on another project <- Bob Bones 0.1	Moon firm quote <- Moon Sales Rep. 1.0	
Benefit-to-Cost Ratio[c]	14 (140% / 10%)	1.4 (70% / 50%)	
Impact Credibility Adjust Cost Credibility Adjust	0.84 (14 * 3 *.2)[d] 0.08 (0.84 * .1)	0.01 (1.4 * .1* .1) 0.01 (0.01 * 1.0)	

[a] Sum of impacts on a requirement may or may not be cumulative.

[b] The % impacts are with respect to the baseline.

[c] The sum of impacts of one design idea may or may not be cumulative. The total may or may not work as an estimate for comparison.

[d] Multiplying probabilities is a heuristic to reduce total to a reasonable magnitude.0.08 (0.84 * .1)

There is a lighter alternative (for prioritization) to these tables that Evo also offers: the use of simple benefit-cost estimates: Each design idea is given a 0-9 ranking for both benefit and cost. Ideally, this is in a group "delphi" ranking session. The best benefit-to-cost ratio ideas are implemented next.

Other Practices and Values

Evo has many detailed practices, tips, and guidelines. A sample of points:

- **Open-ended architecture**— To support evolving or changing designs, and evolutionary delivery, Evo encourages open-ended architectures that encourage easier extension. That is, at predictable variation points, some kind of protection is introduced, such as an interface, data-driven declarations, and so forth.
- **Safety facto**r— The estimated impact of a design should deliver an estimated impact with a defined safety factor, default factor 2 (200% over the target level from the baseline).
- **Client-driven planning**— If you are uncertain which step to do next, ask your dominant stakeholder.
- **Whatever adds value**— Rather than a "we are building it" paradigm, focus on "what can I do for my stakeholders next week?" The techniques (such as Planguage and Impact Estimation) are only support to keep this focus, and should not get in the way.

Values

Evo's key values include:

- Learn rapidly by realistic measurement.
- Deliver real value to stakeholders early, frequently, at every step.
- Be humble about complex systems: simplify and attack problems one small step at a time
- Delegate power to the ultimate user, by focusing on end results and not methods and well-intended bureaucracy.
- Admire, applaud and reward a team based on the flow of measurable results: stakeholder value versus costs.

Common Mistakes and Misunderstandings

or, How to Fail with Evo

Error: Adoption mistakes— Lack of management support. Lack of training in concepts and methods. Lack of clear quantified management objectives as the basis for evolving towards Evo methods. Lack of clarity about the management objectives of using the method — and how to measure these improvements in practice. Lack of a good successful pilot project to prove it works in your environment. Lack of dramatic motivation to change from older methods.

Error: Lack of focus on results— Self-explanatory.

Error: Giving up or not believing short iterations are possible— Giving up too easily when managers or engineers claim they cannot find small early steps (they need training, motivation and help). Giving up too early and falling back on old habits.

Error: Lack of management encouragement— When a team starts delivering something of value every short iteration, that's often a revolutionary event. Management needs to praise and encourage this result, rather than take it for granted.

Error: Failing to use value/cost priority— Not choosing solutions based on highest value-to-cost.

Error: Customers not involved— Evo is customer and results-driven; they need to participate in providing feedback on the results of evolutionary deliveries, and in steering the next iteration.

Error: No measurements— It is a mistake to avoid regular measurement of the impact of delivered solutions. Frequent numeric measurement is a significant shift for many managers, but central to Evo.

Error: Iterations too long— Evo frontroom iterations should be 2–5% of total project time, with a lower bound of one or two weeks.

Error: Each iteration does *not* end in a delivery— Evo is about evolutionary delivery on a "weekly" basis to real stakeholders for useful results, even when backroom development may take months.

Error: Predictive planning— It is a misunderstanding to create, *at the start of the project*, a believable plan laying out exactly how many iterations there will be for a long project, their lengths, and what will occur in each. This is contrasted with Evo or adaptive planning. The Evo team and customer plans the next iteration, and then planning adapts iteration by iteration, based on measurement and feedback.

Sample Projects

Gilb's view is that any project applying IID and evolutionary delivery is an example of Evo. This of course covers thousands of projects. For an early example, the mid-1970s LAMPS project described on p. 83 is considered an Evo project in Gilb's classification.

Process Mixtures

None of the other IID methods covered emphasize weekly evolutionary delivery, and related Evo project management measurement.

Evo + Scrum

Most Scrum practices are compatible with Evo. The Scrum meeting, common project room, and demos to external stakeholders at the end of each iteration enhance Evo's feedback goals. The Scrum backlog and progress tracking approaches are also applicable additions. Scrum does not discuss specific specification methods, and thus Evo's Planguage is still applicable.

Evo's measurement emphasis is compatible; indeed, Jeff Sutherland, one of the Scrum creators, takes a strong interest in measurement when applying Scrum.

Scrum's unchanging 30-day iteration length is not consistent with Evo—Evo iterations are usually shorter.

Evo + UP

The UP is especially for software development, and usually for projects involving multiple iterations before production delivery. Consequently, the UP could be applied to Evo

backroom development work. However, Evo's evolutionary delivery and project management styles are not exactly in the same spirit as the UP, although both share an interest in early identification and mitigation of risks.

The UP has its own set of workproducts and approach to requirements capture: the Use-Case model (and thus, use cases), and Supplementary Specification for description of functions, features, and non-functional requirements. Evo Planguage elements, such as the Performance Requirement Specification, may be used within the UP Supplementary Specification.

Evo's measurement emphasis is compatible or acceptable with the UP.

The upper bound of UP's 2–6 week iteration length is not consistent with Evo—too long.

Evo + XP

XP values and spirit regarding specifications are not exactly compatible with Evo. XP's value of avoiding written or precise requirements, and preferring oral communication between developers and requirement donors is different than Evo's emphasis that when a specification is required, it be written with clarity and measurable qualities. However, Evo allows a scaling down of precision on small projects; the important Evo point is value to the client, and precision is an optional means to that end.

On the other hand, many XP development practices may be consistently applied with Evo, such as test-driven development, pair programming, and so forth.

XP's emphasis on early results and customer-driven adaptive planning is also consistent with Evo. The XP practice of stand-up meeting, common project room, and whole team together supports Evo's feedback goals.

XP's 1–3 week iteration length is consistent with Evo.

Adoption Strategies

As always, coaching by an experienced method expert on the first project is recommended. Evo is results oriented, so not much is sacred in its adoption—other than frequent evolutionary delivery and project management.

Clear, precise, and measurable (though evolutionary) requirements are not that common or enthusiastically developed. One approach to motivate their adoption is to focus early on evolutionary delivery, which of course demands understanding the design ideas, requirements, alternatives, and priorities. Thus, after a few iterations, the participants themselves will better appreciate the value in adopting something like Planguage and greater requirements precision, in order to guide choosing their next step and evaluating the results of the prior one.

Gilb recommends the use of pilot projects to demonstrate the value and viability of Evo.

Fact versus Fantasy

Impact estimation tables are not consistently used by Evo adopters. This may be due to their requiring more analysis and complexity than the priority problem often warrants. As mentioned, a less detailed 0–9 scale for benefit and costs ratios is an Evo alternative.

One-week evolutionary delivery iterations are difficult to sustain; two weeks is easier.

Gilb reports that a significant number of Evo adopters find quantification of their most critical objectives difficult without some coaching.

Evo's PDSA emphasis requires not only estimation and planning, but measuring. Yet, this last step is often dropped under the pressure of work, which of course makes Evo planning less useful.

Strengths versus "Other"

Strengths

- Early, visible results; frequent delivery to stakeholders.
- Measuring the impact of solutions and guiding improvement by measurement data, rather than only by informal guess.
- Customer participation and steering.
- Worker engagement and satisfaction from seeing their solutions quickly implemented.
- Planguage is a simple and compact approach to requirements specification.
- Evolutionary and incremental requirements and development, and adaptive behavior.
- Emphasizes quality through proven inspection methods and through continual process improvement based on measurement and data.
- Practices from other methods (e.g., Scrum or XP) easily included.

Other[6]

[6] Could be viewed as a weakness, strength, or deliberate desirable exclusion depending on point of view.

- Management and requirements overhead of estimating impacts and measuring results.
- As with Scrum, minimal guidance within software-specific disciplines, as Evo is a general project management and systems engineering method.

History

Gilb started some Evo practices in the early 1960s, while consulting (and living) primarily in Europe. In 1976, he wrote about iterative development, evolutionary delivery, and evolutionary project management in his book, *Software Metrics*. This was rather unique in a period dominated by waterfall lifecycle promotion. In the late 1970s, he authored a series of column articles in *Computer Weekly UK* that reiterated and further explored these practices; these articles are arguably the earliest popular press on the subject of IID and adaptive, evolutionary development.

In April 1981, Gilb published "Evolutionary Development" in *ACM Software Engineering Notes*, and in July 1985 published "Evolutionary Delivery versus the 'Waterfall Model'" *ACM Sigsoft Software Requirements Engineering Notes*. These are some of the earliest ACM or IEEE publications related to the subject of IID and adaptive, evolutionary development.

In the 1980s he was also exposed to the work of Deming, and realized that Deming's values and Shewhart's PDSA model captured the intent of Evo.

As mentioned in the introduction, in 1988 Gilb published *Principles of Software Engineering Management*, a milestone early book describing an adaptive, iterative, and evolutionary process, well ahead of its time.

Since then, his early work and Evo have influenced many other methods: XP, Scrum, and the UP all owe debts to Gilb's work. The popular book *Rapid Development* [McConnell96]—

which examines many key best practices in software development—cites Gilb's work in 14 sections.

What's Next?

The next chapter examines some method practices in more detail, and introduces other common tips. The final chapter is a FAQ.

Recommended Readings

- Gilb's 1988 *Principles of Software Engineering Management* is an important step in studying Evo. His 2003 *Competitive Engineering* presents updated refinements, and the details of Planguage; it is the current basis for studying Evo.
- *Software Projects: Evolutionary versus Big-bang Delivery*, Felix Redmill, John Wiley & Sons, 1997. Redmill learned Evo from Gilb in the 1980s and managed projects with it. This book describes his experience and lessons learned.
- Free online articles and draft books by Gilb—on Evo subjects—are available at his Web site: www.gilb.com.
- Useful elaboration and refinements for Evo are also available for download from Niels Malotaux at www.malotaux.nl.

Supporting or related texts that are recommended include:

- *Software Inspection*, by Tom Gilb and Dorothy Graham.
- *Out of the Crisis*, by W. Edwards Deming.
- *The Deming Management Method*, by W. Edwards Deming and Mary Walton.
- *Quality Is Free: The Art of Making Quality Certain*, by Philip Crosby.

Chapter 11. Practice Tips

Prediction is very difficult, especially if it's about the future.

—Niels Bohr

OVERVIEW

- Tips for agile or iterative projects, organized into categories such as Project Management and Requirements.

  - **For example,** *Project Management*: **Multisite iterative development.** *Environment*: **Continuous integration.** *Test*: **Test-driven development.**

What practices are applied when working on iterative and agile projects? Many answers are found in the method-specific chapters. This chapter offers other miscellaneous "tips of the trade" and adds detail to some mentioned elsewhere. Regarding the scope and detail: This chapter (and book) is an introduction.

Naturally, many questions arise when first adopting an iterative method, such as:

- How does one plan the iterations?
- How does continuous integration work?
- What does test-driven development look like?

The answers are organized in common discipline categories: *project management, environment, requirements*, and *test*.

Project Management

This section summarizes some tips related to iteration timing, planning, tasks, tracking, and other project management practices to support iterative or agile development.

There is a trend in some of these tips: The manager is not alone in doing the work of planning, scheduling, estimating, tracking, and so forth. In agile development, it is more of a team sport.

Multiteam or Multisite Early Development

For projects that will be composed of multiple teams, perhaps spread across different locations, consider doing the early iterations at one location, in one common project room, with a small group ideally composed of one or two skilled representatives from each of the subteams. During these iterations, there is an emphasis on requirements analysis and development to discover and build the core architecture of the system—the foundation. Major components, and their collaborations and interfaces are ideally clarified through early programming and testing rather than just speculative design. If UP is your method, this covers the inception and elaboration phase.

In this way, the early project benefits from the close communication, collaboration, common vision, and technical strength of the initial group.

Once the core is built, the representatives return to their respective locations, form larger teams, and the remaining work is done in parallel with multiple subteams. Each representative has developed a clearer picture of the vision and architecture, and can better convey and maintain that for the remainder of the project. Further, each acts as a liaison to the other teams. Also, after having spent some close time with the other subteam leaders, there is improved communication between the subteams. Figure 11.1 illustrates.

Figure 11.1. multiteam development

Difficult Multiteam or Multisite Iteration Planning

It is a worthy goal to have all components, across all subteams or sites, integrated and tested together to conclude the release of a common iteration. However, there are times when relaxing this goal is helpful. For example, consider the example of a multisite (five sites) research-oriented project spread across 16 time zones, working on a 3G telephony system that included creation of everything from the 3G handset to the protocol stack [Crocker02].

In this case, due to barriers in time, communication, and the difficult-to-predict research-oriented development, it was seemingly impossible to get all teams to coordinate and complete all work by a common iteration end date. For example, one team working on one layer of the system might need much longer than expected, or than the other teams, to complete the supporting goals of the iteration.

The successful solution was to establish the rule that at least two of the five teams had to define a common, relatively short iteration with integration between the two (or more) teams, and that no team could go "too long" without participating in a joint iteration. All the subteam project managers and technical leaders would adaptively plan the next iteration. In this way, the product elements were integrated in a series of disjoint iterations. Figure 11.2 illustrates.

Figure 11.2. sub-team iterations

This approach should be the exception rather than the rule. Although helpful for this multisite research-oriented project, the preferred guideline is that all teams work together towards a common goal and integrate on a common iteration end date.

Overlapping or "Pipelining" Activities Across Iterations

Some projects benefit from "pipelining" or overlapping certain activities across iterations, usually requirements analysis and testing [Larman97, JPKP03]. Figure 11.3 illustrates. For example, during iteration N, one or more people are doing requirements analysis for future development in iteration N+1, and a test team is evaluating the release of iteration N–1. In the latter case, defects discovered by the test team are handled in the current iteration if there is planned slack in the iteration schedule for this (and the defect is small), or deferred for a future iteration.

Figure 11.3. pipelining iteration work

Note that if there is a test team evaluating a prior release (probably for difficult or long-running tests), there should still be plenty of testing for the current release. The release should be as stable and fully tested as possible.

There may be efficiencies with pipelining, but not all projects can or should apply this approach. Furthermore, although the example shows pipelining of both requirements and test, it may of course be applied with only one activity. It is more appropriate for larger projects, those with offsite requirements donors, or those with long, complex testing that must be performed by a separate team. Examples of complex testing includes labor-intensive manual or semi-manual GUI testing, and memory-leak stress testing where the system needs to run for many hours or days to discover subtle leaks.

Pipelining has a seductive aura of efficiency when you look at a chart such as Figure 11.3, but as the agile methods stress, projects and people are not machines or computer processors.[1] There are soft issues that carry potential pitfalls [Kruchten00a].

[1] The term itself was inspired by processor architectures.

One pitfall is the slippery slope of overlap and focus. It may succeed for requirements and test, but can be part of a trend to a more general lack of current-iteration focus in other disciplines, such as programming. Working on past or future iteration "stuff" is not in the spirit of "let's work together on the next short step."

Another, related pitfall is a reduction in synergy, cross-discipline insight, "whole team together," "generalists over specialists," "customer close to developers," feedback, and adaptation—all qualities promoted by the agile methods. Pipelining can re-introduce some of the deficiencies of a sequential waterfall model. It requires separation of people by discipline. For example, the developer may no longer be involved in requirements analysis (such as at a requirements workshop), yet research shows this is desirable [KC94]. In a degenerate case, if design work was also pipelined, the programmer would not be the designer. This latter situation is especially inconsistent with agile and adaptive development, where the design emerges from an interplay of some up-front speculation or design thought, combined with programming and testing to prove, disprove, and adapt the evolving system. The programmer and designer must be one and the same, or where design specialization is required—such as in database or UI design—the programmer and designer need to collaborate within the current iteration on common goals. In general, pipelining can reduce interaction and feedback between the disciplines and people; that's undesirable.

In conclusion, it's a double-edged sword.

Rolling Wave Adaptive versus Predictive Planning

One of the big ideas of agile and iterative development is to adapt based on feedback, and this is not only with respect to requirements and design, but also the plan or schedule.

This is the topic of **adaptive planning** versus **predictive planning** [Fowler01]. Adaptive planning in IID methods is a refinement of the well-known **rolling wave planning** concept—more on this later. It implies that there isn't a detailed plan of all the future iterations. There is no fixed plan of how many iterations there will be, how long they are, or what will happen in each. In contrast, predictive planning implies there would be such an iteration-by-iteration plan to the project end.

This does not mean there are no large-grained milestones with dates, or no thinking ahead. In adaptive planning, there *are* milestones with dates, (or at least, there can be) but the path of iterations to those milestones is left flexible or adaptive. And, the milestones themselves may change if in the best interest of the project.

Rolling wave or adaptive planning is a key idea in agile methods. Comments on why this is a good idea—in fact, a better approach than a detailed speculative schedule—will come later, but first an example to help clarify the ideas:

Imagine that after two short exploratory iterations (that include programming and test) into a project, there is enough requirements clarification to be able to say with some certainty that the overall high-level requirements (e.g., use cases and features) are R1, R2, …, R20. Plus, there are rough effort estimates for each, and some understanding of their dependencies. Perhaps R2 must be done before R7, and so forth.

At this point, the customers indicate that they would like to see R1 through R10 by roughly mid-project, and the remainder by the end. The customers ask for an estimate of when R1–R10 can be finished. The customers may be asking for this milestone because of a demo they want to schedule for investors, a trade show, to synchronize with other product development, or simply because they are used to the idea of milestones.

The team does the best it can in estimating R1–R10, and believes it can finish in three months, on July 1, given an estimate of average developer availability. Likewise for R11 to R20, with a completion of September 1. Thus, two milestones have been established, and the team makes a commitment to meet these goals, with an understanding that change is possible later, in light of new insights or priorities. Figure 11.4 illustrates.

Figure 11.4. milestones estimated

Note that these milestones are more coarse grained than iterations; there may be many iterations up to the first milestone.

Predictive planning would go further than this, it would speculate a week-by-week or iteration-by-iteration schedule of the path to these milestones. For example, that there will be three iterations before milestone-1, each four weeks long, that R1 and R2 will be done in iteration-1, R3–R5 in iteration-2, and so forth. See Figure 11.5.

Figure 11.5. predictive plan

Rather, in **adaptive planning**—and this is the key point—*we primarily plan in detail for just the next iteration*. This is not an absolute rule. We may see some obvious dependent work to do in the follow-up iteration, or we may know that Jill the database expert will only be available in June, or that the new servers we ordered will arrive July 15. Such presence or absence of resources places obvious constraints on when we schedule certain tasks or requirements. But, all other things being equal, we primarily focus on deciding what to do in the next iteration, and defer making decisions about future iterations, unless obvious or necessary. See Figure 11.6.

Figure 11.6. adaptive plan

In agile and adaptive planning, the precision is commensurate with the information; the level of detail and commitment to fine-grained scheduling drop as we look further into the future.

Avoiding far-future predictive planning and preferring short-term adaptive planning is not a new or radical idea. It is also called **rolling wave planning** [Githens98], a well-known practice promoted in the PMBOK (Project Management Body of Knowledge) [PMI00], many

management courses and texts, and government organizations. That said, there is often a gap between published management recommendations and actual practice or attitudes.

Government oversight agency: *Oh yes, we promote rolling wave planning, as indicated in Standard-773-1. When will the weekly schedule and PERT chart for our new project be finished?*

Regardless, the essence of rolling wave planning is to plan in detail only up to some realistic planning horizon, beyond which things are too speculative. In agile methods, that horizon is the next iteration.

What plan do external stakeholders see? If they want or need to establish milestones, they see a coarse-grained milestone-oriented schedule. A path of iterations to those milestones is not shown.

Finally, to re-emphasize points made before (since this is a common misunderstanding), adaptive planning *does* allow for coarse-grained milestones with dates, and can allow for some distant iteration plans when there are obvious resource constraints.

Benefits of Rolling Wave Adaptive Planning

Why is this useful? Adaptive planning is closer to optimal in terms of working towards milestones; each step can be the most skillful we know how to plan regarding risk, productivity, and effectiveness because each planning step is taken with maximum—and fresh—information. We take a step, and then ask, "Given what we now know, what is the most skillful thing we should do in the next step to work towards our milestone goal?" And repeat.

In contrast, a predictive plan is suboptimal. In fact, it could be close to the worst or most risky possible path to the milestone goal because it is created with the least amount of information, speculating into the far future. The schedule in Figure 11.5 could be a very poor path to the goals; we can't know. It is not wise or useful to believe that five iterations in the future, 17 weeks from today, the best thing we should do is implement requirement R15.

Not only is predictive planning suboptimal with respect to effectiveness and risk, it doesn't account for opportunities. If half-way into the project the marketing manager discovers our competitor is adding the sexy Gromlit feature, a predictive plan does not account for this. Yet, an adaptive plan does; the team can start adding Gromlits the very next iteration. Adaptive plans embrace change and opportunity; predictive plans fight or ignore it.

In addition to supporting adaptation, another key, basic reason for not creating an early detailed schedule is that in evolutionary methods not all the requirement and design details are known near the start of the project. Two weeks after project initiation, you can't plan for use case X in iteration-7 when you don't even know about use case X.

Conclusion

Although rolling wave or adaptive planning has been promoted for years in leading management circles, there are still some inconsistent software project management texts and courses that teach it is ideal to create a detailed predictive schedule or PERT chart of week-by-week activities through to the end of a software project. And, that failing to follow it is a sign of lack of skill on the part of a manager. Ironically, deviating from a speculative plan in response to risks or opportunities is a sensible response, not a failure.

As with the waterfall model, the heart of the problem with predictive planning is that it is misapplied to software projects. It works for mobile phone manufacturing; software is a domain of new product development, not predictable manufacturing.

Planning: Consider Ending on "Wednesday"

If wrapping up an iteration is nontrivial (for example, on larger projects with several subteams), consider making the last Wednesday (or Thursday) rather than Friday the goal date for baselining the release. Although the goal is definitely Wednesday, allow Thursday and Friday as slack days for unanticipated problems.

Thursday and Friday also provide time for group planning of the next iteration, perhaps another requirements workshop, a group iteration assessment or review, a demo to stakeholders, and so forth.

Further, if there is a separate test team that receives the end-of-iteration release (pipeline testing the release of iteration N-1 during iteration N), make Thursday the day when that team expects to start [Blaustein03], again allowing two slack days. This approach decreases the likelihood of working over the weekend to meet a Monday start-of-iteration hard deadline, when everyone is expected to move forward on a new iteration rather than still be working to wrap up the release to hand over to the test team.

Planning: Whole Team Planning Meetings

Agile methods emphasize collaborative planning. An initial release planning meeting and each iteration planning meeting are ideally held with all developers and customers present. If the project is large and composed of subteams, at least include representatives from each subteam.

Planning: Workers Estimate

The agile methods promote the value that the workers doing the work must estimate the work. At the planning meetings, involve the developers in estimation of large-grained requirements that will be allocated to iterations, and also estimation of the related finer-grained tasks. In XP, for example, it is required that the worker who volunteers for a task must be the one who estimates it.

Planning: Improving Estimates with Wideband Delphi

Estimation is a large topic mostly beyond the scope of this introduction. However, in addition to the agile methods practice of workers estimating their tasks, it is worth considering the technique of **Wideband Delphi** for large or valuable projects [Wiegers00]. This is an iterative, adaptive method (hence its inclusion in this chapter) complementary to other estimation methods.[2] The practice is common in successful outsourcing companies whose estimates have low variance to final actuals.

[2] First developed as the Delphi method at RAND Corp in 1948, and refined as Wideband Delphi by Barry Boehm in the early 1970s.

A key point is that Wideband Delphi is used to *complement* an existing estimation method, not replace it. Here are the steps:

1. Kickoff Meeting: Get at least three people (or three pairs) together to estimate. Discuss the source documents and project for which an estimate is to be made. Discuss the units of estimation. End the meeting.

2. Estimation: Then, each person (or pair) creates estimates. This could take minutes or days, depending on the scope. Any specific estimation technique can be used: COCOMO, micro-estimation, etc.

- Each person/pair creates three estimates, sometimes called the PERT estimates: 1) most likely, 2) optimistic, 3) pessimistic.

3. Meeting: Each estimator gives their estimates to a facilitator, who displays them (see Figure 11.7), with averages. In the purest form of this method, the owners of the estimates are not revealed to the whole group, to reduce the influence of personality or seniority. Finally, each estimator discusses their insights, problems, and assumptions.

Figure 11.7. Wideband Delphi sample data

	(person days) Pesimistic	Most Likely	Optimistic
Round 1			
estimates	(900, 1000, 950)	(800, 900, 850)	(700, 800, 800)
averages	950	850	767
Round 2			
estimates	(1000, 1100, 1050)	(900, 950, 875)	(800, 900, 850)
averages	1050	908	850

4. Repeat steps 2 and 3 at least once.

- Note that this is iterative estimation refinement. The point of this step is to provide the feedback to drive adaptation and improvement in the next iteration of estimation.

5. Calculate the final numbers with the following PERT formula, using the averages from the final cycle.

$$Estimate = \frac{Optimistic + Pessimistic + 4MostLikely}{6}$$

$$LikelyDeviation = \frac{Pessimistic - Optimistic}{6}$$

Note that Wideband Delphi sits on top of any other estimation method, improving it through multiple participants, feedback, and iterative refinement.

Planning: Multiple Iteration Planning Meetings

As discussed, agile methods apply adaptive planning; thus, at each iteration (usually near its end), there is an iteration planning meeting for the subsequent iteration, during which it is decided what to do in the next, followed by detailed task generation.

Planning: Agile Task Generation

During each iteration planning meeting, the team generates finer-grained tasks (for example, in the half-day to two-day range) for the iteration requests. XP promotes the approach of either having the team work together to write these on a whiteboard, or each team member brainstorms tasks and writes them on cards, that are grouped and stuck on a wall. In other words, an "agile" approach to a work breakdown structure is performed.

Re-use is worthwhile here; many tasks repeat across iterations and projects. If the final task list is only handwritten on a wall, consider taking a photo of each version, and share these for inspiration at future planning meetings.

Planning: Don't Forget to Budget for Iteration Overhead Tasks

More frequent risk management, iteration demos, iteration planning meetings, a group iteration assessment, a daily Scrum meeting, some average rate of unexpected defects, infrastructure failures, and so on, all consume time, and need to be reflected in the iteration task list. Managers new to iterative development sometimes miss accounting for these overhead tasks.

Planning: People Estimate Their Time Budget Each Iteration

Timeboxing requires a realistic approach to time management. People have to avoid overcommitting, as there is not much slack. So, it is helpful to calculate personal time budgets—also an XP practice. During each iteration planning, people estimate their total "ideal work hours" or uninterrupted attentive time for project tasks for the iteration. This analysis may take five or ten minutes. A common average is around five hours per day—if it is much higher, be skeptical.

Planning: Volunteering

A simple practice related to commitment, motivation, and job satisfaction is to promote volunteering for tasks rather than task assignment by a manager. This approach is part of XP and Scrum.

During the iteration planning meeting, after task generation, people volunteer for enough tasks to get busy. As the days pass and they finish tasks, they volunteer for more. Hold each daily Scrum by the "task list" wall, and the team has the basic information necessary to volunteer.

What if no one volunteers for a particular task? Rather than answer the question prescriptively, it is instructive to learn the agile project management attitude in a method such as Scrum. In Scrum one would respond to this problem by saying, "It isn't the project manager's responsibility to solve this, although she can offer advice or resources in response to a request. We work by self-directed teams, not manager guided. It is the team's collective problem to solve it." This level of self-direction and decentralization of responsibility is a significant value change for some managers or organizations.

Visible Project Plans

Where possible, prefer to show all iteration tasks on a wall or whiteboard. If a daily Scrum meeting is practiced, hold the meeting by this wall. During the meeting, when tasks are reported complete or added, they can be crossed out or written in; likewise with the project blocks or impediments.

Iteration Goals: Risk, Coverage, Criticality, Skills Development

What should be done in the earliest iterations? Rank requests and iterations by risk, coverage, and criticality.

Risk includes both technical complexity and other factors such as uncertainty of effort, poor specification, political problems, the need for a novel UI look and feel, or usability.

Coverage implies that all major parts of the system are at least touched on in early iterations—perhaps a "wide and shallow" implementation across many components. The

goal is to discover and stabilize the major software and hardware components, their interfaces, and collaborations. That's an important part of the overall architecture. For example, early use case scenarios may be chosen that require execution (and thus development) across many of the components (e.g., from UI to database). These are called **architecturally significant use cases**.

Criticality refers to functions of high business value; these need at least partial implementations in the earlier iterations, even if not technically risky. This driver may be considered a kind of political or business risk: if the paying customer does not see early evidence of things they care about, their confidence or sense of collaboration with the project team drops. Early inclusion of marketplace-relevant features also makes it easier to demo the product and garner attention.

On some projects, another early driver is skills development—one goal is to help the team master new skills such as adopting an agile method or object technologies. On such projects, skills development is a heavily weighted factor that tends to re-organize the iterations into less risky or simpler requirements in early iterations, motivated by learning rather than risk reduction goals.

Iteration Goals: What to Rank?

Use case scenarios, features, defects, and nonfunctional requirements (e.g., internationalization) can all be ranked with the above criteria of risk, coverage, and so on.

Use cases are often composed of many scenarios. For example, the "Process Sale" use case has one scenario involving paying by credit, and another for paying by cash. Some of these use cases are too complex to implement all scenarios within one reasonably short iteration; thus, they need to be decomposed by scenario for both ranking and scheduling.

Therefore, include all types in a ranking list.

Request	Type	…
Process Sale–pay by credit	scenario	
logging	feature	
Handle Returns	use case	
log-in window not closing	defect	
…	…	

Iteration Goals: How to Rank? Dot Voting

Based on the drivers, requests are ranked. The ranking may be informal and qualitative, generated in a group meeting by members mindful of these drivers. This approach is in the spirit of agile modeling.

Consider using the rapid scoring method of **dot voting**. List the requests on a whiteboard, or display them from a computer projector on a whiteboard. Everyone gets a whiteboard marker pen with which they can make 20 dots (for example). As a group, and in silence (to reduce influence), all approach the board and apply dots beside the items, reflecting the voter's priorities. A voter can assign many dots to one item, or distribute them. On completion, sort and discuss.

That may be sufficient. A slight refinement is to do a second round of silent dot voting to reflect updated insight based on first round voting and discussion. This second round provides the feedback and adaptation by which decisions improve.

The requirements ranking will be done before iteration 1, but then again before iteration 2, and so forth. In this way, the scheduling of iterations adapts to current information and opinion.

Iteration Goals: How to Rank? Quantitative Methods

Group discussion and something like dot voting for request ranking are often sufficient—a collaborative, fast, and fuzzy approach very much in the agile spirit. For the more quantitatively minded, variations on the following have been used. The example values and weights are only suggestive; the point is that numeric values, weights, and weighted sums can be used to reason about priorities.

Request	Type	AS	Risk	Criticality	W. Sum
Process Sale-pay by credit	scenario	3	2	3	15
Logging	feature	2	0	1	5
Handle Returns	use case	1	0	0	2
. . .	. . .	. . .	. . .	. . .	. . .

	Weight	Range
AS: achitecturally significant	2	0-3
Risk: risk, complexity, novelty	3	0-3
Criticality: early high biz value	1	0-3

The exact values should not be taken too seriously, and note that the numbers don't tell the whole story. Even though logging is a low-risk, simple feature, it is architecturally significant because it needs to be integrated throughout the code from the start. It would be awkward and would diminish architectural integrity to add it as an afterthought.

Iteration Goals: Related Iteration Length?

Viewing an iteration like any project, the choices boil down to fixing time and filling it with work, or fixing work and filling it with time. The specific approach varies in some IID methods (such as the XP Planning Game), but general outlines follow.

Iteration Length Chosen First

First, comments on factors that affect iteration length:

- Most IID methods recommend in the 1–6 week length. Scrum is specific: 30-calendar-day iterations.
- Smaller teams (such as 10 people) tend towards 1–3 weeks; large projects with sub-teams (such as 200 people) need more time (4–6 weeks) because of the overhead.
- Early iterations with high rates of discovery and change need more time. Later stable iterations can be shorter.

The steps:

1. Decide the length of the next timeboxed iteration.

2. Estimate total developer "ideal work hour" (focused effort on tasks) availability for the iteration, taking into account the usual overheads and lost days.

3. Choose a high-priority request (feature, use case, defect, and so on). If there is no effort estimate, or if it needs review, a new estimate is created.

4. Repeat step 3 until all available resources for the iteration are consumed.

Requests Chosen First

1. Estimate a per-day average total developer "ideal work hour" (focused effort on tasks) availability, taking into account the usual overheads and lost days. For example, four hours per day per developer, or 100 hours per day if there are 25 developers.
2. Choose a candidate set of requests for the iteration. If there are no effort estimates, or they need review, new estimates are created.
3. Estimate elapsed days: total effort divided by daily availability. Round up to the nearest (five-day) week. For example, 800 hours divided by 100 hours per day leads to a two-week iteration.
4. If the length exceeds desired limits (e.g., three weeks for a small team, six weeks for a massive team), shrink the goals.

Either approach is best done in a collaborative planning effort with all team members, or subteam representatives for large projects.

Iteration Goals: Before the First Development Iteration

I've seen a project team start into iteration one, and quickly falter because the team is impeded waiting for the technical environment to get properly established and debugged: The source control management system, the continuous integration build machine, the application server, project Wiki, and so forth. Strive to have this in place beforehand; it always takes longer than expected.

Iteration Goals: The First Development Iteration

Or, what not to do. Iteration one proceeds slower than expected. So, use generous estimates, and choose small goals among the set of high-ranking requests. Then it is more likely that the team will successfully complete on time with less struggle, building confidence.

If the project also includes skill transfer goals (learning about object technologies, or a new agile method), do not dump a big bucket of new ideas and practices into iteration one. Add them incrementally over two or three iterations. For example, if the team has not used version control before, perhaps leave it and the related continuous integration practice for iteration two, and just emphasize test-first development and daily Scrum meetings in iteration one.

Iteration Goals: Use Cases and Scenarios

If use cases are being employed, it is desirable to fully complete a use case (or many use cases) within one iteration. Planning and scheduling are straightforward, and the growing system has functional cohesion.

This is not always practical. As mentioned in some other tips, use cases are composed of many scenarios (related to "extensions" in Cockburn's popular use case terminology). For example, the Process Sale use case has a scenario involving paying by credit, and another for paying by cash. Some use cases are too complex to implement all scenarios within one reasonably short iteration. Perhaps it would take three months to complete all scenarios of Process Sale—undesirably long for an iteration. Thus, they need to be decomposed by scenario for ranking, scheduling, and tracking. In this case, an iteration should complete the scenarios that it starts; a scenario should not be split across iterations (Figure 11.8).

Figure 11.8. scenarios across iterations

Iteration Goals: Primary and Secondary Requests

By frequently tracking remaining effort estimates, it may eventually appear that not all requests will be completed within the iteration. In timeboxed methods the response is to remove or simplify requests, rather than extend the iteration. What to remove? On projects with a strong customer-driven emphasis (such as XP projects), consider this approach: During the iteration planning meeting with customers, after the requests have been chosen, classify some as secondary that may be deferred. Not only does this support the agile spirit of customer-driven work, it also aids expectation management.

Iteration Goals: Don't Add Requests to an Iteration

An important rule in timeboxed methods is to not allow the addition of new requests once the iteration is underway. The product manager can't come over and ask to squeeze in the sexy Gromlit feature. Gromlits have to wait for the next iteration. Given that an average iteration is two or three weeks, this should not be a hardship. This rule creates an island of sanity and control in a sea of change and chaos.[3] Newly discovered tasks related to existing requests can be added by the team members, but not new requests.

[3] Ken Schwaber, a Scrum founder, named his Web site www.controlchaos.com to emphasize this point.

Tracking Iteration Progress

Some methods, such as XP and Scrum, have specific tracking practices discussed in their respective chapters. Some general iterative and agile-oriented principles and practices include:

- Frequent—often daily—task tracking for each worker is important with timeboxed methods; there isn't much slack in a short iteration.
- For smaller projects, a visible wall list of tasks is good. Hold the daily Scrum beside it. Cross out tasks when done. As the Xs grow, people see and feel progress.
- Some projects apply the XP practice of writing task cards that are physically held by the volunteer, or handed off to someone else. When complete, they are handed in to the tracker. Many people report that the physical cards have a positive psychological effect in making the work tangible.
- Asking people on a daily basis to self-record their remaining estimated task effort is rarely sustained or accurate. The XP practice of a daily tracker is a more sustainable alternative. Someone (other than the manager) visits each person each day (or every few days) and asks and records the remaining estimates on incomplete tasks.
- Many find a Scrum Sprint Backlog a simple, useful iteration tracking tool. If a daily tracker is asking about progress, they will be the primary person updating this backlog.

Tracking Iteration Progress—What to Track?

Since timeboxing creates a fixed soon-coming deadline, the critical progress data are the *remaining estimated effort on unfinished tasks*. This is the focus of a Scrum Sprint Backlog.

If a project also collects *actual* hours spent on tasks for each worker, consider using a daily tracker to collect this. One immediately relevant use for this data is to share it with the team members during subsequent iteration planning meetings so they can compare their original task estimates with final actuals. This feedback helps people become more accurate estimators—a valuable skill.

Test-driven development is an increasingly popular and valued practice in IID methods, not only in XP. To promote the early creation of more unit tests, some agile projects track (and update on a wall chart) the total number of unit test classes and test methods. The counting can and should be automated with a simple scanning program that runs as part of the continuous integration build process.

Tracking and Planning: XPlanner

If the simple "low-tech, high-touch" tools preferred in agile methods such as a whiteboard task list, paper task cards, or the Sprint Backlog spreadsheet file are found to be insufficient (more likely on larger projects), there are (free) open source software tools emerging to support agile planning and tracking. An example (not an endorsement) is XPlanner, available at www.xplanner.org. It provides support for stories, tasks, time tracking, and metrics.

Earned Value Tracking on IID Projects

Earned value (EV) tracking is a cost and schedule progress measurement method required on many USA government projects. Thus, it must be considered on government projects that want to apply an IID method. The details are beyond the scope of this introduction, but a non-obvious key concept is to measure progress in terms of the *estimates* (or *budgets*), not only in terms of *actuals*. For example, if creating help Web pages for a new system was

originally budgeted at 50 person-hours, then when the work is complete, regardless of the actual time spent (for example, 80 hours), the project is said to have "earned" 50 hours of value. A key term in this context is the Budgeted Cost for Work Scheduled (BCWS)—the estimates for future tasks, such as 50 person-hours for the help system.

There are a few practices experimenting with EV tracking on iterative and evolutionary projects, although their true worth is not yet known.

One practice is to re-calculate the BCWS (estimate) values each iteration, as more information arises. This has been called a **rubber baseline**.

Another practice is to apply a simple earned value **recognition rule** to iteration tasks. There are alternative rules, but a common one is that as soon as an iteration task is underway, it earns 50% of its value, in terms of progress tracking. Thus, as soon as the help page work starts, the iteration "earns" 25 hours of progress. Progress remains at 50% until the task is complete.

Ranking Risks

IID methods tend to be risk driven. A fast, useful method to prioritize risks is to estimate their probability *and* impact (in cost, time, or effort). The estimates may be quantitative (which are usually very speculative) or simply qualitative (for example, high-medium-low, based on discussion and group dot voting).

The worst risks requiring proactive actions are naturally those both probable and of high impact.

For example:

Risk	Probability	Impact
Insufficient number and quality of skilled object-oriented developers.	H	H
Demo not ready for the upcoming CEBIT convention in Hamburg.	M	H
…	…	…

Agile modeling promotes the practice of visible models, and that is especially useful for a risk list, with associated actions and status. Display this information using whiteboards or posters on the wall of the project room.

Managing Risks

Many iterative methods are risk driven, which includes not only tackling higher-risk technical elements in early iterations, but also more broadly, identifying and proactively working on all risks.

A key to successfully risk management is *proactive* actions *owned* by individuals, that are *tracked*. Consider keeping owner and status information on the project room's wall display of the risk list (see Table 11.1).

Table 11.1. risk list

Risk	…	Actions	Owner	Status
Insufficient number and quality of skilled object-oriented developers; thus, slow	…	PROACTIVE	Jill	approved in review

Table 11.1. risk list				
Risk	**…**	**Actions**	**Owner**	**Status**
development and poor, buggy design.		- Hire temporary consultants. - Classroom education and mentoring. - Design and program in pairs. REACTIVE - …	Jill Team	underway
…	…			

Environment

The project environment includes the physical space and software tools used by the developers. This section includes a few environment tips to support iterative and agile development. Agile methods emphasize the importance of communication or feedback; many of these tips promote it.

Continuous Integration

Part of the XP practices, **continuous integration** (CI) is also useful in other IID methods [FF99]. CI is a refinement and more frequent version of the daily build and smoke test practice popularized at Microsoft. Cycles average 15 to 30 minutes on many Java technology projects. There are at least two open source CI tools: Anthill (www.urbancode.com), and CruiseControl (cruisecontrol.sourceforge.net).

CI is somewhat more than just a frequent build-and-test tool. The details and how it works are explained in Figure 11.9.

Figure 11.9. how continuous integration works

Why bother? If a team practices CI from day one, then the growing application is almost always in a steady state of being integrated and tested. The "drift" from stability is on an extremely small scale. On average, every 15 minutes only a few new components are added. If the build is broken by an addition, it is likely to be a minor infraction. Therefore, the system grows micro-incrementally 15 minutes by 15 minutes, 24/7. There is less need for major end-of-iteration integration across team members or subteams.

What happens if the compile or tests fail? The CI tool sends email to the submitters and chief programmer, and people are expected to react immediately. For example, reverting to a previous version of the components, until the new components are debugged.

What if it takes a very long time to run the tests? Apply the "smoke test" principle promoted by Microsoft: Choose an important set of unit and acceptance tests that can run within 15 or 30 minutes. Run the longer set less frequently, such as four times daily.

Project Wiki Webs

Most people are familiar with blogs or Weblogs, that allow one to easily write and publish to a Web page. The older and more robust version of this idea is **Wiki** Web technology, created by Ward Cunningham (one of the XP founders). XP has the value, "do the simplest thing that could possibly work." In that vein, Cunningham has called Wikis "the simplest online Web database that could possibly work."

Like blogs, Wiki Webs (or Wikis) allow people to edit Web pages using only their browser, but they go farther: They allow one to easily create new pages, and hyperlinks between Wiki pages, using only a browser and special **WikiWords**. Of course, these capabilities are available with myriad tools, but Wikis make the tasks especially simple and fast. Thus, Wikis are a popular tool on agile projects to capture project information, and as a simple knowledge management tool.

Wikis are usually implemented as one or more Perl scripts that you install on an HTTP server. There are many open source Wiki kits. A popular simple version is Usemod (www.usemod.com); a popular elaborate version is Twiki (www.twiki.org)—it includes version control and various fancy features.

You can see how Wikis work and create your own Wiki pages at the original (fascinating) Wiki site: c2.com/cgi/wiki?WelcomeVistors. This site is also the place where the majority of online discussion took place during the mid- and late-1990s on the creation of XP and other agile methods, by their founders. These discussions are still available as Wiki pages.

CASE Tools and Reverse-Engineering

UML-oriented CASE tools support both **forward-engineering** (generation of code from diagrams), and **reverse-engineering** (generation of diagrams from code). If an agile-oriented project uses a CASE tool at all, it is most often simply for the reverse-engineering feature, to support visualization and the more visually-oriented on the team. For example, regeneration of noteworthy (package, class, or sequence) UML diagrams every few days to visualize the growing code base created with a popular IDE such as Eclipse (www.eclipse.org).

When I coach a team through an iteration, we print reverse-engineered UML diagrams zoomed very large on a plotter (if one is available), and stick them on walls. Some developers use the plots during discussions or short design sessions, sketching on them.

It is a misunderstanding to think that agile methods oppose UML diagrams or visual sketching. First, it is only XP that is especially textual-source-code-centric; the other methods are silent on the subject. Second, the XP leaders are not at all opposed to the practices of Agile Modeling, or using a tool to reverse-engineer code to diagrams if the effort is simple and aids communication.

Consider a Plotter

Visualization of information on the walls is promoted in agile methods; large diagrams and font size are important for ease of viewing. Printing large zoomed documents on a laser printer is laborious, as the partial pages have to be puzzle-pieced together. A wide plotter is faster and easier.

Caves and Common Room

The agile methods promote development in a common room rather than separate offices, to increase communication (Figure 11.10). It is a required practice in Scrum and XP. Of course, people also have need for privacy. Ken Auer promotes the "caves and common room" model in which there are also separate private office spaces (floating or dedicated) that developers can use during non-development activities [AM02].

Figure 11.10. agile project common room with walls exposed

Liberate the Walls

Walls are valuable, but not for desks, shelves, or tables. If development is done by teams in common project rooms (a required practice in XP and Scrum), move furniture away from the walls. Expose them to support the agile modeling practices of using many whiteboard spaces, and displaying as much project information (tasks, risk list, vision, and so forth) on the walls as possible. Figure 11.11.

Figure 11.11. floor layout

Cling Sheets or Whiteboard Paint

Agile modeling promotes the use of whiteboard space and freehand drawing for creative modeling work. In addition to true whiteboards (which may not be available), consider using "cling sheets" or "static cling sheets." These can be purchased at most office supply stores and come in packages like flip-chart paper. They are made of a very thin white flexible plastic material that has a static charge; the sheets cling to most surfaces. Wallpaper them to create a giant whiteboard space. Most important, you can use a whiteboard marker to write and erase on them, as with any whiteboard.

Another alternative is the use of "whiteboard" paint.

Digital Cameras

Agile is about fast, simple, light. Capture hand-sketched whiteboard models or lists with a digital camera. Print, plot, or consider placing the results on a project Wiki.

Figure 11.12. sample room elements

Requirements

This section introduces tips that support an agile, rapid, evolutionary approach to requirements, in addition to those already covered in other chapters, such as XP's Planning Game and story cards. Note that many of the tips emphasize face-to-face communication and "low-tech high-touch" — common themes in agile methods.

Agile Modeling

Agile Modeling is a set of values and practices consistent and complementary to all the IID methods, and applicable during requirements analysis. See p. 37 for a summary.

Defining and Keeping the Vision

Establishing and reiterating a common vision is frequent advice from agile leaders. It may seem absurd to highlight such an "obvious" idea, but in over 10 years of post-project reviews with hundreds of project members, Standish Group analysts [Johnson02] did not find even two people who stated the same purpose or vision for their project!

Sometimes the military is used as an example of an unhealthy, rigid management system, but modern battlefield leadership is different. Adaptation to unpredictable battle events (as in agile development) is paramount. Therefore, the leadership emphasizes that fighters know the mission goal or vision, rather than concentrating on a fixed plan.

Microsoft values this as well; it is key in the Microsoft Solutions Framework IID method:

Use high-level vision statements and outline specifications to get projects going.

How? Step one is to form a vision, discussed in some of the following tips, such as creating a vision box and product sheet.

Step two is to reinforce it. Jeff Sutherland, a Scrum founder, emphasizes that one of the key communication responsibilities of the Scrum Master during the daily Scrum meeting is to help the team recall a common project vision aligned with the business objectives.

The Product Vision Box

This is a creative, quick practice to craft a common vision, first created and promoted by Bill Shakelford and Jim Highsmith [Highsmith01], creator of the agile method Adaptive Software Development. The steps are:

1. During day one of the first requirements workshop, break into several small groups. Give each group a box (such as a cereal box).

2. The goal of each group is to create the cover (front and back) of a product box, as though the product were to be sold in a shrink-wrap box.

3. Define and place on the box a name, graphics, a *few* key front-cover selling points for the front, and details (features, operating requirements, and so on) for the back.

4. Each group presents the results.

5. Coalesce the results into a common Moore-style vision statement or common vision box.

A Moore-Style Vision Statement

The vision boxes can be input to a simple, focused vision statement. In Geoffrey Moore's well-known *Crossing the Chasm*, he recommends the following format for a vision statement, which has been widely adopted, including within the UP:

For (target customer)

Who (statement of the need or opportunity)

The (product name) is a (product category)

That (key benefit, compelling reason to buy)

Unlike (primary competitive alternative)

Our product (statement of primary differentiation)

Following the agile modeling practice to display models publicly, this statement goes on the wall—in large font.

Product Sheets

Marketing and requirements experts in product companies promote the early creation of a product data sheet (feature bullets, comparisons, and so forth). It helps clarify the vision and define high-level requirements; plus, the physical limit (one side of one page) forces prioritization and brevity. However, this practice is not only useful for product developers; it helps teams building *internal* systems who want a more agile approach to requirements

analysis. As with vision boxes, a complementary tip is to have several teams create them in parallel and coalesce the results.

Evolutionary Requirements Workshops

Although requirements workshops are an old idea (also under the banner of JAD—Joint Application Design—sessions), the iterative and evolutionary approach is to hold *several* of them, one per iteration during the early iterations, interspersed with programming. Keep them short and timeboxed, such as one or two days. In the first workshop, there is an emphasis on defining a vision and scope, and identifying functions and features at a high level (such as just the *names* of use cases and features). However, most non-functional requirements (e.g., load, internationalization) need early exploration in detail, as these have a significant architectural impact.

As an example: a project ultimately composed of 20 iterations might hold four requirements workshops across the first four iterations. The goal after four workshops is that 80% or more of the requirements are defined in detail, but in contrast to the waterfall, with the benefit of insight from early programming, evaluation, and feedback. Note also that by the end of the same four iterations, perhaps only 10% of the software is developed.

In the first workshop, for example, the team may discover and *name* 20 use cases (the same applies if features rather than use cases are used). They now have an estimate that the goal of 80% detailed requirements after four workshops will mean that 16 use cases should be written. Yet, in the first workshop they explore and write only two (i.e., 10%) of the highest risk and most architecturally significant in a detailed use case format. Combined with analysis of the product's nonfunctional requirements, that's usually enough to get started with architectural work and early programming.

Perhaps the second workshop team discovers three new use cases, eliminates two, and writes 8/21 in detail, refining existing ones, and so on. Figure 11.13 illustrates the process.

Figure 11.13. sample completion of artifacts in 5 of 20 iterations.

Include in each workshop developers who spend time working on the software. It is useful to demo and discuss the results of the most recent iteration. Feedback from building some of the software is crucial to refine and evolve the growing requirements.

This is not the only way to stage evolutionary requirements analysis; pipelining (see p. 251) is another.

The art and science of running a successful workshop is well-covered in *Requirements by Collaboration: Workshops for Defining Needs* by Ellen Gottesdiener.

Tracking Requirements Across Iterations

At the very least, track the lifecycle or status of each request (feature, use case, scenario, bug fix) as they transition through the iterations. XP does status tracking in the simplest way that could possibly work, for example, three piles of cards: "not done," "underway," and "done." Similarly, the Scrum Product Backlog spreadsheet includes a status column.

For use cases that will be completed within one iteration, the use case name is the requirements tracking tag—for example, "Handle Returns." On the other hand, when a complex use case has to be built by different scenarios across several iterations, the trackable requirements tag is a scenario name, not a use case name. If Cockburn's (popular) naming style is used, each scenario has a tag such as "3a" or "4c". Thus, the requirement tracking system can track labels such as "Process Sale-main success" or "Process Sale-3a".

Direct User Involvement in Requirements and Product

Research [KC94] shows that project failure is reduced when there is increased *direct* involvement and links between the *ultimate* clients or users of a new software product and developers. Plus, as an obvious corollary, fewer indirect links via intermediaries and customer proxies (such as business analysts). There are many techniques explicit within IID methods to support this, such as XP's Whole Team Together (Onsite Customer), UP's requirements workshops, and the Scrum demo to users at each iteration. Another popular practice is to send some developers as trainees in the client or target customer setting.

Use Cases are OK

It is a misunderstanding to equate iterative or agile methods with "no detailed written requirements." That value is unique to XP, and is enabled in XP by the presence of onsite customers in the project room who can verbally explain or create the requirements as needed. The other methods, although all are in favor of avoiding excessive documents, allow or support detailed written requirements, created iteratively and interspersed with early programming.

Unless it is an XP project, when written functional requirements are needed, consider use cases (for example, "Provision a Switch"). XP's promotion of feature-oriented "story card" requirements does not mean use cases are unskillful, or wrong for use in other methods. Note, for example, that the well-known agile methodologist Alistair Cockburn (author of *Agile Software Development*) is also an expert in and promoter of use cases, and author of *Writing Effective Use Cases*.

Use cases have the advantage of pulling a set of requirements together—organizing them in the context of scenarios of use. Plus, use case scenarios (e.g., "order, and pay with credit card") make excellent goals for an iteration, because by implementing a scenario you are forced to design and build across many architecturally significant components, from the UI to the database, for example. And, you must early on resolve and test the quality requirements (e.g., reliability, usability) associated with the scenarios. That said, use cases

don't fit for all products; sometimes a feature-oriented (e.g., "support EJB 2.0") or combination approach is more relevant.

Note that meaningful use case work is not *drawing* a UML use case diagram, but *writing* detailed use case text. In a workshop this is best done with a projector, so several people can easily view the text.

A popular approach to writing use cases is described in Cockburn's *Writing Effective Use Cases*.

Quantification Is OK

In the name of being more agile, do not succumb to being vague or sloppy about the details of quality requirements, such as load, response times, usability, and so forth. Most of these need quantification and means of measurement to be of much use. Evo places special emphasis on this value. "The UI needs to be usable," "The system should be maintainable," "The system needs to have good performance under high load," and so forth. These statements do not suffice, expressed verbally or in writing.

On outsourced projects—run with an agile method or otherwise— both the client and service provider need quantified and measurable quality requirements written down in order to avoid messy disputes. Although the Agile Manifesto encourages customer collaboration over contract negotiation, it is a fact of life that misunderstandings about the "…ilities" often lead to severe dissatisfaction—or law suits. I sometimes serve as an expert witness; it happens.

Agile Manifesto

GUIs with Glue

Another practical agile requirements tip is to first prototype UIs and UI navigation (in collaboration with clients) using paper, pen, sticky notes, and so forth, on a wall. This well-known technique goes by many names, but my favorite is Luke Hohmann's "GUIs with glue." Use large sheets of paper to represent Web pages or windows. Place small blue sticky notes on these sheets to represent information and pink sticky notes for UI controls (e.g., a button).

To quickly model navigation between Web pages or windows, consider sketching a UML statechart on a whiteboard, where the states represent the windows or pages, and the transition events represent the UI navigation gestures (for example, clicking a button). See Figure 11.14.

Figure 11.14. UML statechart sketch for UI navigation

A recommended text on usability engineering is *Don't Make Me Think!* by Steve Krug; many texts and Web sites cover "GUIs with glue" agile flavored UI prototyping.

Brainstorming

Brainstorming is useful and quick during both requirements and joint planning. Most think they know the technique, but surprising few follow its official—and important—rule. It is, simply:

When hearing and recording the ideas, no one should comment or laugh.

The facilitator should record, not react. Keep the ideas flowing.

Brainwriting

Brainwriting is a fast adjunct or alternative to brainstorming during a requirements or planning workshop. People are given many index cards or paper scraps. On the topic at hand (e.g., requirements? tasks?), they write a short note on each card as ideas emerge. The cards are collected, and the team does affinity clustering.

Affinity Clustering

Affinity clustering is a rapid technique to dynamically group things—such as brainwriting cards—into cohesive sets. It is usually done by placing the cards randomly on the floor. The team collaborates to organize the cards into the dynamic sets that emerge, such as "database task cards" or "payment feature cards." The results can be posted in groups on walls, or visualized in a mind map.

Mind Maps

Mind maps are another agile-oriented hand-drawing technique to quickly elicit, organize, and expand high-level requirements during a workshop. Deceptively simple, I'm always impressed by how they help the creative discovery process during rapid requirements analysis (Figure 11.15). When facilitating, I sometimes take the output of a brainwriting and affinity clustering session, and mind-map the result. There are several good Web sites; a classic text on the subject is *The Mind Map Book* by the founder, Tony Buzan.

Figure 11.15. sample mind map

Team Rotation Writing

Another quick, collaborative technique for requirements or planning is **rotation writing**. <N> people sit in a circle, each with a laptop computer. For five or ten minutes, they write on the topic at hand (requirements? tasks?). Then they stop and pass their computer to their neighbor, while they themselves receive another computer. The same approach can be achieved with a wireless network and sharable Wiki pages. Each person starts by reviewing the unfamiliar material, and then enhances it, inspired by the new ideas they see. Rotation continues until each person has worked on each computer document. After the meeting, one person receives all the documents, groups and merges the ideas, which are then later reviewed and refined by the whole team together in a second meeting.

Test

Test-Driven Development

Test-driven development (TDD, that includes the sub-practice of **test-first development**) is a key practice in XP, and is now promoted in several IID methods. An important part of TDD is that there are automated tests for (almost) everything, and most tests (especially unit tests) are written *before* the code to be tested.

This bears repetition: Unit tests are written *before* writing the code to be tested, imagining that the code exists.

Further, the tests simply pass or fail. There is no need for human inspection of specific test results. If you have 2,000 unit tests created in the TDD style, it all boils down to one question "Did they all pass?"

A brief programming example is presented to provide a concrete sense of how this key practice works.

Suppose Jill needs to create a *Money* class (in Java), to support multiple currencies, adding money, and so forth.

Since this is a TDD project, rather than start by writing the *Money* class, she starts by writing the *MoneyTest* class, adding one test method.

```
public class MoneyTest extends TestCase {

public void testSimpleAdd() {
     Money m1 = new Money( 12, "usd" ); // ONE
     Money m2 = new Money( 14, "usd" );
     Money expected = new Money( 26, "usd" );
     Money result = m1.add( m2 ); // TWO
     assertEquals( expected, result ); // THREE
   }
} // end of class
```

Jill *imagines*, at point ONE, that there is a Java constructor in the *Money* class that is written and works, and at point TWO, that a *Money.add* method is written and works. Notice also at point THREE that there is an assertion that causes the test to simply fail or pass; no human inspection of the specific results is required.

Then, Jill programs just enough of the *Money* class to make it pass the test *testSimpleAdd*, writing the constructor to satisfy point ONE, and the *add* method to satisfy point TWO:

```
public class Money {

public Money( float value, String currency ) {
// body of method ...
}

public Money add( Money other ) {
// body of method ...
}
} // end of class
```

The implementation details are not important, but the process is. Once the code is written, Jill runs the test. If it fails, she debugs till it passes.

Next, Jill is ready for the *subtract* method. She starts by adding another test to *MoneyTest*:

```
public class MoneyTest extends TestCase {

public void testSimpleAdd() {
// ...
}

public void testSimpleSubtract() { // NEW METHOD
// body of the new test ...
}

} // end of class
```

Then, she updates *Money* to make it pass this second test.

Test-first development has some practical consequences:

- Tests actually get written, which yields a host of benefits. This is obvious, but in many projects that do not practice TDD, tests are just not written.

- It is at least a semi-enjoyable way to do testing—that makes it more sustainable. Agile methods tend to emphasize practices that developers (eventually) like—or at least do not dislike—doing. Traditional ("test last") testing is avoided because it is boring or tedious. By writing the tests first, the developer is engaged in thinking through the proper public behavior of the class not yet written. That's interesting and creative. And, she writes the test and then builds something to make it pass. That gives a small feeling of accomplishment.
- To expand a point made in the last bullet, by writing the tests first, the developer is engaged in thinking through the proper public behavior of the class not yet written. That thought process—treating the new class as a separate black box with public operations—clarifies in the developer's mind the behavior and design of the class before it is programmed. That thinking tends to improve the design and, as a valuable side benefit, a test written.

Finally, note that many IID projects also practice continuous integration. All these tests, growing constantly, become part of the continuous integration test process, re-executed on each build cycle.

continuous integration

Fit or Fitnesse for Acceptance Testing

Ward Cunningham, a key figure behind XP, has created a simple, open-source framework and tool to support acceptance testing: Fit. See fit.c2.com. It has become relatively popular in the agile development community. A related tool, Fitnesse, has been developed by Bob Martin and his team at ObjectMentor, early promoters of XP and agile development. See www.fitnesse.org.

Chapter 12. Frequently Asked Questions

I have often regretted my answers, never my silence.

—adapted from Xenocrates

These assume an audience relatively new to iterative methods, keeping with the book's theme as an introduction.

Question List

Q1: **Is there any proof that iterative development is worthwhile, or better in some qualities than the waterfall?**

Q2: **For example, will an agile or iterative method make my team more productive?**

Q3: **How do you plan an iterative project?**

Q4: **My customer expects a week-by-week schedule and detailed PERT chart. What should I do?**

Q5: **How to handle fixed-price contracts when applying an IID method?**

Q6: **Can IID be applied on projects or contracts (usually fixed-price) in which we are forced to do major up-front requirements analysis?**

Q7: **What are typical risks and mistakes when adopting an iterative process?**

Q8: **How to adopt an iterative, agile process within an existing waterfall process culture?**

Q9: **How to control costs if adaptive planning?**

Q10: **How do we measure quality in an iterative process?**

Q11: **How to coordinate subteams or subcontractors on a large IID project?**

Q12: **How to estimate overall effort or duration for an IID project?**

Q13: **How to estimate the duration of an IID project without having a plan of what will happen week by week?**

Q14: **If we have use cases, how to schedule them with respect to iterations?**

Q15: **How do we track use case requirements across iterations?**

Q16: **How to persuade our customers (or management) to adopt IID?**

Q17: **We want to apply XP, but don't have an onsite customer. What do we do?**

Q18: **We think we are applying XP, but use fairly detailed written specifications for the iteration rather than an onsite customer. Is that OK?**

Q19: **What's going to happen with our existing test and QA department if we adopt an IID method?**

Q20: Can a project fail with an IID method?

Q21: What new skills are needed for managers and developers?

Q22: How to deal with change management in an IID method?

Q23: Is IID useful for commercial products?

Q24: We have to tell the customer what they will get and what it costs—before starting to build it. Therefore we can't work iteratively, true?

Q25: We can't make a solid architecture if we do not know all the requirements up front, true?

Q26: Rework (or refactoring) each iteration sounds expensive. Isn't it cheaper to design it correctly up front?

Q27: What use are iterations for short projects of, say, three months duration?

Q28: How can we get our management to realize they don't need a final, detailed plan on "day one"?

Q29: Our test environment is very complex and run by another organization. How can we iterate and test?

Q30: What do we do when time, budget, and scope are all frozen but we still want to apply an iterative or agile method?

Q31: Doesn't iterative development mean that we don't know when we're finished?

Q32: Should I plan the work for all the future iterations to ensure the scope and resources (e.g., people) fit the desired end date?

Q33: How do I get feedback when there is little or no user interface?

Q34: Should iteration activities overlap? For example, requirements for the next while testing for the previous?

Q35: How long should iterations be?

Q36: How to handle the design of a database with an iterative process?

Q37: Should the customer always be in charge of what gets built each iteration?

Q38: How to plan an iteration?

Q39: Do I give the results of every iteration to my customer?

Q40: How to do documentation for maintenance, when we want to be agile?

Q41: How can I create a work breakdown structure (WBS) without a weekly schedule, or an iteration-by-iteration schedule?

Questions and Answers

Q1: **Is there any proof that iterative development is worthwhile, or better in some qualities than the waterfall?**

A1: Yes. For the whole story, see the Evidence chapter (p. 63).

Q2: **For example, will an agile or iterative method make my team more productive?**

A2: Yes, there is some evidence, both for productivity and defect reduction (p. 76). However, beware silver-bullet "faster, cheaper" claims made speculatively by various agile method promoters. One must also be mindful of the **Hawthorne Effect** when new methods are introduced: That individual productivity and other behaviors *temporarily* improve when people adopt new methods and know they (or their project results) are being studied.

Sustainable productivity improvements only arise over a long period of effort and change, with long-term initiatives and management support. Gerald Weinberg once cautioned against claiming that anything would make more than a 10% improvement—wise advice.

Perhaps more important than asking about more productivity is asking about *less failure*. Software project failure rates are high, averaging 23% in the USA in 2000. Rather than first searching for the next 2% productivity improvement, it may be a rational higher priority to first focus on preventing some of the lost (estimated for the USA) $50 to $70 billion that is spent each year on cancelled software projects.

Interestingly, investigation by the Standish Group into project failure indicated that this level of failure has not been visible at the CEO and VP levels; not all executives know how bad it is in their software groups. To quote from their 1998 report:

For years [software] project failure was simply not discussed. And it certainly was not discussed with the CEO.

In this context, perhaps the most relevant value of IID is that it is correlated with lower failure rates. And, through the process of early feedback, leads to products more closely aligned with what customers really want.

Q3: **How do you plan an iterative project?**

A3: See tips starting on p. 248.

Q4: **My customer expects a week-by-week schedule and detailed PERT chart. What should I do?**

A4: See "Rolling Wave Adaptive versus Predictive Planning" on p. 253.

One approach is to present a short seminar on IID and client-driven adaptive planning, and propose starting the next project based on this approach. If they are not satisfied after some time period, you will agree to revert to their desire for a predictive plan.

If that isn't possible or doesn't work, another approach is to create the predictive plan they desire, but then run the project with client-driven adaptive planning. At the end of the first iteration, show the customers a demo and update them on progress and new insights. They too will have new insights and priorities based on this feedback. Then invite them to tell you their current priorities and refinement

of ideas. Covertly, the project may then slip into adaptive planning as the customer directly sees the benefit of their ability to guide it iteration by iteration.

Q5: **How to handle fixed-price contracts when applying an IID method?**

A5: Also see the follow-on related question and answer: Can IID be applied with contracts (usually fixed-price) in which we are forced to do major up-front requirements analysis? (p. 303)

There are at least two ways to answer this: the ideal, and the usual.

The ideal—which *has* been sold with success by various consulting companies—is to organize the project into two contract phases. Contract phase one corresponds to inception and elaboration (or at least a good part of elaboration) in the UP. See "Fixed-Price Contracts" on p. 18.

Note how this approach reduces the risk for the customer. In phase one they paid for tangible results that moved the project forward, but they didn't commit everything to the project or to one service provider. Also, it is desirable to hire very talented people for this phase one, to create a solid foundation. But it may thus be possible to use less expensive average resources for phase two. Finally, for phase two, the customer will more reliably know the true costs and duration, and will be working with a provider who has a greater chance of remaining both solvent and sane, as the provider accepted the challenge of the project with sufficient initial information to make an informed (rather than desperate) choice.

In summary, this approach balances the risks for the customer and service provider.

As an aside, some consulting companies have run a refinement of this phase one in which one developer from the customer side joined the consulting team. They not only provided insight for the project, but insight to the culture and leaders of the organization. During an end-of-phase demonstration to the customer, the customer-developer herself leads the demonstration created with the consulting team, and thus creates an in-selling effect to help the consulting company win the phase two contract.

The more commonplace answer to the contract question is that service providers are forced to create fixed-price bids without the luxury of the above phase one, and are taking a larger risk. In this case, they bid however they prefer. (Note, as an aside, that the estimation technique of WideBand Delphi can improve their estimates—see p. 260) Yet, running the project iteratively still gives them an advantage: Since iterative development is about tackling the high risks and hard elements in early iterations, they will receive early feedback about how much trouble they are really in, or not! They will discover more quickly if their estimate of the cost or difficulty was low, and be able to take early mitigating actions, such as hiring experienced specialists, looking for preexisting components, early expectation management, and so forth.

Furthermore, by running the project iteratively, they will be showing early visible results of value to the customer. There is an increased chance, through thus winning the confidence of the customer, that they will be able to renegotiate some of the difficult fixed-price contract terms at the suitable strategic moment.

Q6: **Can IID be applied on projects or contracts (usually fixed-price) in which we are forced to do major up-front requirements analysis?**

A6: Yes. even with "the complete requirements" developing via many short iterations has advantages.

To reiterate points from a prior answer, the team receives early feedback about how much trouble they may be in, by building, integrating, and testing early and often (there's nothing like programming to discover what you don't know). This approach drives down the risks, shakes out the requirement bugs, and provides opportunities for reacting sooner rather than later to major problems.

There will be early tangible results for the customer (user, marketing manager, …), leading to confidence building and quality feedback. If the product needs to be rolled out earlier than planned, there's something available.

And research suggests that developing in short, timeboxed steps is associated with higher productivity and lower defect rates.

Q7: **What are typical risks and mistakes when adopting an iterative process?**

A7: Near the top of the list of mistakes or risks is that the customer or executive management does not understand and accept the change in values and practices, or appreciate how deep and far-reaching the changes need to be. I see this manifest in situations like "Congratulations! We've adopted <iterative process X>. When will the requirements be finished, so we can decide how to design the system?" Or, "It's budget season. Please take a few weeks to identify all the projects for next year, and how much they will cost and how long they will take." Inconsistent culture and expectations derived from waterfall or mass-manufacturing values clash with an iterative and agile approach. As another example, the customer does not actively participate, iteration by iteration. And so on.

The solution includes having an executive and customer champion who understands and can communicate this to their peers, education seminars and learning aides (e.g., this book) for these groups, and "post-partum" project sessions in which the stakeholders share their experiences with other customers and management.

Another common mistake is changing or adding to the goals of an iteration, once underway. In a sea of constant change and chaos, some stability and control is necessary. That comes from leaving the team alone, once they've committed to an iteration. Save the change requests for a future step.

Another problem is using so-called iterative or agile consultants or consulting organizations who don't really comprehend evolutionary, adaptive development. They superimpose waterfall values on top of an iterative process, or try to recast their old waterfall process as an iterative one. Then we get corruptions such as misinterpreting the UP inception phase as requirements, elaboration as design, and so forth. Or, we get promotion of excessive specifications and other documents, instead of early programming. Or, we get predictive planning in which a plan is created listing how many iterations there will be, their durations, and what will occur in each. Or, we get misunderstandings such as "let's iterate over the requirements until they are stable, then we can nail down the design in a series of iterations."

Another risk is attempting to transition to an iterative and agile process without coaching from someone who's been there and done that. I sometimes see well-intentioned local managers championing the adoption of an iterative method, who think they or their staff can lead the adoption without help. And sometimes they

can, but a timeboxed, iterative, agile approach is rather different for many teams. Colleagues and I have seen a number of homemade adoption attempts where the adopters didn't appreciate the synergy between the different practices, and they were modifying the newly adopted methods unwisely. For example, claiming to adopt XP by simply eliminating written requirements and writing unit tests occasionally.

Using a coach who isn't steeped in the old local ways and who has the confidence of knowing that the iterative process works is money well spent. And, because we are talking about value changes, it is often more effective that the change agent be from the outside. It seems to be the way of the world that we're never a prophet in our own land, especially regarding software methods.

Another common mistake is overselling or mis-selling the advantages of an iterative, agile method to customers or management. The popular press and a number of agile method books still exhibit the silver-bullet syndrome. The master consultant Gerry Weinberg advised to never promise more than a 10% improvement, not only because greater yields are seldom sustained (and let us not forget the Hawthorne Effect), but more can suggest the *current* management and management practices are really inept, which is seldom true.

On the subject of mis-selling, note that iterative methods are *not* fundamentally for improving productivity or delivery speed or reducing defects, although there is research showing correlations. Rather, they are less ambitiously for reducing the risk of failure and increasing the probability of creating something of value that the stakeholders wanted. Given that recent data shows that 23% of projects fail, this is no small feat. Issues like improved productivity are secondary when one out of four projects simply "goes pear shaped" (as they say so evocatively in the UK), after consuming—on average—$1.2 million USD. Note as a related point that this same failure research indicated that the CIO or CEO has often been "shielded" from these failure rates, and is unaware of the true extent of failure in her development organization.

Big-bang process adoption is another common mistake: Educating many managers and developers in the new method over a short period, and/or applying it on many projects during early adoption. Just like a software project itself, adopt the process iteratively and incrementally, in small steps. Start with pilot projects, and learn from the experience.

Another risk is to try to recast the new process in terms of your current culture's vocabulary and ideas. For example, attempting to adopt the UP but rename the phases and workproducts to old, local names. Or, to make efforts to explain how the new process fits into the ideas and phases of the old one, in a misguided desire to help the new process be successfully adopted. Just surrender to the new; make a clean break from the past.

Some organizations have a small group responsible for process, methods, best practices, and so on. A risky adoption approach is for this group to *speculatively* decide how to apply the new IID process—*armchair methodology*. They try to "enhance" or "refine" it speculatively. Coupled with this problem is the related complication of top-down process advice, which doesn't usually work. For example, if an organization is adopting the UP, it has the concept of tailoring the process to fit the project and organization. A risk is to let this group speculatively create a UP tailoring, an XP tailoring, or whatever. The result will often have little to do with what's really useful in the actual project. Rather, determine how to adopt the new process through experimentation and by the advice of the coach

and practitioners on several pilot projects; more bottom-up than top-down.

Some organizations have a separation of software designers and programmers. Maintaining this separation is another mistake when adopting iterative development, although there is still need and value in expert designers such as a chief architect. The programmer must be an active designer, as the design is not fully pre-cast in these methods, but evolves in response to growing insight, test results, refactoring, and so forth.

Q8: **How to adopt an iterative, agile process within an existing waterfall process culture?**

A8: Suggestions include:

- Have an executive and customer champion who understands and can communicate the ideas to their peers.
- Define a goal or reason to adopt the method, and a quantitative measure of its success. For example, the number of failed projects per year, or the results on satisfaction surveys for developers, managers or customers. Measure and communicate the results. Don't expect quick or dramatic improvement; process change takes time and skill over a series of projects, and a new method is not a silver bullet that will revolutionize things. The famous (and efficient) Toyota manufacturing system took over 10 years to be fully adopted.
- Present education seminars and learning aides (e.g., this book) to executive and customer groups.
- Adopt the method with pilot projects and an incremental approach. Start with one project and a method coach. Drive the adoption from the learning that emerges from these early projects.
- Don't oversell. Don't claim it will improve productivity and so forth, but propose a pilot as an experiment whose results will guide further steps—in other words, an empirical approach.
- A failure on the early projects will—not surprisingly—kill the adoption drive. So, mitigate that risk by using a good method coach. Choose a project big enough to be meaningful but not so big it is dangerous; for example, five or ten people on a six-month project is a good size. Don't introduce too much novelty on the projects, such as many new technologies or unproven third-party components. You don't want the pilots to fail for reasons unrelated to the new method.
- Let the participants in the early pilot projects become the new method leaders (or "process engineers") in subsequent projects.
- Hold "post-partum" project sessions after these early projects in which the stakeholders share their experiences with other customers and management. This in-selling is more powerful than executive or consultant recommendations.
- Results speak louder than theory. Assuming the pilot projects do achieve earlier valuable and visible results with lower risk than the prior waterfall process, record this achievement, and communicate it.
- If the waterfall organization is resistant to the idea of short iterations, propose instead that their next 12-month project be run as two 6-month projects or three 4-month projects, in order to "lower the risks and show early results." Capture a record of positive experiences with this change, communicate it, and on the next project, suggest a shorter step: "We improved with two 6-month steps. We think we can do even better with three 4-month steps on the next project."

On the question of what type of pilot project to choose, XP and Scrum have a different answer than the UP. The latter suggests a not-too-risky project, but the

former methods (especially Scrum) recommend first adopting it on the most difficult project the organization faces. These method leaders feel confident that their methods, applied correctly, will yield success, and that the crisis of a difficult project provides the right fertile ground to truly abandon the old waterfall habits and seize the new ideas wholeheartedly.

Q9: **How to control costs if adaptive planning?**

A9: Before answering the question directly, there's often an implication in the mind of the questioner: That with predictive planning, costs (and schedule) *is* successfully controlled. But research shows this is not true; indeed, predictive planning has a poor track record for software projects. The root problem is the flawed assumption that software development is predictable manufacturing rather than new product development.

Failure research

Evo and evolutionary delivery provide one model for the answer. At some point an overall budget or estimate is generated for the project, although it is acknowledged as unreliable. Thereafter, take a small iteration step that represents between 2% and 5% of the budget (or desired duration). Plus, choose a step with a high value-to-cost ratio. Have a quantified goal (or goals) for the iteration, and measure the impact of the step. In the worst case, we have "wasted" a small percentage of the budget on an unsuccessful step. In the best case, we have made a good return on the small investment.

The adaptive plan emphasizes—using the most recent information—maximizing value for a small cost commitment.

Q10: **How do we measure quality in an iterative process?**

A10: The short answer is, the same as in any process, but earlier and more frequently. Yet, there is a distinctive component I want to emphasize. A useful best practice is to *continuously verify quality*, and this implies not only quality of the product, but of the *process*.

A number of agile methods promote some kind of *iteration assessment*, which I usually prefer to call the *beer party* (being Canadian). That is, at the end of each iteration (or the start of the next), get together for a half hour as a group, and ask some questions: what worked well, what didn't, and what are a couple of concrete actions we could take in the next iteration to improve? Maybe Jill spent too many hours explaining the defect tracking system to new joining consultants, and she should take a few hours to write up a Web page summarizing the introduction for the next incoming batch of people. Maybe the evolving set of use cases are useless, and their creation should be stopped, or improved.

The Scrum meeting also provides a way to measure the quality of an iterative process. We see, day by day, how things are going, what's working, and what isn't.

Q11: **How to coordinate subteams or subcontractors on a large IID project?**

A11: One part of the answer involves establishing early personal liaisons between the subteams, and a leader in each with an understanding of the project vision and architecture. See "Multiteam or Multisite Early Development" on p. 248. That section provides the details.

The value of forming these personal relationships with other liaison team members is most significant during later coordination and communication. And, having forged a common, deep understanding of the vision and architecture lowers the risk that the subteams don't understand or create what is needed.

Another part of the answer is to establish cross-team milestones for integration and testing (system, load, etc.) of all components. These are the **macro-iterations** of the project. For example, the project may have a macro-iteration of six weeks; at its completion, all components across all subteams are integrated. Within this macro-iteration the subteams may choose to decompose the time and their own work into shorter **micro-iterations**, such as three two-week iterations.

These macro-iteration milestones provide the heartbeat and mechanism to force regular coordination between the subteams and subcontractors.

Note that very short macro-iterations for large projects with many subteams *may* become awkward or unproductive. The overhead of pulling everything together and testing it takes nontrivial time and resources; a two-week macro-iteration may be too short for completion of sufficient new work, given the overhead of integration and test.

Another proven variation on this macro-iteration approach is to relax the requirement that *all* subteams must integrate, and instead only require that at least *two* of the subteams must integrate. This may allow a shorter macro-iteration, such as two weeks, as the effort of integration and test is lowered. See "Difficult Multiteam or Multisite Iteration Planning" on p. 249.

All of these variations can (and usually should) be combined with the practice of continuous integration—see p. 275.

Q12: **How to estimate overall effort or duration for an IID project?**

A12: One way to answer is, the same as before. Don't assume that scheduling the tasks iteratively will meaningfully change productivity or duration. You may still use parametric estimation models (such as COCOMO II), micro-estimation methods based on work breakdown structures or use cases, and so forth.

That said, some of the IID methods, such as XP, include specific advice on how to estimate. These are introduced in their respective chapters, although study of dedicated method books (listed in the recommended readings sections) is needed for full details.

However, I do want to recommend an excellent *iterative* and team-based estimation technique that is complementary to other estimation techniques, Wideband Delphi—see p. 260.

Wideband Delphi

Q13: **How to estimate the duration of an IID project without having a plan of what will happen week by week?**

A13: A total effort (or duration) estimate precedes and is not dependent on detailed task scheduling. Rather, it is primarily a function of the requirements, team size, novelty, and so forth. Of course, scheduling issues can affect an estimated completion date, such as if the project spans the summer months of vacation.

I get this question frequently, and the questioner usually really means to ask, "How to estimate the dates of *intermediate* milestones with an IID method?" Major milestone dates can be estimated based on common effort estimation methods, once the goals for the milestones are decided. As always, the reliability of the estimate is commensurate with the quality of the information and the project's current point on the cone of uncertainty.

Q14: **If we have use cases, how to schedule them with respect to iterations?**

A14: Although it is desirable to fully complete a use case within an iteration—it's a straightforward approach—it isn't always best, because some use cases are so complex (with many scenarios) that it would take an excessively long iteration (such as three months) to complete. Short iterations are almost always preferable for a number of reasons. For details: See "Iteration Goals: Use Cases and Scenarios" on p. 269.

Q15: **How do we track use case requirements across iterations?**

A15: To expand the question, how do we track that some scenarios of a use case have been done, and others haven't?

The answer is dependent on the requirements tracking tool, and the way you write use cases. Let's assume you are using the popular www.usecases.org (Cockburn) format for use cases. In this case, for each use case there is a "main success" scenario and various "extensions" (or "alternatives") with labels such as 3a, 3b, etc.

If you're using a tool like Rational's RequisitePro (which adds macros to Microsoft Word), you can use Word to textually highlight a scenario, and then mark the selected text (e.g., the scenario 3a) as its own requirement in the RequisitePro database, with lifecycle state information such as "approved," "underway," "completed."

If you are using a feature or issue-oriented tracking tool, such as Issuezilla or Bugzilla (which is often used for new requirements, not simply defect tracking), you can record the scenario names as labeled issues, with associated lifecycle state. For example, scenario 3a of the Process Sale use case can have the label "process sale-3a" and the state of "complete" in Bugzilla.

Q16: **How to persuade our customers (or management) to adopt IID?**

A16: Don't propose a definite adoption within the organization. Rather, suggest an experiment, motivated from the data and trends: that IID is associated with lower failure rates and earlier results, that it is now used by many organizations, its use is increasing, and so forth. Some of the data in the Evidence chapter (p. 63) may be helpful.

Then, organize a half-day seminar for the executive team, customers, and other relevant stakeholders. Use a seminar speaker who can present the key ideas and make a persuasive case for the experiment, but avoid overselling the benefits. It is useful to emphasize that IID methods support early visible results and ongoing steering by the customers; of course, customers are interested in this point. It must also be stressed that customers will need to take an active and ongoing role in clarifying the requirements, evaluating results, and providing feedback.

Next, get commitment for the experimental pilot project and find an executive champion. Run the project, capture data on the experience and results, and

communicate these. Make a decision to continue, or not.

Q17: **We want to apply XP, but don't have an onsite customer. What do we do?**

A17: See p. 152.

Q18: **We think we are applying XP, but use fairly detailed written specifications for the iteration rather than an onsite customer. Is that OK?**

A18: See p. 156.

Q19: **What's going to happen with our existing test and QA department if we adopt an IID method?**

A19: In a classic waterfall environment, the QA team expects to receive a final system for testing near the end of the project, but may not otherwise be significantly involved in the development. Although a final QA step never hurts, it is not sufficient or efficient—because research shows that it is cheaper to remove defects early rather than late.

With IID methods, there are at least two approaches to working with the QA group. The first is to allocate a QA person (or persons) to the iterative project from the earliest iterations—either full-time or part-time. They are involved, iteration by iteration, in the early creation and execution of tests and other evaluations. They become members of the development team. If the project is running in a common project room, then ideally that's where they work, although there are times this is not possible due to the complexity of the test environment.

Microsoft takes this to the extreme by dedicating more-or-less one tester for each developer, and they collaborate throughout the project. As an aside, Microsoft developers do not usually practice test-first development (see p. 292); it would be very interesting to know if the same number of independent testers would be needed if they did test-first development.

A second approach is to deliver the internal release from each iteration to the QA group for evaluation. While the development team is moving forward with iteration N, the QA team is evaluating the results of iteration N-1. See "Overlapping or "Pipelining" Activities Across Iterations" on p. 251. Their feedback can be handled in the current iteration, or if it is too laborious and there are no slack resources, allocated to the next.

Q20: **Can a project fail with an IID method?**

A20: Certainly, although I like to call IID approaches "fail early" methods, and the waterfall a "fail late" method. A waterfall project can be like the story of the guy who fell off the cliff:

As he was hurtling down, someone yelled, "How are you doing?" The guy replied, "So far, so good!"

In the waterfall, the risks pile up near the end; the project can have the mirage of running smoothly for many months while the less risky and easier work is done. Then, pow!

On an iterative project we discover how much trouble we are in sooner rather than later. We have a better chance to cancel the project before too much is invested, or experiment with solutions.

Q21: **What new skills are needed for managers and developers?**

A21: For managers, perhaps the biggest shift—at least with methods such as XP and Scrum—is to step back and avoid assigning tasks or directing work, not being the taskmaster. Recall that in these methods self-directed teams and volunteering for work is important. The manager's role is to reinforce the project vision and company goals, manage risks, communicate the iteration goals, remove blocks, provide resources, and track progress.

They are also responsible for the new skill of iterative and adaptive planning, which is easier but more frequent than detailed predictive planning.

For developers, they will participate in more project management activities, such as task identification and estimation, each iteration. Especially with XP and Scrum, their biggest shift is perhaps the attitude of "owning" the project and its problems. Recall that in the daily Scrum meeting, it is the team's collective responsibility to spot and fix problems with the project or team members, not the manager's responsibility.

On the technical side, developers require skills in how to set up and do continuous integration, and more frequent and thorough testing than they may have previously been used to.

Q22: **How to deal with change management in an IID method?**

A22: This is specific to the method, although most have in common the following constraint: Once an iteration is underway, no changes are introduced to the iteration. This gives the team a short stable period—some control over the chaos.

Most also have in common the practice of *not* treating change requests informally via talk or email; therein lies a path to project ruin! Rather, changes are captured in a change request (whose form ranges from a simple story card in XP, to an entry in the Bugzilla database), and considered decisions for the requests are made by the key stakeholders during the iteration planning meetings.

Q23: **Is IID useful for commercial products?**

A23: Certainly, and in fact IID methods found early and widespread adoption in the product sector. In many software product companies of Silicon Valley, for example, you will find IID has been commonplace for years.

Q24: **We have to tell the customer what they will get and what it costs—before starting to build it. Therefore we can't work iteratively, true?**

A24: False. There are several ways to answer this. For one, if you are in such a market (still common, for example, with fixed-bid contracts with governments), then do what you must with up-front analysis and estimation. And, the customer might have required a detailed predictive plan; you focus in this plan on identifying what they want in the first and second iteration, and accept that the remaining iterations will be less rigid. Next, start to develop in short iterations, and bring the customer into the evaluation and feedback process. By showing the customer the results of the early iterations quickly, you win confidence. At this point you may say to the customer "Even though we planned <X> for iteration 3, you now have a chance to re-choose according to your latest insight and priorities." And at this point the customer is more likely to view this flexibility and control not as a defect in your skill as a predictive planner, but as a more valuable way. The same practice and psychology applies to the evolution of requirements.

In short, we make a waterfall attempt as desired by the customer, and then run an evolutionary IID project as trust is established, to really benefit the client.

Another perspective is that even if we don't have the degree of requirements evolution we wish, by at least organizing the "frozen" requirements work iteratively, we gain several advantages. As the Evidence chapter shows (p. 63), productivity, defect, and success rates may be improved. And we may still have flexibility over the ordering of the development iterations, to meet our desire to drive down risks early.

Customers don't usually care about fine-grained weekly scheduling. They may want to define a milestone that <X> is completed in two months and <Y> two months later. But, they don't have to see that you organize a two-month phase into four iterations of your choosing.

Q25: **We can't make a solid architecture if we do not know all the requirements up front, true?**

A25: False. What we do need to understand early is the architecturally significant requirements, which is a subset of the total. Plus, architecturally influential factors are mostly nonfunctional quality requirements, such as reliability, security, and so forth, rather than the myriad detailed functional requirements. It is less difficult to learn the former than the latter during early analysis. In addition, if use cases make sense for the project, we can focus in the early phase on understanding the subset of architecturally significant use-case scenarios (which may be 10% of the total set), rather than all use cases.

For example, in the UP, the idea is to analyze something like 10% of the requirements during the inception phase—those that are most architecturally significant. Then, quickly start programming in the elaboration phase, while the majority of the remaining functional requirements are uncovered and evolved—perhaps in a series of requirements workshops—in parallel to programming the core architecture.

Q26: **Rework (or refactoring) each iteration sounds expensive. Isn't it cheaper to design it correctly up front?**

A26: In practice, IID projects infrequently require massive rework; it is more a theoretical than practical concern. This is due to a combination of taking small steps and emphasizing early testing and feedback, so that a solid path is discovered and maintained sooner rather than later.

In addition, modern powerful refactoring tools (in several Java IDEs, for example) make large-scale changes easier and faster.

In any event, complete and near-perfect up-front speculative design or architecture is seldom observed, even when it has been diligently attempted. Decades of failed attempts in waterfall projects demonstrate the difficulty—and expense—of this approach. The reasons are varied: the constant use of new (and unproven) technologies, high complexity, the many degrees of freedom software solutions offer, the unreliability of the requirements on which speculative design decisions are based, and more.

Also, bear in mind that it is only XP which promotes almost no up-front architectural thinking, not the other IID methods. Scrum, UP, Evo (and others) all support some degree of up-front architectural analysis and design, with a

balanced interplay of early programming and testing to prove or disprove the ideas.

Q27: **What use are iterations for short projects of, say, three months duration?**

A27: Organizing the development and priorities in two- or three-week timeboxes still helps with achieving early visible progress, and keeping the complexity manageable. Often in these projects, most requirements are semi-reliably known near the start; very good. If your organization has adopted the UP and its concept of the four phases, the distinction of the first three (inception, elaboration, and construction) is not particularly useful on such short projects; rather, simply a series of "development" iterations prioritized by value and risk, followed by a transition phase, is sufficient.

Q28: **How can we get our management to realize they don't need a final, detailed plan on "day one"?**

A28: Through demonstration, analogy, facts and logic.

Demonstration— Create and run an IID pilot project applying the principles of adaptive planning. Have external management and clients drive the choice of work each iteration, and at each post-iteration demo ask "Is the project proceeding as you want?"

Analogy— When we appreciate that building software is new product development or discovery, we can draw analogies from other industry planning practices. It is *normal* to avoid detailed predictive planning at the start of a project in other discovery-dominant domains. For example, examine how new potential oil fields are planned. Or a new type of car, a new bridge, or a new consumer gadget. In each case, there's a significant exploratory phase before reliable plans are expected.

Facts and logic— Perhaps the most relevant fact is an average of 20–40% requirements change on medium to large software projects See "Change Research" on p. 72. Not surprisingly, then, the historical track record of early detailed predictive planning is poor. The problem isn't bad planners, the problem is high degrees of novelty, uncertainty and change. An early and highly speculative fixed plan in that context is not logical; the wrong (mass-manufacturing) model is being applied to a discovery-dominant domain: software development.

Q29: **Our test environment is very complex and run by another organization. How can we iterate and test?**

A29: Before offering some suggestions, note there *are* similar organizations that do this, iteratively. Microsoft is probably the largest example of a company that applies IID in a complex test environment with separate testing groups.

One part of the solution is continuous integration or the more mild *daily build and smoke test* practice. The separate test team adds unit or acceptance tests to the build environment incrementally, as soon as possible.

Another element is pipelining. In this case, when the development team starts iteration N, the test team starts evaluating the just-finished iteration N–1. See "Overlapping or "Pipelining" Activities Across Iterations" on p. 251.

Q30: **What do we do when time, budget, and scope are all frozen but we still want to apply an iterative or agile method?**

A30: The constraints are irrational, but it happens. Spend more time looking for existing (perhaps open source) components or frameworks, contract with specialists who have done something similar and used the reusable components, hire consultants with an existing template system they can modify to your goals, use the technologies most familiar to the team, don't ignore communication (e.g., a daily Scrum) and lots of testing in a misguided rush to save time, and as usual, rank requirements and implement them across the iterations in rank order. When the deadline comes up, at least you'll have the most valuable elements, if not all.

Q31: **Doesn't iterative development mean that we don't know when we're finished?**

A31: It *is* possible to know when we're finished and what "finished" will mean. There are several ways to tackle this dilemma. One approach is to have an initial requirements workshop (part of the Release Planning Game in XP) in which all or most requirements for the release are identified at a high level, such as just the names of use cases or features (XP story cards), with some brief description. This can be the basis for a *rough* scope, effort, and end-date estimate. Of course, as the project progresses, these high-level requirements will evolve into detailed descriptions and may expand, but this does not imply an endless moving target. Rather, it is a temporarily moving target that over time has smaller and smaller pertubations (see the "cone of uncertainty" on p. 18). In early iterations, the fluctuation in the total requirements set is larger, and then it settles. On average, perhaps 20% into the project, a more complete and stable picture emerges.

One can argue that this period of uncertainty is undesirable and that an up-front waterfall requirements approach is thus preferred, but research shows that the requirements change significantly in any event; evolutionary methods admit and embrace this, waterfall-oriented methods deny or resist it.

Another variation, used in the UP (on larger projects especially), is to not expect a definition of "complete" or an end-date until several iterations into the project, at the end of the elaboration phase. This is analogous to exploratory drilling at an oil field. Management doesn't expect reliable answers until after some phase of investigation. In this approach, there are a series of requirement workshops across the early development iterations. By the end of the last workshop (for example, after three workshops across three iterations), the goal is to have discovered all the requirements at a high level (such as the names of use cases) and defined in detail around 80% of the most significant ones. At this point, there is a relatively reliable definition of what "complete" means.

Other variants are build-to-cost (in the 1970s this was known as design-to-cost) or timeboxing the overall project. In the first, "complete" is defined as whatever is finished by the time a fixed budget is consumed. In the latter, "complete" is defined as whatever is finished by a fixed project end date. Both of these strategies may be coupled with evolutionary delivery.

Q32: **Should I plan the work for all the future iterations to ensure the scope and resources (e.g., people) fit the desired end date?**

A32: Laying out a detailed, predictive schedule does not really satisfy this concern, and in fact by doing so and following it, the team is *less* likely to meet the goal. The underlying problem is superimposing a predictable manufacturing model of planning onto new product development projects. Such a plan can give the illusion of satisfying the concern, but since it is highly speculative, assumes low rates of uncertainty and change, and is not feedback driven, it is less skillful than an adaptive planning method.

Q33: **How do I get feedback when there is little or no user interface?**

A33: Primarily from tests and measurements. This question usually comes up for embedded applications, middleware, or servers, where issues such as memory footprint, memory leaks, load, throughput, responsiveness, and so on are important questions. In a well-run IID project, a growing application is evaluated each iteration with respect to these qualities, in the most realistic test environment possible.

Q34: **Should iteration activities overlap? For example, requirements for the next while testing for the previous?**

A34: In general, no. An exception is discussed on p. 251.

Q35: **How long should iterations be?**

A35: See p. 267.

Q36: **How to handle the design of a database with an iterative process?**

A36: Contrary to whatever fears your database experts may hold, it is both possible and effective to apply evolutionary database design and development, especially with the structured application of **database refactorings**—changes to a database schema that improves its design while retaining both its behavioral and informational semantics.

The details are beyond the scope of this introduction. See www.agiledata.org for discussion of agile database development.

Q37: **Should the customer always be in charge of what gets built each iteration?**

A37: Only XP recommends the customers choose the goals of the next iteration, independent of other advisors. Most other IID methods imply or suggest a collaboration between the customers and chief architect. In early iterations especially, the architect is likely to have recommendations on the priority of requests, prompted by their architectural influence or level of technical risk.

Q38: **How to plan an iteration?**

A38: See many of the tips starting on p. 248.

Q39: **Do I give the results of every iteration to my customer?**

A39: No, except for evaluation and feedback, unless your method is Evo (which promotes evolutionary *delivery* each iteration). This is a common confusion with iterative methods. In fact, there may be 10 or 20 iterations before an application is ready for production or commercial release. The "release" of each iteration (except the last) is an *internal* release for testing and baselining the growing system. Some milestone intermediate releases may be made public for alpha testing. That said, one of the advantages of iterative methods is that some internal releases can become—without extraordinary effort—a production release of lesser goals, if circumstances required.

Q40: **How to do documentation for maintenance, when we want to be agile?**

A40: First, define what to document by need, rather than speculation. Is there anyone who has maintained a prior version of the product? What did they previously find

useful, or miss?

A few tips:

- Put the documentation on a project Web site, such as a Wiki.
- Within many systems there are a few key tricky or subtle elements, or themes. Find those, highlight them, and write a short "technical memo" [Larman01] Wiki page for each.
- It is usually useful to document different **architectural views**. See [Kruchten95] for details.
- **Agile documentation** can be created by splitting the team into pairs, and asking them to document in parallel on different whiteboards. One pair will sketch a logical view of the architecture (perhaps loosely in UML notation) and write some related whiteboard notes, emphasizing the key noteworthy elements in that view. Another pair will sketch a deployment view, another the security view, and so on. A digital picture of each whiteboard is taken, and the pictures inserted on separate Wiki pages, one page for each architectural view. Then, the pairs type in some supporting text on the Wiki page below the picture. Using this approach, I once coached a team that needed only three hours to create the maintenance documentation.
- Some insights are worth capturing with a digital movie; it is quick, low effort, and often rich with information. Place the movie file on the project Wiki. For example, consider an interview with the architect structured so that she discusses each architectural view (logical, deployment, …) in turn. They may be situated at a whiteboard (for sketching) while being filmed. Likewise with an experienced maintenance person.

Q41: **How can I create a work breakdown structure (WBS) without a weekly schedule, or an iteration-by-iteration schedule?**

A41: The key point to appreciate is that a WBS is not—or at least should not be—a schedule. It should be a breakdown of work or tasks independent of how or when they are handled.

Some WBSs are organized at the top level by major project phases—a **phase-oriented WBS**. Such a schedule-oriented, predictive planning approach is not consistent with evolutionary development and adaptive planning.

Some WBSs are organized by a decomposition of tasks within major software design elements (subsystem-1 tasks, subsystem-2 tasks)—a **design-oriented WBS**. This is acceptable if the chosen top-level design elements are sufficiently general or high-level to be guaranteed correct (for example, vague elements such as "UI layer"). However, a design-oriented WBS is usually a dangerous approach, since in evolutionary development there should not be fixed, up-front decision on the major design elements—they need to be discovered and evolved during the early exploratory iterations.

A better approach is a **discipline-oriented WBS** whose top-level elements are major project disciplines with activities that occur in parallel throughout the project (test, change management, project management, development, design, environment, requirements analysis). During an iteration planning session, items from this WBS are chosen (i.e., scheduled) for the iteration.

AGILE

PRACTICE GUIDE

PMI
Project Management Institute
GLOBALSTANDARD

Agile Alliance

AGILE PRACTICE GUIDE

This book was printed utilizing a patented anti-counterfeit print technology designed to prevent unauthorized reproductions. The paper color is gray instead of white. When the pages of the book are copied or scanned a hidden warning message will appear in the background. This security feature is intended to discourage anyone from attempting to illegally reproduce or counterfeit this book.

Library of Congress Cataloging-in-Publication Data has been applied for.

ISBN: 978-1-62825-199-9

Published by:
Project Management Institute, Inc.
14 Campus Boulevard
Newtown Square, Pennsylvania 19073-3299 USA
Phone: +1 610-356-4600
Fax: +1 610-356-4647
Email: customercare@pmi.org
Internet: www.PMI.org

©2017 Project Management Institute, Inc. All rights reserved.

Project Management Institute, Inc. content is copyright protected by U.S. intellectual property law that is recognized by most countries. To republish or reproduce PMI's content, you must obtain our permission. Please go to http://www.pmi.org/permissions for details.

To place a Trade Order or for pricing information, please contact Independent Publishers Group:
Independent Publishers Group
Order Department
814 North Franklin Street
Chicago, IL 60610 USA
Phone: +1 800-888-4741
Fax: +1 312- 337-5985
Email: orders@ipgbook.com (For orders only)

For all other inquiries, please contact the PMI Book Service Center.
PMI Book Service Center
P.O. Box 932683, Atlanta, GA 31193-2683 USA
Phone: 1-866-276-4764 (within the U.S. or Canada) or +1-770-280-4129 (globally)
Fax: +1-770-280-4113
Email: info@bookorders.pmi.org

Printed in the United States of America. No part of this work may be reproduced or transmitted in any form or by any means, electronic, manual, photocopying, recording, or by any information storage and retrieval system, without prior written permission of the publisher.

The paper used in this book complies with the Permanent Paper Standard issued by the National Information Standards Organization (Z39.48—1984).

PMI, the PMI logo, PMBOK, OPM3, PMP, CAPM, PgMP, PfMP, PMI-RMP, PMI-SP, PMI-ACP, PMI-PBA, PROJECT MANAGEMENT JOURNAL, PM NETWORK, PMI TODAY, PULSE OF THE PROFESSION and the slogan MAKING PROJECT MANAGEMENT INDISPENSABLE FOR BUSINESS RESULTS are all marks of Project Management Institute, Inc. For a comprehensive list of

BUSINESS RESULTS. are all marks of Project Management Institute, Inc. For a comprehensive list of PMI trademarks, contact the PMI Legal Department. All other trademarks, service marks, trade names, trade dress, product names and logos appearing herein are the property of their respective owners. Any rights not expressly granted herein are reserved.

SAFe is a registered mark of Scaled Agile, Inc.

Agile Alliance and the Agile Alliance logo are marks of Agile Alliance.

This Practice Guide was jointly funded by Agile Alliance® and was developed in collaboration with members of the Agile Alliance®. Agile Alliance® does not endorse any agile methodology or certification.

10 9 8 7 6 5 4 3 2 1

NOTICE

The Project Management Institute, Inc. (PMI) standards and guideline publications, of which the document contained herein is one, are developed through a voluntary consensus standards development process. This process brings together volunteers and/or seeks out the views of persons who have an interest in the topic covered by this publication. While PMI administers the process and establishes rules to promote fairness in the development of consensus, it does not write the document and it does not independently test, evaluate, or verify the accuracy or completeness of any information or the soundness of any judgments contained in its standards and guideline publications.

PMI disclaims liability for any personal injury, property or other damages of any nature whatsoever, whether special, indirect, consequential or compensatory, directly or indirectly resulting from the publication, use of application, or reliance on this document. PMI disclaims and makes no guaranty or warranty, expressed or implied, as to the accuracy or completeness of any information published herein, and disclaims and makes no warranty that the information in this document will fulfill any of your particular purposes or needs. PMI does not undertake to guarantee the performance of any individual manufacturer or seller's products or services by virtue of this standard or guide.

In publishing and making this document available, PMI is not undertaking to render professional or other services for or on behalf of any person or entity, nor is PMI undertaking to perform any duty owed by any person or entity to someone else. Anyone using this document should rely on his or her own independent judgment or, as appropriate, seek the advice of a competent professional in determining the exercise of reasonable care in any given circumstances. Information and other standards on the topic covered by this

publication may be available from other sources, which the user may wish to consult for additional views or information not covered by this publication.

PMI has no power, nor does it undertake to police or enforce compliance with the contents of this document. PMI does not certify, test, or inspect products, designs, or installations for safety or health purposes. Any certification or other statement of compliance with any health or safety-related information in this document shall not be attributable to PMI and is solely the responsibility of the certifier or maker of the statement.

PREFACE

The Project Management Institute and Agile Alliance® chartered this practice guide to create a greater understanding of agile approaches in their communities. The vision for this practice guide is to equip project teams with tools, situational guidelines, and an understanding of the available agile techniques and approaches to enable better results.

Project teams are using agile approaches in a variety of industries beyond software development. Both organizations realize that expansion has created a need for a common language, open mindedness, and the willingness to be flexible in how products and deliverables are brought to market. In addition, both organizations realize there are multiple ways to achieve successful delivery. There are a broad range of tools, techniques, and frameworks; teams have choices for approaches and practices that fit their project and the organizational culture in order to achieve the desired outcome.

The *Agile Practice Guide* core committee members are from varying backgrounds and use various approaches. Some of the committee members are consultants and some work inside organizations. All have worked in agile ways for many years.

TABLE OF CONTENTS

1. INTRODUCTION

2. AN INTRODUCTION TO AGILE
- 2.1 Definable Work vs. High-Uncertainty Work
- 2.2 The Agile Manifesto and Mindset
- 2.3 Lean and the Kanban Method
- 2.4 Uncertainty, Risk, and Life Cycle Selection

3. LIFE CYCLE SELECTION
- 3.1 Characteristics of Project Life Cycles
  - 3.1.1 Characteristics of Predictive Life Cycles
  - 3.1.2 Characteristics of Iterative Life Cycles
  - 3.1.3 Characteristics of Incremental Life Cycles
  - 3.1.4 Characteristics of Agile Life Cycles
  - 3.1.5 Agile Suitability Filters
  - 3.1.6 Characteristics of Hybrid Life Cycles
  - 3.1.7 Combined Agile and Predictive Approaches
  - 3.1.8 Predominantly Predictive Approach with Some Agile Components
  - 3.1.9 A Largely Agile Approach with a Predictive Component
  - 3.1.10 Hybrid Life Cycles as Fit-For-Purpose
  - 3.1.11 Hybrid Life Cycles as Transition Strategy
- 3.2 Mixing Agile Approaches

3.3 Project Factors That Influence Tailoring

4. IMPLEMENTING AGILE: CREATING AN AGILE ENVIRONMENT
4.1 Start with an Agile Mindset
4.2 Servant Leadership Empowers the Team
4.2.1 Servant Leader Responsibilities
4.2.2 Role of the Project Manager in an Agile Environment
4.2.3 Project Managers Use Servant Leadership
4.3 Team Composition
4.3.1 Agile Teams
4.3.2 Agile Roles
4.3.3 Generalizing Specialists
4.3.4 Team Structures
4.3.5 Dedicated Team Members
4.3.6 Team Workspaces
4.3.7 Overcoming Organizational Silos

5. IMPLEMENTING AGILE: DELIVERING IN AN AGILE ENVIRONMENT
5.1 Charter the Project and the Team
5.2 Common Agile Practices
5.2.1 Retrospectives
5.2.2 Backlog Preparation
5.2.3 Backlog Refinement
5.2.4 Daily Standups
5.2.5 Demonstrations/Reviews
5.2.6 Planning for Iteration-Based Agile
5.2.7 Execution Practices that Help Teams Deliver Value
5.2.8 How Iterations and Increments Help Deliver Working Product
5.3 Troubleshooting Agile Project Challenges
5.4 Measurements in Agile Projects

5.4.1 Agile Teams Measure Results

6. ORGANIZATIONAL CONSIDERATIONS FOR PROJECT AGILITY

6.1 Organizational Change Management

6.1.1 Drivers for Change Management

6.1.2 Readiness for Change

6.2 Organizational Culture

6.2.1 Creating an Environment of Safety

6.2.2 Assessing Culture

6.3 Procurement and Contracts

6.4 Business Practices

6.5 Multiteam Coordination and Dependencies (Scaling)

6.5.1 Frameworks

6.5.2 Considerations

6.6 Agile and the Project Management Office (PMO)

6.6.1 An Agile PMO is Value-Driven

6.6.2 An Agile PMO is Invitation-Oriented

6.6.3 An Agile PMO is Multidisciplinary

6.7 Organizational Structure

6.8 Evolving the Organization

7. A CALL TO ACTION

ANNEX A1
PMBOK® GUIDE MAPPING

ANNEX A2
AGILE MANIFESTO MAPPING

ANNEX A3
OVERVIEW OF AGILE AND LEAN FRAMEWORKS

APPENDIX X1
CONTRIBUTORS AND REVIEWERS

APPENDIX X2

ATTRIBUTES THAT INFLUENCE TAILORING

APPENDIX X3
AGILE SUITABILITY FILTER TOOLS

REFERENCES

BIBLIOGRAPHY

GLOSSARY

LIST OF TABLES AND FIGURES

Figure 2-1. The Four Values of the Agile Manifesto

Figure 2-2. The Twelve Principles Behind the Agile Manifesto

Figure 2-3. The Relationship Between the Agile Manifesto Values, Principles, and Common Practices

Figure 2-4. Agile is a Blanket Term for Many Approaches

Figure 2-5. Uncertainty and Complexity Model Inspired by the Stacey Complexity Model

Figure 3-1. The Continuum of Life Cycles

Figure 3-2. Predictive Life Cycle

Figure 3-3. Iterative Life Cycle

Figure 3-4. A Life Cycle of Varying-Sized Increments

Figure 3-5. Iteration-Based and Flow-Based Agile Life Cycles

Figure 3-6. Agile Development Followed by a Predictive Rollout

Figure 3-7. A Combined Agile and Predictive Approach Used Simultaneously

Figure 3-8. A Largely Predictive Approach with Agile Components

Figure 3-9. A Largely Agile Approach with a Predictive Component
Figure 5-1. Burndown Chart for Remaining Story Points
Figure 5-2. Burnup Chart for Showing Story Points Completed
Figure 5-3. Example of a Kanban Board
Figure 5-4. Feature Chart
Figure 5-5. Product Backlog Burnup Chart
Figure 5-6. Earned Value in an Agile Context
Figure 5-7. Cumulative Flow Diagram of Completed Features
Figure 6-1. The Relationship Between Change Management and Agile Approaches
Figure 6-2. Example of Assessing Organizational Culture
Figure 6-3. Initial Ranked Backlog for Changes
Figure 6-4. Using Backlogs and Kanban Boards to Organize and Track Change Work
Figure A3-1. Agile Approaches Plotted by Breadth and Detail
Figure A3-2. Kanban Board Demonstrating Work in Progress Limits, and a Pull System to Optimize the Flow of Work
Figure A3-3. The Crystal Family of Methods
Figure A3-4. Feature-Driven Development Project Life Cycle
Figure A3-5. DSDM Approach to Constraint-Driven Agility
Figure A3-6. Representatives of Scrum Teams Participating in SoS teams
Figure X3-1. Model for Suitability of Agile Approach
Figure X3-2. Buy-In to Approach Assessment

Figure X3-3. **Trust in Team Assessment**

Figure X3-4. **Assessment for Decision-Making Powers of Team**

Figure X3-5. **Team Size Assessment**

Figure X3-6. **Experience Level Assessment**

Figure X3-7. **Assessment for Access to the Customer/Business**

Figure X3-8. **Likelihood of Change Assessment**

Figure X3-9. **Assessment for Criticality of Product or Service**

Figure X3-10. **Incremental Delivery Assessment**

Figure X3-11. **Suitability Assessment Radar Chart**

Figure X3-12. **Drug Store Project**

Figure X3-13. **Military Messaging Example**

Table 1-1. **In-Scope and Out-of-Scope Items**

Table 3-1. **Characteristics of Four Categories of Life Cycles**

Table 3-2. **Tailoring Options to Improve Fit**

Table 4-1. **Attributes of Successful Agile Teams**

Table 4-2. **Agile Team Roles**

Table 5-1. **Agile Pain Points and Troubleshooting Possibilities**

Table A1-1. **Project Management Process Group and Knowledge Area Mapping**

Table A1-2. **Application of Agile in *PMBOK® Guide* Knowledge Areas**

Table A2-1. **Agile Manifesto Values Covered in the *Agile Practice Guide***

Table A2-2. ***Agile Practice Guide* Mapping of Principles Behind the Agile Manifesto**

Table A3-1. **Scrum Events and Artifacts**

Table A3-2. **The Practices of eXtreme Programming**

Table A3-3. **Defining Principles and Properties of the Kanban Method**

Table A3-4. **The Core Values and Common Properties of Crystal**

Table A3-5. **The Key Elements of the Agile Unified Process**

Table A3-6. **Comparison of LeSS and Scrum**

Table X2-1. **Tailoring Guidelines**

1

INTRODUCTION

Welcome to the *Agile Practice Guide*! This guide was developed as a collaborative effort by the Project Management Institute (PMI) and Agile Alliance®. The members of the core writing team who developed this practice guide included volunteers from both organizations, drawing on subject matter expertise from a broad range of current practitioners and leaders from a diverse range of backgrounds, beliefs, and cultures.

This practice guide provides practical guidance geared toward project leaders and team members adapting to an agile approach in planning and executing projects. While our core writing team recognizes there is staunch support to use predictive approaches and conversely, passion around shifting to an agile mindset, values, and principles, this practice guide covers a practical approach to project agility. This practice guide represents a bridge to understanding the pathway from a predictive approach to an agile approach. In fact, there are similar activities between the two, such as planning, that are handled differently but occur in both environments.

Our core writing team used an agile mindset to collaborate and manage the development of this first edition of the practice guide. As technology and culture changes, future updates and refinements to the practice guide will reflect current approaches.

Our core team adopted a more informal, relaxed writing style for this practice guide than is typical for PMI standards. The guide incorporates new elements, such as tips, sidebars, and case studies to better illustrate key points and concepts. Our team intends for these changes to make this practice guide more readable and user-friendly.

This practice guide goes beyond addressing the use of agile in the computer software development industry, because agile has expanded into non-software

development environments. Manufacturing, education, healthcare and other industries are becoming agile to varying degrees and this use beyond software is within the scope of this practice guide.

AGILE-BASED LEARNING

Education is a prime and fertile ground to expand agile practices beyond software development. Teachers in middle schools, high schools, and universities around the world are beginning to use agile to create a culture of learning. Agile techniques are used to provide focus on prioritizing competing priorities. Face-to-face interaction, meaningful learning, self-organizing teams, and incremental and/or iterative learning that exploit the imagination are all agile principles that can change the mindset in the classroom and advance educational goals (Briggs, 2014).*

*Briggs, Sara. "Agile Based Learning: What Is It and How Can It Change Education?" *Opencolleges.edu.au* February 22, 2014, retrieved from http://www.opencolleges.edu.au/informed/features/agile-based-learning-what-is-it-and-how-can-it-change-education/

So why an *Agile Practice Guide* and why now? Project teams have used agile techniques and approaches in various forms for at least several decades. The Agile Manifesto [1][1] expressed definitive values and principles of agile as the use of agile gained substantial momentum (see Section 2.1). Today, project leaders and teams find themselves in an environment disrupted by exponential advances in technology and demands from customers for more immediate delivery of value. Agile techniques and approaches effectively manage disruptive technologies. In addition, the first principle of agile places customer satisfaction as the highest priority and is key in delivering products and services that delight customers (see Section 2.1). Rapid and transparent customer feedback loops are readily available with the widespread use of social media. Therefore, in order to stay competitive and relevant, organizations can no longer be internally focused but rather need to focus outwardly to the customer experience.

Disruptive technologies are rapidly changing the playing field by decreasing the barriers to entry. More mature organizations are increasingly prone to being highly complex and potentially slow to innovate, and lag behind in delivering new solutions to their customers. These organizations find themselves competing

with smaller organizations and startups that are able to rapidly produce products that fit customer needs. This speed of change will continue to drive large organizations to adopt an agile mindset in order to stay competitive and keep their existing market share.

DISRUPTIVE TECHNOLOGY

Disruptive technology is especially enabled by the transition to cloud computing. Companies across the globe are leveraging the model for quick and cheap access to computing resources and to gain entry into traditional markets. Cloud computing requires a reduced upfront payment, but is paid over time via a subscription service, based upon a pay-as-you-go or pay-what-you-use model. Updated applications, infrastructure, and platforms are released into the cloud in an iterative and incremental fashion, keeping pace with improvements to technology and evolving customer demand.

The *Agile Practice Guide* is project-focused and addresses project life cycle selection, implementing agile, and organizational considerations for agile projects. Organizational change management (OCM) is essential for implementing or transforming practices but, since OCM is a discipline within itself, it is outside the scope of this practice guide. Those seeking guidance in OCM may refer to *Managing Change in Organizations—A Practice Guide* [2].

Additional items that are in scope and out of scope for this practice guide are listed in Table 1-1.

Table 1-1. In-Scope and Out-of-Scope Items

In Scope	Out of Scope
Implementing agile approaches at a project or team level	Implementing agile throughout the organization or creating agile programs
Coverage of most popular agile approaches, as listed in industry surveys	Coverage of niche approaches, company-specific methods, or incomplete life cycle techniques
Suitability factors to consider when choosing an agile approach and/or practice	Recommending or endorsing a particular approach/practice
Mapping agile to *PMBOK® Guide* processes and Knowledge Areas	Change or modification of *PMBOK® Guide* processes and/or Knowledge Areas
Discussion on the use of agile beyond software development	Removal of software industry influence on agile approaches. (Note that software is included in

development	approaches. (Note that software is included in this practice guide even though the use of agile is growing in many other industries beyond software.)
Guidance, techniques, and approaches to consider when implementing agile in projects or organizations	Prescriptive step-by-step instructions on how to implement agile in projects or organizations
Definitions of generally accepted terms	New terms and/or definitions

This practice guide is for project teams who find themselves in the messy middle-ground between predictive and agile approaches, who are trying to address rapid innovation and complexity, and who are dedicated to the team's improvement. This practice guide provides useful guidance for successful projects that deliver business value to meet customer expectations and needs.

This practice guide is organized as follows:

Section 2 An Introduction to Agile—This section includes the Agile Manifesto mindset, values, and principles. It also covers the concepts of definable and high-uncertainty work, and the correlation between lean, the Kanban Method, and agile approaches.

Section 3 Life Cycle Selection—This section introduces the various life cycles discussed in this practice guide. This section also addresses suitability filters, tailoring guidelines, and common combinations of approaches.

Section 4 Implementing Agile: Creating an Agile Environment—This section discusses critical factors to consider when creating an agile environment such as servant leadership and team composition.

Section 5 Implementing Agile: Delivering in an Agile Environment—This section includes information on how to organize teams and common practices teams can use for delivering value on a regular basis. It provides examples of empirical measurements for teams and for reporting status.

Section 6 Organizational Considerations for Project Agility—This section explores organizational factors that impact the use of agile approaches, such as culture, readiness, business practices, and the role of a PMO.

Section 7 A Call to Action—The call to action requests input for continuous improvement of this practice guide.

The annexes, appendixes, references, bibliography, and glossary provide additional useful information and definitions:

- **Annexes.** Contain mandatory information that is too lengthy for inclusion in the main body of the practice guide.
- **Appendixes.** Contain nonmandatory information that supplements the main body of this practice guide.
- **References.** Identify where to locate standards and other publications that are cited in this practice guide.
- **Bibliography.** Lists additional publications by section that provide detailed information on topics covered in this practice guide.
- **Glossary.** Presents a list of terms and their definitions that are used in this practice guide.

---

[1] The numbers in brackets refer to the list of references at the end of this practice guide.

2

AN INTRODUCTION TO AGILE

2.1 DEFINABLE WORK VS. HIGH-UNCERTAINTY WORK

Project work ranges from definable work to high-uncertainty work. Definable work projects are characterized by clear procedures that have proved successful on similar projects in the past. The production of a car, electrical appliance, or home after the design is complete are examples of definable work. The production domain and processes involved are usually well understood and there are typically low levels of execution uncertainty and risk.

New design, problem solving, and not-done-before work is exploratory. It requires subject matter experts to collaborate and solve problems to create a solution. Examples of people encountering high-uncertainty work include software systems engineers, product designers, doctors, teachers, lawyers, and many problem-solving engineers. As more definable work is automated, project teams are undertaking more high-uncertainty work projects that require the techniques described in this practice guide.

High-uncertainty projects have high rates of change, complexity, and risk. These characteristics can present problems for traditional predictive approaches that aim to determine the bulk of the requirements upfront and control changes through a change request process. Instead, agile approaches were created to explore feasibility in short cycles and quickly adapt based on evaluation and feedback.

2.2 THE AGILE MANIFESTO AND MINDSET

Thought leaders in the software industry formalized the agile movement in 2001 with the publication of the Manifesto for Agile Software Development (see Figure 2-1).

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.

© 2001, the Agile Manifesto authors

Figure 2-1. The Four Values of the Agile Manifesto

Twelve clarifying principles flowed from these values as shown in Figure 2-2.

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Figure 2-2. The Twelve Principles Behind the Agile Manifesto

Although originating in the software industry, these principles have since spread to many other industries.

This embodiment of mindset, values, and principles defines what constitutes an agile approach. The various agile approaches in use today share common roots with the agile mindset, value, and principles. This relationship is shown in Figure 2-3.

Figure 2-3. The Relationship Between the Agile Manifesto Values, Principles, and Common Practices

As shown in Figure 2-3, the model, inspired by Ahmed Sidky, articulates agile as a mindset defined by the Agile Manifesto values, guided by the Agile Manifesto principles, and enabled by various practices. It is worth noting that while the term "agile" became popularized after the Manifesto, the approaches and techniques being used by project teams today existed before the Agile Manifesto by many years and, in some cases, decades.

Agile approaches and *agile methods* are umbrella terms that cover a variety of frameworks and methods. Figure 2-4 places agile in context and visualizes it as a blanket term, referring to any kind of approach, technique, framework, method, or practice that fulfills the values and principles of the Agile Manifesto. Figure 2-4 also shows agile and the Kanban Method as subsets of lean. This is because they are named instances of lean thinking that share lean concepts such as: "focus on value," "small batch sizes," and "elimination of waste."

> Is agile an approach, a method, a practice, a technique, or a framework? Any or all of these terms could apply depending on the situation. This practice guide, uses the term "approach" unless one of the other terms is obviously more correct.

Figure 2-4. Agile is a Blanket Term for Many Approaches

In general, there are two strategies to fulfill agile values and principles. The first is to adopt a formal agile approach, intentionally designed and proven to achieve desired results. Then take time to learn and understand the agile approaches before changing or tailoring them. Premature and haphazard tailoring can minimize the effects of the approach and thus limit benefits. *(See Appendix X2 for Tailoring Considerations).*

The second strategy is to implement changes to project practices in a manner that fits the project context to achieve progress on a core value or principle. Use timeboxes to create features, or specific techniques to iteratively refine features. Consider dividing up one large project into several releases, if that works for the specific project context. Implement changes that will help the project succeed—the changes do not need to be part of the organization's formal practices. The end goal is not to be agile for its own sake, but rather to deliver a continuous flow of

value to customers and achieve better business outcomes.

2.3 LEAN AND THE KANBAN METHOD

One way to think about the relationship between lean, agile, and the Kanban Method is to consider agile and the Kanban Method as descendants of lean thinking. In other words, lean thinking is a superset, sharing attributes with agile and Kanban.

This shared heritage is very similar and focuses on delivering value, respect for people, minimizing waste, being transparent, adapting to change, and continuously improving. Project teams sometimes find it useful to blend various methods—whatever works for the organization or team is what should be done regardless of its origin. The objective is the best outcome regardless of the approach used.

The Kanban Method is inspired by the original lean-manufacturing system and used specifically for knowledge work. It emerged in the mid-2000s as an alternative to the agile methods that were prevalent at the time.

The Kanban Method is less prescriptive than some agile approaches and less disruptive, as it is the original "start-where-you-are" approach. Project teams can begin applying the Kanban Method with relative ease and progress toward other agile approaches if that is what they deem necessary or appropriate. For more details on the Kanban Method, see Annex A3 on Overview of Agile and Lean Frameworks.

CASE

There is and probably always will be a lot of discussion around the Kanban Method and whether it belongs to the lean or agile movement. It was conceived in and around lean manufacturing, but is widely used in agile settings.

2.4 UNCERTAINTY, RISK, AND LIFE CYCLE SELECTION

SELECTION

Some projects have considerable uncertainty around project requirements and how to fulfill those requirements using current knowledge and technology. These uncertainties can contribute to high rates of change and project complexity. These characteristics are illustrated in Figure 2-5.

As project uncertainty increases, so too does the risk of rework and the need to use a different approach. To mitigate the impact of these risks, teams select life cycles that allow them to tackle projects with high amounts of uncertainty via small increments of work.

Teams can verify their work when they use small increments and can change what they do next. When teams deliver small increments, they are better able to understand the true customer requirements faster and more accurately than with a static written specification.

Figure 2-5. Uncertainty and Complexity Model Inspired by the Stacey Complexity Model

Teams can plan and manage projects with clear and stable requirements and

clear technical challenges with little difficulty. However, as the uncertainty in the project increases, the likelihood of changes, wasted work, and rework also increases, which is costly and time consuming.

Some teams have evolved project life cycles to use iterative and incremental approaches. Many teams discover that when they explore the requirements iteratively and deliver more often incrementally, the teams adapt to changes more easily. These iterative and incremental approaches reduce waste and rework because the teams gain feedback. These approaches use:

- Very short feedback loops,
- Frequent adaptation of process,
- Reprioritization,
- Regularly updated plans, and
- Frequent delivery.

TIP

What do simple, complicated, and complex projects mean? Consider large projects, such as the Boston Big Dig construction project. On the surface, the project seemed fairly straightforward: move the elevated highway underground. There was high agreement on the requirements (see the Y axis in Figure 2-5). There was low uncertainty on how the project would proceed until the project started. And, as is the case for many large projects, the project encountered surprises along the way.

When a team works on a project where there is little opportunity for interim deliverables or little opportunity for prototyping, the team most likely will use a predictive life cycle to manage it. The team can adapt to what it discovers, but will not be able to use agile approaches to manage the iterative discovery of requirements or incremental deliverables for feedback.

The Big Dig project was not simple by any means. However, many projects that start out in the lower left part of the Stacey Complexity Model have no real means of moving to other approaches. Assess the project, both in the requirements and the means of delivery, to determine the best approach for the life cycle of the project.

These iterative, incremental, and agile approaches work well for projects that involve new or novel tools, techniques, materials, or application domains. (Refer to Section 3 on Life Cycle Selection). They also work well for projects that:

- Require research and development;
- Have high rates of change;
- Have unclear or unknown requirements, uncertainty, or risk; or
- Have a final goal that is hard to describe.

By building a small increment and then testing and reviewing it, the team can explore uncertainty at a low cost in a short time, reduce risk, and maximize business value delivery. This uncertainty may be centered on suitability and requirements (is the right product being built?); technical feasibility and performance (can this product be built this way?); or process and people (is this an effective way for the team to work?). All three of these characteristics—product specification, production capability, and process suitability—typically have elements of high uncertainty.

However, iterative and incremental approaches have their limits of applicability. When both technology uncertainty and requirements uncertainty are very high (the top right of Figure 2-5), the project moves beyond complex to chaotic. In order for the project to become reliably possible, it needs one of the variables (uncertainty or disagreement) to be contained.

3

LIFE CYCLE SELECTION

Projects come in many shapes and there are a variety of ways to undertake them. Project teams need awareness of the characteristics and options available to select the approach most likely to be successful for the situation.

This practice guide refers to four types of life cycles, defined as follows:

- **Predictive life cycle.** A more traditional approach, with the bulk of planning occurring upfront, then executing in a single pass; a sequential process.
- **Iterative life cycle.** An approach that allows feedback for unfinished work to improve and modify that work.
- **Incremental life cycle.** An approach that provides finished deliverables that the customer may be able to use immediately.
- **Agile life cycle.** An approach that is both iterative and incremental to refine work items and deliver frequently.

WHAT TO CALL NON-AGILE APPROACHES?

There is no single term that is universally used to describe non-agile approaches. Initially, the practice guide used the term *plan-driven* to describe the emphasis on an upfront plan and then execution of that plan. Some people prefer the terms *waterfall* or *serial* to describe this life cycle. In the end, we settled on the term *predictive* since it is used in *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)* [3] and the *Software Extension to the PMBOK® Guide Fifth Edition* [4].

Many organizations do not experience either of these extremes and instead

occupy some middle ground. That is natural, but we still need a way to talk about both ends of the spectrum. If *agile* is at one end, we call the other end *predictive*.

3.1 CHARACTERISTICS OF PROJECT LIFE CYCLES

Table 3-1 summarizes the characteristics of the four life cycle categories covered in this practice guide.

Table 3-1. Characteristics of Four Categories of Life Cycles

Characteristics				
Approach	**Requirements**	**Activities**	**Delivery**	**Goal**
Predictive	Fixed	Performed once for the entire project	Single delivery	Manage cost
Iterative	Dynamic	Repeated until correct	Single delivery	Correctness of solution
Incremental	Dynamic	Performed once for a given increment	Frequent smaller deliveries	Speed
Agile	Dynamic	Repeated until correct	Frequent small deliveries	Customer value via frequent deliveries and feedback

It is important to note that all projects have these characteristics—no project is completely devoid of considerations around requirements, delivery, change, and goals. A project's inherent characteristics determine which life cycle is the best fit for that project.

Another way to understand how project life cycles vary is by using a continuum ranging from predictive cycles on one end, to agile cycles on the other end, with more iterative or incremental cycles in the middle.

Figure X3-1 of Appendix X3 of the *PMBOK® Guide* – Sixth Edition displays the continuum as a flat line. This view emphasizes the shifting of project characteristics from one end to the other. Another way to visualize the continuum is with a two-dimensional square, as shown in Figure 3-1.

Figure 3-1. The Continuum of Life Cycles

No life cycle can be perfect for all projects. Instead, each project finds a spot on the continuum that provides an optimum balance of characteristics for its context. Specifically,

- **Predictive life cycles.** Take advantage of things that are known and proven. This reduced uncertainty and complexity allows teams to segment work into a sequence of predictable groupings.
- **Iterative life cycles.** Allow feedback on partially completed or unfinished work to improve and modify that work.
- **Incremental life cycles.** Provide finished deliverables that the customer may be able to use immediately.
- **Agile life cycles.** Leverage both the aspects of iterative and incremental characteristics. When teams use agile approaches, they iterate over the product to create finished deliverables. The team gains early feedback and provides customer visibility, confidence, and

control of the product. Because the team can release earlier, the project may provide an earlier return on investment because the team delivers the highest value work first.

PLANNING IS ALWAYS THERE

A key thing to remember about life cycles is that each of them share the element of planning. What differentiates a life cycle is not whether planning is done, but rather how much planning is done and when.

At the predictive end of the continuum, the plan drives the work. As much planning as is possible is performed upfront. Requirements are identified in as much detail as possible. The team estimates when they can deliver which deliverables and performs comprehensive procurement activities.

In iterative approaches, prototypes and proofs are also planned, but the outputs are intended to modify the plans created in the beginning. Earlier reviews of unfinished work help inform future project work.

Meanwhile, incremental initiatives plan to deliver successive subsets of the overall project. Teams may plan several successive deliveries in advance or only one at a time. The deliveries inform the future project work.

Agile projects also plan. The key difference is that the team plans and replans as more information becomes available from review of frequent deliveries. Regardless of the project life cycle, the project requires planning.

3.1.1 CHARACTERISTICS OF PREDICTIVE LIFE CYCLES

Predictive life cycles expect to take advantage of high certainty around firm requirements, a stable team, and low risk. As a result, project activities often execute in a serial manner, as shown in Figure 3-2.

In order to achieve this approach, the team requires detailed plans to know what to deliver and how. These projects succeed when other potential changes are restricted (e.g., requirements changes; project team members change what the team delivers). Team leaders aim to minimize change for the predictive project.

When the team creates detailed requirements and plans at the beginning of the project, they can articulate the constraints. The team can then use those

constraints to manage risk and cost. As the team progresses through the detailed plan, they monitor and control changes that might affect the scope, schedule, or budget.

By emphasizing a departmentally efficient, serialized sequence of work, predictive projects do not typically deliver business value until the end of the project. If the predictive project encounters changes or disagreements with the requirements, or if the technological solution is no longer straightforward, the predictive project will incur unanticipated costs.

Figure 3-2. Predictive Life Cycle

3.1.2 CHARACTERISTICS OF ITERATIVE LIFE CYCLES

Iterative life cycles improve the product or result through successive prototypes or proofs of concept. Each new prototype yields new stakeholder feedback and team insights. Then, the team incorporates the new information by repeating one or more project activities in the next cycle. Teams may use timeboxing on a given iteration for a few weeks, gather insights, and then rework the activity based on those insights. In that way, iterations help identify and reduce uncertainty in the project.

Projects benefit from iterative life cycles when complexity is high, when the project incurs frequent changes, or when the scope is subject to differing stakeholders' views of the desired final product. Iterative life cycles may take longer because they are optimized for learning rather than speed of delivery.

Figure 3-3 illustrates some elements of an iterative project life cycle for a single product delivery.

Figure 3-3. Iterative Life Cycle

Have you ever been involved on a project where the requirements seemed to change daily and thought, "We will know the requirements when we deliver a prototype that the business approves." If so, this was a project where agile approaches could have helped. A prototype encourages feedback and a better understanding of the requirements that can be incorporated into each deliverable.

3.1.3 CHARACTERISTICS OF INCREMENTAL LIFE CYCLES

Some projects optimize for speed of delivery. Many businesses and initiatives cannot afford to wait for everything to be completed; in these cases, customers are willing to receive a subset of the overall solution. This frequent delivery of smaller deliverables is called an incremental life cycle (see Figure 3-4).

Figure 3-4. A Life Cycle of Varying-Sized Increments

TIP

Are you unsure of how a new business service might work in practice? Create a proof of concept with evaluation criteria to explore desired outcomes. Use iterative approaches when you suspect the requirements will change based on customer feedback.

Incremental life cycles optimize work for delivering value to sponsors or customers more often than a single, final product. Teams plan initial deliverables before beginning their work, and they begin working on that first delivery as soon as possible. Some agile projects deliver value within days of project initiation. Others could take longer, ranging from 1 week to several weeks.

As the project continues, the team may deviate from the original vision. The team can manage the deviations, because the team delivers value sooner. The degree of change and variation is less important than ensuring customers get value sooner than at the end of the project.

Completeness and delivery are subjective. The team may need feedback on a prototype and may then choose to deliver a minimum viable product (MVP) to a subset of customers. The customers' feedback helps the team to learn what they need to provide for subsequent delivery of the final finished feature.

Agile teams, as a key differentiator, deliver business value often. As the product adds a broader set of features and a broader range of consumers, we say it is delivered incrementally.

Providing a customer a single feature or a finished piece of work is an example of the incremental approach.

For example, builders may want to show a finished room or floor of a building before they continue with the remainder of the building. In that case, they may complete a floor with fixtures, paint, and everything else intended for the finished floor before proceeding to the next floor. The customer is able to see and approve of the style, color, and other details, allowing adjustments to be made before further investments of time and money are made. This reduces potential rework and/or customer dissatisfaction.

3.1.4 CHARACTERISTICS OF AGILE LIFE CYCLES

In an agile environment, the team expects requirements to change. The iterative and incremental approaches provide feedback to better plan the next part of the project. However, in agile projects, incremental delivery uncovers hidden or misunderstood requirements. Figure 3-5 illustrates two possible ways to achieve incremental delivery so the project aligns with customer needs and can be adapted as necessary.

Figure 3-5. Iteration-Based and Flow-Based Agile Life Cycles

In iteration-based agile, the team works in iterations (timeboxes of equal duration) to deliver completed features. The team works on the most important feature, collaborating as a team to finish it. Then the team works on the next most important feature and finishes it. The team may decide to work on a few features at a time, but the team does not address all of the work for the iteration at once (i.e., does not address all of the requirements, followed by all of the analyses, etc.).

In flow-based agile, the team pulls features from the backlog based on its capacity to start work rather than on an iteration-based schedule. The team defines its workflow with columns on a task board and manages the work in progress for each column. Each feature may take a different amount of time to finish. Teams keep work-in-progress sizes small to better identify issues early and reduce rework should changes be required. Without iterations to define planning and review points, the team and business stakeholders determine the most appropriate schedule for planning, product reviews, and retrospectives.

Agile life cycles are those that fulfill the principles of the Agile Manifesto. In particular, customer satisfaction increases with early and continuous delivery of valuable products. Moreover, an incremental deliverable that is functional and provides value is the primary measure of progress. Agile life cycles combine both iterative and incremental approaches in order to adapt to high degrees of change and deliver project value more often.

3.1.5 AGILE SUITABILITY FILTERS

Various assessment models exist to help determine the likely fit or gaps for using agile approaches. These models assess project and organizational factors associated with adoption and suitability and then provide scores indicating alignment or potential risk areas. Appendix X3 provides a synthesis of popular assessment models for use as an agile suitability filter.

EXAMPLE OF A HYBRID LIFE CYCLE PROJECT

A pharmaceutical company that had a time-consuming U.S. Food and Drug Administration (FDA) approval process tagged onto the end of its development process and its entire life cycle looked like Figure 3-6. While project teams undertook drug trials in an agile fashion, they had to present the drugs to an external group to perform the FDA approval process. A consultant helped to integrate the FDA approval process portion into the agile development process to create a more streamlined hybrid approach.

The short version of the story is that because FDA approval is required to be completed at the end of the development process or repeated after any change (this includes even after the most minor change), the process had to remain at the end as a separate phase. Integration using the iterative process was unsuccessful. However, the consultant created some useful quick-start

guides and testing protocols that shortened the final FDA approval process.

3.1.6 CHARACTERISTICS OF HYBRID LIFE CYCLES

It is not necessary to use a single approach for an entire project. Projects often combine elements of different life cycles in order to achieve certain goals. A combination of predictive, iterative, incremental, and/or agile approaches is a hybrid approach.

Figure 3-6 depicts the basic, pure approaches to project types that combine to form a hybrid model. The early processes utilize an agile development life cycle, which is then followed by a predictive rollout phase. This approach can be used when there is uncertainty, complexity, and risk in the development portion of the project that would benefit from an agile approach, followed by a defined, repeatable rollout phase that is appropriate to undertake in a predictive way, perhaps by a different team. An example of this approach is the development of a new high-tech product followed by rollout and training to thousands of users.

Figure 3-6. Agile Development Followed by a Predictive Rollout

3.1.7 COMBINED AGILE AND PREDICTIVE APPROACHES

Another approach is to use a combination of agile and predictive approaches throughout the life cycle.

Figure 3-7. A Combined Agile and Predictive Approach Used Simultaneously

In Figure 3-7, a combination of both predictive and agile approaches are used in the same project. Perhaps the team is incrementally transitioning to agile and using some approaches like short iterations, daily standups, and retrospectives,

but other aspects of the project such as upfront estimation, work assignment, and progress tracking are still following predictive approaches.

Using both predictive and agile approaches is a common scenario. It would be misleading to call the approach agile since it clearly does not fully embody the agile mindset, values, and principles. However, it would also be inaccurate to call it predictive since it is a hybrid approach.

3.1.8 PREDOMINANTLY PREDICTIVE APPROACH WITH SOME AGILE COMPONENTS

Figure 3-8 shows a small agile element within a chiefly predictive project. In this case, a portion of the project with uncertainty, complexity, or opportunity for scope creep is being tackled in an agile way, but the remainder of the project is being managed using predictive approaches. An example of this approach would be an engineering firm that is building a facility with a new component.

Figure 3-8. A Largely Predictive Approach with Agile Components

While the majority of the project may be routine and predictable, like many other facility projects the organization has undertaken before, this project incorporates a new roofing material. The contractor may plan for some small-scale installation trials on the ground first to determine the best installation method and to uncover issues early while there is plenty of time to solve them and incrementally improve processes through experimentation and adaptation.

3.1.9 A LARGELY AGILE APPROACH WITH A PREDICTIVE COMPONENT

Figure 3-9 depicts a largely agile approach with a predictive component. This approach might be used when a particular element is non-negotiable or not executable using an agile approach. Examples include integrating an external component developed by a different vendor that cannot or will not partner in a collaborative or incremental way. A single integration is required after the component is delivered.

Figure 3-9. A Largely Agile Approach with a Predictive Component

A government department had a credit insurance application development project. The multi-year project was to replace its aging underwriting system with a new, more responsive user interface and system integrations. The bulk of the project was undertaken using an agile approach with continual business input.

The premium rate calculations were handed down from the Organisation for Economic Co-operation and Development (OECD) as a 200-page specification. The steps were very clearly explained with little opportunity for confusion (or interim result confirmation by the business) and were coded up by a separate team working its way through the calculation steps. The two teams collaborated on the input variables required for the calculation and how to consume and display the output values, but beyond that, the calculation team worked in a largely predictive manner.

When the calculation team's portion was complete, the outputs from the premium rate calculations were displayed on the screens and in the reports. Then the business users provided feedback on the appearance and use of the information. The two teams ran concurrently, but had little need for interaction. Having them physically close to each other made it easier to check in on development progress, but largely they were separate subprojects.

3.1.10 HYBRID LIFE CYCLES AS FIT-FOR-PURPOSE

Project teams may design a hybrid life cycle based on project risks. For example, a campus construction project may have multiple buildings to improve and build. An incremental approach would focus resources on completing some buildings earlier than others, accelerating the return on investment. Each individual delivery may be sufficiently well known to benefit from a predictive life cycle for that building alone.

The goal of project management is to produce business value in the best possible way given the current environment. It does not matter if that way is

agile or predictive. The question to ask is: “How can we be most successful?”

Is feedback needed as the team produces value? If so, increments will help. Is it necessary to manage risk as ideas are explored? If so, iterations or agile will help.

When the organization cannot deliver intermediate value, agile approaches may not be useful. That is okay—agile for the sake of agile is not the goal. The point is to select a life cycle or a combination of life cycles that work for the project, the risks, and the culture.

Agile is about customer-based delivery on a frequent basis. That delivery creates feedback for the team. The team uses that feedback to plan and replan the next chunk of work.

3.1.11 HYBRID LIFE CYCLES AS TRANSITION STRATEGY

Many teams are not able to make the switch to agile ways of working overnight. Agile techniques look and feel very different to those who are accustomed to and have been successful in a predictive environment. The larger the organization and the more moving parts, the longer it will take to transition. For that reason, it makes sense to plan a gradual transition.

A gradual transition involves adding more iterative techniques to improve learning and alignment among teams and stakeholders. Later, consider adding more incremental techniques to accelerate value and return on investment to sponsors. This combination of various approaches is considered a hybrid approach.

Try these new techniques on a less risky project with a medium- to low-degree of uncertainty. Then, when the organization is successful with a hybrid approach, try more complex projects that require more of those techniques to be added. This is a way to tailor the progressive hybrid transition to the organization's situation and specific risks and the team's readiness to adapt and embrace the changes.

3.2 MIXING AGILE APPROACHES

Agile teams rarely limit their practices to one agile approach. Each project context has its own peculiarities, such as the varied mix of team member skills

and backgrounds; the various components of the product under development; and the age, scale, criticality, complexity, and regulatory constraints of the environment in which the work takes place.

Agile frameworks are not customized for the team. The team may need to tailor practices to deliver value on a regular basis. Often, teams practice their own special blend of agile, even if they use a particular framework as a starting point.

BLENDING APPROACHES

As an example of tailoring agile frameworks, one of the most common blends in widespread use involves a coordinated use of the Scrum framework, the Kanban Method, and elements of the eXtreme Programming (XP) method. Scrum provides guidance on the use of a product backlog, a product owner, scrum master, and a cross-functional development team, including sprint planning, daily scrum, sprint review, and sprint retrospective sessions. A kanban board helps the team to further improve its effectiveness by visualizing the flow of work, making impediments easily visible, and allowing flow to be managed by adjusting work in process limits. In addition, XP-inspired engineering practices such as use of story cards, continuous integration, refactoring, automated testing, and test-driven development further increase the effectiveness of the agile team. In summary, the blend of practices from these various sources produces a synergistic result of higher performance than each individual component in isolation.

3.3 PROJECT FACTORS THAT INFLUENCE TAILORING

Sometimes project attributes require tailoring an approach for a better fit. Table 3-2 identifies some project factors and tailoring options to consider.

Table 3-2. Tailoring Options to Improve Fit

Project Factor	Tailoring Options
Demand pattern: steady or sporadic	Many teams find that using a cadence (in the form of a regular timebox) helps them demo, retrospect, and take in new work. In addition,

	some teams need more flexibility in their acceptance of more work. Teams can use flow-based agile with a cadence to get the best of both worlds.
Rate of process improvement required by the level of team experience	Retrospect more often and select improvements.
The flow of work is often interrupted by various delays or impediments	Consider making work visible using kanban boards and experimenting with limits for the various areas of the work process in order to improve flow.
The quality of the product increments is poor	Consider using the various test-driven development practices. This mistake-proofing discipline makes it difficult for defects to remain undetected.
More than one team is needed to build a product	To scale from one to several agile teams, with minimal disruption, first learn about agile program management or formal scaling frameworks. Then, craft an approach that fits the project context.
The project team members are inexperienced in the use of agile approaches	Consider starting by training team members in the fundamentals of the agile mindset and principles. If the team decides to use a specific approach such as Scrum or Kanban, provide a workshop on that approach so the team members can learn how to use it.

For additional guidance on factors that influence tailoring see Appendix X2 on Attributes that Influence Tailoring.

4

IMPLEMENTING AGILE: CREATING AN AGILE ENVIRONMENT

4.1 START WITH AN AGILE MINDSET

Managing a project using an agile approach requires that the project team adopt an agile mindset. The answers to the following questions will help to develop an implementation strategy:

- How can the project team act in an agile manner?
- What can the team deliver quickly and obtain early feedback to benefit the next delivery cycle?
- How can the team act in a transparent manner?
- What work can be avoided in order to focus on high-priority items?
- How can a servant-leadership approach benefit the achievement of the team's goals?

4.2 SERVANT LEADERSHIP EMPOWERS THE TEAM

Agile approaches emphasize servant leadership as a way to empower teams. Servant leadership is the practice of leading through service to the team, by focusing on understanding and addressing the needs and development of team members in order to enable the highest possible team performance.

The role of a servant leader is to facilitate the team's discovery and definition

of agile. Servant leaders practice and radiate agile. Servant leaders approach project work in this order:

- **Purpose.** Work with the team to define the “why” or purpose so they can engage and coalesce around the goal for the project. The entire team optimizes at the project level, not the person level.
- **People.** Once the purpose is established, encourage the team to create an environment where everyone can succeed. Ask each team member to contribute across the project work.
- **Process.** Do not plan on following the “perfect” agile process, but instead look for the results. When a cross-functional team delivers finished value often and reflects on the product and process, the teams are agile. It does not matter what the team calls its process.

The following characteristics of servant leadership enable project leaders to become more agile and facilitate the team's success:

- Promoting self-awareness;
- Listening;
- Serving those on the team;
- Helping people grow;
- Coaching vs. controlling;
- Promoting safety, respect, and trust; and
- Promoting the energy and intelligence of others.

Servant leadership is not unique to agile. But once having practiced it, servant leaders can usually see how well servant leadership integrates into the agile mindset and value.

When leaders develop their servant leadership or facilitative skills, they are more likely to become agile. As a result, servant leaders can help their teams collaborate to deliver value faster.

Successful agile teams embrace the growth mindset, where people believe they can learn new skills. When the team and the servant leaders believe they can all learn, everyone becomes more capable.

4.2.1 SERVANT LEADER RESPONSIBILITIES

4.2.1 SERVANT LEADER RESPONSIBILITIES

Servant leaders manage relationships to build communication and coordination within the team and across the organization. These relationships help the leaders navigate the organization to support the team. This kind of support helps to remove impediments and facilitates the team to streamline its processes. Because servant leaders understand agile and practice a specific approach to agile, they can assist in fulfilling the team's needs.

4.2.1.1 SERVANT LEADERS FACILITATE

When project managers act as servant leaders, the emphasis shifts from “managing coordination” to “facilitating collaboration.” Facilitators help everyone do their best thinking and work. Facilitators encourage the team's participation, understanding, and shared responsibility for the team's output. Facilitators help the team create acceptable solutions.

Servant leaders promote collaboration and conversation within the team and between teams. For example, a servant leader helps to expose and communicate bottlenecks inside and between teams. Then the teams resolve those bottlenecks.

Additionally, a facilitator encourages collaboration through interactive meetings, informal dialog, and knowledge sharing. Servant leaders do this by becoming impartial bridge-builders and coaches, rather than by making decisions for which others should be responsible.

4.2.1.2 SERVANT LEADERS REMOVE ORGANIZATIONAL IMPEDIMENTS

The first value of the Agile Manifesto is individuals and interactions over processes and tools. What better responsibility for a servant leader to take on than to take a hard look at processes that are impeding a team's or organization's agility and work to streamline them? For example, if a department requires extensive documentation, the role of the servant leader could be to work with that department to review required documentation, assist with creating a shared understanding of how agile deliverables meet those requirements, and evaluate the amount of documentation required so teams are spending more time delivering a valuable product instead of producing exhaustive documentation.

Servant leaders should also look at other processes that are lengthy, causing bottlenecks and impeding a team's or organization's agility. Examples of

processes or departments that may need to be addressed include finance, change control boards, or audits. Servant leaders can partner and work with others to challenge them to review their processes to support agile teams and leaders. For example, what good is it for the team to deliver working product every 2 weeks only to have the product fall into a queue or process that could take 6 or more weeks to release due to lengthy release processes? Far too many organizations have these “bottleneck” processes that prevent teams from quickly delivering valuable products or services. The servant leader has the ability to change or remove these organizational impediments to support delivery teams.

INTERPERSONAL SKILLS VERSUS TECHNICAL SKILLS

In addition to servant leadership, team members emphasize their interpersonal and emotional intelligence skills—not just technical skills. Everyone on the team works to exhibit more initiative, integrity, emotional intelligence, honesty, collaboration, humility, and willingness to communicate in various ways so that the entire team can work together well.

The team needs these skills so they can respond well to changes in project direction and technical product changes. When everyone can adapt to the work and to each other, the entire team is more likely to succeed.

4.2.1.3 SERVANT LEADERS PAVE THE WAY FOR OTHERS’ CONTRIBUTION

In agile, the team manages its work process and its work product. That self-management and self-organization applies to everyone serving and supporting the organization and project. Servant leaders work to fulfill the needs of the teams, projects, and organization. Servant leaders may work with facilities for a team space, work with management to enable the team to focus on one project at a time, or work with the product owner to develop stories with the team. Some servant leaders work with auditors to refine the processes needed in regulatory environments, and some servant leaders work with the finance department to transition the organization to incremental budgeting.

The servant leader focuses on paving the way for the team to do its best work. The servant leader influences projects and encourages the organization to think differently.

4.2.1.4 CONSIDER THESE SERVANT LEADER RESPONSIBILITIES

Servant leaders can have many possible titles, but what is most important is what they do. Here are some examples of the responsibilities a servant leader may have:

- Educate stakeholders around why and how to be agile. Explain the benefits of business value based on prioritization, greater accountability and productivity of empowered teams, and improved quality from more frequent reviews, etc.
- Support the team through mentoring, encouragement, and support. Advocate for team members training and career development. The oxymoronic quote “We lead teams by standing behind them” speaks to the role of the leader in developing their team members. Through support, encouragement, and professional development, team members gain confidence, take on larger roles, and contribute at higher levels within their organizations. A key role of the servant leader is to nurture and grow team members through and beyond their current roles, even if that means losing them from the team.
- Help the team with technical project management activities like quantitative risk analysis. Sometimes team members may not have knowledge or experience in roles or functions. Servant leaders who may have more exposure or training in techniques can support the team by providing training or undertaking these activities.
- Celebrate team successes and support and bridge building activities with external groups. Create upward spirals of appreciation and good will for increased collaboration.

4.2.2 ROLE OF THE PROJECT MANAGER IN AN AGILE ENVIRONMENT

The role of the project manager in an agile project is somewhat of an unknown, because many agile frameworks and approaches do not address the role of the project manager. Some agile practitioners think the role of a project manager is not needed, due to self-organizing teams taking on the former responsibilities of the project manager. However, pragmatic agile practitioners

and organizations realize that project managers can add significant value in many situations. The key difference is that their roles and responsibilities look somewhat different.

TIP

> *The value of project managers is not in their position, but in their ability to make everyone else better.*

4.2.3 PROJECT MANAGERS USE SERVANT LEADERSHIP

The *PMBOK® Guide* – Sixth Edition, defines the project manager as "the person assigned by the performing organization to lead the team that is responsible for achieving the project objectives."

Many project managers are accustomed to being at the center of coordination for the project, tracking and representing the team's status to the rest of the organization. This approach was fine when projects were decomposed into siloed functions.

However, in high-change projects, there is more complexity than one person can manage. Instead, cross-functional teams coordinate their own work and collaborate with the business representative (the product owner).

When working on an agile project, project managers shift from being the center to serving the team and the management. In an agile environment, project managers are servant leaders, changing their emphasis to coaching people who want help, fostering greater collaboration on the team, and aligning stakeholder needs. As a servant leader, project managers encourage the distribution of responsibility to the team: to those people who have the knowledge to get work done.

4.3 TEAM COMPOSITION

A core tenet in both the values and the principles of the Agile Manifesto is the importance of individuals and interactions. Agile optimizes the flow of value, emphasizing rapid feature delivery to the customer, rather than on how people are "utilized."

TIP

> *Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.*

When teams think about how to optimize the flow of value, the following benefits become apparent:

- People are more likely to collaborate.
- Teams finish valuable work faster.
- Teams waste much less time because they do not multitask and have to re-establish context.

4.3.1 AGILE TEAMS

Agile teams focus on rapid product development so they can obtain feedback. In practice, the most effective agile teams tend to range in size from three to nine members. Ideally, agile teams are colocated in a team space. Team members are 100% dedicated to the teams. Agile encourages self-managing teams, where team members decide who will perform the work within the next period's defined scope. Agile teams thrive with servant leadership. The leaders support the teams’ approach to their work.

Cross-functional agile teams produce functional product increments frequently. That is because the teams collectively own the work and together have all of the necessary skills to deliver completed work.

Regardless of the overall agile approach, the more a team limits its work in progress, the more likely its members can collaborate to expedite work across the board. Team members in successful agile teams work to collaborate in various ways (such as pairing, swarming, and mobbing) so they do not fall into the trap of mini-waterfalls instead of collaborative work. Mini-waterfalls occur when the team addresses *all* of the requirements in a given period, then attempts to do *all* of the design, then moves on to do *all* of the building. Using this scenario, at some point in the building or the testing following the building, the team may realize it had assumptions that are no longer valid. In this case, the team wasted time in addressing *all* of the requirements. Instead, when team members collaborate to produce a small number of features across the board,

they learn as they proceed and deliver smaller finished features.

Agile projects benefit from project team structures that improve collaboration within and among the teams. Table 4-1 shows how collaborative team members boost productivity and facilitate innovative problem solving.

Table 4-1. Attributes of Successful Agile Teams

Attribute	Goal
Dedicated people	• Increased focus and productivity • Small team, fewer than ten people
Cross-functional team members	• Develop and deliver often • Deliver finished value as an independent team • Integrate all the work activities to deliver finished work • Provide feedback from inside the team and from others, such as the product owner
Colocation or ability to manage any location challenges	• Better communication • Improved team dynamics • Knowledge sharing • Reduced cost of learning • Able to commit to working with each other
Mixed team of generalists and specialists	• Specialists provide dedicated expertise and generalists provide flexibility of who does what • Team brings their specialist capabilities and often become generalizing specialists, with a focus specialty plus breadth of experience across multiple skills
Stable work environment	• Depend on each other to deliver • Agreed-upon approach to the work • Simplified team cost calculations (run rate) • Preservation and expansion of intellectual capital

4.3.2 AGILE ROLES

In agile, three common roles are used:

- Cross-functional team members,
- Product owner, and
- Team facilitator.

Table 4-2 describes these team roles.

Table 4-2. Agile Team Roles

Role	Description
Cross-functional team member	Cross-functional teams consist of team members with all the skills necessary to produce a working product. In software development, cross-functional teams are typically comprised of designers, developers, testers, and any other required roles. The cross-functional development teams consist of professionals who deliver potentially releasable product on a regular cadence. Cross-functional teams are critical because they can deliver finished work in the shortest possible time, with higher quality, without external dependencies.
Product owner	The product owner is responsible for guiding the direction of the product. Product owners rank the work based on its business value. Product owners work with their teams daily by providing product feedback and setting direction on the next piece of functionality to be developed/delivered. That means the work is small, often small enough to be described on one index card.
	The product owner works with stakeholders, customers, and the teams to define the product direction. Typically, product owners have a business background and bring deep subject matter expertise to the decisions. Sometimes, the product owner requests help from people with deep domain expertise, such as architects, or deep customer expertise, such as product managers. Product owners need training on how to organize and manage the flow of work through the team.
	In agile, the product owners create the backlog for and with the team. The backlog helps the teams see how to deliver the highest value without creating waste.
	A critical success factor for agile teams is strong product ownership. Without attention to the highest value for the customer, the agile team may create features that are not appreciated, or otherwise insufficiently valuable, therefore wasting effort.
Team facilitator	The third role typically seen on agile teams is of a team facilitator, a servant leader. This role may be called a project manager, scrum master, project team lead, team coach, or team facilitator.
	All agile teams need servant leadership on the team. People need time to build their servant leadership skills of facilitation, coaching, and impediment removal.
	Initially, many organizations invite external agile coaches to help them when their internal coaching capability is not yet fully developed.
	External coaches have the advantage of experience, but the disadvantage of weak relationships in the client organization. Internal coaches, on the other hand, have strong relationships in their organization, but may lack the breadth of experience that would make

	them highly effective.

> “I-SHAPED PEOPLE AND T-SHAPED PEOPLE”
>
> Some people have deep specializations in one domain, but rarely contribute outside of that domain. These people are known in agile communities as “I-shaped people” since, like the letter “I,” they have depth, but not much breadth. By contrast “T-shaped people” supplement their expertise in one area with supporting, but less-developed skills in associated areas and good collaboration skills. As an example, a person who can test some areas of the product and develop different areas of the product is considered to be a T-shaped person.
>
> A T-shaped person has a defined, recognized specialization and primary role, but has the skills, versatility, and aptitude for collaboration to help other people when and where necessary. This collaboration reduces hand-offs and the constraints of only one person being able to do the job.

4.3.3 GENERALIZING SPECIALISTS

Agile teams are cross-functional, but the people often do not start off that way. However, many successful agile teams are made up of generalizing specialists, or “T-shaped” people.

This means team members have both a focus specialty plus a breadth of experience across multiple skills, rather than a single specialization. Agile team members work to develop such characteristics due to intense collaboration and self-organization to swarm and get work done quickly, which requires them to routinely help each other.

A single person's throughput is not relevant. Focusing on a single person's throughput may even be harmful if it creates a bottleneck for the rest of the team. The goal is for the *team* to optimize the delivery of finished work to get feedback.

If the customer desires great results, such as rapid feature delivery with excellent quality, the team cannot be structured just with specialist roles in an

attempt to maximize resource efficiency. The team's objective is flow efficiency, optimizing the throughput of the entire team. Small batch sizes promote working together as a team. The product owner's job is to make sure the team works on the highest-value work.

4.3.4 TEAM STRUCTURES

Teams have adopted agile principles and practices across many industries. They organize people into cross-functional teams to iteratively develop working products.

CASE

The core team assembled to write this practice guide had varied backgrounds—some represented PMI and some represented Agile Alliance. They were self-organizing and worked in increments to complete the work. PMI assembled a group of subject matter experts to inspect the work, and this allowed the team to incorporate feedback and improve the product as it was developed. However the core team was not representative of a typical agile team because its members' time was not 100% dedicated to this endeavor.

Some organizations have been able to create colocated, cross-functional teams; others have a different situation. Instead of having all team members colocated, some organizations have distributed or dispersed teams. Distributed teams have cross-functional teams in different locations. Dispersed teams may have each team member working in a completely different location, either in an office or from home. While these arrangements are not ideal due to increased communication costs, they may still be workable.

In one large, U.S.-based financial institution there was a program with a set of teams where the team members were based on the East Coast of the United States and several locations throughout India. When the team first started, it was one large dispersed team (UX, analysts, developers,

and testers) doing a “follow the sun”[2] development practice where some working time overlapped across the team members to do warm hand-offs with the work. Team members conducted daily standups together and used webcams to include all team members. Key roles (analysts, product owners, UX designers, and development leads) in the U.S. would come in early to answer any questions from their India-based team members and help to resolve impediments.

As the product started getting larger, and more funding came through, they decided to break into five smaller teams. To do this, they decided to build colocated, distributed teams in various locations. They made the decision to build cross-functional, colocated teams in each of these locations consisting of developers and testers.

They also had a core set of analysts, based in the two U.S. locations, who worked with their U.S.-based product manager and product owners and then worked with each of the teams, respectively. Although they had some structure in place where they conducted product reviews as an entire program, most of the other activities were conducted at a team level, based on what worked best for each team, to allow them to self-organize.

4.3.5 DEDICATED TEAM MEMBERS

What happens when the team members’ time is not 100% dedicated to the team? While this condition is not ideal, unfortunately, it sometimes cannot be avoided.

The key problem with having someone invest only a capacity of 25% or 50% on the team is that they will multitask and task switch. Multitasking reduces the throughput of the team's work and impacts the team's ability to predict delivery consistently.

TIP

Multitasking slows the progress of the entire team, because team members waste time context switching and/or waiting for each other to finish other work. When people are 100% dedicated to the team, the team

has the fastest possible throughput.

People experience productivity losses somewhere between 20% and 40% when task switching. The loss increases exponentially with the number of tasks.

When a person multitasks between two projects, that person is not 50% on each project. Instead, due to the cost of task switching, the person is somewhere between 20% and 40% on each project.

People are more likely to make mistakes when they multitask. Task-switching consumes working memory and people are less likely to remember their context when they multitask.

When everyone on the team is 100% allocated to one project, they can continuously collaborate as a team, making everyone's work more effective.

CASE

Since core team members developing this practice guide cannot dedicate 100% of their capacity to the team's efforts, their throughput is substantially lower than what it might be if they could afford to collocate and invest their attention full-time to the project. However, while it is economically viable to collaborate, even if dispersed and operating at a fraction of their full capacity, it is not feasible to colocate and focus at full capacity. Therefore, the team identified their dispersion as a potential risk. The team tracks and monitors the progress of their work through the use of collaborative tools and adjusts assignments based on individual capacity accordingly.

See Table A1-2 on Project Management Process Group and Knowledge Area Mapping for more tips on teams in agile environments, specifically the processes in the Project Resource Management Knowledge Area.

TIP

Not all teams have all the roles that they need. For example, some teams

need support from database administrators or research analysts. When a team has temporarily assigned specialists, it is important to ensure that everyone has the same set of expectations. Is this specialist 100% allocated to the team and for how long? Set expectations with everyone (the specialist and the team) to clarify the level of commitment so the team can deliver. Part-time assignments create risks for the project.

4.3.6 TEAM WORKSPACES

Teams need a space in which they can work together, to understand their state as a team, and to collaborate. Some agile teams all work in one room together. Some teams have a team workspace for their standups and charts, and work on their own in cubicles or offices.

While companies are moving toward open, collaborative work environments, organizations also need to create quiet spaces for workers who need uninterrupted time to think and work. Therefore, companies are designing their offices to balance common and social areas (sometimes called “caves and common”) with quiet areas or private spaces where individuals can work without being interrupted.

When teams have geographically distributed members, the team decides how much of their workplace is virtual and how much is physical. Technology such as document sharing, video conferencing, and other virtual collaboration tools help people collaborate remotely.

Geographically distributed teams need virtual workspaces. In addition, consider getting the team together in person at regular intervals so the team can build trust and learn how to work together.

Some techniques to consider for managing communication in dispersed teams are *fishbowl windows* and *remote pairing*:

- Create a fishbowl window by setting up long-lived video conferencing links between the various locations in which the team is dispersed. People start the link at the beginning of a workday, and close it at the end. In this way, people can see and engage spontaneously with each other, reducing the collaboration lag otherwise inherent in the geographical separation.
- Set up remote pairing by using virtual conferencing tools to share

screens, including voice and video links. As long as the time zone differences are accounted for, this may prove almost as effective as face-to-face pairing.

TIP

Form teams by bringing people with different skills from different functions together. Educate managers and leaders about the agile mindset and engage them early in the agile transformation.

4.3.7 OVERCOMING ORGANIZATIONAL SILOS

The best place to start when forming agile teams is by building a foundational trust and a safe work environment to ensure that all team members have an equal voice and can be heard and considered. This, along with building the agile mindset is the underlying success factor—all other challenges and risks can be mitigated.

Often, siloed organizations create impediments for forming cross-functional agile teams. The team members needed to build the cross-functional teams typically report to different managers and have different metrics by which managers measure their performance. Managers need to focus on flow efficiency (and team-based metrics) rather than resource efficiency.

To overcome organizational silos, work with the various managers of these team members and have them dedicate the necessary individuals to the cross-functional team. This not only creates team synergy but also allows the organization to see how leveraging its people will optimize the project or product being built.

For more information about teams see Appendix X2 on Attributes that Influence Tailoring.

TIP

As an agile project leader, first focus on how you can create a team that is cross-functional and 100% dedicated to one team. Even if it means just getting key team members, such as the developers and testers, to work and

communicate together on a daily basis, that is a step in the right direction toward agility.

---

[2] A follow-the-sun development process is one where work is handed off at the end of every day from one site to the next, many time zones away in order to speed up product development.

5

IMPLEMENTING AGILE: DELIVERING IN AN AGILE ENVIRONMENT

5.1 CHARTER THE PROJECT AND THE TEAM

Every project needs a project charter so the project team knows why this project matters, where the team is headed and what the project objective is. However, the project charter itself may not be enough for the team. Agile teams require team norms and an understanding of how to work together. In that case, the team might need a team charter.

The chartering process helps the team learn how to work together and coalesce around the project.

At a minimum, for an agile project, the team needs the project vision or purpose and a clear set of working agreements. An agile project charter answers these questions:

- Why are we doing this project? This is the project vision.
- Who benefits and how? This may be part of the project vision and/or project purpose.
- What does done mean for the project? These are the project's release criteria.
- How are we going to work together? This explains the intended flow of work.

A servant leader may facilitate the chartering process. A team can coalesce by working together, and the project charter is a great way to start working. In addition, team members may want to collaborate to understand how they will

work together.

Teams do not need a formal process for chartering as long as the teams understand how to work together. Some teams benefit from a team chartering process. Here are some chartering ideas for team members to use as a basis for their social contract:

- Team values, such as sustainable pace and core hours;
- Working agreements, such as what “ready” means so the team can take in work; what “done” means so the team can judge completeness consistently; respecting the timebox; or the use of work-in-process limits;
- Ground rules, such as one person talking in a meeting; and
- Group norms, such as how the team treats meeting times.

The servant leader together with the team may decide to address other behaviors.

Remember that the team's social contract—its team charter—is how the team members interact with each other. The goal of the team charter is to create an agile environment in which team members can work to the best of their ability as a team.

5.2 COMMON AGILE PRACTICES

Sections 5.2.1 through 5.2.8 describe a few of the most common agile project practices.

5.2.1 RETROSPECTIVES

The single most important practice is the retrospective because it allows the team to learn about, improve, and adapt its process.

Retrospectives help the team learn from its previous work on the product and its process. One of the principles behind the Agile Manifesto is: “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.”

Many teams use iterations—especially 2-week iterations—because the

iteration prompts a demonstration and a retrospective at the end. However, the team does not need iterations in order to retrospect. Team members may decide to retrospect at these key times:

- When the team completes a release or ships something. It does not have to be a monumental increment. It can be any release, no matter how small.
- When more than a few weeks have passed since the previous retrospective.
- When the team appears to be stuck and completed work is not flowing through the team.
- When the team reaches any other milestone.

Teams benefit from allocating enough time to learn, either from an interim retrospective or an end-of-the-project retrospective. Teams need to learn about their work product and work process. For example, some teams have trouble finishing work. When teams plan enough time, they can structure their retrospective to gather data, process that data, and decide what to try later as an experiment.

First and foremost, a retrospective is not about blame; the retrospective is a time for the team to learn from previous work and make small improvements.

The retrospective is about looking at the qualitative (people's feelings) and quantitative (measurements) data, then using that data to find root causes, designing countermeasures, and developing action plans. The project team may end up with many action items to remove impediments.

Consider limiting the number of action items to the team's capacity to address improvement in the upcoming iteration or work period. Trying to improve too many things at once and not finishing any of them is much worse than planning to complete fewer items and successfully completing all of them. Then, when time allows, the team can work on the next improvement opportunity on the list. When the team selects the improvements, decide how to measure the outcomes. Then, in the next time period, measure the outcomes to validate success or failure of each improvement.

A facilitator from the team leads them through an activity to rank the importance of each improvement item. Once the improvement items are ranked by the team, the team chooses the appropriate number to work on for the next

iteration (or adds work to the flow if flow-based).

5.2.2 BACKLOG PREPARATION

The backlog is the ordered list of all the work, presented in story form, for a team. There is no need to create all of the stories for the entire project before the work starts—only enough to understand the first release in broad brushstrokes and then sufficient items for the next iteration.

Product owners (or a product owner value team that includes the product manager and all relevant product owners for that area of the product,) might produce a product roadmap to show the anticipated sequence of deliverables over time. The product owner replans the roadmap based on what the team produces. (See Appendix X3 on Agile Suitability Filter Tools for examples of roadmaps.)

5.2.3 BACKLOG REFINEMENT

In iteration-based agile, the product owner often works with the team to prepare some stories for the upcoming iteration during one or more sessions in the middle of the iteration. The purpose of these meetings is to refine enough stories so the team understands what the stories are and how large the stories are in relation to each other.

There is no consensus on how long the refinement should be. There is a continuum of:

- Just-in-time refinement for flow-based agile. The team takes the next card off the to-do column and discusses it.
- Many iteration-based agile teams use a timeboxed 1-hour discussion midway through a 2-week iteration. (The team selects an iteration duration that provides them frequent-enough feedback.)
- Multiple refinement discussions for iteration-based agile teams. Teams can use this when they are new to the product, the product area, or the problem domain.

TIP

Consider using impact mapping to see how the product fits together. Under normal circumstances, the product owner leads this work. A servant leader can facilitate any necessary meetings as a way of serving the project.

Refinement meetings allow the product owner to present story ideas to the team and for the team to learn about the potential challenges or problems in the stories. If the product owner is unsure of the dependencies, the product owner can request the team to spike the feature in order to understand the risks.

There are many ways for the product owner to conduct backlog preparation and refinement meetings, including for example:

- Encourage the team to work as triads of developer, tester, business analyst/product owner to discuss and write the story.
- Present the overall story concept to the team. The team discusses and refines it into as many stories as required.
- Work with the team to find various ways to explore and write the stories together, making sure all of the stories are small enough so the team can produce a steady flow of completed work. Consider becoming able to complete a story at least once a day.

Teams often have a goal of spending not more than 1 hour per week refining stories for the next batch of work. Teams want to maximize the time spent doing work as opposed to planning work. If the team needs to spend more than 1 hour per week refining stories, the product owner could be overpreparing, or the team may be lacking some critical skills needed to evaluate and refine the work.

5.2.4 DAILY STANDUPS

Teams use standups to microcommit to each other, uncover problems, and ensure the work flows smoothly through the team.

Timebox the standup to no longer than 15 minutes. The team “walks” the Kanban or task board in some way, and anyone from the team can facilitate the standup.

In iteration-based agile, everyone answers the following questions in a round-

robin fashion:

- What did I complete since the last standup?
- What am I planning to complete between now and the next standup?
- What are my impediments (or risks or problems)?

Questions like these generate answers that allow the team to self-organize and hold each other accountable for completing the work they committed to the day before and throughout the iteration.

Flow-based agile has a different approach to standups, focusing on the team's throughput. The team assesses the board from right to left. The questions are:

- What do we need to do to advance this piece of work?
- Is anyone working on anything that is not on the board?
- What do we need to finish as a team?
- Are there any bottlenecks or blockers to the flow of work?

One of the antipatterns typically seen in standups is they become status meetings. Teams who have traditionally worked in a predictive environment may tend to fall into this antipattern since they are used to providing a status.

Another antipattern typically seen in standups is that the team begins to solve problems as they become apparent. Standups are for realizing there are problems —not for solving them. Add the issues to a parking lot, and then create another meeting, which might be right after the standup, and solve problems there.

Teams run their own standups. When run well, standups can be very useful, provided the nature of the team's work requires intense collaboration. Make a conscious decision about when the team needs, or can effectively use, standups.

TIP

Encourage any team member to facilitate the standup instead of a project manager or leader to ensure it does not turn into a status meeting, but instead is used as a time for the team to self-organize and make commitments to each other.

5.2.5 DEMONSTRATIONS/REVIEWS

As the team completes the features usually in the form of user stories, the team periodically demonstrates the working product. The product owner sees the demonstration and accepts or declines stories.

In iteration-based agile, the team demonstrates all completed work items at the end of the iteration. In flow-based agile, the team demonstrates completed work when it is time to do so, usually when enough features have accumulated into a set that is coherent. Teams, including the product owner, need feedback to decide how early to ask for product feedback.

As a general guideline, demonstrate whatever the team has as a working product at least once every 2 weeks. That frequency is enough for most teams, so team members can get feedback that prevents them from heading in a wrong direction. That is also frequent enough so that the teams can keep the product development clean enough to build a complete product as often as they want or need to.

A fundamental part of what makes a project agile is the frequent delivery of a working product. A team that does not demonstrate or release cannot learn fast enough and is likely not adopting agile techniques. The team may require additional coaching to enable frequent delivery.

5.2.6 PLANNING FOR ITERATION-BASED AGILE

Each team's capacity is different. Each product owner's typical story size is different. Teams consider their story size so they do not try to commit to more stories than there is team capacity to complete within one iteration.

When people are unavailable (e.g., holidays, vacations, or anything that prevents people from participating in the next set of work), the product owner understands that the team has reduced capacity. The team will not be able to finish the same amount of work as it finished in the previous time period. When teams have a reduced capacity, they will only plan for work that meets that capacity.

Teams estimate what they can complete, which is a measure of capacity (see Section 4.10 for examples). Teams cannot predict with 100% certainty what they can deliver, as they cannot know the unexpected. When product owners make the stories smaller and teams see progress in the form of a finished product,

teams learn what they are able to do for the future.

Agile teams do not plan just once in one single chunk. Instead, agile teams plan a little, deliver, learn, and then replan a little more in an ongoing cycle.

TIP

> *Draw the team's attention to the antipattern and help the team to discover how to improve its standups.*

5.2.7 EXECUTION PRACTICES THAT HELP TEAMS DELIVER VALUE

If the team does not pay attention to quality, it will soon become impossible to release anything rapidly.

The following technical practices, many of which come from eXtreme Programming, may help the team to deliver at their maximum speed:

- **Continuous integration.** Perform frequent incorporation of work into the whole, no matter the product, and then retest to determine that the entire product still works as intended.
- **Test at all levels.** Employ system-level testing for end-to-end information and unit testing for the building blocks. In between, understand if there is a need for integration testing and where. Teams find smoke testing helpful as a first look at whether the work product is any good. Teams have found that deciding when to run regression tests and which ones helps them maintain product quality with good build performance. Agile teams have a strong preference for automated tests so they can build and maintain a momentum of delivery.
- **Acceptance Test-Driven Development (ATDD).** In ATDD, the entire team gets together and discusses the acceptance criteria for a work product. Then the team creates the tests, which allows the team to write just enough code and automated tests to meet the criteria. For non-software projects, consider how to test the work as the team completes chunks of value.

- **Test-Driven Development (TDD) and Behavior-Driven Development (BDD).** Writing automated tests before writing/creating the product actually helps people design and mistake-proof the product. For non-software projects, consider how to “test-drive” the team's designs. Hardware and mechanical projects often use simulations for interim tests of their designs.
- **Spikes (timeboxed research or experiments).** Spikes are useful for learning and may be used in circumstances such as: estimation, acceptance criteria definition, and understanding the flow of a user's action through the product. Spikes are helpful when the team needs to learn some critical technical or functional element.

5.2.8 HOW ITERATIONS AND INCREMENTS HELP DELIVER WORKING PRODUCT

Iterations help a team create a cadence for delivery and many kinds of feedback. Teams produce increments of value for delivery and feedback. The first part of this delivery is a demonstration. Teams receive feedback about how the product looks and operates through a demo. Team members retrospect to see how they can inspect and adapt their process to succeed.

Demonstrations or reviews are a necessary part of the agile project flow. Schedule the demonstration as appropriate for the team's delivery cadence.

5.3 TROUBLESHOOTING AGILE PROJECT CHALLENGES

Agile approaches were born out of the need to solve issues associated with high rates of change, uncertainty, and complexity on projects. Due to these origins, they contain a variety of tools and techniques for dealing with issues that present problems in predictive approaches. Refer to Table 5-1.

TIP

Teams should demo often for feedback and to show progress. Encourage

the PMO and other interested parties to watch demonstrations so the people deciding on the project portfolio can see the actual progress.

Table 5-1. Agile Pain Points and Troubleshooting Possibilities

Pain Point	Troubleshooting Possibilities
Unclear purpose or mission for the team	Agile chartering for purpose–vision, mission, and mission tests
Unclear working agreements for the team	Agile chartering for alignment–values, principles, and working agreements
Unclear team context	Agile chartering for context–boundaries, committed assets, and prospective analysis
Unclear requirements	Help sponsors and stakeholders craft a product vision. Consider building a product roadmap using specification by example, user story mapping, and impact mapping. Bring the team and product owner together to clarify the expectations and value of a requirement. Progressively decompose roadmap into backlog of smaller, concrete requirements.
Poor user experience	User experience design practices included in the development team involve users early and often.
Inaccurate estimation	Reduce story size by splitting stories. Use relative estimation with the entire team to estimate. Consider agile modeling or spiking to understand what the story is.
Unclear work assignments or work progress	Help the team learn that they self-manage their work. Consider kanban boards to see the flow of work. Consider a daily standup to walk the board and see what work is where.
Team struggles with obstacles	A servant leader can help clear these obstacles. If the team doesn't know the options they have, consider a coach. Sometimes, the team needs to escalate stories the team or servant leader has not been able to remove.
Work delays/overruns due to Insufficiently refined product backlog items	Product owner and team workshop stories together. Create a definition of ready for the stories. Consider splitting stories to use smaller stories.
Defects	Consider the technical practices that work for the

	environment. Some possibilities are: pair work, collective product ownership, pervasive testing (test-driven and automated testing approaches) and a robust definition of done.
Work is not complete	Team defines definition of done for stories including acceptance criteria. Also add release criteria for projects.
Technical debt (degraded code quality)	Refactoring, agile modeling, pervasive testing, automated code quality analysis, definition of done
Too much product complexity	For software and non-software encourage the team always to be thinking “What is the simplest thing that would work?” and apply the agile principle of “Simplicity--the art of maximizing the amount of work not done”. These help reduce complexity.
Slow or no improvement in the teamwork process	Capture no more than three items to improve at each retrospective. Ask the servant leader to help the team learn how to integrate those items.
Too much upfront work leading to rework	Instead of much upfront work, consider team spikes to learn. In addition, measure the WIP during the beginning of the project and see what the team's options are to deliver value instead of designs. Shorten iterations and create a robust definition of done.
False starts, wasted efforts	Ask the product owner to become an integral part of the team.
Inefficiently ordered product backlog items	Rank with value including cost of delay divided by duration (CD3) and other value models
Rush/wait uneven flow of work	Plan to the team's capacity and not more. Ask people to stop multitasking and be dedicated to one team. Ask the team to work as pairs, a swarm, or mob to even out the capabilities across the entire team.
Impossible stakeholder demands	Servant leadership to work with this stakeholder (and possibly product owner).
Unexpected or unforeseen delays	Ask the team to check in more often, use kanban boards to see the flow of work and work in progress limits to understand the impact of the demands on the team or product. Also track impediments and impediment removal on an impediment board.

Siloed teams, instead of cross-functional teams	Ask the people who are part of projects to self-organize as cross-functional teams. Use servant leadership skills to help the managers understand why agile needs cross-functional teams.

5.4 MEASUREMENTS IN AGILE PROJECTS

Transitioning to agile means using different measurements. Using agile means looking at new metrics that matter to the team and to management. These metrics matter because they focus on customer value.

One problem with status reporting is the team's ability to predict completion or to use traffic light status to describe the project. For instance, project leaders describe the project as “90% done.” At that point the team tries to integrate the pieces into a product. The team discovers missing requirements or surprises, or finds that the product doesn't integrate the way they thought it would.

The project is only partway done, and the traffic light status reporting does not reflect the real state. Too often, the project team realizes it will need just as much time to complete the remainder of the project. For too many projects, the team realizes they are—at most—10% done because of issues the team discovered.

The problem with predictive measurements is that they often do not reflect reality. It often happens that a project status light is green up until 1 month before the release date; this is sometimes referred to as a watermelon project (green on the outside, red on the inside). Oftentimes project status lights turn red with seemingly no warnings, because there is no empirical data about the project until 1 month before the release date.

Metrics for agile projects contain meaningful information that provide a historical track record, because agile projects deliver value (finished work) on a regular basis. Project teams can use such data for improved forecasts and decision making.

Surrogate measurements such as percent done are less useful than empirical measurements such as finished features. See Section 4.10 for more information on value management. Agile helps teams see problems and issues so the team can diagnose and address them.

In addition to quantitative measures, the team can consider collecting qualitative measures. Some of these qualitative measures focus on practices the

team has chosen and assess how well the team uses those practices, for example, the business satisfaction with delivered features, the morale of the team; and anything else the team wants to track as a qualitative measure.

5.4.1 AGILE TEAMS MEASURE RESULTS

Agile favors empirical and value-based measurements instead of predictive measurements. Agile measures what the team delivers, not what the team predicts it will deliver.

A team that is accustomed to having project baselines and estimates of earned value and ROI might be puzzled about working on a project and not managing to a baseline. Agile is based on working products of demonstrable value to customers.

Baselines are often an artifact of attempted prediction. In agile, the team limits its estimation to the next few weeks at most. In agile, if there is low variability in the team's work and if the team members are not multitasking, the team's capacity can become stable. This allows better prediction for the next couple of weeks.

After the team completes work in iteration or flow, the team can replan. Agile does not create the ability to do more work. However, there is evidence that the smaller the chunk of work, the more likely people are to deliver it.

Software product development, like other knowledge work, is about learning —learning while delivering value. Hardware development and mechanical development are similar in the design parts of the project. Learning takes place by experimenting, delivering small increments of value, and getting feedback on what has been accomplished thus far. Many other product development projects incorporate learning also.

Sponsors usually want to know when the project will be done. Once the team establishes a reliable velocity (average stories or story points per iteration) or the average cycle time, the team can predict how much longer the project will take.

As an example, if the team averages 50 story points per iteration, and the team estimates there are about another 500 points remaining, the team estimates it has about 10 iterations remaining. As the product owner refines

the stories remaining and as the team refines its estimates, the project estimate could go up or down, but the team can provide an estimate.

If the team averages a cycle time of three days per story and there are 30 remaining stories, the team would have 90 business days remaining, approximately 4 to 5 months.

Reflect the estimate variability with hurricane-style charts, or some other variability measure that the sponsors will understand.

Because learning is such a large part of the project, the team needs to balance uncertainty and provide value to the customers. The team plans the next small part of the project. The team reports empirical data and replans further small increments to manage the project uncertainty.

Some iteration-based projects use burndown charts to see where the project is going over time. Figure 5-1 shows an example of a burndown chart where the team planned to deliver 37 story points. Story points rate the relative work, risk, and complexity of a requirement or story. Many agile teams use story points to estimate effort. The dotted burndown line is the plan. In Figure 5-1, the team can see by Day 3 that they are at risk for that delivery.

Figure 5-1. Burndown Chart for Remaining Story Points

Some project teams prefer burnup charts. The same data used in Figure 5-1 is shown in Figure 5-2 in a burnup chart.

Figure 5-2. Burnup Chart for Showing Story Points Completed

Burnup charts show the work completed. The two charts in Figures 5-1 and 5-2 are based on the same data, but displayed in two different ways. Teams may prefer how to see their data.

When a team sees what it has not yet completed as it works through an iteration, the team may become dispirited and possibly rush to complete the work without meeting the acceptance criteria. However, the team could have any number of good reasons for not completing work as it expected. Burndowns show the effect of team members multitasking, stories that are too large, or team members out of the office.

Especially with teams new to agile, the burnup will show changes in scope during the iteration. Burnups allow teams to see what they have accomplished, which helps the team proceed to the next piece of work.

Whether teams use burndown or burnup charts, they see what they have completed as the iteration progresses. At the end of the iteration, they might base their next measure of capacity (how many stories or story points) on what they completed in this iteration. That allows the product owner along with the team to

replan what the team is more likely to succeed in delivering in the next iteration.

Velocity, the sum of the story point sizes for the features actually completed in this iteration, allows the team to plan its next capacity more accurately by looking at its historical performance.

Flow-based agile teams use different measurements: lead time (the total time it takes to deliver an item, measured from the time it is added to the board to the moment it is completed), cycle time (the time required to process an item), and response time (the time that an item waits until work starts). Teams measure cycle time to see bottlenecks and delays, not necessarily inside the team.

TIP

Teams might discover it can take four to eight iterations to achieve a stable velocity. The teams need the feedback from each iteration to learn about how they work and how to improve.

Figure 5-3. Example of a Kanban Board

Lead time is useful to understand cycle time from the first look at a particular feature to the length of time it took to release it to the customer. The work in progress (WIP) limits at the top of each column, shown in boxes here, allows the team to see how to pull work across the board. When the team has met its WIP limits, the team cannot pull work from the left into the next column. Instead, the team works from the right-most full column and asks, “What do we do as a team to move this work into the next column?”

Each feature is unique, so its cycle time is unique. However, a product owner might notice that smaller features have smaller cycle times. The product owner wants to see throughput, so the product owner creates smaller features or works with the team to do so.

Burnups, burndowns (capacity measures) and lead time, and cycle time (predictability measures) are useful for in-the-moment measurements. They help a team understand how much more work they have and whether the team might finish on time.

Measuring story points is not the same as measuring completed stories or features. Some teams attempt to measure story points without completing the actual feature or story. When teams measure only story points, they measure capacity, not finished work, which violates the principle of "the primary measure of progress is working software" (or, other product if not software).

Each team has its own capacity. When a team uses story points, be aware that the number of story points a team can complete in a given time is unique to that team.

TIP

> *When teams depend on external people or groups, measure cycle time to see how long it takes for the team to complete the work. Measure lead time to see the external dependencies after the team completes its work. Teams can also measure the reaction time, the time from ready to the first column, to see how long it takes them—on average—to respond to new requests.*

When teams provide their own units of measure, teams are better able to assess and estimate and deliver their work. The downside of relative estimation is that there is no way to compare teams or add velocity across teams.

The team can measure completed work in a feature burnup/burndown chart and in a product backlog burnup chart. These charts provide trends of completion over time, as shown in Figure 5-4.

Figure 5-4. Feature Chart

Feature burnup/burndown charts may show that requirements grew during the project. The features complete line shows that the team completes features at a regular pace. The total features line shows how the project's total features changed over time. The features remaining burndown line shows that the rate of feature completion varies. Every time features are added to the project, the burndown line changes.

Earned value in agile is based on finished features, as shown in Figure 5-5. The product backlog burnup chart shows completed work compared to total expected work at interval milestones or iterations.

Figure 5-5. Product Backlog Burnup Chart

A team can only finish one story at a time. To complete a large feature that contains several stories, the team will have remaining stories to complete and may not complete that entire feature until several more time periods have passed. The team can show its completed value with a product backlog burnup chart as shown in Figure 5-5.

If a team needs to measure earned value, it can consider using this burnup chart in Figure 5-6 as an example: Note that the left Y axis represents story points as scope, and the right Y axis represents the project spend.

Figure 5-6. Earned Value in an Agile Context

Traditional EVM metrics like schedule performance index (SPI) and cost performance index (CPI) can be easily translated into agile terms. For example, if the team planned to complete 30 story points in an iteration, but only completed 25 then the SPI is 25/30 or 0.83 (the team is working at only 83% of the rate planned). Likewise, CPI is the earned value (completed features value) to date divided by the actual costs to date or, as shown in Figure 5-6, $2.2M / $2.8M = 0.79. This means a result of only 79 cents on the dollar compared to plan (but of course this assumes that the prediction is still correct.)

A cumulative flow diagram, illustrated in Figure 5-7, shows the work in progress across a board. If a team has many stories waiting for test, the testing band will swell. Work accumulation can be seen at a glance.

Teams have trouble with accumulating work: the team has work in progress instead of work completed. When teams have a lot of work in progress, they delay their overall feature delivery. The longer it takes for a team to deliver, the more pressure a team will have for yet more features in the same period of time.

Figure 5-7. Cumulative Flow Diagram of Completed Features

Adapt this cumulative flow to the project task board.

6

ORGANIZATIONAL CONSIDERATIONS FOR PROJECT AGILITY

Every project exists in an organizational context. Cultures, structures, and policies can influence both the direction and the outcome of any project. These dynamics can challenge project leaders.

While project leaders may not have the ability to change organizational dynamics as they see fit, they are expected to navigate those dynamics skillfully.

This section explores the way the organization and in some circumstances, the project context, influences projects. Leaders can explore options for change, to increase project success.

> Project agility is more effective and sustained as the organization adjusts to support it.

6.1 ORGANIZATIONAL CHANGE MANAGEMENT

Organizational change management covers the skills and techniques for influencing changes that support agility.

The PMI publication, *Managing Change in Organizations: A Practice Guide* [2], describes a comprehensive and holistic approach for successfully introducing meaningful change. The recommendations offered there include:

- Models for describing change dynamics,

- Framework for achieving change, and
- Application of change management practices at the project, program, and portfolio levels.

Sections 6.1.1 and 6.1.2 explore the considerations of change management specific to an agile context.

Figure 6-1 shows the relationship between these two topics.

Figure 6-1. The Relationship Between Change Management and Agile Approaches

6.1.1 DRIVERS FOR CHANGE MANAGEMENT

All projects are about change. However, there are two key factors that further motivate the use of change management practices in an agile context:

- **Changes associated with accelerated delivery.** Agile approaches emphasize delivering project outputs early and often. However, the

receiving organization may not be fully prepared to incorporate those outputs at an increased pace. Accelerating delivery will test the organization's ability to accommodate that delivery. Successfully discovering and delivering a project's features is not enough. If the organization resists the project's output, then the targeted return on investment is delayed. Customer acceptance of and alignment with project outputs becomes even more prevalent in an agile environment.

- **Changes associated with agile approaches.** Organizations just beginning to use agile approaches also experience high degrees of change. Higher degrees of collaboration may require more frequent handoffs between teams, departments, or vendors. Decomposing work into iterative prototypes involves rework that could be viewed negatively. Leaders should consider change management techniques to address the hurdles of transitioning to the use of agile approaches.

6.1.2 READINESS FOR CHANGE

Organizations beginning to use agile approaches should understand the relative compatibility of those methods with their current approaches. Some organizations will have characteristics that more easily support agile principles of cross-department collaboration, continuous learning, and evolving internal processes. Examples of these change-friendly characteristics include:

- Executive management's willingness to change;
- Organization's willingness to shift the way it views, reviews, and assesses employees;
- Centralization or decentralization of project, program, and portfolio management functions;
- Focus on short-term budgeting and metrics versus long-term goals; and
- Talent management maturity and capabilities.

Conversely, there are other institutional characteristics that may be roadblocks to achieving the changes associated with organizational agility. Examples of these include:

- Work is decomposed into departmental silos, creating dependencies that prevent accelerated delivery instead of building cross-functional

teams with guidance from centers of competencies.

- Procurement strategies are based on short-term pricing strategies, rather than long-term competencies.
- Leaders are rewarded for local efficiencies rather than end-to-end flow of project delivery or optimizing the whole (in regard to the organization).
- Employees are specialized contributors with limited tools or incentives to diversify their skills instead of building T-shaped specialists.
- Decentralized portfolios pull employees simultaneously onto too many projects at once instead of keeping them focused on one project at a time.

The degree to which an organization is willing to review and modify these practices will determine how quickly and effectively agile approaches can be adopted. However, in response to these organizational impediments to agility, project leaders can try various approaches to accelerate a cultural compatibility for:

- Visible and active executive sponsorship,
- Change management practices, including communication and coaching,
- Progressively pacing the adoption of agile practices on a project-by-project basis
- Incremental introduction of agile practices to the team; and
- Leading by example by using agile techniques and practices where possible.

6.2 ORGANIZATIONAL CULTURE

An organization's culture is its DNA—its core identity. Culture will always influence the use of agile approaches. Organizational culture runs along a continuum, from highly predictive plans to lean startup where everything is an experiment. Although agile approaches fit well with the lean startup culture, a highly predictive organization can encourage empirical measurements, small experiments, and learning so they can move toward agility.

> “Culture eats strategy for breakfast”—Peter Drucker
>
> This statement stresses the importance of people's commitment and passion for a cause. No matter what strategy or plan you implement with your team, its success is going to be governed by the people implementing the plan. If the people who are driving the strategy aren't passionate about the change, or worse, are apathetic about their job and their organization, then you stand little chance of implementing the change.

6.2.1 CREATING AN ENVIRONMENT OF SAFETY

Organizational culture is difficult to change, but the most important cultural norm in an organization willing to try any new method or technique is enabling a safe work environment.

Only in a safe, honest, and transparent environment can team members and leaders truly reflect on their successes to ensure their projects continue to advance, or apply lessons learned on failed projects so they do not fall back into the same patterns.

6.2.2 ASSESSING CULTURE

Every project finds itself in tension with competing aspirations. How can the team go fast without compromising quality? How can the team preserve flexibility while also hitting a firm date? Most importantly, how does the team satisfy and meet the requirements of the customer?

Project leaders may feel their job is to meet every expectation of every stakeholder; but, when compelled to make a choice, there is often a priority depending on the culture and requirements of the organization's business environment. For example, a mobile telecom project has a greater bias for speed, where a government program may have a greater bias for generalization and stability.

To navigate these dynamics, project leaders should take the time to assess where emphasis is most often applied in the organization. Figure 6-2 illustrates what an assessment might look like. In this example, a project leader initiates a

conversation about organizational priorities with stakeholders, team members, and senior management. Those priorities are then recorded as positions on a sliding scale between two extremes. The results are then used to find agile techniques that best fit with those priorities.

Figure 6-2. Example of Assessing Organizational Culture

Several models exist for assessing such dynamics; however, the model or method used is not that important. It is more critical that project leaders invest the effort to understand the forces that shape their context. Understanding the organization and the industry requirements that an organization needs to satisfy allows for choosing the right conversations, the right tradeoffs, and, especially, the right techniques.

6.3 PROCUREMENT AND CONTRACTS

As mentioned earlier in this practice guide, the Agile Manifesto values "customer collaboration over contract negotiation." Many project failures stem from breakdowns in the customer–supplier relationship. Projects incur more risk when those involved in the contract take the perspective of winners vs. losers. A collaborative approach is one that pursues a shared-risk-reward relationship,

where all sides win. Some contracting techniques that can formalize this dynamic include the following:

- **Multi-tiered structure.** Rather than formalizing an entire contracting relationship in a single document, project parties can achieve more flexibility by describing different aspects in different documents. Mostly fixed items (e.g., warranties, arbitration) can be locked in a master agreement. Meanwhile, all parties list other items subject to change (e.g., services rates, product descriptions) in a schedule of services. The contract can reference them in the master services agreement. Finally, more dynamic items such as scope, schedule, and budget can be formalized in a lightweight statement of work. Isolating the more changing elements of a contract into a single document simplifies modifications and thus flexibility.
- **Emphasize value delivered.** Many vendor relationships are governed by fixed milestones or “phase gates” focused on intermediate artifacts, rather than a full deliverable of incremental business value. Often, these controls limit the use of feedback to improve the product. Instead, milestones and payment terms can be structured based on value-driven deliverables in order to enhance the project's agility.
- **Fixed-price increments.** Rather than lock an entire project scope and budget into a single agreement, a project can decompose the scope into fixed-price microdeliverables, such as user stories. For the customer, this gives more control over how the money is spent. For the supplier, it limits the financial risk of over-commitment to a single feature or deliverable.
- **Not-to-exceed time and materials.** Customers incur unwanted risk from a traditional time and materials approach. One alternative is to limit the overall budget to a fixed amount. This allows the customer to incorporate new ideas and innovations into the project not originally planned. When customers want to incorporate new ideas, they will have to manage to a given capacity, replacing original work with new work. Work should be closely monitored as hours allocated reach their limit. Also, additional contingency hours could be planned into the maximum budget if considered helpful.
- **Graduated time and materials.** Another alternative is a shared financial risk approach. In agile, the quality criteria are part of what

done means. Therefore, the supplier can be rewarded with a higher hourly rate when delivery is earlier than the contracted deadline. Conversely, the supplier would suffer a rate reduction for late delivery.

- **Early cancellation option.** When an agile supplier delivers sufficient value with only half of the scope completed, the customer should not be bound to pay the remaining half if the customer no longer needs it. Instead, a contract can offer the customer to buy the remainder of the project for a cancellation fee. The customer limits budget exposure and the supplier earns positive revenue for services no longer required.
- **Dynamic scope option.** For those contracts with a fixed budget, a supplier may offer the customer the option to vary the project scope at specified points in the project. The customer can adjust features to fit the capacity. Then the customer can leverage innovation opportunities, while limiting the supplier's risk of over commitment.
- **Team augmentation.** Arguably the most collaborative contracting approach is to embed the supplier's services directly into the customer organization. Funding teams instead of a specific scope preserves the customer's strategic discretion on what work should actually be done.
- **Favor full-service suppliers.** In order to diversify risk, customers may seek a multisupplier strategy. However, the temptation will be to contract the work such that each supplier does only one thing, which creates a web of dependencies before any usable service or product emerges. Instead, place an emphasis on engagements that deliver full value (as in the idea of completed independent feature sets).

TIP

Culture versus Structure

Some people insist new organizational structures be installed before any cultural shift can begin. Others maintain the opposite—those new organizational structures are only superficial adjustments until the collective culture moves in a meaningful direction. In reality, one cannot progress without the other. Project leaders wanting to achieve agility should consider the current and future states of both of these aspects in

their organization.

It is possible to create agile contracts. Agile is built on a synergy of collaboration and trust. The supplier can help by delivering value early and often. The customer can help by providing timely feedback.

6.4 BUSINESS PRACTICES

The willingness and ability to create new competences within an organization when the need arises is a mark of organizational agility. These do not have to be earth-shattering changes and could be less disruptive in an organization that is focused on agility and the results it provides. Transparency and open collaboration are absolutely key.

As cross-functional teams deliver value, the teams and individuals might encounter problems with various support functions in the organization.

As team delivers value on a regular basis, finance departments may have the opportunity to capitalize the product differently. If the team has contracts with other organizations, procurement departments may need to change those contracts to help the other organizations deliver value frequently and synchronize with the team.

Once teams start to work in a cohesive and cooperative manner, they will challenge internal management policies. Human resources may notice individual incentives make less sense, and managers may struggle with the performance appraisals of self-organizing employees. In each case, these are opportunities to review the degree to which existing practices support agile ways of working.

As organizations progress to greater agility, there will be obvious needs for additional business units to change the way they interact and perform their responsibilities. The changes that have benefited other areas of the organization should now be embraced so the effectiveness of the entire organization can be realized.

6.5 MULTITEAM COORDINATION AND DEPENDENCIES (SCALING)

Many projects incur dependencies, even when they are not managed within a given program. For this reason, it is necessary to have an understanding of how agile works within an existing program and portfolio management context.

6.5.1 FRAMEWORKS

The guidance of the most widespread agile methods such as Scrum and eXtreme Programming focus on the activities of a single, small, usually colocated, cross-functional team. While this is very useful for efforts that require a single team, it may provide insufficient guidance for initiatives that require the collaboration of multiple agile teams in a program or portfolio.

A range of frameworks (such as the Scaled Agile Framework, Large Scale Scrum, and Disciplined Agile) and approaches (e.g., Scrum of Scrums) have emerged to cater to just such circumstances. More details on these can be found in Annex A3.

6.5.2 CONSIDERATIONS

There is more than one way to scale work. The team might need to scale the work of several agile projects into a single agile program. Alternatively, the organization can design a structure that supports agile approaches across the entire portfolio.

For example, it is helpful to start small and learn as rapidly as possible what works well in the organizational context. Teams can achieve successful outcomes even when everything is not completely transformed into an agile approach.

Regardless of the approach, a critical success factor is the healthy agile team. If using an agile approach for a single team is not successful, do not try to scale up to using it more broadly; instead, address the organizational impediments that prevent teams from working in an agile way.

The goal of large-scale agile projects is to coordinate the efforts of different teams to bring value to customers. There is more than one way to do that. Teams may use a formal framework or apply agile thinking to adjust existing program management practices.

6.6 AGILE AND THE PROJECT MANAGEMENT

6.6 AGILE AND THE PROJECT MANAGEMENT OFFICE (PMO)

The PMO exists to shepherd business value throughout the organization. It might do this by helping projects achieve their goals. Sometimes, the PMO educates teams (or arranges for training) and supports projects. Sometimes, the PMO advises management about the relative business value for a given project or set of projects.

Because agile creates cultural change, over time, the organization might need to change, including the PMO. For example, managers make decisions about which projects to fund and when, and teams decide what they need for training or advice.

6.6.1 AN AGILE PMO IS VALUE-DRIVEN

Any project should deliver the right value, to the right audience, at the right time. The PMO's objective is to facilitate and enable this goal. An agile-based PMO approach is based on a customer-collaboration mindset and is present in all PMO programs. In many cases, this means the PMO operates as if it were a consulting business, tailoring its efforts to meet specific needs requested by a given project. Some projects may need tools and templates, while others may benefit from executive coaching. The PMO should strive to deliver what is needed and keep the pulse on its customers to ensure that it knows and is able to adapt to their needs. This intrapreneur approach focuses on the PMO activities that are perceived as the most valuable to the projects it supports.

6.6.2 AN AGILE PMO IS INVITATION-ORIENTED

In order to accelerate progress on a value-based charter, a PMO may be tempted to mandate certain solutions or approaches, for example, to make everyone do it the same way to get some quick wins. However, a more deliberate perspective incorporates the desire for employee engagement. This is achieved by inviting only those interested to engage with PMO services. Higher engagement with PMO practices makes it easier for those practices to be “sticky.” If the PMO is delivering value to its clients, it is more likely that clients will request its services and adopt its practices.

6.6.3 AN AGILE PMO IS MULTIDISCIPLINARY

In order to support project-specific needs, the PMO needs to be conversant in several competencies beyond project management itself, because different projects require distinct capabilities. For instance, one project may need organizational design to address staffing challenges while another may require organizational change management techniques for stakeholder engagement or unique business models to support customer goals.

Some organizations have been transforming their PMOs into agile centers of excellence that provide such services as:

- **Developing and implementing standards.** Provide templates for user stories, test cases, cumulative flow diagrams, etc. Provide agile tools and educate supporting groups on iterative development concepts.
- **Developing personnel through training and mentoring.** Coordinate agile training courses, coaches, and mentors to help people transition to an agile mindset and upgrade their skills. Encourage and support people to attend local agile events.
- **Multiproject management.** Coordinate between agile teams by communicating between projects. Consider sharing items such as progress, issues, and retrospective findings and improvement experiments. Help manage major customer releases at the program level and investment themes at the portfolio level using an appropriate framework.
- **Facilitating organizational learning.** Gather project velocity profiles and capture, store, and index retrospective findings.
- **Managing stakeholders.** Provide product owner training, guidance on acceptance testing, and how to evaluate and give feedback on systems. Champion the importance of subject matter experts (SMEs) to projects.
- **Recruiting, selecting, and evaluating team leaders.** Develop guidelines for interviewing agile practitioners.
- **Executing specialized tasks for projects.** Train and provide retrospective facilitators, create agreements with agile project troubleshooters, and provide mentors and coaches.

6.7 ORGANIZATIONAL STRUCTURE

The structure of an organization strongly influences its ability to pivot to new information or shifting market needs. Here is a listing of key characteristics:

- **Geography.** Geographically distributed and dispersed project organizations may find several challenges impeding their work on any project. Project leaders and regional managers may have alternative or even competing goals. Additionally, cultural differences, language barriers, and lower visibility can slow down productivity. Fortunately, the use of agile approaches can encourage more collaboration and confidence than would otherwise exist. Project leaders in these contexts should encourage dialog at the team and executive level to tailor techniques for the context and to manage expectations about the effort required to do so.
- **Functionalized structures.** Some organizations are structured on a spectrum ranging from highly projectized to matrixed to highly functionalized. Projects with highly functionalized structures may find general resistance to collaboration across its organization.
- **Size of project deliverable.** Reducing the size of a project deliverable will motivate more frequent handoffs across departments, and thus more frequent interactions and a faster flow of value across the organization.
- **Allocation of people to projects.** Another approach is to ask for a single person from each department to be temporarily, yet fully allocated, to the highest priority project.
- **Procurement-heavy organizations.** Some organizations choose to implement projects primarily through vendors. Although project goals may be clear, vendors have a responsibility to look after their own financial viability. Moreover, once vendors complete their obligations and leave the engagement, the associated project knowledge goes with them. This limits the internal competencies needed for sustained flexibility and speed. Agile techniques such as retrospectives and follow up on possible improvement areas when the vendor is still engaged can help mitigate loss of product knowledge.

6.8 EVOLVING THE ORGANIZATION

When addressing an individual challenge area or implementing a new hybrid or agile approach, it is recommended to undertake the work incrementally. A common practice is to treat the change process as an agile project with its own backlog of changes that could be introduced and prioritized by the team, based on perceived value or other considerations. Each of the changes can be treated as an experiment, which is tested for a short period of time to determine suitability as-is or the need for further refinement/consideration.

Use kanban boards to track progress, showing the new approaches already in use as "done," those being tried as "in progress," and those still waiting to be introduced as "to do." See Figure 6-3 for the initial board with a ranked backlog. Figure 6-4 shows an example of what a board might have as work progresses.

Figure 6-3. Initial Ranked Backlog for Changes

Figure 6-4. Using Backlogs and Kanban Boards to Organize and Track Change Work

Using these tools to organize and manage the change implementation provides visibility into progress and also models the approaches being implemented. Rolling out changes in a transparent and appealing way improves the likelihood of their success.

7

A CALL TO ACTION

The adoption of agile and its approaches for managing projects has increased dramatically since the Agile Manifesto was first published in 2001. Adoption and the desire to operate with an agile mindset is no longer limited to a certain sized organization or those specializing only in information technology. The mindset applies universally and the approaches are successful in many settings.

Today, the demand for "being agile" is higher than ever. The debate over the best path to agility continues to keep the conversation and innovation evolving. One truth remains constant—inspection, adaptation, and transparency are critical to successfully delivering value.

You may not see everything you expected to see in this practice guide. Our core team realizes you may disagree with some elements or approaches we did choose to present—and passionately so. We call on your passion to continue the conversation and improve the next iteration of this practice guide. This is your journey—learn, experiment, gain feedback, and experiment again. Then help us retrospect; give us feedback on the guidance and contribute to future editions of this practice guide. After all, inspection without adaptation is wasted effort.

Lastly, we want to encourage you to be engaged in the broader communities of project management and agile to further conversations on these topics. Look for representatives from both PMI and Agile Alliance® at conferences and meetings and engage them in discussion. Use social media and blog your thoughts and opinions.

You can provide feedback and engage in conversation regarding the contents of this practice guide at the blog called "Agile in Practice" at https://www.projectmanagement.com/blogs/347350/Agile-in-Practice.

ANNEX A1
PMBOK® GUIDE MAPPING

Table A1-1 illustrates the mapping of Project Management Process Groups to the Knowledge Areas defined in the *PMBOK® Guide* – Sixth Edition.

This annex describes how hybrid and agile approaches address the attributes described in the *PMBOK® Guide* Knowledge Areas (see Table A1-2). It covers what stays the same and what may be different along with some guidelines to consider for increasing the likelihood of success.

Table A1-1. Project Management Process Group and Knowledge Area Mapping

Knowledge Areas	Project Management Process Groups				
	Initiating Process Group	Planning Process Group	Executing Process Group	Monitoring and Controlling Process Group	Closing Process Group
4. Project Integration Management	4.1 Develop Project Charter	4.2 Develop Project Management Plan	4.3 Direct and Manage Project Work 4.4 Manage Project Knowledge	4.5 Monitor and Control Project Work 4.6 Perform Integrated Change Control	4.7 Close Project or Phase
5. Project Scope Management		5.1 Plan Scope Management 5.2 Collect Requirements 5.3 Define Scope 5.4 Create WBS		5.5 Validate Scope 5.6 Control Scope	
6. Project Schedule Management		6.1 Plan Schedule Management 6.2 Define Activities 6.3 Sequence Activities 6.4 Estimate Activity Durations 6.5 Develop Schedule		6.6 Control Schedule	
7. Project Cost Management		7.1 Plan Cost Management 7.2 Estimate Costs 7.3 Determine Budget		7.4 Control Costs	
8. Project Quality Management		8.1 Plan Quality Management	8.2 Manage Quality	8.3 Control Quality	
9. Project Resource Management		9.1 Plan Resource Management 9.2 Estimate Activity Resources	9.3 Acquire Resources 9.4 Develop Team 9.5 Manage Team	9.6 Control Resources	
10. Project Communications Management		10.1 Plan Communications Management	10.2 Manage Communications	10.3 Monitor Communications	
11. Project Risk Management		11.1 Plan Risk Management 11.2 Identify Risks 11.3 Perform Qualitative Risk Analysis 11.4 Perform Quantitative Risk Analysis 11.5 Plan Risk Responses	11.6 Implement Risk Responses	11.7 Monitor Risks	
12. Project Procurement Management		12.1 Plan Procurement Management	12.2 Conduct Procurements	12.3 Control Procurements	
13. Project Stakeholder Management	13.1 Identify Stakeholders	13.2 Plan Stakeholder Engagement	13.3 Manage Stakeholder Engagement	13.4 Monitor Stakeholder Engagement	

Table A1-2. Application of Agile in *PMBOK® Guide* Knowledge Areas

PMBOK® Guide Knowledge Area	Application in an Agile Work Process
Section 4 **Project Integration Management**	Iterative and agile approaches promote the engagement of team members as local domain experts in integration management. The team members determine how plans and components should integrate.
	The expectations of the project manager as noted in the *Key Concepts for Integration Management* sections in the *PMBOK® Guide* do not change in an adaptive environment, but control of the detailed product planning and delivery is delegated to the team. The project manager's focus is on

	building a collaborative decision-making environment and ensuring the team has the ability to respond to changes. This collaborative approach can be further enhanced when team members possess a broad skill base rather than a narrow specialization.
Section 5 **Project Scope Management**	In projects with evolving requirements, high risk, or significant uncertainty, the scope is often not understood at the beginning of the project or it evolves during the project. Agile methods deliberately spend less time trying to define and agree on scope in the early stage of the project and spend more time establishing the process for its ongoing discovery and refinement. Many environments with emerging requirements find that there is often a gap between the real business requirements and the business requirements that were originally stated. Therefore, agile methods purposefully build and review prototypes and release versions in order to refine the requirements. As a result, scope is defined and redefined throughout the project. In agile approaches, the requirements constitute the backlog.
Section 6 **Project Schedule Management**	Adaptive approaches use short cycles to undertake work, review the results, and adapt as necessary. These cycles provide rapid feedback on the approaches and suitability of deliverables, and generally manifest as iterative scheduling and on-demand, pull-based scheduling, as discussed in the Key Trends and Emerging Practices section for Project Schedule Management in the *PMBOK® Guide*.
	In large organizations, there may be a mixture of small projects and large initiatives requiring long-term roadmaps to manage the development of these programs using scaling factors (e.g., team size, geographical distribution, regulatory compliance, organizational complexity, and technical complexity). To address the full delivery life cycle for larger, enterprise-wide systems, a range of techniques utilizing a predictive approach, adaptive approach, or a hybrid of both, may need to be adopted. The organization may need to combine practices from several core methods, or adopt a method that has already done so, and adopt a few principles and practices of more traditional techniques.
	The role of the project manager does not change based on managing projects using a predictive development life cycle or managing projects in adaptive environments. However, to be successful in using adaptive approaches, the project manager will need to be familiar with the tools and techniques to understand how to apply them effectively.
Section 7	Projects with high degrees of uncertainty or those where the scope is not yet fully defined may not benefit from detailed

Project Cost Management	scope is not yet fully defined may not benefit from detailed cost calculations due to frequent changes. Instead, lightweight estimation methods can be used to generate a fast, high-level forecast of project labor costs, which can then be easily adjusted as changes arise. Detailed estimates are reserved for short-term planning horizons in a just-in-time fashion.
	In cases where high-variability projects are also subject to strict budgets, the scope and schedule are more often adjusted to stay within cost constraints.
Section 8 **Project Quality Management**	In order to navigate changes, agile methods call for frequent quality and review steps built in throughout the project rather than toward the end of the project.
	Recurring retrospectives regularly check on the effectiveness of the quality processes. They look for the root cause of issues then suggest trials of new approaches to improve quality. Subsequent retrospectives evaluate any trial processes to determine if they are working and should be continued or new adjusting or should be dropped from use.
	In order to facilitate frequent, incremental delivery, agile methods focus on small batches of work, incorporating as many elements of project deliverables as possible. Small batch systems aim to uncover inconsistencies and quality issues earlier in the project life cycle when the overall costs of change are lower.
Section 9 **Project Resource Management**	Projects with high variability benefit from team structures that maximize focus and collaboration, such as self-organizing teams with generalizing specialists.
	Collaboration is intended to boost productivity and facilitate innovative problem solving. Collaborative teams may facilitate accelerated integration of distinct work activities, improve communication, increase knowledge sharing, and provide flexibility of work assignments in addition to other advantages.
	Although the benefits of collaboration also apply to other project environments, collaborative teams are often critical to the success of projects with a high degree of variability and rapid changes, because there is less time for centralized tasking and decision making.
	Planning for physical and human resources is much less predictable in projects with high variability. In these environments, agreements for fast supply and lean methods are critical to controlling costs and achieving the schedule.
Section 10	Project environments subject to various elements of ambiguity and change have an inherent need to

Project Communications Management	ambiguity and change have an inherent need to communicate evolving and emerging details more frequently and quickly. This motivates streamlining team member access to information, frequent team checkpoints, and colocating team members as much as possible.
	In addition, posting project artifacts in a transparent fashion, and holding regular stakeholder reviews are intended to promote communication with management and stakeholders.
Section 11 **Project Risk Management**	High-variability environments, by definition, incur more uncertainty and risk. To address this, projects managed using adaptive approaches make use of frequent reviews of incremental work products and cross-functional project teams to accelerate knowledge sharing and ensure that risk is understood and managed. Risk is considered when selecting the content of each iteration, and risks will also be identified, analyzed, and managed during each iteration.
	Additionally, the requirements are kept as a living document that is updated regularly, and work may be reprioritized as the project progresses, based on an improved understanding of current risk exposure.
Section 12 **Project Procurement Management**	In agile environments, specific sellers may be used to extend the team. This collaborative working relationship can lead to a shared risk procurement model where both the buyer and the seller share in the risk and rewards associated with a project.
	Larger projects may use an adaptive approach for some deliverables and a more stable approach for other parts. In these cases, a governing agreement such as a master services agreement (MSA) may be used for the overall engagement, with the adaptive work being placed in an appendix or supplement. This allows changes to occur on the adaptive scope without impacting the overall contract.
Section 13 **Project Stakeholder Management**	Projects experiencing a high degree of change require active engagement and participation with project stakeholders. To facilitate timely, productive discussion and decision making, adaptive teams engage with stakeholders directly rather than going through layers of management. Often the client, user, and developer exchange information in a dynamic co-creative process that leads to more stakeholder involvement and higher satisfaction. Regular interactions with the stakeholder community throughout the project mitigate risk, build trust, and support adjustments earlier in the project cycle, thus reducing costs and increasing the likelihood of success for the project.
	In order to accelerate the sharing of information within and

	across the organization, agile methods promote aggressive transparency. The intent of inviting any stakeholders to project meetings and reviews or posting project artifacts in public spaces is to surface as quickly as possible any misalignment, dependency, or other issue related to the changing project.

ANNEX A2
AGILE MANIFESTO MAPPING

This annex describes how the elements of the Agile Manifesto are covered in the *Agile Practice Guide*.

Table A2-1. Agile Manifesto Values Covered in the *Agile Practice Guide*

Value	*Agile Practice Guide* Coverage by Section and Title
Individuals and interactions over processes and tools	4.2 Servant Leadership Empowers the Team 4.3 Team Composition 5.1 Charter the Project and the Team 5.2.4 Daily Standups 6.2 Organizational Culture
Working software over comprehensive documentation	5.2.2 Backlog Preparation 5.2.3 Backlog Refinement 5.2.5 Demonstrations/Reviews 5.2.7 Execution Practices that Help Teams Deliver Value
Customer collaboration over contract negotiation	4.3 Team Composition 5.4 Measurements in Agile Projects 6.2 Organizational Culture 6.3 Procurement and Contracts 6.7 Organizational Structure
Responding to change over following a plan	5.2.1 Retrospectives 5.2.3 Backlog Refinement 5.2.5 Demonstrations/Reviews

Table A2-2. *Agile Practice Guide* Mapping of Principles Behind the Agile Manifesto

Principle	Agile Practice Guide Coverage
Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	3.1 Characteristics of Project Life Cycles 5.2.7 Execution Practices that Help Teams Deliver Value
Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	5.2.3 Backlog Refinement
Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.	5.2 Common Agile Practices
Business people and developers must work together daily throughout the project.	4.2 Servant Leadership Empowers the Team 5.2.2 Backlog Preparation 5.2.3 Backlog Refinement
Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	4.3 Team Composition 5.1 Charter the Project and the Team 5.2.1 Retrospectives
The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	4.3.4 Team Structures 5.2.4 Daily Standups
Working software is the primary measure of progress.	5.2.7 Execution Practices that Help Teams Deliver Value 5.2.8 How Iterations and Increments Help Delivery Working Product
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	5.1 Charter the Project and the Team
Continuous attention to technical excellence and good design enhances agility.	5.2 Common Agile Practices
Simplicity–the art of maximizing the amount of work not done is essential.	5.2.2 Backlog Preparation 5.2.3 Backlog Refinement
The best architectures, requirements, and designs emerge from self-organizing teams.	4.3 Team Composition
At regular intervals, the team reflects	

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.	5.2.1 Retrospectives

ANNEX A3
OVERVIEW OF AGILE AND LEAN FRAMEWORKS

This annex describes some of the commonly used agile approaches. These approaches can be used as is or combined to adapt to what works best for a given environment or situation. It is not necessary to use any of these; an agile approach can be developed from scratch as long as it adheres to the mindset, values, and principles of the Agile Manifesto. If the agile principles are followed to deliver value at a sustainable pace, and the developed approach promotes collaboration with the customer, a specific approach is not required. A link to more information regarding each approach can be found in the Bibliography section of this guide.

A3.1 SELECTION CRITERIA FOR THE *AGILE PRACTICE GUIDE*

There are too many agile approaches and techniques to be explicitly included in this practice guide. Figure A3-1 depicts a sample of agile approaches based on their depth of guidance and breadth of their life cycles. The specific approaches selected for discussion are popular examples that are:

- **Designed for holistic use.** Some agile approaches are centered on a single project activity, such as estimation or reflecting. The listed examples include only the more holistic agile frameworks. Some are more full-featured than others, but all of the selected approaches are those intended to guide a broad set of project activities.

- **Formalized for common use.** Some agile frameworks are proprietary in nature and designed for specific use by a single organization or within a single context. The frameworks described in Sections A3.2 through A3.14 focus on those intended for common use in a variety of contexts.
- **Popular in modern use.** Some agile frameworks are holistically designed and well formalized, but are simply not commonly being used in most projects or organizations. The agile frameworks described in this annex have been adopted by a significant number of industries, as measured by a collection of recent industry surveys.

Figure A3-1. Agile Approaches Plotted by Breadth and Detail

A3.2 SCRUM

Scrum is a single-team process framework used to manage product development. The framework consists of Scrum roles, events, artifacts, and

rules, and uses an iterative approach to deliver working product. Scrum is run on timeboxes of 1 month or less with consistent durations called sprints where a potentially releasable increment of product is produced. Table A3-1 lists Scrum events and artifacts utilized for project execution.

The Scrum team consists of a product owner, development team, and scrum master.

- The product owner is responsible for maximizing the value of the product.
- The development team is a cross-functional, self-organizing team consisting of team members who have everything they need within the team to deliver working product without depending on others outside of the team.
- The scrum master is responsible for ensuring the Scrum process is upheld and works to ensure the Scrum team adheres to the practices and rules as well as coaches the team on removing impediments.

Table A3-1. Scrum Events and Artifacts

Events	Artifacts
Sprint	Product backlog
Sprint planning	Sprint backlog
Daily scrum	Increments
Sprint review	
Sprint retrospective	

A3.3 EXTREME PROGRAMMING

eXtreme Programming (XP) is a software development method based on frequent cycles. The name is based on the philosophy of distilling a given best practice to its purest, simplest form, and applying that practice continuously throughout the project.

XP is most known for popularizing a holistic set of practices intended to improve the results of software projects. The method was first formalized as a set of twelve primary practices, but then gradually evolved to adopt several other

corollary practices. These are listed in Table A3-2.

Table A3-2. The Practices of eXtreme Programming

XP Practice Area	Primary	Secondary
Organizational	• Sit together • Whole team • Informative workspace	• Real customer involvement • Team continuity • Sustainable pace
Technical	• Pair programming • Test-first programming • Incremental design	• Shared code/collective ownership • Documentation from code and tests • Refactoring
Planning	• User stories • Weekly cycle • Quarterly cycle • Slack	• Root cause analysis • Shrinking teams • Pay per use • Negotiated scope contract • Daily standups
Integration	• 10-minute build • Continuous integration • Test-first	• Single code base • Incremental deployment • Daily deployment

This evolution was the result of designing and adopting techniques through the filter of core values (communication, simplicity, feedback, courage, respect), and informed by key principles (humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, accepted responsibility).

A3.4 KANBAN METHOD

Kanban in lean manufacturing is a system for scheduling inventory control and replenishment. This process of “just-in-time” inventory replenishment was originally seen in grocery stores when shelves were restocked based on the gaps in the shelves and not supplier inventory. Inspired by these just-in-time inventory systems, Taiichi Ohno developed Kanban and it was applied at the main Toyota manufacturing facility in 1953.

The word *kanban* is literally translated as “visual sign” or “card.” Physical kanban boards with cards enable and promote the visualization and flow of the

work through the system for everyone to see. This information radiator (large display) is made up of columns that represent the states the work needs to flow through in order to get to done. The simplest of boards could have three columns (i.e., to do, doing, and done), but it is adaptable to whatever states are deemed needed by the team utilizing it.

The Kanban Method is utilized and applicable in many settings and allows for a continuous flow of work and value to the customer. The Kanban Method is less prescriptive than some agile approaches and thus less disruptive to begin implementing as it is the original "start where you are" method. Organizations can begin applying Kanban Methods with relative ease and progress toward fully implementing the method if that is what they deem necessary or appropriate.

Unlike most agile approaches, the Kanban Method does not prescribe the use of timeboxed iterations. Iterations can be used within the Kanban Method, but the principle of pulling single items through the process continuously and limiting work in progress to optimize flow should always remain intact. The Kanban Method may be best used when a team or organization is in need of the following conditions:

- **Flexibility.** Teams are typically not bound by timeboxes and will work on the highest priority item in the backlog of work.
- **Focus on continuous delivery.** Teams are focused on flowing work through the system to completion and not beginning new work until work in progress is completed.
- **Increased productivity and quality.** Productivity and quality are increased by limiting work in progress.
- **Increased efficiency.** Checking each task for value adding or non-value-added activities and removing the non-value adding activities.
- **Team member focus.** Limited work in progress allows the team to focus on the current work.
- **Variability in the workload.** When there is unpredictability in the way that work arrives, and it becomes impossible for teams to make predictable commitments; even for short periods of time.
- **Reduction of waste.** Transparency makes waste visible so it can be removed.

The Kanban Method is derived from lean thinking principles. The defining

principles and the core properties of the Kanban Method are listed in Table A3-3.

The Kanban Method is a holistic framework for incremental, evolutionary process and systems change for organizations. The method uses a “pull system” to move the work through the process. When the team completes an item, the team can pull an item into that step.

Table A3-3. Defining Principles and Properties of the Kanban Method

Defining Principles	Core Properties
Start with current state Agree to pursue incremental, evolutionary change Respect the current process, roles, responsibilities, and titles Encourage acts of leadership at all levels	Visualize the workflow Limit work in progress Manage flow Make process policies explicit Implement feedback loops Improve collaboratively

Kanban boards, such as the one shown in Figure A3-2, are a low-tech, high-touch technology that may seem overly simplistic at first, but those using them soon realize their power. Utilizing policies for entry and exit to columns, as well as constraints such as limiting work in process, kanban boards provide clear insight to workflow, bottlenecks, blockers, and overall status. Additionally the board acts as an information radiator to anyone who sees it, providing up-to-date information on the status of the work of the team.

Figure A3-2. Kanban Board Demonstrating Work in Progress Limits, and a Pull System to Optimize the Flow of Work

In the Kanban Method, it is more important to complete work than it is to start new work. There is no value derived from work that is not completed so the team works together to implement and adhere to the work in progress (WIP) limits and get each piece of work through the system to "done."

A3.5 CRYSTAL METHODS

Crystal is a family of methodologies. Crystal methodologies are designed to scale, and provide a selection of methodology rigor based on project size (number of people involved in the project) and the criticality of the project.

Figure A3-3. The Crystal Family of Methods

Crystal Methodology realizes that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project's unique characteristics. The family of methodologies use different colors based on "weight" to determine which methodology to use. The use of the word crystal comes from the gemstone where the various "faces" represent underlying core principles and values. The faces are a representation of techniques, tools, standards, and roles listed in Table A3-4.

Table A3-4. The Core Values and Common Properties of Crystal

Core Values	Common Properties[A]
People	Frequent delivery
Interaction	Reflective improvement
Community	Close or osmotic communication
Skills	Personal safety

Talents	Focus
Communications	Easy access to expert users
	Technical environment with automated tests, configuration management, and frequent integration

[A] The more these properties are in a project, the more likely it is to succeed.

A3.6 SCRUMBAN

Scrumban is an agile approach originally designed as a way to transition from Scrum to Kanban. As additional agile frameworks and methodologies emerged, it became an evolving hybrid framework in and of itself where teams use Scrum as a framework and Kanban for process improvement.

In Scrumban, the work is organized into small “sprints” and leverages the use of kanban boards to visualize and monitor the work. The stories are placed on the kanban board and the team manages its work by using work-in-progress limits. Daily meetings are held to maintain the collaboration between the team and to remove impediments. A planning trigger is set in place for the team to know when to plan next, typically when the work-in-progress level is lower than a predetermined limit. There are no predefined roles in Scrumban—the team retains their current roles.

A3.7 FEATURE-DRIVEN DEVELOPMENT

Feature-Driven Development (FDD) was developed to meet the specific needs of a large software development project. Features relate to a small business value capability.

There are six primary roles on a Feature-Driven Development project where individuals can take on one or more of the following roles:

- Project manager,
- Chief architect,
- Development manager,
- Chief programmer,
- Class owner, and/or

- Domain expert.

A Feature-Driven Development project is organized around five processes or activities, which are performed iteratively:

- Develop an overall model,
- Build a features list,
- Plan by feature,
- Design by feature, and
- Build by features.

The life cycle flow and interaction of these five processes is illustrated in Figure A3-4.

Feature-Driven Development activities are supported by a core set of software engineering best practices:

- Domain object modeling,
- Developing by feature,
- Individual class ownership,
- Feature teams,
- Inspections,
- Configuration management,
- Regular builds, and
- Visibility of progress and results.

Figure A3-4. Feature-Driven Development Project Life Cycle

A3.8 DYNAMIC SYSTEMS DEVELOPMENT METHOD

Dynamic Systems Development Method (DSDM) is an agile project delivery framework initially designed to add more rigor to existing iterative methods popular in the 1990s. It was developed as a noncommercial collaboration among industry leaders.

DSDM is known best for its emphasis on constraint-driven delivery. The framework will set cost, quality, and time at the outset, and then use formalized prioritization of scope to meet those constraints as shown in Figure A3-5.

Figure A3-5. DSDM Approach to Constraint-Driven Agility

Eight principles guide the use of the DSDM framework:

- Focus on the business need.
- Deliver on time.
- Collaborate.
- Never compromise quality.
- Build incrementally from firm foundations.
- Develop iteratively.
- Communicate continuously and clearly.
- Demonstrate control (use appropriate techniques).

A3.9 AGILE UNIFIED PROCESS

The Agile Unified Process (AgileUP) is an offshoot of the Unified Process (UP) for software projects. It features more accelerated cycles and less heavyweight processes than its Unified Process predecessor. The intent is to perform more iterative cycles across seven key disciplines, and incorporate the associated feedback before formal delivery. The disciplines along with guiding

principles are listed in Table A3-5.

Table A3-5. The Key Elements of the Agile Unified Process

Disciplines within a Release	Principles Guiding the Disciplines
Model	The team knows what it's doing
Implementation	Simplicity
Test	Agility
Deployment	Focus on high-value activities
Configuration management	Tool independence
Project management	Tailoring to fit
Environment	Situationally specific

A3.10 SCALING FRAMEWORKS

A3.10.1 SCRUM OF SCRUMS

Scrum of Scrums (SoS), also known as “meta Scrum,” is a technique used when two or more Scrum teams consisting of three to nine members each need to coordinate their work instead of one large Scrum team. A representative from each team attends a meeting with the other team representative(s), potentially daily but typically two to three times a week. The daily meeting is conducted similar to the daily standup in Scrum where the representative reports completed work, next set of work, any current impediments, and potential upcoming impediments that might block the other team(s). The goal is to ensure the teams are coordinating work and removing impediments to optimize the efficiency of all the teams.

Large projects with several teams may result in conducting a Scrum of Scrum of Scrums, which follows the same pattern as SoS with a representative from each SoS reporting into a larger group of representatives as shown in Figure A3-6.

Figure A3-6. Representatives of Scrum Teams Participating in SoS teams

A3.11 SCALED AGILE FRAMEWORK

The Scaled Agile Framework (SAFe®) focuses on providing a knowledge base of patterns for scaling development work across all levels of the enterprise.

SAFe® is focused on the following principles:

- Take an economic view.
- Apply systems thinking.
- Assume variability; preserve options.
- Build incrementally with fast, integrated learning cycles.
- Base milestones on objective evaluation of working systems.
- Visualize and limit work in progress, reduce batch sizes, and manage queue lengths.
- Apply cadence; synchronize with cross-domain planning.

- Unlock the intrinsic motivation of knowledge workers.
- Decentralize decision making.

SAFe® focuses on detailing practices, roles, and activities at the portfolio, program, and team levels with an emphasis on organizing the enterprise around value streams that focus on providing continuous value to the customer.

A3.12 LARGE SCALE SCRUM (LeSS)

Large Scale Scrum (LeSS) is a framework for organizing several development teams toward a common goal extending the Scrum method shown in Figure A3-6. The core organizing principle is to retain as much as possible of the elements of the conventional single-team Scrum model. This helps minimize any extensions to the model that might create unnecessary confusion or complexity. Table A3-6 shows a comparison of LeSS and Scrum.

Table A3-6. Comparison of LeSS and Scrum

Similarities of LeSS and Scrum	LeSS Techniques Added to Scrum
One single product backlog One definition of done for all teams One potentially shippable product increment at the end of each sprint One product owner Complete, cross-functional teams One sprint	Sprint planning is more formally divided into two parts of what and how Organic cross-team coordination Overall cross-team refinement Overall retrospective focused on cross-team improvements

In order to extend Scrum without losing its essence, LeSS promotes the use of certain discerning principles, such as systems thinking, whole product focus, transparency, and others.

A3.13 ENTERPRISE SCRUM

Enterprise Scrum is a framework designed to apply the Scrum method on a more holistic organizational level rather than a single product development effort. Specifically, the framework advises organization leaders to:

- Extend the use of Scrum across all aspects of the organization;
- Generalize the Scrum techniques to apply easily at those various aspects; and
- Scale the Scrum method with supplemental techniques as necessary.

The intent is to use agile approaches beyond project execution by enabling disruptive innovation.

A3.14 DISCIPLINED AGILE (DA)

Disciplined Agile (DA) is a process decision framework that integrates several agile best practices into a comprehensive model. DA was designed to offer a balance between those popular methods deemed to be either too narrow in focus (e.g., Scrum) or too prescriptive in detail (e.g., AgileUP). To achieve that balance, it blends various agile techniques according to the following principles:

- **People-first.** Enumerating roles and organization elements at various levels.
- **Learning-oriented.** Encouraging collaborative improvement.
- **Full delivery life cycle.** Promoting several fit-for-purpose life cycles.
- **Goal-driven.** Tailoring processes to achieve specific outcomes.
- **Enterprise awareness.** Offering guidance on cross-departmental governance.
- **Scalable.** Covering multiple dimensions of program complexity.

APPENDIX X1
CONTRIBUTORS AND REVIEWERS

X1.1 *AGILE PRACTICE GUIDE* CORE COMMITTEE

The following individuals were members of the project Core Committee responsible for drafting the guide, including review and adjudication of reviewer recommendations.

X1.1.1 REPRESENTING THE PROJECT MANAGEMENT INSTITUTE:

Mike Griffiths, PMP, PMI-ACP, (Committee Chair)
Jesse Fewell, CST, PMI-ACP
Horia Sluşanschi, PhD, CSM
Stephen Matola, BA, PMP

X1.1.2 REPRESENTING AGILE ALLIANCE:

Johanna Rothman, MS (Committee Vice Chair)
Becky Hartman, PMI-ACP, CSP
Betsy Kauffman, ICP-ACC, PMI-ACP

X1.2 *AGILE PRACTICE GUIDE* SUBJECT

MATTER EXPERT REVIEWERS

The following individuals were invited subject matter experts who reviewed the draft and provided recommendations through the SME Review.

Joe Astolfi, PMP, PSM
Maria Cristina Barbero, PMI-ACP, PMP
Michel Biedermann, PhD, PMI-ACP
Zach Bonaker
Robert Bulger, PfMP, CSM
Sue Burk
Shika Carter, PMP, PMI-ACP
Lauren Clark, PMP, CSM
Linda M Cook, CSM, CSPO
Pamela Corbin-Jones, PMI-ACP, CSM
Jeff Covert
Alberto Dominguez, MSc, PMP
Scott P. Duncan, CSM, ICP-ACC
Sally Elatta, PMI-ACP, EBAC
Frank R. Hendriks, PMP, PMI-ACP
Derek Huether
Ron Jeffries
Fred Koos
Philippe B. Kruchten, PhD, PEng
Steve Mayner, SPCT4, PMP
Michael S. McCalla, PMI-ACP, CSP
Don B. McClure, PMP, PMI-ACP
Anthony C. Mersino, PMI-ACP, CSP
Kenneth E. Nidiffer, PhD, PMP
Michael C. Nollet, PMP, PMI-ACP
Laura Paton, MBA, PMP
Yvan Petit, PhD, PMP
Dwayne Phillips, PhD, PMP
Piyush Prakash, PMP, Prince2
Dave Prior, PMP, CST
Daniel Rawsthorne, PhD, PMP
Annette D. Reilly, PMP, PhD
Stephan Reindl, PMI-ACP, PMP
Reed D. Shell, PMP, CSP
Cindy Shelton, PMP, PMI-ACP

Cindy Shelton, PMP, PMI-ACP
Teresa Short
Lisa K. Sieverts, PMP, PMI-ACP
Christopher M. Simonek, PMP, CSM
Robert “Sellers” Smith, PMP, PMI-ACP
Ram Srinivasan, PMP, CST
Chris Stevens, PhD
Karen Strichartz, PMP, PMI-ACP
Rahul Sudame, PMI-ACP
Joanna L. Vahlsing, PMP
Erik L. van Daalen
Annette Vendelbo, PMP, PMI-ACP
Dave Violette, MPM, PMP
Anton Vishnyak, PMI-ACP, CSM
Chuck Walrad, MA, MS

X1.3 FORMAT FOCUS GROUP

The following individuals assisting in the development of new content style and formatting elements for the *Agile Practice Guide.*

Goran Banjanin, PgMP, PMP
Andrew Craig
Cătălin-Teodor Dogaru, PhD, PMP
Jorge Espinoza, PMP
Jennifer M. Forrest, CSM, PMP
Helen Fotos, PMP, PMI-ACP
Dave Hatter, PMP, PMI-ACP
Christopher Healy, PMP
Mike Hoffmann, MBA, PMP
Chadi Kahwaji, PMP
Rajaraman Kannan, PMP, MACS CP
Amit Khanna PMP, PMI–ACP
Ariel Kirshbom, PMI-ACP, CSP
Bernardo Marques, PMP
Noura Saad, PMI-ACP, CSPO
Kurt Schuler, PMP
Demetrius L. Williams, MBA, PMP
Liza Wood
Melody Yale, CSP, SPC4

X1.4 PMI STANDARDS MEMBER ADVISORY GROUP (MAG)

The following individuals are members of the PMI Standards Member Advisory Group, who provided direction to and final approval on behalf of PMI for the *Agile Practice Guide.*

Maria Cristina Barbero, PMI-ACP, PMP
Brian Grafsgaard, PMP, PgMP
Hagit Landman, PMP, PMI-SP
Yvan Petit PhD, PMP
Chris Stevens, PhD
Dave Violette, MPM, PMP
John Zlockie, MBA, PMP, PMI Standards Manager

X1.5 AGILE ALLIANCE® BOARD

The following individuals are members of Agile Alliance Board of Directors, who provided direction to and final approval on behalf of Agile Alliance for the *Agile Practice Guide.*

Juan Banda
Phil Brock (Managing Director)
Linda Cook
Stephanie Davis
Ellen Grove
Paul Hammond (Chair)
Victor Hugo Germano
Rebecca Parsons (Secretary)
Craig Smith
Declan Whelan

X1.6 PMI SUPPORT STAFF AND ACADEMIC RESEARCH SUPPORT

The following individuals worked to support the core committee in the

development and approval of the draft, in support of the Format Focus Group, and in PMI marketing efforts.

Melissa M. Abel, Marketing Communications Specialist
Karl F. Best, PMP, CStd, Standards Specialist
Alicia C. Burke, MBA, CSM, Product Manager, Credentials
Edivandro C. Conforto, PhD, PMI Consultant on Agile Research
Dave Garrett, CSPO, Vice President, Transformation
Erica Grenfell, Administrative Assistant to VP, Organization Relations
M. Elaine Lazar, MA, MA, AStd, Project Specialist
Andrew Levin, PMP, Project Manager
Tim E. Ogline, User Experience Designer
Stephen A. Townsend, Director of Network Programs
Michael Zarro, PhD, UX Researcher

X1.7 PMI PRODUCTION STAFF

Donn Greenberg, Manager, Publications
Kim Shinners, Publications Production Associate
Roberta Storer, Product Editor
Barbara Walsh, Production Supervisor

APPENDIX X2
ATTRIBUTES THAT INFLUENCE TAILORING

X2.1 INTRODUCTION

This appendix provides high-level guidance on when and how to tailor agile approaches. It can be used to determine circumstances that might warrant changing or introducing new techniques, and then offers some recommendations to consider.

X2.2 FIRST SOME CAUTIONS

Tailoring is an advanced topic that should be undertaken by experienced practitioners who have been successful using agile approaches as originally described in multiple environments before they consider tailoring them. In other words, gain experience and be successful with one approach before attempting to tailor the approach.

The Shu-Ha-Ri model of skills acquisition describes progression from obeying the rules (Shu , means to obey and protect), through consciously moving away from the rules (Ha , means to change or digress), and finally through steady practice and improvement finding an individual path (Ri , means to separate or leave). We need to start and practice at the Shu level before we are ready to move to the Ha level to tailor the process or the Ri

level to invent a new custom process.

A common response when struggling to adopt an agile practice is to consider whether to do it or not. A statement like “Retrospectives were unpopular so we decided to drop them” illustrates this issue and indicates a more fundamental problem on the team that is unlikely to be addressed by tailoring the method. The situation will be made worse by omitting the retrospective activity that aims to improve the process.

Finally, tailoring should be undertaken in collaboration with the teammates or whoever the change is likely to impact. People need to be engaged in the thinking and decision-making process about changing processes in order for them to commit and buy-in to the changes in order to have a successful transition. Omitting people from tailoring a process is likely to result in resistance and resentment to the change, even if it makes good sense technically. Often, experienced coaches or leaders can help to engage people effectively.

X2.3 HOW TO USE THIS APPENDIX

To benefit from the guidance listed in this appendix, we recommend first successfully using the agile approaches as designed. Then review the tailoring guidelines in Table X2-1 that match the situation and read the associated recommendations. Next, discuss the change with the people it will impact and agree on a course of action.

As discussed in Section 5, a good way to evaluate a change is try it for an iteration or two first before adopting it permanently. Or, consider a flow-based approach to try to deliver several features. Then, reflect with a retrospective and reassess.

When people know they can experiment and provide feedback on the experiment, they are more likely to try something new. Having tried it for a timeboxed period, the team should review its effectiveness at a retrospective to determine whether it should be continued as-is, modified to improve it, or dropped from use.

Finally, successfully adopted, tailored approaches can be institutionalized into the standard processes used for projects that share these characteristics. It is also

recommended that guidelines from Section 5 be followed that describe adopting (or tailoring) new approaches.

X2.4 TAILORING RECOMMENDATIONS

Listed below are some good practices to consider before tailoring an approach.

X2.4.1 BEWARE OF TAKING THINGS AWAY

Many of the agile practices act as self-supporting pairs. For instance, colocation and frequent business conversations allow for lightweight requirements since gaps in understanding can be filled quickly. Likewise, XP's ruthless testing allows for courageous refactoring as one practice supports the other. Removing something without understanding or addressing its counterbalanced practice will likely create more problems than it solves.

X2.4.2 USE THE TAILORING GUIDELINES TABLE

Using Table X2-1, find the circumstances that match a given situation and consider recommendations for tailoring. Discuss any changes with those who will be impacted by the change and plan a short trial first, along with an honest follow-up review before committing to the change.

Table X2-1. Tailoring Guidelines

Situation	Tailoring Recommendation
Very large project teams	Restructure large projects as multiple smaller projects. Try a technology trial project first and then an implementation project.
	Consider more frequent releases of fewer features each, which allows for the creation of smaller project teams.
	Consider reducing the team down to its critical core members. Often too many people hinder a process, not help it. Reducing a team size can reduce churn as well as costs.
	Break large teams into multiple smaller teams and use program management to synchronize and coordinate.
	Use agile and lean program management to organize the

	Use agile and lean program management to organize the larger effort.
	Consider a scaled agile or lean framework such as DA, SAFe®, or LeSS. Each offers some useful ideas, and each carries implementation risks and process weight/cost.
Dispersed teams	Many projects have (some) dispersed team members. Tools like instant messaging, video conferencing, and electronic team boards help bridge many of the communication gaps.
	When teams are likely to remain stable, set up face-to-face meetings as soon as possible to make future remote conversations more effective. People who have met face-to-face are more likely to enter unfiltered debate because of higher trust.
	When conducting meetings with remote participants where there is a loss of facial and body-language cues, consider round-robin check-ins to ensure participation and check consensus for decisions.
	Also, consider the use of iteration-based agile approaches. When team members are many time zones apart, consider using whole-project interactions less frequently, while encouraging more personal meetings (two or three people at a time) more frequently.
Some safety critical products may require additional documentation and conformance checks beyond what agile processes suggest out-of-the-box	Agile approaches can still be used in these environments, but they need to have the appropriate additional layers of conformance review, documentation, and certification that is required by the domain. In that case, documentation could be part of what the team delivers along with finished features. Features may not be done until the documentation is completed.
	Consider using a hybrid approach (multiple agile approaches) to get the benefits of improved collaboration and communication brought by agile with the added rigor required by the product environment. Aircraft flight system developers and drug companies use agile approaches coupled with their own additional processes to leverage the benefits and retain appropriate controls.
Stable requirements and execution process	Is agile really needed? If uncertainty in requirements is low, low rates of change, or minimal execution risk, the full suite of agile approaches may not be needed. While any project benefits from increased collaboration and transparency; some of the iterative build and review cycles might be overkill.
	If build/feedback cycles do not routinely uncover or refine requirements, consider extending their durations to minimize the cost impact of review time.

	If the project has high rates of change during design and development, but rolling it out to customers is a defined and repeatable process, hybrid approaches that use the appropriate life cycle model for each project phase may make more sense.
Teams are in functional silos inside functional organizations	Agile is built on the idea of cross-functional teams. Consider asking people to create cross-functional teams themselves, without management involvement and see what happens.
	If the compensation system is organized to recognize and reward functional areas, consider changing that first. People might not act in the interest of the product or the team until it affects their compensation in some way.
Transparency can cause fear	Agile creates a culture of transparency: people show and share their work throughout development. This sharing of interim deliverables and being open and honest about successes, failures, and current state is transparency. Transparency requires courage.
	Lead by example and demonstrate transparency in decision-making processes by using a status board or whiteboard.
Many of the team members have little technical domain knowledge	Agile approaches encourage and make use of self-directing teams to make local decisions about work items, such as task sequencing and which approach to use when solving a problem. When the majority of team members are inexperienced, consensus-based approaches may lead to problems and rework. So, for these teams, additional help “assigning” and “directing” may be necessary until the team gains the necessary skills. In other words, do not just declare that agile will be used and let an inexperienced team try to figure everything out because they are empowered and self-directing. Consider building centers of competencies to help provide guidance and build domain knowledge.
Lack of executive buy-in	When executive buy-in is missing, teams will encounter a clash between the agile mindset and approaches and the more predictive mindset and approaches.
	Find common ground, areas for improvement based on the organization's needs, and then use experiments and retrospectives to progress.
	Consider education/training for executives. Consider explaining agile in terms of lean thinking: short cycles, small batch sizes, frequent reviews, and retrospectives with small improvements.
	Modify the terms so people will understand and agree to the

Agile terms and language do not fit the organizational culture	Modify the terms so people will understand and agree to the activities, if not the agile language. Be specific about what each term means. For example, If the organization finds the word “game” unprofessional, don't use terms such as “planning game.” Instead, consider using the term “planning workshop.”

APPENDIX X3
AGILE SUITABILITY FILTER TOOLS

X3.1 INTRODUCTION

Agile literature contains many agile suitability filter tools to help assess under what circumstances an agile approach is appropriate to use. In 1994, the Dynamic Systems Development Method (DSDM) developed an Agile Project Suitability Questionnaire and an Organizational Suitability Questionnaire to help gauge likely fit and potential problem areas.

The Crystal family of approaches also employed suitability criteria, ranking projects by team size and the criticality of the product or service being developed. Crystal recommends that smaller, less critical projects be undertaken with lighter controls and simpler approaches. Large, mission or life critical projects were recommended to use more rigor and validation.

Since the development of these approaches, there have been many more models created to help determine where and when to employ agile approaches. Boehm and Turner adopted some of the elements from DSDM and Crystal to develop a popular assessment model to help determine if projects should be undertaken with agile or more traditional approaches.

Based on these previous models and expanded to consider the middle ground of hybrid approaches, the following model is proposed. It represents a synthesis of several suitability filter attributes to help organizations assess and discuss whether projects should be undertaken using predictive, hybrid, or agile approaches.

X3.2 OVERVIEW OF THE MODEL

Organizational and project attributes are assessed under three main categories:

- **Culture.** Is there a supportive environment with buy-in for the approach and trust in the team?
- **Team.** Is the team of a suitable size to be successful in adopting agile, do its members have the necessary experience and access to business representatives to be successful?
- **Project.** Are there high rates of change? Is incremental delivery possible? How critical is the project?

Questions in each of these categories are answered and the results plotted on a radar chart. Clusters of values around the center of the chart indicate a good fit for agile approaches. Results around the outside indicate a predictive approach may be more suitable. Values in the middle portion (between agile and predictive) indicate a hybrid approach could work well. An example is shown in Figure X3-1.

Figure X3-1. Model for Suitability of Agile Approach

X3.3 INSTRUCTIONS FOR USE

X3.3.1 COMPLETE THE QUESTIONNAIRE AS A GROUP

For small projects, this group may simply be the sponsor, technical lead, and a customer. For large projects, this may include representatives from the sponsoring group, project execution team, impacted business group(s), project governance group(s), and customer community. The idea is that just as no single stakeholder should estimate or plan a project because of representing only one

viewpoint and having personal bias; so too should no single person assess the suitability of an approach since any one person will also have a limited view with a bias.

Instead, the value of the tool is the conversation it encourages with the invested parties of the project. Even if the results point to a hybrid approach, but the stakeholders want to proceed with a largely agile or predictive approach, follow the stakeholder consensus. This tool is a high-level diagnostic only, the final decision should rest and be supported by the people involved.

X3.3.2 SCORE THE QUESTIONS FROM 1 TO 10

As a group, discuss and agree (or compromise) on a score that most accurately reflects the subjective evaluation of the question. While definitive options are only provided for the start, middle, and end points of the answer spectrum representing scores of 1, 5, and 10, it is fine (and desirable) to use scores such as 2 for “almost a 1, but not quite,” or 7 for “somewhere between a 5 and a 10.” Again, the assessment is a discussion tool—views will be subjective and shades of gray are to be expected.

When the group cannot agree on a score, discuss the issues openly and honestly. Before suggesting compromises (i.e., using average scores or marking PMO scores with a blue “X” and the development team with a green “O”), consider how successful is the project likely to be when the participants cannot agree on completing a simple assessment? When discussing the issues, if the differences of opinion can be identified—then great, it is working; now come to an agreement. Likewise, if the assessment indicates a predictive approach but everyone wants to try an agile approach (or vice versa) that is fine too, just understand the issues and discuss how the impacts of the approach will be handled.

X3.3.3 INTERPRET THE RESULTS

Mark the answers from the questions on a blank suitability assessment chart and connect the points. Results clustered around the center in the agile zone indicate a good fit for a purely agile approach.

Results predominantly in the hybrid zone indicate some combination of agile and predictive approaches might work best. However, it is also possible that an

agile approach with some additional risk reduction steps such as extra education and training or extra validation and documentation rigor in the case of high criticality projects may suffice. Alternatively, a predictive approach with some proof-of-concept work or extra processes could also work.

Results predominantly in the predictive zone indicate a good fit for a purely predictive approach. As mentioned in Section X3.3.2 (Score the Questions step), this diagnostic tool is aimed at starting meaningful conversations with the impacted parties about the most appropriate approach to use. If the approach suggested by the tool is not acceptable it is allowed to use a different approach. Use the results as inputs to the risk management process, since the tool indicates mismatches that will need to be managed.

X3.4 SUITABILITY FILTER QUESTIONS

X3.4.1 CATEGORY: CULTURE

X3.4.1.1 BUY-IN TO APPROACH

Is there senior sponsor understanding and support for using an agile approach for this project? See Figure X3-2.

Figure X3-2. Buy-In to Approach Assessment

X3.4.1.2 TRUST IN TEAM

Considering the sponsors and the business representatives who will be working with the team. Do these stakeholders have confidence that the team can

transform their vision and needs into a successful product or service—with ongoing support and feedback going both directions? See Figure X3-3.

Figure X3-3. Trust in Team Assessment

X3.4.1.3 DECISION-MAKING POWERS OF TEAM

Will the team be given autonomy to make their own local decisions about how to undertake work? See Figure X3-4.

Figure X3-4. Assessment for Decision-Making Powers of Team

X3.4.2 CATEGORY: TEAM

X3.4.2.1 TEAM SIZE

What is the size of the core team? Use this scale: 1-9 = 1, 10-20 = 2, 21-30 = 3, 31-45 = 4, 46-60 = 5, 61-80 = 6, 81-110 = 7, 111-150 = 8, 151 – 200 = 9, 201+ = 10. See Figure X3-5.

Figure X3-5. Team Size Assessment

X3.4.2.2 EXPERIENCE LEVELS

Considering the experience and skill levels of the core team roles. While it is normal to have a mix of experienced and inexperienced people in roles, for agile projects to go smoothly; it is easier when each role has at least one experienced member. See Figure X3-6.

Figure X3-6. Experience Level Assessment

X3.4.2.3 ACCESS TO THE CUSTOMER/BUSINESS

Will the team have daily access to at least one business/customer representative to ask questions and get feedback? See Figure X3-7.

Figure X3-7. Assessment for Access to the Customer/Business

X3.4.3 CATEGORY: PROJECT

X3.4.3.1 LIKELIHOOD OF CHANGE

What percentage of requirements are likely to change or be discovered on a monthly basis? See Figure X3-8.

Figure X3-8. Likelihood of Change Assessment

X3.4.3.2 CRITICALITY OF PRODUCT OR SERVICE

To help determine likely levels of additional verification and documentation rigor that may be required, assess the criticality of the product or service being built. Using an assessment that considers loss due to possible impact of defects, determine what a failure could result in. See Figure X3-9.

Figure X3-9. Assessment for Criticality of Product or Service

X3.4.3.3 INCREMENTAL DELIVERY

Can the product or service be built and evaluated in portions? Also, will business or customer representatives be available to provide timely feedback on increments delivered? See Figure X3-10.

Figure X3-10. Incremental Delivery Assessment

X3.5 SUITABILITY ASSESSMENT CHART

Figure X3-11 is the radar chart used for the suitability assessment.

Figure X3-11. Suitability Assessment Radar Chart

X3.5.1 CASE STUDIES

To illustrate how the radar chart works, here are two examples of using the model to score very different types of projects. The first is an example of an online drug store project (see Figure X3-12) and the second (Figure X3-13) is an example of a military messaging system. These two case studies illustrate some of the variances seen on projects. Central clustering indicates a good fit for agile approaches, peripheral scores indicate predictive approaches might be more suitable. Some projects are centered around the middle but then spike out on one or two axes. These projects may be best solved with a hybrid approach.

Figure X3-12. Drug Store Project

X3.5.1.1 DRUG STORE EXAMPLE

The project was to develop an online drug store to sell cheaper Canadian prescription drugs to (primarily) U.S. customers. The sale of these drugs is a contentious subject in Canada as well as the U.S. and as a result the industry is characterized by swift regulation changes and fierce competition. The project faced extremely volatile requirements with major changes week on week. It used very short (2-day) iterations and weekly releases to tackle the high rates of change.

As shown in Figure X3-12, high levels of buy-in and trust are evident for those who worked in an empowered way. The visual nature of the website made it easy to show new increments of functionality, but the system criticality was

fairly high with essential funds for the pharmacy at stake. As mentioned, there were very high rates of change, but the small experienced team handled them well and had easy access to a knowledgeable business representative. The approach was very successful and extremely agile.

X3.5.1.2 MILITARY MESSAGING SYSTEM EXAMPLE

Contrast the first example with a large project to develop military messaging system that had already been running for 5 years when the assessment was made. See Figure X3-13.

Figure X3-13. Military Messaging Example

Buy-in for an agile approach was lacking because an agile approach was not

being considered. Trust in the vendors was mixed but generally respected. Decision making was not local, but instead made by architecture and requirements committees. While elements of the design could be tested incrementally in a laboratory, they could not be gathered together for an end to end demonstration of functionality. Many lives were potentially at risk, so criticality was very high. Requirements were locked down because changes impacted so many subcontractor organizations.

The project was large with more than 300 people from one vendor alone, but each role had many experienced practitioners. Finally, access to the business/customer was not possible, but contract analysts were available to ask specification questions to and they usually replied or asked clarifying questions within 10 days. Parts of the project could have been carved off and run as agile projects, but at the heart of the initiative was a single large project.

X3.6 SUMMARY

Agile suitability filters are useful tools for identifying potential fits and gaps for agile approaches. They should not be used as definitive inclusion or exclusion gates, but instead as topics for objective discussion with all interested parties.

REFERENCES

[1] *Manifesto for Agile Software Development.* (2001). Retrieved from http://agilemanifesto.org/

[2] Project Management Institute. 2013. *Managing Change in Organizations: A Practice Guide.* Newtown Square, PA: Author.

[3] Project Management Institute. 2017. *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)* – Sixth Edition. Newtown Square, PA: Author.

[4] Project Management Institute. 2013. *Software Extension to the PMBOK® Guide Fifth Edition.* Newtown Square, PA: Author.

BIBLIOGRAPHY

The following are suggested additional reading materials, subdivided by section and/or topic:

SECTION 2—AN INTRODUCTION TO AGILE

Briggs, Sara. "Agile Based Learning: What Is It and How Can It Change Education?" *Opencolleges.edu.au* February 22, 2014, retrieved from http://www.opencolleges.edu.au/informed/features/agile-based-learning-what-is-it-and-how-can-it-change-education/.

Manifesto for Agile Software Development, 2001, http://agilemanifesto.org/.

Peha, Steve. "Agile Schools: How Technology Saves Education (Just Not the Way We Thought it Would)." InfoQ. June 28, 2011, retrieved from https://www.infoq.com/articles/agile-schools-education.

Principles behind the Agile Manifesto, 2001, http://agilemanifesto.org/principles.html.

Rothman, Johanna. 2007. *Manage It! Your Guide to Modern, Pragmatic Project Management.* Raleigh: Pragmatic Bookshelf.

Sidky, Ahmed (Keynote). 2015. https://www.slideshare.net/AgileNZ/ahmed-sidky-keynote-agilenz.

Stacey Complexity Model. 2016. http://www.scrum-tips.com/2016/02/17/stacey-complexity-model/.

SECTION 3—LIFE CYCLE SELECTION

"Agile Modeling (AM) Home Page: Effective Practices for Modeling and Documentation," *Agile Modeling*, (n.d.), http://www.agilemodeling.com/

Anderson, David, and Andy Carmichael. 2016. *Essential Kanban Condensed.* Seattle: Blue Hole Press.

Anderson, David. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business.* Seattle: Blue Hole Press.

Benson, Jim, and Tonianne DeMaria Barry. 2011. *Personal Kanban: Mapping Work | Navigating Life.* Seattle: Modus Cooperandi Press.

Burrows, Mike. 2014. *Kanban from the Inside: Understand the Kanban Method, connect it to what you already know, introduce it with impact.* Seattle: Blue Hole Press.

Domain Driven Design Community. 2016. http://dddcommunity.org/.

Gothelf, Jeff, and Josh Seiden. 2016. *Lean UX: Designing Great Products with Agile Teams.* Sebastopol: O'Reilly Media.

Hammarberg, Marcus, and Joakim Sunden. 2014. *Kanban in Action.* Shelter Island: Manning Publications.

"Kanban," *Wikipedia*, last modified May 4, 2017, retrieved on November 22, 2016 from https://en.wikipedia.org/wiki/Kanban.

"Kanban *(development)*," Wikipedia, last modified May 4, 2017, retrieved on November 29, 2016 from https://en.wikipedia.org/wiki/Kanban_(development).

Larsen, Diana, and Ainsley Nies. 2016. *Liftoff: Start and Sustain Successful Agile Teams.* Raleigh: Pragmatic Bookshelf.

"Learning Kanban," *Leankit*, (n.d.), https://leankit.com/learn/learning-kanban/.

Leopold, Klaus, and Siegrfried Kaltenecker. 2015. *Kanban Change Leadership: Creating a Culture of Continuous Improvement.* Hoboken: Wiley.

"Make a big impact with software products and projects!" *Impact Mapping*, (n.d.), https://www.impactmapping.org/.

Patton, Jeff, and Peter Economy. 2014. *User Story Mapping: Discover the Whole Story, Build the Right Product.* Sebastopol: O'Reilly Media.

Reinertsen, Donald. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development.* Redondo Beach: Celeritas Publishing.

Rothman, Johanna. "Dispersed vs. Distributed Teams," *Rothman Consulting Group*, Inc., October 25, 2010, http://www.jrothman.com/mpd/2010/10/dispersed-vs-distributed-teams/.

Schwaber, Ken, and Jeff Sutherland. "The Scrum Guide™," *Scrum.org*, July 2016, http://www.scrumguides.org/scrum-guide.html and http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf#zoom=100.

Skarin, Mattias. 2015. *Real-World Kanban: Do Less, Accomplish More with Lean Thinking.* Raleigh: Pragmatic Bookshelf.

"The High Cost of Multitasking: 40% of Productivity Lost by Task Switching," *Wrike.com*, September 24, 2015, https://www.wrike.com/blog/high-cost-of-multitasking-for-productivity/.

Wells, Don. "Extreme Programming: A Gentle Introduction," *Extreme Programming*, October 8, 2013, http://www.extremeprogramming.org/.

SECTION 4—IMPLEMENTING AGILE:

Amabile, Teresa, and Steven Kramer. 2011. *The Progress Principle: Using Small Wins to Ignite Joy, Engagement, and Creativity at Work.* Boston: Harvard Business Review Press.

"Early Warning Signs of Project Trouble—Cheat Sheet, 2017, https://agilevideos.com/wp-content/uploads/2017/02/WarningSignsOfProjectTrouble-CheatSheet.pdf.

Dweck, Carol. 2006. *Mindset: The New Psychology of Success.* New York: Penguin Random House.

Kaner, Sam. *Facilitator's Guide to Participatory Decision-Making.* 3rd ed. 2014. San Francisco: Jossey-Bass.

Keith, Kent. *The Case for Servant Leadership.* 2008. Westfield: Greenleaf

Center for Servant Leadership.

Rothman, Johanna. 2016. *Agile and Lean Program Management: Scaling Collaboration Across the Organization.* Victoria, British Columbia: Practical Ink.

Rothman, Johanna. "Dispersed vs. Distributed Teams," *Rothman Consulting Group*, Inc., October 25, 2010, http://www.jrothman.com/mpd/2010/10/dispersed-vs-distributed-teams/.

Rothman, Johanna. 2007. *Manage It! Your Guide to Modern, Pragmatic Project Management.* Raleigh: Pragmatic Bookshelf.

Rothman, Johanna. 2016. *Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects.* Raleigh: Pragmatic Bookshelf.

Schwaber, Ken, and Jeff Sutherland. "The Scrum Guide™," *Scrum.org*, July 2016, http://www.scrumguides.org/scrum-guide.html and http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf#zoom=100.

Sinek, Simon. 2011. *Start with Why: How Great Leaders Inspire Everyone to Take Action.* New York: Portfolio, Penguin Random House.

"The High Cost of Multitasking: 40% of Productivity Lost by Task Switching," *Wrike.com*, September 24, 2015, https://www.wrike.com/blog/high-cost-of-multitasking-for-productivity/.

EXPERIENCE REPORTS:

"Experience Reports," *Agile Alliance*, (n.d.), https://www.agilealliance.org/resources/experience-reports/.

PROJECT AND TEAM HEALTH:

"Early Warning Signs of Project Trouble—Cheat Sheet." 2017. https://agilevideos.com/wp-content/uploads/2017/02/WarningSignsOfProjectTrouble-CheatSheet.pdf

"TeamHealth Radar – Summary View," *Agilehealth.* 2014. http://agilityhealthradar.com/wp-content/uploads/2014/11/bigradar.gif.

RESOURCE EFFICIENCY:

Modig, Niklas, and Pär Åhlström. 2015. *This is Lean: Resolving the Efficiency Paradox*. London: Rheologica Publishing.

Rothman, Johanna. "Resource Efficiency vs. Flow Efficiency, Part 5: How Flow Changes Everything," *Rothman Consulting Group, Inc.*, September 20, 2015, http://www.jrothman.com/mpd/agile/2015/09/resource-efficiency-vs-flow-efficiency-part-5-how-flow-changes-everything/.

SCALING:

Disciplined Agile 2.X—A Process Decision Framework. 2016. http://www.disciplinedagiledelivery.com/.

Kniberg, Henrik. "Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds," *Crisp*, November 14, 2012, http://blog.crisp.se/2012/11/14/henrikkniberg/scaling-agile-at-spotify.

"Overview—Large Scale Scrum," *LeSS*. 2016. http://less.works/.

"SAFe® for Lean Software and System Engineering," *SAFe®*. 2016. http://www.scaledagileframework.com/.

SKILLS:

Beck, Kent. *Paint Drip People*, August 4, 2016, https://www.facebook.com/notes/kent-beck/paint-drip-people/1226700000696195/.

"Generalizing Specialists: Improving Your IT Career Skills," *Agile Modeling*, (n.d.), http://www.agilemodeling.com/essays/generalizingSpecialists.htm.

Hunter, Brittany. "Of Software Designers & Broken Combs," *Atomic Object*, June 27, 2013, https://spin.atomicobject.com/2013/06/27/broken-comb-people/.

SECTION 5—IMPLEMENTING AGILE: DELIVERING IN AN AGILE ENVIRONMENT

Larsen, Diana, and Ainsley Nies. 2016. *Liftoff: Start and Sustain Successful Agile Teams*. Raleigh: Pragmatic Bookshelf.

RETROSPECTIVES:

Derby, Esther, and Diana Larsen. 2006. *Agile Retrospectives: Making Good Teams Great.* Raleigh: Pragmatic Bookshelf.

Gonçalves, Luis, and Ben Linders. 2015. *Getting Value out of Agile Retrospectives: A Toolbox of Retrospective Exercises.* Victoria, British Columbia: Leanpub.

BACKLOG:

Adzic, Gojko, Marjory Bissett, and Tom Poppendieck. 2012. *Impact Mapping: Making a Big Impact with Software Products and Projects.* Woking, Surrey: Provoking Thoughts.

Patton, Jeff, and Peter Economy. 2014. *User Story Mapping: Discover the Whole Story, Build the Right Product.* Sebastopol: O'Reilly Media.

Rothman, Johanna. "We Need Planning; Do We Need Estimation?" *Rothman Consulting Group, Inc.*, January 21, 2015, http://www.jrothman.com/mpd/project-management/2015/01/we-need-planning-do-we-need-estimation/.

STANDUPS:

Brodzinski, Pawel. "Effective Standups around Kanban Board," *Brodzinski.com*, December 30, 2011, http://brodzinski.com/2011/12/effective-standups.html.

Fowler, Martin. "It's Not Just Standing Up: Patterns for Daily Standup Meetings," *Martinfowler.com*, February 21, 2016, http://martinfowler.com/articles/itsNotJustStandingUp.html.

Hefley, Chris. "How to Run Effective Standups and Retrospectives," *Leankit*, September 15, 2014, https://leankit.com/blog/2014/09/run-effective-standups-retrospectives/.

EARNED VALUE:

Griffiths, Mike. "A Better S Curve and Simplified EVM," *Leading Answers*, June 6, 2008, http://leadinganswers.typepad.com/leading_answers/2008/06/a-better-s-

curve-and-simplified-evm.html.

SECTION 6—ORGANIZATIONAL CONSIDERATIONS FOR AGILE PROJECTS

Bankston, Arlen, and Sanjiv Augustine. *Agile Team Performance Management: Realizing the Human Potential of Teams*, June 14, 2010, www.lithespeed.com/transfer/Agile-Performance-Management.pptx.

Browder, Justin, and Brian Schoeff. *Perfect Strangers: How Project Managers and Developers Relate and Succeed.* CreateSpace Independent Publishing Platform, 2016, https://www.createspace.com/.

Griffiths, Mike. "Agile Talent Management," *Leading Answers*, October 14, 2015, http://leadinganswers.typepad.com/leading_answers/2015/10/agile-talent-management.html.

Kohn, Alfie. 1999. *Punished by Rewards: The Trouble with Gold Stars, Incentive Plans, A's, Praise, and Other Bribes.* New York: Mariner Books.

Mar, Kane. "How to do Agile Performance Reviews," *Scrumology*, (n.d.), https://scrumology.com/how-to-do-agile-performance-reviews/.

McChrystal, Stanley, Tantum Collins, David Silverman, and Chris Fussell. 2015. *Team of Teams: New Rules of Engagement for a Complex World.* New York: Portfolio, Penguin Random House.

Pink, Daniel. 2011. *Drive: The Surprising Truth About What Motivates Us.* New York: Riverhead Books.

SECTION 7—A CALL TO ACTION (INSPECTION WITHOUT ADAPTATION IS FUTILE)

Dennis, Pascal. 2006. *Getting the Right Things Done: A Leader's Guide to Planning and Execution.* Cambridge: Lean Enterprise Institute.

Griffiths, Mike. "Introducing Agile Methods: Mistakes to Avoid—Part 3," *Leading Answers*, March 15, 2007, http://leadinganswers.typepad.com/leading_answers/2007/03/introducing_a

Little, Jason. *Lean Change Management: Innovative Practices for Managing Organizational Change*. Happy Melly Express, 2014, http://www.happymelly.com/category/hm-express/.

Rising, Linda, and Mary Lynne Manns. 2004. *Fearless Change: Patterns for Introducing New Ideas.* Upper Saddle River: Addison-Wesley Professional.

"The IDEAL Model," *Software Engineering Institute*, *Carnegie Mellon*, 2006, http://www.sei.cmu.edu/library/assets/idealmodel.pdf.

ANNEX A1—PMBOK® GUIDE MAPPING

Larsen, Diana and Ainsley Nies. 2016. *Liftoff: Start and Sustain Successful Agile Teams.* Raleigh: Pragmatic Bookshelf.

ANNEX A2—AGILE MANIFESTO MAPPING

Manifesto for Agile Software Development, 2001, http://agilemanifesto.org/.

Principles behind the Agile Manifesto, 2001, http://agilemanifesto.org/principles.html.

ANNEX A3—OVERVIEW OF AGILE AND LEAN FRAMEWORKS

Agile Business Consortium, 2014, https://www.agilebusiness.org/what-is-dsdm.

Ambler, Scott. "The Agile Unified Process," *Ambysoft*, May 13, 2006, http://www.ambysoft.com/unifiedprocess/agileUP.html.

Anderson, David. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business.* Seattle: Blue Hole Press.

Beedle, Mike. *Enterprise Scrum: Executive Summary: Business Agility for the 21st Century*, January 7, 2017, http://www.enterprisescrum.com/enterprise-scrum/.

Cockburn, Alistair. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Upper Saddle River: Pearson Education.

Cockburn, Alistair. "Crystal Methodologies," *alistair.cockburn.us*, March 28, 2014, http://alistair.cockburn.us/Crystal+methodologies.

Disciplined Agile 2.X—A Process Decision Framework, 2016, http://www.disciplinedagiledelivery.com/.

Joint MIT-PMI-INCOSE Community of Practice on Lean in Program Management. 2012. *The Guide to Lean Enablers for Managing Engineering Programs.* Newtown Square, PA: Author.

"Kanban," *Wikipedia*, last modified May 4, 2017, retrieved on November 22, 2016 from https://en.wikipedia.org/wiki/Kanban.

"Kanban *(development)*," *Wikipedia*, last modified May 4, 2017, retrieved on November 29, 2016 from https://en.wikipedia.org/wiki/Kanban_(development).

Reddy, Ajay, and Jack Speranza. 2015. *The Scrumban [R]Evolution: Getting the Most Out of Agile, Scrum, and Lean Kanban.* Boston: Addison-Wesley Professional.

"Overview—Large Scale Scrum," *LeSS*, 2016, http://less.works/.

"SAFe® for Lean Software and System Engineering," *SAFe®*, 2016, http://www.scaledagileframework.com/.

Schwaber, Ken, and Jeff Sutherland. "The Scrum Guide™," *Scrum.org*, July 2016, http://www.scrumguides.org/scrum-guide.html and http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf#zoom=100.

"Scrum of Scrums," *Agile Alliance*, (n.d.), https://www.agilealliance.org/glossary/scrum-of-scrums/.

"Scrumban," Wikipedia, March 2, 2017, https://en.wikipedia.org/wiki/Scrumban.

"State of Agile Report: Agile Trends," *VersionOne*, 2017, http://stateofagile.versionone.com/.

Sutherland Jeff. "Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies." *Cutter IT Journal* 14, no. 12 (2001): 5–11.

http://www.controlchaos.com/storage/scrum-articles/Sutherland200111proof.pdf.

"The 2015 State of Agile Development," *Scrum Alliance®*, 2015, https://www.forrester.com/report/The+2015+State+Of+Agile+Developmer /E-RES122910

Wells, Don. "Extreme Programming: A Gentle Introduction," *Extreme Programming*, October 8, 2013, http://www.extremeprogramming.org/.

Why Scrum? State of Scrum Report, 2016, https://www.scrumalliance.org/why-scrum/state-of-scrum-report/2016-state-of-scrum.

APPENDIX X2—ATTRIBUTES THAT INFLUENCE TAILORING

Griffiths, Mike. "Agile Suitability Filters," *Leading Answers*, 2007, http://leadinganswers.typepad.com/leading_answers/files/agile_suitability_

Jeffries, Ron. "We Tried Baseball and It Didn't Work," *ronjeffries.com*, May 2, 2006, http://ronjeffries.com/xprog/articles/jatbaseball/.

Rothman, Johanna. "One Experimental Possibility: Self-Organization from Component Teams to Feature Teams," *Rothman Consulting Group, Inc.*, September 23, 2014, http://www.jrothman.com/mpd/agile/2014/09/one-experimental-possibility-self-organization-from-component-teams-to-feature-teams/.

GLOSSARY

1. ACRONYMS

ATDD	acceptance test-driven development
BDD	behavior-driven development
BRD	business requirement documents
DA	Disciplined Agile
DoD	definition of done
DoR	definition of ready
DSDM	Dynamic Systems Development Method
Evo	evolutionary value delivery
LeSS	Large-Scale Scrum
LSD	Lean Software Development
PDCA	Plan-Do-Check-Act
PMO	project management office
ROI	return on investment
RUP	rational unified process
SAFe®	Scaled Agile Framework®
SBE	specification by example

SBE	specification by example
XP	eXtreme Programming

2. DEFINITIONS

A3. A way of thinking and a systematic problem-solving process that collects the pertinent information on a single A3-size sheet of paper.

Acceptance Test-Driven Development (ATDD). A method of collaboratively creating acceptance test criteria that are used to create acceptance tests before delivery begins.

Agile. A term used to describe a mindset of values and principles as set forth in the Agile Manifesto.

Agile Coach. An individual with knowledge and experience in agile who can train, mentor, and guide organizations and teams through their transformation.

Agile Life Cycle. An approach that is both iterative and incremental to refine work items and deliver frequently.

Agile Manifesto. The original and official definition of agile values and principles.

Agile Mindset. A way of thinking and behaving underpinned by the four values and twelve principles of the Agile Manifesto.

Agile Practitioner. A person embracing the agile mindset who collaborates with like-minded colleagues in cross-functional teams. Also referred to as agilist.

Agile Principles. The twelve principles of agile project delivery as embodied in the Agile Manifesto.

Agile Unified Process. A simplistic and understandable approach to developing business application software using agile techniques and concepts. It is a simplified version of the Rational Unified Process (RUP).

Agilist. See *Agile Practitioner*.

Anti-Pattern. A known, flawed pattern of work that is not advisable.

Automated Code Quality Analysis. The scripted testing of code base for bugs and vulnerabilities.

Backlog. See *Product Backlog*.

Backlog Refinement. The progressive elaboration of project requirements and/or the ongoing activity in which the team collaboratively reviews, updates, and writes requirements to satisfy the need of the customer request.

Behavior-Driven Development (BDD). A system design and validation practice that uses test-first principles and English-like scripts.

Blended Agile. Two or more agile frameworks, methods, elements, or practices used together such as Scrum practiced in combination with XP and Kanban Method.

Blocker. See *Impediment*.

Broken Comb. Refers to a person with various depths of specialization in multiple skills required by the team. Also known as Paint Drip. See also *T-shaped* and *I-shaped*.

Burndown Chart. A graphical representation of the work remaining versus the time left in a timebox.

Burnup Chart. A graphical representation of the work completed toward the release of a product.

Business Requirement Documents (BRD). Listing of all requirements for a specific project.

Cadence. A rhythm of execution. See also *Timebox*.

Collective Code Ownership. A project acceleration and collaboration technique whereby any team member is authorized to modify any project work product or deliverable, thus emphasizing team-wide ownership and accountability.

Continuous Delivery. The practice of delivering feature increments immediately to customers, often through the use of small batches of work and automation technology.

Continuous Integration. A practice in which each team member's work products are frequently integrated and validated with one another.

Cross-Functional Team. A team that includes practitioners with all the skills necessary to deliver valuable product increments.

Crystal Family of Methodologies. A collection of lightweight agile software

development methods focused on adaptability to a particular circumstance.

Daily Scrum. A brief, daily collaboration meeting in which the team reviews progress from the previous day, declares intentions for the current day, and highlights any obstacles encountered or anticipated. Also known as daily standup.

Definition of Done (DoD). A team's checklist of all the criteria required to be met so that a deliverable can be considered ready for customer use.

Definition of Ready (DoR). A team's checklist for a user-centric requirement that has all the information the team needs to be able to begin working on it.

DevOps. A collection of practices for creating a smooth flow of delivery by improving collaboration between development and operations staff.

Disciplined Agile (DA). A process decision framework that enables simplified process decisions around incremental and iterative solution delivery.

Double Loop Learning. A process that challenges underlying values and assumptions in order to better elaborate root causes and devise improved countermeasures rather than focusing only on symptoms.

Dynamic Systems Development Method (DSDM). An agile project delivery framework.

Evolutionary Value Delivery (EVO). Openly credited as the first agile method that contains a specific component no other methods have: the focus on delivering multiple measurable value requirements to stakeholders.

eXtreme Programming. An agile software development method that leads to higher quality software, a greater responsiveness to changing customer requirements, and more frequent releases in shorter cycles.

Feature-Driven Development. A lightweight agile software development method driven from the perspective of features valued by clients.

Fit for Purpose. A product that is suitable for its intended purpose.

Fit for Use. A product that is usable in its current form to achieve its intended purpose.

Flow Master. The coach for a team and service request manager working in a continuous flow or Kanban context. Equivalent to *Scrum Master*.

Framework. A basic system or structure of ideas or facts that support an approach.

Functional Requirement. A specific behavior that a product or service should perform.

Functional Specification. A specific function that a system or application is required to perform. Typically represented in a functional specifications document.

Hoshin Kanri. A strategy or policy deployment method.

Hybrid Approach. A combination of two or more agile and non-agile elements, having a non-agile end result.

IDEAL. An organizational improvement model that is named for the five phases it describes: initiating, diagnosing, establishing, acting, and learning.

Impact Mapping. A strategic planning technique that acts as a roadmap to the organization while building new products.

Impediment. An obstacle that prevents the team from achieving its objectives. Also known as a blocker.

Increment. A functional, tested, and accepted deliverable that is a subset of the overall project outcome.

Incremental Life Cycle. An approach that provides finished deliverables that the customer may be able to use immediately.

Information Radiator. A visible, physical display that provides information to the rest of the organization enabling up-to-the-minute knowledge sharing without having to disturb the team.

I-shaped. Refers to a person with a single deep area of specialization and no interest or skill in the rest of the skills required by the team. See also *T-Shaped* and *Broken Comb*.

Iteration. A timeboxed cycle of development on a product or deliverable in which all of the work that is needed to deliver value is performed.

Iterative Life Cycle. An approach that allows feedback for unfinished work to improve and modify that work.

Kaizen Events. Events aimed at improvement of the system.

Kanban Board. A visualization tool that enables improvements to the flow of work by making bottlenecks and work quantities visible.

Kanban Method. An agile method inspired by the original Kanban inventory control system and used specifically for knowledge work.

Large Scale Scrum (LeSS). Large-Scale Scrum is a product development framework that extends Scrum with scaling guidelines while preserving the original purposes of Scrum.

Lean Software Development (LSD). Lean software development is an adaptation of lean manufacturing principles and practices to the software development domain and is based on a set of principles and practices for achieving quality, speed, and customer alignment.

Life Cycle. The process through which a product is imagined, created, and put into use.

Mobbing. A technique in which multiple team members focus simultaneously and coordinate their contributions on a particular work item.

Organizational Bias. The preferences of an organization on a set of scales characterized by the following core values: exploration versus execution, speed versus stability, quantity versus quality, and flexibility versus predictability.

Organizational Change Management. A comprehensive, cyclic, and structured approach for transitioning individuals, groups, and organizations from the current state to a future state with intended business benefits.

Paint-Drip. See *Broken Comb*.

Pairing. See *Pair Work*.

Pair Programming. Pair work that is focused on programming.

Pair Work. A technique of pairing two team members to work simultaneously on the same work item.

Personas. An archetype user representing a set of similar end users described with their goals, motivations, and representative personal characteristics.

Pivot. A planned course correction designed to test a new hypothesis about the product or strategy.

Plan-Do-Check-Act (PDCA). An iterative management method used in

organizations to facilitate the control and continual improvement of processes and products.

Plan-Driven Approach. See *Predictive Approach*.

Predictive Approach. An approach to work management that utilizes a work plan and management of that work plan throughout the life cycle of a project.

Predictive Life Cycle. A more traditional approach, with the bulk of planning occurring up-front, then executing in a single pass; a sequential process.

Project Management Office (PMO). A management structure that standardizes the project-related governance processes and facilitates the sharing of resources, methodologies, tools, and techniques.

Product Backlog. An ordered list of user-centric requirements that a team maintains for a product.

Product Owner. A person responsible for maximizing the value of the product and who is ultimately responsible and accountable for the end product that is built. See also *Service Request Manager*.

Progressive Elaboration. The iterative process of increasing the level of detail in a project management plan as greater amounts of information and more accurate estimates become available.

Refactoring. A product quality technique whereby the design of a product is improved by enhancing its maintainability and other desired attributes without altering its expected behavior.

Retrospective. A regularly occurring workshop in which participants explore their work and results in order to improve both process and product.

Rolling Wave Planning. An iterative planning technique in which the work to be accomplished in the near term is planned in detail, while the work in the future is planned at a higher level.

Scaled Agile Framework (SAFe®). A knowledge base of integrated patterns for enterprise-scale lean–agile development.

Scrum. An agile framework for developing and sustaining complex products, with specific roles, events, and artifacts.

Scrumban. A management framework that emerges when teams employ Scrum as the chosen way of working and use the Kanban Method as a lens through

which to view, understand, and continuously improve how they work.

Scrum Board. An information radiator that is utilized to manage the product and sprint backlogs and show the flow of work and its bottlenecks.

Scrum Master. The coach of the development team and process owner in the Scrum framework. Removes obstacles, facilitates productive events and defends the team from disruptions. See also *Flow Master*.

Scrum of Scrums. A technique to operate Scrum at scale for multiple teams working on the same product, coordinating discussions of progress on their interdependencies, and focusing on how to integrate the delivery of software, especially in areas of overlap.

Scrum Team. Describes the combination of development team, scrum master, and process owner used in Scrum.

Self-Organizing Team. A cross-functional team in which people fluidly assume leadership as needed to achieve the team's objectives.

Servant Leadership. The practice of leading through service to the team, by focusing on understanding and addressing the needs and development of team members in order to enable the highest possible team performance.

Service Request Manager. The person responsible for ordering service requests to maximize value in a continuous flow or Kanban environment. Equivalent to product owner.

Siloed Organization. An organization structured in such a way that it only manages to contribute a subset of the aspects required for delivering value to customers. For contrast, see *Value Stream*.

Single Loop Learning. The practice of attempting to solve problems by just using specific predefined methods, without challenging the methods in light of experience.

Smoke Testing. The practice of using a lightweight set of tests to ensure that the most important functions of the system under development work as intended.

Specification by Example (SBE). A collaborative approach to defining requirements and business-oriented functional tests for software products based on capturing and illustrating requirements using realistic examples instead of abstract statements.

Spike. A short time interval within a project, usually of fixed length, during which a team conducts research or prototypes an aspect of a solution to prove its viability.

Sprint. Describes a timeboxed iteration in Scrum.

Sprint Backlog. A list of work items identified by the Scrum team to be completed during the Scrum sprint.

Sprint Planning. A collaborative event in Scrum in which the Scrum team plans the work for the current sprint.

Story Point. A unit-less measure used in relative user story estimation techniques.

Swarming. A technique in which multiple team members focus collectively on resolving a specific impediment.

Technical Debt. The deferred cost of work not done at an earlier point in the product life cycle.

Test-Driven Development. A technique where tests are defined before work is begun, so that work in progress is validated continuously, enabling work with a zero defect mindset.

Timebox. A fixed period of time, for example, 1 week, 1 fortnight, 3 weeks, or 1 month. See also *Iteration*.

T-shaped. Refers to a person with one deep area of specialization and broad ability in the rest of the skills required by the team. See also *I-Shaped* and *Broken Comb*.

User Story. A brief description of deliverable value for a specific user. It is a promise for a conversation to clarify details.

User Story Mapping. A visual practice for organizing work into a useful model to help understand the sets of high-value features to be created over time, identify omissions in the backlog, and effectively plan releases that deliver value to users.

UX Design. The process of enhancing the user experience by focusing on improving the usability and accessibility to be found in the interaction between the user and the product.

Value Stream. An organizational construct that focuses on the flow of value to

customers through the delivery of specific products or services.

Value Stream Mapping. A lean enterprise technique used to document, analyze, and improve the flow of information or materials required to produce a product or service for a customer.

LEARNING MADE EASY

3rd Edition

Agile Project Management

for dummies®

A Wiley Brand

Make performance be progress

Deliver customer value in weeks, not months

Turn agile principles into practice

Mark C. Layton
CST, PMP, MBA²

Steven J Ostermiller
CST, PMP, ICP-ACC

Dean J. Kynaston
CSP-SM, CSP-PO, MBA

Agile Project Management

for dummies®

A Wiley Brand

Agile Project Management

3rd Edition

by Mark C. Layton, Steven J Ostermiller, and Dean J. Kynaston

for dummies®
A Wiley Brand

Agile Project Management For Dummies®, 3rd Edition

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Copyright © 2020 by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. SAFe and Scaled Agile Framework are registered trademarks of Scaled Agile, Inc. Certified Scrum Developer, Certified Scrum Product Owner, Certified Scrum Professional, Certified Scrum Trainer, and Certified ScrumMaster are registered trademarks of Scrum Alliance. PMI Agile Certified Practitioner and PMI-ACP are registered trademarks of Project Management Institute, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit https://hub.wiley.com/community/support/dummies.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2020942690

ISBN 978-1-119-67699-7 (pbk); ISBN 978-1-119-67706-2 (ebk); ISBN 978-1-119-67705-5 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents at a Glance

Introduction 1

Part 1: Understanding Agility 5
CHAPTER 1: Modernizing Project Management 7
CHAPTER 2: Applying the Agile Manifesto and Principles 21
CHAPTER 3: Why Being Agile Works Better 49
CHAPTER 4: Agility Is about Being Customer Focused 69

Part 2: Being Agile 89
CHAPTER 5: Agile Approaches 91
CHAPTER 6: Agile Environments in Action 109
CHAPTER 7: Agile Behaviors in Action 123
CHAPTER 8: The Permanent Team 149

Part 3: Agile Planning and Execution 159
CHAPTER 9: Defining the Product Vision and Product Roadmap 161
CHAPTER 10: Planning Releases and Sprints 183
CHAPTER 11: Working throughout the Day 215
CHAPTER 12: Showcasing Work, Inspecting, and Adapting 239

Part 4: Agility Management 251
CHAPTER 13: Managing a Portfolio: Pursuing Value over Requirements 253
CHAPTER 14: Managing Scope and Procurement 269
CHAPTER 15: Managing Time and Cost 287
CHAPTER 16: Managing Team Dynamics and Communication 307
CHAPTER 17: Managing Quality and Risk 331

Part 5: Ensuring Success 355
CHAPTER 18: Building a Foundation 357
CHAPTER 19: De-Scaling across Teams 373
CHAPTER 20: Being a Change Agent 395

Part 6: The Part of Tens 421
CHAPTER 21: Ten Key Benefits of Agile Product Development 423
CHAPTER 22: Ten Key Factors for Agile Product Development Success 431
CHAPTER 23: Ten Signs That You're Not Agile 437
CHAPTER 24: Ten Valuable Resources for Agile Professionals 449

Index 455

Table of Contents

INTRODUCTION 1
About This Book. 1
Foolish Assumptions 1
Icons Used in This Book 2
Beyond the Book 2
Where to Go from Here 3

PART 1: UNDERSTANDING AGILITY 5

CHAPTER 1: Modernizing Project Management 7
Project Management Needed a Makeover 8
The origins of modern project management 8
The problem with the status quo 9
Introducing Agile Project Management 11
How agile projects work 14
Agile Project Management Is Becoming Agile Product Management 16
Differences between managing a project versus developing a product 16
Why agile product development works better 18

CHAPTER 2: Applying the Agile Manifesto and Principles 21
Understanding the Agile Manifesto 21
Outlining the Four Values of the Agile Manifesto 24
Value 1: Individuals and interactions over processes and tools 25
Value 2: Working software over comprehensive documentation 26
Value 3: Customer collaboration over contract negotiation 28
Value 4: Responding to change over following a plan 29
Defining the 12 Agile Principles 30
Agile principles of customer satisfaction 32
Agile principles of quality 34
Agile principles of teamwork 36
Agile principles of product development 38
Adding the Platinum Principles 42
Resisting formality 42
Thinking and acting as a team 43
Visualizing rather than writing 44
Changes as a Result of Agile Values 45
The Agile Litmus Test 47

CHAPTER 3: **Why Being Agile Works Better** 49
Evaluating Agile Benefits 49
How Agile Approaches Beat Historical Approaches 54
Greater flexibility and stability 55
Reduced nonproductive tasks 57
Higher quality, delivered faster 60
Improved team performance 61
Tighter control 62
Faster and less costly failure 63
Why People Like Being Agile 64
Executives 64
Product development and customers 65
Management 66
Development teams 67

CHAPTER 4: **Agility Is about Being Customer Focused** 69
Knowing Your Customers 69
Common methods for identifying your customer 71
Figuring Out the Problem Your Customer Needs to Solve 79
Using the scientific method 79
Failing early is a form of success 81
Defining customer-focused business goals 82
Story mapping 83
Liberating structures — simple rules to unleash a culture of innovation 83
Understanding Root Cause Analysis 84
Pareto rule 85
Five why's 86
Ishikawa (fishbone) 87

PART 2: BEING AGILE 89

CHAPTER 5: **Agile Approaches** 91
Diving under the Umbrella of Agile Approaches 91
Reviewing the Big Three: Lean, Scrum, and Extreme Programming 95
An overview of lean 95
An overview of scrum 100
An overview of extreme programming 105
Putting It All Together 107

CHAPTER 6: **Agile Environments in Action** 109
Creating the Physical Environment 110
Collocating the team 110
Setting up a dedicated area 112
Removing distractions 113

Low-Tech Communicating 114
High-Tech Communicating. 116
Choosing Tools. 118
The purpose of the tool 119
Tools that encourage the success of forced team dislocation . . . 119
Organizational and compatibility constraints 121

CHAPTER 7: **Agile Behaviors in Action** 123
Establishing Agile Roles 123
Product owner 124
Development team member 128
Scrum master. 130
Stakeholders. 132
Agile mentor. 134
Establishing New Values. 134
Commitment 135
Focus 136
Openness 137
Respect 138
Courage. 138
Changing Team Philosophy 139
Dedicated team 140
Cross-functionality. 141
Self-organization 143
Self-management 144
Size-limited teams 146
Ownership 147

CHAPTER 8: **The Permanent Team** 149
Enabling Long-Lived Product Development Teams 149
Leveraging long-term knowledge and capability 150
Navigating Tuckman's phases to performance. 151
Focusing on fundamentals. 153
Creating a working agreement 154
Enabling Autonomy, Mastery, and Purpose 155
Autonomy 155
Mastery 155
Purpose. 156
Highly aligned and highly autonomous teams 157
Building Team Knowledge and Capability 157

PART 3: AGILE PLANNING AND EXECUTION 159

CHAPTER 9: **Defining the Product Vision and Product Roadmap** 161
Agile Planning 162
Progressive elaboration 164
Inspect and adapt 165
Defining the Product Vision 165
Step 1: Developing the product objective 167
Step 2: Creating a draft vision statement 167
Step 3: Validating and revising the vision statement 169
Step 4: Finalizing the vision statement 170
Creating a Product Roadmap 171
Step 1: Identifying product stakeholders 172
Step 2: Establishing product requirements 173
Step 3: Arranging product features 175
Step 4: Estimating efforts and ordering requirements 176
Step 5: Determining high-level time frames 180
Saving your work 180
Completing the Product Backlog 180

CHAPTER 10: **Planning Releases and Sprints** 183
Refining Requirements and Estimates 183
What is a user story? 184
Steps to create a user story 186
Breaking down requirements 190
Estimation poker 192
Affinity estimating 195
Release Planning 197
Preparing for Release 200
Preparing the product for deployment 201
Prepare for operational support 201
Preparing the organization 203
Preparing the marketplace 204
Sprint Planning 205
The sprint backlog 206
The sprint planning meeting 207

CHAPTER 11: **Working throughout the Day** 215
Planning Your Day: The Daily Scrum 215
Tracking Progress 219
The sprint backlog 219
The task board 222

Agile Roles in the Sprint 224
Keys for daily product owner success 225
Keys for daily development team member success. 226
Keys for daily scrum master success 227
Keys for daily stakeholder success 228
Keys for daily agile mentor success 228
Creating Shippable Functionality 229
Elaborating 230
Developing 230
Verifying 231
Identifying roadblocks 234
Information Radiators. 235
The End of the Day 236

CHAPTER 12: **Showcasing Work, Inspecting, and Adapting** 239
The Sprint Review 239
Preparing to demonstrate 240
The sprint review meeting 241
Collecting feedback in the sprint review meeting. 244
The Sprint Retrospective 245
Planning for retrospectives 247
The retrospective meeting 248
Inspecting and adapting. 250

PART 4: AGILITY MANAGEMENT 251

CHAPTER 13: **Managing a Portfolio: Pursuing Value over Requirements** 253
Understanding the Differences in Agile Portfolio Management 254
Should we invest? 255
Factors for forecasting product investment returns 256
Managing Agile Product Portfolios 261
Should we continue investing? 266
Inspecting and adapting to the next opportunity. 267

CHAPTER 14: **Managing Scope and Procurement**. 269
What's Different about Agile Scope Management? 270
Managing Agile Scope. 272
Understanding scope throughout product development 273
Introducing scope changes 275
Managing scope changes 275
Using agile artifacts for scope management. 277
What's Different about Agile Procurement? 278
Managing Agile Procurement 280

Determining need and selecting a vendor. 280
Understanding cost approaches and contracts for services. 282
Working with a vendor 285
Closing a contract 286

CHAPTER 15: **Managing Time and Cost** 287
What's Different about Agile Time Management? 287
Managing Agile Schedules 289
Introducing velocity. 289
Monitoring and adjusting velocity. 291
Managing scope changes from a time perspective 297
Managing time by using multiple teams 298
Using agile artifacts for time management 298
What's Different about Agile Cost Management?. 299
Managing Agile Budgets. 300
Creating an initial budget. 301
Creating a self-funding product. 302
Using velocity to determine long-range costs. 303
Using agile artifacts for cost management 306

CHAPTER 16: **Managing Team Dynamics and Communication**. 307
What's Different about Agile Team Dynamics?. 307
Managing Team Dynamics. 309
Becoming self-managing and self-organizing 310
Supporting the team: The servant-leader 314
Working with a dedicated team. 316
Working with a cross-functional team 317
Reinforcing openness 319
Limiting development team size 320
Managing product development with dislocated teams 321
What's Different about Agile Communication?. 324
Managing Agile Communication 325
Understanding agile communication methods 325
Status and progress reporting. 328

CHAPTER 17: **Managing Quality and Risk** 331
What's Different about Agile Quality?. 331
Managing Agile Quality. 334
Quality and the sprint 335
Proactive quality 335
Quality through regular inspecting and adapting. 341
Automated testing. 342

What's Different about Agile Risk Management? 345
Managing Agile Risk 348
Reducing risk inherently 348
Identifying, prioritizing, and responding to risks early. 353

PART 5: ENSURING SUCCESS 355

CHAPTER 18: Building a Foundation 357
Organizational and Individual Commitment 357
Organizational commitment 358
Individual commitment 359
Getting commitment. 360
Can you make the transition? 361
Timing the transition. 362
Choosing the Right Pilot Team Members 363
The agile champion 363
The agile transition team 364
The product owner 365
The development team 366
The scrum master 366
The stakeholders 367
The agile mentor 367
Creating an Environment That Enables Agility 368
Support Agility Initially and Over Time 371

CHAPTER 19: De-Scaling across Teams 373
Multi-Team Agile Development 374
Making Work Digestible through Vertical Slicing 376
Scrum of scrums 376
Multi-Team Coordination with LeSS 380
LeSS, the smaller framework 380
LeSS Huge framework. 381
Sprint review bazaar 382
Observers at the daily scrum. 383
Component communities and mentors. 383
Multi-team meetings. 383
Travelers 384
Aligning through Roles with Scrum@Scale 384
The scrum master cycle 385
The product owner cycle 387
Synchronizing in one hour a day. 388
Joint Program Planning with SAFe. 388
Joint program increment planning 391
Clarity for managers 392
Disciplined Agile Toolkit 392

CHAPTER 20: **Being a Change Agent** 395
Becoming Agile Requires Change 395
Why Change Doesn't Happen on Its Own 396
Strategic Approaches to Implementing and Managing Change 397
Lewin 398
ADKAR's five steps to change 399
Kotter's eight steps for leading change 400
Platinum Edge's Change Roadmap 401
Step 1: Conduct an agile audit to define an implementation strategy with success metrics 403
Step 2: Build awareness and excitement 404
Step 3: Form a transformation team and identify a pilot 405
Step 4: Build an environment for success 407
Step 5: Train sufficiently and recruit as needed 408
Step 6: Kick off the pilot with active coaching 408
Step 7: Execute the Roadmap to Value 410
Step 8: Gather feedback and improve 410
Step 9: Mature and solidify improvements 411
Step 10: Progressively expand within the organization 412
Leading by Example 412
The role of a servant-leader in an agile organization 413
Keys for successful servant leadership 413
Avoiding Transformation Pitfalls 414
Avoiding agile leadership pitfalls 417
Signs Your Changes Are Slipping 418

PART 6: THE PART OF TENS 421

CHAPTER 21: **Ten Key Benefits of Agile Product Development** 423
Higher Customer Satisfaction 423
Better Product Quality 424
Reduced Risk 425
Increased Collaboration and Ownership 426
More Relevant Metrics 426
Improved Performance Visibility 427
Increased Investment Control 428
Improved Predictability 429
Optimized Team Structures 429
Higher Team Morale 430

CHAPTER 22: **Ten Key Factors for Agile Product Development Success** 431
Dedicated Team Members 431
Collocation 432
Done Means Shippable 433
Address What Scrum Exposes 433
Clear Product Vision and Roadmap 433
Product Owner Empowerment 434
Developer Versatility 434
Scrum Master Clout 435
Leadership Support for Learning 435
Transition Support 436

CHAPTER 23: **Ten Signs That You're Not Agile** 437
A Non-Shippable Sprint Product Increment 437
Long Release Cycles 438
Disengaged Stakeholders 439
Lack of Customer Contact 440
Lack of Skill Versatility 441
Automatable Processes Remain Manual 442
Prioritizing Tools over the Work 442
High Manager-to-Creator Ratio 444
Working around What Scrum Exposes 445
Practicing Faux Agile 446

CHAPTER 24: **Ten Valuable Resources for Agile Professionals** 449
Agile Project Management For Dummies Online Cheat Sheet 449
Scrum For Dummies 450
The Scrum Alliance 450
The Agile Alliance 450
International Consortium for Agile (ICAgile) 451
Mind the Product and ProductTank 451
Lean Enterprise Institute 451
Extreme Programming 452
The Project Management Institute Agile Community 452
Platinum Edge 452

INDEX 455

Introduction

Welcome to *Agile Project Management For Dummies,* 3rd Edition. Agile project management has grown to be as common as any management technique for product development — and not only software product development. For nearly two decades, we have trained and coached companies big and small, all over the world, about how to become more nimble, adaptive, and responsive in both the development of their products and their organizations — in other words, how to become more agile. Through this work, we found there was a need to write a digestible guide that anyone, regardless of experience, could understand.

About This Book

Agile Project Management For Dummies, 3rd Edition is more than just an introduction to agile practices and approaches; you also discover the steps to become more agile in mindset and behavior. The material here goes beyond theory and is meant to be a field guide for all experience levels, giving you the tools and information you need to be successful with agile techniques in the trenches of product development.

Foolish Assumptions

This book was written as a reference guide for anyone wanting to learn more about business agility. If you strive to be more agile in responding to customer needs and problems — whether or not you're an organizational leader, a project manager, a member of a product team, an agile enthusiast, or a product stakeholder — this book will help you on your journey.

Regardless of your experience or level of familiarity, this book provides insights you may find helpful. We hope it brings clarity to any confusion or myths regarding agile product development you may have encountered.

Icons Used in This Book

Throughout this book, you'll find the following icons.

TIP

Tips are points to help you along your agile product development journey. Tips can save you time and help you quickly understand a particular topic, so when you see them, take a look!

REMEMBER

The Remember icon is a reminder of something you may have seen in past chapters. It also may be a reminder of a commonsense principle that is easily forgotten. These icons can help jog your memory when an important term or concept appears.

WARNING

The Warning icon indicates that you want to watch out for a certain action or behavior. Read these to steer clear of big problems!

TECHNICAL STUFF

The Technical Stuff icon indicates information that is interesting but not essential to the text. If you see a Technical Stuff icon, you don't need to read it to understand agile product development, but the information there might just pique your interest.

ON THE WEB

On the Web means that you can find more information on the book's website at `www.dummies.com/go/agileprojectmanagementfd3e`.

Beyond the Book

Although this book broadly covers the agile project management spectrum, we can cover only so much in a set number of pages! If you find yourself at the end of this book thinking, "This was an amazing book! Where can I learn more about how to advance my products under an agile approach?" check out Chapter 24 or head over to `www.dummies.com` for more resources.

We've provided a cheat sheet for tips on assessing your current product development efforts in relation to agile principles as well as free tools for managing projects using agile techniques. To get to the cheat sheet, go to `www.dummies.com`, and then type *Agile Project Management For Dummies Cheat Sheet* in the Search box. This is also where you'll find any significant updates or changes that occur between editions of this book.

Where to Go from Here

We wrote this book so that you could read it in just about any order. Depending on your role, you may want to pay extra attention to the appropriate sections of the book. For example:

- If you're just starting to learn about product development and agile approaches, start with Chapter 1 and read the book straight through to the end.
- If you're a member of a product team and want to know the basics of agile product development, check out the information in Part 3 (Chapters 9 through 12).
- If you're a project manager transitioning to agile approaches to product development, you may be interested to learn how agile techniques improve the management of time, cost, scope, procurement, quality, and risk. Review Part 4 (Chapters 13 through 17).
- If you know the basics of agile product development and are looking at bringing agile practices to your company or expanding your agile footprint across your organization, Part 5 (Chapters 18 through 20) will provide you with helpful information.

1 Understanding Agility

IN THIS PART . . .

Understand why project management has modernized due to the flaws and weaknesses in historical approaches to project management.

Find out why agile methods are becoming more product-focused than project-focused, and become acquainted with the foundation of agile product development: the Agile Manifesto and the 12 Agile Principles.

Discover the advantages that your products, projects, teams, customers, and organization can gain from adopting agile techniques.

Understand the importance of placing the customer's needs first and why agile techniques help to make the customer central to every decision, functionality, and problem.

IN THIS CHAPTER

- Understanding why project management needs to change
- Seeing how agile project management is becoming agile product management
- Finding out about agile product development

Chapter 1

Modernizing Project Management

Agile is a descriptor of a mindset approach to project management that focuses on early delivery of business value, continuous improvement of the product being created and the processes used to create the product, scope flexibility, team input, and delivering well-tested products that reflect customer needs.

In this chapter, you find out why agile processes emerged as an approach to software development project management in the mid-1990s and why agile methodologies have caught the attention of project managers, customers who invest in the development of new products and services, and executives whose companies fund product development. While business agility is popular in software product development, agile values, principles, and techniques apply in a multitude of industries and applications — not just software. This chapter also explains the advantages of agile approaches over long-standing project management methodologies.

Project Management Needed a Makeover

A *project* is a planned program of work that requires a definitive amount of time, effort, and planning to complete. Projects have goals and objectives and often must be completed in some fixed period of time and within a certain budget.

Because you're reading this book, you're likely a project manager or someone who initiates projects, works on projects, or is affected by projects in some way.

Agile approaches are a response to the need to modernize project management. To understand how agile approaches are revolutionizing product development, it helps to know a little about the history and purpose of project management and the issues that projects face today.

The origins of modern project management

Projects have been around since ancient times. From the Great Wall of China to the Mayan pyramids at Tikal, from the invention of the printing press to the invention of the Internet, people have accomplished endeavors big and small in projects.

As a formal discipline, project management as we know it has been around only since the middle of the twentieth century. Around the time of World War II, researchers around the world were making major advances in building and programming computers, mostly for the United States military. To complete those projects, they started creating formal project management processes. The first processes were based on step-by-step manufacturing models the United States military used during World War II.

People in the computing field adopted these step-based manufacturing processes because early computer-related projects relied heavily on hardware, with computers that filled up entire rooms. Software, by contrast, was a smaller part of computer projects. In the 1940s and 1950s, computers might have thousands of physical vacuum tubes but fewer than 30 lines of programming code. The 1940s manufacturing process used on these initial computers is the foundation of the project management methodology known as waterfall.

In 1970, a computer scientist named Winston Royce wrote "Managing the Development of Large Software Systems," an article for the IEEE that described the phases in the waterfall methodology. The term *waterfall* was coined later, but the phases, even if they are sometimes titled differently, are essentially the same as originally defined by Royce:

1. Requirements
2. Design
3. Development
4. Integration
5. Testing
6. Deployment

On waterfall projects, you move to the next phase only when the prior one is complete — hence the name waterfall.

TECHNICAL STUFF

Pure waterfall project management — completing each step in full before moving to the next step — is actually a misinterpretation of Royce's suggestions. Royce identified that this approach was inherently risky and recommended developing and testing within iterations to create products — suggestions that were overlooked by many organizations that adopted the waterfall methodology.

The waterfall methodology was the most common project management approach in software development until it was surpassed by improved approaches based on agile techniques around 2008.

The problem with the status quo

Computer technology has, of course, changed a great deal since the last century. Many people have a computer on their wrist with more power, memory, and capabilities than the largest, most expensive machine that existed when people first started using waterfall methodologies.

At the same time, the people using computers have changed as well. Instead of creating behemoth machines with minimal programs for a few researchers and the military, people create hardware and software for the general public. In many countries, almost everyone uses a tablet or smartphone, directly or indirectly, every day. Software runs our cars, our appliances, our homes; it provides our daily information and daily entertainment. Even young children use computers —2-year-olds are almost more adept with the iPhone than their parents. The demand for newer, better products is constant.

Somehow, during all this growth of technology, processes were not left behind. Software developers are still using project management methodologies from the 1950s, and all these approaches were derived from manufacturing processes meant for the hardware-heavy computers of the mid-twentieth century.

Today, traditional projects that do succeed often suffer from one problem: *scope bloat,* the introduction of unnecessary product features. Think about the software products you use every day. For example, the word-processing program we're typing on right now has many features and tools. Even though we write with this program every day, we use only some of the features all the time. We use other elements less frequently. And we have never used quite a few tools — and come to think of it, we don't know anyone else who has used them, either. The features that few people use are the result of scope bloat.

Scope bloat appears in all kinds of software, from complex enterprise applications to websites that everyone uses. Figure 1-1 shows data from a Standish Group study that illustrates just how common scope bloat is. In the figure, you can see that 80 percent of requested features are infrequently or never used.

FIGURE 1-1: Actual use of requested software features.

© Copyright 2017 Standish Group

The numbers in Figure 1-1 illustrate an enormous waste of time and money. That waste is a direct result of traditional project management processes that are unable to accommodate change. Project managers and stakeholders know that change is not welcome mid-project, so their best chance of getting a potentially desirable feature is at the start of a project. Therefore, they ask for

- Everything they need
- Everything they think they may need
- Everything they want
- Everything they think they may want

The result is the bloat in features that results in the statistics in Figure 1-1.

SOFTWARE PROJECT SUCCESS AND FAILURE

Stagnation in traditional project management approaches is catching up with the software industry. In 2015, a software statistical company called the Standish Group did a study on the success and failure rates of 10,000 projects in the US. The results of the study showed that

- *29 percent of traditional projects failed outright.* The projects were cancelled before they finished and did not result in any product releases. These projects delivered no value whatsoever.
- *60 percent of traditional projects were challenged.* The projects were completed but had gaps between expected and actual cost, time, quality, or a combination of these elements. The average difference between the expected and actual project results — looking at time, cost, and features not delivered — was well over 100 percent.
- *11 percent of projects succeeded.* The projects were completed and delivered the expected product in the originally expected time and budget.

Of the hundreds of billions of dollars spent on product development in the US alone, billions of dollars were wasted on projects that never deployed a single piece of functionality.

The problems associated with using outdated management and development approaches are not trivial. These problems waste billions of dollars a year. The billions of dollars lost in project failure in 2015 (see the sidebar, "Software project success and failure") could equate to millions of jobs around the world.

Over the past three decades, people working on projects have recognized the growing problems with traditional project management and have been working to create a better model.

Introducing Agile Project Management

The seeds for agile techniques have been around for a long time. In fact, agile values, principles, and practices are simply a codification of common sense. Figure 1-2 shows a quick history of agile project management, dating to the 1930s with Walter Sherwart's Plan-Do-Study-Act (PDSA) approach to project quality.

FIGURE 1-2: Agile project management timeline.

In 1986, Hirotaka Takeuchi and Ikujiro Nonaka published an article called "The New New Product Development Game" in the *Harvard Business Review.* Takeuchi and Nonaka's article described a rapid, flexible development strategy to meet fast-paced product demands. This article first paired the term *scrum* with product development. (*Scrum* referred to a player formation in rugby.) Scrum eventually became one of the most popular agile frameworks for delivering value to customers.

In 2001, a group of software and project experts got together to talk about what their successful projects had in common. This group created the *Manifesto for Agile Software Development* (commonly referred to as the Agile Manifesto), a statement of values for successful software development:

> Manifesto for Agile Software Development*
>
> 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.
>
> ** Agile Manifesto Copyright © 2001: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.*
>
> *This declaration may be freely copied in any form, but only in its entirety through this notice.*

These experts also created 12 *principles behind the Agile Manifesto* that help support the values in the Agile Manifesto. We list the Agile Principles and describe the Agile Manifesto in more detail in Chapter 2.

Agile, in product development terms, is a descriptor for approaches that focus on people, communications, the product, and flexibility. If you're looking for *the* agile methodology, you won't find it. However, all agile methodologies (for example, Crystal), frameworks (for example, scrum), techniques (for example, user story requirements), and tools (for example, relative estimating) have one thing in common: adherence to the Agile Manifesto and the 12 Agile Principles.

TIP

Martin Fowler, one of the co-authors of the Agile Manifesto, writes that many different words were discussed for naming their movement. They considered *lightweight methods*, *adaptive*, and many others until landing on *agile* as the best descriptor of the adaptiveness and responsiveness to change they were seeking.

Other synonyms are resilient, nimble, and healthy. When you think of agile, think healthy. Healthy organizations and teams are agile, resilient, nimble, and responsive.

How agile projects work

Agile approaches are based on an *empirical control method* — a process of making decisions based on the realities observed in the project. In the context of software development methodologies, an empirical approach can be effective in both new product development and enhancement and upgrade projects. By using frequent and firsthand inspection of the work to date, you can make immediate adjustments, if necessary. Empirical control requires

- **Unfettered transparency:** Everyone involved in an agile project knows what is going on and how the project is progressing.
- **Frequent inspection:** The people who are invested in the product and process the most regularly evaluate the product and process.
- **Immediate adaptation:** Adjustments are made quickly to minimize problems; if an inspection shows that something should change, it is changed immediately.

To accommodate frequent inspection and immediate adaptation, agile projects work in *iterations* (smaller segments of the overall project). An agile product development effort involves the same type of work as in a traditional waterfall project: You create requirements and designs, develop product features, document what was done and why, and continuously integrate new features. You test the product, fix any problems, and deploy the product for use. However, instead of completing these steps for all product features at once, as in a waterfall project, you break the project into iterations, also called *sprints.*

Figure 1-3 shows the difference between a linear waterfall project and an agile project.

WARNING

Mixing traditional project management methods with agile approaches is like saying, "I have a Tesla Model S. However, I'm using a wagon wheel on the front left side. How can I make my car as fast and efficient as the other Teslas?" The answer, of course, is you can't. If you fully commit to an agile approach, you will have a better chance of project success.

FIGURE 1-3: Waterfall versus agile project.

Agile Project Management Is Becoming Agile Product Management

Traditional projects, by definition, are organized into temporary, build-only team efforts designed to accomplish specific benefits projected in a business case. Project initiation, which is when we know the least, is when budgets, schedules, and expectations are set. When projects end, the teams are disbanded and unfamiliar operational teams are left to support the customer and product. If more work is required, new team members are allocated to a new project and must refamiliarize themselves with the product architecture. Projects typically end once deliverables are released into production, leaving others to support and evaluate the effect on business.

Today, products are considered long-term, value-creating assets requiring permanent teams who iteratively elaborate, design, develop, test, integrate, document, and even support products until business outcomes are achieved. A high-performing team continuously inspects and adapts until the customer's problems are solved, and the team retains this hard-earned knowledge. The team and customer collaborate to create value over following written specifications.

More and more we see a project-driven approach as a constraint to delivering customer value early and often. Taking an agile approach to product development limits you not to time or money but to value. Organizations most effectively deliver value when they use the following formula to determine when it's time to shift priorities:

AC + OC > V, that is, actual cost + opportunity cost > value

When the actual cost of working on a product's additional requirements plus the opportunity cost of not working on a different investment opportunity exceed the expected value of delivering those remaining product requirements, the team shifts to developing the more valuable investment opportunity.

Differences between managing a project versus developing a product

Three primary differences exist between managing a project and developing a product:

- Products benefit most from stable, long-lived, and even permanent teams.
- Products can be not only short-term assets but also long-term assets. Active products are never really finished because they require maintenance and improvements.
- Products are part of a portfolio designed to maximize value over following specifications.

Permanent team over temporary team

Long-lived products are best developed and maintained by long-lived and even permanent teams. The longer a team works together iteratively building an emergent architecture and expanding its capability and high performance, the better the team understands the customer and the more predictable the team becomes. Project-focused teams come together for a specific amount of time and then move on to something new. Lessons learned at the end of a project may not even apply to the next project because the context of people, technologies, and customers will most likely be different. Stable permanent teams enable transparency, inspection, and adaptation (empirical process control).

TIP

Permanent doesn't mean that agile product teams don't change and career aspirations are inhibited. However, team personnel changes are an exception rather than the rule. People — especially people who become more valuable because of their expanding capability — are presented with career-enhancing opportunities. Ideally, permanent team members behave more like a family than as a temporary, one-project-only group.

Products as long-term assets rather than project deliverables

Product development is risky. Uncertainty abounds at every corner. But uncertainty is what makes agile product development ideal! Traditional projects are tasked with accomplishing specific system deliverables within a fixed time frame, but agile product development iteratively reduces uncertainty by building useable, fully functional product increments, gathering and implementing feedback throughout development. The product becomes a customer-aligned, problem-solving asset. Active products are never finished because maintenance must be performed and improvements can be made.

Investments in time, money, and people, particularly with today's capital expenditure strategies, change products into depreciable assets that improve bottom-line results for not only revenue but also cost savings. Treating product

development as an asset creator rather than a cost expenditure changes the perspective of everyone involved. Continuous delivery of customer value through agile product development increases the likelihood of additional funding.

TECHNICAL STUFF

Capital expenditures, commonly known as *CapEx,* are funds used by a company to acquire, upgrade, and maintain physical assets such as property, buildings, industrial plants, technology, and equipment. CapEx is often used to undertake new projects or investments by the firm.

Pursuing value over specifications

Early failure is a key tenant of agility. Agile teams are obsessed with taking risks to create customer value. Like scientists, they create a hypothesis, test it in the real world, evaluate the results, and then adjust the hypothesis and test it again. They repeat this process again and again, aligning the product more closely to the customer's needs with each iteration. Teams trade reams of documented specifications for real-world feedback from the customer. With agile product development, functionality priorities are set by the people most familiar with the problem to be solved.

Why agile product development works better

Throughout this book, you see how product development with an agile approach works better than with a traditional approach. Agile approaches can produce more successful products. The Standish Group study, mentioned in the sidebar "Software project success and failure," found that while 29 percent of traditional projects failed outright, that number dropped to only 9 percent with agile techniques. The decrease in failure for agile product development is a result of agile teams making immediate adaptations based on frequent inspections of progress and customer satisfaction.

Here are some key areas where agile product development approaches are superior to traditional project management methods:

- **Project success rates:** In Chapter 17, you find out how the risk of catastrophic project failure falls to almost nothing with agile product development. Agile approaches of prioritizing by business value and risk ensure early success or failure. Agile approaches to testing throughout the product development help ensure that you find problems early, not after spending a large amount of time and money.

- **Scope creep:** In Chapters 9, 10, and 14, you see how agile approaches accommodate changes throughout product development, minimizing scope creep. Following agile principles, you can add new requirements at the beginning of each sprint without disrupting development flow. By fully developing prioritized features first, you prevent scope creep from threatening critical functionality.
- **Inspection and adaptation:** In Chapters 12 and 16, you find details of how regular inspections and adaptation work throughout agile product development. Agile teams — armed with frequent feedback from complete development cycles and working, shippable functionality — can improve their processes and their products with each sprint.

Throughout many chapters in this book, you discover how business agility can help you gain control of product outcomes. Testing early and often, adjusting priorities as needed, using better communication techniques, and regularly demonstrating and releasing product functionality allow you to fine-tune your control over a wide variety of factors.

IN THIS CHAPTER

- Defining the Agile Manifesto and the 12 Agile Principles
- Describing the Platinum Principles
- Understanding what has changed in project management
- Taking the agile litmus test

Chapter 2

Applying the Agile Manifesto and Principles

This chapter describes the basics of what it means to be agile as outlined in the Agile Manifesto, with its four values, and the 12 Agile Principles behind the Agile Manifesto. We also expand on these basics with three additional Platinum principles, which Platinum Edge (owned by Mark) crafted after years of experience helping organizations improve their business agility.

This foundation provides product development teams with the information needed to evaluate whether the team is following agile principles, as well as whether its actions and behaviors are consistent with agile values. When you understand these values and principles, you'll be able to ask, "Is this agile?" and be confident in your answer. To learn more about development teams, see Chapter 7.

Understanding the Agile Manifesto

In the mid-1990s, the Internet was changing the world right before our eyes. The people working in the booming dot-com industry were under constant pressure to be the first-to-market with fast-changing technologies. Development teams worked day and night, struggling to deliver new software releases before competitors made their companies obsolete. The information technology (IT) industry was completely reinvented in a few short years.

Given the pace of change at that time, cracks inevitably appeared in conventional project management practices. Using traditional methodologies such as waterfall, which is discussed in Chapter 1, didn't allow developers to be responsive enough to the market's dynamic nature and to emerging new approaches to business. Development teams started exploring alternatives to these outdated approaches to project management. In doing so, they noticed some common themes that produced better results.

In February 2001, 17 of these new methodology pioneers met in Snowbird, Utah, to share their experiences, ideas, and practices; to discuss how best to express them; and to suggest ways to improve the world of software development. They couldn't have imagined the effect their meeting would have on the future of project management. The simplicity and clarity of the manifesto they produced and the subsequent principles they developed transformed the world of information technology and continue to revolutionize product development in every industry, not just software.

Over the next several months, these leaders constructed the following:

- **The Agile Manifesto (originally the Manifesto for Agile Software Development):** An intentionally streamlined expression of core development values
- **The Agile Principles:** A set of 12 guiding concepts that support product development teams in delivering value and staying on track
- **The Agile Alliance:** A community development organization focused on supporting individuals and organizations applying agile principles and practices

The group's work was destined to make the software industry more productive, more humane, and more sustainable.

The Agile Manifesto is a powerful statement, carefully crafted using fewer than 75 words:

> **Manifesto for Agile Software Development**
>
> 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.

* *Agile Manifesto Copyright © 2001: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas*

This declaration may be freely copied in any form, but only in its entirety through this notice.

No one can deny that the Agile Manifesto is both a concise and an authoritative statement. Whereas traditional approaches emphasize a rigid plan, avoid change, document everything, and encourage hierarchical-based control, the manifesto focuses on

- People
- Communications
- Product
- Flexibility

The Agile Manifesto represents a big shift in focus in how products are conceived, conducted, and managed. If we read only the items on the left, we understand the new paradigm that the manifesto signers envisioned. They found that by focusing more attention on individuals and interactions, teams would more effectively produce working software through valuable customer collaboration and by responding well to change. In contrast, the traditional primary focus on processes and tools often produces comprehensive or excess documentation to comply with contract negotiations and to follow an unchanging plan.

Research and experience illustrate why agile values are so important:

- **Individuals and interactions over processes and tools:** Why? Because research shows a 50 times increase in performance when we get individuals and interactions right. One of the ways to get this right is by collocating a development team with an empowered product owner.
- **Working software over comprehensive documentation:** Why? Because failure to test for and correct defects during the sprint can take up to 24 times more effort and cost in the next sprint. And after the functionality is deployed to the market, if a production support team that wasn't involved in product development performs the testing and fixing, the cost is up to 100 times more.
- **Customer collaboration over contract negotiation:** Why? Because a dedicated and accessible product owner can generate a fourfold increase in

productivity by providing in-the-moment clarification to the development team, aligning customer priorities with the work being performed.

- **Responding to change over following a plan:** Why? Because 80 percent of features developed under a waterfall model are infrequently or never used (as discussed in Chapter 1). Starting with a plan is vital, but the start is when we know the least. Product development teams don't plan less than waterfall teams — they plan as much or more. However, teams take a just-in-time approach, planning just enough when needed in support of a strategic product vision and roadmap. Adaptation of the plan to the realities along the way is how teams avoid wasteful functionality and deliver products that delight customers.

The creators of the Agile Manifesto originally focused on software development because they worked in the IT industry. However, agile techniques have spread beyond software development and even outside computer-related products. Today, agile approaches such as scrum are disrupting biotech, manufacturing, aerospace, engineering, marketing, building construction, finance, shipping, automotive, utility, and energy industries with companies such as Apple, Microsoft, and Amazon leading the way. If you want early empirical feedback on the product or service you're providing, you can benefit from agile methods.

The *State of Scrum 2017-2018* report quoted a Scrum Alliance board member who said, "Any organization that does not go through an Agile transformation will die. It is the same as a company refusing to use computers."

REMEMBER

The Agile Manifesto and Agile Principles directly refer to software; we leave these references intact when quoting the manifesto and principles throughout the book. If you create non-software products, try substituting your product as you read on. Agile values and principles apply to all product development activities, not just software.

Outlining the Four Values of the Agile Manifesto

The Agile Manifesto was generated from experience, not from theory. As you review the values described in the following sections, consider what they would mean if you put them into practice. How do these values support meeting time-to-market goals, dealing with change, and valuing human innovation?

Although the agile values and principles are not numbered, we've numbered them here and throughout the book for ease of reference. The numbering matches their order in the manifesto.

Value 1: Individuals and interactions over processes and tools

When you allow each person to contribute his or her unique value to a product, the result can be powerful. When these human interactions focus on solving problems, a unified purpose can emerge. Moreover, the agreements come about through processes and tools that are much simpler than conventional ones.

A simple conversation in which you talk through a product issue can solve many problems in a relatively short time. Trying to emulate the power of a direct conversation with email, spreadsheets, and documents results in significant overhead costs and delays. Instead of adding clarity, these types of managed, controlled communications are often ambiguous and time-consuming and distract the development team from the work of creating a product.

Consider what it means if you value individuals and interactions highly. Table 2-1 shows some differences between valuing individuals and interactions and valuing processes and tools.

ON THE WEB

You can find a blank template of Table 2-1 on the book's companion website at `http://www.dummies.com/go/agileprojectmanagementfd3e`. Jot down the pros and cons of each approach that apply to you and your products.

REMEMBER

If processes and tools are seen as the way to manage product development and everything associated with it, people and the way they approach the work must conform to the processes and tools. Conformity makes it hard to accommodate new ideas, new requirements, and new thinking. Agile approaches, however, value people over process. This emphasis on individuals and teams puts the focus on their energy, innovation, and ability to solve problems. You use processes and tools in agile product management, but they're intentionally streamlined and directly support product creation. The more robust a process or tool, the more you spend on its care and feeding and the more you defer to it. With people front and center, however, the result is a leap in productivity. An agile environment is human-centric and participatory and can be readily adapted to new ideas and innovations.

TABLE 2-1 **Individuals and Interactions versus Processes and Tools**

	Individuals and Interactions Have High Value	Processes and Tools Have High Value
Pros	Communication is clear and effective. Communication is quick and efficient. Teamwork becomes strong as people work together. Development teams can self-organize. Development teams have more chances to innovate. Development teams can quickly adjust processes as necessary. Development team members can take personal ownership of the product. Development team members can have deeper job satisfaction.	Processes are clear and can be easy to follow. Written records of communication exist.
Cons	To enable more team empowerment and less command and control, managers may have to unlearn traditional leadership tendencies. People may need to let go of ego to work well as members of a team.	People may over-rely on processes instead of finding the best ways to create good products. One process doesn't fit all teams — different people have different work styles. One process doesn't fit all products. Communication can be ambiguous and time-consuming.

Value 2: Working software over comprehensive documentation

A development team's focus should be on producing working functionality. With agile development, the only way to measure whether you are truly finished with a product requirement is to produce the working functionality associated with that requirement. For software products, working software means the software meets what we call the *definition of done:* at the very least, developed, tested, integrated, and documented. After all, the working product is the reason for the investment.

Have you ever been in a status meeting where you reported that you were, say, 75 percent done with your project? What would happen if your customer told you, "We ran out of money. Can we have our 75 percent now?" On a traditional project, you wouldn't have any working software to give the customer; 75 percent done traditionally means you are 75 percent in progress and 0 percent done. With

agile product development, however, by using the definition of done, you would have working, potentially shippable functionality for 75 percent of your product requirements — the highest-priority 75 percent of requirements.

REMEMBER

Although agile approaches have roots in software development, you can use them for other types of products. This second agile value can easily read, "Working *functionality* over comprehensive documentation."

Tasks that distract from producing valuable functionality must be evaluated to see whether they support or undermine the job of creating a working product. Table 2-2 shows a few examples of traditional project documents and their usefulness. Think about whether documents produced on a recent project you were involved in added value to the functionality being delivered to your customer.

REMEMBER

With agile product development, *barely sufficient* is a positive description, meaning that a task, document, meeting, or almost anything created includes only what it needs to achieve the goal. Being barely sufficient is practical and efficient — it's sufficient, just enough. The opposite of barely sufficient is *gold-plating*, or adding unnecessary frivolity — and effort — to a feature, task, document, meeting, or anything else.

All development requires some documentation. With agile product development, documents are useful only if they support development and are barely sufficient to serve the design, delivery, and deployment of a working product in the most direct, unceremonious way. Agile approaches dramatically simplify the administrative paperwork relating to time, cost control, scope control, or reporting.

ON THE WEB

You can find a blank template of Table 2-2 at `www.dummies.com/go/agileprojectmanagementfd3e`. Use that form to assess how well your documentation directly contributed to the product and whether it was barely sufficient.

TIP

We'll often stop producing a document and see who complains. After we know the requester of the document, we'll strive to better understand why the document is necessary. The *five whys* work great in this situation — ask "why" after each successive answer to get to the root reason for the document. After you know the core reason for the document, see how you can satisfy that need with an agile artifact or streamlined process.

Product development teams produce fewer, more streamlined documents that take less time to maintain and provide better visibility into potential issues. In the coming chapters, you find out how to create and use simple tools (such as a product backlog, a sprint backlog, and a task board) that allow teams to understand requirements and assess real-time status daily. With agile approaches, teams spend more time on development and less time on documentation, resulting in a more efficient delivery of a working product. See Chapter 9 for more on the product backlog, and see Chapter 10 to learn more about the sprint backlog.

TABLE 2-2 Identifying Useful Documentation

Document	Does the Document Add to Product Value?	Is the Document Barely Sufficient or Gold-Plated?
Project schedule created with expensive project management software, complete with Gantt chart	No. Start-to-finish schedules with detailed tasks and dates tend to provide more than what is necessary for product development. Also, many of these details change before you develop future features.	Gold-plated. Although project managers may spend a lot of time creating and updating project schedules, team members tend to want to know only key deliverable dates. Management often wants to know only whether the deliverable is on time, ahead of schedule, or behind.
Requirements documentation	Yes. All products have requirements — details about product features and needs. Development teams need to know those needs to create a product.	Possibly gold-plated; should be barely sufficient. Requirements documents can easily grow to include unnecessary details. Agile approaches provide simple ways to enable product requirement conversations.
Product technical specifications	Yes. Documenting how you created a product can make future changes easier.	Possibly gold-plated; should be barely sufficient. Agile documentation includes just what it needs — development teams often don't have time for extra flourishes and are keen to minimize documentation.
Weekly status report	No. Weekly status reports are for management purposes but do not assist product creation.	Gold-plated. Knowing status is helpful, but traditional status reports contain outdated information and are much more burdensome than necessary.
Detailed project communication plan	No. Although a contact list can be helpful, the details in many communication plans are useless to product development teams.	Gold-plated. Communication plans often end up being documents about documentation — an egregious example of busywork.

Value 3: Customer collaboration over contract negotiation

The customer is not the enemy. Really.

Historical project management approaches usually limit customer involvement to a few development stages:

- **Start of a project:** When the customer and the project team negotiate contract details.
- **Any time the scope changes during the project:** When the customer and the project team negotiate changes to the contract.
- **End of a project:** When the project team delivers a completed product to the customer. If the product doesn't meet the customer's expectations, the project team and the customer negotiate additional changes to the contract.

This historical focus on negotiation, avoidance of scope change, and limitation of direct customer involvement discourages potentially valuable customer input and can even create an adversarial relationship between customers and project teams.

WARNING

You will never know less about a product than at its start. Locking product details into a contract at the beginning of development means you have to make decisions based on incomplete knowledge. If you have flexibility for change as you learn more about a product and the customer the product is serving, you'll ultimately create better products.

The agile pioneers understood that collaboration, rather than confrontation, produced better, leaner, more useful products. As a result of this understanding, agile methods make the customer part of the product development on an ongoing basis.

Using an agile approach in practice, you'll experience a partnership between the customer and the product development team in which discovery, questioning, learning, and adjusting during the course of development are routine, acceptable, and systematic. This partnership results in superior products better suited for the customer's needs.

Value 4: Responding to change over following a plan

Change is a valuable tool for creating great products. Teams that can respond quickly to customers, product users, and the market are able to develop relevant, helpful products that people want to use.

Unfortunately, traditional project management approaches attempt to wrestle the change monster and pin it to the ground so it goes out for the count. Rigorous change management procedures and budget structures that can't accommodate new product requirements make changes difficult. Traditional project teams often find themselves blindly following a plan, missing opportunities to create more valuable products or, even worse, unable to react timely to changing market conditions.

Figure 2-1 shows the relationship between time, opportunity for change, and the cost of change on a traditional project. As time — and knowledge about your product — increases, the ability to make changes decrease, and costs more.

FIGURE 2-1: Traditional project opportunity for change.

By contrast, agile development accommodates change systematically. The flexibility of agile approaches increases stability because product changes are predictable and manageable — in other words, it's expected and nondisruptive to a product development team. In later chapters, you discover how agile approaches to planning, working, and prioritization allow teams to respond quickly to change.

As new events unfold, the team incorporates these realities into the ongoing work. Any new item becomes an opportunity to provide additional value instead of an obstacle to avoid, giving development teams a greater opportunity for success.

Defining the 12 Agile Principles

In the months following the publication of the Agile Manifesto, the original signatories continued to communicate. To support teams making the transition to agile approaches, they augmented the four values of the Agile Manifesto with 12 principles.

REMEMBER

These principles, along with the Platinum principles (explained later in the "Adding the Platinum Principles" section), can be used as a litmus test to see whether the specific practices of your team are true to the intent of the agile movement.

Following is the text of the original 12 principles, published in 2001 by the Agile Alliance:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These agile principles provide practical guidance for development teams.

Another way of organizing the 12 principles is to consider them in the following four distinct groups:

- Customer satisfaction
- Quality
- Teamwork
- Product development

The following sections discuss the principles according to these groups.

Agile principles of customer satisfaction

Agile approaches focus on customer satisfaction, which makes sense. After all, the customer is the reason for developing the product in the first place.

While all 12 principles support the goal of satisfying customers, principles 1, 2, 3, and 4 stand out for us:

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.

You may define the customer of a product in a number of ways:

- The customer is the person or group paying for the product.
- In some organizations, the customer may be a client, external to the organization.
- In other organizations, the customer may be a stakeholder or a group of stakeholders in the organization.
- The person who ends up using the product is also a customer. For clarity and to be consistent with the original 12 Agile Principles, we call that person the *user*.

How do you enable these principles? Consider the following:

- Scrum teams (*scrum* is a popular agile framework you learn more about in Chapter 5) include a *product owner,* a person who is responsible for translating what the customer wants into product requirements. To learn more about the role of the product owner, see Chapter 7.
- The product owner prioritizes product features in order of business value or risk and communicates priorities to the development team. The development team delivers the most valuable features on the list in short cycles of development, known as *iterations,* or *sprints.*

- The product owner has deep and ongoing involvement throughout each day to clarify priorities and requirements, make decisions, provide feedback, and quickly answer the many questions that pop up during product development.
- Frequent delivery of working product features allows the product owner and the customer to have a full sense of how the product is developing.
- As the development team continues to deliver complete, working, potentially shippable functionality every one to eight weeks or less, the value of the total product grows incrementally, as do its functional capabilities.
- The customer accumulates value for his or her investment regularly by receiving new, ready-to-use functionality throughout development, rather than waiting until the end for the first, and maybe only, delivery of releasable product features.

In Table 2-3, we list some customer satisfaction issues that commonly arise during product development. Use Table 2-3 and gather some examples of customer dissatisfaction that you've encountered. Do you think becoming more agile would make a difference? Why or why not?

TABLE 2-3 Customer Dissatisfaction and How Agile Might Help

Examples of Customer Dissatisfaction with Product Development	How Agile Approaches Can Increase Customer Satisfaction
The product requirements were misunderstood by the development team.	Product owners work closely with the customer to define and refine product requirements and provide clarity to the development team. Product development teams demonstrate and deliver working functionality at regular intervals. If a product doesn't work the way the customer thinks it should work, the customer can provide feedback at the end of the sprint, not at the end of development, when the feedback would be too late.
The product wasn't delivered when the customer needed it.	Working in sprints allows teams to deliver high-priority functionality early and often.
The customer can't request changes without additional cost and time.	Agile processes are built for change. Development teams can accommodate new requirements, requirement updates, and shifting priorities with each sprint — offsetting the cost of these changes by removing the lowest-priority requirements — functionality that likely will never or rarely get used.

ON THE WEB

You can find a blank template of the form at `www.dummies.com/go/agileprojectmanagementfd3e`.

TIP

Agile strategies for customer satisfaction include the following:

- Producing, in each iteration, the highest-priority features first
- Ideally, locating the product owner and the other members of the team in the same place to eliminate communication barriers
- Breaking requirements into small chunks of value that can be delivered in short iterations
- Keeping written requirements simple, forcing more robust and effective face-to-face communication
- Getting the product owner's acceptance as soon as functionality is completed
- Revisiting the feature list regularly to ensure that the most valuable requirements continue to have the highest priority

Agile principles of quality

A product development team commits to producing quality in every product increment it creates — from development through documentation to integration and test results — every day. Each team member contributes his or her best work all the time. Although all 12 principles support the goal of quality delivery, principles 1, 3, 4, 6–9, and 12 stand out for us:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
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.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These principles, in practice on a day-to-day basis, can be described as follows:

- The development team members must have full ownership of technical quality and be empowered to solve problems. They carry the responsibility for determining how to create the product, deciding the technical work needed to create it, and organizing product development. People not doing the work don't tell them how to do it.
- With software development, an agile approach requires architectures that make coding and testing the product modular, flexible, and extensible. The design should address today's problems and make inevitable changes as simple as possible.
- A set of designs on paper can never tell you that something will work because everything works on paper. When the product quality is such that it can be demonstrated and ultimately shipped in short intervals, everyone knows that the product works — at the end of every sprint.
- As the development team completes features, the team shows the product owner the product functionality to get validation that it meets the acceptance criteria. The product owner's reviews should happen throughout the iteration, ideally the same day that the development of the requirement was completed. Feedback from the product owner is often necessary even during the development of a feature.
- At the end of every iteration (lasting two weeks or less for most teams), working functionality is demonstrated to the customer. Progress is clear and easy to measure.
- Testing is an integral, ongoing part of development and happens throughout the day, not at the end of the iteration. As much as possible, testing is automated. To learn more about automated testing, see Chapter 17.
- With software development, ensuring new code is tested and integrates with previous versions occurs in small increments, possibly several times a day (or thousands of times a day in some organizations, such as Google, Amazon, and Facebook). This process, called *continuous integration (CI),* helps ensure that the entire solution continues to work when new code is added to the existing code base.
- With software development, examples of technical excellence include establishing coding standards, using service-oriented architecture, implementing automated testing, and building for future change.

REMEMBER

Agile principles apply to more than software products. Technical excellence is crucial whether you're developing marketing campaigns, publishing books, involved in manufacturing, or engaged in research and development. All disciplines have a set of technical practices that teams can use to build in quality all along the way.

TIP

Agile approaches provide the following strategies for quality management:

- Defining what *done* means (that is, shippable) at the beginning of development and then using that definition as a benchmark for quality
- Testing aggressively and daily through automated means
- Building only the functionality needed when it's needed
- Reviewing the software code and streamlining (refactoring)
- Showcasing to stakeholders and customers only the functionality that has been accepted by the product owner
- Having multiple feedback points throughout the day, iteration, and product lifecycle

Agile principles of teamwork

Teamwork is critical to agile product development. Creating good products requires cooperation among all members of the team, including customers and stakeholders. Agile approaches support team-building and teamwork, and they emphasize trust in self-managing development teams. A permanent, skilled, motivated, unified, and empowered team is a successful team. To learn more about permanent teams, see Chapter 8.

Although all 12 principles support the goal of teamwork, principles 4–6, 8, 11, and 12 stand out for us as supporting team empowerment, efficiency, and excellence:

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

TIP

Agile approaches focus on sustainable development; as knowledge workers, our brains are the value we bring to product development. If only for selfish reasons, organizations should want fresh, well-rested brains working for them. Maintaining a regular work pace, rather than having periods of intense overwork, helps keep each team member's mind sharp and product quality high. This fact was known as early as 1908, when Dr. Ernst Abbe quantified that reducing daily work from 12 to 8 hours increased cumulative output. P. Sargant Florence, author of *The Economics of Fatigue and Unrest*, showed that 8-hour days produce between 16 to 20 percent higher total output than 9-hour days.

Here are some practices you can adopt to make this vision of teamwork a reality:

- Ensure that your development team members have the proper skills and motivation.
- Provide training sufficient to the task.
- Support the self-organizing development team's decisions about what to do and how to do it; don't have managers tell the team what to do.
- Hold team members responsible as a single team, not as individuals.
- Use face-to-face communication to quickly and efficiently convey information.

  WARNING

  Suppose that you usually communicate to Sharon by email. You take time to craft your message and then send it. The message sits in Sharon's inbox, and she eventually reads it. If Sharon has any questions, she writes another email in response and sends it. That message sits in your inbox until you eventually read it. And so forth. This type of table tennis communication is too inefficient to use in the middle of a rapid iteration. A five-minute discussion addresses the issue quickly and with less risk of misunderstanding — and with a reduced cost of delay.
- Have spontaneous conversations throughout the day to build knowledge, understanding, and efficiency.
- Collocate teammates in close proximity to increase clear and efficient communication. If collocation isn't possible, use video chat rather than email. Teams who rely on written communication for collaboration are slower and more prone to miscommunication errors. Written intra-team communication is a liability.
- Make sure that *lessons learned* is an ongoing feedback loop rather than an end-of-project-only occurrence. Retrospectives should be held at the end of each iteration, when reflection and adaptation can improve development team productivity immediately going forward, creating ever higher levels of efficiency. A lessons learned meeting at the end of development has minimal value because the next product created might have a different group of people and practices. To learn more about retrospectives, see Chapter 12.

The first retrospective is as valuable (or even more valuable) as any future retrospective because, at the beginning, the team has the opportunity to make changes that can benefit the rest of the product development moving forward.

TIP

The following strategies promote effective teamwork:

- Collocate the development team so it has no physical barriers to effective and real-time communication.
- Put together a physical environment conducive for collaboration: a team room with whiteboards, colored pens, and other tactile tools for developing and conveying ideas to ensure shared understanding.
- Create an environment where team members are encouraged to speak their minds.
- Meet face-to-face whenever possible. Don't send an email if a conversation can handle the issue.
- Get clarifications throughout the day as they're needed.
- Encourage the team to solve problems rather than having managers solve problems for the team.
- Resist the temptation to shuffle team members. Allow the team to become a stable, permanent, high-performing, capability-expanding team.

REMEMBER

A long-term product perspective requires long-term, permanent teams. High-performing teams take years to build. Their understanding of the customer, their feedback from each release, the product support they provide, and the product development environment logically encourage teams to remain as stable as possible. Team members may seek new opportunities for career development outside the team, but for the most part teams should remain as constant as possible for maximum value. As each new feature is built, the team remains constant, able to support and learn from the product's adoption by the customer.

Agile principles of product development

Agility in product management encompasses three key areas:

- Making sure the development team can be productive and can sustainably increase productivity over long periods of time

- Ensuring that information about the product's progress is available to stakeholders without interrupting the flow of development activities by asking the development team for updates
- Handling requests for new features as they occur and integrating them into the product development cycle

An agile approach focuses on planning and executing the work to produce the best product that can be released. The approach is supported by communicating openly, avoiding distractions and wasteful activities, and ensuring that the progress of the product development is clear to everyone.

All 12 principles support product management, but principles 1–3 and 7–10 stand out for us:

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.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.

Following are some advantages of adopting agile product management:

- Product development teams achieve faster time-to-market and, consequently, cost savings. They start development earlier than in traditional approaches because agile approaches minimize the exhaustive up-front planning and documentation that is conventionally part of the early stages of a waterfall project.
- Product development teams are self-organizing and self-managing. The managerial effort normally put into telling developers how to do their work can be applied to removing impediments and organizational distractions that slow down the team.
- Agile development teams determine how much work they can accomplish in an iteration and commit to achieving those goals. Ownership is fundamentally

different because the development team is establishing the commitment, not complying with an externally developed commitment.

- An agile approach asks, "What is the minimum goal we can set that still adds value?" instead of focusing on including all features and extra refinements that could possibly be needed. An agile approach usually means streamlining: barely sufficient documentation, removal of unnecessary meetings, avoidance of inefficient communication (such as email), and minimizing complexity of what's under the hood (just enough to make it work).

  WARNING

  Creating complicated documents that aren't useful for product development is a waste of effort. It's okay to document a decision, but you don't need multiple pages on the history and nuances of how the decision was made. Keep the documentation barely sufficient (that is, sufficient but just barely), and you'll have more time to focus on supporting the development team.

- By encapsulating development into short sprints that last several weeks or less, you can adhere to the goals of the current iteration while accommodating change in subsequent iterations. The length of each sprint remains the same throughout development to provide a predictable rhythm for the team long-term.

- Planning, elaborating on requirements, developing, testing, and demonstrating functionality occur within an iteration, lowering the risk of heading in the wrong direction for extended periods of time or developing something that the customer doesn't want.

- Agile practices encourage a steady pace of development that is productive and healthy. For example, in the popular agile software development set of practices called extreme programming (XP), the maximum workweek is 40 hours, and the preferred workweek is 35 hours. Agile product development is constant and sustainable, as well as more productive, especially long term.

  WARNING

  Traditional approaches routinely feature a *death march,* in which the team puts in extremely long hours for days and even weeks at the end to meet a previously unidentified and unrealistic deadline. As the death march goes on, productivity tends to drop dramatically. More defects are introduced, and because defects need to be corrected in a way that doesn't break a different piece of functionality, correcting defects is the most expensive work that can be performed. Defects are often the result of overloading a system — specifically demanding an unsustainable pace of work. Check out our presentation on the negative effects of "Racing in Reverse" (`https://platinumedge.com/overtime`).

- Priorities, experience with the existing product, and, eventually, the speed at which development will likely occur within each sprint are clear, making for good decisions about how much can or should be accomplished in a given amount of time.

If you've worked on a traditional project before, you might have a basic understanding of project management activities. In Table 2-4, we list a few project management tasks, along with how you would meet those needs with agile approaches. Use Table 2-4 to capture your thoughts about your experiences and how agile approaches look different from traditional project management.

ON THE WEB

A blank template of Table 2-4 is available at `www.dummies.com/go/agileprojectmanagementfd3e`.

TABLE 2-4 Contrasting Historical Project Management with Agile Product Management

Traditional Project Management Tasks	Agile Approach to Product Development Tasks
Create a fully detailed project requirement document at the beginning of the project. Try to control requirement changes throughout the project.	Create a product backlog — a simple list of requirements by priority. Quickly update the product backlog as requirements and priorities change throughout product development.
Conduct weekly status meetings with all project stakeholders and developers. Send detailed meeting notes and status reports after each meeting.	The development team meets quickly, for no longer than 15 minutes, at the start of each day to coordinate and synchronize that day's work and any roadblocks. The team can update the centrally visible burndown chart in under a minute at the end of each day. Anyone, including stakeholders, can see the real-time progress on demand.
Create a detailed project schedule with all tasks at the beginning of the project. Try to keep the project tasks on schedule. Update the schedule on a regular basis.	Work within sprints and identify only specific tasks for the active sprint.
Assign tasks to the development team.	Support the development team by removing impediments and distractions. Development teams define and pull (as opposed to push) their own tasks.

TIP

Successful product development is facilitated by the following agile approaches:

- Supporting the development team with real-time answers to its questions, shielding it from competing priorities, and empowering it to develop solutions and determine how much work to take on in each iteration
- Producing barely sufficient documents
- Streamlining status reporting so that information is pushed out by the development team in seconds rather than pulled out by a project manager over a longer period of time

- Minimizing nondevelopment tasks
- Setting expectations that change is normal and beneficial, not something to be feared or evaded
- Adopting a just-in-time requirements refinement to minimize change disruption and wasted effort
- Collaborating with the development team to create realistic schedules, targets, and goals
- Protecting the team from organizational disruptions that could undermine product goals by introducing work that is not relevant to the product objectives
- Understanding that an appropriate balance between work and life is a component of efficient development
- Viewing the product as a long-term investment requiring permanent teams pursuing value over specifications

Adding the Platinum Principles

Through in-the-trenches experience working with teams transitioning to agile product development — and field testing in large, medium, and small organizations worldwide — we developed three additional principles of agile product development that we call the Platinum principles:

- Resist formality.
- Think and act as a team.
- Visualize rather than write.

You can explore each principle in more detail in the following sections.

Resisting formality

Even the most agile product development teams can drift toward excessive formalization. For example, it isn't uncommon for us to find team members waiting until a scheduled meeting to discuss simple issues that could be solved in seconds. These meetings often have an agenda and meeting minutes and require a certain level of demobilization and remobilization just to attend. In an agile approach, this level of formalization isn't required.

WARNING

You should always question formalization and unnecessary, showy displays. For example, is there an easier way to get what you need? How does the current activity support the development of a quality product as quickly as possible? Answering these questions helps you focus on productive work and avoid unnecessary tasks.

In an agile system, discussions and the physical work environment are open and free-flowing; documentation is kept to the lowest level of quantity and complexity such that it contributes value to the product, not hamper it; and flashy displays, such as well-decorated presentations, are avoided. Professional, frank communications are best for the team, and the entire organizational environment has to make that openness available and comfortable.

TIP

Strategies for success in resisting formality include the following:

- Reducing organizational hierarchy wherever possible by eliminating titles in the team
- Avoiding aesthetic investments such as elaborate slide presentations or extensive meeting minute forms, especially when demonstrating shippable functionality at the end of a sprint
- Educating stakeholders who request complicated displays about the high costs and low returns of such displays

Thinking and acting as a team

Team members should focus on how the team as a whole can be most productive. This focus can mean letting go of individual niches and performance metrics. In an agile environment, the entire team should be aligned in its commitment to the goal, its ownership of the scope of work, and its acknowledgment of the time available to achieve that commitment.

Following are some strategies for thinking and acting as a team:

- Develop in pairs and switch partners often. Both pair programming (each partner is knowledgeable in the area) and shadowing (only one partner is knowledgeable in the area) raise product quality and team member capability and reduce single points of failure. You can learn more about pair programming in Chapter 17.

- Replace individual work titles with a uniform product developer title. Development activities include all tasks necessary to take requirements through to functionality, including design, implementation (for example, coding), testing, and documentation — not just writing code or turning a screwdriver.
- Report at the team level only, as opposed to creating special management reports that subdivide the team.
- Replace individual performance metrics with team performance metrics.

Visualizing rather than writing

A product development team should use visualization as much as possible, whether through simple diagrams or computerized modeling tools. Images are much more powerful than words. When you use a diagram or mockup instead of a document, your customer can relate better to the concept and the content.

Our ability to define the features of a system increases exponentially when we step up our interaction with the proposed solution: A graphical representation is almost always better than a textual one, and experiencing functionality hands-on is best.

TIP

Even a sketch on a piece of paper can be a more effective communication tool than a formal text-based document. A picture is worth a thousand words. A textual description is the weakest form of communication if you're trying to ensure common understanding — especially when the description is delivered by email with the request to "let me know if you have any questions."

Examples of strategies for visualization include the following:

- Stocking the work environment with plenty of whiteboards, poster paper, pens, and paper so that drawing tools are readily available
- Using models instead of text to communicate concepts
- Reporting status through charts, graphs, and dashboards, such as those in Figure 2-2

FIGURE 2-2: Charts and graphs for providing transparency.

Changes as a Result of Agile Values

The publication of the Agile Manifesto and the 12 Agile Principles legitimized and focused the agile movement in the following ways:

- **Agile approaches changed attitudes toward product management processes.** In trying to improve processes, methodologists in the past worked to develop a universal process that could be used under all conditions, assuming that more process and greater formality would yield improved results. This approach, however, required more time, overhead, and cost and often diminished quality. The manifesto and the 12 principles acknowledged that too much process is a problem, not a solution, and that the right process in the right amount differs in each situation.
- **Agile approaches changed attitudes toward knowledge workers.** IT groups began to remember that development team members aren't disposable resources but individuals whose skills, talents, and innovation make a difference to every product. The same product created by different team members will be a different product.

- **Agile approaches changed the relationship between business and IT groups.** Agile product development addressed the problems associated with the historical separation between business and IT by bringing these contributors together on the same team, at equal levels of involvement and with shared goals.
- **Agile approaches corrected attitudes toward change.** Historical approaches viewed change as a problem to be avoided or minimized. The Agile Manifesto and its principles helped identify change as an opportunity to ensure that the most informed ideas were implemented.

CHANGES TO COME

Enterprises are leveraging agile techniques on a large-scale basis to solve business problems. Although the methodologies of agile IT groups, as well as non-IT groups, have undergone radical transformation, the organizations around these groups have often continued to use historical methodologies and concepts. For example, corporate funding and spending cycles are still geared toward the following:

- Long development efforts that deliver working software at the end of the project
- Annual budgeting
- An assumption that certainty is possible at the beginning of a project
- Corporate incentive packages focused on individual rather than team performance

The resulting tension keeps organizations from taking full advantage of the efficiency and significant savings that agile techniques promise.

A truly integrated agile approach encourages organizations to move away from the last century's traditions and develop a structure at all levels that continually asks, "What's best for the customer, the product, and the team?"

A product development team can be only as agile as the organization it serves. As the movement continues to evolve, the values articulated in the Agile Manifesto and its principles provide a strong foundation for the changes necessary to make individual products, customer-driven solutions, and entire organizations more productive and profitable. This evolution will be driven by passionate methodologists who continue to explore and apply agile principles and practices.

The Agile Litmus Test

To be agile, you need to be able to ask, "Is this agile?" If you're ever in doubt about whether a particular process, practice, tool, or approach adheres to the Agile Manifesto or the 12 principles, refer to the following list of questions:

1. Does what we're doing at this moment support the early and continuous delivery of valuable software?
2. Does our process welcome change and take advantage of change?
3. Does our process lead to and support the delivery of working functionality?
4. Are the developers and the product owner working together daily? Are customers and business stakeholders working closely with the team?
5. Does our environment give the team the support it needs to get the job done?
6. Are we communicating in person face-to-face rather than over the phone or through email?
7. Are we measuring progress by the amount of working functionality produced?
8. Can we maintain our current pace of development indefinitely?
9. Do we support technical excellence and good design that allows for future changes?
10. Are we maximizing the amount of work not done — namely, doing as little as necessary to fulfill the product goal for our customer?
11. Is this development team self-organizing and self-managing? Does it have the freedom to succeed?
12. Are we reflecting at regular intervals and adjusting our behavior accordingly?

If you answered "yes" to all these questions, congratulations; you're likely becoming more agile. If you answered "no" to any of these questions, what can you do to change that answer to "yes"? You can come back to this exercise at any time and use the agile litmus test for yourself, as well as with your team and the wider organization.

IN THIS CHAPTER

- Discovering the benefits of agile product development
- Comparing agile approaches to historical approaches
- Finding out why people like agile techniques

Chapter 3

Why Being Agile Works Better

Agile approaches work well in the real world. Why is this? In this chapter, you examine the mechanics of how agile processes improve the way people work and how they prevent burdensome overhead. Comparisons with historical methods highlight the improvements agile techniques bring.

When talking about agile product development advantages, the bottom line is twofold: success and stakeholder satisfaction.

Evaluating Agile Benefits

The agile concept of product development is different from previous project management approaches and methodologies. As mentioned in Chapter 1, agile approaches address key challenges of historical project management methods such as waterfall, but they also go much deeper. Agile principles provide a framework for how we *want* to work — how we naturally function when we solve complex problems.

Historical methods of project management were developed not for contemporary development lifecycles, such as new product development, but for simpler and obvious systems. It's no wonder that these project management methods don't fit

when attempting to build more complex, modern products, such as artificial intelligence, aircraft, cybersecurity, medical devices, financial management systems, mobile applications, and web-centric, object-oriented applications, which require constant innovation to stay competitive. Even with older technologies, the track record of traditional methodologies is abysmal, especially when applied to software development. For more details on the high failure rates of projects that are run traditionally, check out the studies from the Standish Group shown in Chapter 1.

REMEMBER

You can use agile product development techniques in many industries besides software development. If you're creating a product and want early feedback throughout the process, you can benefit from agile processes.

When you have a critical looming deadline, your instinct is to *go agile.* Formality goes out of the window as you roll up your sleeves and focus on what has to get done. You solve problems quickly, practically, and in descending order of necessity, making sure you complete the most critical tasks.

More than going agile — it's about *being agile.* When you become agile, you don't institute unreasonable deadlines to force greater focus. Instead, you realize that people function well as practical problem solvers, even under stress. For example, a popular team-building exercise titled the marshmallow challenge involves groups of four people building the tallest free-standing structure possible out of 20 sticks of spaghetti, a yard of tape, and a yard of string, and then placing a marshmallow on the top — in 18 minutes. See `https://tomwujec.com` for background information about the concept. On that site, you can also view the associated TED Talk by Tom Wujec.

Wujec points out that young children usually build taller and more interesting structures than most adults because children build incrementally on a series of successful structures in the time allotted. Adults spend a lot of time planning, produce one final version, and then run out of time to correct any mistakes. The youngsters provide a valuable lesson that *big bang development* — namely, excessive up-front planning and then one shot at product creation — doesn't work. Formality, excessive time detailing uninformed future steps, and a single plan are often detriments to success.

The marshmallow challenge sets opening conditions that mimic those in real life. You build a structure (which equates to developing a product) using fixed resources (four people, spaghetti, and so on) and a fixed time (18 minutes). What you end up with is anyone's guess, but an underlying assumption in historical project management approaches is that you can determine the precise destination (the features or requirements) in the beginning and then estimate the people, resources, and time required.

This assumption is upside down from how life really is. As you can see in Figure 3-1, the theories of historical methods are the reverse of agile approaches. We pretend that we live in the world on the left, but we actually live in the world on the right.

FIGURE 3-1: A comparison of historical project management and agile concepts.

In the historical approach, which locks the requirements and delivers the product all in one go, the result is all or nothing. We either succeed completely or fail absolutely. The stakes are high because everything hinges on work that happens at the end (that is, putting the marshmallow on the top) of the final phase of the cycle, which includes integration and customer testing.

In Figure 3-2, you can see how each phase of a waterfall project is dependent on the previous one. Teams design and develop all features together, meaning you don't get the highest-priority feature until you've finished developing the lowest-priority feature. The customer has to wait until the end of the project to get final delivery of any element of the product.

In the testing phase of a waterfall project, the customer finally gets to start seeing parts of the long-awaited product. By that time, the investment and effort have been huge, and the risk of failure is high. Finding defects among all completed product requirements is like looking for a weed in a cornfield.

Agile approaches turn the concept of how product development should be done upside down. Using agile methods, you develop, test, and integrate small groups of product requirements in short iterative cycles, known as *iterations*, or *sprints*, as illustrated in Figure 3-3. Testing occurs during each iteration rather than at the end of development. Development teams identify and remove defects, preventing them from ever finding their way to the customer, just like a gardener can more easily find a weed in a flowerpot than in a cornfield. They not only find and remove the weeds, they prevent the weed seeds from germinating.

FIGURE 3-2: The waterfall project cycle is a linear methodology.

FIGURE 3-3: Agile approaches have an iterative development cycle.

WHERE THE WATERFALL FALLS SHORT

As we mention in Chapter 1, before 2008, waterfall was the most widely used traditional project management methodology. The following list summarizes the major aspects of the waterfall approach to project management:

- The team must know all requirements up front to estimate time, budgets, team members, and resources. Knowing all the requirements at the project start means you have a high investment in detailed requirements gathering before any development begins.
- Estimation is complex and requires a high degree of competence and experience and a lot of effort to complete.
- The customer and stakeholders may not be available to answer questions during the development period, because they may assume that they provided all the information needed during the requirements-gathering and design phases.
- The team needs to resist the addition of new requirements or document them as change orders, which adds more work to the project and extends the schedule and budget.
- The team must create and maintain volumes of process documentation to manage and control the project.
- Although some testing can be done as you go, final testing can't be completed until the end of the project, when all functionality has been developed and integrated.
- Full and complete customer feedback is not possible until the end of the project, when all functionality is complete.
- Funding is ongoing, but the value appears only at the end of the project, creating a high level of risk.
- The project has to be fully complete for value to be achieved. If funding runs out prior to the end of the project, the project delivers zero value.

Product owner, *scrum master*, and *sprint* are terms from *scrum*, a popular agile framework for organizing work and exposing progress. *Scrum* refers to a rugby huddle, in which rugby players come together to gain possession of the ball. Scrum as an approach, like rugby, encourages the team to work together closely toward a common goal and take responsibility for the result. (You find out more about scrum and other agile techniques in Chapter 5.) We use scrum as an example to explain many of the concepts in the rest of this chapter.

Moreover, with agile product development, the customers get to see their product at the end of every short cycle. You can create the highest-priority features first,

which gives you the opportunity to ensure maximum value early on, when less of the customer's money has been invested.

An agile approach to product development will reduce risk during every iteration. In addition, if your product has market value, revenue can be coming in even during development. Now you have a self-funding product!

How Agile Approaches Beat Historical Approaches

Agile frameworks promise significant advantages over historical methods, including greater flexibility and stability, less nonproductive work, faster delivery with higher quality, improved development team performance, tighter control, and faster failure detection. We describe all these results in this section.

However, these results can't be achieved without a highly competent, enduring, and functional development team. The development team is pivotal to the success of the product. Agile methods emphasize the importance of the support provided to the development team as well as the importance of team members' actions and interactions.

REMEMBER

The first value in the Agile Manifesto is "Individuals and interactions over processes and tools." Nurturing the development team is central to agile product development and the reason why you can have such success with agile approaches.

Scrum teams are centered on development teams (which includes feature creators, testers, designers, and anyone else who does the actual work of creating the product), as well as the following two important team members, without which the development team couldn't function:

- **Product owner:** The *product owner* is a team member who is an expert on the product and the customer's business needs. The product owner works with the business community and prioritizes product requirements, and supports the development team by being available to provide daily clarifications and final acceptance to the development team. (Chapter 2 has more on the product owner.)
- **Scrum master:** The *scrum master* acts as a buffer between the development team and distractions that might slow down the development effort. The scrum master also provides expertise on agile processes and helps remove obstacles that hinder the development team from making progress. They facilitate consensus building and continuous team improvement.

You can find complete descriptions of the product owner, the development team, and the scrum master in Chapter 7. Later in this chapter, you see how the product owner and scrum master's highest priority is supporting and optimizing the development team's performance.

Greater flexibility and stability

By way of comparison, agile product development offers both greater flexibility and greater stability than traditional projects. First, you find out how agile development offers flexibility, and then we discuss stability.

A team, regardless of its project or product management approach, faces two significant challenges at the beginning of product development:

- The team has limited knowledge of the product end state.
- The team cannot predict the future.

This limited knowledge of the product and of future business needs almost guarantees changes.

REMEMBER

The fourth core value in the Agile Manifesto is "Responding to change over following a plan." Agile frameworks are created with flexibility in mind.

With agile approaches, teams can adapt to new knowledge and new requirements that emerge as development progresses. We provide many details about the agile processes that enable flexibility throughout this book. Here's a simple description of some processes that help product development teams manage change:

- At the start of product development, the product owner gathers high-level product requirements from stakeholders and prioritizes them. The product owner doesn't need all the requirements broken down in detail up front; he or she needs just enough to have a good understanding of what the product must accomplish.
- The development team and the product owner work together to break down the initial highest-priority requirements into more detailed requirements. The result is small chunks of valuable work that the development team can start developing immediately.
- You focus on the top priorities in each sprint regardless of how soon before the sprint those priorities were set.

  Iterations, or sprints are short — they last up to four weeks, and are often one or two weeks. You can find details about sprints in Chapters 10–12.

TIP

- The development team works on groups of requirements within sprints and learns more about the product with each successive sprint.
- The development team plans one sprint at a time and drills further into requirements at the beginning of each sprint. The development team generally works only on the next highest-priority requirements.
- Concentrating on one sprint at a time and on the highest-priority requirements allows the team to accommodate new high-priority requirements at the beginning of each sprint.
- When changes arise, the product owner updates a list of requirements that remains to be dealt with in future sprints. The product owner reprioritizes the list regularly based on changing market or business conditions.
- The product owner can financially invest in high-priority features first and can choose which features to fund throughout development.
- The product owner and development team collect client feedback at the end of each sprint and act on that feedback. Client feedback often leads to changes to existing functionality or to new, valuable requirements. Feedback can also lead to removing or reprioritizing requirements that are not really necessary.
- The product owner can stop development once he or she deems the product has sufficient functionality to fulfill product goals. Agile product development typically ends early, running out of value before it runs out of time or money.

Figure 3-4 illustrates how making changes with agile development can be more stable than making changes in waterfall. Think of the two images in the figure as steel bars. In the top image, the bar represents a two-year project. The bar's length makes it much easier to distort, bend, and break. Project changes can be thought of in the same way — long projects are structurally vulnerable to instability because the planning stage of a project is different than the execution, when reality sets in, and there is no natural point of give in a long project.

FIGURE 3-4: Stability in flexibility with agile product development.

Now look at the bottom image in Figure 3-4. The small steel bars represent a two-week iteration. It is much easier for those small bars to be stable and unchanging than it is for the larger bar. In the same manner, it is easier to have stability in smaller increments with known flexibility points. Telling a business there can be no changes for two weeks is much easier and more realistic than saying there can be no changes for two years.

Agile product development is tactically flexible because it is strategically stable. Agile approaches are great at accommodating change because the means for regular change are built into everyday processes. At the same time, iterations offer distinct areas for stability. Teams accommodate changes to the product backlog anytime but do not generally accommodate external changes to scope during the sprint. The product backlog may be constantly changing, but, except in emergencies, the sprint is generally stable.

At the beginning of the iteration, the development team plans the work it will complete for that sprint. After the sprint begins, the development team works only on the planned requirements. A couple of exceptions to this plan can occur — if the development team finishes early, it can request more work; if an emergency arises, the product owner can cancel the sprint. In general, however, the sprint is a time of great stability for the development team.

This stability can lead to innovation. When development team members have stability — that is, they know what they will be working on in a set period of time — they will think about their tasks consciously at work. They may also think about tasks unconsciously away from work and tend to come up with solutions at any given time.

Agile product development provides a constant cycle of development, feedback, and change, allowing teams the flexibility to create products with only the right features and the stability to be creative.

Reduced nonproductive tasks

At any point in your working day when you're creating a product, you can work on either developing the product or on the peripheral processes that are supposed to manage and control the creation of the product. Clearly, there's more value in the first, which you should try to maximize, than in the second, which you want to minimize.

To develop a product, you have to work on the solution. As obvious as this statement sounds, it's routinely neglected on waterfall projects. Programmers on some software projects spend only 20 percent of their time generating functionality, with the rest of the time in meetings, writing emails, or creating unnecessary presentations and documentation.

Product development can be an intense activity that requires sustained periods of focus. Many developers can't get enough development time during their normal workday to keep up with the schedule of a project because they're doing other types of tasks. The following causal chain is the result:

> Long workday = tired developers = unnecessary defects = more defect fixing = delayed release = longer time to value

WARNING

Don't be fooled into thinking that your development team will work only one weekend. If you work one weekend, you'll probably work most weekends from now on. Working overtime sets a false watermark. After you do it, people don't expect less moving forward; if anything, they'll expect more.

To maximize productive work, the goal is to eliminate overtime and have developers creating functionality during the working day. To increase productive work, you have to reduce unproductive tasks, period.

Meetings

Meetings can be a large waste of valuable time. On traditional projects, development team members may find themselves in long meetings that provide little or no benefit to the developers. The following agile approaches can help ensure that development teams spend time only in productive, meaningful meetings:

- Agile processes include only a few formal meetings. These meetings are focused, with specific topics and limited time. With agile product development, you generally don't need to attend non-agile meetings.
- Part of the scrum master's job is to prevent disruptions to the development team's working time, including requests for non-agile meetings. When there's a demand to pull developers away from development work, the scrum master asks "why" to understand the true need. The scrum master, working with the product owner, then may figure out how to satisfy that need without disrupting the development team.
- With agile product development, the current status is often visually available to the entire organization, removing the need for status meetings. You can find ways to streamline status reporting in Chapter 16.

Email

Email is not an efficient mode of communication to resolve issues. Product development teams aim to use email sparingly, perhaps when information needs to be sent out or a simple yes-or-no answer is required. Even then, better tools exist, such as persistent chat tools in which the conversation taking place is the same for all participants. The email process is asynchronous and slow: You send an

email, wait for an answer, have another question, and send another email. This process eats up time that could be spent more productively.

Instead of sending emails, teams use face-to-face discussions to resolve questions and issues on the spot.

Presentations

When preparing for a presentation of the functionality to the customer, product development teams often use the following techniques:

- **Demonstrate, don't present.** In other words, show the customer what you've created, rather than describing what you've created. Product development teams always have a shippable increment of their product ready to demonstrate, avoiding the temptation to report hypothetically on work in progress.

  Customers and stakeholders can provide even better feedback if the demonstration includes the opportunity for them to experience the product increment for themselves by putting their hands on the keyboard or product themselves.

TIP

- **Show how the functionality delivers on the requirement and fulfills the acceptance criteria.** In other words, say, "This was the requirement. These are the criteria needed to indicate that the feature was complete. Here is the resulting functionality meeting those criteria."
- **Avoid formal slide presentations and all the preparation they involve.** When you demonstrate the working functionality, it will speak for itself. Keep demonstrations raw and real.

Process documentation

Documentation has been the burden of project managers and developers for a long time. Product development teams can minimize documentation with the following approaches:

- **Use iterative development.** A lot of documentation is created to reference decisions made months or years ago. Iterative development shortens the time between decision and developed product from months or years to days. The product and associated automated tests, rather than extensive paperwork, document the decisions made.
- **Remember that one size doesn't fit all.** You don't have to create the same documents for every development effort. Choose what makes sense for your product, stakeholders, and customers.

- **Use informal, flexible documentation tools.** Whiteboards, sticky notes, charts, and other visual representations of the work plan are sufficient tools.
- **Include simple tools that provide adequate information for management about product development progress.** Don't create special progress reports, such as extensive status reports, for the sake of reporting. Product development teams use visual charts, such as burndown charts, to readily convey status while remembering that "working software (or product) is the primary measure of progress" (Principle 7).

Higher quality, delivered faster

On traditional projects, the period from completion of requirements gathering to the beginning of customer testing can be painfully long. During this time, the customer is waiting to see some sort of result, and the development team is wrapped up in developing. The project manager is making sure that the project team is following the plan, keeping changes at bay, and updating everyone with an interest in the outcome by providing frequent and detailed reports.

When testing starts, near the end of the project, defects can cause budget increases, create schedule delays, and even kill a project. Testing is a project's largest unknown, and in traditional projects, it is an unknown carried until the end.

Agile product development is designed to deliver high-quality, shippable functionality quickly. Agile product development achieves better quality and quick delivery with the following:

- The client reviews working functionality at the end of each sprint, and gives immediate feedback to the team for inspection and adaptation as soon as the next sprint.
- Short development iterations (sprints) limit the number and complexity of features in development at any given time, making the finished work easier to test in each sprint. Only so much can be created in each sprint. Development teams break down features too complex for one sprint.
- The development team builds and tests daily and maintains a working product throughout product development.
- The product owner is involved throughout the day to answer questions and clarify misunderstandings quickly.
- The development team is empowered and motivated and has a reasonable workday. Because the development team is not worn out, fewer defects occur.

- Errors are detected quickly because developers test their work as it's completed. Extensive automated testing happens frequently — with every code check-in if necessary.
- Modern software development tools allow many requirements to be written as test scripts, without the need for programming, which makes automated testing quicker.

Improved team performance

Central to agile product development is the experience of the team members. Compared with traditional approaches such as waterfall, product development teams get more environmental and organizational support, can spend more time focusing on their work, and can contribute to the continuous improvement of the process. To find out what these characteristics mean in practice, continue reading.

Support for the team

The development team's ability to deliver potentially shippable functionality is central to getting results with agile approaches and is achieved with the following support mechanisms:

- A common agile practice is *collocation* — keeping the development team, scrum master, and product owner together in one place and physically close to the customer. Collocation encourages collaboration and makes communication faster, clearer, and easier. You can get out of your seat, have a direct conversation, and eliminate any vagueness or uncertainty immediately.
- The product owner can respond to development team questions and requests for clarification without delay, eliminating confusion and allowing work to proceed smoothly.
- The scrum master removes impediments and ensures that the development team has everything it needs to focus and achieve maximum productivity.

Focus

Using agile processes, the development team can focus as much of its work time as possible on the development of the product. The following approaches help agile development teams focus:

- Development team members are allocated 100 percent to one team goal, eliminating the time and focus lost by switching context.

- Development team members know that their teammates will be fully available.
- Developers focus on small units of functionality that are as independent as possible from other functionality. Every morning, the development team knows what it means to be successful that day.
- The scrum master has an explicit responsibility to help protect the development team from organizational distractions.
- The time the development team spends on creating and related productive activities increases because nonproductive work decreases.

Continuous improvement

An agile process isn't a mindless check-the-box approach. Different types of products and different teams are able to adapt around their specific situation, as you see in the discussion of sprint retrospectives in Chapter 12. Here are some ways that teams can continuously improve:

- Iterative development makes continuous improvement possible because each new iteration involves a fresh start.
- Because sprints happen over only a week or two, teams can incorporate process changes quickly.
- A review process called the *retrospective* takes place at the end of each iteration and gives all team members a specific forum for identifying and planning actions for improvements.
- The entire scrum team — product owner, development team members, and scrum master — reviews aspects of the work it feels might need improvement.
- The scrum team applies the lessons it learns from the retrospective to the sprints that follow, which thus become more productive.

Tighter control

The work goes more quickly with agile development than under waterfall conditions. Elevated productivity helps increase control with the following:

- Agile processes provide a constant flow of information. Development teams plan their work together every morning in daily scrum meetings, and they update task status throughout each day.

- For every sprint, the customer has the opportunity to reprioritize product requirements based on business needs.
- Based on the working functionality you deliver at the end of each sprint, you determine the work for the next sprint according to current knowledge and updated priorities. You're not locked in to priorities set days, weeks, months, or years ago.
- When the product owner sets priorities for the next sprint, this action has no effect on the current sprint. With agile development, a change in requirements adds no administrative costs or time and doesn't disrupt the current work.
- Agile techniques make product termination easier. At the end of each iteration, you can determine whether the features of the product are now adequate. Low-priority items may never need to be developed.

In waterfall, project metrics may be outdated by weeks, and demonstrable functionality may be months away. In an agile context, metrics are fresh and relevant every day, work completed is compiled and integrated daily, and working product is demonstrated, at most, every few weeks. From the first sprint to the close of development, every team member knows whether the team is delivering. Up-to-the-minute knowledge and the ability to quickly prioritize make high levels of control possible.

Faster and less costly failure

In a waterfall project, opportunities for failure detection are theoretical until close to the end of the project schedule, when all the completed work comes together and when most of the investment is gone. Waiting until the final weeks or days of the project to find out that the product has serious issues is risky for all concerned. Figure 3-5 compares the risk and investment profile for waterfall with that for agile approaches.

Along with opportunities for tighter control, the agile framework offers you

- Earlier and more frequent opportunities to detect failure
- An assessment and action opportunity every few weeks
- Reduction in failure costs

What sorts of failures have you seen on projects? Would agile approaches have helped? You can find out more about risk on agile product development in Chapter 17.

FIGURE 3-5: A risk and investment chart comparing waterfall and agile methodologies.

Why People Like Being Agile

You've seen how an organization can benefit from agile product development with faster delivery and lower costs. In the following sections, you find out how the people involved can benefit as well, whether directly or indirectly.

Executives

Agile development provides two benefits that are especially attractive to executives: efficiency and a higher and quicker return on investment.

Efficiency

Agile practices allow for vastly increased efficiency in the development process in the following ways:

- Agile development teams are very productive. They organize the work themselves, focus on development activities, and are protected from distractions by the product owner and scrum master.
- Nonproductive efforts are minimized. An agile approach eliminates unfruitful work; the focus is on development.
- By using simple, timely, on-demand visual aids — such as graphs and diagrams — to display what's been done, what's in progress, and what's to come, the progress of development is easier to understand at a glance.

- Through continuous testing, defects are detected and corrected early.
- Development can be halted when it has enough functionality.

Increased ROI opportunity

ROI is significantly enhanced using agile approaches for the following reasons:

- **Functionality is delivered to the marketplace earlier.** Features are fully completed and then released in groups, rather than waiting until the end of all development and releasing 100 percent of the features at once.
- **Product quality is higher.** The scope of development is broken down into manageable chunks that are tested and verified on an ongoing basis.
- **Revenue opportunity can be accelerated.** Increments of the product are released to the market earlier than with traditional project approaches. Speed-to-market advantage is indefensible.
- **Products can self-fund.** A release of functionality might generate revenue while development of further features is ongoing.

Product development and customers

Customers like agile product development because they can accommodate changing requirements and generate higher-value products.

Improved adaptation to change

Changes to product requirements, priorities, timelines, and budgets can greatly disrupt traditional projects. In contrast, agile processes handle changes in beneficial ways. For example:

- Agile development creates an opportunity for increased customer satisfaction and return on investment by incorporating change even late in development, nondisruptively.
- Because the team members and the sprint length remain constant, product changes pose fewer problems than with traditional approaches. Necessary changes are slotted into the features list based on priority, pushing lower-priority items down the list. Ultimately, the product owner chooses when development will end, at the point where future investment won't provide enough value.

- Because the development team develops the highest-value items first and the product owner controls the prioritization, the product owner can be confident that business priorities are aligned with development activity.

Greater value

With iterative development, product features can be released as the development team completes them. Iterative development and releases provide greater value in the following ways:

- Teams deliver highest-priority product features earlier.
- Teams can deliver valuable products earlier.
- Teams can adjust requirements based on market changes and customer feedback.

Management

People in management tend to like agile development for the higher quality of the product, the decreased waste of time and effort, and the emphasis on the value of the product over checking off lists of features of dubious usefulness.

Higher quality

With software development, through such techniques as test-driven development, continuous integration, and frequent customer feedback on working software, you can build higher quality into the product up front.

Perhaps you work on product development that does not include software. Technical practices exists for ensuring quality for any type of product. With non-software development, what are ways you can think of to build in quality up front?

Less product and process waste

With agile development, wasted time and features are reduced through a number of strategies, including the following:

- **Just-in-time (JIT) elaboration:** Amplification of only the currently highest-priority requirements means that time isn't wasted working on details for features that might never be developed.

- **Customer and stakeholder participation:** Customers and other stakeholders can provide feedback in each sprint, and the development team incorporates that feedback into the product as soon as the next sprint. As development and feedback continue, value to the customer increases.
- **A bias for face-to-face conversation:** Faster, clearer communication saves time and confusion.
- **Built-in exploitation of change:** Only high-priority features and functions are developed.
- **Emphasis on the evidence of working functionality:** If a feature doesn't work or doesn't work in a valuable way, it's discovered early at a lower cost.

Emphasis on value

The agile principle of simplicity supports the elimination of processes and tools that don't support development directly and efficiently, and the exclusion of features that add little tangible value. This principle applies to administration and documentation as well as development in the following ways:

- Fewer, shorter, more focused meetings
- Reduction in pageantry
- Barely sufficient documentation
- Joint responsibility between customer and the team for the quality and value of the product

Development teams

Agile approaches empower development teams to produce their best work under reasonable conditions. Agile methods give development teams

- A clear definition of success through joint sprint goal creation with the product owner and identification of the acceptance criteria during requirements development
- The power and respect to organize development as they see fit
- The customer feedback they need to provide value
- The protection of a dedicated scrum master to remove impediments and prevent disruptions
- A humane, sustainable pace of work

- A culture of learning that supports both personal development and product improvement
- A structure that minimizes non-development time

Under the preceding conditions, the development team thrives and delivers results faster and with higher quality.

REMEMBER

On Broadway and in Hollywood, performers who are on stage and onscreen are often referred to as "the talent." They are the reason many entertainment customers come to a show, and the supporting writers, directors, and producers ensure that they shine. In an agile environment, the development team is "the talent." When the talent is successful, everyone succeeds.

IN THIS CHAPTER

- Knowing your customers
- Understanding your customer's problems
- Preventing root causes rather than treating symptoms

Chapter 4

Agility Is about Being Customer Focused

Product development teams build and maintain products that customers need. Understanding who your customer is and the problems he or she needs solved makes product development valuable. In this chapter, you learn how teams work to understand who their customer is, the problem the customer needs solved, and the root causes rather than the symptoms.

Knowing Your Customers

"Who is my customer?" is a fundamental question everyone must ask as they begin product development. Product development teams explore this question frequently, using product and customer usage data to see trends and watch industries and markets closely. Not clearly understanding who your customer is makes product development difficult and rudderless.

Customers range from external paying customers to internal users. Sometimes the customer is even several layers away, spanning wholesalers, distributors, and retailers. Product landscapes also change dramatically and are uncertain. Each new generation of users from baby boomers to millennials brings a new set of needs or considerations. Add to that cultures, races, and ethnicities; new or

changing technologies or inventions; new laws or regulations; and even results from competitive benchmarking. Customers continually demand faster, better, and cheaper. From cars that receive operating system updates every week to wearables to home automation, customers expect more — and more often. Product providers who can meet these demands remain relevant.

Knowing who your customer is will put your product development effort on the right path. But how can you know if you're on the right path? This uncertainty can be unnerving.

Getting comfortable with uncertainty

Modern product development is fraught with uncertainty. In fact, it's the one aspect of product development that's certain. Rarely can we develop the same solution for our customer's ever-changing problems. Product providers continually question, "Have we accurately identified our customer and are we addressing the right problem?" Product development teams need to be comfortable with uncertainty.

Certainty synonyms are predictability, sureness, assurance, and certitude — aspects you would expect of a low-risk investment. Uncertainty is fraught with vagueness, ambiguity, and insecurity — aspects of a high-risk investment. Product development teams continually strive to address the highest risk and highest value for their customers. Throughout their product development journey, customers' needs become more certain as the team learns, grow, and gains experience and opportunity.

With iterative product development, the team can present at every sprint a fully functioning, working product increment — not a half-baked increment — and get feedback. At each sprint, the team aligns the product closer and closer to what the customer needs through a tight, consistent feedback loop. The team becomes more and more certain that what they're creating is what the customer wants. The team validates assumptions along the way, thus reducing uncertainty.

Being comfortable with uncertainty is important because it changes the team's perspective. It enhances the team's curiosity rather than giving them a false sense of assurance. Innovation and better ideas result from a curious team, an attribute shared by all famous inventors.

Uncertainty enables a team to channel its vagueness, ambiguity, and risk into a fruitful learning experience with growth and opportunity. Uncertainty motivates the team, which aligns with principle 5, "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done." Product development is risky business, in a good way.

Common methods for identifying your customer

Although you can identify customers and their needs in many ways, we discuss several popular methods used by product development teams. In the context of agile development, we refer to *customers* as the end users and *clients* as the people paying for the product or service.

Product owners and developers who are experts in techniques for better understanding their customer are valuable to their organizations and teams. Scrum masters who can facilitate using these techniques are likewise valuable. Each technique puts the team on the path towards certainty.

Product canvas

In *Inspired: How to Create Tech Products Customers Love* (Audible Studios), Marty Cagan wrote, "discover a product that is valuable, usable and feasible." What he meant by this, as shown in Figure 4-1, is that successful product development hits the sweet spot at the cross-section of

- **Valuable:** Will customers buy it?
- **Usable:** Do customers need it?
- **Feasible:** Can we do it?

FIGURE 4-1: The sweet spot where a product is valuable, feasible, and usable.

Many teams use a visualization technique, such as a product canvas, to start to explore and understand the key factors, partners, unique value proposition,

problems, and potential solutions that may contribute to finding the sweet spot, where value, usability, and feasibility meet.

The product canvas is a collaborative tool that enables teams, in a brief amount of time, to accomplish two tasks: One, start identifying desired goals or product outcomes. Two, validate assumptions about the problem to solve for the customer and ready the team for development. Product canvas exercises can inspire a clear product vision. You learn more about product vision in Chapter 9.

Teams use the product canvas as a visualization activity to create a shared understanding of the customer and his or her needs. The tool serves as a starting point for the team's assumptions and enables them to dive deep in gathering new product insights.

TIP

Product development teams create a shared understanding through visualization. The most effective communication medium for them is face-to-face with a whiteboard, flipchart, or other holistic surface. Why? Whiteboards allow people to not only explain but also draw what they mean. Peter Lencioni, author of "The Five Dysfunctions of a Team" said, "If you could get all the people in an organization rowing in the same direction, you could dominate any industry, in any market, against any competition, at any time." A shared understanding helps teams row in the same direction.

Many variations of canvases are available, such as a lean canvas or a business opportunity canvas. They all serve a similar purpose of organizing ideas, challenging assumptions, and collaborating to find a strategic direction. Figure 4-2 demonstrates the product canvas we often use with agile product teams. The left half addresses market and customer issues, while the right half addresses product and business issues. The left half defines the customer segment, customer problems with alternatives, the value proposition, channels, and revenue projections. The right half defines the solution, key stakeholders, success factors, resources, partners, and costs. Both halves enable a team to do a more detailed evaluation of their product.

Following are some categories a team might use in building its product canvas:

- **Customer segment:** Describe the target customer segment that needs the problem solved. For whom is the value being created?
- **Early adopter:** Describe the initial target customer. Remember, your product can't be everything to everybody — at least not at first. What market segments are opportunities for testing your product idea first?
- **Problem:** Describe the primary problem experienced by the target customer segment.

Product Canvas™

Customer Segment	Problem	Unique Value Proposition	Solution	Key Success Factors
For whom are we solving a problem? For whom are we creating value?	What is the top problems faced by our target Customer Segment?	How are we uniquely going to solve our customer's problems or satisfy their needs? A single, clear, compelling message that states why you are different and worth buying.	What are the primary ways we are going to solve the problem faced by our Customer Segment?	How will we measure success? What key metrics are we trying to move?
Early Adopter: Who is a potential early user of the solution?	**Existing Alternatives** How are they solving the problem today?	**Channels** How will we get (acquire), keep (retain) and grow (sell more to existing) customers? Get/Acquire: How will we drive awareness, interest, activation, usage? Keep/Retain: How will we keep customers coming back? Grow: How will we up-sell/cross-sell customers, encourage referrals?	**Key Stakeholders** Who are the most important stakeholders whose buy-in we need? Which executives do we need to convince? Who will be our executive champion? Who are the key influencers to these stakeholders? Who else do we need to include in our coalition-of-the-willing?	**Key Resources & Partners** What are the critical internal and external resources we need to deliver the solution to the customer?
Revenue/Business Value What is the business value of delivering the product/service/capability? (E.g., drive revenue, save money, increase CSAT, competitive differentiator, market positioning, etc.)		**Cost Structure** What are most important costs inherent in our product model? Which Key Resources are most expensive? Which key activities are most expensive – product development, marketing, customer support?		

MARKET/CUSTOMER **PRODUCT/BUSINESS**

© Shardul Mehta 2015
http://streetsmartproductmanager.com

Product Canvas™ is adapted from The Business Model Canvas (http://www.businessmodelgeneration.com) and is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of the license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Shardul Mehta

FIGURE 4-2: The product canvas.

- **Existing alternatives:** Describe available alternatives to your product.
- **Unique value proposition:** Describe a single, clear, and compelling message that states why or how your product is different and worth buying.
- **Channels:** Describe the channels for acquiring, retaining, and increasing customers. List ideas for creating awareness, interest, activation, and usage. Describe what will entire customers to return and encourage referrals. List upselling opportunities.
- **Solution:** Describe the ways the target customer segment's problems will be solved.
- **Key stakeholders:** List the people who are most important to your product. This list may include people whose buy-in and support you need, executives, and influencers. Who are the people you can trust to criticize your product and tell you the truth?
- **Key success factors:** Describe how success will be measured — measure outcomes not outputs. Are there key metrics you can use to test your hypotheses?

REMEMBER

Using metrics allows product development teams to evaluate the success or failure of each product release. Metrics help them see if they truly understand and are solving their customer's problems. Product development teams don't consider their work completely done until those objectives are met. The product canvas and other tools outlined in this chapter can help identify these business metrics early. Agile product development techniques increase the likelihood that bad or wrong ideas are quickly discarded.

- **Key resources and partners:** Describe the critical internal and external people, equipment, or resources needed to deliver the solution to the customer.
- **Revenue/business value:** Describe the business value of delivering the product, service, or capability. Consider what will drive revenue, save money, increase customer satisfaction, differentiate you from your competition, improve market positioning, and more.
- **Cost structure:** Describe the important costs inherent in the product model. Identify what will be most expensive, for example, resources, activities, development, marketing, or support.

Using the foundation of a product canvas not only helps the team to better understand the customer and desired outcomes but also enables the product owner to build a concise yet strategic product vision statement with a supporting product roadmap. The product vision statement and roadmap are discussed in Chapter 9.

Customer map

Other customer-focused mapping tools can be powerful visualization activities for a team seeking to better understand its customers. We introduce two common customer mapping tools: a journey map and an empathy map.

A *journey map* helps the team visualize the day-to-day experience a customer goes through when accomplishing a goal or addressing a specific problem. Goals are followed by actions in a timeline format, by calling out the user's emotions or thoughts. The journey map creates a narrative. The insights gained inform product designs. Figure 4-3 outlines the flow of a journey map and the relationships between the customer's goals and his or her steps, insights, and emotions.

An *empathy map* (see Figure 4-4) helps the team think through the user's emotions and senses. It explores what the customer sees, hears, thinks, feels, and does. It identifies pain the customer experiences and how the product can give the customer the gains he or she seeks.

FIGURE 4-3: A customer journey map.

FIGURE 4-4: A customer empathy map.

Teams build customer maps together, learning and digging deeper into their customer's needs, motivations, and challenges. They openly discuss their perceptions, observations, and insights to validate their understanding along the way.

Job-to-be-done lens

Jobs theory, developed by Clayton Christensen and the Harvard Business School in response to his overwhelmingly popular theory on disruptive innovation, refers to an approach used to discover functionally, socially, and emotionally why customers make the choices they do. According to the theory, products and services are hired to help the customer make progress. They call the customer's progress the *job-to-be-done,* which when understood enables significant innovation. Beware that if a product can be hired, it can also be fired.

UNDERSTANDING THE JOB-TO-BE-DONE OF A CONDOMINIUM

Clayton Christensen tells the story of a home builder in Detroit in 2006 who sold condominiums to down-sizers — retirees looking to move out of the family home, divorced parents, and single parents. In an effort to increase home sales, the builder considered many options with a focus group. They considered adding a bay window, changing colors, or making structural changes, and then evaluated to see whether their change made a difference in sales. The results were negligible.

The builder decided to interview his previous customers to learn the reasons and timeline that led to their condo purchase. Unfortunately, no patterns emerged except one: Every buyer mentioned a concern about their dining room table. Confused at first, the builder realized that while the table was somewhat inexpensive, well-used and possibly dated, it was central to the family's activities: homework, birthdays, holidays, and other gatherings. The dining room table represented their family.

Buyers were hesitant about buying condos not because of the physical structure but because of the anxiety associated with having to give up something that had profound meaning. The job-to-be-done was to enable buyers to maintain family gatherings, memories, and traditions even when downsizing.

In response, condominiums were redesigned with a larger space for the dining room table and each new buyer was given a two-year storage unit with a sorting area for their belongings. Christensen cited that these changes allowed the builder to not only differentiate himself from his competitors but also raise his price. Even better, in an industry experiencing a severe economic decline at the time, his business grew by 25 percent. Understanding the job-to-be-done made all the difference.

The jobs-to-be-done approach adheres to four principles:

- ***Job* is shorthand for what an individual really seeks to accomplish in a given circumstance.** Product development teams need to understand the experience the customer is trying to create, which is not a straightforward task.
- **The circumstances are more important than customer characteristics, product attributes, new technologies, and trends.** This principle speaks to seeing the innovation through the lens of the customer's circumstances.
- **Good innovations solve problems that formerly had only inadequate solutions — or no solution.** If customers see only two options, a third option

that addresses all the relevant criteria can help fickle observers become customers.

» **Jobs are never simply about function — they have powerful social and emotional dimensions.** Understanding the social and emotional dimensions of a customer's choice makes all the difference in the customer's purchasing decisions.

Christensen shares how viewing products through the jobs-to-be-done lens significantly increased sales of his sample populations. Specific examples he cites include milk shakes, condominiums, and even Reese's peanut butter cups.

Product development teams use the job-to-be-done theory throughout their product development, particularly as they evaluate the timeline from when customers identify a need to their purchase. Figure 4-5 shows an example job-to-be-done timeline. Close interactions with stakeholders and customers help validate their assumptions.

FIGURE 4-5: The job-to-be-done timeline.

Product development teams can benefit by considering the job their product or service must fulfill for the customers. Better understanding the job-to-be-done requires customer interviews so you can truly understand their needs.

Customer interviews

What better way to understand a customer's needs than to talk to the customer? Teams interview customers because they value "customer collaboration over contract negotiation" and "individuals and interactions over processes and tools." Customer interviews are critical for enabling progressive elaboration techniques such as decomposition and story mapping. See Chapter 9 to learn about progressive elaboration.

The key for performing a successful interview is to get customers to talk by allowing them to tell stories about their problems and imagine possible solutions. You escape from your perspective into the customer's. Interviewees will discover things they didn't know and interviewers will discover whether they're making wrong assumptions. Consider the interview as an opportunity to validate the team's assumptions and to vet both good and bad ideas.

Table 4-1 outlines the types of questions interviewers should and shouldn't ask.

The types of question on the left invite storytelling. They encourage the customer to answer beyond any biases you may have. The examples on the right give significant constraints to customers on how they can answer the question, and don't encourage much conversation or storytelling. Instead, get them talking.

TABLE 4-1 Customer Interview Do's and Don'ts

Do This	Don't Do This
Use open ended questions beginning with why, how, and what. Follow up with more questions such as, "Tell me more."	Start with a pitch or description of your product idea.
Use questions that invite storytelling.	Multiple choice or one word/one phrase responses.
Tell me about. . .	Do you like. . .? Do you want. . .?
How do you know. . .?	Would you use this?
Can you give me an example of. . .?	What do you think of our product?
What do you wish you could do. . .?	What requirement should we add?
What do you know about. . .?	How often do you. . .?
How do you feel about. . .?	Would you ever use. . .?
How would life be different with. . .?	Would you pay money for. . .?
When was the last time you. . .?	How often would you. . .if you could. . .?
What happened when you. . .?	Take vague terms like "difficult," "expensive," "complex" at face value.
Ask about past behavior, rather than hypothetical/ideal	
What do you mean by. . .?	

Product discovery workshop

Product development teams use *product discovery workshops* to gather product ideas and insights from stakeholders and scrum team members. These workshops have many different formats and applications.

Workshops, if done correctly, allow creative juices to flow. What is key is creating a safe environment where ideas can be shared openly. Timeboxes (setting a time limit on the discussion) for topics are helpful, as well as plenty of Post-it Notes and markers, and the freedom to diverge, converge, explore, and discover.

Some people, especially introverts, find that receiving prepared agendas beforehand is helpful for getting their thoughts flowing. Agendas can also help everyone to arrive prepared and ensure that the workshop objectives are accomplished. Be careful to not use the agenda to provide excess structure, however. Lightweight structure, particularly with workshops, can help the discussion go where it needs to go.

The frequency and participants of the workshops depend on the team, the product, and the problem to solve.

Figuring Out the Problem Your Customer Needs to Solve

The solution to a customer's problem may not be what you think it should be. It may not even be what the customer thinks it should be at first. It may not be what the customer tells you he or she wants. The customer paying you to develop the product and end-users may even have conflicting views of what's needed at first. One of the most difficult things about product development is that customers often don't know what they want until they interact with it.

Clearly understanding who your customer is makes understanding their problems much easier. Teams understand this and take the time to understand customer problems and their root causes. Addressing symptoms can be costly. Charles Kettering, owner of 186 patents, said, "A problem well-stated is half-solved." Several methods exist for digging deeper into customer problems and their root causes.

Using the scientific method

Central to agile product development is the scientific method. *The scientific method,* which has characterized natural science since the seventeenth century, consists of systematic observation, measurement, and experiment, and formulating, testing,

and modifying hypotheses. Product development teams ask questions to form hypotheses, test those hypotheses, and then evaluate the results, over and over again.

Teams use the scientific method to move their understanding of customer needs and problems from uncertainty to certain, as shown in Figure 4-6.

FIGURE 4-6: The scientific method.

To use the scientific method, the team begins by making observations or asking questions about a particular customer problem. The team forms a hypothesis about that problem. Then it discusses possible solutions in the form of requirements to test the hypothesis. Each requirement identifies the desired outcomes to be measured so the team can evaluate the results.

The team builds the product increment during the sprint and gathers feedback at the sprint review. Feedback is prioritized against the backlog. Product owners release the increment when they feel enough value is created and the requirement is ready to be tested in the wild. The team then positions itself for capturing more customer feedback. The cycle repeats as necessary. Every product backlog item should achieve a desired outcome.

TIP

Teams use the scientific method not only for product development but also to improve the development process during retrospectives. At each sprint retrospective, they form a new hypothesis for improving, test the hypothesis with an experiment during the sprint, and then evaluate the results — another example of how the scientific method is central to agility.

THE SCIENTIFIC METHOD AND R&D

Research and development clients have a hypothesis based on a set of premises. Each sprint empirically validates or invalidates that premise as they collect feedback from subject matter experts who critique the empirical findings. If the validation is weak, the scrum team iterates on that premise until it is defensible. If the validation is strong, the team progresses to the next premise. At the end, the team writes a white paper or journal article saying "This hypothesis is either valid or invalid and here is how we empirically know."

Failing early is a form of success

The mantra "fail early, fail often, but always fail forward" as stated by John C. Maxwell motivates teams to accelerate their learning through continuous and early delivery of value. In other words, try a few inexpensive experiments and then evaluate the results before investing more. This allows the team to spend the least amount of time and effort on solutions that may be thrown away. Understanding the problems customers need solved can take a few attempts. Product development teams embrace failure as learning, especially when they learn inexpensively.

REMEMBER

It's not a failure when you learn something after only a week's worth of effort!

With agile approaches to product discovery, invalidating assumptions is as good as validating assumptions. The goal is to learn early what does and doesn't work before a larger investment is made.

REMEMBER

The *term minimum viable product (MVP)* is a concept introduced in lean startup by Eric Ries to help teams learn through failure quickly and cheaply. The point is to quit trying to create everything before you deliver and test anything. Product owners and development teams look for the next minimum viable product increment needed to test and validate whether their solution is viable. Agile Principle 10: "Simplicity — the art of maximizing the amount of work not done — is essential."

NORDSTROM INNOVATION LAB

Nordstrom used a team called Innovation Lab to experiment with various product ideas in its stores. During the course of a week, the team would run a sprint to prove an idea by creating a product based on rapid feedback loops directly with customers in the store.

In one case, the team created an iPad app for using photos to compare sunglasses while choosing sunglasses to purchase. Every time the development team implemented a new requirement, it would give the app to a sales rep to use with a customer, and then gather feedback right away about how the customer used it or liked it. The team scrapped some requirements customers didn't like and, based on feedback, created requirements the team hadn't thought of in its planning.

At the time, the customers and the staff liked the product. However, the rest of the story is what's interesting. The product is not used at Nordstrom's today. Why? We don't know. Does that mean it was a failure? We don't think so. If they wasted any time on developing the product, it was only one week. They failed cheap, which freed their team to work on the next, more valuable problem.

Defining customer-focused business goals

Agile product development is purpose driven, that is; it always begins by clearly defining and understanding the desired outcomes for the customer. Every release, sprint, requirement, and task is planned with a specific outcome in mind. Establishing goals in customer terms enables a team to keep its focus on the customer, maintain strategic stability, and reserve tactical flexibility to accomplish the problem to be solved.

TIP

Product development teams achieve desired outcomes. They continually validate that their ladder is up against the right wall. Creating more of a bad product faster is not a good idea.

Several goal formats are available to help a team focus on the customer. The most effective goal statements are simple, emotionally engaging, motivating, and memorable. Product development teams discuss the goal every day during their daily scrum.

REMEMBER

US President John F. Kennedy communicated the following goal in 1961: "The US should commit itself to achieving the goal, before this decade is out, of landing a man on the Moon and returning him safely to the Earth." The goal was brief, emotionally engaging, motivating, and powerful! Obtaining this goal was the desired outcome that drove product development to follow.

Product release goals begin with "Enable my customer to . . .." Iteration or sprint goals begin with "Demonstrate the ability to . . .." Even at the individual requirement level (sometimes referred to as a user story), you describe the requirement using goal statements because they are succinct, define the target audience, and define what benefit will be received and why. Regardless of the specific pattern used, it's important to keep the desired outcomes for the customer front and center. (Read Chapter 10 to learn more about release planning, sprint planning, and user stories.)

In this way, everything a team does throughout the sprint benefits the customer and solves his or her problem in some way. Each task is more meaningful, and each roadblock must be overcome. High-performing teams use goals to accomplish business outcomes iteratively, step by step, learning by learning.

Goals should be SMART, or Specific, Measurable, Acceptable, Realistic, and Timebound, as attributed to Peter Drucker's Management by Objectives in his 1954 book *The Practice of Management*.

Story mapping

Story mapping, made popular by Jeff Patton and Peter Economy in their book *User Story Mapping* (O'Reilly Media, Inc.), is another visualization activity used by product development teams. Similar to a journey map, a *story map* helps a team gain a shared understanding of the process or journey users and customers need their product to take as well as the various alternatives for iteratively improving the experience. The story map is organized to clarify for everyone the minimum viable product as well as a path for future releases. Teams can see a more holistic view of how their ideas fit into the overall user experience.

For example, Figure 4-7 outlines how a user might submit a transaction dispute if he or she were charged incorrectly while using a mobile banking application. The sequence with the least amount of functionality to accomplish the desired outcome is across the first row. Other alternatives for eventually improving the user's experience at each step are prioritized vertically.

FIGURE 4-7: A user story map.

By mapping and visualizing the process as a team, the individual requirements make a lot more sense. The team can draft the product backlog together, creating a cohesive team-defined solution based on the customer's experience.

Liberating structures — simple rules to unleash a culture of innovation

The last but certainly not least approach we discuss is liberating structures, which are a set of microstructures compiled and shared by Keith McCandless and Henri Lipmanowicz. *Liberating structures* (`www.liberatingstructures.com`) provide

alternative ways of structuring and facilitating collaboration. They enable liberation of content and ideas through minimally structured interactions and are designed to help people understand and solve complex problems.

TECHNICAL STUFF

Macrostructures refer to organizational policies and processes that constrain microstructure activities. Conventional (non-liberating) microstructures include activities such as traditional presentations, discussions, brainstorms, and reporting.

A menu of dozens of liberating structures defines an invitation for participation, how to arrange the physical space and materials needed, how participation is distributed, how groups are configured, and a sequence of steps and timeboxes. Each liberating structure can be used for brainstorming, discussing, and discovering solutions to complex and challenging problems.

An example of a liberating structure often used by product development teams is *1-2-4-All,* which is designed to engage everyone in the workshop to simultaneously generate questions, ideas, and suggestions. The exercise begins with a one-minute silent self-reflection on a shared challenge, framed as a question. Next, pairs are formed and spend two minutes building on the ideas. Then groups of four spend four minutes noting the differences and similarities and developing new ideas. Lastly, each group conducts a five-minute presentation of one important idea that stood out to them. A bell or timer helps everyone work within their timebox. The cycle can be repeated, if needed. In a few minutes, the group is able to generate more and better ideas harnessing the mind-power of their collective group.

Liberating structures are perfect for the types of problems agile product development teams are solving for customers every day.

Understanding Root Cause Analysis

Root cause analysis (RCA) is a systematic approach for identifying root causes of problems or events as well as an approach for responding to them. It's based on the basic idea that effective management requires that you not just deal with symptoms of problems but also find a way to prevent problems.

RCA is a critical aspect to consider when evaluating the problem your customer is trying to solve. Like physicians, product development teams strive to address the problem at the root rather than symptoms. When the root is addressed, or even prevented, the symptoms disappear. Addressing the root cause can be more challenging but often yields a more elegant and simple solution.

Following are the typical steps teams follow as they perform root cause analysis:

1. **Define the problem.** Collaboratively create a problem statement.
2. **Collect data.** Assemble data to support the problem.
3. **Identify possible causal factors.** List factors that cause the problem from the gathered data.
4. **Identify the root causes.** List the root causes.
5. **Recommend and implement solutions.** Form a hypothesis to test.

TIP

RCA is extremely useful not only to product owners and developers when solving customer-focused problems, but also to scrum masters as they work to remove impediments or teach the team to do so. Impediments need to be resolved at the root so that they never raise their ugly head again. Scrum masters resolve impediments not only tactically (remove reactively) but also strategically (prevent proactively).

At any time, product development teams may use several approaches to better understand root causes of the problems they're trying to solve. In this section, we discuss three approaches:

- **The Pareto (80/20) rule:** If 20 percent of root causes are addressed, they can positively benefit the remaining 80 percent.
- **Five Why's:** A problem can be debated five generations or layers back to discover the root cause.
- **Ishikawa (or Fishbone diagram):** A problem can be evaluated across multiple categories — typically people, process, tools, and culture — to expose root causes.

Pareto rule

The Pareto rule, named after economist Vilfredo Pareto (1985), also known as the 80/20 rule, specifies that 80 percent of the effects come from 20 percent of the causes. This rule is a reminder that the relationship between inputs and outputs is not balanced.

Teams use the Pareto rule as part of root cause analysis as they gather data from help desk incidents or customer complaints, for example. From the data, the team is able to categorize 20 percent of the top failures as affecting 80 percent of the

reported incidents. Consider a few other applications of the Pareto rule to product development:

- 80 percent of the complaints come from 20 percent of your customers.
- 80 percent of the product's functionality comes from 20 percent of the developer's effort.
- 80 percent of profits come from 20 percent of product customers.
- Of all the apps on a mobile phone, users frequently use about 20 percent.
- Getting the 20 percent right yields a higher return and solves root causes, reducing symptoms.

TIP

The Pareto rule is often used by product owners as they manage their product backlog. According to the rule, 20 percent of the backlog will create 80 percent of the value for the customer. Prioritize the top 20 percent first. Then reevaluate, but be ready to move on to the next investment idea after the 20 percent is met. Agile projects run out of value before they run out of time or money.

Five why's

The *five why's*, also known as the *nine why's* in liberating structures, is a method that uses a series of *why* questions to drill down into successive layers of a problem. The basic idea is that each time you ask why, the answer becomes the basis of the next why. It's a simple tool that is useful for less complex problems.

Mind maps, which start with a central problem statement, can be helpful for guiding the group or team through this analysis. As the group reaches the fifth (or ninth) why, you can be more certain that the group is nearing a root cause.

One application of this technique is to more deeply analyze the results of a Pareto analysis. Here's an example of how to use the five why's with the same mobile banking app we considered earlier in this chapter:

Problem: Customers complain about excessive overdraft fees.

Why 1: Why are customers complaining about excessive overdraft fees?

Answer 1: Customers are not notified that their account is overdrawn.

Why 2: Why are customers not notified about overdrafts?

Answer 2: An indicator or alert is not provided to the customer when an overdraft exists.

Why 3: Why does the customer not see an indicator or receive an alert when an overdraft exists?

Answer 3: Bank policy requires a physical letter be sent when an overdraft exists.

Why 4: Why does bank policy require a physical overdraft letter be sent to customers?

Answer 4: The new account contract, signed by the customer, stipulates a physical letter.

Why 5: Why does the new account contract stipulate overdraft letters will be mailed?

Answer 5: The mobile app did not exist when the new account contracts were created.

Of course, you may need to ask "Why?" more than five times to solve the problem. The point is to peel away surface-level issues to get to the root cause. Five why's can be used whenever the team is debating a problem or needs to better understand a problem with either the product or the development process.

Ishikawa (fishbone)

An *Ishikawa diagram*, or *fishbone diagram*, is a causal diagram created by Kaoru Ishikawa in 1968 that shows the causes of a specific event. Ishikawa diagrams sort possible causes into various categories that branch from the original problem, as shown in Figure 4-8. Also called a cause-and-effect diagram, an Ishikawa diagram may have multiple sub-causes branching off each identified category.

FIGURE 4-8: The Ishikawa diagram.

To begin, teams align on a single problem statement. They choose wording that everyone can agree to. Multiple iterations of the problem statement may be required until everyone is comfortable.

After the problem statement is clear, the team chooses categories for the spine of the fishbone. Categories can be whatever the team believes will be helpful, such as analyzing the problem across people, process, culture, and tools. Other helpful categories are materials, environment, management, product line, country, and state.

Combining the Ishikawa diagram with a five why's analysis can help trace each fishbone to its root causes. Note that the analysis often results in the same root causes mentioned multiple times throughout. This is a sure sign that you're on the right track to truly understanding the problem rather than a symptom.

In this chapter, we discussed how understanding customers and their problems are central to agility. Forgetting them leads to waste and regret. Remembering them leads to innovation and success. Although identifying customers and their problems can be difficult to grasp, you've learned several techniques to guide your team conversations to move your uncertain product iteratively closer towards certainty.

With root causes understood, teams, like physicians, can address problems rather than symptoms. Various agile frameworks are available for helping product development teams ferret out root causes. Those frameworks are described in the next chapter.

2 Being Agile

IN THIS PART . . .

Understand what it means to be agile and how to put agile practices into action.

Get an overview of the three most popular agile approaches, and discover how to create the right environment for both virtually distributed and physically collocated space, communication, and tools to facilitate agile interactions.

Examine the behavior shift in values, philosophy, roles, and skills needed to operate in an agile team.

Discover the advantages of small, self-organizing, self-managing, long-lived, and enduring "permanent" teams.

IN THIS CHAPTER

- Applying agile practices
- Understanding lean, scrum, and extreme programming
- Connecting agile techniques

Chapter 5
Agile Approaches

In previous chapters, you read about the history of agile product management. You may have even heard of common agile frameworks and techniques. Are you wondering what agile frameworks, methods, and techniques actually look like? In this chapter, you get an overview of three of the most common agile approaches used today.

Diving under the Umbrella of Agile Approaches

The Agile Manifesto and the agile principles on their own wouldn't be enough to launch you into agile product development, eager as you might be to do so. The reason is that principles and practices are different. The approaches described in this book, however, provide you with the necessary practices to be successful.

Agile is a descriptive term for a number of techniques and methods that have the following similarities:

- Demonstrating valuable and potentially shippable functionality in short iterations called *iterative development*
- Emphasis on simplicity, transparency, and situation-specific strategies

- Cross-functional, self-organizing teams
- Working functionality as the measure of progress
- Responsiveness to changing requirements

Synonyms of the word *agile* — including resilient, flexible, nimble, adaptive, lightweight, and responsive — give additional insights into what it means to *be* agile.

Similarly, throughout this book, we reference agile teams. *Agile teams,* which include scrum teams (scrum being the most popular agile framework), are teams who adhere to the agile values and principles to become more resilient, flexible, nimble, adaptive, lightweight, and responsive in meeting their customer's needs.

Agile product development is an empirical approach. In other words, you do something in practice and adjust your approach based on experience rather than theory.

With regards to product development, an empirical approach is braced by the following pillars:

- **Unfettered transparency:** Everyone involved in the process understands and can contribute to the development of the process.
- **Frequent inspection:** The inspector must inspect the product regularly and possess the skills to identify variances from acceptance criteria.
- **Immediate adaptation:** The development team must be able to adjust quickly to minimize further product deviations.

A host of approaches have agile characteristics. Three, however, are commonly used: lean product development, scrum, and extreme programming (XP). These three approaches work perfectly together and share many common elements, although they use different terminology or have a slightly different focus. Broadly, lean and scrum focus on structure. Extreme programming does that, too, but is more specific about development practices, focusing more on technical design, coding, testing, and integration. (From an approach called *extreme programming,* this type of focus is to be expected.)

Most organizations we work with that are using an agile approach for product development are usually working in an environment that is lean, with constant attention to limiting work in progress, wasteful practices, and process steps; using scrum to organize their work and expose progress; and using extreme programming practices to frontload quality. Each of these approaches is explained in more detail later in this chapter.

Like any systematic approach, agile techniques didn't arise out of nothing. The concepts have historical precedents, some of which have origins outside software development, which isn't surprising given that software development hasn't been around that long in the history of human events.

The basis for agile approaches is not the same as that of traditional project management methodologies such as waterfall, which was rooted in a defined control method used for World War II materials procurement. Early computer hardware product development pioneers used the waterfall process to manage the complexity of the first computer systems, which were mostly hardware: 1,600 vacuum tubes but only 30 or so lines of hand-coded software. (See Figure 5-1.) An inflexible process is effective when the problems are simple and the marketplace is static, but today's product development environment is too complex for such an outdated model.

SSEM — "Baby"

Tom Kilburn's program for "Baby"

FIGURE 5-1: Early hardware and software.

Enter Dr. Winston Royce. In his article, "Managing the Development of Large Systems," published in 1970, Dr. Royce codified the step-by-step software development process known as waterfall. When you look at his original diagram in Figure 5-2, you can see where that name came from.

Over time, however, the computer development situation reversed. Hardware became repeatable through mass production, and software became the more complex and diverse aspect of a complete solution.

FIGURE 5-2: The origins of waterfall.

The irony here is that, even though the diagram implies that you complete tasks step by step, Dr. Royce himself added the cautionary note that you need iteration. Here's how he stated it:

> "If the computer program in question is being developed for the first time, arrange matters so that the version being delivered to the customer for operational deployment is actually the second version insofar as critical design/operations areas are concerned."

Royce even included the diagram shown in Figure 5-3 to illustrate that iteration.

Now, we're not sure if the page with this diagram was stuck with chewing gum to other pages, but the software development community by and large lost this part of the story. Historically, product development in general has been constrained by this linear and defined process control thinking. By embracing the idea that you might not know everything when you first start developing a component and will probably have to revisit your assumption of what is the right thing to ensure that it becomes what your customer needs, you take the first step to becoming more agile. Agile techniques might have come to prominence 40 years earlier if people had taken Dr. Royce's advice to heart!

This story is specific to technology product development, but iterative, empirical process controls apply to non-software products as well.

FIGURE 5-3: Iteration in waterfall.

Reviewing the Big Three: Lean, Scrum, and Extreme Programming

Now that you have a brief history of the waterfall approach to project management, you're ready to find out more about three popular agile approaches to product development: lean, scrum, and extreme programming.

An overview of lean

Lean has its origins in manufacturing. Mass-production methods, which have been around for more than 100 years, were designed to simplify assembly processes (for example, putting together a Model-T Ford). These processes use complex, expensive machinery and low-skilled workers to inexpensively churn out an item of value. The idea is that if you keep the machines and people working and stockpile inventory, you generate a lot of efficiency.

The simplicity is deceptive. Traditionally, mass production requires wasteful supporting systems and large amounts of indirect labor to ensure that manufacturing continues without pause. It generates a huge inventory of parts, extra workers, extra space, and complex processes that don't add direct value to the product. Sound familiar?

Cutting the fat as lean emerges in manufacturing

In Japan in the 1940s, a small company called Toyota wanted to produce cars for the Japanese market but couldn't afford the huge investment that mass production requires. The company studied supermarkets, noting how consumers buy just what they need because they know there will always be a supply and how the stores restock shelves only as they empty. From this observation, Toyota created a just-in-time process that it could translate to the factory floor.

The result was a significant reduction in inventory of parts and finished goods and a lower investment in machines, people, and space.

One big cost of the mass production processes at the time was that humans on the production line were treated like machines: People had no autonomy and could not solve problems, make choices, or improve processes. The work was boring and set aside human potential. By contrast, the just-in-time process gives workers the ability to make decisions about what is most important to do next, in real time, on the factory floor. The workers take responsibility for the results. Toyota's success with just-in-time processes has helped change mass manufacturing approaches globally.

Understanding lean and product development

The term *lean* was coined in the 1990s in *The Machine That Changed the World: The Story of Lean Production* (Free Press) by James P. Womack, Daniel T. Jones, and Daniel Roos. eBay was an early adopter of lean principles for product development. The company led the way with an approach that responded daily to customers' requests for changes to the website, developing high-value features in a short time.

The focus of lean is maximizing business value and minimizing activities outside product development. Mary and Tom Poppendieck discuss a group of lean principles on their blog and in their books on lean software development. Following are the lean principles from their 2003 book, *Lean Software Development* (Addison-Wesley Professional):

» **Eliminate waste.** Doing anything that is beyond barely sufficient (process steps, artifacts, meetings) slows down the flow of progress. Waste includes

failing to learn from work, building the wrong thing, and thrashing (context switching between tasks or objectives) — which results in only partially creating lots of product features but not completely creating any.

- **Amplify learning.** Learning drives predictability. Enable improvement through a mindset of regular and disciplined transparency, inspection, and adaptation. Encourage an organization-wide culture that allows failure for the sake of learning from it.
- **Decide as late as possible.** Allow for late adaptation. Don't deliver late, but leave your options open long enough to make decisions at the last responsible moment based on facts rather than uncertainty — when you know the most. Learn from failure. Challenge standards. Use the scientific method by experimenting with hypotheses to find solutions. We discuss the scientific method in more detail in Chapter 4.
- **Deliver as fast as possible.** Speed, cost, and quality are compatible. The sooner you deliver, the sooner you receive feedback. Work on fewer things at once, limiting work in progress and optimizing flow. Manage workflow, rather than schedules. Use just-in-time planning to shorten development and release cycles.
- **Empower the team.** Working autonomously, mastering skills, and believing in the purpose of work can motivate development teams. Managers do not tell developers how to do their jobs but instead support them to self-organize around the work to be done and remove their impediments. Make sure teams and individuals have the environment and tools they need to do their job well.
- **Build in integrity (quality).** Establish mechanisms to catch and correct defects when they happen and before final verification. Quality is built in from the beginning, not at the end. Break dependencies, so you can develop and integrate functionality at any time without regressions.
- **See the whole.** An entire system is only as strong as its weakest link. Optimizing only one part will sub-optimize the whole system. Solve problems, not just symptoms. Continually pay attention to bottlenecks throughout the flow of work and remove them. Think long-term when creating solutions.

Understanding kanban

Beyond the lean principles, one of the most common lean approaches used by product development teams is kanban, sometimes referred to as *lean-kanban*. Adapted from the Toyota Production System approach, *kanban* is essentially a method for removing waste to improve flow and throughput in a system.

Kanban practices can be applied in almost any situation because they're designed to start with where you are — you don't have to change anything about your existing workflow to get started. Kanban practices include the following:

- Visualize.
- Limit work in progress (WIP).
- Manage flow.
- Make policies explicit.
- Implement feedback loops.
- Improve collaboratively, evolve experimentally (using models and the scientific method).

The last three practices are commonly found in other agile frameworks, such as scrum and XP (both discussed later in this chapter). The first three enhance effectiveness for product development teams:

- **Visualize:** Visualizing a team's workflow is the first step in identifying potential waste. Traditional bloated processes exist in many organizations but do not reflect reality, even if visualized. As teams visualize the flow of their work (on a whiteboard, on a wall, or in a drawing) and identify where productivity breaks down, they can easily analyze the root cause and see how to remove the constraint. And then do it again, and again, and again.

  TECHNICAL STUFF

  Kanban is a Japanese word that has various meanings depending on which alphabet it's written in, but in essence it means *sign*, *large visual board*, *signal cards*, or *visual signal.* Hanging on the factory wall or the development workspace wall, where everyone can see it, the kanban board shows the items that teams need to produce next. Slotted into the board are cards representing units of production. As production progresses, the workers remove, add, and move cards. As the cards move, they act as a signal to workers when work or inventory replenishment is needed. Product development teams use kanban boards or task boards to expose their progress and manage their flow of work (described in more detail in Chapters 6 and 11).

- **Limit work in progress (WIP):** When teams keep starting work but don't finish it, their work in progress continues to grow. Being agile is all about getting done and receiving feedback on what has been done, so the goal is to start things only when other things are completed. Working on multiple things at once does not mean you complete them all faster — you actually complete them more slowly than if you had worked on them one at a time. When product development teams limit their work in progress, items get completed faster, speeding the pace of completing each item in their queue.

- **Manage flow:** We've all experienced what happens on a busy street during rush hour. When there are more cars than the lanes of traffic can handle, all cars move more slowly. Everyone wants to get somewhere at the same time, and so everyone has to wait longer to get there. To manage flow better, we need to regulate the entry of vehicles into the flow of traffic or increase the number of lanes of traffic where congestion is highest. Like cars in traffic, product development work items move more slowly if development team members try to take them all on at once. Working on one thing at a time and identifying and removing constraints increases the flow of all items through the system.

  TECHNICAL STUFF

  Measuring lead and cycle times helps teams monitor their management of flow. A team determines the lead time by tracking the amount of time it takes a request for functionality to go from acceptance in the queue to completion. The team knows the cycle time by tracking the time from when work begins to when it is completed. The team optimizes flow by identifying and removing bottlenecks that keep its lead and cycle times from decreasing.

The foundational principles of kanban enable its effective use with other frameworks such as scrum. These principles include the following:

- Start with what you do now.
- Agree to pursue evolutionary change.
- Initially, respect existing roles, responsibilities, and job titles.
- Encourage acts of leadership at all levels in your organization — from individual contributors to senior management.

REMEMBER

To support good product development practices, remember the following:

- Don't develop features that you're unlikely to use.
- Make the development team central to the product because doing so adds the biggest value.
- Have the customers prioritize features — they know what's most important to them. Tackle high-priority items first to deliver value.
- Use tools that support great communication across all parties.

Today, lean principles continue to influence the development of agile techniques — and to be influenced by them. Any approach should be agile and adapt over time.

An overview of scrum

Scrum, the most popular agile framework according to Digital.ai's 14th annual *State of Agile Report 2020*, is an iterative approach that has at its core the *sprint* (the scrum term for iteration). To support this process, scrum teams use specific roles, artifacts, and events. To make sure that they meet the goals of each part of the process, scrum teams use transparency, inspection, and adaptation throughout development. The scrum approach is shown in Figure 5-4.

FIGURE 5-4: The scrum approach.

Going the distance with the sprint

Within each sprint, the development team develops and tests a functional part of the product until the product owner accepts it, often daily, and the functionality becomes a potentially shippable increment of the overall product demonstrated for stakeholder feedback. Based on this feedback, the product owner determines the next steps, whether to release the increment, and what adjustments to make to the product backlog moving forward.

When one sprint finishes, another sprint starts. Releasing functionality to the market often occurs at the end of multiple sprints, when the product owner determines that enough value exists. However, the product owner may decide to release functionality after every sprint, or even as many times as needed during a sprint.

REMEMBER

A core principle of the sprint is its cyclical nature: The sprint, as well as the processes within it, repeats over and over, as shown in Figure 5-5.

You use the tenets of transparency, inspection, and adaptation on a daily basis as part of scrum:

- During a sprint, you make all progress transparent and conduct *constant inspections* to assess progress toward the sprint goal, and consequentially, toward the release goal.

FIGURE 5-5: Sprints are recurring processes.

- To organize the day, you hold a *daily scrum meeting* by coordinating what the team will work on today. Essentially, the scrum team inspects its progress toward the sprint goal and adjusts its plan to achieve the sprint goal based on the reality of the day.
- At the end of the sprint, you use a *sprint review meeting* and a *sprint retrospective meeting* to assess product improvements and team performance, respectively, and plan necessary adaptations.

These inspections and adaptations may sound formal and process-laden, but they aren't. Use inspection and adaptation to solve issues and don't overthink this process. The problem you're trying to solve today will be different than the problem you'll need to solve in the future anyway.

Understanding scrum roles, artifacts, and events

The scrum framework defines specific roles, artifacts, and events.

Scrum's three *roles* — the people working on the product — are as follows:

- **Product owner:** Represents and speaks for the business needs of the product.
- **Development team:** Performs the day-to-day technical implementation work. The development team is dedicated to the product and each team

member is *multi-skilled* — that is, although team members may have certain strengths, each member is capable of doing multiple product development jobs.

- **Scrum master:** Protects the team from organizational distractions, clears roadblocks, ensures that scrum is played properly, and continuously improves the team's environment.

Additionally, scrum teams find that they're more effective and efficient when they work closely with two non-scrum–specific roles:

- **Stakeholders:** Anyone who is affected by or has input on the product. Although stakeholders are not official scrum roles, it is essential for scrum teams and stakeholders to work closely together throughout development.
- **Agile mentor:** Someone who has experience implementing agile principles, practices, and techniques, and can share that experience with a team. Sometimes called an *agile coach*. Often this person is external to the product's department or organization, so he or she can support the scrum team objectively with an outsider's point of view.

In the same way that scrum has specific roles, scrum also has three *artifacts*, which are tangible deliverables that the scrum team makes transparent and uses to continuously inspect and adapt:

- **Product backlog:** The full list of requirements that defines the product, often documented in terms of business value from the perspective of the end user. The product backlog is fluid throughout the product lifecycle. All scope items, regardless of level of detail, are in the product backlog. The product owner owns the product backlog, determining what goes in it and in what priority.
- **Sprint backlog:** The list of requirements and tasks in a given sprint enabling the team to achieve a specific sprint goal. The product owner and the development team select the requirements for the sprint in sprint planning, with the development team breaking down these requirements into tasks. Unlike the product backlog, sprint backlog tasks can be changed only by the development team as it sees fit to ensure that it achieves the sprint goal.
- **Product increment:** The usable, potentially shippable functionality. Within the context of a single sprint, the product increment includes functionality from requirements that have been elaborated, designed, developed, tested, integrated, documented, and approved to meet the business needs for which it was intended. Whether the product is a website or a new house, the product increment should be complete enough to demonstrate its working

functionality. Product increments are released to the customer after enough shippable functionality has been demonstrated to meet the customer's business goals. In other words, it may take more than one sprint to generate enough valuable functionality to ship to the customer.

Finally, scrum also has five events:

- **Sprint:** Scrum's term for iteration. The *sprint* is the container for each of the other scrum events, in which the scrum team creates potentially shippable functionality. Sprints are short cycles, no longer than a month, typically between one and two weeks, and in some cases as short as one day. Consistent sprint length reduces variance; a scrum team can confidently extrapolate what it can do in each sprint based on what it has accomplished in previous sprints. Sprints give scrum teams the opportunity to make adjustments for continuous improvement immediately, rather than at the end.
- **Sprint planning:** Takes place at the start of each sprint. In sprint planning meetings, scrum teams decide the business goal, scope, and supporting tasks will be part of the sprint backlog.
- **Daily scrum:** Takes place daily for no more than 15 minutes. During the daily scrum, development team members inspect their progress and make adjustments to their plan to achieve their sprint goal and coordinate removal of impediments with the scrum master.
- **Sprint review:** Takes place at the end of each sprint. In this meeting, the development team demonstrates to the stakeholders and the entire organization the accepted parts of the product the team completed during the sprint. The key to the sprint review is collecting feedback from the stakeholders, which informs the product owner how to update the product backlog and consider the next sprint goal.
- **Sprint retrospective:** Takes place at the end of each sprint. The sprint retrospective is an internal team meeting in which the scrum team members (product owner, development team, and scrum master) discuss what went well during the sprint, what didn't work well, and how they can make improvements for the next sprint. This meeting is action-oriented and ends with tangible improvement plans for the next sprint.

Scrum is simple: three roles, three artifacts, and five events. Each plays a part to ensure that the scrum team has continuous transparency, inspection, and adaptation throughout product development. As a framework, scrum accommodates many other agile techniques, methods, and tools for executing the technical aspects of building functionality.

ESSENTIAL CREDENTIALS

If you are — or want to be — an agile practitioner, you may consider getting one or more agile certifications. The certification training alone can provide valuable information and the chance to practice agile processes — lessons you can use in your everyday work. Many organizations want to hire people with proven agile knowledge, so certification can also boost your career.

You can choose from a number of well-recognized, entry-level certifications, including the following:

- **Certified ScrumMaster (CSM):** The Scrum Alliance, a professional organization that promotes the understanding and use of scrum, offers a certification for scrum masters. The CSM requires a two-day training class provided by a Certified Scrum Trainer (CST) and completing a CSM evaluation. CSM training provides an overall view of scrum and is a good starting point for people starting their agile journey. See `http://scrumalliance.org`.
- **Certified Scrum Product Owner (CSPO):** The Scrum Alliance also provides a certification for product owners. Like the CSM, the CSPO requires two days of training from a CST. CSPO training provides a deep dive into the product owner role. See `http://scrumalliance.org`.
- **Certified Scrum Developer (CSD):** For development team members, the Scrum Alliance offers the CSD as a technical-track certification, requiring five days of training from a CST and passing an exam on agile engineering techniques. CSM or CSPO training can count toward two of the five days required for CSD; the remaining three days are a technical skills course focusing on extreme programming practices such as test-driven development, continuous integration, coding standards, simple design, and refactoring. You learn more about these practices in the next section. See `http://scrumalliance.org` for more about CSD certification.
- **PMI Agile Certified Practitioner (PMI-ACP):** The Project Management Institute (PMI) is the largest professional organization for project managers in the world. In 2012, PMI introduced the PMI-ACP certification. The PMI-ACP requires training, general project management experience, experience working on agile product development, and passing an exam on your knowledge of agile fundamentals. See `http://pmi.org`.

  Advanced and professional certifications are also available for all three scrum roles as well as agile leadership. See `http://scrumalliance.org`.

An overview of extreme programming

One popular approach to product development, specific to software, is extreme programming (XP). Extreme programming takes the best practices of software development to an extreme level. Created in 1996 by Kent Beck, with the help of Ward Cunningham and Ron Jeffries, the principles of XP were originally described in Beck's 1999 book, *Extreme Programming Explained* (Addison-Wesley Professional), which has since been updated. Ron Jeffries continues to update the XP practices on his website at `http://ronjeffries.com`.

The focus of extreme programming is customer satisfaction. XP teams achieve high customer satisfaction by working collaboratively with customers to develop the functionality the customer needs, when the customer needs them. New requests are part of the development team's daily routine, and the team is empowered to deal with these requests whenever they crop up. The team organizes itself around any problem that arises and solves it as efficiently as possible.

TECHNICAL STUFF

As XP has grown as a practice, XP roles have blurred. A typical development effort now consists of people in customer, management, technical, and product support groups. Each person may play a different role at different times.

Discovering extreme programming principles

Basic approaches in extreme programming are based on agile principles. These approaches are as follows:

- **Coding is the core activity.** Software code not only delivers the solution but can also be used to explore problems. For example, a programmer can explain a problem using code.
- **XP teams do lots of testing during development, not at the end.** If doing just a little testing helps you identify some defects, a lot of testing will help you find more. In fact, developers don't start coding until they've worked out the success criteria for the requirement and designed the unit tests. A defect is not a failure of code; it's a failure to define the right test.
- **Communication between customer and programmer is direct.** The programmer must understand the business requirement to design a technical solution.
- **For complex systems, some level of overall design, beyond any specific function, is necessary.** With XP development, the overall design is considered during regular *refactoring* — namely, using the process of systematically improving the code to enhance readability, reduce complexity, improve maintainability, and ensure extensibility across the entire code base.

REMEMBER

You will find extreme programming combined with lean or scrum because the process elements are so similar that they marry well.

Getting to know some extreme programming practices

In XP, some practices are similar to other agile approaches, but others aren't. Table 5-1 lists key XP practices, most of which are commonsense practices and many of which are reflected in agile principles.

TABLE 5-1 Key Practices of Extreme Programming

XP Practice	Underpinning Assumption
Whole team	The customer needs to be collocated (physically located together) with the development team and be available. This accessibility enables the team to ask more minor questions, quickly get answers, and ultimately deliver a product more aligned with customer expectations.
Planning game	All members of the team should participate in planning. No disconnect exists between business and technical people.
Customer tests	As part of presenting each desired feature, the XP customer defines one or more automated acceptance tests to show that the feature is working. The system always improves and never backslides. Automation is important because in the press for time, manual tests get skipped.
Small releases	Release value to the customer as often as possible. Some organizations release multiple times daily. Avoid building up large stores of unreleased code requiring extensive risky regression and integration efforts. Get feedback from your customer as early and as often as possible.
Simple design	The simpler the design, the lower the cost to change the software code.
Pair programming	Two people work together on a programming task. One person is strategic (the driver) and the other person is tactical (the navigator). They explain their approach to each other. They take turns as driver and navigator. No piece of code is understood by only one person. Defects can be more easily found and fixed before the code is merged and integrated with the system.
Test-driven development (TDD)	Write automated customer acceptance and unit tests before you code anything. Write a test, run it, and watch it fail. Then write just enough code to make the test pass, refactoring until it does (red-green-clean). Test your success before you claim progress.
Design improvement (refactoring)	Continuously improve design by refactoring code — removing duplications and inefficiencies within the code. A lean code base is simpler to maintain and operates more efficiently.

XP Practice	Underpinning Assumption
Continuous integration	Team members should be working from the latest code. Integrate code components across the development team as often as possible to identify issues and take corrective action before problems build on each other. XP teams promote multiple builds each day!
Collective code ownership	The entire team is responsible for the quality of code. Shared ownership and accountability bring about the best designs and highest quality. Any engineer can modify another engineer's code to enable progress to continue.
Coding standard	Use coding standards to empower developers to make decisions and to maintain consistency throughout the product; don't constantly reinvent the basics of how to develop products in your organization. Standard code identifiers and naming conventions are two examples of coding standards.
Metaphor	When describing how the system works, use an implied comparison, a simple story that is easily understood (for instance, "the system is like cooking a meal"). This provides additional context that the team can fall back on in all product discovery activities and discussions.
Sustainable pace	Overworked people are not effective. Too much work leads to mistakes, which leads to more work, which leads to more mistakes. Avoid working more than 40 hours per week for an extended period of time.

REMEMBER

Extreme programming intentionally pushes the limits of development customs by dramatically increasing the intensity of best-practice rituals, which has resulted in a strong track record of XP improving development efficiency and success.

If you're not doing software product development, you can probably replace XP with a set of technical practices specific to your industry that would build in quality throughout product development.

Putting It All Together

All three agile approaches — lean, scrum, and extreme programming (XP) — have common threads. The biggest thing these approaches have in common is adherence to the Agile Manifesto and the 12 Agile Principles. Table 5-2 shows a few more of the similarities among the three approaches.

TABLE 5-2 **Similarities between Lean, Scrum, and Extreme Programming**

Lean	Scrum	Extreme Programming
Engaging everyone	Cross-functional development team	Entire team Collective ownership
Optimizing the whole	Product increment	Test-driven development Continuous integration
Delivering fast	Sprints of four weeks or less	Small release

In addition to more extensive agile frameworks and practices, scrum also accommodates a variety of accouterments that consistently increase success with agile product development. Just like a physical home is framed to support the plumbing, electrical, ventilation, and internal convenience features, scrum provides the framework for many other agile tools and techniques to do the job well. Here is a sampling, most of which you learn more about in the following chapters:

- Product vision statement (an engaging elevator pitch — a clear inspirational statement of direction for reaching the outer boundary of the product)
- Product roadmap (a representation of the features required to achieve the product vision)
- Velocity (a tool— *not a metric* —for scrum teams to understand historical workload for each sprint and empirically predict the delivery of functionality long-term)
- Release planning (establishing a specific mid-range goal, the trigger for releasing functionality to the market)
- User stories (structuring requirements from an end-user's point of view to clarify business value)
- Relative estimation (using self-correcting relative complexity and effort rather than inaccurate absolute measures, which give a false sense of precision)
- Swarming (cross-functional teams working together on one requirement at a time until completion to get the job done faster)

IN THIS CHAPTER

- Creating your agile workspace
- Rediscovering low-tech communication and using the right high-tech communication
- Finding and using the tools you need

Chapter 6

Agile Environments in Action

Conjure up a mental picture of your current working environment. Perhaps it looks like the following setup. The IT team sits in cube city in one departmental area with the project manager somewhere within walking distance. You work with an offshore development team eight time zones away. The business customer is on the other side of the building. Your manager has a small office tucked away somewhere. Conference rooms are usually fully booked, and even if you were to get into one, someone would chase you out within the hour.

Or perhaps every one of the people we just listed is in a different physical location, maybe even in a different time zone, with no common physical workspace, as nearly all of us experienced in 2020 because of the stay-at-home orders related to COVID-19.

In addition, your project documents are stored in folders on a shared drive. The development team gets at least 100 emails a day. The project manager holds a team meeting every week and, referring to the project plan, tells the developers what to work on. The project manager also creates a weekly status report and posts it on the shared drive. The product manager is usually too busy to talk to the project manager to review progress but periodically sends emails with some new thoughts about the product.

Although this description might not describe your particular situation, you most likely see parts of it in any given corporate setting. In contrast, scrum teams develop in short, focused iterative cycles, relying on timely feedback from team members and stakeholders. To operate and become more agile, your working environment is going to have to change.

This chapter shows you how to create a working space that facilitates communication, one that will help you best become agile.

Creating the Physical Environment

Scrum teams flourish when scrum team members work closely together in an environment that supports continuous and close collaboration. As noted in other chapters, the development team members are central to success. Creating the right environment for them to operate in goes a long way toward supporting their success.

Collocating the team

If at all possible, the scrum team needs to be *collocated* — that is, physically located together. When a scrum team is collocated, the following practices are possible and significantly increase efficiency and effectiveness:

- Communicating face-to-face, taking advantage of the fullness of both verbal and nonverbal communication
- Standing up — rather than sitting — as a group for the daily scrum meeting (to keep meetings brief and on topic)
- Using simple, low-tech tools for communication
- Getting real-time clarifications from scrum team members
- Being aware of what others are working on
- Asking for help with a task
- Supporting others with their tasks

One of the benefits of collocation you don't recognize until it's gone is osmotic communication. When people work in the same physical environment, they hear what's going on around them, even if they're not paying close attention. They can join a conversation happening within earshot, contributing something that might have been missed. In addition, they can sense tension or relief as challenges are addressed by other members of the team. Absorbing information that's happening around you contributes to better informed and empowered team members.

All these practices uphold agile processes. When everyone resides in the same area, it's much easier for one person to lean over, ask a question, and get an immediate answer. And when the question is complex, a face-to-face conversation, with all the synergy it creates, is much more effective and efficient than any form of electronic communication. (See Principle 6.)

TECHNICAL STUFF

This improved communication effectiveness is due to *communication fidelity* — the degree of accuracy between the meaning intended and the meaning interpreted. Albert Mehrabian, PhD, a professor at UCLA, has shown that for complex, incongruent communication, 55 percent of meaning is conveyed by physical body language, 38 percent is conveyed through cultural-specific voice tonality interpretation, and only 7 percent is conveyed by words. Keep this in mind during your next conference call to discuss the design nuances of a system that doesn't yet exist.

Alistair Cockburn, one of the Agile Manifesto signatories, created the graph in Figure 6-1. This graph shows the effectiveness of different forms of communication. Notice the difference in effectiveness between paper communication and two people at a whiteboard — with collocation, you get the benefit of better communication.

FIGURE 6-1: Better communication through collocation.

REMEMBER

Face-to-face communication means we are physically face-to-face as we communicate. Although technology today supports bringing geographically dispersed people together more effectively than ever before, technology cannot replicate the social effects of people working in the same physical location on effective and real-time collaboration.

Scrum teams are most effective when they are physically collocated. However, that doesn't mean agile frameworks such as scrum will not work for a dislocated team. In fact, for the success of a dislocated team, a framework such as scrum is

even more important due to its clarity of roles, transparency, and tight empirical feedback loop. You learn more about scrum roles, artifacts, and events in Chapters 7, 10, 11, and 12.

In the following sections, we describe the ideal situation for enabling scrum team communication, realizing that opportunities to achieve the desired benefits of collocation exist even when collocation isn't possible.

Setting up a dedicated area

If the scrum team members are in the same physical place, you want to create as ideal a working environment for them as you can. The first step is to create a dedicated area.

Set up an environment where the scrum team can work in close physical proximity. If possible, the scrum team should have its own room, sometimes called a *team room* or a *scrum room.* The scrum team members create the setup they need in this room, putting whiteboards and bulletin boards on the walls and moving the furniture. By arranging the space for productivity, it becomes part of how they work. If a separate room isn't possible, a *pod* — with workspaces around the edges and a table or collaboration center in the middle — works well.

If you're stuck in cube city and can't tear down walls, be creative and ask for a group of empty cubes and configure them in a way that works for your team, even if that means removing panels. Create a space that you can treat as your team room.

REMEMBER

The right space allows the scrum team to be fully immersed in solving problems and crafting solutions. Visualizing ideas and work-in-progress brings about shared understanding among the entire team. Unfettered access to other team members is also key for effective and real-time collaboration. Create a space that enables these things.

The situation you have may be far from perfect, but it's worth the effort to see how close you can get to the ideal. Before you start an agile transition in your organization, ask management for the resources necessary to create optimal conditions. Resources will vary, but at a minimum, they should include whiteboards, bulletin boards, markers, pushpins, and sticky notes. You'll be surprised at how quickly the efficiency gains more than pay for the investment.

For example, when one client company allocated a dedicated team room and made a $6,000 investment in multiple monitors for developers, they increased productivity and saved almost two months and $60,000 over the life of development. That's a pretty good return on a simple investment. We show you how to quantify these savings in Chapter 15.

Removing distractions

The development team needs to focus, focus, focus. Agile methods are designed to create structure for highly productive work carried out in a specific way. The biggest threat to this productivity is distraction, such as . . . hold on, let me quickly respond to this text message.

Okay, I'm back. The good news is that a scrum team has someone dedicated to deflecting or eliminating distractions: the scrum master. Whether you're going to be taking on a scrum master role or some other role, you need to understand what sorts of distractions can throw the development team off course and how to handle them. Table 6-1 is a list of common distractions and do's and don'ts for dealing with distractions.

REMEMBER

Distractions sap the development team's focus, energy, and performance. The scrum master needs strength and courage to manage and deflect interruptions. Every distraction averted is a step toward success.

TABLE 6-1 Common Distractions

Distraction	Do	Don't
Multiple objectives	Do make sure that the development team is dedicated 100 percent to a single product objective at a time.	Don't fragment the development team between multiple objectives, operations support, and special duties.
Multitasking	Do keep the development team focused on a single task, ideally developing one piece of functionality at a time. A task board can help keep track of the tasks in progress and quickly identify whether someone is working on multiple tasks at once.	Don't let the development team switch between requirements. Switching tasks creates a huge overhead (a minimum of 30 percent) in terms of lost productivity.
Over-supervising	Do let development team members decide among themselves how to accomplish the work to be done after you collaborate on iteration goals; they can organize themselves. Watch their productivity skyrocket.	Don't interfere with the development team or allow others to do so. The sprint review meeting provides ample opportunity to assess progress.
Outside influences	Do redirect any distracters. If a new idea outside the sprint goal surfaces, challenge the product owner to add the item to the product backlog rather than put the accomplishment of the sprint goal at risk.	Don't mess with the development team members and their work. They're pursuing the sprint goal, which is the top priority during an active sprint. Assigning even a seemingly quick task can throw off work for the entire day.
Management	Do shield the development team from direct requests from management (unless management wants to give team members a bonus for their excellent performance).	Don't allow management to negatively affect the productivity of the development team. Make interrupting the development team the path of greatest resistance.

Low-Tech Communicating

When a scrum team is collocated, the members can communicate in person with ease and fluidity. Particularly when you begin your agile transition, you want to keep the communication tools low-tech. Rely on face-to-face conversations and good old-fashioned pen and paper. Low-tech promotes informality, allowing scrum team members to feel that they can change work processes and be innovative as they learn about the product.

The primary tool for communication should be face-to-face conversation. Tackling problems in person is the best way to accelerate production:

- **Have short daily scrum meetings in person.** Scrum teams stand throughout a meeting to discourage it from running longer than 15 minutes. They physically gather around the team's task board.
- **Ask the product owner questions.** Also, make sure the product owner is involved in discussions about product features so he or she can provide clarity when necessary. The conversation shouldn't end when planning ends. In other words, make sure the product owner is accessible at any time and as close by as other team members. With movable desks and chairs, the scrum team, which includes the product owner, can reconfigure their space for better access to each other. Being mobile enables freer collaboration and more freedom overall.
- **Speak with your team members.** If you have questions about features, progress, or integrating, talk, don't email back and forth with team members. The entire development team is responsible for creating the product, and team members need to talk throughout the day.

As long as the scrum team is in close proximity, you can use physical and visual approaches to keep everyone on the same page. The tools should enable everyone to see

- The goal of the sprint
- The functionality necessary to achieve the sprint goal
- What has been accomplished in the sprint
- What's coming next in the sprint
- Who is working on which task
- A clear definition of what it means to be shippable
- What remains to be done

Only a few tools are needed to support this low-tech communication:

- A whiteboard or two (ideally, mobile — on wheels or lightweight). Nothing beats a whiteboard for collaboration. The scrum team can use one for brainstorming solutions or sharing ideas.
- A huge supply of sticky notes in different colors (including poster-sized ones for communicating critical information you want readily visible — such as architecture, coding standards, and the team's definition of done). See Chapter 11 to learn more about making team artifacts more transparent.

  Our personal favorite is giving each developer at least one tabletop dry erase/sticky note easel pad combination, with a lightweight easel. These low-cost tools are fantastic at facilitating communication.
- Lots of colorful pens.
- A sprint-specific task or kanban board (described in Chapters 5 and 11) for tracking progress tactilely.

TIP

If you decide to have a sprint-specific kanban board, use sticky notes to represent *units of work* (features broken down into tasks). For your work plan, you can place sticky notes on a large surface (a wall or your second whiteboard), or you can use a kanban board with cards. You can customize a kanban board in many ways, such as using different-colored sticky notes for different types of tasks, red flag stickers for features that have an impediment, and team member stickers to easily see who is working on which task.

TECHNICAL STUFF

An *information radiator* is a tool that physically displays information to the scrum team and anyone else in the scrum team's work area. Information radiators include kanban boards, whiteboards, bulletin boards, *burndown charts,* which show the iteration's status, and any other sign with details about the product development or the scrum team.

Basically, you move sticky notes or cards around the board to show the status (see Figure 6-2). Everyone knows how to read the board and how to act on what it shows. In Chapter 11, you find out the details of what to put on the boards.

TIP

Whatever tools you use, avoid spending time making things look perfectly neat and pretty. Formality in layout and presentation (what you might call *pageantry*) can give an impression that the work is tidy and elegant. However, the work is what matters, so focus your energy on activities that support the work. Pageantry is the enemy of agility.

FIGURE 6-2: A scrum task board on a wall or whiteboard.

High-Tech Communicating

Although collocation almost universally improves effectiveness, many scrum teams can't be collocated. Some development efforts have teammates scattered across multiple offices; others have off-shore development teams around the world. If you have multiple, geographically scattered scrum teams, try first to reallocate existing talent to form scrum teams collocated within each geographic location. If this move isn't possible, don't give up on an agile transition. Instead, simulate collocation as much as possible.

When scrum team members work in different places, you have to make a greater effort to set up an environment that creates a sense of connectedness. To span distance and time zones, you need more sophisticated communication mechanisms. Although high-tech tools are available to simulate a collocated environment, everyone's contributions are required to be effective. If one team member is remote, everyone on the team may need to be on camera to help the remote team member participate and be engaged.

DON'T REINVENT THE WHEEL!

In the past, manufacturing processes often involved partially completed items being shipped to another location for completion. In these situations, the kanban board on a factory wall in the first location needed to be seen by shop floor management at the second location. Electronic kanban board software was developed to resolve this problem, but the software display looked like a literal kanban board on the wall and was used in the same way. If the software tool had required filtering or scrolling to see all information on the kanban board, it would have been a significant downgrade for the teams, because what made their physical kanban board so effective was being able to see the board updated real-time in its entirety.

When determining which types of high-tech communication tools to support, don't choose tools that degrade direct and real-time conversation or introduce unnecessary complexity:

- **Videoconferencing:** Videoconferencing tools, such Zoom, Teams, and Hangouts, can create a sense of being together. If you have to communicate remotely, at the very least make sure you can see and hear each other clearly. Body language provides the majority of the message.

  REMEMBER

  Decisions aren't made during meetings — they're communicated in meetings. Decisions are made in less formal settings, such as during a hallway conversation, over lunch, at the water cooler, or in an impromptu discussion in someone's office. Unless you're face-to-face, replicating these types of interactions is difficult. The closest you might get is using something like a telepresence robot (a remote-controlled stand on wheels with a tablet on top that displays the remote person's face). The robot can move around an office space so that the single remote person can have hallway and water cooler conversations, almost as if he or she were physically there. Probably not a realistic option, but it shows that you can't beat face-to-face interactions.

- **Persistent chat:** Although instant messaging doesn't convey nonverbal communication, it is real time, accessible, and easy to use. Current examples include Slack, HipChat, and Teams. Several people can also share a session and share files. Persistent chat can be useful for conveying information but not for resolving issues. Have face-to-face conversations for resolving issues.

- **Web-based desktop sharing:** Especially for the development team, sharing your desktop allows you to visually highlight issues and updates in real time. Seeing the problem is always better than just talking about it hypothetically over the phone. Most videoconferencing tools include this capability.

» **Collaboration tools:** These tools allow you to do everything from sharing simple documentation so that everyone has the latest information to using a virtual whiteboard for brainstorming. Some examples include tools such as Google Drive, Miro, Mural, and Jamboard.

Technology is constantly evolving. We can't wait to see what tools are available by the time we release the next edition of this book.

TIP

Using online collaboration tools (such as those just listed) allows you to get out of the status-reporting business and focus your efforts on doing the real work of creating value for your customers. Use these tools to make the artifacts you're already using (such as your sprint backlog) available for stakeholders to peruse on demand. When managers request status updates, you can simply direct them to the collaboration site to pull the information they need. By updating these documents daily, you provide managers with better information than they would have with formalized status reporting procedures under a traditional project management cycle. Avoid creating separate status reports for management; these reports duplicate information in the sprint burndowns and don't support production.

WARNING

When you have a collaboration site with shared documentation, don't assume that everyone automatically understands everything in the documentation. Use a collaboration site to make sure everything is published, accessible, and transparent, but don't let it give your team a false sense of shared understanding that comes from conversations.

Choosing Tools

As noted throughout the chapter, low-tech tools are best suited for agile development, especially initially, while the scrum team becomes accustomed to a more agile way of working and collaborating. Our advice on choosing a tool is less about which specific tool is best and more about whether the tool enables or impedes delivering value to your customer.

We like to start our teams off with low-fidelity tools such as whiteboards, flipcharts, sticky notes, and markers. Later, if they feel they need to invest in more sophisticated technology, we encourage them to find a tool that supports the way they've found works best for them, rather than find a new tool that dictates how they do their work.

In this section, we describe a few points to consider when choosing agile tools: the purpose of the tool and organizational and compatibility constraints.

The purpose of the tool

When choosing tools, the primary question you need to ask is, "What is the purpose of the tool?" Tools should solve a specific problem and support agile processes, the focus of which is pushing forward with the work.

Above all, don't choose anything more complicated than you need. Some tools are sophisticated and take time to learn before you can use them to be productive. If you're working with a collocated scrum team, the training and adoption of agile practices can be enough of a challenge without adding a suite of complicated tools to the mix. If you're working with a dislocated scrum team, introducing new tools can be even more difficult.

Good litmus tests for tools include:

- As a development team member, can I update my task status in a minute or less per day?
- Does the tool help us do our work, or has administering it become our work?
- Is the tool impeding or improving transparency?
- Does it promote or hinder important face-to-face conversations?
- Does the cost of the tool's administration justify the need?
- Does the tool enable leadership interaction consistent with agile values and principles?

WARNING

You can find a lot of agile-centric websites, software, and other tools on the market. Many are useful, but you shouldn't invest in expensive agile tools in your early days of implementing agile. This investment is unnecessary and adds another level of complexity to adoption. As you go through the first few iterations and modify your approach, the scrum team will start identifying procedures that can be improved or need to change. One of these improvements might be the need for additional tools or replacement tools. When a need emerges naturally, from the scrum team, finding organizational support for purchasing the necessary tools is often easier because the need can be tied to a product issue.

Tools that encourage the success of forced team dislocation

During the COVID-19 pandemic of 2020, governments worldwide issued stay-at-home orders that forced a significant majority of people throughout the world to work remotely — dislocated — for months. Countless organizations and industries pivoted product delivery, collaboration, and even business models. Disruption spanned almost every industry.

Our business, Platinum Edge, was no exception. Before the pandemic, the scrum certification programs we delivered, such as Certified ScrumMaster (CSM) and Certified Scrum Product Owner (CSPO), were required by Scrum Alliance to be taught only in-person, physically in a classroom setting. This requirement was temporarily lifted, enabling us to deliver much needed training and experience to people live and online. The experience was powered by digital tools that brought students and our Certified Scrum Trainers together for a valuable learning experience.

We quickly adapted to this challenge and found ways to enable effective collaboration and learning in a virtual environment. We used videoconferencing with breakout rooms for small-group collaboration. Figure 6-3 shows a student's computer screen of the virtual classroom setting.

FIGURE 6-3: Virtual classroom setting.

We also shared visual content (prepared slides as well as real-time drawing) to explain concepts, and enabled small groups to practice and apply concepts on a virtual working canvas that simulated many of the tools mentioned in this chapter (see Figure 6-4). Virtual flipcharts, whiteboards, and sticky notes provided a near-tactile experience of moving, removing, and creating items to share.

The experience wasn't the same as being physically face-to-face, and given the choice, we and our students would choose in-person interactivity. But it was sufficient for effective learning, "better than expected" to quote quite a few students.

FIGURE 6-4: Virtual collaboration board.

Using these tools, the students practiced all the same activities and experienced working as a scrum team just as they did in the real world. As a side benefit, the students were able to retain the virtual board after the class for their continued collaboration.

Although collocated teams build better products faster, many students were able to successfully navigate their forced virtual working conditions by using high-tech tools. Other organizations who invested in tools to help their employees work virtually continued operations even during a worldwide pandemic.

Organizational and compatibility constraints

The tools you choose must operate in your organization. Unless you're using solely non-electronic tools, you'll likely have to take into account organizational policies with respect to hardware, software, and services as well as cloud computing, security, and telephony systems.

The key to creating an agile environment for scrum teams is to do so at the strategic organizational level. Scrum teams drive agile products, so enlist your organization's leadership early to provide tools that will empower your teams to succeed.

IN THIS CHAPTER

- Clarifying agile roles
- Embracing agile values in your organization
- Transforming your team's philosophy
- Sharpening important skills

Chapter 7
Agile Behaviors in Action

In this chapter, you look at the behavioral dynamics that need to shift for your organization to benefit from the performance advantages that agile techniques enable. You find out about the different roles on a product development team and see how you can change a team's values and philosophy about product development. Finally, we discuss some ways for a team to hone key skills for agile success.

Establishing Agile Roles

In Chapter 5, we describe scrum, the most popular agile framework in use today. The scrum framework defines common agile roles in an especially succinct manner. We use scrum terms to describe agile roles throughout this book. These roles are

- Product owner
- Development team member
- Scrum master

The product owner, development team, and scrum master together make up the *scrum team.* Each role is a peer to the others — no one is the boss of anyone else on the team.

The following roles are not part of the scrum framework but are still critically important to agile product development:

- Stakeholders
- Agile mentor

The scrum team together with the stakeholders make up the *product team.* At the center of it all is the *development team*. The product owner and scrum master fulfill roles that ensure the development team's success. Figure 7-1 shows how these roles and teams fit together. This section discusses these roles in detail.

FIGURE 7-1: The product team, scrum team, and development team.

Product owner

The product owner, sometimes called the *customer representative* in non-scrum environments, is responsible for bridging the gaps between the customer, business stakeholders, and the development team. The product owner is an expert on the product and the customer's needs and priorities. The product owner, who is a peer member of the scrum team, shields the development team from business distractions (competing priorities), works with the development team daily to help clarify requirements, and accepts completed work throughout the sprint in preparation for the sprint review.

Product owners make the decisions about what the product does and does not include. Add to that the responsibility of deciding what to release to the market and when to do it, and you see that you need a smart and savvy person to fill this role.

With agile product development, the product owner will

- Develop strategy and direction for the product and set long- and short-term goals.
- Maximize the value of the product resulting from the work of the development team.
- Provide or have access to product expertise.
- Understand and facilitate discussions about the customer's and other business stakeholders' needs with the development team.
- Gather, prioritize, and manage product requirements.
- Take responsibility for the product's budget and profitability.
- Decide when to release completed functionality.
- Work with the development team on a daily basis to answer questions and make decisions.
- Accept or reject completed work — as it's completed — during the sprint.
- Present the scrum team's accomplishments at the end of each sprint, before the development team demonstrates these accomplishments.

What makes a good product owner? Decisiveness. Good product owners understand the customer thoroughly and are empowered by the organization to make difficult business decisions every day. Although able to gather requirements from stakeholders, product owners are knowledgeable about the product in their own right. They can prioritize with confidence.

REMEMBER

Good product owners interact well with the business stakeholder community, the development team, and the scrum master. They are pragmatic and able to make trade-offs based on reality. They are accessible to the development team and also ask for what they need. They are patient, especially with questions from the development team.

Figure 7-2 shows how a product owner works with stakeholders, customers or users, and their scrum team. These relationships are critical for communicating across the organization and for receiving product feedback.

Table 7-1 outlines the responsibilities and matching characteristics of a product owner.

FIGURE 7-2: Product owner communication cycle.

TABLE 7-1 **Characteristics of a Good Product Owner**

Responsibility	A Good Product Owner . . .
Supplies product strategy and direction	Envisions the completed product Firmly understands company strategy
Provides product expertise	Has worked with similar products in the past Understands the needs of the people who will use the product
Understands customer and other stakeholder needs	Understands relevant business processes Creates a solid customer input and feedback channel Works well with business stakeholders
Manages and prioritizes product requirements	Is decisive Focuses on effectiveness Remains flexible Turns stakeholder feedback into valuable, customer-focused functionality Is practical about prioritizing financially valuable features, high-risk features, and strategic system improvements Shields the development team from business distractions (competing stakeholder requests) and has the courage to say "no"

Responsibility	A Good Product Owner . . .
Is responsible for budget and profitability	Understands which product features can deliver the best return on investment Manages budgets effectively
Decides on release dates	Understands business needs regarding timelines
Works with development team	Is accessible for daily clarification of requirements Works with the development team to understand capabilities and technical risks Collaborates well with developers Adeptly describes product features
Accepts or rejects work	Understands acceptance criteria of requirements and ensures that completed functionality work correctly
Presents completed work at the end of each sprint	Clearly introduces the accomplishments of the sprint before the development team demonstrates the sprint's working functionality

The product owner takes on a great deal of business-related responsibility during development. Although the sponsor funds and owns the budget, the product owner manages how the budget is spent.

Product discovery

Product discovery is a key responsibility of product owners, who are often found in the field performing customer interviews, striving to better understand challenges and problems. They meet with stakeholders and host workshops to gather ideas for improving ROI and product value. They discover what the product needs to become.

WARNING

Product discovery is not a relabeling of a waterfall's big, up-front planning phase. Product discovery is ongoing, through continuous inspection and adaptation of the customer's needs.

Gathering data is another activity performed by product owners. They watch product usage trends, looking for problematic areas. They dive into customer service issues to understand patterns and opportunities for improving the customer's experience. With the help of talented experts in user interface (UI), user experience (UX), engineering, and other subject matter, they continually look for opportunities to improve product design. See Chapter 11 to learn other product discovery activities performed by a product owner.

Product ownership is a full-time, fully dedicated role because they not only perform product discovery but also enable product development.

Product development

As the product is developed, the product owner is key for helping the team achieve sprint and release goals. Throughout development, many questions will arise needing clarification. The product owner attempts to answer these questions.

They do the necessary preparations before the release and sprint planning, product backlog refinement activities, and sprint reviews and retrospectives (discussed in Chapters 10 and 12). They watch the team's task board. If a task is taking longer than expected or blocked, they work with the scrum master and development team to find opportunities where they can help. See Chapter 11 to better understand how the product owner works throughout the sprint to support product development.

With a dedicated and decisive product owner, the development team has all the business support they needs to turn requirements into working functionality. The following section explains how the product owner helps ensure that the development team understands the product they will create.

Development team member

Development team members are the people who create the product. In software development, programmers, testers, UI designers, writers, data engineers, UX designers, and anyone else with a hands-on role in product development are development team members. With other types of product, the development team members may have different skills.

With agile product development, the development team is

- **Directly accountable for creating deliverables — the product features and functionality.**
- **Self-organizing and self-managing.** The development team members determine their own tasks and how they want to complete those tasks.
- **Cross-functional.** Collectively, the development team possesses all skills required to elaborate, design, develop, test, integrate, and document requirements into working functionality, including skills needed to automate the deployment of their product increments. High-performing development teams have all the skills needed to create and release product increments to the customers and users.

- **Multi-skilled.** Development teams aren't just cross-functional as a whole; development team members are also versatile — they're not tied to a single skill set. They have existing skills to immediately contribute at the beginning of development, but they are also willing to learn new skills and to teach what they know to other development team members.

  Every developer should be able to do more than one skill. And every skill should be able to be done by more than one person. Specialized skill developers ("I do only one thing") are great until they're not. Scrum teams need access to all skills every day, not just at certain phases in development as in waterfall projects. If the one developer who knows how to run tests calls in sick even one day, the whole team is unable to get to done for that day. You need at least one other person on the team who, in a pinch, can step in to keep the team moving forward.

- **Ideally dedicated to one product objective for the duration of development.**

- **Ideally collocated.** The team (including the product owner and scrum master) should be working together in the same area of the same office, preferably in a team room.

REMEMBER

What makes a good development team member? Versatility. You want developers who are intellectually curious — who look for ways to contribute to the sprint goal a little more today than the day before. Some development team members aren't fully versatile at first. Ideally, during the team's work, developers with one skill take the opportunity to gain exposure across the stack of skills as they shadow and pair with other developers, becoming more T-, Pi-, or M-shaped in their skillset, as shown in Figure 7-3.

FIGURE 7-3: Development team member skill development.

High-performing scrum teams have developers whose skill set is Pi (π) or M shaped. In other words, in addition to their primary skill and their broad exposure to all the skills needed by the team, they are proficient in one more skill

(Pi-shaped) or two more skills (M-shaped). Development teams with Pi- and M-shaped skilled team members typically have a higher velocity because they eliminate single points of failure.

Take a look at the team responsibilities and matching characteristics in Table 7-2.

TABLE 7-2 **Characteristics of a Good Development Team Member**

Responsibility	A Good Development Team Member . . .
Creates the product	Enjoys creating products Is skilled in more than one of the jobs necessary to create the product
Is self-organizing and self-managing	Exudes initiative and independence Understands how to work through impediments to achieve goals Coordinates the work to be done with the rest of the team
Is cross-functional	Has curiosity Willingly contributes to areas outside his or her mastery Enjoys learning new skills Enthusiastically shares knowledge
Is dedicated and collocated	Is part of an organization that understands the gains in efficiency and effectiveness associated with focused, collocated teams

The two other members of the scrum team, the product owner and the scrum master, help support the development team's efforts in creating the product. Whereas the product owner ensures that the development team is effective (working on the right things), the scrum master helps clear the way for the development team to work as efficiently as possible.

Scrum master

A scrum master, sometimes called a *facilitator* or *team coach* in non-scrum environments, is responsible for supporting the development team, clearing organizational roadblocks, and keeping processes true to agile principles.

A scrum master is different from a project manager. Teams using traditional project approaches work *for* a project manager. A scrum master, on the other hand, is a servant-leader peer who supports the team so that it is fully functional and productive. The scrum master role is an enabling role, rather than an accountability role. You can find more about servant leadership in Chapter 16.

With agile product development, the scrum master will

- Act as a process coach and agile champion, helping the team and the organization follow scrum values and practices.
- Help remove impediments — both reactively and proactively — and shield the development team from external interferences.
- Work with the product owner to foster close cooperation between stakeholders and the development team.
- Facilitate consensus building within the scrum team.
- Protect the scrum team from organizational distractions.

TIP

We compare the scrum master to the aeronautical engineer whose job is to reduce drag on the aircraft. Drag is always there but can be reduced through innovative and proactive engineering. Likewise, all teams have organizational impediments creating drag on the team's efficiency, and there is always another constraint that can be identified and removed. One of the most significant parts of a scrum master's role is challenging the status quo to remove roadblocks and prevent distractions to the development team's work. A scrum master who is good at these tasks is priceless to product development and to the organization. If a development team has seven people, the effect of a good scrum master is times seven.

What makes a good scrum master? A scrum master doesn't need project manager experience. A scrum master is an expert in agile processes and can coach others. He or she knows the right questions to ask to guide the team to higher performance through introspection and retrospection. The scrum master also works collaboratively with the product owner and the stakeholder community.

TIP

Facilitation skills cut through the noise of group gatherings and ensure that everyone on the scrum team is focused on the right priority at the right time.

Scrum masters have strong communication skills, with enough organizational clout to secure the conditions for success by negotiating for the right environment, protecting the team from distractions, and removing impediments. Scrum masters are great facilitators and great listeners. They can negotiate their way through conflicting opinions and help the team help itself. Review the scrum master's responsibilities and matching characteristics in Table 7-3.

TIP

Clout is not the same as authority. Organizations need to empower their scrum masters so they can influence change in the team and organization without formal authority over others. Clout involves earned respect, often through success and experience. Some types of clout that empower scrum masters come about through expertise (usually a niche knowledge), longevity ("I've been at the

company a long time and know its history first hand"), charisma ("people generally like me"), or associations ("I know important people"). Don't underestimate the value of a scrum master with organizational clout.

TABLE 7-3 Characteristics of a Good Scrum Master

Responsibility	A Good Scrum Master . . .
Upholds scrum values and practices	Is an expert on scrum processes Is passionate about agile techniques
Removes roadblocks and prevents disruptions	Has organizational clout and can resolve problems quickly Is articulate, diplomatic, and professional Is a good communicator and a good listener Is firm about the development team's need to focus only on the product objectives and the current sprint
Fosters close cooperation between external stakeholders and the scrum team	Looks at the needs of the product team as a whole Avoids cliques and helps break down group silos
Facilitates consensus building	Understands techniques to help groups reach agreements
Is a servant-leader	Does not need or want to be in charge or be the boss Ensures that all members of the development team have the information they need to do the job, use their tools, and track progress Truly desires to help the scrum team

The members of the scrum team — the product owner, development team, and scrum master — work together every day.

As we mention earlier in the chapter, the scrum team plus stakeholders make up the product team. Sometimes stakeholders have less active participation than scrum team members but still can have considerable effect and provide a great deal of value to a product.

Stakeholders

Stakeholders are anyone with an interest in the product. They are not ultimately responsible for executing the product, but they provide input and are affected by the product's outcome. The group of stakeholders is diverse and can include people from different departments or even different companies.

GAINING CONSENSUS: THE FIST OF FIVE

Part of working as a team means agreeing on decisions as a team. An important part of being a scrum master is helping the team build consensus. We've all worked with groups where it was difficult to arrive at consensus, from how long a task would take to where to go for lunch. A quick, casual way to find out whether a group agrees with an idea is to use the *fist of five.*

On the count of three, each person holds up a number of fingers, reflecting their degree of comfort with a proposed path forward as a way to deal with the issue in question:

- 5: I love the idea.
- 4: I think it's a good idea.
- 3: I can support the idea.
- 2: I have reservations, so let's discuss.
- 1: I am opposed to the idea.

If some people have three, four, or five fingers up, and some have only one or two, discuss the idea. Find out why the people who support the idea think it will work, and what reservations the people who oppose the idea have. After all group members show at least three fingers — they don't need to love the idea, but they can support it — you have consensus and can move forward. The scrum master's consensus-building skills are essential for this task.

You can also quickly get an idea of consensus on a decision by asking for a simple thumb up (support), thumb down (don't support), or thumb to the side (either way is fine). Some people refer to this as a Roman vote. It's quicker than a fist of five, and is great for answering yes-or-no questions.

With agile product development, stakeholders

- Include the customer
- May include technical people whose subject-matter expertise can support the development team to do its work
- May include the legal department, account managers, salespeople, marketing experts, and customer service representatives who can affect the product
- May include product or market subject matter experts besides the product owner

Stakeholders may help provide key insights about the product and its use. Stakeholders might work closely with the product owner during the sprint, and will give feedback about the product during the sprint review at the end of each sprint.

Stakeholders and the part they play vary among products and organizations. Almost all product teams have stakeholders outside the scrum team.

Some product teams also have agile mentors, especially teams that are new to agile processes.

Agile mentor

A mentor is a great idea for any area in which you want to develop new expertise. The *agile mentor,* sometimes called an *agile coach,* is someone who has experience implementing agile principles, practices, and techniques, and can share that experience with a team. The agile mentor can provide valuable feedback and advice to new teams and to teams wanting to perform at a higher level.

With agile product development, the agile mentor

- Serves in a mentoring role only and is not part of the scrum team
- Is often a person from outside the organization, and can provide objective guidance, without personal or political considerations
- Is an agile expert with significant diversity of experience in implementing agile techniques for products of varying contexts

You may want to think of an agile mentor the way you think of a golf coach. Most people use a golf coach not because they don't know how to play the game of golf but because a golf coach objectively observes things that a player engaged in the game never notices. Golf, like implementing agile techniques, is an exercise where small nuances make a world of difference in performance.

Establishing New Values

Lots of organizations post their core values on the wall. In this section, however, we are talking about values that represent a way of working together every day, supporting each other, and doing whatever it takes to achieve the scrum team's commitments.

In addition to the values from the Agile Manifesto, the five core values for scrum are

- Commitment
- Courage
- Focus
- Openness
- Respect

The following sections provide details about each of these values.

Commitment

Commitment implies engagement and involvement. With agile product development, the scrum team pledges to achieve specific goals. Confident that the scrum team will deliver what it promises, the organization mobilizes around the pledge to meet each goal.

Agile processes, including the idea of self-organization, provide people with all the authority they need to meet commitments. There is no need for managers to hold scrum teams accountable for every specific task they identify could be done to achieve their business goals. Tasks may change, but business goals are what matter and drive outcomes. Scrum teams hold each other accountable because they are purpose-driven or outcome-driven. With strategic stability, they maintain tactical flexibility.

Commitment requires a conscious effort. Consider the following points:

- Scrum teams must be realistic when making commitments, especially for short sprints. Aim high, but not unrealistically high.
- Scrum teams must fully commit to goals. This includes having consensus among the team that the goal is achievable. After the scrum team agrees on a goal, the team does whatever it takes to reach that goal.
- The scrum team is pragmatic but ensures that every sprint has a tangible value. Achieving a sprint goal and completing every item in the goal's scope are different. For example, a sprint goal of proving that a product can perform a specific action is much better than a goal stating that exactly seven requirements will be complete during the sprint. Effective scrum teams focus on the goal and remain flexible in the specifics of how to reach that goal.

- Scrum teams are willing to be accountable for results. The scrum team has the power to be in charge of the product. As a scrum team member, you can be responsible for how you organize your day, the day-to-day work, and the outcome.

Consistently meeting commitments is central to using agile approaches for long-term planning. In Chapter 15, you read about how to use performance to accurately determine schedules and budgets.

Focus

Working life is full of distractions. Plenty of people in your organization would love to use your time to make their day easier. Disruptions, however, are costly. Jonathan Spira, from the consulting firm Basex, published a report called "The Cost of Not Paying Attention: How Interruptions Impact Knowledge Worker Productivity." His report details how businesses in the United States lose close to $600 *billion* a year through workplace distractions.

Scrum team members can help change those dysfunctions by insisting on an environment that allows them to focus. To reduce distractions and increase productivity, scrum team members can

- **Physically separate themselves from company distracters.** One of our favorite techniques for ensuring high productivity is to find an annex away from the company's core offices and have that be the scrum team's work area. Sometimes the best defense is distance.
- **Ensure that you're not spending time on activities unrelated to the sprint goal.** If someone tries to distract you from the sprint goal with something that "has to be done," explain your priorities. Ask, "How will this request move the sprint goal forward?" This simple question can push a lot of activities off the to-do list.
- **Figure out what needs to be done and do only that.** The development team determines the tasks necessary to achieve the sprint goal. If you're a development team member, use this ownership to drive your focus to the priority tasks at hand.
- **Balance focused time with accessibility to the rest of the scrum team.** Francesco Cirillo's Pomodoro technique — splitting work into 25-minute time blocks, with breaks in between — helps achieve balance between focus and accessibility. We often recommend giving development team members noise-canceling headsets, the wearing of which is a "do not disturb" sign.

However, we also suggest a team agreement that all scrum team members have a minimum set of "office hours" in which they are available for collaboration.

- **Check that you're maintaining your focus.** If you're unsure of whether you are maintaining focus — it can be hard to tell — go back to the basic question, "Are my actions consistent with achieving the overall goal and the near-term goal (such as completing the current task)?"

As you can see, task focus is not a small priority. Extend the effort up front to create a distraction-free environment that helps your team succeed.

Openness

Secrets have no place on a scrum team. If the team is responsible for the result of the product, it only makes sense that the team has all the facts at its disposal. Information is power, and ensuring that everyone — both the scrum team and stakeholders — have access to the information necessary to make the right decisions requires a willingness to be transparent. They agree to be transparent with not only the progress of work but also the challenges with performing the work. To leverage the power of openness, you can

- **Ensure that everyone on the team has access to the same information.** Everything from the product vision down to the smallest detail about the status of tasks needs to be in the public domain as far as the product team is concerned. Use a centralized repository as the single source for information, and then avoid the distraction of "status reporting" by putting all status (burndowns, impediment list, and so forth) and information in this one place. We often send a link to this repository to the stakeholders and say, "All the information we have is a click away. There is no faster way to get updated."
- **Be open and encourage openness in others.** Team members must feel free to speak openly about problems and opportunities to improve, whether the issues are something that they're dealing with themselves or see elsewhere in the team. Openness requires trust within the team, and trust takes time to develop.
- **Defuse internal politics by discouraging gossip.** If someone starts talking to you about what another team member did or didn't do, ask him or her to take the issue to the person who can resolve it. Don't gossip yourself. Ever.
- **Always be respectful.** Openness is never an excuse to be destructive or mean. Respect is critical to an open team environment.

Small problems unaddressed often grow to become crises. Use an open environment to benefit from the input of the entire team and ensure that your development efforts are focused on the product's true priorities.

Respect

Each individual on the team has something important to contribute. Your background, education, and experiences have a distinctive influence on the team. Share your uniqueness and look for, and appreciate, the same in others. Scrum team members respect each other as capable, independent people. You encourage respect when you

- **Foster openness.** Respect and openness go hand in hand. Openness without respect causes resentment; openness with respect generates trust.
- **Encourage a positive work environment.** Happy people tend to treat one another better. Encourage positivity, and respect will follow.
- **Seek out differences.** Don't just tolerate differences; try to find them. The best solutions come from diverse opinions that have been considered and appropriately challenged.
- **Treat everyone on the team with the same degree of respect.** All team members should be accorded the same respect, regardless of their role, level of experience, or immediate contribution. Encourage everyone to give his or her best.

REMEMBER

Respect is the safety net that allows innovation to thrive. When people feel comfortable raising a wider range of ideas, the final solution can improve in ways that would never be considered without a respectful team environment. Use respect to your team's advantage.

Courage

The last scrum value is courage. We list it last because it requires courage to live the other four scrum values. In other words, it takes courage to commit to goals. It takes courage to focus. It takes courage to be open, and it takes courage to show and expect respect. Bottom line, it takes courage to do scrum — not just at first, but always.

Embracing agile techniques is a change for many organizations. Successfully making changes requires courage in the face of resistance. It takes courage to do the right thing and to work on tough problems. Following are some tips that foster courage:

- **Realize that the processes that worked in the past won't necessarily work now.** Sometimes you need to remind people of this fact. If you want to be successful with agile techniques, your everyday work processes need to change.
- **Be ready to buck the status quo.** The status quo will push back. Some people have vested interests and will not want to change how they work.
- **Temper challenge with respect.** Senior members of the organization might be especially resistant to change because they often created the old rules. You're challenging those rules. Respectfully remind them that you can achieve the benefits of agile techniques only by following the 12 Agile Principles faithfully. Ask them to give change a try.
- **Embrace the scrum values.** Have the courage to make commitments and stand behind those commitments. Have the courage to focus and tell distracters "no." Have the courage to be open and to acknowledge that there is always an opportunity to improve. And have the courage to be respectful and tolerant of other people's views, even when they challenge your views.

As you replace your organization's antiquated processes with more modern approaches, expect to be challenged. Take on that challenge; the rewards can be worth it in the end.

Changing Team Philosophy

An agile development team operates differently from a team using a waterfall approach. Development team members must change their roles based on each day's priorities, organize themselves, and think about product development in a whole new way to achieve their commitments.

To be successful scrum teams should embrace the following attributes:

- **Dedicated team:** Each scrum team member works only on the product objectives decided by the scrum team, and not on other teams or products. Product development may finish on one product, and then new development may start on a new product, but the team stays the same long-term.
- **Cross-functionality:** The willingness and ability to work on different types of tasks to create the product.
- **Self-organization:** The ability and responsibility to determine how to go about the work of product development.

- **Self-management:** The ability and responsibility to keep work on track.
- **Size-limited teams:** Right-size development teams to ensure effective communication. Smaller is better; the development team should never be larger than nine people.

  We've found that development teams of four to six people are the right size. The lines of communication to self-organize around the work are not too complex, adequate skill coverage exists, and the cost is manageable.
- **Ownership:** Take initiative for work and responsibility for results. Have pride in your craftsmanship.

TIP

The following sections look at each of these ideas in more detail.

Dedicated team

A traditional approach to resource allocation (we prefer the term *talent allocation* — people are not inanimate objects) is to allocate portions of team members' time across multiple teams and projects to get to full 100 percent utilization to justify the expense of employing team members. For management, knowing that all hours of the week are accounted for and justified is gratifying. However, the result is lower productivity due to continual *context switching* — the cost associated with cognitive demobilization and remobilization to switch from one task to another. Work in progress has no value; completed work does.

Other common talent allocation practices include moving a team member from team to team to temporarily fill a skill gap or a manpower gap, and tasking a team with multiple projects at once. These tactics are often employed to try to do more with less, but all the input variances make it nearly impossible to predict outputs.

These approaches have similar results: a significant decrease in productivity and an inability to extrapolate performance. Studies clearly show a minimum of 30 percent increase in the time required to complete product objectives run in parallel instead of serially.

TECHNICAL STUFF

Thrashing is another term for context switching between tasks. Avoid thrashing by dedicating team members to a single product objective at a time.

The following results occur when you dedicate scrum teams to work on only one objective at a time:

- **More accurate release projections:** Because the same people are consistently doing the same tasks every sprint with the same amount of time allocated from sprint to sprint, scrum teams can accurately and empirically

extrapolate how long it will take to complete their remaining backlog items with more certainty than traditional splintered approaches.

- **Effective, short iterations:** Sprints are short because the shorter the feedback loop, the more quickly scrum teams can respond to feedback and changing needs. There just isn't enough time for thrashing team members between competing priorities.
- **Fewer and less costly defects:** Context switching results in more defects because distracted developers produce lower quality functionality. It costs less to fix something while it is still fresh in your mind (during the sprint) than later, when you have to try to remember the context of what you were working on. Studies show that defects cost 6.5 times more to fix after the sprint ends and you've moved on to other requirements, 24 times more to fix when preparing for release, and 100 times more to fix after the product is in production.

TIP

If you want more predictability, higher productivity, and fewer defects, dedicate your scrum team members. We've found this to be one of the highest factors of agile transition success.

Cross-functionality

On traditional projects, experienced team members are often typecast as having a single skill. For example, a .NET programmer may always do .NET work, and a tester may always do quality control work. Team members with complementary skills are often considered to be part of separate groups, such as the programming group or the testing group.

Agile approaches bring the people who create products together into a cohesive group — the development team. People on agile development teams try to avoid titles and limited roles. Development team members may start out on the team with one skill, but learn to perform many different jobs throughout as they help create the product.

Cross-functional development teams collectively possess all the skills required to take product requirements from idea to delivered value. But cross-functional teams aren't enough. Cross-functional *individuals* make development teams more efficient.

For example, suppose a daily scrum meeting uncovers testing as the highest priority task to complete the requirement that day. A programmer or a designer who also has some skill at testing steps in to help and finish the task more quickly, or possibly in the absence of another development team member who called in sick that day. When the development team is cross-functional, it can *swarm* (a

technique for limiting work in progress) on one product requirement at a time, with as many people working on a single requirement as possible, to quickly complete the feature.

Cross-functionality also helps eliminate single points of failure. Consider traditional projects, where each person knows how to do one job. When a team member gets sick, goes on vacation, or leaves the company, no one else may be capable of doing his or her job. The tasks that person was doing are delayed. By contrast, cross-functional agile development team members are capable of doing many jobs. When one person is unavailable, another can step in.

Cross-functionality encourages each team member to

- **Set aside the narrow label of what he or she can do.** Titles have no place on a scrum team. Skills and an ability to contribute are what matter. Start thinking of yourself as a Special Forces commando — knowledgeable enough in different areas that you can take on any situation.
- **Work to expand skills.** Don't work only in areas you already know. Try to learn something new during each sprint. Techniques such as *mob programming* — where entire teams work together to code one item — can help you learn new skills quickly and increase overall product quality. A more detailed explanation of mob programming is discussed in Chapter 11.
- **Step up to help someone who has run into a roadblock.** Helping someone with a real-world problem is a great way to learn a new skill.
- **Be flexible.** A willingness to be flexible helps to balance workloads and makes the team more likely to reach its sprint goal.

With cross-functionality in place, you avoid waiting for key people to work on tasks. Instead, a motivated, even if somewhat less knowledgeable, development team member can work on the next highest priority piece of functionality today rather than starting on something of lower priority. That development team member learns and improves, and the workflow continues to be balanced. Cross-functional teams enable the next available person to pull the next task needing to be completed, whatever that might be (rather than having someone push work to the person).

One big payback of cross-functionality is that the development team completes work quickly. Post-sprint review afternoons are often celebration time. Go to the movies together, play video games, head to the beach or the bowling alley, or go home early.

Self-organization

Agile techniques emphasize self-organizing development teams to take advantage of development team members' varied knowledge and experience.

REMEMBER

If you've read Chapter 2, you may recall Agile Principle #11: The best architectures, requirements, and designs emerge from self-organizing teams.

Self-organization is an important part of being agile. Why? In a word: ownership. Self-organized teams are not complying with orders from others; they own the solution developed and that makes a huge difference in team member engagement and solution quality.

For development teams used to a traditional command-and-control project management model, self-organization may take some extra effort at first. Agile development teams do not have a project manager to tell them what to do. Instead, self-organizing development teams

- **Commit to their own sprint goals.** At the beginning of each sprint, the development team works with the product owner to identify an objective it can reach, based on priorities.
- **Identify their tasks.** Development team members determine the tasks necessary to meet each sprint goal. The development team works together to figure out who takes on which task, how to get the work done, and how to address risks and issues.
- **Estimate the effort necessary for requirements and related tasks.** The development team knows the most about how much effort it will take to create specific product features.
- **Focus on communication.** Successful agile development teams hone their communication skills by being transparent, communicating face-to-face, being aware of nonverbal communication, participating, and listening.

  TIP

  The key to communication is clarity. With complex topics, avoid one-way, potentially ambiguous modes of communication, such as email. Face-to-face communication prevents misunderstandings and frustration. You can always summarize the conversation in a quick email later if details need to be retained. Learn more about using effective mediums of communication in Chapter 6.
- **Collaborate.** Getting the input of a diverse scrum team almost always improves the product but requires solid collaboration skills. Collaboration is the foundation of an effective scrum team. Development teams take input from stakeholders but own their final solution.

REMEMBER

No successful product is an island. Collaboration skills help scrum team members take risks with ideas and bring innovative solutions to problems. A safe and comfortable environment is a cornerstone of a successful agile development effort.

- **Decide with consensus.** For maximum productivity, the entire development team must be on the same page and committed to the goal at hand. The scrum master often plays an active role in building consensus, but the development team ultimately takes responsibility for reaching agreement on decisions, and everyone owns the decisions.
- **Actively participate.** Self-organization may be challenging for the shy, but all development team members must actively participate. No one is going to tell the development team what to do to create the product. The development team members tell themselves what to do. And when. And how.

TIP

In our agile coaching experience, we've heard new development team members ask questions like, "So, what should I do now?" A good scrum master answers by asking the developer what he or she needs to do to achieve the sprint goal, or by asking the rest of the development team what they suggest. If a requirement isn't completed but there are no new tasks to start, before beginning a new requirement (and increasing the team's work-in-progress), ask, "Is there someone you can help?" or "Is there something you can learn by shadowing?" Answering questions with questions can be a helpful way to guide a development team toward being self-organizing.

Being part of a self-organizing development team takes responsibility, but it also has its rewards. Self-organization gives development teams the freedom to succeed. Self-organization increases ownership, which can result in better products, which can help development team members find more satisfaction in their work and pride in their craftsmanship.

Self-management

Self-management is closely related to self-organization. Agile development teams have a lot of control over how they work; that control comes with the responsibility for ensuring the product is successful. To succeed with self-management, development teams

- **Allow situational leadership to ebb and flow.** With agile product development, each person on the development team has the opportunity to lead. For different tasks, different leaders will naturally emerge; leadership will shift throughout the team based on skill expertise and previous experiences, not title.

- **Rely on agile processes and tools to manage the work.** Agile methods are tailored to make self-management easy. With an agile approach, meetings have clear purposes and time limits, and artifacts expose information but rely on minimal effort to create and maintain. Taking advantage of these processes allows development teams to spend most of their time creating the product.
- **Report progress regularly and transparently.** Each development team member is responsible for accurately updating work status on a daily basis. Luckily, progress reporting is a quick task. In Chapter 11, you find out about burndown charts, which provide status but only require a few minutes each day to update. Keeping status current and truthful makes planning and issue management easier.
- **Manage issues within the development team.** Many obstacles can arise: Development challenges and interpersonal problems are a couple of examples. The development team's first point of escalation for most issues is the development team itself.
- **Create a team agreement.** Development teams sometimes make up a team agreement, a document that outlines the expectations each team member will commit to meet. Working agreements provide a shared understanding of behavioral expectations and empower the facilitator to keep the team on track according to what it has already agreed together.
- **Inspect and adapt.** Figure out what works for your team. Best practices differ from team to team. Some teams work best by coming in early; others work best by coming in late. The development team is responsible for reviewing its own performance and identifying techniques to continue and techniques to change.
- **Actively participate.** As with self-organization, self-management works only when development team members join in and commit to guiding the product's direction.

TIP

The development team is primarily responsible for self-organization and self-management. However, the scrum master can assist the development team in a number of ways. When development team members look for specific directions, the scrum master can remind them that they have the power to decide what to do and how to do it. If someone outside the development team tries to give orders, insist on tasks, or dictate how to create the product, the scrum master can intervene. The scrum master can be a powerful ally in the development team's self-organization and self-management.

Size-limited teams

Agile development teams are intentionally small. A small development team is a nimble team. As the development team size grows, so does the overhead associated with orchestrating task flow and communication flow.

Ideally, agile development teams have the least number of people necessary to be self-encapsulated (can do everything necessary to produce the product) and not have single points of failure. To have skill coverage, teams typically won't be any smaller than three people. Statistically, scrum teams are fastest with six developers, and cheapest with four to five developers. Keeping the development team size between three and nine people helps teams act as cohesive teams, and avoids creating subgroups, or *silos.*

Limiting development team size

- Encourages diverse skills to be developed.
- Facilitates good team communication. (Each additional team member increases team communication channels geometrically — that's not exponentially, but it's close — as shown in Figure 7-4.)
- Maintains the team in a single unit.
- Promotes joint ownership, cross-functionality, and face-to-face communication.

FIGURE 7-4: Team communication complexity is a function of team size.

When you have a small development team, a similarly limited and focused scope follows. Development team members are in close contact throughout the day as tasks, questions, and peer reviews flow back and forth among teammates. This cohesiveness ensures consistent engagement, increases communication, and reduces risk.

When you have a large product and a correspondingly large development team, split the work between multiple scrum teams. For more on scaling — or better yet de-scaling — across the enterprise, see Chapter 19.

Ownership

Being part of a cross-functional, self-organized, self-managing development team requires responsibility and ownership. The top-down management approaches on traditional projects do not always foster the maturity of ownership necessary for taking responsibility for products and results. Even seasoned development team members may need to adjust their behavior to get used to making decisions.

Development teams can adapt behavior and increase their level of ownership by doing the following:

- **Take initiative.** Instead of waiting for someone else to tell you what to work on, take action. Do what is necessary to help meet commitments and goals.
- **Succeed and fail as a team.** With agile product development, accomplishments and failures alike belong to the team. If problems arise, be accountable as a group, rather than finding blame. When you succeed, recognize the group effort necessary for that success.
- **Trust the ability to make good decisions.** Development teams can make mature, responsible, and sound decisions about product development. This takes a degree of trust as team members become accustomed to having more control.

Behavioral maturity and ownership doesn't mean that agile development teams are perfect. Rather, they take ownership for the scope they commit to, and they take responsibility for meeting those commitments. Mistakes happen. If they don't, you aren't pushing yourself outside your comfort zone. A mature development team identifies mistakes honestly, accepts responsibility for mistakes openly, and learns and improves from its mistakes consistently.

IN THIS CHAPTER

- Understanding why permanent teams are needed
- Figuring out what motivates people
- Seeing why and how permanent teams continuously improve knowledge and capability

Chapter 8
The Permanent Team

Describing a product development team as a *permanent team* may seem a bit extreme. How can a team working in today's ever-changing business environment become permanent? Product development teams who work together consistently on long-lived products become more effective with time. Certainly, team adjustments may occur so that team members can pursue career growth or other business opportunities, but each change in the team's composition causes the team to step back and relearn. For this reason, teams should be as long-lived, stable, and enduring as possible for sustained quality product development.

Enabling Long-Lived Product Development Teams

Today's products are required to meet a multitude of both near-term and long-term needs. With traditional project management, the horizon for realizing value from a product development project spans months to a year or even multiple years. Return on product-focused investments, however, can start much earlier and last much longer. It's not uncommon for a product to last and evolve effectively for six or ten or more years. New ideas or needs are continually identified through maintenance and enhancement requests. Stable, enduring, and long-lived

product development teams are best suited for building long-lived, valuable products.

Instead of being a cost center, high-performing teams can become a revenue and cost-saving center. They become valuable organizational assets capable of tackling difficult problems.

At an individual level, long-lived teams become almost like a family. Family-sized. Vulnerable and honest with each other. Accomplishing challenges or hurdles placed in its path. They may even eat or socialize together, and they definitely learn and work together day in and day out. Being part of a great team can be one of the most rewarding experiences of a person's career.

Peter Senge, in his book *The Fifth Discipline: The Art and Practice of the Learning Organization* (Doubleday), wrote: "When you ask people what it is like being part of a great team, what is most striking is the meaningfulness of the experience. People talk about being part of something larger than themselves, of being connected, of being generative. It becomes quite clear that, for many, their experiences as part of truly great teams stand out as singular periods of life lived to the fullest. Some spend the rest of their lives looking for ways to recapture that spirit."

Agile principles and values were defined to help teams work together. Keep in mind the following key principles:

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
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.

Leveraging long-term knowledge and capability

Product development teams use empirical process controls to inspect and adapt and learn. They learn about the customer, users, product, architecture, and sponsors. Every day they learn better techniques for improving their product and working more effectively with each other. Because the teams are ideally collocated, they know their teammates' personal lives, work aspirations, dreams, and goals. With time, they develop a valuable shared memory that they use as a reference for all their work.

WARNING

Handoffs are one of the "Seven Wastes of Software Development" mentioned by Mary and Tom Poppendieck in their book *Implementing Lean Software Development: From Concept to Cash* (Addison-Wesley Professional). Half of a team's knowledge is lost during a hand-off, and the situation compounds — if Bob tells Jim who tells Sue, then 75 percent of the knowledge is lost. Asking someone else to pick up where you left off requires costly context switching, repeated research, and a re-understanding of the product work. Swapping team members into and out of the team and asking them to hand off their work can be costly.

Product development teams become incubators for learning. Capability is developed as the team works together. When team members learn something new that will benefit their team, they are excited and willingly share it. The team creates a safe environment for experimentation, failure, and vulnerability.

Teammates want each other to be successful. Everyone on the team knows that the more skilled they become, the more effective they'll be at creating better products faster. Sprint review after sprint review and sprint retrospective after sprint retrospective uncover new understanding that incrementally move the team forward. (Find out more about sprint reviews and sprint retrospectives in Chapter 12.)

From a product perspective, over time product development teams gain a deep understanding about the product and its architecture. They understand intimately how the product functions, what to test for, areas to avoid, product documentation, and much more. Change is easily implemented because they understand every part of the product that could be affected by the change. Emergent architecture comes to life because the team builds an architecture that is barely sufficient for the current need and evolves when needed.

Staffing product development teams for the long-term maximizes the benefit for the product, the organization, the team, and the individuals on the team.

Navigating Tuckman's phases to performance

When teams learn to work with one another, the process is known as *team development.* Bruce Tuckman, an educational psychologist, outlined the following five phases that most teams follow:

- **Forming:** During this phase, the team orients itself and gets acquainted. In this period of uncertainty, team members seek to understand common expectations, personal fit within the team, and how they'll benefit from interacting with their teammates. Team interactions are social as team members get to know one another.

- **Storming:** This phase can be the most visibly difficult for the team. After team members become acquainted, the realization sets in that they must work with each other in accomplishing a goal. Personality types clash, interpersonal conflicts arise, frustrations increase, and competition heightens. This phase, if not carefully facilitated by a scrum master, can bog down the team and result in long-term problems. Team performance is often at its lowest during storming.
- **Norming:** If the team successfully emerges from storming, a sense of unity emerges. Roles become clearer, interpersonal differences are resolved, and performance begins to improve. If conflict goes unresolved, the team may slide back into storming.
- **Performing:** During this phase, the team begins to hit its stride. Alignment on goals is clear and each person begins to understand how they can best help their teammates. Problems and conflicts may arise, but they are dealt with constructively. The team experiences an openness and transparency with one another that leads to improved performance.
- **Adjourning:** At some point far in the future (we hope), the product or team may reach its conclusion and end. If a new product is started for an existing team, however, the team will experience a bit of adjourning as it transitions from the old product to the new product. The team's journey may navigate again through the previous four stages.

No team escapes navigating these phases, and no team can skip a phase because each is crucial for the team's development. Many teams go through multiple stages more than once. Unfortunately, many teams wander the storming phase longer than they want. Yet the storming phase, like the refiner's fire, makes reaching the performing phase as a team that much healthier. Storming builds team character, alignment, and strength.

TIP

Address the storming phase head on. Each interpersonal conflict and poor behavior must be addressed. Scrum masters lead the team in inspecting its working agreement (discussed in the "Creating a working agreement" section) to build consensus, improved behavior, and team alignment.

Disrupting a performing team with staff changes sends the entire team, at least to a certain degree, back to the forming phase as each person once again learns how the change effects where they fit. High-performing teams should be leveraged rather than disrupted to build new teams. Agile organizations encourage teams to pull from an organizational backlog of value-driven opportunities rather than push people to staff traditional projects.

WARNING

Team member thrashing (multitasking) is costly. Effective multitasking is a myth. High-performing teams focus on one objective, do it well, and then attack the next most important objective. Thrashing is a failure of prioritization. If you want to improve performance, change the structures in place for prioritization. See Chapters 5 and 7 to learn more about the evils of thrashing.

The journey to become a high-performing team is not easy. Each of Tuckman's phases is filled with challenges, learning opportunities, and growth. When a team performs together well, keep it together as long as you possibly can.

In their famous 1986 article "The New New Product Development Game," which started the scrum analogy, Hirotaka Takeuchi and Ikujiro Nonaka revealed that high-performing teams share three attributes: They are autonomous, self-transcendent, and cross-fertilized. A high-performing team discovers and delivers outcomes that have significant effects on the customer:

- **Autonomous** in that they are able to set their own direction and act independently. They are self-organizing and self-managing.
- **Self-transcendent** in that the team pursues pushing the limit, continually establishing and then elevating its own goals.
- **Cross-fertilized** in that they are a diverse, multi-skilled team willing to share its knowledge and expertise.

These three factors above all else lead to high performance.

Focusing on fundamentals

Sports coaches often say, "Practice the fundamentals." The same is true for product development teams. High-performing teams became high performing because of their mastery of the fundamentals. This mastery is gained through experience. Understanding the fundamentals allows the team to continually improve its execution or team development. The migration of an team's journey to improved performance is often referred to as *Shu Ha Ri.*

Shu Ha Ri, a Japanese marital art concept in Aikido, is used to describe the stages of learning to mastery. In Chapter 20, we discuss *Shu Ha Ri* in the concept of maturing and solidifying organizational change movements. In this chapter, it's helpful to understand the importance of achieving *Shu* to get the fundamentals right first.

- ***Shu* (obey):** During this initial phase students follows the teachings of one master precisely. The concentration is on how to do the task. Multiple ways may be possible for completing the task, but the students focus on the one way taught by the master. Students commit the skill to memory and make it automatic.

- ***Ha* (detach):** At this point, students begin to branch out. They start to learn the underlying principles and theory behind the technique. They begin to consider ideas from other masters and incorporate those ideas into their practice. Students learn more about the skill from successes and failures as they improvise.
- ***Ri* (transcend):** During this phase students begin learning from their own practice rather than from other people. Students create their own approach and adapt what they've learned to their own circumstance. At the end of this phase, the skill comes naturally to these now-former students, who will know what works and what doesn't.

Shu Ha Ri focuses early stages of learning on steps to imitate, and then shifts to understanding principles and lastly to self-directed innovation. Solid understanding of the fundamentals make *Ha* and *Ri* possible.

High-performing teams use *Shu Ha Ri*. They learn and practice extensively the agile fundamentals over and over again. Sprint after sprint, their learning and performance improve, until they move into *Ha*, which brings a new set of learning for the team. *Ri* finally comes, but only after time and extensive effort have been invested in *Shu* and *Ha*, again reinforcing the importance of a long-lived team.

Creating a working agreement

Fundamental to a team's success is their *working agreement*, or expected norms for team behavior. Working agreements are guidelines defined by the team (not imposed on the team) as to how it agrees to work together to create a positive, productive development process. Many teams write out their agreements, sign them, and then post them on a wall for quick reference and constant reminders. A team will empower a team facilitator (often the scrum master) to help the team self-manage the agreement. Wise scrum masters help their teams to self-evaluate behavior against the agreed-to norms.

Working agreements benefit teams by helping them to

- Develop a sense of shared responsibility
- Increase awareness of each member's own behavior
- Empower the facilitator to lead the group according to the agreements
- Enhance the quality of the group process

Agreements work well when the agreement items are important to the team (meaning the team identifies the need and how to handle it), are limited in number, and are fully supported by each team member.

Examples of typical working agreement topics may include the following:

- Meetings: Start and end times, meeting behaviors, and etiquette
- Working together: Participation expectations, transparency methods, and consensus building

Team working agreements build behavior alignment. The best working agreements are defined by the team and for the team and are enforced by the team. Keep in mind that working agreements are different for every team. Just because one team struggles with a particular behavior doesn't mean every team's agreement needs to address that behavior. The best working agreements are brief — three to five items — making them easy to remember and self-enforce. The best working agreements should address only those items important to your team.

Enabling Autonomy, Mastery, and Purpose

Many organizations debate the best way to motivate people. Some question if money or other tangible rewards are motiving. Is it better to use the proverbial stick or carrot? Daniel H. Pink, in his best-selling 2009 book *Drive: The Surprising Truth About What Motivates Us* (Riverhead Books), reveals from his research that true motivation comes by giving team members autonomy, mastery, and purpose.

Autonomy

Autonomy is the freedom or independence to be creative in problem solving — to be the captain of your own destiny. High-performing teams are made up of people who are free to solve customer problems in the best way they see fit. Autonomy allows the team to become self-organizing and self-managing. High-performing teams can also behave autonomously, becoming empowered to do what is needed for their customers, sponsors, and stakeholders.

Mastery

Mastery is the internal drive the team demonstrates to excel in its trade. The opportunity to become curious and learn necessary skills, even multiple skills, is rewarding. Team members have pride in their craftsmanship and use their mastery to build quality into their products. After a skill or concept is learned, team members are quick to share what they've learn so that everyone on the team can improve. "Learn then teach" reinforces and supports the team's mastery.

Purpose

Purpose is the transcendent overarching objective for product development teams. It's the greater good for which they strive and give their full effort. The purpose is what brings them to the office each day feeling passion and determination for accomplishing great things. Studies, including a 2018 *Harvard Business Review* article written by Shawn Achor, Andrew Reece, Gabriella Rosen Kellerman, and Alexi Robichaux ("9 Out of 10 People Are Willing to Earn Less Money to Do More Meaningful Work"), show that people are willing to earn less money to do more meaningful work.

Teams and organizations who enable autonomy, mastery, and purpose are on the path toward improved team member motivation. As Principle 5 states, "Build projects around motivated individuals. Give them the environment and support they need, then trust them to get the job done."

TEAMMATES HELPING TEAMMATES

An extremely talented development team member was passionate about the quality of his team's product. At times, however, his passion became contentious as he berated his teammates for their lack of attention to detail and poor quality. Like a storm blowing in and leaving a wake of scattered debris, each occurrence lowered the entire team's energy and motivation. Discussions about this team member with HR were frequent.

To help his teammate, the scrum master called each morning during their morning commute to hold what they called a venting session. The scrum master asked questions, and then just listened while his teammate shared his concerns, fears, and frustrations. Never judging, but simply trying to be his friend, the scrum master was able to help his teammate reframe his perspective in preparation for the day of work.

The results from the daily ritual were amazing! The team member trusted the scrum master, having worked with him for an extended period, and was able to adjust his behavior. Teammates recognized the change and the entire team improved — it started reaching its sprint goals and more!

Both the team member and the scrum master look back on the experience with fondness, recognizing the deep foundation for their friendship, which has lasted longer than their employment with the organization. Enduring team relationships matter.

Highly aligned and highly autonomous teams

High-performing agile organizations have highly aligned and highly autonomous teams — almost like decomposing a large organizational ship into smaller speed boats, all pointing in the same direction.

Team autonomy helps the team to thrive. The team becomes empowered with the freedom and independence to do whatever is necessary to help the organization accomplish the aligned goal. Team autonomy helps each team leverage its uniqueness in solving the problems that only it can solve.

Leaders in agile organizations set strong visions for their teams but allow the teams to determine for themselves how to reach that vision.

Figure 8-1 describes the leaders' role in building alignment and autonomy across four quadrants from low to high. The maximum benefit is achieved when teams become both highly aligned and highly autonomous.

FIGURE 8-1: Highly aligned and autonomous team quadrants.

Building Team Knowledge and Capability

High-performing teams invest in education. They understand that learning is essential for building knowledge and capability. Although training in specific skills can be valuable, many teams find that becoming part of a community of practice or guild is also helpful.

A *community of practice (CoP)* is a group of people who share a common concern, set of problems, or interest in a topic and who come together to fulfill both individual and group goals. For example, product owners who want to learn how to improve their role participate in a community of practice with all other product owners, or people interested in learning more about product ownership participate to develop their skills. They discuss their challenges and keys for success, inspiring each other to experiment in their own situation. They collectively engage additional help, mentoring, and coaching from experts beyond their own expertise.

Communities of practice often focus on sharing best practices and creating knowledge to advance a domain of professional practice. Interaction on an ongoing basis is an important part of this. Many communities of practice rely on face-to-face meetings as well as web-based collaborative environments to communicate, connect, and conduct community activities.

Many communities of practice use the lean coffee approach to generating and discussing important topics, which is a simple way to ensure that the topics people care about most are given the majority of the community's time.

TECHNICAL STUFF

Lean coffee is a simple meeting facilitation technique. Participants take a few minutes at the beginning to brainstorm ideas they'd like to discuss with the group. Once submitted, the ideas are organized into themes. Each person is given an opportunity to then cast five dot votes (dots on topic Post-it Notes with a marker). Each person can allocate their dot votes however they like, whether it's placing five on one topic or one on five separate topics. Topics receiving the most votes are prioritized first. Each topic is given a set amount of time for discussion, say 8 minutes. When the time expires, the group votes using thumbs up or down to add 8 more minutes, then 5, then 3, and then they either agree or disagree to stop and move to the next topic. Participants leave lean coffee discussions having covered the topics most valuable to them.

Communities of practice can be formed for all other team roles, interests, or organizational disciplines such as architecture, user experience, security, training, and customer support. The sky is the limit! Community members leave their community of practice discussions with pragmatic ideas for building team knowledge and capability.

By understanding the hard work, dedication, learning, and capability development required to become a high-performing team, it stands to reason that the team will maximize its contribution the longer it stays together. Customers who use long-term products benefit most from high-performing, long-lived, and enduring teams.

3 Agile Planning and Execution

IN THIS PART . . .

Follow the Roadmap to Value, from product vision to execution.

Define and estimate requirements.

Create working functionality and showcase it in iterations.

Inspect your work and adapt your processes for continuous improvement.

IN THIS CHAPTER

- Planning agile product development
- Establishing the product vision
- Creating features and a product roadmap

Chapter 9

Defining the Product Vision and Product Roadmap

To start, let's dispel a common myth. If you've heard that agile product development doesn't include planning, dismiss that thought right now. You will plan not only the overall product but also every release, every sprint, and every day. Planning is fundamental to agile product development success.

If you're a project manager, you probably do the bulk of your planning at the beginning of a project. You may have heard the phrase, "Plan the work, then work the plan," which sums up non-agile project management approaches.

Agile product development, in contrast, involves planning up front and throughout the entire product lifecycle. By planning at the last responsible moment, right before an activity starts, you know the most about that activity. This type of planning, called *just-in-time planning* or a *situationally informed strategy,* is a key to success. Scrum teams plan as much as, if not more than, traditional project teams. However, agile planning is more evenly distributed throughout the product's life (see Figure 9-1) and is done by the entire team that will be working on the product.

FIGURE 9-1: Traditional planning versus scrum planning.

TECHNICAL STUFF

Helmuth von Moltke, a nineteenth-century German field marshal and military strategist, once said, "No plan survives contact with the enemy." That is, in the heat of a battle — much like in the thick of developing a product feature — plans always change. Just-in-time planning allows you to accommodate real-world changes non-disruptively and to be well-informed as you plan specific tasks.

This chapter describes how just-in-time planning works with agile product development. You also go through the first two steps of planning: creating the product vision and the product roadmap.

Agile Planning

Planning happens at a number of points. A great way to look at planning activities is with the Roadmap to Value. Figure 9-2 shows the roadmap as a whole.

The Roadmap to Value has seven stages:

- In stage 1, the product owner identifies the *product vision.* The product vision is your product's destination or end goal. The product vision includes the outer boundary of what your product will be, how the product is different than the competition, how the product will support your company or organization's strategy, who will use the product, and why people will use the product. The product vision should be revisited at least once a year.
- In stage 2, the product owner creates a *product roadmap.* The product roadmap is a high-level view of the product requirements, with a general time frame for when you will develop those requirements. It also gives context to the vision by showing the tangible features that will be produced during development. Identifying product requirements and then prioritizing and roughly estimating the effort for those requirements allow you to establish requirement themes and identify requirement gaps. The product owner, with support from the development team, should revise the product roadmap at least biannually.

FIGURE 9-2: Stages of agile planning and execution with the Roadmap to Value.

- In stage 3, the product owner creates a release plan. The *release plan* identifies a high-level timetable for the release of working functionality to the customer. The release serves as a mid-term boundary against which the scrum team can mobilize. Many releases may be required to accomplish the product vision and the highest-priority features should appear first. You create a release plan at the beginning of each release, which according to Principle 3 should be "frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale." Read more about release planning in Chapter 10.
- In stage 4, the product owner, the development team, and the scrum master will plan iterations, also called sprints, and start creating the product functionality in those sprints. *Sprint planning* sessions take place at the start of each sprint. During sprint planning, the scrum team determines a sprint goal, which establishes the immediate boundary of work that the team forecasts to accomplish during the sprint, with requirements that support the goal and can be completed in the sprint. The scrum team also outlines how to complete those requirements. Read more about sprint planning in Chapter 10.

- In stage 5, the development team has *daily scrum* meetings during each sprint to coordinate the day's priorities for accomplishing the sprint goal. In the daily scrum meeting, based on what was completed up to that point, you coordinate what you will work on today and any roadblocks, so that you can address issues immediately. Read about daily scrums in Chapter 11.
- In stage 6, the scrum team holds a *sprint review* at the end of every sprint. In the sprint review, you demonstrate the working product to the product stakeholders. Find out how to conduct sprint reviews in Chapter 12.
- In stage 7, the scrum team holds a *sprint retrospective.* The sprint retrospective is a meeting where the scrum team discusses the completed sprint with regard to their processes and environment and makes plans for process improvements in the next sprint. Like the sprint review for inspecting and adapting the product, a sprint retrospective is held at the end of every sprint to inspect and adapt your processes and environment. Find out how to conduct sprint retrospectives in Chapter 12.

Each stage of the Roadmap to Value is repeatable and contains planning activities. Agile planning, like agile development, is iterative and barely sufficient.

Progressive elaboration

During each stage of product development, you plan only as much as you need to plan. In the early stages of your work, you plan widely and holistically to create a broad outline of how the product will shape up over time. In later stages, you narrow your planning and add more details to ensure success in the immediate development effort. This process is called a *progressive elaboration of requirements.*

Planning broadly at first and in detail later, when necessary, prevents you from wasting time on planning lower-priority product requirements that may never be implemented. This model also lets you add high-value requirements during product development without disrupting the flow.

The more just-in-time your detailed planning is, the more effective your planning becomes.

REMEMBER

Standish Group studies show that customers rarely or never use as much as 80 percent of the features in an application. In the first few development cycles of an agile product development effort, you complete features that have the highest priority and that people will use. Typically, you release those groups of features as early as possible to gain market share through first-mover advantage; receive customer feedback for viability; monetize functionality early to optimize return on investment (ROI); and avoid internal and external obsolescence.

Inspect and adapt

Just-in-time planning brings into play two fundamental tenets of agile techniques: Inspect and adapt. At each stage of development, you need to look at the product and the process (inspect) and make changes as necessary (adapt).

Agile planning is a rhythmic cycle of inspecting and adapting. Consider the following:

- Each day during the sprint, the product owner provides feedback to help improve the product as the development team creates the product.
- At the end of each sprint, in the sprint review, stakeholders provide feedback to further improve the product.
- At the end of each sprint in the sprint retrospective, the scrum team discusses the lessons it learned during the past sprint to improve the development process.
- After a release, the customers can provide feedback for improvement. Feedback might be direct, when a customer contacts the company about the product, or indirect, when potential customers either do or don't purchase the product.

Together, inspect and adapt are fantastic tools for delivering the right product in the most efficient manner.

REMEMBER

At the beginning of development, you know the least about the product you're creating, so trying to plan fine details at that time just doesn't work. Being agile means, you do the detailed planning when you need it, and immediately develop the specific requirements you defined with that planning. Remember the Agile Manifesto value: "Responding to change over following a plan."

Now that you know a little more about how agile planning works, it's time to complete the first step: defining the product vision.

Defining the Product Vision

The first stage in the Roadmap to Value is defining your product vision. The *product vision statement* is an elevator pitch, or a quick summary, to communicate how your product supports the company's or organization's strategies. The vision statement must articulate the end state for the product.

The product might be a commercial product for release to the marketplace or an internal solution that will support your organization's day-to-day functions. For example, say your company is XYZ Bank and your product is a mobile banking application. What company strategies does a mobile banking application support? How does the application support the company's strategies? Your vision statement clearly and concisely links the product to your business strategy. As well-known author and spokesperson Simon Sinek explains, this is your "Why."

Figure 9-3 shows how the vision statement — stage 1 of the Roadmap to Value — fits with the rest of the stages and activities in product development.

A common agile practice

Stage 1: PRODUCT VISION

Description: Goals for the product and its alignment with the company's strategy
Owner: Product owner
Frequency: At least annually

FIGURE 9-3: The product vision statement as part of the Roadmap to Value.

The product owner is responsible for knowing about the product, its goals, and its requirements throughout development. For those reasons, the product owner creates the vision statement, although other people may have input. After the vision statement is complete, it becomes a guiding light, the "what we are trying to achieve" statement that the product owner, development team, scrum master, and stakeholders refer to throughout their work.

When creating a product vision statement, follow these four steps:

1. **Develop the product objective.**
2. **Create a draft vision statement.**
3. **Validate the vision statement with product stakeholders and revise it based on feedback.**
4. **Finalize the product vision statement.**

The look of a vision statement follows no hard-and-fast rules. However, anyone involved, from the development team to the CEO, should be able to understand the statement. The vision statement should be internally focused, clear, nontechnical, emotionally connecting, and as brief as possible. The vision statement should also be explicit and avoid marketing jargon.

Step 1: Developing the product objective

To write your vision statement, you must understand and be able to communicate the product's objective. You need to identify the following:

- **Key product goals:** How will the product benefit the company that is creating it? The goals may include benefits for a specific department in your company, such as customer service or the marketing department, as well as the company as a whole. What specific company strategies does the product support? The product canvas discussed in Chapter 4 is helpful for defining product goals.
- **Customer:** Who will use the product? This question might have more than one answer.
- **Need:** Why does the customer need the product? What features are critical to the customer? What problem will the product solve, as discussed in Chapter 4?
- **Competition:** How does the product compare with similar products?
- **Primary differentiation:** What makes this product different from the status quo or the competition or both?

Step 2: Creating a draft vision statement

After you have a good grasp of the product's objective, create a first draft of your vision statement.

You can find many templates for a product vision statement. For an excellent guide to defining the overall product vision, see *Crossing the Chasm*, by Geoffrey Moore (published by HarperCollins), which focuses on how to bridge the gap (chasm) between early adopters of new technologies and the majority who follow.

The adoption of any new product is a gamble. Will users like the product? Will the market take to the product? Will there be an adequate return on investment for developing the product? An effectively written product vision statement can start you on the path to quickly learning the answers to these questions.

TECHNICAL STUFF

Return on investment, or *ROI*, is the benefit or value a company gets from paying for something. ROI can be quantitative, such as the additional money ABC Products makes from selling widgets online after investing in a new website. ROI can also be something intangible, such as better customer satisfaction for XYZ Bank customers who use the bank's new mobile banking application.

By creating your vision statement, you help convey your product's quality, maintenance needs, and longevity.

Moore's product vision approach is pragmatic. In Figure 9-4, we construct a template based on Moore's approach to more explicitly connect the product to the company's strategies. If you use this template for your product vision statement, it will stand the test of time as your product goes from early adoption to mainstream usage.

Vision Statement for Product

For _____ (target customer)
who _____ (needs)
the _____ (product name)
is a _____ (product category)
that _____ (product benefit, reason to buy)
Unlike _____ (competitors)
our product _____ (differentiation/value proposition)

FIGURE 9-4: Expansion of Moore's template for a vision statement.

TIP

One way to make your product vision statement more compelling is to write it in the present tense, as if the product already exists. Using present tense helps readers imagine the product in use.

Using our expansion of Moore's template, a vision statement for a mobile banking application might look like the following:

For XYZ Bank customers

who want access to banking capability while on the go,

the MyXYZ

is a mobile application

that allows secure, on-demand banking, 24 hours a day.

Unlike online banking from your home or office computer,

our product allows users immediate access,

which supports our strategy to provide quick, convenient banking services, anytime, anywhere. (Platinum Edge addition)

As you can see, a vision statement identifies a future state for the product when the product reaches completion. The vision focuses on the conditions that should exist when the product is complete.

WARNING

Avoid generalizations in your vision statement such as "make more money" or "make customers happy" or "sell more products." You want the vision statement to help you make scope decisions throughout product development. Also watch out for too much technological specificity, such as "using release 9.x of Java, create a program with four modules that . . ." At this early stage, defining specific technologies might limit you later.

Here are a few extracts from vision statements that should ring warning bells:

- Secure additional customers for the MyXYZ application.
- Satisfy our customers by December.
- Eliminate all defects and improve quality.
- Create a new application in Java.
- Beat the Widget Company to market by six months.

Step 3: Validating and revising the vision statement

After you draft your vision statement, review it against the following quality checklist:

- Is this vision statement clear, focused, and written for an internal audience?
- Does the statement provide a compelling description of how the product meets customer needs?
- Does the vision describe the best possible outcome?
- Is the business objective specific enough that the goal is achievable?
- Does the statement deliver value that is consistent with corporate strategies and goals?
- Is the vision statement compelling?
- Is the vision concise?

These yes-or-no questions will help you determine whether your vision statement is thorough. If any answers are no, revise the vision statement.

When all answers are yes, move on to reviewing the statement with others, including the following:

- **Product stakeholders:** The stakeholders will be able to identify that the vision statement includes everything the product should accomplish.
- **Your development team:** The people who will be creating the product must also understand and have ownership of what the product needs to accomplish. Many product owners create the product vision with the development team, aligning purpose and motivation (Principle 5).
- **Scrum master:** A strong understanding of the product will help the scrum master proactively remove roadblocks, enabling the team to accomplish the product vision.
- **Agile mentor:** Share the vision statement with your agile mentor, if you have one. The agile mentor is independent of the organization and can provide an external perspective, qualities that can make for a great objective voice.

See whether others think the vision statement is clear and delivers the message you want to convey. Review and revise the vision statement until the stakeholders, development team, and scrum master fully understand the statement.

Step 4: Finalizing the vision statement

After you finish revising the vision statement, make sure the development team, scrum master, and stakeholders have the final copy. You might even put a copy on the wall in the scrum team's work area, where you can see it every day. You will refer to the vision statement throughout the life of the product.

If your product development will be more than a year long, you may want to revisit the vision statement. Review the product vision statement at least once a year to make sure the product reflects the marketplace and supports any changes in the company's needs. Because the vision statement is the long-term boundary of the product, product development investment should end when the vision is achieved and expansion of the vision is no longer viable.

REMEMBER

The product owner owns the product vision statement and is responsible for its preparation and communication across and outside the organization. The product vision sets expectations for stakeholders and helps the development team stay focused on the goal.

Congratulations. You've just completed the initial definition of strategy and desired value outcome in your agile product development. Now it's time to create a product roadmap.

Creating a Product Roadmap

The product roadmap, stage 2 in the Roadmap to Value (see Figure 9-5), is an overall view of the product's requirements and a valuable tool for planning and organizing the journey of product development. Use the product roadmap to categorize requirements, prioritize them, identify gaps and dependencies, and determine a sequence for releasing to the customer.

A common agile practice

Stage 2: PRODUCT ROADMAP

Description: Holistic view of product features that create the product vision
Owner: Product owner
Frequency: At least semiannually

FIGURE 9-5: The product roadmap as part of the Roadmap to Value.

As with the product vision statement, the product owner creates the product roadmap with help from the development team and stakeholders. The development team often participates to a greater degree than it did during the creation of the vision statement.

TIP

Keep in mind that you will refine requirements and effort estimates throughout development. In the product roadmap phase, it's okay for your requirements detail, estimates, and time frames to be at a very high level.

To create your product roadmap, you do the following:

1. **Identify stakeholders.**
2. **List product requirements and visualize them.**
3. **Arrange the product requirements based on value, risk, and dependencies.**
4. **Estimate the development effort at a high level and prioritize the product's requirements.**
5. **Determine high-level time frames for releasing groups of functionality to the customer.**

Because priorities can change, plan to update your product roadmap at least twice a year.

TIP

Your product roadmap can be as simple as sticky notes arranged on a physical or virtual whiteboard, which makes updates as easy as moving a sticky note from one section of the whiteboard to another.

You use the product roadmap to plan releases — stage 3 in the Roadmap to Value. *Releases* are groups of usable product functionality that you release to customers to gather real-world feedback and to generate return on investment.

The following section details the steps to create a product roadmap.

Step 1: Identifying product stakeholders

When initially establishing the product vision, it's likely you will have identified only a few key stakeholders who are available to provide high-level feedback. At the product roadmap stage, you put more context to the product vision and identify how you achieve the vision, which gives more insight into who will have a stake in your product.

This is the time to engage with existing and newly identified stakeholders to gather feedback on the functionality you want to implement to achieve the vision. The product roadmap is your first cut at a high-level product backlog, discussed later in this chapter. With this first round of detail identified, you'll want to engage more than just the scrum team, the product sponsor, and obvious users. Consider including the following people:

- **Marketing department:** Your customers need to know about your product, and that's what the marketing department provides. They need to understand your plans and may have input into the order in which you release functionality to the market, based on their experience and research.
- **Customer service department:** Once your product is in the market, how will it be supported? Specific roadmap items might identify the person you'll need to prepare for support. For instance, a product owner may not see much value in plugging in a live online chat feature, but a customer service manager may see it differently because his or her representatives can handle simultaneously only one phone call but as many as six chat sessions. Plus, your customer service representatives actually talk to your end users on a daily basis and probably have many insights you should consider.
- **Sales department:** Make sure that the sales team sees the product so that it starts selling the same thing you're building. Like the marketing department, the sales department will have first-hand knowledge about what your customers are looking for.

- **Legal department:** Especially if you're in a highly regulated industry, review your roadmap with legal counsel as early as possible to make sure you haven't missed anything that could put your product at risk if discovered later.
- **Additional customers:** While identifying features on your roadmap, you may discover additional people who will find value in what you will create. Give them a chance to review your roadmap to validate your assumptions.

Step 2: Establishing product requirements

The second step in creating a product roadmap is to identify, or define, the different requirements for your product.

When you first create your product roadmap, you typically start with large, high-level requirements. The requirements on your product roadmap will most likely be at two different levels: themes and features. *Themes* are logical groups of features and requirements at their highest levels. *Features* are parts of the product at a very high level and describe a new capability the customer will have once the feature is complete.

DECOMPOSING REQUIREMENTS

Throughout product development, you'll break down requirements into smaller, more manageable parts using a process called *decomposition,* or *progressive elaboration.* You can break down requirements into the following sizes, listed from largest to smallest:

- **Themes:** A *theme* is a logical group of features and is also a requirement at its highest level. You may group features into themes in your product roadmap.
- **Features:** *Features* are parts of products at a very high level. Features describe a new capability the customers will have once the feature is complete. You use features in your product roadmap.
- **Epic user stories:** *Epics* are medium-sized requirements that are decomposed from a feature and often contain multiple actions or channels of value. You need to break down your epics before you can start creating functionality from them. You can find out how you use epics for release planning in Chapter 10.

(continued)

(continued)

- **User stories:** *User stories* are requirements that contain a single action or integration and are small enough to start implementing into functionality. You see how you define user stories and use them at the release and sprint level in Chapter 10.
- **Tasks:** *Tasks* are the execution steps required to develop a requirement into working functionality and generally mirror your definition of done as well as the tasks necessary to accomplish the story's acceptance criteria. You can find out about tasks and sprint planning in Chapter 10.

Keep in mind that each requirement may not go through all these sizes. For example, you may create a particular requirement at the user story level, and never think of it on the theme or epic scale. You may create a requirement at the epic user story level, but it may be a lower-priority requirement. Because of just-in-time planning, you may not take the time to decompose that lower-priority epic user story until you complete development of all the higher-priority requirements.

To identify product themes and features, the product owner can work with stakeholders and the development team. It may help to have a product discovery workshop, where the stakeholders and the development team meet and write down as many requirements as possible. Each item should be written in the customer's words rather than using technical jargon. For example, you might begin each item with the words, "My customer can now . . ." to reinforce how the requirements should relate to the customer, their problem, and the product vision. For example:

My customer can now

See her account balance

Pay bills

See her latest transactions

Provide feedback

Get help

TIP

When you start creating requirements at the theme and feature level, it can help to write those requirements on index cards or big sticky notes. Using a physical card that you can move from one category to another and back again can make organizing and prioritizing those requirements very easy.

While you create the product roadmap, the features you identify start to make up your *product backlog* — the full list of what is in scope for a product, regardless of level of detail. Once you have identified your first product features, you have your product backlog started.

Step 3: Arranging product features

After you identify your product features, you work with the stakeholders to group them into *themes* — common, logical groups of features. A stakeholder meeting works well for grouping features, just like it works for creating requirements. You can group features by usage flow, technical similarity, or business need.

Visualizing themes and features on your roadmap allow you to assign business value and risks associated with each feature relative to others. The product owner, along with the development team and stakeholders, can also identify dependencies between features, locate any gaps, and prioritize the order in which each feature should be developed based on each of these factors.

Here are questions to consider when grouping and ordering your requirements:

- How would customers use our product?
- If we offered this requirement, what else would customers need to do? What else might they want to do?
- Can the development team identify technical affinities or dependencies?

Use the answers to these questions to identify your themes. Then group the features by these themes. For example, in the mobile banking application, the themes might be

- Account information
- Transactions
- Customer service functions
- Integration with other accounts

Figure 9-6 shows features grouped by themes.

FIGURE 9-6: Features grouped by themes.

Step 4: Estimating efforts and ordering requirements

You've identified your product requirements and arranged those requirements into logical groups. Next, you estimate and prioritize the requirements. Here are a few terms you need to be familiar with:

- *Effort* is the ease or difficulty of creating functionality from a particular requirement.
- An *estimate,* as a noun, can be the number or description you use to express the estimated effort of a requirement.
- *Estimating* a requirement, as a verb, means to come up with an approximate idea of how easy or hard (how much effort) that requirement will be to create.
- *Ordering,* or *prioritizing,* a requirement means to determine that requirement's value and risk in relation to other requirements, and in what order you will implement them.

- *Value* means how beneficial a product requirement might be to the customer and therefore the organization creating that product.
- *Risk* refers to the negative effect a requirement can have due to customer uncertainty or on product development.

TIP

You can estimate and prioritize requirements of any estimate size at any level, from themes and features down to single user stories.

Prioritizing requirements is really about ordering them. You can find various methods — many of them complicated — for determining the priority of product backlog items. We keep things simple by creating an ordered to-do list of product backlog items based on business value, risk, and effort, listed in the order in which we will implement them. Forcing an order requires making a priority decision for every requirement relative to every other requirement. A scrum team works on one thing at a time, so it's important to format your product roadmap accordingly. In Chapter 13, additional prioritization techniques are discussed.

To estimate and assign effort values to your requirements, you work with two different groups of people:

- The development team determines the effort to implement the functionality for each requirement. Only the people who will do the work should provide effort estimates. The development team also provides critical feedback to the product owner for understanding how technical risks affect the ordering of the product backlog.
- The product owner, with support from the stakeholders, determines the value and risk of the requirement to the customer and the business.

Estimating effort

To order requirements, the development team must first estimate the effort for each requirement relative to all other requirements.

In Chapter 10, we show you relative estimation techniques that scrum teams use to accurately estimate effort. Traditional estimation methods aim for precision by using absolute time estimates at every level of the project schedule, whether the team is working on the work items today or two years from now. This practice gives traditional teams a false sense of precision and isn't accurate in reality (as thousands of failed projects prove). How could you possibly know what each team member will be working on six months from now, and how long it will take to do that work, when you are just starting to learn about the product at the beginning?

Relative estimating is a self-correcting mechanism that allows scrum teams to be more accurate because it's much easier to be right when comparing one requirement against another and determining whether one is bigger than another, and by roughly how much. Product development teams value accuracy over precision.

To order your requirements, you also want to know any dependencies. *Dependencies* mean that one requirement is a predecessor for another requirement. For example, if you have an application that requires users to set up a user profile, they'll need to be authenticated by a username and password. The requirement for creating the username and password would be dependent on setting up a profile because you generally need a username and password to establish a user profile.

Assessing business value and risk

Together with stakeholders, the product owner identifies the highest business value items (either high potential ROI or other perceived value to the end customer), as well as those items with high negative effect if unresolved.

Similar to effort estimates, values or risks can be assigned to each product roadmap item. For example, you might assign value using monetary ROI amounts or, for an internally used product, assign value or risk by using high, medium, or low.

Effort, business value, and risk estimates inform the product owner's prioritization decisions for each requirement. The highest value and risk items should be at the top of the product roadmap. High-risk items should be explored and implemented first to avoid rear-loading the risk. If a high-risk item will cause the product or its development to fail (an issue that cannot be resolved), scrum teams learn about it early. If something is going to fail, you want to fail early, fail cheap, and move on to a new opportunity that has value. In that sense, failure is a form of success for a scrum team.

After you have your value, risk, and effort estimates, you can determine the relative priority, or order, of each requirement.

- A requirement with high value or high risk (or both) and low effort will have a high relative priority. The product owner might order this item at the top of the roadmap.
- A requirement with low value or low risk (or both) and high effort will have a lower relative priority. This item will likely end up towards the bottom of the roadmap or, better yet, be removed. If anything on your roadmap does not support the fulfillment of your product vision, you may want to ask whether it is truly needed. Remember Principle 10, "Simplicity — the art of maximizing the amount of work not done — is essential."

WARNING

Relative priority is only a tool to help the product owner make decisions and prioritize requirements. It isn't a mathematical universal that you must follow. Make sure your tools help rather than hinder.

Prioritizing requirements

To determine the overall priority for your requirements, answer the following questions:

- What is the relative priority of the requirement?
- What are the prerequisites for any requirement?
- What set of requirements belong together and will constitute a solid set of functionalities you can release to the customer?

Using the answers to these questions, you can place the highest-priority requirements first in the product roadmap. When you've finished prioritizing your requirements, you'll have something that looks like Figure 9-7.

FIGURE 9-7: Product roadmap with ordered requirements.

Your prioritized list of requirements is called a *product backlog.* Your product backlog is an important agile document, or *artifact.* You use this backlog throughout your entire product development.

With a product backlog in hand, you can start adding target releases to your product roadmap.

Step 5: Determining high-level time frames

When you create your product roadmap, your time frames for releasing product requirements are at a very high level. For the initial roadmap, choose a logical time increment for your product development, such as a certain number of days, weeks, months, quarters (three-month periods), or even larger increments. Using both the requirement and the priority, you can add requirements to each increment of time.

REMEMBER

Creating a product roadmap might seem like a lot of work, but every team we've worked with has a product vision, product roadmap, and release plan for its first release, and is ready to start its sprint in as little as two to three days! To begin developing the product, you need only enough requirements for your first sprint. Having bought yourself some time, you can determine the rest as the team learns more from reality through progressive elaboration.

Saving your work

Up until now, you could do all your roadmap planning with whiteboards and sticky notes. After your first full draft is complete, however, save the product roadmap, especially if you need to share the roadmap with remote stakeholders or development team members. You could take a photo of your sticky notes and whiteboard, or you could type the information into a document and save it electronically. Whatever format you choose, ensure that the roadmap can be easily changed and transparently accessed.

You update the product roadmap throughout development, as priorities change. For now, the contents of the first release should be clear — and that's all you need to worry about to start executing and delivering value.

Completing the Product Backlog

The product roadmap contains high-level features and some tentative release timelines. The requirements on your product roadmap are the first version of your *product backlog.*

The product backlog is the list of all requirements associated with the product. The product owner is responsible for creating and maintaining the product backlog by updating (adding, changing, removing) and prioritizing requirements. The scrum team uses the prioritized product backlog throughout development to plan its work each release and each sprint.

Figure 9-8 shows a sample product backlog. At a minimum, when creating your product backlog, be sure to do the following:

- Include a description of your requirement.
- Order the requirements based on priority.
- Add the effort estimate.

PRODUCT BACKLOG

Order	ID	Item	Type	Status	Estimate
1	121	As an Administrator, I want to link accounts to profiles, so that customers can access new accounts.	Requirement	Not Started	5
2	113	Update requirements traceability matrix.	Overhead	Not Started	2
3	403	Test automation training for Michael.	Improvement	Not Started	3
4	97	Refactor Login Class.	Maintenance	Not Started	8
5	68	As a Site Visitor, I want to find locations, so that I can use bank services.	Requirement	Not Started	8

FIGURE 9-8: Product backlog items sample.

We also like to include the type of backlog item as well as the status. Teams will work mainly on developing features as described in the words of the user (user stories). But there may be a need for other types of product backlog items, such as overhead items (things the team determines are needed but don't contribute to the functionality), maintenance items (design improvements that need to be done to the product or system but don't directly increase value to the customer), or improvement items (action items for process improvements identified in the sprint retrospective). You can see examples of each of these in Figure 9-8. The product owner prioritizes all product backlog items through the lens of the customer and stakeholders.

REMEMBER

In Chapter 2, we explain how documents for agile product development should be barely sufficient, with only information that is absolutely necessary to create the product. If you keep your product backlog format simple and barely sufficient, you'll save time updating it throughout product development.

The scrum team refers to the product backlog as the main source for requirements. If a requirement exists, it's in the product backlog.

The user stories in your product backlog will change throughout development in several ways. For example, as the team completes user stories, you mark those stories as complete in the backlog. You also record any new user stories. Some user stories will be updated with new or clarified information, broken down into smaller user stories, or refined in other ways. Additionally, you update the priority and effort scores of existing user stories as needed.

The total number of story points in the product backlog — all user story points added together — is your current *product backlog estimate.* This estimate changes daily as user stories are completed and new user stories are added. See Chapter 15 for more about using the product backlog estimate to predict the release length and cost.

REMEMBER

Keep your product backlog up to date so that you always have accurate cost and schedule estimates. A current product backlog also gives you the flexibility to prioritize newly identified product requirements — a key agile benefit — against existing features.

After you have a product backlog, you can begin planning releases and sprints, which we describe in the next chapter.

IN THIS CHAPTER

- Decomposing requirements and creating user stories
- Creating a product backlog, release plan, and sprint backlog
- Getting the product ready to ship and preparing the rest of the organization for the release
- Making sure the marketplace is ready

Chapter 10

Planning Releases and Sprints

After you create a product roadmap for your product (see Chapter 9) it's time to start elaborating on your product details. In this chapter, you discover how to break down your requirements to a more granular level, refine your product backlog, create a release plan, and build a sprint backlog for execution. We also discuss how to prepare the rest of the organization for the release, including operational support, and ensuring that the marketplace will be ready for your release.

First, you see how to break down the larger requirements from your product roadmap into smaller, more manageable requirements called *user stories.*

Refining Requirements and Estimates

You start agile development with very large requirements. As the work progresses and you get closer to developing those requirements, you will break them down into smaller parts — small enough to begin developing. See Chapter 9 for more about this process, which is known as *decomposition.*

One clear, effective pattern for defining product requirements is the user story. In this section, you find out how to create a user story, prioritize user stories, and estimate user story effort.

What is a user story?

The *user story* is a simple description of a product requirement in terms of what that requirement must accomplish for whom. It's called a *story* because the simplest way people tell stories is by talking to each other. User stories are most effective when they're told by speaking to each other face-to-face. The written pattern described in this section can be used to help with that conversation.

Traditional software requirements usually read something like this: "The system shall [insert technical description]." This requirement addresses only the technical nature of what will be done; the overall business objective is unclear. Because the development team has the context to engage more deeply through the user story pattern, its work becomes more personal and real. The team comes to know the benefit of each requirement to the user (or the customer or the business) and delivers what the customer wants faster and with higher quality.

Your user story will have, at a minimum, the following parts:

Title (recognizable name for the user story)

As a (type of user)

I want to (take this action)

so that (I get this benefit)

The user story tells the "who," "what," and "why" for the desired functionality. The user story should also be supported by a list of validation steps *(acceptance criteria)* to take so you can confirm whether the working requirement for the user story is correct. Acceptance criteria follows this pattern:

When I (take this action), (this happens)

User stories may also include the following:

- **A user story ID:** A unique identifying number to differentiate this user story from other user stories in the product backlog tracking system.
- **The user story value and effort estimate:** *Value* is how beneficial a user story might be to the organization creating that product. *Effort* is the ease or difficulty in creating that user story. We introduce how to score a user story's business value, risk, and effort in Chapter 9.

» **The name of the person who thought of the user story:** Anyone can create a user story.

TIP

Although agile product development approaches encourage low-tech tools, the scrum team should also find out what works best in each situation. A lot of electronic user story tools are available, some of which are free. Some are simple and are only for user stories. Others are complex and will integrate with other product documents. We love the simplicity of index cards, but use what works best for your scrum team and your product.

Figure 10-1 shows a typical user story card, front and back. The front has the main description of the user story. The back shows how you will confirm that the requirement works correctly, after the development team has created the functionality.

FIGURE 10-1: Card-based user story example.

TIP

The three Cs formula for user story creation — card, conversation, and confirmation — illustrates how the user story pattern enables scrum teams to create customer value. By limiting user stories to fit on a 3x5 index *card*, we encourage a *conversation* for achieving a shared understanding of the job to be done for the customer (rather than excessive documentation that implies nothing is left to discuss). If the conversation is supported by the answers to the quiz up front (acceptance criteria that is a *confirmation* that the actions the user will take meet the intended needs), you're probably on the right track.

The product owner gathers the user stories and manages them (that is, determines the priority and initiates the decomposition discussions). It is not the sole responsibility of the product owner to write user stories. The development team and other stakeholders are also involved in creating and decomposing user stories to ensure clarity and shared understanding across the scrum team.

TIP

Note that user stories aren't the only way to describe product requirements. You could simply make a list of requirements without any given structure. However, because user stories include a lot of useful information in a simple, compact format, we find that they're very effective in conveying exactly what a requirement needs to do for the customer.

The big benefit of the user story pattern is realized when the development team starts to create and test requirements. The development team members know exactly for whom they are creating the requirement, what the requirement should do, and how to double-check that the requirement satisfies its intention. Using the user's voice is something all can understand and relate to — not so much with technical jargon.

We use user stories as examples of requirements for software product development throughout the chapter and the book. Keep in mind that anything we describe that you can do with user stories, you can do also with more generically expressed requirements and other product types.

Steps to create a user story

When creating a user story, follow these steps:

1. **Identify the stakeholders.**
2. **Identify who will use the product.**
3. **Working with the stakeholders, write down, in a user story format, what the product will need to do.**

Find out how to follow these three steps in the following sections.

REMEMBER

Being agile and adaptive requires iterating. Don't spend a ton of time trying to identify every single requirement your product might have. You can always add items to your product backlog later. The best changes often come at the end, when you know the most about the product and the end-customers.

Identifying product stakeholders

You probably have a good idea about who your stakeholders are — anyone involved with, affected by, or who can affect the product and its creation. Stakeholders provide valuable feedback about every product increment you deliver each sprint.

REMEMBER

You will work with stakeholders also when you create the product vision and product roadmap.

Make sure the stakeholders are available to help you gather and write product backlog items. Stakeholders of the sample mobile banking application introduced in Chapter 9 might include the following:

- People who interact with customers on a regular basis, such as customer service representatives or bank branch personnel.
- Business experts for the different areas where your product's customers interact. For example, XYZ Bank might have one manager in charge of checking accounts, another manager in charge of savings accounts, and a third manager in charge of online bill payment services. If you're creating a mobile banking application, all these people would be stakeholders.
- Users of your product.
- Experts of the type of product you're creating. For example, a developer who has created mobile applications, a marketing manager who knows how to create mobile campaigns, and a user experience specialist who specializes in mobile interfaces all might be helpful on the sample XYZ Bank mobile banking product.
- Technical stakeholders. These are people who work with the systems that might need to interact with your product.

Identifying users

As discussed in Chapter 4, agile product development is customer focused. Building on the personas you've defined, your understanding of their needs, and the problems to be solved helps the team more clearly understand the product requirements. Knowing who your end users are and how they will interact with your product drives how you define and implement each item on your product roadmap.

With your product roadmap visualized, you can identify each type of user. For the mobile banking application, you would have individual and business bankers. The individual categories would include youth, young adults, students, and single, married, retired, and wealthy users. Businesses of all sizes might be represented. Employee users would include tellers, branch managers, account managers, and fund managers. Each type of user will interact with your application in different ways and for different reasons. Knowing who these people are enables you to better define the purpose and desired benefits of each of their interactions.

We like to define users using *personas,* or a written description about a type of user represented by a hypothetical person. For instance, "Ellen is a 65-year-old retired engineer who is spending her retirement traveling the world. Her net worth is $1,000,000, and she has residual income from several investment real estate properties." Read more about personas in Chapter 4.

Ellen represents 30 percent of XYZ Bank's customers, and a good portion of the product roadmap includes features that someone like Ellen will use. Instead of repeating all the details about Ellen every time the scrum team discusses these features, the team can simply refer to the type of user as Ellen. The product owner might identify several personas, as needed, and will even print the descriptions with a stock photo of what Ellen might look like and post them on the wall in the team's work area to refer to throughout development.

TIP

Know who your users are, so you can develop features they'll actually use.

Suppose that you're the product owner for the XYZ Bank's mobile banking product. You're responsible for the department that will bring the product to market, preferably in the next six months. You have the following ideas about the application's users:

- The customers (the end users of the application) probably want quick access to up-to-date information about their balances and recent transactions.
- Maybe the customers are about to buy a large-ticket item, and they want to make sure they can charge it.
- Maybe the customers' ATM cards were just refused, but they have no idea why, and they want to check recent transactions for possible fraudulent activities.
- Maybe the customers just realized that they forgot to pay their credit card bill and will have penalty charges if they don't pay the card today.

Who are your personas for this application? Here are a few examples:

- **Persona #1:** Jason is a young, tech-savvy executive who travels a lot. When he has a spare moment, he wants to handle personal business quickly. He carefully invests his money in high-interest portfolios. He keeps his available cash low.
- **Persona #2:** Carol is a small-business owner who stages properties when clients are trying to sell their home. She shops at consignment centers and often finds furnishings she wants to buy for her clients.
- **Persona #3:** Nick is a student who lives on student loans and a part-time job. He knows he can be flaky with money because he's flaky with everything else. He just lost his checkbook.

TIP

Your product stakeholders can help you create personas. Find people who are experts on the day-to-day business for your product. Those stakeholders will know a lot about your potential customers.

Determining product requirements and creating user stories

After you have identified your different users, you can start to determine product requirements and create user stories for the personas. A good way to create user stories is to bring your stakeholders together for a product discovery workshop. See more about product discovery workshops in Chapter 4.

Have the stakeholders write down as many requirements as they can think of, using the user story format. One user story for the product and personas from the previous sections might be as follows:

- Front side of card:
  - **Title** See bank account balance
  - **As** Jason,
  - **I want to** see my checking account balance on my smartphone
  - **so that** I can decide whether I have enough money in my account to make a transaction
- Back side of card:
  - **When I** sign into the XYZ Bank mobile application, my checking account balance appears.
  - **When I** sign into the XYZ Bank mobile application after making a purchase or a deposit, my checking account balance reflects that purchase or deposit.

You can see sample user stories in card format in Figure 10-2.

REMEMBER

Be sure to continuously add and prioritize new user stories to your product backlog. Keeping your product backlog up-to-date will help you have the highest-priority user stories when it is time to plan your sprint.

You will create new user stories throughout product development. You'll also take existing large requirements and decompose them until they're manageable enough to work on during a sprint.

Title Transfer money between accounts

As Carol,

I want to categorize expenses,

so that I can easily identify my purchases made for my clients.

Value	Author	Estimate
	Jennifer	

Title Put stop on a check

As Nick,

I want to stop payment on a lost or stolen check,

so that I can avoid any unauthorized activity on my account.

Value	Author	Estimate
	Caroline	

FIGURE 10-2: Sample user stories.

Breaking down requirements

You refine requirements many times throughout development. For example:

- When you create the product roadmap (see Chapter 9), you create *features* (capabilities your customers will have after you develop the features), as well as *themes* (logical groups of features). Although features are intentionally large, we require features at the product roadmap level to be no larger than 144 story points on the Fibonnaci scale (see the "Estimation poker" section later in the chapter to learn about Fibonacci sizing). Both features and themes are considered large by a development team.
- When you plan releases, you break down the features into more concise user stories. User stories at the release plan level can be either *epics,* very large user stories with multiple actions, or individual user stories, which contain a single action. For our clients, user stories at the release plan level should be no larger than 34 story points. You find out more about releases later in this chapter.
- When you plan sprints, you break down requirements even further. User stories are broken down to eight points or fewer. See Figure 10-3 for a helpful requirement decomposition guide.

FIGURE 10-3: User story decomposition guidelines.

To decompose requirements, you'll want to think about how to break down the requirement into individual actions. Table 10-1 shows a requirement from the XYZ Bank application introduced in Chapter 9 that is decomposed from the theme level down to the user story level.

TABLE 10-1 Decomposing a Requirement

Requirement Level	Requirement
Theme	See bank account data on a mobile device.
Features	See account balances. See a list of recent withdrawals or purchases. See a list of recent deposits. See my upcoming automatic bill payments. See my account alerts.
Epic user stories — decomposed from "see account balances"	See checking account balance. See savings account balance. See loan balance. See investment account balance. See retirement account balance.
User stories — decomposed from "see checking account balance"	See a list of my accounts once securely logged in. Select and view my checking account. See account balance changes after withdrawals. See account balance changes after purchases. See day's end account balance. See available account balance. Change account view.

USER STORIES AND THE INVEST APPROACH

You may be asking, just how decomposed does a user story have to be? Bill Wake, in his blog at XP123.com, describes the INVEST approach to ensure quality in user stories. We like his method so much we include it here.

Using the INVEST approach, user stories should be

- **Independent:** To the extent possible, a user story should need no other user stories to implement the feature that the story describes.
- **Negotiable:** Not overly detailed. The user story has room for discussion and an expansion of details.
- **Valuable:** The user story demonstrates product value to the customer. It describes features, not technical tasks to implement it. The user story is in the user's language and is easy to explain. The people using the product or system can understand the user story.
- **Estimable:** The story is descriptive, accurate, and concise, so the developers can generally estimate the work necessary to create the functionality in the user story.
- **Small:** It is easier to plan and accurately estimate small user stories. A good rule of thumb is that the development team can complete 6-10 user stories in a sprint.
- **Testable:** You can easily validate the user story, and the results are definitive.

Estimation poker

As you refine your requirements, you need to refine your estimates of the work required to complete your user stories as well. It's time to have some fun!

One of the most popular ways of estimating user stories is by playing *estimation poker*, sometimes called *planning poker*, a game to determine user story size and to build consensus among the development team members.

REMEMBER

The scrum master can help coordinate estimation, and the product owner can provide information about features, but the development team is responsible for estimating the level of effort required for the user stories. After all, the development team has to do the work to create the features that these stories describe.

To play estimation poker, you need a deck of cards like the one in Figure 10-4. You can get a digital version online at our website (`www.platinumedge.com/estimationpoker`), or you can make your own with index cards and markers. The

numbers on the cards are from the Fibonacci sequence, which follows this progression:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, and so on

If we start with the numbers 1 and 2, each subsequent number in the Fibonacci sequence is derived by taking the sum of the previous two numbers.

FIGURE 10-4: A deck of estimation poker cards.

Each user story receives an estimate relative to other user stories. For instance, a user story that is a 5 requires more effort than a 3, a 2, and a 1. It is about 5 times as much effort as a 1, more than double the effort of a 2, and roughly the amount of effort as a 3 and a 2 combined. It is not as much effort as an 8, but is just over half the effort of an 8.

As user stories and epic user stories increase in size, the difference between Fibonacci numbers gets bigger. Acknowledging these increasing gaps in precision for larger requirements is why the Fibonacci sequence works so well for relative estimation.

To play estimation poker, follow these steps:

1. **Provide each member of the development team with a deck of estimation poker cards.**
2. **From the list of user stories presented by the product owner, the team agrees on one user story that would be a 5.**

   This user story is the *anchor user story* for the team. The scrum master helps the development team reach consensus by using a fist of five or thumbs up/thumbs down (as described in Chapter 7), with discussion until everyone agrees on a user story that represents an estimate of 5.
3. **The product owner reads a high-priority user story to the players.**

4. **Each player selects a card representing his or her estimate of the effort involved in the user story and lays the card facedown on the table.**

   You don't want the players to see each other's cards until all cards have been played. This limits how much one player can influence others to vote a certain way. The players should compare the user story to other user stories they have estimated. (The first time through, the players compare the user story to only the anchor story.)

5. **All players turn over their cards simultaneously.**

6. **If the players have different story points:**

   a. *It's time for discussion.*

      Discussion is the value-add of estimation poker, which enables team alignment and consensus. The players with the highest and lowest scores talk about their assumptions and why they think the estimate for the user story should be higher or lower, respectively. The players compare the effort for the user story against the anchor story. The product owner provides more clarification about the story, as necessary.

   b. *Once everyone agrees on assumptions and has any necessary clarifications, the players reevaluate their estimates and place their new selected cards on the table.*

   c. *If the story points are different, the players repeat the process, usually up to three times.*

   d. *If the players can't agree on the estimated effort, the scrum master helps the development team determine a score that all the players can support (he or she may use a fist of five or thumbs up/thumbs down, as described in Chapter 7) or determine that the user story requires more detail or needs to be further broken down.*

7. **The players repeat Steps 3 through 6 for each user story.**

REMEMBER

Consider each part of the definition of *done* — developed, integrated, tested (including test automation), and documented — when you create estimates.

You can play estimation poker at any point — but definitely play during the product roadmap development and as you progressively break down user stories for inclusion in releases and sprints. With practice, the development team will get into a planning rhythm and become more adept at quickly estimating.

TIP

On average, development teams may spend about 10 percent of their time each sprint decomposing and refining product backlog items, including estimating and re-estimating. Make your estimation poker games fun! Bring in snacks, take breaks as needed, use humor, and keep the mood light.

Affinity estimating

Estimation poker can be effective, but what if you have many user stories? Playing estimation poker for, say, 500 user stories could take a long time. You need a way to estimate your entire product roadmap, but one that allows you to focus on only the user stories you must discuss to gain consensus.

When you have a large number of user stories, many of them are probably similar and would require a similar amount of effort to complete. One way to determine the right stories for discussion is to use affinity estimating. In *affinity estimating*, you quickly categorize your user stories and then apply estimates to these categories of stories.

TIP

When estimating by affinity, write your user stories on index cards or sticky notes. These types of user story cards work well when quickly categorizing stories.

Affinity estimating can be a fast and furious activity — the development team may choose to have the scrum master help facilitate affinity estimating sessions. To estimate by affinity, follow these steps:

1. **Taking no more than 60 seconds for each category, the development team agrees on a single user story in each of the following categories:**
   - Extra-small user story
   - Small user story
   - Medium user story
   - Large user story
   - Extra-large user story
   - Epic user story that is too large to come into the sprint
   - Needs clarification before estimating
2. **Taking no more than 60 seconds per user story, the development team puts all remaining stories into the categories listed in Step 1.**

   If you're using index cards or sticky notes for your user stories, you can physically place those cards into categories on a table or a whiteboard, respectively. If you divide the user stories among the development team members, having each development team member categorize a group of stories, this step can go quickly!

3. **Taking another 30 minutes, maximum, for each 100 user stories, the development team reviews and adjusts the placement of the user stories.**

   The entire development team must agree on the placement of the user stories into size categories.

4. **The product owner reviews the categorization.**

5. **When the product owner's expected estimate and the team's actual estimate differ by more than one story size, they discuss that user story.**

   The development team may or may not decide to adjust the story size.

   Note that after the product owner and the development team discuss clarifications, the development team has the final say on the user story size.

REMEMBER

6. **The development team plays estimation poker on the user stories in both the epic and the needs clarification categories.**

   The number of user stories in these categories should be minimal.

User stories in the same size category will have the same user story score. You can play a round of estimation poker to double-check a few, but you won't need to spend time in unnecessary discussion for every user story.

Story sizes are like T-shirt sizes and should correspond to Fibonacci scale numbers, as shown in Figure 10-5.

SIZE	POINTS
Extra small (XS)	1
Small (S)	2
Medium (M)	3
Large (L)	5
Extra large (XL)	8

FIGURE 10-5: Story sizes as T-shirt sizes and their Fibonacci numbers.

TIP

You can use the estimating and prioritizing techniques in this chapter for requirements at any level, from themes and features down to single user stories.

That's it. In a few hours, your entire product backlog was estimated. In addition, your scrum team has a shared understanding of what the requirements mean, having discussed them face-to-face rather than relying on interpretations of extensive documentation.

Release Planning

A *release* is a group of usable product features that you deploy to the market. A release does not need to include all the functionality outlined in the product roadmap but should include at least the *minimal marketable features*, the smallest group of product features that you can effectively deploy and promote in the marketplace. Your early releases will include highest priority (high value, or high risk, or both) items and exclude lower-priority requirements you identified during the product roadmap stage.

When planning a release, you establish the next set of minimal marketable features and identify an imminent product launch date around which the team can mobilize. As when creating the vision statement and the product roadmap, the product owner is responsible for creating the release goal and establishing the release date. However, the development team's estimates, with the scrum master's facilitation, contribute to the process.

Release planning is stage 3 in the Roadmap to Value (refer to Chapter 9 to see the roadmap as a whole). Figure 10-6 shows how release planning fits into product development.

FIGURE 10-6: Release planning as part of the Roadmap to Value.

Release planning involves completing two key activities:

- **Revising the product backlog:** In Chapter 9, we tell you that the product backlog is a comprehensive list of all the user stories you currently know for your product, whether or not they belong in the current release. Keep in mind that your list of user stories will probably change throughout development.
- **Creating the release plan:** This activity consists of defining the release goal, release target date, and prioritization of product backlog items that support the release goal. Whereas the product vision provides the long-range goal of the product, the release plan provides a midrange goal that the team can accomplish.

WARNING

Don't create a new, separate backlog during release planning. The task is unnecessary and reduces the product owner's flexibility. Prioritizing the existing product backlog based on the release goal is sufficient and enables the product owner to have the latest information when he or she commits to the scope during sprint planning.

The product backlog and release plan are some of the most important information radiators between the product owner and the development team. (See Chapter 11 for more on information radiators.) In Chapter 9, you find out how to complete a product backlog. How to create a release plan is described next.

The release plan contains a release schedule for a specific set of features. The product owner creates a release plan at the start of each release. To create a release plan, follow these steps:

1. **Establish the release goal.**

   The release goal is an overall business goal for the product features in your release. The product owner and development team collaborate to create a release goal based on business priorities and the development team's development speed and capabilities.

2. **Identify a target release date.**

   Some scrum teams determine release dates based on the completion of functionality; others may have hard dates, such as March 31 or September 1. The first case has a fixed scope and flexible date; the second case has a fixed date and a flexible scope.

WARNING

If a fixed date and fixed scope are determined for the release, adjustments may need to be made to the number of teams to accomplish the release goal according to schedule. The development team, not the product owner, estimates the effort required to implement product backlog items. Imposing a fixed scope and timeline without adjusting quality or resources (in this case, talent resources) will not be successful.

3. **Review the product backlog and the product roadmap to determine the highest-priority user stories that support your release goal (the minimum marketable features).**

   These user stories will make up your first release.

TIP

We like to achieve a release goal with about 80 percent of the user stories, using the final 20 percent to add robust features that will meet the release goal while adding to the product's "wow" factor. This approach provides appropriate flexibility and slack for the scrum team to deliver value without having to complete every single task.

4. **Refine the user stories in your release goal.**

   During release planning, dependencies, gaps, or new details are often identified that affect estimates and prioritization. This is the time to make sure the portion of the product backlog supporting your release goal is sized. (Refer to Figure 10-3.) We like to make sure that items supporting the current release goal have been decomposed and are sized appropriately for the release. The development team helps the product owner by updating estimates for any added or revised user stories, and commits to the release goal with the product owner.

TIP

   Release planning is the initial opportunity to identify and break down dependencies before they become impediments. Dependencies are anti-patterns to becoming agile. Teams should work to become highly aligned and highly autonomous. Dependencies are an indication that your team does not have the capability to do whatever it is dependent on.

5. **Estimate the number of sprints needed, based on the scrum team's velocity.**

TECHNICAL STUFF

   Scrum teams use velocity as an input to plan how much work they can take on in a release and sprint. *Velocity* is the sum of all user story points completed within a sprint. So, if a scrum team completed six user stories during its first sprint with sizes 8, 5, 5, 3, 2, 1, its velocity for the first sprint is 24. The scrum team would plan its second sprint keeping in mind that it completed 24 story points during the first sprint.

After multiple sprints, scrum teams can use their running average velocity as an input to determine how much work they can take on in a sprint, as well as to extrapolate their release schedule by dividing the total number of story points in the release by their average velocity. You learn more about velocity in Chapter 15.

Be aware that some teams add a *release sprint* to a release to conduct activities unrelated to product development but necessary to release the product to customers. If you need a release sprint, be sure to factor that into the date you choose.

REMEMBER

Delaying crucial development tasks such as testing until the end of development rearloads risk. Agile techniques frontload risk to avoid the surprises and defects that result from delayed testing. If a scrum team requires a release sprint, it probably means the broader organization can't support being truly shippable each sprint, which is an impediment to becoming agile. The goal for scrum teams is to have every type of work or activity required to release functionality to the market as part of the sprint-level definition of done. Scrum masters should work together to remove organizational impediments preventing teams from being able to release at scale according to their sprint-level definition of done.

In some traditional or project-focused organizations, some tasks, such as security testing or load testing a software product, can't be completed within a sprint because the task's environment takes time to set up and request. Although release sprints allow scrum teams to plan for these types of activities, doing so is an anti-pattern, or the opposite of being agile. In these examples, the scrum master would work with the organizational leaders who manage the security or load testing environments to find ways to enable scrum teams to accomplish security or lead testing during the sprint.

Each planned release shifts from what was a tentative plan (high-level product roadmap items) to a more concrete goal ready to execute in the sprint(s) of the release. Figure 10-7 represents a typical release plan.

FIGURE 10-7: Sample release plan.

TIP

Bear in mind the pen-pencil rule: You can commit to (write in pen) the plan for the first release, but anything beyond the first release is tentative (written in pencil). In other words, use just-in-time planning (see Chapter 7) for each release. After all, things change, so why bother getting microscopic too early?

Preparing for Release

In release planning, you also need to prepare your organization for the product release. The next section discusses how to prepare for supporting the new functionality in the marketplace and how to get stakeholders in your company or organization ready for product deployment.

Preparing the product for deployment

In each sprint, a valuable working product increment is created in support of the release goal. With a definition of done that means each sprint's increment is shippable, there's not much more you need to do to prepare the product for technical deployment. Every sprint, you're ready to release if there is enough value accumulated for the customer.

TECHNICAL STUFF

With software product development, releasing to production is done through continuous integration (CI) and continuous deployment (CD), which are extreme programming (XP) practices used with software product development. (You can read more about CI in Chapter 17.) The product code is checked in and moved to quality assurance (QA) and then to the production (live) environments as quickly and seamlessly as possible. Advancements in technology enable teams to automate a pipeline for building, integrating, testing, and fixing without delay. Combining a CI/CD pipeline with a robust automated testing harness raises the bar of your product development agility. Teams developing non-software products should use techniques that automate testing and integration of new functionality to the existing product as much as possible.

TECHNICAL STUFF

In information technology (IT), usually involving software development, *development operations (DevOps)* is the integration of software development and IT operations (which includes functions such as systems administration and server maintenance). Taking a DevOps approach enables everyone involved (user experience, testing, infrastructure, database, coding, design) to work together, to eliminate handoffs, and to streamline collaboration for reduced deployment cycle times. Multiple teams working on the same product will not be successful without a reliable CI/CD pipeline.

TIP

Not all agile product development efforts use release planning. Some scrum teams release functionality for customer use with every sprint or even every day. The development team, product, organization, customers, stakeholders, and product's technological complexity can all help determine your approach to product releases. Helpful for this discussion are Principles 1 and 3, respectively: "Satisfy the customer through early and continuous delivery" and "Deliver. . .frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale." (For details, see Chapter 2.)

Prepare for operational support

After your product is released to the customer, someone will have to support it, a responsibility that involves responding to customer inquiries, maintaining the system in a production environment, and enhancing existing functionality to fill minor gaps. Although new development work and operational support work are both important, they involve different approaches and cadences.

Separating new development and support work ensures that new development teams can focus on continuing to bring innovative solutions to customers at a faster rate than if the team frequently switches between the two types of work.

We recommend a model that separates new development and maintenance work, as illustrated in Figure 10-8.

FIGURE 10-8: Operational support scrum team model.

For a scrum team of nine developers, for instance, we would divide the development team into two teams, one with six developers, and the other with three. (These numbers are flexible.) The team of six does new development work from the product backlog in one-week to two-week sprints, as described in Chapters 9 through 12. The work that the team commits to during the sprint planning meeting will be the only work it does.

The members of the three-person team are our firefighters and do maintenance and support work in one-day sprints or by using kanban. (You learn about one-day sprints in Chapter 11 and kanban in Chapter 5.) Single-day sprints allow the scrum team to triage all incoming requests from the previous day, plan the highest-priority items, implement those items as a team, and review the results at the end of the day (or even earlier) for go or no-go approval before pushing the changes to production. For continuity, the product owner and scrum master are the same for each team.

REMEMBER

Although the newly modified product development team is smaller than before, there are still enough developers to keep new development efforts moving forward, uninterrupted by maintenance work. By the time you begin releasing functionality to the market, your scrum team will be working well together and the developers will have increased their versatility by being able to complete more types of tasks than when the project first started.

Teams should rotate team members between the two activities at sprint boundaries (every three to five sprints, for example) to give everyone an opportunity to learn from both types of work. If support is excessive, the product owner may want to reevaluate the product backlog to see if there are ways to reduce the weight of support and its distraction. Support distractions cause the team to focus on tactical resolutions rather than strategic value creation. Team stability and acceleration will improve.

When preparing for release, establishing expectations up front regarding how the functionality will be supported in production allows the scrum team to develop the product in a way that enables the team to effectively support the product after it's deployed. Establishing expectations also increases ownership across the scrum team and heightens the team's awareness and dedication to long-term success.

Product owners who maintain a strong working relationship with help desks or those providing customer support benefit greatly by understanding how their products are used by real users. Help desk reporting can be valuable for evaluating product backlog candidates or upcoming priorities. Help desks benefit by knowing that the scrum team is working to address any escalated incidents. The product owner involves these groups in release planning to ensure that all are prepared for operational support well in advance of release.

Preparing the organization

A product release often affects a number of departments in a company or an organization. To get the organization ready for the new functionality to be released next, the product owner coordinates with the rest of the organization regarding what to expect and what will be needed from them during release planning. When product owners do this effectively, there shouldn't be surprises at release time.

Release planning addresses not only the activities for the development team to release but also the activities to be performed by the rest of the organization. These might include the following:

- **Marketing:** Do marketing campaigns related to the new product need to launch at the same time as the product?
- **Sales:** Do specific customers need to know about the product? Will the new product cause an increase in sales?
- **Logistics:** Is the product a physical item that includes packaging or shipping?

- **Product support:** Does the customer service group have the information it needs to answer questions about the new product? Will this group have enough people on hand in case customer questions increase when the product launches?
- **Legal:** Does the product meet legal standards, including pricing, licensing, and correct verbiage, for release to the public?

The departments that need to be ready for the release and the specific tasks these groups need to complete will vary from organization to organization. A key to release success, however, is that the product owner and scrum master involve the right people and ensure that those people clearly understand what they need to do to be ready for the functionality release.

During release planning, you also need to include one more group: the customer. The next section discusses getting the marketplace ready for your product.

Preparing the marketplace

The product owner is responsible for working with other departments to ensure that the marketplace — existing customers and potential customers — is ready for what's coming. The marketing or sales teams may lead this effort; team members look to the product owner to keep them informed as to the release date and the features that will be part of the release.

REMEMBER

Some software products are for only internal employee use. Certain things you're reading in this section might seem like overkill for an internal application — that is, an application released only within your company. However, many of these steps are still good guidelines for promoting internal applications. Preparing customers, whether internal or external, for new products is a key part of product success.

To help prepare customers for the product release, the product owner may want to work with different teams to ensure the following:

- **Marketing support:** Whether you're dealing with a new product or new features for an existing product, the marketing department should leverage the excitement of the new product functionality to help promote the product and the organization.
- **Customer testing:** If possible, work with your customers to get real-world feedback about the product from a subset of end users. (Some people use focus groups.) Your marketing team can also use this feedback to translate into testimonials for promoting the product right away.

- **Marketing materials:** An organization's marketing group also prepares the promotional and advertising plans, as well as packaging for physical media. Media materials, such as press releases and information for analysts, need to be ready, as do marketing and sales materials.
- **Support channels:** Ensure that customers understand the available support channels in case they have questions about the product.

Review the items on your release backlog from the customer's standpoint. Think of the personas you used when creating your user stories. Do those personas need to know something about the product? Update your launch checklist with items that would be valuable to customers represented by your personas. You can find more information about personas in Chapter 4.

Finally, you're there — release day. Whatever role you played along the way, this is the day you worked hard to achieve. It's time to celebrate!

Sprint Planning

With agile product development, a *sprint* is a consistent iteration of time in which the development team creates a specific group of product capabilities from start to finish. At the end of each sprint, the functionality that the development team has created should be working, ready to demonstrate, and potentially shippable to the customer.

Sprints should be the same length. Keeping the sprint lengths consistent helps the development team measure its performance and plan better at each new sprint.

Sprints generally last one to four weeks. One month is the longest amount of time any sprint should last; longer iterations make changes riskier, defeating the purpose of being agile. We rarely see sprints lasting longer than two weeks, and more often see sprints lasting a week. One-week sprints are a natural cycle with the Monday-to-Friday business week, which structurally prevents weekend work. When priorities change on a daily basis, some scrum teams work in one-day sprints, as discussed in Chapter 11.

Market and customer needs are changing more and more quickly, and the amount of time you can afford between opportunities to gather customer feedback only gets shorter. Our rule of thumb is that your sprint shouldn't be longer than your stakeholders can consistently go without changes in priority regarding what the scrum team should be working on in the sprint. Sprint duration is a function of the business's need for change.

Each sprint includes the following:

- Sprint planning at the beginning of the sprint
- Daily scrum meetings
- Development work — the bulk of the sprint
- A sprint review and a sprint retrospective at the end of the sprint

Discover more about daily scrums, development work, sprint reviews, and sprint retrospectives in Chapters 11 and 12.

Sprint planning is stage 4 in the Roadmap to Value, as you can see in Figure 10-9. The entire scrum team — the product owner, scrum master, and development team — works together to plan sprints.

Stage 4: SPRINT PLANNING

Description: Establishment of specific iteration goals and tasks

Owner: Product owner and development team

Frequency: At the start of each sprint

FIGURE 10-9: Sprint planning as part of the Roadmap to Value.

The sprint backlog

The *sprint backlog* is a list of user stories associated with the current sprint and related tasks. When planning your sprint, you do the following:

- Establish the goal for your sprint.
- Choose the product backlog items (user stories) that support your goal.
- Break user stories into specific development tasks.
- Create a sprint backlog. The *sprint backlog* consists of the following:
  - The list of user stories in the sprint in order of priority.
  - The relative effort estimate for each user story.
  - The tasks necessary to develop each user story.

- The effort, in hours, to complete each task (if needed). At the task level, if you estimate the number of hours each task will take to complete, use hours instead of using story points. Because your sprint has a specific short length, and thus a known number of available working hours, you can use the time each task takes to determine whether the tasks will fit into the team's capacity of the sprint. Each task should take one day or less for the development team to complete.

  TIP

  Some mature development teams may not need to estimate their tasks as they get more consistent at breaking down their user stories into executable tasks. Estimating tasks is helpful for newer development teams to ensure that they understand their capacity and plan each sprint appropriately.

- A burndown chart, which shows the status of the work in progress for the sprint.

TECHNICAL STUFF

Tasks should take a day or less to complete for two reasons. The first reason involves basic psychology: People are motivated to get to the finish line. If you have a task that you know you can complete quickly, you're more likely to finish it on time, just to check it off your to-do list. The second reason is that one-day tasks provide good red flags that development targets might be veering off course. If a development team member reports that he or she is working on the same task for more than one or two days, that team member probably has a roadblock and the scrum master should investigate what might be keeping the team member from finishing work. (For more on managing roadblocks, see Chapter 11.)

The development team collaborates to create and maintain the sprint backlog, but only the development team can modify the sprint backlog. The sprint backlog should reflect an up-to-the-day snapshot of the sprint's progress. Figure 10-10 shows a sample sprint backlog at the end of the sprint planning meeting. You can use this example, find other samples, or even use a whiteboard.

The sprint planning meeting

On the first day of each sprint, often a Monday morning, the scrum team holds the sprint planning meeting.

TIP

For a successful sprint planning meeting, make sure everyone involved in the session (the product owner, the development team, the scrum master, and anyone else the scrum team requests) is dedicated to the effort for the entire meeting.

MyXYZ Mobile Banking App – Sprint 1

Sprint Dates: February 4 – February 8

Sprint Goal

Demonstrate the ability for personal banking users to view account balances and pending translations.

Available working hours in the sprint					Days in sprint : 4.5	
	M	Tu	W	Th	F	Total
Developers	**4 (Start w/ sprint planning)**	**5**	**6**	**7**	**8 (End w/ sprint review + retrospective)**	
Suraj	4	6	6	6	4	26
Nancy	4	6	6	6	4	26
Kavita	4	6	6	6	4	26
Liam	4	6	6	6	4	26
Paul	4	6	6	6	4	26
					Total sprint hours:	130
					Total per day:	29

Feature Burndown – Based on estimated hours remaining

ID	Task	Story Points	Responsible	M 4 (Start w/ sprint planning)	Tu 5	W 6	Th 7	F 8 (End w/ sprint review + retrospective)	Done (Y)	Accepted (Y/N)
125	**View account balance**	**8**	**Developer name**	**8**	**8**	**8**	**8**	**8**		**Y**
	Write automated unit test and develop API		Suraj	6	6	6	6	6		
	Implement UI		Nancy	3	3	3	3	3		
	Write automated functional test		Kavita	3	3	3	3	3		
	Write automated integration test		Paul	4	4	4	4	4		
	Write automated regression test		Liam	2	2	2	2	2		
	Conduct peer review			1	1	1	1	1		
	Update wiki			1	1	1	1	1		
	Promote to QA environment			1	1	1	1	1		
0059	**View pending transactions**	**5**	**Developer name**	**5**	**5**	**5**	**5**	**5**		
	Write automated unit test			5	5	5	5	5		
	Implement UI			2	2	2	2	2		
	Write automated functional test			4	4	4	4	4		
	Write automated integration test			5	5	5	5	5		
	Write automated regression test			2	2	2	2	2		
	Conduct peer review			3	3	3	3	3		
	Update wiki			3	3	3	3	3		
	Promote to QA environment			3	3	3	3	3		

FIGURE 10-10: Sprint backlog example.

Base the length of your sprint planning meeting on the length of your sprints. Sprint planning should take no longer than two hours per week of the sprint, or no longer than a full day for a one-month sprint. This timebox helps ensure that the meeting stays focused and on track. Figure 10-11 is a good quick reference for your sprint planning meeting lengths.

If my sprint is this long…	My sprint planning meeting should last no longer than…
One week	Two hours
Two weeks	Four hours
Three weeks	Six hours
Four weeks	Eight hours

FIGURE 10-11: Ratio of sprint planning meeting to sprint length.

TECHNICAL STUFF

With agile product development, the practice of limiting the time of your meetings is sometimes called *timeboxing.* Keeping your meetings timeboxed provides focus and ensures that the development team has the time it needs to create the product.

You'll split your sprint planning meetings into two parts: one to set a sprint goal (the "why") and choose user stories for the sprint (the "what"), and another to break down your user stories into individual tasks (the "how" and "how much"). The details on each part are discussed next.

Part 1: Setting goals and choosing user stories

In the first part of your sprint planning meeting, the product owner and development team, with support from the scrum master, decide what to do during the sprint by doing the following:

1. **Discuss and set a sprint goal.**
2. **Select the user stories from the product backlog that support the sprint goal, further refine them for understanding, and revisit their relative estimates.**
3. **If needed, create user stories to fill gaps to achieve the sprint goal.**
4. **Determine what the team can commit to in the current sprint.**

TIP

Consistently refining the product backlog ensures that the scrum team plans items into each sprint with which the team is already familiar. This refinement is also critical for ensuring that sprint planning stays within the timebox and results in a clear plan to deliver potentially shippable functionality at the end of each sprint. Scrum teams, on average, spend about 10 percent of their sprint in product backlog refinement for future sprints.

At the beginning of your sprint planning meeting, the product owner should propose a sprint goal that identifies a problem to solve for the customer and then together with the development team discuss and agree on the sprint goal. The sprint goal should be an overall description of the working customer functionality that the team will demonstrate and possibly release at the end of the sprint. The goal is supported by the highest-priority user stories in the product backlog. A sample sprint goal for the mobile banking application (refer to Chapter 9) might be as follows:

> Demonstrate the ability of a mobile banking customer to log in and view account balances and pending and prior transactions.

Using the sprint goal, you determine the user stories that belong in the sprint and refine the estimates for those user stories, as needed. For the mobile banking

application sprint goal example, the group of user stories for the sprint might include the following:

- Log in and access my accounts.
- View account balances.
- View pending transactions.
- View prior transactions.

All these would be high-priority user stories in the product backlog that support the sprint goal.

REMEMBER

Don't forget to bring at least one improvement item agreed to during a previous sprint retrospective.

The second part of reviewing user stories is confirming that the effort estimates for each user story have been reviewed and adjusted, if needed, and reflect the development team's current knowledge of the user story. Adjust the estimate if necessary. With the product owner in the meeting, resolve any outstanding questions. At the beginning of the sprint, the scrum team has the most up-to-date knowledge about the system and the customer's needs, so make sure the development team and product owner have one more chance to clarify and size the user stories going into the sprint.

Finally, after you know which user stories support the sprint goal, the development team should agree and confirm that it can complete the goal planned for the sprint. If any of the user stories you discussed earlier don't fit in the current sprint, remove them from the sprint and add them back into the product backlog.

WARNING

Always plan and work on one sprint at a time. An easy trap to fall into is to place user stories into specific future sprints. For example, when you're still planning sprint 1, don't decide that user story X should go into sprint 2 or 3. Instead, keep the ordered list of user stories up to date in the product backlog and focus on always developing the next highest-priority stories. Commit to planning only for the current sprint. What you learn in sprint 1 may fundamentally change how you go about sprint 2 or 10 or 100.

After you have a sprint goal, user stories for the sprint, and a commitment to the goal, move on to the second part of sprint planning.

TIP

Because a sprint planning meeting for sprints longer than one week might last a few hours, you might want to take a break between the two parts of the meeting.

Part 2: Breaking down user stories into tasks for the sprint backlog

In the second part of the sprint planning meeting, the scrum team does the following:

1. **The development team creates the sprint backlog tasks associated with each user story. Make sure that tasks encompass each part of the definition of done: developed, integrated, tested (including test automation), and documented.**
2. **The development team double-checks that it can complete the tasks in the time available in the sprint.**
3. **Each development team member should choose his or her first task to accomplish before leaving the meeting.**

TIP

Development team members should each work on only one task on one user story at a time to enable *swarming* — the practice of the entire development team working on one user story until completion. Swarming can be an efficient way to complete work in a short amount of time. In this way, scrum teams avoid getting to the end of the sprint with all user stories started but few finished.

At the beginning of part two of the meeting, break the user stories into individual tasks and allocate a number of hours to each task. The development team's target should be completing a task in a day or less. For example, a user story for the XYZ Bank mobile application might be as follows:

> Log in and access my accounts.

The team decomposes this user story into tasks, such as the following:

- Write the unit test.
- Write the user acceptance test.
- Create an authentication screen for a username and password, with a Submit button.
- Create an error screen for the user to reenter credentials.
- Create a screen (once logged in) displaying a list of accounts.
- Using authentication code from the online banking application, rewrite code for an iPhone/iPad/Android application. (This task could potentially be three different tasks.)
- Create calls to the database to verify the username and password.

- Re-factor code for mobile devices.
- Write the integration test.
- Promote the product increment to QA.
- Update the regression test automation suite.
- Run the security test.
- Update the wiki documentation.

After you know the number of hours that each task will take, do a final check to make sure that the number of hours available to the development team reasonably matches the total of the tasks' estimates. If the tasks exceed the hours available, one or more user stories will have to come out of the sprint. Discuss with the product owner what tasks or user stories are the best to remove.

If extra time is available within the sprint, the development team might be able to include another user story. Just be careful about over-committing at the beginning of a sprint, especially during the first few sprints.

After you know which tasks will be part of the sprint, choose what you will work on first. Each development team member should select his or her initial task to accomplish for the sprint. Team members should focus on one task at a time.

TIP

As the development team members think about what they can complete in a sprint, use the following guidelines to ensure that they don't take on more work than they can handle while they're learning new roles and techniques:

- **Sprint 1:** 25 percent of what the development team thinks it can accomplish. Include overhead for learning the new process and starting product development.
- **Sprint 2:** Assuming the scrum team was able to complete sprint 1 successfully, 50 percent of what the development team thinks it can accomplish.
- **Sprint 3:** Assuming success in sprint 2, 75 percent of what the development team thinks it can accomplish.
- **Sprint 4 and forward:** Assuming success in sprint 3, 90 percent. The development team will have developed a rhythm and velocity, gained insight into agile principles and the product, and will be working at close to full pace.

TIP

Avoid planning 100 percent of capacity for a sprint. Scrum teams should build in slack in their sprint to account for unknowns that inevitably come up. Instead of padding estimates, simply be wise and don't commit every available hour, assuming everything will go as planned. Teams that finish early accelerate faster.

The scrum team should constantly evaluate the sprint backlog against the development team's progress on the tasks. At the end of the sprint, the scrum team can also assess estimation skills and capacity for work during the sprint retrospective (see Chapter 12). This evaluation is especially important for the first sprint.

TIP

For the sprint, how many total working hours are available? In a one-week sprint or a 40-hour week, you could wisely assume that 4.5 working days are available to develop user stories. Why 4.5 days? About one-fourth of day one is taken up with sprint planning, and about one-fourth of day five is taken up with the sprint review (when the stakeholders review the completed work) and the sprint retrospective (when the scrum team identifies team improvements for future sprints). That leaves 4.5 days of development. If you assume each full-time team member has 30 hours per week (6 productive hours per day) to focus on the sprint goal, the number of working hours available is

$$\text{Number of team members} \times 6 \text{ hours} \times 4.5 \text{ days}$$

After sprint planning is finished, the development team can immediately start working on the tasks to create the product!

The scrum master should make sure the product vision and roadmap, product backlog, definition of done, and sprint backlog are in a prominent place and accessible to everyone during sprint planning as well as in the area in which they work. In this way, stakeholders can view the product information and progress on demand without interrupting the development team. For details, see Chapter 11.

IN THIS CHAPTER

- Planning each day
- Tracking daily progress
- Developing and testing every day
- Ending the day

Chapter 11
Working throughout the Day

It's Tuesday, 9 a.m. Yesterday, you completed sprint planning, and the development team started work. For the rest of the sprint, you'll be working *cyclically*, where each day follows the same pattern.

In this chapter, you find out how to use agile principles daily throughout each sprint. You see the work that you will do every day as part of a scrum team: planning and coordinating your day, tracking progress, creating and verifying usable functionality, inspecting and adapting, and identifying and dealing with impediments to your work. You see how the different scrum team members work together each day during the sprint to ensure transparency as they help create the product.

Planning Your Day: The Daily Scrum

With agile product development, you make plans throughout the entire development effort — and on a daily basis. Agile development teams start each workday with a *daily scrum* meeting to evaluate their progress and adapt their plan for the day based on what was accomplished previously. They identify and coordinate the resolution of impediments (roadblocks requiring scrum master involvement),

note completed items, and synchronize and plan what each team member will do during the day to achieve the sprint goal.

The daily scrum is stage 5 on the Roadmap to Value. You can see how the sprint and the daily scrum fit into product development in Figure 11-1. Note how they both repeat.

FIGURE 11-1: The sprint and the daily scrum in the Roadmap to Value.

In the daily scrum meeting, each development team member addresses the following four topics, which facilitate team coordination:

- **What was completed yesterday** to help accomplish the sprint goal?

  Avoid using the daily scrum as a status report by having each developer give an accounting of what they did the day before or by moving completed items around on the task board. Developers should update their task as soon as they complete it, or at the very least at the end of the day, so that when the scrum team comes together for its daily scrum the next day, the status is already reflected. In other words, don't spend time on what was accomplished yesterday unless it effects how to go about the work to be done today.

- **What will be done today** to help accomplish the sprint goal?
- **What impediments** are in the way of accomplishing the sprint goal?
- **This is how I feel.** (We added the fourth question to help the scrum master better understand team health daily rather than once per sprint.)

WARNING

TECHNICAL STUFF

Other names you might hear for the daily scrum meeting are the *daily huddle* or the *daily standup* meeting. Daily scrum, daily huddle, and daily standup all refer to the same thing. Daily scrum is how scrum refers to it.

We also have the scrum master address these three statements regarding the team's impediments:

- Impediments resolved yesterday
- Impediments that need to be resolved today (and order of priority)
- Impediments that need to be escalated

What does the product owner do during the daily scrum? Listen. The product owner listens to see if there is anything he or she can do to help the team accomplish its work more effectively. The product owner may provide clarification as needed, and might say something if he or she hears something that indicates that the development team is working on something outside the sprint goal. An engaged, decisive product owner makes life easier for a development team.

One of the rules of scrum is that the daily scrum meeting should last 15 minutes or less. Longer meetings eat into the development team's day. Standing encourages shorter meetings (which is why the meeting is referred to also as the daily standup). You can also use props to keep daily scrum meetings quick.

TIP

We start meetings by tossing an item, such as a squeaky burger-shaped dog toy — don't worry; it's clean — to a random development team member. Each person addresses the four topics and then passes the squeaky toy to someone else. If people are long-winded, we change the prop to a 500-page ream of copy paper, which weighs about five pounds. Each person can talk for as long as he or she can hold the ream out to one side. Either meetings will quickly become shorter, or development team members will quickly build up their arm strength — in our experience, it's the former.

To keep daily scrums brief and effective, the scrum team can follow several guidelines:

- **Anyone may attend a daily scrum, but only the development team, the scrum master, and the product owner may talk.** The daily scrum is the scrum team's opportunity to coordinate daily activities, not take on additional requirements or changes from stakeholders. Stakeholders can discuss questions with the scrum master or product owner afterward, but stakeholders should not distract the development team from the focus of the sprint.
- **The focus is on immediate priorities.** The scrum team should review only completed tasks, tasks to be done, and roadblocks.

- **Daily scrum meetings are for coordination, not problem-solving.** The development team and the scrum master are responsible for having relevant discussions of the tasks they're working on and removing roadblocks during the day.

  To keep meetings from drifting into problem-solving sessions, scrum teams can

  - Create a list on a whiteboard to keep track of issues that need immediate attention, and then address those issues directly after the meeting with only those team members who need to be involved.
  - Hold a meeting, called an *after-party,* to solve problems after the daily scrum is finished. Some scrum teams schedule time for an after-party every day; others meet only as needed.

- **The daily scrum is for peer-to-peer coordination.** It is not used for an individual to report status to one person, such as the scrum master or product owner. Status is reported at the end of each day in the sprint backlog, and should take developers about one minute.
- **Such a short meeting must start on time.** It's not unusual for the scrum team to have a working agreement ensuring that meetings start and end on time. Creative punishments for tardiness include doing pushups or adding penalty money into a team celebration fund. Whatever punishment is used, the scrum team agrees on it together; the method is not dictated to them by someone outside the team, such as a manager.
- **The scrum team may request that daily scrum attendees stand up — rather than sit down — during the meeting.** Standing up makes people eager to finish the meeting and get on with the day's work.

Daily scrum meetings are effective for keeping the development team focused on the right tasks for any given day. Because the development team members are accountable for their work in front of their peers, they are less likely to stray from their daily commitments. Daily scrum meetings also help ensure that the scrum master and development team can deal with roadblocks immediately. These meetings are so useful that even organizations that are not using any other agile techniques sometimes adopt daily scrums.

TIP

We like to hold daily scrum meetings 30 minutes after the development team's normal start time to allow for traffic jams, emails, coffee, and other rituals when starting the day. Having a later daily scrum meeting also allows the development team time to review defect reports from automated testing tools that were run the night or weekend before.

The daily scrum is for discussing progress and planning each upcoming day. As you see next, you also track progress — not just discuss it — every day.

Tracking Progress

You also need to track the progress of your sprint daily. This section discusses ways to keep track of the tasks in your sprint.

Two tools for tracking progress are the sprint backlog and a task board. Both the sprint backlog and the task board enable the scrum team to show the sprint's progress to anyone at any given time.

REMEMBER

The Agile Manifesto values individuals and interactions over processes and tools. Make sure your tools support, rather than hinder, your scrum team. Modify or even replace tools if needed. Read more about the Agile Manifesto in Chapter 2.

The sprint backlog

During sprint planning, you concentrate on adding user stories and tasks to the spring backlog. During the sprint itself, you update the sprint backlog daily, tracking progress of the development team's tasks for each working day. Figure 11-2 shows the sprint backlog for this book's sample application, the XYZ Bank's mobile banking application, as it would appear after day 4 of the first sprint. (Chapter 10 discusses details of the sprint backlog.)

MyXYZ Mobile Banking App – Sprint 1

Sprint Dates: February 4 – February 8

Sprint Goal

Demonstrate the ability for personal banking users to view account balances and pending translations.

Available working hours in the sprint — **Days in sprint : 4.5**

Developers	M 4 (Start w/ sprint planning)	Tu 5	W 6	Th 7	F 8 (End w/ sprint review + retrospective)	Total
Suraj	4	6	6	6	4	26
Nancy	4	6	6	6	4	26
Kavita	4	6	6	6	4	26
Liam	4	6	6	6	4	26
Paul	4	6	6	6	4	26
					Total sprint hours:	130
					Total per day:	29

Feature Burndown – Based on estimated hours remaining

ID	Task	Story Points	Responsible	M 4 (Start w/ sprint planning)	Tu 5	W 6	Th 7	F 8 (End w/ sprint review + retrospective)	Done (Y)	Accepted (Y/N)
125	**View account balance**	**8**	**Developer name**	**8**	**8**	**0**	**0**	**0**	**Y**	**Y**
	Write automated unit test and develop API		Suraj	6	0	0	0	0		
	Implement UI		Nancy	3	0	0	0	0		
	Write automated functional test		Kavita	3	0	0	0	0		
	Write automated integration test		Paul	4	0	0	0	0		
	Write automated regression test		Suraj	2	1	0	0	0		
	Conduct peer review		Nancy	1	1	0	0	0		
	Update wiki		Paul	1	1	0	0	0		
	Promote to QA environment		Nancy	1	1	0	0	0		
0059	**View pending transactions**	**5**	**Developer name**	**5**	**5**	**5**	**0**	**0**	**Y**	**Y**
	Write automated unit test		Kavita	5	5	0	0	0		
	Implement UI		Paul	2	2	0	0	0		
	Write automated functional test		Suraj	4	4	0	0	0		
	Write automated integration test		Nancy	5	5	0	0	0		
	Write automated regression test		Nancy	2	2	0	0	0		
	Conduct peer review		Suraj	3	3	0	0	0		
	Update wiki		Paul	3	3	0	0	0		
	Promote to QA environment		Kavita	3	3	1	0	0		

FIGURE 11-2: Sample sprint backlog.

Make the sprint backlog available to the entire team every day. That way, anyone who needs to know the sprint status can find it instantly.

Near the top left of Figure 11-2, note the *sprint burndown chart,* which shows the progress that the development team is making. You can see that the development team members have completed tasks close to the even burn rate of their available hours, and the product owner has accepted several user stories as complete.

You can include burndown charts on your sprint backlog and on your product backlog. (This chapter concentrates on the sprint backlog.) Figure 11-3 shows the burndown chart in detail.

FIGURE 11-3: A burndown chart.

A burndown chart is a powerful tool for visualizing progress and the work remaining. The chart shows the following:

- The outstanding work (in hours) on the first vertical axis
- Time, in days along the horizontal axis

Some sprint burndown charts, like the one in Figure 11-3, also show the outstanding story points on a second vertical axis that is plotted against the same horizontal time axis as hours of work remaining.

A burndown chart enables anyone to see the status of the sprint at a glance. Progress is clear. By comparing the realistic number of hours available to the work remaining, you can find out daily whether the effort is going as planned, is in better shape than expected, or is in trouble. This information helps you determine

whether the development team is likely to accomplish the sprint goal and helps you make informed decisions early in the sprint.

ON THE WEB

You can create a sprint backlog using a spreadsheet and charting program such as Microsoft Excel. You can also download our free sprint backlog template, which includes a burndown chart, from the book's website at `www.dummies.com/go/agileprojectmanagementfd3e`.

Figure 11-4 shows samples of burndown charts for sprints in different situations. Looking at these charts, you can tell how the work is progressing:

- **1. Expected:** This chart shows a normal sprint pattern. The remaining work hours rise and fall as the development team completes tasks, ferrets out details, and identifies tactical work it may not have initially considered. Although work occasionally increases, it is manageable, and the team mobilizes to complete all user stories by the end of the sprint.
- **2. More complicated:** In this sprint, the work increased beyond the point at which the development team felt it could accomplish everything. The team identified this issue early, worked with the product owner to remove some user stories, and still achieved the sprint goal. The key to scope changes within a sprint is that they are always initiated by the development team — no one else.
- **3. Less complicated:** In this sprint, the development team completed some critical user stories faster than anticipated and worked with the product owner to identify additional user stories it could add to the sprint.
- **4. Not participating:** A straight line in a burndown means that the team didn't update the burndown or made zero progress that day. Either case is a red flag for future problems.

  WARNING

  Just like on a heartbeat graph, a horizontal straight line on a sprint burndown chart is never a good thing.
- **5. Lying (or conforming):** This burndown pattern is common for a new agile development team that might be accustomed to reporting the hours that management expects, instead of the time the work really takes, and consequently tends to adjust the team's work estimates to the exact number of remaining hours. This pattern often reflects a fear-based environment, where the managers lead by intimidation.
- **6. Failing fast:** One of the strongest benefits of this simple visualization of progress is the immediate proof of progress or lack thereof. This pattern shows an example of a team that wasn't participating or progressing. Halfway through the sprint, the product owner decided to cut losses by killing the sprint and starting a new sprint with a new sprint goal. Only product owners can end a sprint early.

FIGURE 11-4: Profiles of burndown charts.

The sprint backlog helps you track progress throughout each sprint. You can also refer to earlier sprint backlogs to compare progress from sprint to sprint. You make changes to your process in each sprint (read more about the concept of inspect and adapt in Chapter 12). Constantly inspect your work and adapt to make it better. Hold on to those old sprint backlogs.

Another way to keep track of your sprint is by using a task board. Read on to find out how to create and use one.

The task board

Although the sprint backlog is a great way to track and show development progress, it's probably in an electronic format, so it might not be immediately accessible to anyone who wants to see it. Some scrum development teams use a task board along with their sprint backlog. A *task board* provides a quick, easy view of the items in the sprint that the development team is working on and has completed.

We like task boards because you can't deny the status they show. Like the product roadmap, the task board can be made up of sticky notes on a whiteboard. The task board will have at least the following four columns, from left to right:

- **To Do:** The user stories and tasks that remain to be accomplished are in the far left column.
- **In Progress:** User stories and tasks that the development team is currently working on are in the In Progress column. Only one user story should be in this column. Having more user stories in progress is an alert that

development team members are not working cross-functionally and, instead, are hoarding desired tasks (not swarming). You risk having multiple user stories partially done instead of more user stories completely done by the end of the sprint.

- **Accept:** After the development team completes a user story, it moves the story to the Accept column. User stories in the Accept column are ready for the product owner to review and either provide feedback or accept.
- **Done:** When the product owner has reviewed a user story and verifies that the user story is complete, the product owner can move that user story to the Done column.

TIP

Limit your work in progress! Only select one task at a time. Leave other tasks available in the To Do column. Ideally, a development team will work on only one user story at a time and swarm on the tasks of that user story to complete it quickly. High-performing teams and organizations do one item well before moving onto the next item.

Because the task board is tactile — people physically move a user story card through its completion — it can engage the development team more than an electronic document ever could. The task board encourages thought and action just by existing in the scrum team's work area, where everyone can see the board.

TIP

Allowing only the product owner to move user stories to the Done column prevents misunderstandings about user story status.

Figure 11-5 shows a typical task board. As you can see, the task board is a strong visual representation of the work in progress.

TECHNICAL STUFF

The task board is a lot like a kanban board. *Kanban* is a Japanese term that means *visual signal.* Toyota created these boards as part of its lean manufacturing process.

In Figure 11-5, the task board shows four user stories, each separated by a horizontal line called *swim lanes.* The first user story is done. All tasks are completed, and the product owner has accepted the work done. For the second user story, the development work is completed but is waiting for acceptance by the product owner. The third user story is in progress, and the fourth user story has not yet been started. At a glance, the status of each user story is clear not only to the scrum team, making tactical coordination faster and easier, but also to interested stakeholders.

FIGURE 11-5: Sample task board.

Agile product development day-to-day work involves more than just planning and tracking progress. In the next section, you see what most of your day's work will include, whether you're a member of the development team, a product owner, or a scrum master.

REMEMBER

The product owner owns the product backlog. The development team owns the sprint backlog. Ownership means keeping the backlog updated, clarified, and transparent.

Agile Roles in the Sprint

Each member of a scrum team has specific daily roles and responsibilities during the sprint. The day's focus for the development team is producing shippable functionality. For the product owner, the focus is on preparing the product backlog for future sprints while supporting the development team's execution of the sprint backlog with real-time clarifications. The scrum master is the agile coach and

maximizes the development team's productivity by removing roadblocks and protecting the development team from external distractions.

Following are descriptions of the responsibilities of each member of the product team during the sprint.

Keys for daily product owner success

A successful product owner focuses on ensuring that the development team has everything it needs to keep the development speed high. The product owner works to understand the problems and needs of customers by meeting with them frequently. Product owners shield the development team from competing priorities, allowing the team to focus on the sprint goal. By sitting with the rest of the scrum team, the product owner can provide instant feedback as work is completed, enabling the development team to turn requirements into valuable working functionality.

The product owner has the following responsibilities during a typical day in a sprint:

- **Proactive contributions:**
  - Look forward to the next sprint and elaborate user stories in readiness for the next backlog refinement or sprint planning meeting.
  - Add new user stories to the product backlog as necessary and ensure that new user stories support the product vision, release goal, and sprint goal.
  - Collaborate with other product owners or stakeholders to align on release or sprint goals. Maintain the product backlog as necessary.
  - Review the product budget to stay abreast of product expenses and revenues.
  - Review product performance information and trends in the marketplace.
  - With the scrum master, watch for opportunities to proactively remove impediments that could impede development if not addressed early, such as product questions arising during sprint planning or legal wording.
- **Reactive contributions:**
  - Provide immediate clarification and decisions about requirements to keep the development team developing.
  - Remove business impediments brought by other members of the scrum team, such as unplanned requests from other teams or stakeholders. Shield the team from business distractions.
  - Review completed user story functionality and provide feedback to the development team.

Keys for daily development team member success

Successful development team members have pride in their work. They build high-quality, enduring products. They engineer their work for change, realizing refactoring is necessary and expected as more is learned. They sit next to their teammates and perform tasks, even unfamiliar tasks, to expand their capabilities and contributions to their team. They excel in their particular discipline and look to expand their capability each day.

If you're a member of the development team, you

- **Proactive contributions:**
  - Select the tasks of highest need and complete them as quickly as possible.
  - Collaborate with other development team members to design the approach to a specific user story, seek help when you need it, and provide help when another development team member needs it.
  - Collaborate with other scrum development teams to technically align on release or sprint goals.
  - Continually improve regression test automation, CI/CD pipelines, and unit testing.
  - Evaluate opportunities to improve the product architecture and development processes.
  - Alert the scrum master to any roadblocks you can't effectively remove on your own.
- **Reactive contributions:**
  - Request clarification from the product owner when you're unclear about a user story.
  - Conduct peer reviews on one another's work.
  - Take on tasks beyond your normal role as the sprint demands.
  - Fully develop functionality as agreed to in the definition of done (described in the next section, "Creating Shippable Functionality").
  - Report daily on the amount of work remaining for your tasks in the sprint backlog.

Keys for daily scrum master success

Successful scrum masters are both coaches and facilitators. They coach the team to improved performance and facilitate team interactions to help the team reach decisions quickly. They inspire, lead, challenge, and serve. Because scrum masters also sit with the team, each day they look for opportunities to serve their team members by removing impediments, by coaching the broader organization to better work with the team, and by making sure the organization's environment enables success. After low-hanging-fruit team improvements are made, a scrum master's job only gets harder as he or she works to remove more difficult organizational impediments affecting the team.

If you're a scrum master, you do the following during a typical day:

- **Proactive contributions:**
  - Uphold agile values and practices by coaching the product owner, development team, and organization when necessary.
  - Remove roadblocks and organizational issues, both tactically for immediate problems and strategically for potential long-term issues. Scrum masters question the status quo of organizational constraints that strategically impede scrum teams from becoming higher functioning. In Chapter 7, we compare the scrum master to an aeronautical engineer, continually removing and preventing organizational drag on the development team.
  - Build relationships to foster close cooperation with people working with the scrum team. Build clout and champion agility throughout the organization.

    Nonverbal communication says a lot. Scrum masters can benefit from understanding body language to identify unspoken tensions in the scrum team.
  - Prepare for upcoming facilitation opportunities, such as researching retrospective models to help the team maximize its retrospective discussion and acquiring supplies to facilitate affinity estimation.
  - Shield the development team from external distractions.
  - Collaborate with other scrum masters and stakeholders to resolve or escalate impediments.
- **Reactive contribution:**
  - Facilitate consensus building in the scrum team, as needed.

REMEMBER

TIP

We often tell scrum masters, "Never lunch alone. Always be building relationships." You never know when you'll need to rely on relationships to remove an obstacle.

Keys for daily stakeholder success

As members of the product team, successful stakeholders know how to work with the product owner to ensure product success. They counsel, collaborate, and listen. They give feedback and support. In flat, agile organizations, stakeholders empower, coach, and serve the scrum team rather than direct its activities from the outside or top-down. Each day, they participate in team discussions when asked but otherwise reserve their feedback for sprint review discussions. For more on the product stakeholder role, see Chapter 7.

If you're a product stakeholder, you do the following during a typical day:

- **Proactive contributions:**
  - Counsel with the product owner on customer needs and backlog priorities.
  - Look for opportunities to remove team impediments. Continually ask how you can help.
- **Reactive contributions:**
  - Participate in sprint reviews and provide feedback. Be available to attend any other discussion requested by the team.
  - Look at team burndowns or task boards. Look for opportunities to help the team become successful.
  - Practice Principle 5, "Give the team the environment and support they need, then trust them to get the job done."

Keys for daily agile mentor success

For teams new to agile techniques, agile mentors are critical sounding boards for the team. They challenge the team's thinking, helping to create healthy tension. Similar to scrum masters, they coach, challenge, and serve. They teach them to find answers themselves (rather than just give them the answers). The team understands that it will get honesty and candor from the agile mentor. Agile mentors work to become redundant, transitioning their experience and expertise to the scrum master. Agile mentors participate daily in whatever way the team needs them.

Strategically, agile mentors engage with the organization's leaders to help maximize the value created by the teams. The scrum team's maximum pace is determined by the environment in which they're working. Agile mentors help leaders improve the environment according to agile values and principles. See Chapter 18 for more on the agile mentor's strategic role.

If you're an agile mentor, you do the following during a typical day:

- **Proactive contributions:**
  - Counsel with the scrum master, primarily to help them build expertise, clout, and capability to effectively coach, trailblaze, and facilitate.
  - Provide agility mentoring to developers and the product owner in the form of in-the-moment course corrections as they strive to learn and improve their roles.
  - Coach stakeholders and other organizational leaders on how they can best support the scrum team to deliver valuable and potentially shippable functionality for the customer at every sprint.
- **Reactive contributions:**
  - Observe scrum team events as well as informal interactions and provide feedback and guidance.
  - Attend any discussion requested by the team.
  - Inspect team burndowns or task boards. Provide feedback on opportunities to help the team become successful.

As you can see, each scrum team member has a specific job in the sprint. In the next section, you see how the product owner and development team work together to create the product.

Creating Shippable Functionality

The objective of the day-to-day work of a sprint is to create shippable functionality for the product in a form that can be delivered to a customer or user.

Within the context of a single sprint, a *product increment* or *shippable functionality* means that a work product has been developed, integrated, tested, and documented according to the definition of done and is deemed ready to release. The development team may or may not release the increment at the end of the sprint because release timing depends on the release plan. The release plan may require multiple sprints before the product contains the set of minimum marketable features necessary to justify a market release.

TIP

It helps to think about shippable functionality in terms of user stories. A user story starts out as a written requirement on a card. As the development team creates functionality, each user story becomes an action a user can take. Shippable functionality equals completed user stories.

To create shippable functionality, the development team and the product owner are involved in three major activities:

- Elaborating
- Developing
- Verifying

During the sprint, any or all of these activities can be happening at any given time. As you review them in detail, remember that they don't always occur in a linear way.

Elaborating

With agile product development, *elaboration* is the process of determining the details of a product feature. Whenever the development team tackles a new user story, elaboration ensures that any unanswered questions about a user story are answered so that the process of development can proceed.

The product owner works with the development team to elaborate user stories, but the development team should have the final say on design decisions. The product owner should be available for consultation if the development team needs further clarification on requirements throughout the day.

WARNING

Collaborative design is a major factor for successful products. Remember these agile principles: "The best architectures, requirements, and designs emerge from self-organizing teams," and "Business people and developers must work together daily throughout a project." Watch out for development team members who have a tendency to try to work alone on elaborating user stories. If a member of the development team separates himself or herself from the team, perhaps part of the scrum master's job should be coaching that person on upholding agile values and practices.

Developing

During product development, most of the activity naturally falls to the development team. The product owner continues to work with the development team on an as-needed basis to provide clarification and to approve developed functionality.

TIP

The development team should have immediate access to the product owner. Ideally, the product owner sits with the development team when he or she is not interacting with customers and stakeholders.

The scrum master should also sit with the development team. He or she focuses on protecting the development team from outside disruptions and removing impediments that the development team encounters.

To sustain agile practices during development, be sure to implement the type of development practices from extreme programming we show you in Chapter 5, including the following:

- **Pair up development team members to complete tasks.** Doing so enhances the quality of the work and encourages the sharing of skills.
- **Follow the development team's agreed-upon design standards.** If you can't follow them for whatever reason, revisit these standards and improve them.
- **Start development by setting up automated tests.** You can find more about automated testing in the following section and in Chapter 17.
- **Avoid coding new features that are outside the sprint goal.** If new, nice-to-have features become apparent during development, add them to the product backlog.
- **Integrate changes that were coded during the day, one set at a time.** Test for 100 percent correctness. Integrate changes at least once a day; some teams integrate many times a day.
- **Undertake code reviews to ensure that the code follows development standards.** Identify areas that need revising. Add the revisions as tasks in the sprint backlog.
- **Create technical documentation as you work.** Don't wait until the end of the sprint or, worse, the end of the sprint prior to a release.

TECHNICAL STUFF

Continuous integration is the term used in software development for integrating and comprehensively testing with every code build. Continuous integration helps identify problems before they become crises. Continuous integration (CI) paired with continuous deployment (CD) is known as CI/CD. Together, teams are able to release early and often. Read more about CI/CD in Chapter 10.

Verifying

Verifying the work done in a sprint has three parts: automated testing, peer review, and product owner review.

TIP

It is exponentially cheaper to prevent a defect than it is to rip it out of a deployed system.

Automated testing

Automated testing means using a computer program to do the majority of your testing for you. With automated testing, the development team can quickly develop and test the product, which is a big benefit for improved team agility.

Often, scrum teams develop during the day, with regression test automation and security vulnerability scanning run on a nightly or weekly cycle. After the cycle completes, the team can review the defect report that the testing program generated, report on any problems during the daily scrum, and correct those issues immediately during the day.

Software automated testing can include the following:

- **Unit testing:** Testing source code in its smallest parts — the component level
- **System testing:** Testing the code with the rest of the system
- **Integration testing:** Verifying that new functionality created in the development environment still works when integrated with the existing functionality
- **Regression testing:** Testing the product increment with previous product increments to ensure that previous functionality continues to work
- **Vulnerability or penetration testing:** Security testing to evaluate the product's exposure to internal and external threats
- **User acceptance testing:** Validating that the new functionality satisfies the acceptance criteria
- **Static testing:** Verifying that the product's code meets standards based on rules and best practices that the development team has agreed upon

Peer review and team development techniques

Peer review and pair programming are techniques teams use to build product increments. *Peer review* simply means that development team members review one another's work. *Pair programming* means that two people work together, with one person driving (the pilot) and one observing from behind (the navigator). Both practices improve product quality, build or expand team member capability, and reduce single-point-of-failure exposure.

A newer trend gaining momentum is mob programming. *Mob programming* is an approach for product development in which the entire team works on the same thing, at the same time, in the same space, and at the same computer. The entire team continuously collaborates at a single computer to deliver a single work item at a time. Customers are often invited to participate with the team as well. Mob

programming extends the benefits of pair programming from two people to the entire team.

Benefits of mob programming include a broader technical understanding of the product, a faster resolution of communication and decision-making problems, preventing the need to do more than is barely sufficient, reduced technical debt, reduced thrashing of the team and team members, and reduced work in progress.

However the team chooses to review each other's work, collocation helps make the review easy and informal — you can turn to the person next to you and ask him or her to take a quick look at what you just completed. Self-managing teams should decide what works best for them.

Product owner review

When a user story has been developed and tested, the development team moves the stories to the Accept column on the task board discussed later in this chapter. The product owner then reviews the functionality and verifies that it meets the goals of the user story, per the user story's acceptance criteria. The product owner verifies (accepts or rejects) user stories throughout each day as the development team completes them.

As discussed in Chapter 10, the back side of each user story card has verification steps. These steps allow the product owner to review and confirm that the code works and supports the user story. Figure 11-6 shows a sample user story card's verification steps.

When I do this:	This happens:
When I go to the accounts page:	I am able to see my active account balance.
When I select transfer funds:	I am able to select "Transfer to Account" and amount.
When I submit transfer requests:	I get an account confirmation funds were transferred.

FIGURE 11-6: User story verification.

Finally, the product owner should run through some checks to verify that the user story in question meets the definition of done. When a user story meets its acceptance criteria as well as the definition of done, the product owner updates the task board by moving the user story from the Accept column to the Done column.

While the product owner and the development team are working together to create shippable functionality for the product, the scrum master helps the scrum team to identify and clear roadblocks that appear along the way.

Identifying roadblocks

A major part of the scrum master's role is managing and helping resolve roadblocks that the scrum team identifies. Roadblocks are anything that thwarts a team member from working to full capacity.

Although the daily scrum is a good place for the development team to identify roadblocks, the development team may come to the scrum master with issues anytime throughout the day.

Examples of roadblocks follow:

- Local, tactical issues, such as
  - A manager trying to pull away a team member to work on a "priority" sales report.
  - The development team needing additional hardware, software, or access to facilitate progress.
  - A development team member who doesn't understand a user story and says the product owner isn't available to help.
- Organizational impediments, such as
  - An overall resistance to agile techniques, especially when the company established and maintained prior processes at significant cost.
  - Managers who might not be in touch with the work on the ground. Technologies, development practices, and project management practices are always progressing.
  - External departments that may not be familiar with scrum needs and the pace of development when using agile techniques.
  - An organization that enforces policies that don't make sense for scrum teams. Centralized tools, budget restrictions, and standardized processes that don't align with agile processes can all cause issues for scrum teams.

REMEMBER

The most important trait a scrum master can have is organizational clout or influence. Organizational clout gives the scrum master the ability to have difficult conversations and make the small and large changes necessary for the scrum team to be successful. We provide examples of different types of clout in Chapter 5.

Beyond the primary focus of creating shippable functionality, other things happen during the day. Many of these tasks fall to the scrum master. Table 11-1 shows potential roadblocks and the action that the scrum master can take to remove the impediments.

TABLE 11-1 Common Roadblocks and Solutions

Roadblock	Action
The development team needs simulation software for a range of mobile devices so that it can test the user interface and functionality.	Do some research to estimate the cost of the software, prepare a summary with the product owner, and have a discussion about funding. Process the purchase through procurement, and deliver the software to the development team.
Management wants to borrow a development team member to write a couple of reports. All your development team members are fully occupied.	Tell the requesting manager that the person is not available and probably will not be for the duration of the sprint. Recommend that the requester discuss the need with the product owner so he or she can prioritize it against the rest of the product backlog. As you're likely a problem solver, you may want to suggest alternative ways in which the manager could get what he or she needs.
A development team member can't move forward on a user story because he or she does not fully understand the story. The product owner is out of the office for the day on a personal emergency.	Work with the development team member to determine if any work can happen around this user story while waiting for an answer. Help locate another person (stakeholder, customer, or subject matter expert) who could answer the question. Failing that, ask the development team to review upcoming tasks (not related to this stopped one) and move things around to keep progressing.
A user story has grown in complexity and now appears to be too large for the sprint length.	Have the development team work with the product owner to break the user story down so that some demonstrable value can be completed in the current sprint and the rest can be put back into the product backlog. The goal is to ensure that the sprint ends with completed user stories, even smaller ones, rather than incomplete user stories.

Information Radiators

Each day, the team uses information radiators to broadcast important information to not only itself but also to stakeholders. An *information radiator* is a poster, task board, list, or any artifact that can be viewed on-demand. Information radiators such as a sprint backlog or a task board reduce questions such as, "What's the status of a story?" or "Is the team on track to meet the sprint goal?" Most information radiators are posted in the team's physical workspace or, if the team is collaborating with other teams, in a common collaboration or meeting area. If teams are using digital collaboration tools, these information radiators are clearly made transparent through obvious links and easy access.

WARNING

Low-fidelity tactile information radiators are our favorite because they're in your face and can't be ignored, even unintentionally. They are referenced more frequently than digital tools, which you have to click or search to find. One way that a scrum master creates an environment for team success is by ensuring transparency of useful artifacts through information radiators.

Information radiators that teams find helpful include the following:

- **Product vision statement and product roadmap:** Provides constant visibility and clarity on the strategic product direction. See Chapter 9.
- **Product canvas:** Visualizes personas, needs, objectives, and other considerations established at the beginning of product development, and can be updated as the scrum team learns more about the customer and marketplace. See Chapter 4.
- **Product backlog:** Helps scrum team members and stakeholders visualize product capability and what priorities are coming up next. See Chapter 9.
- **Sprint backlog:** Displays the scope of the sprint and the status of each task.
- **Task board:** Displays the status of each user story in the sprint. See an example of a task board earlier in this chapter.
- **Team working agreement:** Reminds the team of the behaviors it agreed to uphold as it works together. The agreement can be updated during sprint retrospectives and other team discussions. See Chapter 8.
- **Release and sprint burndown charts with goals:** Visualizes daily the progress and trends of each iteration towards goals. See Chapter 10.
- **Definition of done:** Reminds the team what shippability means and the work each user story requires. Updated during retrospectives and as the team's abilities evolve. See Chapters 2, 10, 12, and 17.
- **Personas:** Reminds the team through visualization of who their customers are as the team performs their work. See Chapter 4.
- **Agile Manifesto, Agile Principles, and scrum values:** Reminds the team of the guiding values and principles it is trying to enable, and are referred to frequently by scrum masters and agile mentors as they coach the team throughout the day. See Chapters 2 and 7.

So far in this chapter, you have seen how the scrum team starts its day and works throughout the day. The scrum team wraps up each day with a few tasks as well. The next section shows you how to end a day within a sprint.

The End of the Day

At the end of each day, the development team reports on the progress of tasks by updating the sprint backlog with which tasks were completed and how much work, in hours, remains to be done on new tasks started. Depending on the tool

that the scrum team uses for tracking progress, the sprint backlog data may automatically update the sprint burndown chart as well.

TIP

Update the sprint backlog with the amount of work remaining — not the amount of time already spent — on open tasks. The important point is how much time and effort remains, which informs the team as to whether it is on track to meet its sprint goal. If possible, avoid spending time tracking how many hours were spent working on tasks, which is less necessary with self-correcting agile models. Also, we follow the rule that it should take a development team less than one minute to update enterprise status reporting. If it's taking longer than that, you have the wrong tool. Our free sprint backlog template mentioned earlier in the chapter can help with this. To download it, go to `www.dummies.com/go/agileprojectmanagementfd3e`.

The product owner should also update the task board at least at the end of the day, and move any user stories that have passed review to the Done column.

The scrum master should review the sprint backlog or task board for any risks or impediments before the next day's daily scrum.

The scrum team follows this daily cycle until the end of the sprint, when it will be time to step back, inspect, and adapt at the sprint review and the sprint retrospective meetings.

IN THIS CHAPTER

- Showcasing work and collecting feedback
- Reviewing the sprint and improving processes

Chapter 12

Showcasing Work, Inspecting, and Adapting

At the end of each sprint, the scrum team gets a chance to put the results of its valuable work on display in the sprint review. The sprint review is where the product owner and the development team demonstrate to the stakeholders the sprint's completed and potentially shippable functionality. In the sprint retrospective, the scrum team (the product owner, development team, and scrum master) review how the sprint went and determine possible improvement opportunities for the next sprint. Underpinning both events is the agile concept of inspect and adapt, which Chapter 9 explains.

In this chapter, you discover how to conduct a sprint review and a sprint retrospective.

The Sprint Review

The *sprint review* is a meeting to review and demonstrate the shippable and valuable functionality that the development team completed during the sprint, and for the product owner to gather feedback and update the product backlog accordingly. The sprint review is open to anyone interested in reviewing the sprint's accomplishments. This means all stakeholders get a chance to see progress on the product and provide feedback.

The sprint review is stage 6 in the Roadmap to Value. Figure 12-1 shows how the sprint review fits into agile product development.

FIGURE 12-1: The sprint review in the Roadmap to Value.

The following sections show you what you need to do to prepare for a sprint review, how to run a sprint review meeting, and the importance of collecting feedback.

Preparing to demonstrate

Preparation for the sprint review meeting should not take more than a few minutes at most. Even though the sprint review might sound formal, for scrum teams, the essence of showcasing is informality. The meeting needs to be prepared and organized but doesn't require flashy materials. Instead, the sprint review focuses on demonstrating what the development team has done.

WARNING

If your sprint review is overly showy, ask yourself if you're covering up for not spending enough time developing. Get back to working on value — creating a working and shippable product. **Pageantry is the enemy of agility.**

The preparation for the sprint review meeting involves the product owner and the development team, facilitated by the scrum master as needed. The product owner knows exactly what the development team completed in the sprint because he or she was working alongside them to accept or reject the items completed as valuable and shippable. The development team needs to be ready to demonstrate completed, shippable functionality.

The time needed to prepare for sprint review should be minimal — usually no more than 20 minutes — just enough to make sure everyone knows who is doing what and when, so the demonstration goes smoothly.

REMEMBER

Work not delivered has no business value. Within the context of a single sprint, *shippable functionality* means that the development team has satisfied its definition of done for each requirement, and the product owner has verified that the work product meets all acceptance criteria and could be released, or *shipped,* to the market if the value and timing are right for the marketplace. The actual release may

be at a later time, per the communicated release plan. Find out more about shippable functionality in Chapter 11.

For the development team to demonstrate the functionality in the sprint review, it must be complete according to the definition of done. In other words, the product increment is fully

- Developed
- Tested
- Integrated
- Documented

As user stories are moved to a status of done throughout the sprint, the product owner and development team should check that the product meets these standards as well as the user stories' acceptance criteria. This continuous validation throughout the sprint reduces end-of-sprint risks and helps the scrum team spend as little time as possible preparing for the sprint review.

Knowing the completed user stories and being ready to demonstrate those stories' functionality prepare you to confidently start the sprint review meeting.

The sprint review meeting

Sprint review meetings have three activities: Demonstrate and showcase the scrum team's finished work, allow stakeholders to provide feedback on that work, and make product adaptations based on reality and stakeholder feedback. Figure 12-2 shows the different loops of feedback a scrum team receives about a product.

FIGURE 12-2: Agile project feedback loops.

This cycle of feedback repeats throughout the project as follows:

- Each day, development team members work together in a collaborative environment that encourages feedback through peer reviews and informal communication.
- Throughout each sprint, as soon as the development team completes each requirement, the product owner provides feedback by reviewing the working functionality for acceptance. The development team then immediately incorporates that feedback, if any, to satisfy the user story's acceptance criteria. When the story is complete, the product owner gives final acceptance of the functionality created for the user story, according to the user story's acceptance criteria.
- At the end of each sprint, project stakeholders provide feedback about completed functionality in the sprint review meeting.
- With each release, end-customers provide feedback about new working functionality.

The sprint review usually takes place later in the day on the last day of the sprint, often a Friday for a sprint that starts on a Monday. One of the rules of scrum is to spend no more than four hours in sprint review for a one-month sprint, so you usually spend no more than one hour in a sprint review meeting for every week of the sprint, as shown in Figure 12-3.

If my sprint is this long…	My sprint review meeting should last no more than…
One week	One hour
Two weeks	Two hours
Three weeks	Three hours
Four weeks	Four hours

FIGURE 12-3: Ratio of sprint review meeting to sprint length.

Here are some guidelines for your sprint review meeting:

- No slides! Show actual working functionality. Refer to the sprint backlog if you need to display a list of completed user stories.
- The entire scrum team should participate in the meeting.

- Anyone who is interested in the product may attend. The product stakeholders, the summer interns, and the CEO could all theoretically be in a sprint review. Customers may also be invited whenever available.
- The product owner introduces the release goal, the sprint goal, and the new capabilities.
- The development team demonstrates the working product increment *completed* during the sprint. Typically, the development team showcases new features or architecture. When negative or critical feedback is given, resist the temptation to become defensive because it will discourage stakeholder feedback.
- The demonstration should be on equipment and environments as close as possible to the planned production environment. For example, if you're creating a mobile application, present the features on a smartphone — perhaps hooked up to a monitor — rather than from a simulator on a laptop.
- The stakeholders can ask questions and provide feedback on the demonstrated product increment.
- Do not use non-disclosed rigged functionality, such as hard-coded values and other programming shortcuts that make the application look more mature than it currently is. Rigged functionality creates more work for the scrum team in future sprints, catching up to what the stakeholders think already exists. Build trust by setting accurate expectations.
- Sprint reviews are excellent opportunities to reflect with transparency on the remaining budget to help the product owner evaluate the value of the remaining backlog.

  The formula AC+OC>V, which is discussed in Chapter 15, is helpful here for deciding when to stop or shift development. When the actual cost (AC) plus the opportunity cost (OC) of the future development is greater than the value (V), stop or shift development. Sprint reviews are great opportunities for concluding with stakeholders that future investment in the feature or product development should end.
- The product owner can lead a discussion about what is coming next based on the features just presented and new items added to the product backlog during the current sprint.

TIP

REMEMBER

By the time you get to the sprint review, the product owner has already seen the functionality for each of the user stories that will be presented and has agreed that they are complete. If the product owner does not accept a user story that the development team worked on, the story doesn't get demonstrated in the sprint review. As the product is iteratively built sprint after sprint, the sprint review is critical for ensuring alignment with the stakeholders and the product vision. Every sprint, the team should be obsessed with solving customer problems.

The sprint review meeting is valuable for the development team because it can show its work to stakeholders and customers and get its efforts acknowledged. The meeting contributes to development team morale, keeping the team motivated to accomplishing the product vision, solving customer problems, and achieving the desired business outcomes. Stakeholder confidence and trust in the development team increases as they hear the team use business language during the product demonstration. (Another benefit of having stable teams is that they retain hard-earned knowledge about the customer and the business.)

TIP

Inspection is a valuable tool. The largest number of world records are broken at the Olympics. Why? Because millions of people from all over the world are watching. Sprint reviews provide a similar, albeit much smaller, stage for scrum teams. Accountability is higher because of the frequently exposed transparency of their work.

Next, you see how to note and use the stakeholders' feedback during the sprint review meeting.

Collecting feedback in the sprint review meeting

Gather sprint review feedback informally. The product owner or scrum master can take notes on behalf of the development team, as team members often will be engaged in the presentation and resulting conversation. Capturing feedback publicly on a whiteboard, for example, validates that the feedback was given and received as intended. The transparency also prevents duplicates.

Because the sprint goal was selected based on the team's assumptions about what the customer wanted, the sprint review offers the team the opportunity to validate its assumptions with stakeholders and, even better, customers.

Keep in mind the example project we use throughout the book: a mobile application for XYZ Bank. Stakeholders responding to functionality they saw for the XYZ Bank mobile application might have comments such as the following:

- From a person in sales or marketing: "You might want to consider letting the customers save their preferences, based on the results you showed. It will make for a more personalized experience going forward."
- From a functional director or manager: "Given what I've seen, you might be able to leverage some of the code modules that were developed for the ABC project last year. They needed to do similar data manipulation."

» From someone who works with the quality or user experience professionals in the company: "I noticed your logins were pretty straightforward. Will the application be able to handle special characters?"

New user stories may come out of the sprint review. These stories could be new features or changes to existing functionality. Both are welcome.

TIP

In the first few sprint reviews, the scrum master may need to remind stakeholders about agile principles and practices. Some people hear the word *demonstration* and immediately expect fancy slides and printouts. The scrum master can shield the scrum team by managing these expectations and helping stakeholders uphold agile values and practices.

The product owner needs to add any new user stories to the product backlog and order those stories by priority. The product owner also adds back to the product backlog any stories scheduled for the current sprint but not completed, and reorders those items based on the most recent priorities.

The product owner needs to complete updates to the product backlog in time for the next sprint planning meeting.

When the sprint review is over, it's time for the sprint retrospective. You may want to take a brief break between the sprint review and the sprint retrospective so that scrum team members can come to the retrospective discussion fresh and relaxed.

Having just completed the sprint review, the scrum team will come into the retrospective ready to inspect its processes and will have ideas for adaptation.

The Sprint Retrospective

The *sprint retrospective* is a meeting in which the product owner, development team, and scrum master discuss how the sprint went and what they can do to improve the next sprint. The scrum team should conduct this meeting in a self-directed way. If managers or supervisors attend sprint retrospectives, scrum team members will avoid being open with each other, which limits the effectiveness of the team's ability to inspect and adapt in a self-organizing way.

The team might invite others with whom they regularly interact (such as stakeholders) to participate in the sprint retrospective, but these invitations are generally an exception.

The sprint retrospective is stage 7 in the Roadmap to Value. Figure 12-4 shows how the sprint retrospective fits into agile product development.

Stage 7: SPRINT RETROSPECTIVE

Description: Team refinement of environment and processes to optimize efficiency
Owner: Scrum team
Frequency: At the end of each sprint

FIGURE 12-4: The sprint retrospective in the Roadmap to Value.

The goal of the sprint retrospective is to continuously improve your processes, environment, collaboration, skillsets, practices, and tools. Improving and customizing how people work together according to the needs of your scrum team increases scrum team morale, improves effectiveness in achieving desired outcomes, and increases *velocity* — work output. (See Chapter 15 for details on velocity.)

However, what works for one team won't necessarily work for another team. Managers outside the scrum team should not dictate how all scrum teams should overcome their challenges and should instead allow them to find the best solutions for themselves.

Your sprint retrospective results may be unique for your scrum team. For example, members of one scrum team we worked with decided that they would like to come to work early and leave early, so they could spend summer afternoons with their families. Members of another team at the same organization felt that they did better work late at night and decided to come to the office in the afternoon and work into the evenings. The result for both teams was increased morale, effectiveness, and velocity.

Use the information you learn in the retrospective to review and revise your work processes and make your next sprint better.

REMEMBER

Agile approaches — particularly scrum — quickly reveal product development problems. Scrum doesn't fix problems; it simply exposes them and provides a framework for inspecting and adapting exposed issues. Data from the sprint backlog shows exactly where the development team has been slowed down. The development team talks and collaborates. These tools and practices help reveal inefficiencies and allow the scrum team to refine practices to improve, sprint after sprint. Pay attention to what gets exposed. Don't ignore it and don't work around it.

STOPPING THE LINE

Taiichi Ohno, who built the Toyota Production System in the 1950s and '60s — the beginning of lean manufacturing — decentralized assembly-line management to empower the line workers to make decisions. Line workers actually had a responsibility to stop the line by pushing a red button when they found a defect or problem on the assembly line. Traditionally, plant managers viewed stopping the line as a failure and focused on running the assembly line at capacity as many hours of the day as possible to maximize throughput. Ohno's philosophy was by removing constraints as they occur, you proactively create a better system rather than trying to optimize your existing process.

When first introduced, the productivity of managers who implemented this practice took an initial drop because they spent more time fixing defects in the system than the managers' teams who did not adopt the practice. The latter teams declared this a victory at first. However, it didn't take long until the former teams not only caught up but also began producing more quickly, more cheaply, and with fewer defects and variance than the teams who weren't making continuous improvements in their system. This process of regular and continuous improvement is what made Toyota so successful.

In the following sections, you find out how to plan for a retrospective, how to run a sprint retrospective meeting, and how to use the results of each sprint retrospective to improve future sprints.

Planning for retrospectives

For the first sprint retrospective, everyone on the scrum team should think about a few key things and be ready to discuss them. What went well during the sprint and what should we keep doing more of? What would we change, and how?

Everyone on the scrum team may want to make a few notes beforehand or even take notes throughout the sprint. The scrum team could keep the roadblocks from the sprint's daily scrum meetings in mind. For the second sprint retrospective forward, you can also start to compare the current sprint with prior sprints, and track progress on the improvement efforts from sprint to sprint. In Chapter 11, we mention saving sprint backlogs from prior sprints; this is one instance where they might come in handy.

If the scrum team has honestly and thoroughly thought about what went right and what could be better, it will go into the sprint retrospective ready to have a useful and actionable conversation.

The retrospective meeting

The retrospective meeting is an action-oriented meeting. The scrum team immediately applies what it learned in the retrospective to the next sprint.

REMEMBER

The sprint retrospective meeting is an action-oriented meeting, not a justification meeting. If you are hearing *because . . .*, the conversation is moving away from action and toward rationale.

One of the rules of scrum is to spend no more than three hours in sprint retrospective for a one-month sprint. So you usually spend no more than 45 minutes in a sprint retrospective meeting for every week of the sprint. Figure 12-5 shows a quick reference of this timetable.

If my sprint is this long…	My sprint retrospective meeting should last no more than…
One week	45 minutes
Two weeks	1.5 hours
Three weeks	2.25 hours
Four weeks	3 hours

FIGURE 12-5: Ratio of sprint retrospective meeting to sprint length.

The sprint retrospective should cover three primary questions:

- What went well during the sprint?
- What would we like to change?
- How can we implement that change?

The following areas are examples of topics for inspection:

- **Results:** Compare the amount of work planned with what the development team completed. Review the sprint or release burndown charts and what they tell the team about how it's working.
- **People:** Discuss team composition and alignment.
- **Relationships:** Talk about communication, collaboration, and how the team works together.

- **Processes:** Go over support, development, and peer review processes.
- **Tools:** How are the different tools working for the scrum team? Think about the artifacts, electronic tools, communication tools, and technical tools.
- **Productivity:** How can the team improve productivity and get the most work done in the next sprint while maintaining a sustainable pace? Remember Principle 8, "Agile processes promote sustainable development." Perhaps there are opportunities to "maximize the amount of work not done" (Principle 10) or to work smarter?

If the team works in one-week sprints, it will have nearly 52 opportunities each year to hold a retrospective. To maintain engagement in each retrospective, many retrospective formats exist, and it's helpful for the team to have variety and to have these discussions in a structured format. Esther Derby and Diana Larsen, authors of *Agile Retrospectives: Making Good Teams Great* (Pragmatic Bookshelf, 2006), offer a great framework for sprint retrospectives that keep the team focused on discussions that will lead to real improvement:

1. **Set the stage.**

   Establishing the goals and scope for the retrospective up front will help keep your scrum team focused on providing the right kind of feedback later in the meeting. As you progress into later sprints, you may want to have retrospectives that focus on one or two specific areas for improvement.

2. **Gather data.**

   Discuss the facts about what went well in the last sprint and what needed improvement. Create an overall picture of the sprint; consider using a whiteboard to write down the input from meeting attendees.

3. **Generate insights.**

   Take a look at the data gathered and come up with ideas about how to make improvements for the next sprint.

4. **Decide what to do.**

   Determine — as a team — which ideas you want to put into place. Decide on specific actions you can take to make the ideas reality.

5. **Close the retrospective.**

   Reiterate your plan of action for the next sprint. Thank people for contributing. Also find ways to make the next retrospective better!

For some scrum teams, it might be difficult to open up at first. The scrum master may need to ask specific questions to start discussions. Participating in retrospectives takes practice. What matters is to encourage the scrum team to take responsibility for the sprint — to truly embrace being self-managing.

In other scrum teams, a lot of debate and discussion ensues during the retrospective. The scrum master facilitates these discussions to guide them towards the desired outcome and keeps the meeting within its allotted time.

TIP

Any action item coming out of a sprint retrospective should be added to the product backlog. All work the scrum team will do to achieve the product vision, such as features, technical debt, overhead, and improvements, should be added and prioritized in the product backlog. Scrum teams should agree to include at least one improvement item from a previous retrospective in each sprint to continuously improve how they go about the work of delivering potentially shippable functionality every sprint.

Be sure to use the results from your sprint retrospectives to inspect and adapt every sprint throughout your product development, not just when it is convenient.

Inspecting and adapting

The sprint retrospective is one of the best opportunities you have to put the ideas of inspect and adapt into action. You came up with challenges and solutions during the retrospective — don't leave those solutions behind after the meeting. Make the improvements part of your work every day.

You could record your recommendations for improvement informally. Some scrum teams post the actions identified during the retrospective meeting in the team area to ensure visibility and action on the items listed. Don't forget to add action items to the product backlog as a reminder to implement them during an upcoming sprint.

In subsequent sprint retrospective meetings, it's important to review the evaluations of the prior sprint and make sure you put the suggested improvements into place. High-performing teams have learned how to convert retrospectives into velocity acceleration.

4 Agility Management

IN THIS PART . . .

Incrementally pursue creating value (outcomes) rather than meeting specifications (outputs).

Respond effectively to changes in scope.

Manage vendors and contracts for success.

Monitor and adjust schedules and budget.

Self-organize for optimal communications.

Inspect and adapt to increase quality and mitigate risk.

IN THIS CHAPTER

- Learning how agile portfolio management is different
- Discovering ways to make portfolio investment decisions
- Learning how to manage an agile product portfolio

Chapter 13

Managing a Portfolio: Pursuing Value over Requirements

Organizational leaders today face the daunting challenge of delivering value faster than ever before. With limited resources and funding to keep up with rapidly changing marketplaces, deciding which product investment opportunities will maximize an organization's return is not easy. In this chapter, we share various agile approaches and techniques used to manage a product portfolio. Portfolio management helps organizations to pursue value (outcomes) over meeting requirements (outputs).

Like product development, agile portfolio management is based on the Agile Manifesto values and principles and yields the following results:

- Highest business value delivered first
- More frequent delivery of value
- Lowered cost of pivoting (reprioritization)
- Higher transparency

- Timely data from shorter feedback loops
- Ease in managing by exception due to team and product transparency
- Small, incremental continuous improvement
- Sustainable productivity
- Improved morale due to improved focus and sustainable pace
- Not only fulfilled requirements, but also maximized value

Understanding the Differences in Agile Portfolio Management

Because scrum teams continually focus on the highest value and risk items first, agile product development efforts often run out of value before they run out of time or money. The same can be said of a portfolio of products. In fact, many of the same principles for maximizing value from a single product can be applied to a portfolio of products.

Agile portfolio management is the art and science of selecting and overseeing a group of product investments that meet an organization's long-term objectives and risk tolerance. Managing a portfolio of products requires the ability to weigh strengths and weaknesses, opportunities, and threats across the full spectrum of opportunities. The choices involve trade-offs: short term versus long term, expanding market segments versus contracting ones, domestic versus international, and growth versus existing product investment.

Weighing the various trade-offs truly is an art and a science. Product owners and portfolio leaders evaluate strengths, weaknesses, opportunities, and threats (SWOT). Effective portfolio management enables maximization of value for customers and stakeholders.

TECHNICAL STUFF

A SWOT analysis can be conducted on anything the team wants, such as its product, company, team, or portfolio. The sky is the limit. After determining the topic of their conversation, the team brainstorms the strengths, weaknesses, opportunities, and strengths. The outcome of the analysis is a concise summary of the most important factors to be considered by the team, which may guide the product portfolio, backlog, vision, and roadmap.

All agile principles can be helpful for managing organizational product portfolios:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

As a portfolio of products is evaluated from a macro level, key decisions must be made about which product investment opportunities should be pursued in general, which should be pursued next, and which might be pursued later. A careful balance must also be set to ensure that the organizational and team capacity is healthy and thriving. Overburdening leads to reduced innovation, tiredness, and acceptance of the status quo.

Should we invest?

As with financial investments, capital budgeting uses a key metric to estimate the profitability of potential investments: the internal rate of return (IRR). IRR considers the cost of the initial investment against the net present value of the future stream of revenue or cost savings. The higher the IRR, the more confident a portfolio leader can be that the product investment will be profitable.

These same financial return principles can be applied to product portfolio management. The cost of the initial product, including development, labor, licensing, and maintenance, is compared to the future stream of annualized revenue or cost savings. If the internal rate of return exceeds the cost of capital and the initial investment cost, it might be a worthwhile investment. Other costs to consider are the balance between CapEx (capital expenditures) and OpEx (operational expenditures). The more expenses that can be capitalized, the more profitable an investment.

TECHNICAL STUFF

Capital expenditures, or CapEx, are funds used by a company to acquire, upgrade, and maintain physical assets such as property, buildings, an industrial plant, technology, and equipment. CapEx is often used to undertake new projects or investments by the firm. An operating expense — also known as an operating expenditure, operational expense, or operational expenditure (OpEx) — is an ongoing cost for running a product, business, or system.

Understanding the internal rate of return and balance between CapEx and OpEx helps answer the question, "Should we invest?" Following are other factors to consider in forecasting financial product return:

- **Value and risk:** Portfolio prioritization is a factor of value and risk. Value, because the portfolio must meet the customer's needs. Risk, to enable failing early and failing cheap, giving you the longest runway for finding a solution when the system is the simplest and the most financial resources are available.
- **Short-term versus long-term:** Trade-offs of short-term gains for long-term value.
- **Product mix:** Balance of products to diversify and take advantage of product lifecycles of both new and sunsetting products (discovery to market).

We discuss each in more detail next.

Factors for forecasting product investment returns

Several factors, such as value and risk prioritization, short-term versus long-term, and balancing the product mix, should be considered when forecasting product investment returns. We discuss each of these in the next section.

Value and risk prioritization

Agile product teams are encouraged to focus on one thing, do it well to completion, and then move as a team to the next most important thing. The team works on one item at a time to avoid the significant cost delays of context switching.

Prioritization across a portfolio is similar. The portfolio may have a backlog of investment opportunities, but wise portfolio leaders reduce work in progress and focus on one opportunity at a time. They evaluate the desired outcomes of their focus, and once acceptable, move to the next opportunity.

They prioritize the portfolio according to value and risk. Value criteria could include return on investment and internal rate of return, market share, revenue, cost savings, corporate image, product enhancements, maintenance, security, regulatory compliance, and more. Risk criteria may include the risk of users adopting a new product, of using a technology unfamiliar to the organization, of the organization being unable to deliver the value in the time frame needed, and more. Portfolio prioritization is a function of value and risk. Learn more about using value and risk for prioritization in Chapter 9.

Figure 13-1 shows a helpful value and risk matrix tool that product portfolio leaders use to evaluate various product investment opportunities. Products in the highest value, highest risk quadrant should be attempted first while products with the lowest value, highest risk should be avoided.

FIGURE 13-1: Risk versus value quadrant.

GAINING ORGANIZATIONAL AND PORTFOLIO PRIORITY ALIGNMENT

A healthcare client of ours struggled with annual planning due to the traditional project management approach used by the organization. Capital was reserved and forecasts were continually updated throughout the year so allocation adjustments could be made. Managers watched the CapEx and OpEx targets like hawks. Progress was difficult to make with so many active and dependent pieces of work going all at once. People with specialized skills were required to work on multiple projects at the same time.

In planning the new year, a professional agile mentor was brought in to help. A workshop was scheduled to help them work through the budgeting and productivity challenges. The mentor walked the group through an approach that helped them to begin by agreeing to criteria for evaluating risk and value.

Each leader then filled out 3x5 cards with the title of their planned product investment opportunity and placed it on a risk-value matrix on the wall. In the value label field, they recorded their estimated return on investment ($500K, $300K, $100K, and so on) and defended their ROI estimate as each card was discussed.

After all the cards were placed on the wall, the first acknowledgment they made was that there were too many investment opportunities — their plan was unrealistic. The second observation was that they could prioritize their portfolio by understanding the value, risk, and rough order estimate. The subset of the most valuable opportunities emerged by priority and, more importantly, had organizational alignment.

The client then laid the best opportunities in a spectrum across the wall in prioritized order, starting with the most valuable/most risky opportunities on the left and the least valuable/least risky opportunities on the right. In essence, they created their backlog of product portfolio investment opportunities.

The results were significantly improved. Highest priority opportunities were either successful earlier or the high risk, low value opportunities were quickly removed from the list. The organization's teams could pull from a prioritized backlog of opportunities when they became available, allowing more and more of the organization's talent to focus on delivering the work that mattered most at a pace in their available capacity.

A prioritized and ordered portfolio of product opportunities enables teams to pull from the backlog of investment opportunities. After the first available team pulls from the backlog, the next team checks to see if the first team needs help; if not, it pulls the next most valuable product investment opportunity, and so forth. This approach ensures that the highest valuable product investment opportunities

receive the organization's attention first. It also helps to keep the work in progress (WIP) low while moving at the pace of the organization's capability.

Short-term versus long-term decisions

Portfolio leaders, in partnership with product owners, make short-term versus long-term product portfolio decisions. The question they often ask is, "Will my short-term product investment opportunity be thrown away?" Or, "Will my short-term investment create a path to accomplish the longer-term strategic vision?"

SHORT-TERM PAYMENT OF ORGANIZATIONAL TECHNICAL DEBT YIELDS LONG-TERM RESULTS

The short-term decision to invest in paying down organizational technical debt weighed heavily on the minds of the healthcare organization's newly formed agile transition teams (ATT) as they took seriously their responsibility to support the agile transformation. (See Chapter 18 to learn more about agile transition teams.) As the pilot scrum teams started working, the lack of test automation became exposed. Thousands of test cases that could have been automated were manually used day-in and day-out by everyone. Each new day of created functionality caused more manual test cases to be created, making the pile of manually testing debt even higher.

After consulting with the scrum teams, the ATT decided that the best approach for addressing the debt was to focus a new scrum team on the problem.

The first item on the new scrum team's backlog was to build a testing framework that could be used by all teams. Each pilot team embedded the new testing framework in its definition of done. The framework allowed all the teams to automate their own tests, adding them to the shared pool of tests. While the scrum teams automated the tests on their new sprint's work, the newly formed test automation scrum team tackled the backlog of test automation debt using the new framework. Test case by test case, the debt was paid down.

The product's quality as well as the teams' productivity improved. Changes were much easier to introduce and the teams shifted priorities effortlessly. Manual testers gained new skills in building test automation, which opened doors for learning other skills.

In the long-term, the velocity, or relative pace, of development across all the teams increased due to the investment in automation. The short-term payment of technical debt yielded long-term results.

Scrum teams frequently ask, "What should we do next?" Decisions must be made to either fix something or build or implement something new. A balance between proactive work and reactive work as well as the decision to pay down technical debt or invest in automation or platform upgrades for faster product development must be decided.

To answer these short-term versus long-term decisions, portfolio leaders work closely with product owners, stakeholders, and development teams. The strong communication cycles and quick feedback loops inherent in agile approaches inform their decisions. As they interact with stakeholders and product owners, they learn about customer needs and opportunities. The product owners and stakeholders then quickly build product vision statements and roadmaps with development teams to gain funding support. Release plans and sprint plans follow. Initial sprints give early information about the viability of the opportunity. With minimal investment, information is revealed to inform their short-term versus long-term decision.

Balancing the product mix

Product mix, also known as product assortment, refers to the total number of product lines an organization offers to its customers. For example, think of the different products sold by Nike (shoes, socks, pants, shirts, sweatshirts, team gear, sports equipment, and so on). The four dimensions to a company's product mix include width, length, depth, and consistency:

- **Width:** The number of product lines that an organization offers or the variety offered, for example, the number of product lines from shoes to other clothing or team gear.
- **Length:** The number of products in the company or a given product line, for example, the number of different running shoe types.
- **Depth:** The total number of variations for each product, such as the various sizes of shoes, styles, or colors.
- **Consistency:** The connection between products in the product line and the way they reach the consumer. It describes how closely product lines are related to each other — in terms of use, manufacturing, and distribution. For example, two lines of shoes may be used, manufactured, and distributed in the same way so that their product lines are consistent.

TIP

The product mix is important in determining the image of your business and brand, because it helps you maintain consistency in the eyes of your target market.

Various strategies can be employed to balance the product mix. Small companies usually start with a product mix limited in width, depth, and length and have a high level of consistency. Over time, the organization may want to differentiate products or acquire new ones to enter new markets. They may also add to their lines similar products that are of higher or lower quality to offer different choices and price points. They stretch the product line — upward to add more expensive products and downward to add lesser quality, lower priced items.

A well-balanced marketing mix in a product portfolio is important for a number of reasons. It helps portfolio leaders to maximize the value of their investment. It defines the steps necessary to make a profit by differentiating the products and to take advantage of new markets, customer segments, and price points. Lastly, it sets the course for product marketing.

Managing Agile Product Portfolios

The many moving parts involved in managing an agile product portfolio can make the process complex. Effective portfolios focus on setting realistic and strategic priorities, and reducing work in progress so portfolio items can get completed as quickly as possible. They adhere closely to Principle 10: Simplicity, the art of maximizing the amount of work not done — is essential.

Brent Barton, in his article "The 5 Simple Rules of Agile Portfolio Management," describes the challenges when managing and planning a product portfolio. He concluded that planning new capabilities for multiple products in an organizational silo was like predicting the weather in micro-climates with a less than 50 percent accuracy. The accuracy was so low because the micro-climate meteorologists weren't looking (or couldn't look) beyond their own domain to account for how the bigger system affected weather patterns. In other words, product portfolio planning without the proper perspective for reasonably matching capacity to demand is nearly impossible.

For this reason, Barton defined five simple rules to reduce the complexity of agile product portfolio management:

» **All work is forced ranked.** Organizations that make everything the highest priority cannot articulate real priorities. Force ranking a portfolio of investment opportunities leads to clarity and focus. Similar to answering the "Should we invest?" question, portfolios are best prioritized when they consider value to the customer as well as business risk, technical risk, or both. Tackle the most valuable and most risky opportunities first.

REMEMBER

In his book *Essentialism: The Disciplined Pursuit of Less* (Virgin Books), Greg McKeown said:

The word priority came into the English language in the 1400s. It was singular. Only in the 1900s did we pluralize the term and start talking about "priorities." Illogically we reasoned that by changing the word we could bend reality.

- **Operate on good enough data.** It is unrealistic to expect to have perfectly detailed data for all portfolio decisions when they need to be made. Certain levels of detail in one area do not necessitate expensive-to-obtain detail in other areas. As with all empirical control processes, start with what you know, and then inspect and adapt based on what is learned.
- **Near-term capacity is fixed.** Make investment opportunity decisions based on existing organizational capacity. Adding complexity by forecasting team or skill counts based on the hope influencing portfolio management outcomes is problematic. Forecasting and onboarding new teams in a timely manner is more challenging than leaders want to acknowledge.
- **Each unique value-based delivery capability has a portfolio.** Simplify and decompose the work into the smallest, most valuable increments as possible. Enable teams in the portfolio to become highly aligned and highly autonomous. Connect their work to strategy through progressive elaboration.
- **Each portfolio has one intake system.** Strategic decisions in a portfolio should have full visibility into the entire scope of work required to innovate, build, release, evolve, support, and sunset technology.

These five simple rules for agile product portfolio management help reduce complexity, enable organizational focus, and reward action and speed over analysis paralysis. Like all guidelines, they serve as a good starting point but will need to be adjusted upon inspection.

Other factors to consider for effectively prioritizing a product portfolio follow:

- **Visualize the portfolio.** A map to your destination is easier to follow than verbal or written directions. Similarly, visualization enables portfolios of product teams to collaborate more effectively. In a self-organizing environment, full transparency is essential for inspecting and adapting.

  Ensure that everyone can see how your product portfolio aligns to your corporate vision, objectives (desired outcomes), success criteria, and strategy. Your roadmap of products and capabilities needs to be visible and supportive of the decision-making process. Visualization helps to inform decisions that can be made at the last responsible moment.

Figure 13-2 shows an example of a prioritized portfolio backlog of opportunities aligned with the strategic vision stack, ranked by value, risk, and estimate.

- **Optimize talent allocation.** Ensure that talented scrum teams are optimized for creating maximum benefit.
- **Evaluate product performance.** Ensure that customer driven metrics are available for adapting to changing market conditions and customer demands.
- **Determine prospective value.** Determine new products to design and develop as well as search for new ideas. Monitor your business and your competitors, getting feedback from customers and anticipating clients' future needs. New technologies, laws, or regulations may also influence prospective value.
- **Search possible endeavors.** Search for and invest in understanding possible endeavors. Options include guessing at a product's market potential, doing a case study, or running a focus group.
- **Commence endeavors.** New products or features have to be developed by the team. Initial experimentation can help you see the market potential of the product.

FIGURE 13-2: Prioritized portfolio backlog of investment opportunities.

- **Incrementally fund products.** Every potential endeavor requires funding, including the primary funding for new teams for their inception or development efforts and the budget for ongoing construction, transition, and operations after the product launch. Instead of allocating funding for the entire development effort, consider incrementally funding instead. For example, allocate 90 days' worth of funding and see what kind of value can be created in that time. If successful, fund another 90 days. This funding will also require constant monitoring by the product owner and sponsors to ensure that the money is spent wisely.
- **Engage vendors.** Supplier or vendor management is a key aspect of agile portfolio management. This task entails procuring or awarding contracts, narrowing down and identifying possible vendors, overseeing ongoing projects, and closing a contract. Ensuring that vendors enable agile principles is essential because they'll need to adapt with your changing priorities and customer needs. Many times, you can use an incremental funding model with vendors to more effectively manage portfolios. Enable product owners or scrum masters to engage in vendor relationships early.

Table 13-1 lists important considerations for effective portfolio management with suggestions for what to do and what not to do in each situation.

TABLE 13-1 Keys for Effective Agile Portfolio Management

Topic	Do	Don't
Partner with product owners	Empowered product owners are close to their customers and understand their needs and problems. Engaging product owners in portfolio decisions leads to better aligned portfolios.	Don't exclude product owners from contract negotiations, discovery, or portfolio decisions. Product owners are accountable for a product's return on investment.
Prioritize and limit the work to be done	Tackle the single, next most valuable opportunity, do it well, validate that the desired outcomes were achieved, and then move to the next opportunity. Prioritization and limiting work to be done improves speed to market and organizational focus and builds organic cross-functional capability.	Don't enable teams to work on lower value, low risk work at the expense of the higher value, higher risk work. Working on too many things at once will prevent you from working on the higher priority things that matter most to your customer.
Focus on balancing demand and capacity	Avoid overburdening teams and organizations. Overburdening leads to tired and inefficient people who will make more mistakes, which cause delays and are expensive to correct. Even worse, their ability to innovate will be constrained.	Don't push velocity expectations onto a team. Scrum teams are more effective when they pull work into releases and sprints given their capacity constraints.

Topic	Do	Don't
Maximize both internal and external value	Focus on creating value for your customers as well as keeping a broader perspective of doing what's best for your organization. Agile product portfolio management is more about improving your organization for the future and less about completing projects. It's about directing funds where they'll have the most effect.	Don't focus solely on internal or external value. History shows that internally focused businesses who lose sight of their customers or who neglect to take care of their workforce cease to exist.
Re-prioritize frequently	As market conditions and customer demands change, inspect and adapt. Continuously prioritize your product investment opportunities.	Don't become irrelevant by not adapting to change.
Deliver value in smaller chunks	When planning your portfolio, focus on how small and simple you can make the value increments that you'll release to the market. Doing so not only creates a faster cashflow and return on investment but also reduces complexities and risk.	Don't place value on a shelf, like inventory. When a barely sufficient amount of value is created, release it.
Avoid parallel product work	Product portfolio leaders understand that organizations get less done when they work on several products at the same time. Prioritize, and then select the next most valuable opportunity and finish it. Next, pull the next most valuable investment opportunity only when capacity and capability are available.	Don't allow parallel work. Simplify product development by reducing work in process, which has no value.

In Tom Demarco's book *Slack: Getting Past Burn-out, Busywork, and the Myth of Total Efficiency* (Currency), the anti-pattern of parallel product work is further explained. Tom suggests that people and teams are not fungible (not exchangeable or replaceable). People or teams required to support multiple product development efforts in parallel pay a high cost in context switching. According to the American Psychology Association, the cost can be as high as 40 percent of someone's productive time.

A study performed by Gloria Mark from the University of California, Irvine and Daniela Gudith and Ulrich Klocke from Humboldt University in Germany added that people interrupted in their work also experience higher workloads, more stress, higher frustration, and more time pressure and effort. The same principles apply to parallel, interrupted product development.

Figure 13-3 illustrates the difference between running serial product development efforts (dedicated teams) and parallel efforts (thrashing teams).

In this example, a portfolio has three projects, each running in succession (serially). Assume that one unit of value can be produced in one unit of time and each product development effort when complete produces one unit of value ($).

FIGURE 13-3: Financial cost of delay due to thrashed parallel product development.

Thrashing the team, or causing lack of focus by multitasking or context switching, results in at least 30 percent (as much as 40 percent) more time to finish each effort — 33 percent for this scenario. Over the three product development efforts (99 percent), thrashing the teams around is roughly the equivalent of an entire effort's length.

Going one additional unit of time past deployment of the serial efforts, the parallel efforts finally finish and return $$$ (3$ — one $ for each effort deployed) at the end of that period. The serial product development efforts return $$$$$$+$$$$ — 10$, which is more than three times the return on investment (ROI) compared to parallel development.

REMEMBER

Stop thrashing. Run one project at a time through one team at a time. When you dedicate teams, everyone gets value delivered earlier, as well as an earlier and overall higher ROI. It simply doesn't make sense to overload your portfolio and thrash your teams on more than one thing at a time.

Should we continue investing?

The question of whether you should continue investing is one that all portfolio leaders in collaboration with product owners must answer. Many factors must be considered for making the decision. Leaders and product owners consider "When is the right time to sunset a product?" or "Have we reached the point of diminishing returns, where the investment return will not be as great as other product investments?"

TECHNICAL STUFF

The *law of diminishing returns,* also referred to as the law of diminishing marginal returns, states that in a production process, as one input variable is increased, there will be a point at which the marginal per unit output will start to decrease, holding all other factors constant.

Figure 13-4 shows how to apply the law of diminishing returns to a product portfolio. During the early stage of the portfolio, knowledge value is earned, followed by a steep increase in customer value (or outcomes) as the new capabilities are made available to customers. (The "highest returns" section is the most productive section of the curve in the figure.) It pays to invest more time and effort during this stage. Next is the "diminishing returns" section, when each added input leads to a decreasing rate of output. It's best to stop somewhere within this phase and "trim the tail." In other words, more valuable opportunities should now be considered. The last stage, "negative returns," is the one to avoid. Not only do you not get a return for your effort, but you decrease your overall output.

FIGURE 13-4: The law of diminishing returns.

When the point of diminishing returns is met, it's time to stop or reduce investing in a product. In other words, it's time to shift to the next most valuable investment opportunity using the V < AC + OC equation discussed in Chapter 15.

Inspecting and adapting to the next opportunity

After you've achieved the accomplishment of getting to done, what's next? Maintaining a refined portfolio backlog of investment opportunities makes this

decision easier. This section describes some keys for continuously prioritizing and shifting to the next most valuable investment in your product portfolio.

Shifting to the next most valuable item

Shifting to the next most valuable item is easier to do when a continually prioritized portfolio of investment opportunities is maintained. As with a product backlog, focus on the next portfolio backlog item.

Revising product portfolio investments

If the outcomes from your portfolio investment do not accomplish their intent, it may be time to adjust and adapt. The beauty of an agile product portfolio is its capability to change priorities with minimized disruption. Scrum teams that implement shippable functionality at the end of every sprint have frequent inspection points to determine if enough value has been achieved.

REMEMBER

Agile portfolio management ensures that an organization provides its clients with the best value for their investment while doing what is best for the organization. A good portfolio leader, in partnership with empowered product owners, understands and follows agile principles. They also understand how to align the portfolio to the strategic direction, prioritize, reduce work in progress, and help others to join them in their journey. They pursue value (outcomes) over meeting requirements (outputs).

IN THIS CHAPTER

- Seeing how agile product development changes scope management
- Managing scope and scope changes with agile techniques
- Understanding the different approach that agile practices bring to procurement
- Managing procurement

Chapter 14

Managing Scope and Procurement

Scope management is part of every product development effort. To create a product, you have to understand basic product requirements and the work it will take to fulfill those requirements. You need to be able to prioritize and manage scope changes as new requirements arise. You have to verify that finished product features fulfill customers' needs.

Procurement is also part of many product development efforts. If you need to look outside your organization for help completing development, you should know how to go about procuring goods and services. You will want to know how to collaborate with vendor teams during the product lifecycle. You should also know something about creating contracts and different cost structures.

In this chapter, you find out how to manage scope and take advantage of agile methods' welcoming approach to informed change. You also find out how to manage procurement of goods and services to deliver product scope. First, we review traditional scope management.

What's Different about Agile Scope Management?

Historically, a large part of project management is scope management. *Product scope* is all the features and requirements that a product includes. *Project scope* is all the work involved in creating the project-funded features of a product.

Traditional project management treats changing requirements as a sign of failure in up-front planning. With agile product development, however, scope is variable so that scrum teams can immediately and incrementally incorporate learning and feedback, and ultimately create better products. The signers of the Agile Manifesto recognized that scope change is natural and beneficial. Agile approaches specifically embrace change and use it to make better-informed decisions and more useful products.

TIP

If you use agile product development and your requirements don't change because you learned nothing along the way, that is a failure. Your product backlog should change often as you learn from stakeholder and customer feedback. It's unlikely that you knew everything at the beginning.

REMEMBER

Chapter 2 details the Agile Manifesto and the 12 Agile Principles. (If you haven't yet checked out that chapter, flip back to it now. We'll wait.) The manifesto and the principles answer the question, "How agile are we?" The degree to which your approach supports the manifesto values and the principles helps determine how agile your methods are.

The agile principles that relate the most to scope management follow:

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.

10. Simplicity — the art of maximizing the amount of work not done — is essential.

Agile approaches to scope management are fundamentally different than traditional methods for scope management. Consider the differences you see in Table 14-1.

TABLE 14-1 **Traditional versus Agile Scope Management**

Scope Management with Traditional Approaches	Scope Management with Agile Approaches
Project teams attempt to identify and document complete scope at the beginning of the project, when the teams are the least informed about the product.	The product owner gathers high-level requirements at the beginning of product development, breaking down and further detailing requirements that are going to be implemented in the immediate future. Requirements are gathered and refined throughout development as the team's knowledge of customer needs and product realities grow.
Organizations view scope change after the requirements phase is complete as a failure.	Organizations view change as a positive way to improve a product as development progresses. Changes later in development, when you know more about the product, are often the most valuable changes.
Project managers rigidly control and discourage changes after stakeholders sign off on requirements.	Change management is an inherent part of agile processes. You assess scope and have an opportunity to include new requirements with every sprint. The product owner determines the value and priority of new requirements and adds or swaps those requirements to the product backlog.
The cost of change increases over time, while the ability to implement changes decreases.	You fix resources and schedule initially. New features with high priority don't necessarily cause budget or schedule slip; they simply push out the lowest-priority features. Iterative development allows for changes with each new sprint.
Projects often include *scope bloat,* unnecessary product features included out of fear of mid-project change.	The scrum team determines scope by considering which features directly support the product vision, the release goal, and the sprint goal. The development team creates the most valuable features first to guarantee their inclusion and to ship those features as soon as possible. Less valuable features might never be created, which may be acceptable to the business and the customer after they have the highest-value features.

At any point during agile product development, anyone — the scrum team, stakeholders, or anyone else in the organization with a good idea — can identify new product requirements. The product owner determines the value and priority of new requirements and prioritizes those requirements against other requirements in the product backlog.

TECHNICAL STUFF

Traditional project management has a term to describe requirements that change after the project's initial definition phase: *scope creep.* Waterfall doesn't have a positive way to incorporate changes mid-project, so scope changes often cause large problems with a waterfall project's schedule and budget. (For more on the waterfall methodology, see Chapter 1.) Mention "scope creep" to a seasoned project manager, and you might even see him or her shudder.

During sprint planning at the beginning of each sprint, the scrum team can use the product backlog priority to help decide whether a new requirement should be part of the sprint. Lower-priority requirements stay in the product backlog for future consideration. You can read about planning sprints in Chapter 10.

The next section addresses how to manage scope with agile product development.

Managing Agile Scope

Welcoming scope change helps you create the best product possible. Embracing change, however, requires that you understand the current scope and know how to deal with updates as they arise. Luckily, agile approaches have straightforward ways to manage new and existing requirements:

- The product owner ensures that the rest of the team — the scrum team plus the stakeholders — clearly understands the existing product scope in terms of the product vision, the current release goal, and the current sprint goal.
- The product owner determines the value and priority of new requirements in relation to the product vision, release goal, sprint goals, and existing requirements.
- The development team creates product requirements in order of priority to release the most important parts of the product first.

In the following sections, you find out how to understand and convey scope in different parts of product development. You see how to evaluate priorities as new requirements arise. You also find out how to use the product backlog and other agile artifacts to manage scope.

Understanding scope throughout product development

At each stage in development, the scrum team manages scope in different ways. A good way to look at scope management throughout development is by using the Roadmap to Value, first presented in Chapter 9 and shown again in Figure 14-1.

FIGURE 14-1: The Roadmap to Value.

Consider each part of the Roadmap to Value:

- **Stage 1, product vision:** The product vision statement establishes the outer boundary of the functionality that the product will include, and is the first step in establishing scope. The product owner is responsible for ensuring that all members of the product team know the product vision statement and that everyone on the team interprets the statement correctly.
- **Stage 2, product roadmap:** During product roadmap creation, the product owner refers to the vision statement and ensures that features support the vision statement. As new features materialize, the product owner needs to understand them and be able to clearly communicate to the development team and stakeholders the scope of these features and how they support the product vision.

- **Stage 3, release planning:** During release planning, the product owner needs to determine a release goal — the midterm boundary of functionality that is planned to go to market at the next release — and select only the scope that supports that release goal.
- **Stage 4, sprint planning:** During sprint planning, the product owner needs to ensure that the scrum team understands the release goal and plans each sprint goal — the immediate boundary of functionality to be potentially shippable at the end of the sprint — based on that release goal. The product owner and development team select only the scope that supports the sprint goal as part of the sprint. The product owner will also ensure that the development team understands the scope of the individual user stories selected for the sprint.
- **Stage 5, daily scrum:** In the daily scrum meeting, the scrum team coordinates its focus and progress towards achieving the sprint goal. The meeting can be a launching point for scope change for future sprints.

  TIP

  When topics come up that warrant a bigger discussion than the time and format of the daily scrum meeting allows, a scrum team can decide to have an after-party meeting. In the after-party, scrum team members talk about issues affecting their progress toward the sprint goal. If opportunities for new functionality — new scope — are identified during the sprint, the product owner evaluates them and may add and prioritize them on the product backlog for a future sprint.
- **Stage 6, sprint review:** The product owner sets the tone of each sprint review meeting by reiterating the scope of the sprint — the sprint goal that the scrum team pursued and what was completed. Especially during the first sprint review, it's important that the stakeholders in the meeting have the right expectations about scope.

  Sprint reviews can be inspiring. When the entire team is in one room, interacting with the working product, members may look at the product in new ways and come up with ideas to improve the product. The product owner updates the product backlog based on feedback received in the sprint review.
- **Stage 7, sprint retrospective:** In the sprint retrospective, the scrum team members can discuss how well they met the scope commitments they made at the beginning of the sprint. If the development team was not able to achieve the sprint goal identified during sprint planning, its members will need to refine planning and work processes to make sure they can select the right amount of work for each sprint. If the development team met its goals, it can use the sprint retrospective to come up with ways to add more scope to future sprints. Scrum teams aim to improve productivity with every sprint.

Introducing scope changes

Many people, even people outside the organization, can suggest a new product feature. You might see new ideas for features from the following:

- User community feedback, including groups or people who are given an opportunity to preview the product
- Business stakeholders who see a new market opportunity or threat
- Executives and senior managers who have insight into long-term organizational strategies and changes
- The development team, which is learning more about the product every day, and is closest to the working product
- The scrum master, who may find an opportunity while working with external departments or clearing development team roadblocks
- The product owner, who often knows the most about the product and the stakeholders' needs

Because you will receive suggestions for product changes throughout product development, you want to determine which changes are valid and manage the updates. Read on to see how.

Managing scope changes

When you get new requirements, use the following steps to evaluate and prioritize the requirements and update the product backlog.

WARNING

Do not add new requirements to sprints already in progress, unless the development team requests them, usually due to unexpected increased capacity.

1. **Assess whether the new requirement should be part of the product, the release, or the sprint by asking some key questions about the requirement:**

   (a) *Does the new requirement support the product vision statement?*

   - If yes, add the requirement to the product backlog and product roadmap.
   - If no, the requirement shouldn't be part of the product. It may be a good candidate for a separate product.

(b) *If the new requirement supports the product vision, does the new requirement support the current release goal?*

- If yes, the requirement is a candidate for the current release plan.
- If no, leave the requirement on the product backlog for a future release.

(c) *If the new requirement supports the release goal, does the new requirement support the current sprint goal?*

- If yes and if the sprint has not started, the requirement is a candidate for the current sprint backlog.
- If no or the sprint has already started or both, leave the requirement on the product backlog for a future sprint.

2. **Estimate the effort for the new requirement.**

   The development team estimates the effort. Find out how to estimate requirements in Chapter 9.

3. **Prioritize the requirement against other requirements in the product backlog and add the new requirement to the product backlog, in order of priority.**

   Consider the following:

   - The product owner knows the most about the product's business needs and how important the new requirement may be in relation to other requirements. The product owner may also reach out to product stakeholders for additional insight to a requirement's priority.
   - The development team may also have technical insight about a new requirement's priority. For example, if Requirement A and Requirement B have equal business value, but you need to complete Requirement B for Requirement A to be feasible, the development team will need to alert the product owner. Requirement B may need to be completed first.
   - Although the development team and product stakeholders can provide information to help prioritize a requirement, determining priority is ultimately the product owner's decision.
   - Adding new requirements to the product backlog may mean other requirements move down the list in the product backlog. Figure 14-2 shows the addition of a new requirement in the product backlog.

The product backlog is a complete list of all known scope for the product and is your most important tool for managing scope change.

FIGURE 14-2: Adding a new requirement to the product backlog.

Keeping the product backlog up to date will allow you to quickly prioritize and add new requirements. With a current product backlog, you always understand the remaining scope. Chapter 9 has more information about prioritizing requirements.

Using agile artifacts for scope management

From the vision statement through the product increment, each agile artifact supports you in your scope management efforts. Progressively decompose, or break down, requirements as features rise to the top of the priority list. We talk about decomposition and progressive elaboration of requirements in Chapter 9.

Table 14-2 reveals how each agile artifact, including the product backlog, contributes to ongoing scope refinement.

TABLE 14-2 Agile Artifacts and Scope Management Roles

Artifact	Role in Establishing Scope	Role in Scope Change
Vision statement: A definition of the product's end goal. Chapter 9 has more about the vision statement.	Use the vision statement as a benchmark to judge whether features belong in the scope for the product.	When someone introduces new requirements, those requirements must support the product vision statement.
Product roadmap: A holistic view of product features that create the product vision. Chapter 9 has more about the product roadmap.	The product roadmap encompasses the scope of the product. Requirements at a feature level are good for business conversations about what it means to realize the product vision.	Update the product roadmap as requirements arise or change. The product roadmap provides visual communication of the new feature's inclusion in the product.

(continued)

TABLE 14-2 *(continued)*

Artifact	Role in Establishing Scope	Role in Scope Change
Release plan: A digestible mid-term target focused around a minimum set of marketable features. Chapter 10 has more about the release plan.	The release plan shows the scope of the current release. You may want to plan your releases by themes — logical groups of requirements.	Add new features that belong in the current release to the release plan. If the new user story doesn't belong in the current release, leave it on the product backlog for a future release.
Product backlog: A complete list of all known scope for the product. Chapters 9 and 10 offer more about the product backlog.	If a requirement is in the scope of the product vision, it is part of the product backlog.	The product backlog contains all scope changes. New, high-priority features push lower-priority features down on the product backlog.
Sprint backlog: The product backlog items and tasks in the scope of the current sprint. Chapter 10 has more about the sprint backlog.	The sprint backlog contains the product backlog items that are in scope for the current sprint.	The sprint backlog establishes what is allowed in the sprint. After the development team commits to the sprint goal in the sprint-planning meeting, only the development team can modify the sprint backlog.

What's Different about Agile Procurement?

Another part of agile product development is *procurement*, managing the purchase of services or goods needed to deliver the product's scope. Like scope, procurement is part of the investment side of product development.

Chapter 2 explains that the Agile Manifesto values *customer collaboration over contract negotiation.* This sets an important tone for procurement relationships.

Valuing customer collaboration more than contract negotiation doesn't mean that agile development efforts have no contracts: Contracts and negotiation are critical to business relationships. However, the Agile Manifesto sets forth the idea that a buyer and seller should work together to create products, and that the relationship between the two is more important than quibbling over ill-informed details and checking off contract items that may or may not ultimately be valuable to customers.

All 12 Agile Principles apply to procurement. However, the following seem to stand out the most when securing goods and services for product development:

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.
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.

Table 14-3 highlights the differences between procurement on traditional projects and procurement with agile product development.

TABLE 14-3 **Traditional versus Agile Procurement Management**

Procurement Management with Traditional Approaches	Procurement Management with Agile Approaches
The project manager and the organization are responsible for procurement activities.	The self-managing development team plays a larger part in identifying items needing procurement. The scrum master facilitates the acquisition of needed items for the development team.
Contracts with service providers often include provisions for fixed requirements, extensive documentation, a comprehensive project plan, and other traditional deliverables based on a waterfall lifecycle.	Contracts for agile product development are based on the evaluation of working functionality at the end of each sprint, not on fixed deliverables and documentation that may or may not contribute to delivering quality products.
Contract negotiation between buyers and sellers can sometimes be challenging. Because negotiation is often a stressful activity, it can damage the relationship between the buyer and the seller before work even starts on a project.	Scrum teams focus on keeping a positive, cooperative relationship between buyers and sellers from the start of the procurement process.
Switching vendors after a project starts can be costly and time-consuming because a new vendor must try to understand the old vendor's massive amount of work in progress.	Vendors provide completed, working functionality at the end of each sprint. If vendors change mid-sprint, the new vendor can immediately start developing requirements for the next sprint, avoiding a long, costly transition.

TECHNICAL STUFF

Both waterfall and scrum teams are interested in vendor success. Traditional project approaches were firm in their accountability for compliance, defining success as checking off documents and deliverables in a list. Agile approaches, by contrast, are firm in their accountability for end results, defining success as working functionality that achieves the customer's desired outcome.

The next section shows how to manage procurement.

Managing Agile Procurement

This section focuses on how scrum teams go through the procurement process: from determining need, selecting a vendor, and creating a contract through working with a vendor and closing out the contract at the end of a buyer-seller product development effort.

Determining need and selecting a vendor

With agile product development, procurement starts when the development team decides it needs a tool or the services of a third-party to create the product.

REMEMBER

Agile development teams are self-managing and self-organizing, and they get to make the decisions about what is best for maximizing development output. Self-management applies to all product management areas, including procurement. Find out more about self-managing teams in Chapters 7 and 16.

Development teams have a number of opportunities to consider outside goods and services:

- **Product vision stage:** The development team may start thinking about the tools and skills necessary to help reach the product vision. At this stage, it may be prudent to research needs but not begin the purchase process.
- **Product roadmap stage:** The development team starts to see specific features to create and may know some of the goods or services necessary to help create the product.
- **Release planning:** The development team knows more about the product and can identify specific goods or services that will help meet the next release goal. Start mobilizing the procurement at this time.
- **Sprint planning:** The development team is in the trenches of development and may identify urgent needs for the sprint.

- **Daily scrum:** Development team members state impediments. Procuring goods or services may help remove these impediments.
- **Throughout the day:** Development team members communicate with one another and collaborate on tasks. Specific needs may arise from the development team's conversations.
- **Sprint review meeting:** Product stakeholders may identify new requirements for future sprints that warrant procurement of goods or services.
- **Sprint retrospective:** The development team may discuss how having a specific tool or service could have helped the past sprint and suggest a purchase for future sprints.

After the development team determines it needs a good or service, the development team and the scrum master work with the product owner to procure any necessary funds. The product owner is responsible for managing scope against the budget, so the product owner is ultimately responsible for any purchases. The scrum master usually manages the vendor relationship on behalf of the scrum team after procurement is initiated with the vendor.

When procuring goods, the development team may need to compare tools and vendors before deciding on a purchase. When procuring goods, after you choose what to buy and where to get it, the process is usually straightforward: Make the purchase and take delivery.

Procuring services is usually a longer and more complex process than purchasing goods. Some agile-specific considerations for selecting a services vendor include the following:

- Whether the vendor can work in an agile product development environment and, if so, how much experience the vendor has with agile techniques
- Whether the vendor can work on-site with the development team
- Whether the relationship between the vendor and the scrum team is likely to be positive and collaborative

WARNING

The organization or company you work for may be subject to laws and regulations for choosing vendors. Companies involved in government work, for example, often need to gather multiple proposals and bids from companies for work that will cost more than a certain amount of money. Although your cousin or your friend from college might be the most qualified person to complete the work, you may run into trouble if you don't follow applicable laws. Check with your company's legal department if you're in doubt about how to streamline bloated processes.

After you choose a service vendor, you need to create a contract so that the vendor can start work. The next section explains how contracts work with agile product development.

Understanding cost approaches and contracts for services

After the development team and the product owner have chosen a vendor, they need a contract to ensure agreement on the services and pricing. To start the contract process, you should know about different pricing structures and how they work with agile product development. After you understand these approaches, you see how to create a contract.

Cost structures

When you're procuring services for agile product development, it is important to know the difference between *fixed price*, *fixed time*, *time and materials*, and *not to exceed*. Each approach has its own strengths in an agile setting:

- **Fixed price:** Starts out with a set budget. With a fixed price, a vendor works on the product and creates releases until that vendor has spent all the money in the budget or until you have delivered enough product features, whichever comes first.

  For example, if you have a $250,000 budget, and your vendor costs are $10,000 a week, the vendor's portion of the development will be able to last 25 weeks. Within those 25 weeks, the vendor creates and releases as much shippable functionality as possible.

- **Fixed time:** Has specific deadlines. For example, you may need to launch a product in time for the next holiday season, for a specific event, or to coincide with the release of another product. With a fixed time, you determine costs based on the cost of the vendor's team for the duration of development, along with any additional resource costs, such as hardware or software.

- **Time and materials:** Is more open-ended than fixed priced or fixed time. With time and materials, your work with the vendor lasts until enough product functionality is complete, without regard to total cost. You know the total cost at the end of development, after your stakeholders have determined that the product has enough features to call the product complete.

  For example, suppose your development costs $10,000 a week. After 20 weeks, the stakeholders feel that they have enough valuable product features, so your total cost is $200,000. If the stakeholders instead deem that they have enough value by the end of 10 weeks, the cost is half that amount, $100,000.

- **Not to exceed:** Is used when time and materials have a fixed-price cap.

THE FALLACY OF LOW-BALLING THE VENDOR

Trying to bully vendors into providing the lowest possible price is always a lose-lose proposition. Contractors in industries where projects always go to the lowest bidder have a saying: *Bid it low, and watch it grow.* It is common for vendors to provide a low price during a project's proposal process and then add multiple change orders until the client ends up paying as much or more than he or she would have for higher-priced offers.

Waterfall project management supports this practice by locking in scope and price at the project start, when you know the least about the project. Change orders — and their accompanying cost increases — are inevitable.

A better model is for the vendor and client to collaborate on defining the scope, within fixed cost and schedule constraints, as product development unfolds. Both parties can reap the benefits of what they learn during development, and you end up with a better product full of the highest-value functionality delivered and identified at the end of each sprint. Instead of trying to be a tough negotiator, be a good collaborator.

REMEMBER

Regardless of the cost approach, with agile product development concentrate on completing the highest-value product features first.

Contract creation

After you know the cost approach, the scrum master might help create a contract. Contracts are legally binding agreements between buyers and sellers that set expectations about work and payment.

The person responsible for creating contracts differs by organization. In some cases, a person from the legal or procurement department drafts a contract and then asks the scrum master to review it. In other cases, the opposite occurs: The scrum master drafts the contract and has a legal or procurement expert review it.

Regardless of who creates the contract, the scrum master usually acts on behalf of the scrum team to do any of the following: Initiate the contract creation, negotiate the contract details, and route the contract through any necessary internal approvals.

The agile approach of placing value on collaboration over negotiation is a key to maintaining a positive relationship between a buyer and a seller while creating and negotiating a contract. The scrum master works closely with the vendor and communicates openly and often with the vendor throughout the contract creation process.

WARNING

The Agile Manifesto does *not* imply that contracts are unnecessary ("customer collaboration over contract negotiation"). Regardless of the size of your company or organization, you should create a contract between your company and your vendor for services. Skipping the contract can leave buyers and sellers open to confusion about expectations, can result in unfinished work, and can even lead to legal problems. Contracts should develop through customer collaboration, implying a non-adversarial approach.

At the very least, most contracts have legal language describing the parties and the work, the budget, the cost approach, and payment terms. A contract for agile product development may also include the following:

- **A description of the work that the vendor will complete:** The vendor may have its own product vision statement, which may be a good starting point to describe the vendor's work. You may want to refer to the product vision statement in Chapter 9.
- **Agile approaches that the vendor may use:** They may include
  - Meetings that the vendor will attend, such as the daily scrum, sprint planning, sprint review, and sprint retrospective meetings
  - Delivery of working functionality at the end of each sprint
  - The definition of done (discussed in Chapter 11): Work that is developed, tested, integrated, and documented, per an agreement between the product owner, the development team, and the scrum master
  - Artifacts that the vendor will provide, such as a sprint backlog with a burndown chart for visualizing progress
  - People whom the vendor will assign to the product, such as the development team
  - Where the vendor will work, such as on-site at your company
  - Whether the vendor will work with a scrum master from the vendor's organization or from your organization
  - A definition of what may constitute the end of the engagement: The end of a fixed budget or fixed time, or enough complete, working functionality
- **For a vendor that doesn't use an agile approach, a description of how the vendor and the vendor's work will integrate with the client's development team and sprints**

This is not a comprehensive list; contract items vary by product and organization.

The contract will likely go through a few rounds of reviews and changes before the final version is complete. One way to clearly communicate changes and maintain a good relationship with a vendor is to speak with the vendor each time you propose a change. If you email a revised contract, follow up with a call to explain what you changed and why, to answer any questions, and to discuss any ideas for further revision. Open discussion helps the contract process to be positive.

If anything substantial about the vendor's services changes during contract discussions, it is a good idea for the product owner or the scrum master to review those changes with the development team. The development team especially needs to know and provide input about changes to the service the vendor will provide, the vendor's approach, and the people on the vendor's team.

TIP

It is quite likely that your company and the vendor will require reviews and approvals by people outside their respective teams. People who review contracts might include high-level managers or executives, procurement specialists, accounting people, and company attorneys. This differs by organization; the scrum master needs to ensure that anyone who needs to read the contract does so.

Now that you understand a little about how to select a vendor and create a contract, it's time to look at how to work with a vendor.

Working with a vendor

How you work with a vendor on an agile product development effort depends in part on the vendor team's structure. In an ideal situation, vendor teams are fully integrated with the client's organization. The vendor's team members are collocated with the client's scrum team. Vendor team members work as part of the client's development team for as long as necessary.

REMEMBER

Just because the vendor is not part of your organization, it doesn't mean the vendor's team members are not part of the scrum team. Because you want your vendor integrated as part of your development team, scrum teams include vendor team members in all scrum events.

Vendor teams also can be integrated but dislocated. If the vendor can't work on-site at the client's company, it can still be part of the client's scrum team. Chapter 16 has more information on team dynamics.

If a vendor can't be collocated, or if the vendor is responsible for a discrete, separate part of the product, the vendor may have a separate scrum team. The vendor's scrum team works on the same sprint schedule as the client's scrum team. See Chapters 15 and 19 to find out how to work with more than one scrum team.

If a vendor doesn't use agile product management processes, the vendor's team works separately from the client's scrum team, outside the sprints, and on its own schedule. The vendor's traditional project manager helps ensure that the vendor can deliver its services when the development team needs them. The client's scrum master may need to step in if the vendor's processes or timeline becomes a roadblock or a disruption for the development team. See the "Managing product development with dislocated teams" section in Chapter 16 for information about working with non-scrum teams.

Vendors may provide services for a defined amount of time, or for the life of the development effort. After the vendor's work is complete, the contract is closed.

Closing a contract

After a vendor completes work on a contract, the client's scrum master usually has some final tasks to close the contract.

If development finishes normally, according to the contract terms, the scrum master may want to acknowledge the end of the contract in writing. If the contract stipulates time and materials, the scrum master should definitely end it in writing to ensure that the vendor doesn't keep working on lower-priority requirements — and billing for them.

Depending on the organizational structure and the contract's cost structure, the scrum master may be responsible for notifying the client's company accounting department after work is complete to ensure that the vendor is paid properly.

If product development finishes before the contract dictates the end (enough value has been delivered to redeploy capital on a new product development effort), the scrum master needs to notify the vendor in writing and follow any early termination instructions from the contract.

TIP

End the engagement on a positive note. If the vendor did a good job, the scrum team may want to acknowledge the people on the vendor's team at the sprint reviews. Everyone could potentially work together again, and a simple, sincere "thank-you" can help maintain a good relationship for future development.

IN THIS CHAPTER

- Understanding what's unique about time management with agile product development
- Recognizing how cost management is different with agile product development

Chapter 15

Managing Time and Cost

Managing time and controlling costs are key aspects of managing agile product development. In this chapter, you see agile approaches to time and cost management. You find out how to use a scrum team's development speed to determine time and cost and how to increase development speed to lower your product's time and cost.

What's Different about Agile Time Management?

With agile product development, *time* refers to the processes that ensure timely completion and the effective use of time. To understand agile time management, it helps to review some of the Agile Principles we discuss in Chapter 2:

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

Table 15-1 shows some of the differences between time management on traditional projects and agile product development.

TABLE 15-1 Traditional versus Agile Time Management

Time Management with Traditional Approaches	Time Management with Agile Approaches
Fixed scope directly drives the schedule.	Scope is not fixed. Time can be fixed, and development teams can create the requirements that will fit into a specific time frame.
Project managers determine time based on the requirements gathered at the beginning of the project.	During development, scrum teams assess and reassess how much work they can complete in a given time frame.
Teams work at one time in phases on all project requirements, such as requirements gathering, design, development, testing, and deployment. No schedule difference exists between critical requirements and optional requirements.	Scrum teams work in sprints and complete all the work on the highest-priority, highest-value requirements first.
Teams do not start actual product development until later in the project, after the requirements-gathering and design phases are complete.	Scrum teams start product development in the first sprint.
Time is more variable on traditional projects.	Timeboxed sprints stay stable, enabling predictability.
Project managers try to predict schedules at the project start, when they know little about the product.	Scrum teams determine long-range schedules on actual development performance in sprints. Scrum teams adjust time estimates throughout development as they learn more about the product and the development team's speed, or *velocity.* You learn more about velocity later in this chapter.

REMEMBER

A fixed-schedule or fixed-price approach has a lower risk with agile techniques because agile development teams always deliver the highest-priority functionality within the time or budget constraints.

A big benefit of agile time management techniques is that scrum teams can deliver products much earlier than traditional project teams. For example, starting development earlier and completing functionality in iterations often allow scrum teams that work with our company, Platinum Edge, to bring value to the market 30 percent to 40 percent faster.

TIP

The reason agile product development efforts finish sooner isn't complicated; they simply start development sooner.

In the next section, find out how to manage time.

Managing Agile Schedules

Agile practices support both strategic and tactical schedules and time management:

- Your early planning is strategic in nature. The high-level requirements in the product roadmap and the product backlog can help you get an early idea of the overall schedule. Find out how to create a product roadmap and product backlog in Chapter 9.
- Your detailed planning for each release and at each sprint is tactical. Read more about release planning and sprint planning in Chapter 10.
  - At release planning, you can plan your release to match a specific date, with minimal marketable features.
  - You also can plan your release with enough time to create a specific set of features.
  - During each sprint planning meeting, in addition to selecting the scope for the sprint, the development team estimates the time, in hours, to complete individual tasks for each of that sprint's requirements. Use the sprint backlog to manage detailed time allocations throughout the sprint.
- After development is underway, use the scrum team's velocity (development speed) to fine-tune your scheduling. We discuss velocity in the next section.

REMEMBER

In Chapter 10, we describe planning releases for *minimal marketable features*, the smallest group of product functionality providing enough value that you can effectively deploy and promote in the marketplace.

To determine how much functionality an agile development team can deliver within a set amount of time, you need to know your development team's velocity. In the next section, you take a look at how to calculate velocity, how to use velocity as a planning input, and ways you might increase velocity throughout product development.

Introducing velocity

One of the most important considerations about time management with agile product development is the use of velocity, a powerful empirical data set that scrum teams use to forecast long-term timelines. *Velocity*, in agile terms, is a development team's work speed. In Chapter 9, we describe estimation of effort for implementing requirements, or user stories, in story points. You measure velocity by the number of user story points that the development team completes, meeting its definition of done, in each sprint.

DETERMINING PRODUCT DEVELOPMENT DURATION

A few factors determine how long agile product development should take:

- **Assigned deadline:** For business reasons, scrum teams may want to set a specific end date. For example, you may want to get a product to market for a specific shopping season or to coincide with the timing of a competitor's product release. In that case, you set a specific end date, and create as much shippable functionality as possible from the start until the end date.
- **Budget considerations:** Scrum teams may also have budget considerations that affect the amount of time product development will last. For example, if you have a $1,600,000 budget, and your team costs $20,000 a week to run, your development effort will last 80 weeks. You'll have 80 weeks to create and release as much shippable functionality as possible. A fixed date can set a limit for your investment decision.
- **Functionality completed:** Agile product development may also last only until enough functionality is complete to achieve some specified amount of value delivered. Scrum teams may run sprints until the requirements with the highest value are complete, and then determine that the lower-value requirements — the ones that few people will use or that will not generate much revenue — aren't necessary.

REMEMBER

A *user story* is a simple description of a product requirement, identifying what a requirement must accomplish, and for whom. User story points are relative numbers that describe the amount of effort necessary to develop and implement a user story. Chapter 10 delves into the details of defining user stories and estimating the effort using story points.

When you know the development team's velocity, you can use it as a long-range planning tool. Velocity can help you forecast how long the scrum team will take to complete a certain number of requirements and how much development may cost.

In the next section, you dive into velocity as a tool for time management. You see how scope changes affect a timeline. You also find out how to work with multiple scrum teams and review agile artifacts for time management.

Monitoring and adjusting velocity

After development starts, the scrum team starts to monitor its velocity at the end of each sprint. You use velocity for long-term schedule and budget planning as well as for sprint planning.

In general, people are good at planning and estimating in the short term, so identifying hours for tasks in an upcoming sprint works well. At the same time, people are often terrible at estimating distant tasks in absolute terms such as hours. Tools such as relative estimating and velocity, which are based on actual performance, are more accurate measurements for longer-term planning.

Velocity is a good trending tool. You can use it to determine future timelines because the activities and development time within sprints is the same from sprint to sprint.

WARNING

Velocity is a post-sprint fact, not a goal. Avoid attempting to guess or commit to a certain velocity before development starts or in the middle of a sprint. You'll only set unrealistic expectations about how much work the team can complete. If velocity turns into a target rather than a past measurement, scrum teams may be tempted to exaggerate estimated story points to meet that target, rendering estimates and velocity meaningless. Instead, use the scrum team's actual velocity to forecast how much longer the development may take and cost. Also focus on increasing velocity by removing constraints identified during the sprint and at the sprint retrospective. Agile product development is pulled, not pushed.

In the next section, you see how to calculate velocity, how to use velocity to predict a schedule, and how to increase your scrum team's velocity.

Calculating velocity

At the end of each sprint, the scrum team looks at the requirements it has finished and adds up the number of story points associated with those requirements. The total number of completed story points is the scrum team's velocity for that sprint. After the first few sprints, you will start to see a trend and will be able to calculate the average velocity.

WARNING

Because velocity is a number, managers and executives may be tempted to use velocity as a performance metric for compensating and comparing teams. Velocity is *not* a performance metric, is team-specific, and should not be used outside the scrum team. It is no more than a planning tool scrum teams can use to forecast remaining work. Principle 7 reminds us that "working software (or working product) is the primary measure of progress." Not velocity.

The *average velocity* is the total number of story points completed, divided by the total number of sprints completed. For example, if the development team's velocity was

Sprint 1 = 15 points

Sprint 2 = 13 points

Sprint 3 = 16 points

Sprint 4 = 20 points

your total number of story points completed will be 64. Your average velocity will be 16: 64 story points divided by 4 sprints.

You don't have to run many sprints before you have real data to forecast. In fact, after you've run only one sprint, you'll have your first empirical data point of the scrum team's velocity. Of course, after you run more sprints, you'll have more empirical data points to use to fine-tune your forecast — based on reality rather than theory.

Using velocity to estimate the development timeline

When you know your velocity, you can determine how long product development will last. Follow these steps:

1. **Add up the number of story points for the remaining requirements in the product backlog.**
2. **Determine the number of sprints you'll need by dividing the number of story points remaining in the product backlog by the velocity:**
   - To get a pessimistic estimate, use the lowest velocity the development team has accomplished.
   - To get an optimistic estimate, use the highest velocity the development team has accomplished.
   - To get a most likely estimate, use the average velocity the development team has accomplished.

TIP

Using this empirical data — actual output speed — a product owner can give stakeholders a range of release outcomes, and they can work together to make business prioritization decisions early. These decisions might include whether there is a need to spin up an additional scrum team to develop more scope items, adjust market release dates, or request additional budget. Even better, the product owner may realize early which features to abandon.

3. **Determine how much time it will take to complete the story points in the product backlog by multiplying sprint length by the number of remaining sprints.**

   For example, assume that

   - Your remaining product backlog contains 400 story points.
   - Your development team velocity averages 20 story points per sprint.

   How many more sprints will your product backlog need? Divide the number of story points by your velocity, and you get your remaining sprints. In this case, 400/20 = 20.

   If you're using two-week sprints, your product development will last 40 weeks.

After the scrum team knows its velocity and the number of story points for the requirements, you can use the velocity to determine how long any given group of requirements will take to create. For example:

- You can calculate the time an individual release may take if you have an idea of the number of story points that will go into that release. At the release level, your story point estimates will be more high level than at the sprint level. If you're basing your release timing on delivering specific functionality, your release date may change as you refine your user stories and estimates throughout product development.
- You can calculate the time you need for a specific group of user stories — such as all high-priority stories or all stories relating to a particular theme — by using the number of story points in that group of user stories.

TIP

Another approach to long-range planning is the #noestimates movement, which advocates decomposing product backlog items into equally sized items, rather than estimating items of different story point sizes. *Velocity* in this case refers to how many product backlog items can be accomplished each sprint. Taking the total number of product backlog items in the product backlog and dividing it by the number of product backlog items the team can complete in a sprint (velocity) results in a schedule of how many sprints it will take to complete the product.

Velocity differs from sprint to sprint. In the first few sprints, when the product is new, the scrum team will typically have a low velocity. As product development progresses, velocity should increase because the scrum team will have learned more about the product and will have matured as a team working together. Setbacks within specific sprints can temporarily decrease velocity from time to time, but agile processes such as the sprint retrospective can help the scrum team ensure that those setbacks are temporary.

TIP

With new teams, velocity will vary considerably from sprint to sprint. Velocity will become more consistent over time, as long as the scrum team members remain consistent. In Chapter 8, we discuss the value of creating long-lived and even permanent scrum teams.

Scrum teams can also increase their velocity, making product development shorter and less costly. In the next section, you find ways to increase velocity in each consecutive sprint.

Increasing velocity

If a scrum team has a product backlog with 400 story points and an average velocity of 20 story points, product development will last 20 sprints — 40 weeks, with 2-week sprints. But what if the scrum team could increase its velocity?

- Increasing the average velocity to 23 story points per sprint would mean 17.39 sprints. If you round that up to 18 sprints, the same product development effort would last 36 weeks.
- An average velocity of 26 would take about 15.38, rounded up to 16 sprints, or 32 weeks.
- An average velocity of 31 would take about 12.9, rounded up to 13 sprints, or 26 weeks.

As you can see, even marginally increasing velocity can save a good deal of time and, consequently, money.

Velocity can naturally increase with each sprint, as the scrum team finds its rhythm of working together. However, opportunities exist to also raise velocity, past the common increases that come with time. Everyone on a scrum team plays a part in helping get higher velocity with every successive sprint:

- **Remove roadblocks.** One way to increase velocity is to quickly remove roadblocks, or impediments. Roadblocks are anything that keeps a development team member from working to full capacity. By definition, roadblocks can decrease velocity. Clearing roadblocks as soon as they arise increases velocity by helping the scrum team to be fully functional and productive. Find out more about removing impediments in Chapter 11.
- **Avoid roadblocks.** The best way to increase velocity is to strategically create ways to avoid or prevent roadblocks in the first place. By knowing — or learning about — the processes and the specific needs of groups your team will work with, you can head off roadblocks before they arise.

- **Eliminate distractions.** Another way to increase velocity is for the scrum master to protect the development team from distractions. By making sure people don't request work outside the sprint goal from the development team — even tasks that might take a small amount of time — the scrum master will be able to help keep the development team focused on the sprint.

  Having a dedicated scrum master who continually helps prevent and remove constraints for the scrum team will result in continually increasing velocity. The value of a dedicated scrum master is quantifiable.

REMEMBER

- **Solicit input from the team.** Finally, everyone on the scrum team can provide ideas for increasing velocity in the sprint retrospective meeting. The development team knows its work the best, and may have ideas on how to improve output. The product owner may have insights into the requirements that can help the development team work faster. The scrum master will have seen any repetitive roadblocks and can discuss how to prevent the roadblocks in the first place.

TIP

Increasing velocity is valuable, but remember that you may not see changes overnight. Scrum team velocity often has a pattern of slow increases, some big velocity jumps, a flat period, and then slow increases again as the scrum team identifies, experiments, and corrects constraints that are holding it back. As discussed in Chapter 4, they use the scientific method to consistently improve their team.

PREVENTING ROADBLOCKS

One development team we worked with needed feedback from its company's legal department but had not been able to get a response via email or voicemail. In a daily scrum meeting, one of the development team members stated this lack of response as a roadblock. After the scrum meeting was over, the scrum master walked over to the legal department and found the right person to work with. After talking to that person, the scrum master found out that her email was constantly flooded with requests, and her voicemail was not much better.

The scrum master then suggested a process for future legal requests: Moving forward, the development team members could walk over to the legal department with requests and get feedback right there, in person, immediately. The new process took only a few minutes, but saved days on turnaround from the legal department, effectively preventing similar roadblocks in the future. Finding ways to prevent roadblocks helps increase the scrum team's velocity.

Consistency for useful velocity

Because velocity is a measure of work completed in terms of story points, it's an accurate indicator and predictor of performance only when you use the following practices:

- **Consistent sprint lengths:** Each sprint should last the same amount of time throughout the life of product development. If sprint lengths are different, the amount of work the development team can complete in each sprint will be different, and velocity won't be relevant in predicting the remaining time.
- **Consistent work hours:** Individual development team members should work the same number of hours in each sprint. If Sandy works 45 hours in one sprint, 23 in another, and 68 in yet another, she will naturally complete a different amount of work from sprint to sprint. However, if Sandy always works the same number hours in one sprint, her velocity will be comparable between sprints.
- **Consistent development team members:** Different people work at different rates. Tom might work faster than Bob, so if Tom works on one sprint and Bob works on the next sprint, the velocity of Tom's sprint will not be a good prediction for Bob's sprint.

When sprint lengths, work hours, and team members remain consistent throughout development, you can use velocity to truly know whether development speed is increasing or decreasing and to accurately estimate the timeline. For this reason, scrum teams adhere to Principle 8: "Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely."

WARNING

Performance does not scale linearly with available time. For example, if you have two-week sprints with 20 story points per sprint, going to three-week sprints does not guarantee 30 story points. The new sprint length will generate an unknown change in velocity.

Although changing sprint lengths does introduce variance into a scrum team's velocity and projections, we rarely discourage scrum teams from decreasing their sprint lengths (from three weeks to two, or from two weeks to one) because shorter feedback loops allow scrum teams to react faster to customer feedback, enabling them to deliver more value to their customer. However, changing sprint lengths always comes with the same caution: Velocity does not scale linearly in the opposite direction either, and scrum teams will have to establish a new velocity for their shorter sprint before their projections will become reliable again.

REMEMBER

Shorter is better. If you go from a two-week sprint to a three-week sprint, you have to wait three weeks before you have your first empirical data point. In that same three-week period, going from a two-week sprint to a one-week sprint will give you three new empirical data points.

When you know how to accurately measure and increase velocity, you have a powerful tool for managing time and cost. In the next section, we talk about how to manage a timeline in an ever-changing agile environment.

Managing scope changes from a time perspective

Scrum teams welcome changing requirements at any time throughout development, which means scope reflects the real priorities of the business. It is "requirements Darwinism" at its purest — development teams complete requirements of highest priority first. Fixed sprint lengths force out requirements that sound like good ideas in theory but never win the "either this requirement or that requirement" contest.

New requirements may have no effect on a timeline; you just have to prioritize. Working with the stakeholders, the product owner can determine to develop only the requirements that will fit in a certain window of time or budget. The priority ranking of items in the product backlog determines which requirements are important enough to develop. The scrum team can guarantee completing higher-priority requirements. The lower-priority requirements might be part of another product or may never be created.

REMEMBER

In Chapter 14, we discuss how to manage scope changes with the product backlog. When you add a new requirement, you prioritize that requirement against all other items in your product backlog and add the new item into the appropriate spot in the product backlog. This may move other product backlog items down in priority. If you keep your product backlog and its estimates up-to-date as new requirements arise, you'll always have a good idea of the timeline, even with constantly changing scope.

On the other hand, the product owner and the stakeholders may determine that all the requirements in the product backlog, including new requirements, are useful enough to include in the product. In this case, you extend the development end date to accommodate the additional scope, increase velocity, or divide the scope among multiple scrum teams that will work simultaneously on different product features. Learn more about multi-team product development in Chapter 19.

Scrum teams often make schedule decisions about lower-priority requirements toward the end of development. The reasons for these just-in-time decisions are because marketplace demands for specific scope items change, and also because velocity tends to increase as the development team gets into a rhythm. Changes in velocity increase your predictions about how many product backlog items the development team can complete in a given amount of time. With agile product

development, you wait until the last responsible moment — when you know the most about the question at hand — to make decisions you'll be committed to for the rest of your work.

The next section shows you how to work with more than one scrum team on a common goal.

Managing time by using multiple teams

For larger development efforts, multiple scrum teams working in parallel may be able to complete development in a shorter time frame.

You may want to use multiple scrum teams if

- Your development effort is very large and will require more than a single development team of nine or fewer development team members to complete.
- Your development effort has a specific end date that you must meet, and the scrum team's velocity will not be sufficient to complete the most valuable requirements by that end date.

REMEMBER

The ideal size for a development team is no less than three and no more than nine people. Groups of more than nine people start to build silos, and the number of communication channels makes self-management more difficult. (In some cases, we've seen these issues in teams smaller than nine.) When your product development requires more development team members than can effectively communicate, it may be time to consider using multiple scrum teams.

In Chapter 19, we show you several techniques for scaling product development work across multiple teams.

Using agile artifacts for time management

The product roadmap, product backlog, release plan, and sprint backlog all play a part in time management. Table 15-2 shows how each artifact contributes to time management.

In the next sections, you dive into cost management for agile product development. Cost management is directly related to time management. You compare traditional approaches to cost management to those with agile product development. You find out how to estimate costs and how to use velocity to forecast your long-term budget.

TABLE 15-2 Agile Artifacts and Time Management

Artifact	Role in Time Management
Product roadmap: The product roadmap is a prioritized, holistic view of the high-level requirements that support the product's vision. Find more about the product roadmap in Chapter 9.	The product roadmap is a strategic look at the overall product priorities. Although the product roadmap likely will not have specific dates, it will have general date ranges for groups of functionality and will allow an initial framing for bringing the product to market.
Product backlog: The product backlog is a complete list of all currently known product requirements. Find more about the product backlog in Chapters 9 and 10.	The requirements in your product backlog will have estimated story points. After you know your development team's velocity, you can use the total number of story points in the product backlog to determine a realistic end date.
Release plan: The release plan contains a release schedule for a minimum set of requirements. Find more about the release plan in Chapter 10.	The release plan will have a target release date for a specific goal supported by a minimal set of marketable features. Scrum teams plan and work on only one release at a time.
Sprint backlog: The sprint backlog contains the requirements and tasks for the current sprint. Find more about the sprint backlog in Chapter 10.	During your sprint planning meeting, you estimate individual tasks in the backlog in hours. At the end of each sprint, you take the total completed story points from the sprint backlog to calculate your development team's velocity for that sprint.

What's Different about Agile Cost Management?

Cost is a product's financial budget. When you work with an agile product development approach, you focus on value, exploit the power of change, and aim for simplicity. Agile Principles 1, 2, and 10 state the following:

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.
10. Simplicity — the art of maximizing the amount of work not done — is essential.

Because of this emphasis on value, change, and simplicity, agile product development takes a different approach to budget and cost management than traditional projects. Table 15-3 highlights some of the differences.

TECHNICAL STUFF

When costs increase, sponsors sometimes find themselves in a kind of hostage situation. A waterfall approach does not call for any complete product functionality until the end of a project. Because traditional approaches to development are all-or-nothing proposals, if costs increase and stakeholders don't pay more for

the product, they will not get *any* finished requirements. The incomplete product becomes a kidnapped hostage; pay more, or get nothing.

TABLE 15-3 Traditional versus Agile Cost Management

Cost Management with Traditional Approaches	Cost Management with Agile Approaches
Cost, like time, is based on fixed scope.	Schedule, not scope, has the biggest effect on cost. You can start with a fixed cost and a fixed amount of time, and then complete requirements as potentially shippable functionality that fit into your budget and schedule.
Organizations estimate project costs and fund projects before the project starts.	Product owners often secure funding after the product roadmap stage is complete. Some organizations even fund one release at a time; product owners will secure funding after completing release planning for each release.
New requirements mean higher costs. Because project managers estimate costs based on what they know at the project start, which is very little, cost overruns are common.	Scrum teams can replace lower-priority requirements with new, equivalently sized high-priority requirements with no effect on time or cost.
Scope bloat (see Chapter 12) wastes large amounts of money on features that people simply do not use.	Because agile development teams complete requirements by priority, they concentrate on creating only the product features that users need, whether those features are added on day 1 or day 100 of development.
Projects cannot generate revenue until the project is complete.	Scrum teams can release working, revenue-generating functionality early, creating a self-funding product.

In the following sections, you find out about cost approaches to agile product development, how to estimate costs, how to control your budget, and how to lower costs.

Managing Agile Budgets

With agile product development, cost is mostly a direct expression of time. Because scrum teams consist of full-time, dedicated team members, they have a set team cost — generally expressed as an hourly or fixed rate per person — that should be the same for each sprint. Consistent sprint lengths, work hours, and team members enable you to accurately use velocity to predict development speed. Once you use velocity to determine remaining sprints — that is, how long your product development effort will be — you can know how much your product development effort will cost.

Cost also includes the cost for resources like hardware, software, licenses, and any other supplies you might need to complete development.

In this section, you find out how to create an initial budget and how to use the scrum team's velocity to determine long-range costs.

Creating an initial budget

To create the budget for your product, you need to know the cost for your scrum team, per sprint, and the cost for any additional resources you need to complete development.

Typically, you calculate the cost for your scrum team by using an hourly rate for each team member. Multiply each team member's hourly rate by his or her available hours per week by the number of weeks in your sprints to calculate your scrum team's per-sprint cost. Table 15-4 shows a sample budget for a scrum team — the product owner, five development team members, and the scrum master — for a two-week sprint.

TABLE 15-4 Sample Scrum Team Budget for a Two-Week Sprint

Team Member	Hourly Rate	Weekly Hours	Weekly Cost	Sprint Cost (2 Weeks)
Don	$80	40	$3,200	$6,400
Peggy	$70	40	$2,800	$5,600
Bob	$70	40	$2,800	$5,600
Mike	$65	40	$2,600	$5,200
Joan	$85	40	$3,400	$6,800
Tommy	$75	40	$3,000	$6,000
Pete	$55	40	$2,200	$4,400
Total		280	$20,000	$40,000

The cost for additional resources will vary. In addition to scrum team member costs, take the following into account when determining your costs:

- Hardware costs
- Software, including license costs

- Hosting costs
- Training costs
- Miscellaneous team expenses, such as additional office supplies, team lunches, travel costs, and the price of any tools you may need

These costs may be one-time costs, rather than per-sprint costs. We suggest separating these costs in your budget; as you see in the next section, you need your cost for each sprint to determine the cost for development. (To keep calculations simple throughout this chapter, we assume that the per-sprint cost is $40,000, which includes scrum team member costs as well as any additional resources, such as those just listed.)

TIP

Resources typically refer to inanimate objects, not people. Resources need to be managed. When discussing resources, refer to people as *team members, talent,* or just *people.* This issue may seem minor, but the more you focus on individuals and interactions over processes and tools, even in the details, the more your mindset will change to think and be more agile.

Creating a self-funding product

A big benefit of agile product development is the capability to become self-funding. Scrum teams deliver working functionality at the end of each sprint and make that functionality available to the marketplace at the end of each release cycle. If your product is an income-generating product, you could use revenue from early releases to help fund the rest of product development.

For example, an ecommerce website that holds all completed work for one big release after six months instead of releasing functionality incrementally may be able to generate $100,000 per month. However, the same ecommerce website might be able to start generating $15,000 a month in sales after the first release of some essential valuable functionality, $40,000 a month after the second incremental release of valuable functionality, and so on. Tables 15-5 and 15-6 compare income on a sample traditional project to the income from a self-funding agile product development effort.

In Table 15-5, the project created $100,000 in income after six months of development. Now compare the income in Table 15-5 to the income generated in Table 15-6.

In Table 15-6, the product generated income with the first release. By the end of six months, the product had generated $330,000 — $230,000 more than the project in Table 15-5.

TABLE 15-5 Income from a Traditional Project with a Final Release after Six Months

Month	Income Generated	Total Project Income
January	$0	$0
February	$0	$0
March	$0	$0
April	$0	$0
May	$0	$0
June	$0	$0
July	$100,000	$100,000

TABLE 15-6 Income with Monthly Releases and a Final Release after Six Months

Month/Release	Income Generated	Total Income
January	$0	$0
February	$15,000	$15,000
March	$25,000	$40,000
April	$40,000	$80,000
May	$70,000	$150,000
June	$80,000	$230,000
July	$100,000	$330,000

Using velocity to determine long-range costs

The "Using velocity to estimate the development timeline" section, earlier in this chapter, shows you how to determine how much time product development will take, using the scrum team's velocity and the remaining story points in the product backlog. You can use the same information to determine the cost of developing the entire product or just the current release.

After you know the scrum team's velocity, you can calculate the cost for the remainder of product development.

In the velocity example from earlier in this chapter, where your scrum team velocity averages 16 story points per sprint, your product backlog contains 400 story points, and your sprints are 2 weeks long, your product will take 25 sprints, or 50 weeks, to complete.

To determine the remaining cost of product development, multiply the cost per sprint by the number of sprints the scrum team needs to complete the product backlog.

If your scrum team cost is $40,000 per sprint and you have 25 sprints left, your remaining cost for development will be $1,000,000.

In the next sections, you find out different ways to lower your costs.

Lowering cost by increasing velocity

In the time management section of this chapter, we talk about increasing the scrum team's velocity. Using the examples from the earlier section, and the $40,000 per two-week sprint from Table 15-4, increasing velocity could reduce your costs, as follows:

- If the scrum team increases its average velocity from 16 to 20 story points per sprint
  - You will have 20 remaining sprints.
  - Your remaining development will cost $800,000, saving you $200,000.
- If the scrum team increases its velocity to 23 story points
  - You will have 18 remaining sprints.
  - Your remaining development will cost $720,000, saving you an additional $80,000.
- If the scrum team increases its velocity to 26 story points
  - You will have 16 remaining sprints.
  - Your remaining development will cost $640,000, an additional $80,000 savings.

As you can see, increasing the scrum team's velocity by removing impediments can provide real savings. See how to help the scrum team become more productive in the "Increasing velocity" section, earlier in this chapter.

Lowering cost by reducing time

You can also lower your costs by not completing lower-priority requirements, thus lowering the number of sprints you need. Because completed functionality is delivered with each sprint with agile product development, the stakeholders can make a business decision to end development when the cost of future development is higher than the value of that future development.

Stakeholders can then use the remaining budget from the canceled development effort to develop something even more valuable. The practice of moving the budget from one development effort to another is called *capital redeployment.*

To determine when development should end based on cost, you need to know

- The business value (V) of the remaining requirements in the product backlog
- The actual cost (AC) of the work it will take to complete the requirements in the product backlog
- The opportunity cost (OC), or the value of having the scrum team work on something new

When $V < AC + OC$, product development can stop because the cost you'll sink into the remaining product requirements will be more than the value you will receive from them.

For example, consider the following about a company using agile product development techniques:

- The remaining features in the product backlog will generate $100,000 in income (V = $100,000).
- It will take three sprints with a cost of $40,000 per sprint to create those features, a total of $120,000 (AC = $120,000).
- The scrum team could be working on a new product initiative that would generate $150,000 after three sprints, minus the scrum team's cost (OC = $150,000).
- The product value, $100,000, is less than the actual costs plus opportunity costs, or $270,000. This would be a good time to stop and redeploy capital to the next product initiative.

The opportunity for capital redeployment sometimes arises in emergencies, when an organization needs the scrum team to pause or pivot to something more valuable or both. Sponsors sometimes evaluate the remaining value and cost before restarting new product development.

WARNING

Pausing and pivoting can be expensive. The costs associated with demobilization and remobilization — saving work in progress, documenting current state, debriefing paused team members, retooling for the new development, briefing team members on the new development, learning new skills required on the new development — can be significant and should be evaluated before making the decision to pause development that may need to be remobilized again in the future. $V < AC + OC$ can help with this decision.

Sponsors may also compare the product backlog value to remaining development costs throughout product development, so they know just the right time to end development and receive the most value.

Determining other costs

Similar to time management, after you know the scrum team's velocity, you can determine the cost of development. For example:

- You can calculate the cost for an individual release if you have an idea of the number of story points that will go into that release. Divide the number of story points in the release by the scrum team's velocity to determine how many sprints will be required. At the release, your story point estimates will be more high-level than at the sprint, so your costs may change, depending on how you determine your release date.
- You can calculate the cost for a specific group of user stories, such as all high-priority stories or all stories relating to a particular theme, by using the number of story points in that group of user stories.

Using agile artifacts for cost management

You can use the product roadmap, release plan, and sprint backlog for cost management. Refer to Table 15-2 to see how each artifact helps you measure and evaluate development time and costs.

Time and cost forecasts based on the team's empirically proven development pace are more accurate than forecasts based on hypotheticals or what the team hopes to accomplish.

IN THIS CHAPTER

- Recognizing how agile principles change team dynamics
- Understanding how communication differs with agile product development
- Seeing how communication works

Chapter 16

Managing Team Dynamics and Communication

Team dynamics and communication are significant parts of agile product development. In this chapter, you find out about traditional and agile approaches to teams and communication. You see how a high value on individuals and interactions makes agile teams great teams to work on. You also find out how face-to-face communication helps make agile product development successful.

What's Different about Agile Team Dynamics?

What makes an agile team unique? The core reason agile teams are different from traditional teams is their team dynamics. The Agile Manifesto (refer to Chapter 2) sets the framework for how agile team members work together: The very first item of value in the manifesto is *individuals and interactions* over processes and tools.

The following agile principles, also from Chapter 2, support valuing people on the team and how they work together:

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

The agile principles apply to many different product management areas. You see some of these principles repeated in different chapters of this book.

REMEMBER

With agile product development, the development team contains the people who do the physical work of creating the product. The scrum team contains the development team, plus the product owner and the scrum master. The product team is the scrum team and your stakeholders. Everyone on the scrum team has responsibilities related to self-management.

Table 16-1 shows some differences between team management with traditional projects and agile product development.

TIP

We avoid the term *resources* when referring people. Referring to people and equipment with the same term is the beginning of thinking of team members as interchangeable objects that can be swapped in and out. Resources are things, utilitarian and expendable. The people on your team are human beings, with emotions, ideas, and priorities inside and outside. People can learn and create and grow throughout their experience of working together. Respecting your fellow team members by referring to them as *people* instead of *resources* is a subtle but powerful way to reinforce the fact that people are at the core of an agile mindset.

The following sections discuss how working with a dedicated, cross-functional, self-organizing, size-limited team benefits agile product development. You find out more about servant leadership and creating a good environment for a scrum team. In short, you find out how team dynamics help agile product development succeed.

TABLE 16-1 **Traditional versus Agile Team Dynamics**

Team Management with Traditional Approaches	Team Dynamics with Agile Approaches
Project teams rely on *command and control* — a top-down approach to project management, where the project manager is responsible for assigning tasks to team members and attempting to control what the team does.	Agile teams are self-managing, self-organizing, and benefit from *servant leadership.* Instead of top-down management, a servant-leader coaches, removes obstacles, and prevents distractions to enable the team to thrive.
Companies evaluate individual employee performance.	Agile organizations evaluate team performance. Agile teams, like any sports team, succeed or fail as a whole team. Whole-team performance encourages individual team members to increase the ways they can contribute to the team's success.
Team members often find themselves working on more than one project at a time, switching their attention back and forth.	Development teams are dedicated to one goal at a time, and reap the benefits of focus.
Development team members have distinct roles, such as programmer or tester.	Agile organizations focus on skills instead of titles. Development teams work cross-functionally, doing different jobs within the team to ensure that they complete priority requirements quickly.
Development teams have no specific size limits.	Development teams are intentionally limited in size. Ideally, development teams have no fewer than three and no more than nine people.
Team members are commonly referred to as *resources,* a shortened term for *human resources.*	Team members are called *people, talent,* or simply *team members.* With agile product development, you probably will not hear the term *resource* used to refer to people.

Managing Team Dynamics

Time and again, when we talk with product owners, developers, and scrum masters, we hear the same thing: People enjoy agile product development. Scrum team dynamics enable people to do great work in the best way they know how. People on scrum teams have opportunities to learn, to teach, to lead, and to be part of a cohesive, self-managing team.

The following sections show you how to work as part of an agile team (using scrum as the context) and why agile approaches to teamwork make agile development successful.

Becoming self-managing and self-organizing

With agile product development, scrum teams are directly accountable for creating deliverables. Scrum teams manage themselves, organizing their own work and tasks. No one person tells the scrum team what to do. This doesn't mean that agile development efforts have no leadership. Each member of the scrum team has the opportunity to lead informally, based on his or her skills, ideas, and initiative.

The idea of self-management and self-organization is a mature way of thinking about work. Self-management assumes that people are professional, motivated, and dedicated enough to commit to a job and see it through. At the core of self-management is the idea that the people who are doing a job from day to day know the most about that job and are best qualified to determine how to complete it. Working with a self-managing scrum team requires trust and respect within the team and within the team's organization as a whole.

Nonetheless, let's be clear: Accountability is at the core of agile product development. The difference is that agile teams are held accountable for tangible results that you can see and demonstrate. Traditionally, companies held teams accountable for compliance to the organization's step-by-step process — stripping them of the ability or incentive to be innovative. Self-management, however, returns innovation and creativity to development teams.

TIP

For a scrum team to be self-managing, you need an environment of trust. Everyone on the scrum team must trust one another to do his or her best for the scrum team and the product. The scrum team's company or organization must also trust the scrum team to be competent, to make decisions, and to manage itself. To create and maintain an environment of trust, each member of the scrum team must commit, individually and as a team, to the product and to one another.

Self-managing development teams create better product architectures, requirements, and design for a simple reason: ownership. When you give people the freedom and responsibility to solve problems, they are more mentally engaged in their work.

Scrum team members play roles in all areas of development. Table 16-2 shows how scrum teams and development teams manage scope, procurement, time, cost, team dynamics, communication, stakeholders, quality, and risk.

TABLE 16-2 **Product Management and Self-Managing Teams**

Area of Product Management	How Product Owners Self-Manage	How Development Teams Self-Manage	How Scrum Masters Self-Manage
Scope	Use the product vision, the release goal, and each sprint goal to determine if and where scope items belong. Use product backlog prioritization to determine which requirements are developed.	May suggest features based on technical affinity. Work directly with the product owner to clarify requirements. Identify how much work they can take on in a sprint. Identify the tasks to complete scope in the sprint backlog. Determine the best way to create specific features.	Remove impediments that limit the amount of scope the development team can create. Through coaching, help development teams become more productive with each successive sprint.
Procurement	Secure necessary funding for tools and equipment for development teams.	Identify the tools they need to create the product. Work with the product owner to get those tools.	Help procure tools and equipment that accelerate development team velocity.
Time	Ensure that the development team correctly understands product features so that development teams can correctly estimate the effort to create those features. Use velocity — development speed — to forecast long-term timelines.	Provide effort estimates for product features. Identify what features they can create in a given time frame — the sprint. Often provide time estimates for tasks in each sprint. Choose their own daily schedules and manage their own time.	Facilitate estimation poker games. Help development teams increase velocity, which affects time. Shield the team from organizational time-wasters and distractions.
Cost	Ultimately responsible for the budget and return on investment. Use velocity to forecast long-term costs, based on timelines.	Provide effort estimates for product features.	Facilitate estimation poker games. Help development teams increase velocity, which affects cost.

(continued)

TABLE 16-2 ***(continued)***

Area of Product Management	How Product Owners Self-Manage	How Development Teams Self-Manage	How Scrum Masters Self-Manage
Team dynamics	Commit to their product as an integrated peer member of the scrum team.	Prevent bottlenecks by working cross-functionally, and are willing to take on different types of tasks. Continuously learn and teach one another. Commit, both individually and as part of the scrum team, to their product and to one another. Strive to build consensus when making important decisions.	Facilitate scrum team collocation. Help remove impediments to scrum team self-management. Servant-leaders who are integrated members of the scrum team. Strive to build consensus within the scrum team when making important decisions. Facilitate relationships between the scrum team and stakeholders.
Communication	Communicate information about the product and the business needs to development teams on an ongoing basis. Communicate information about the development progress to stakeholders. Help present working functionality to stakeholders at the sprint review meetings at the end of each sprint.	Inspect progress, coordinate upcoming tasks, and identify roadblocks in their daily scrum meetings. Keep the sprint backlog up-to-date daily, providing accurate, immediate information about development status. Present working functionality to stakeholders at the sprint review meetings at the end of each sprint.	Encourage face-to-face communication between all scrum team members. Foster close cooperation between the scrum team and other departments within the company or organization.
Stakeholders	Set vision, release, and sprint goal expectations. Shield development team from business noise. Collect feedback during sprint reviews. Gather requirements throughout project. Communicate release dates and how new feature requests affect release dates.	Demonstrate working functionality at sprint reviews. Work through product owner to decompose requirements. Report on progress through release and sprint burndown charts. Update task status no less than at the end of each day.	Coach on scrum and agile principles as they relate to their interaction with the scrum team. Shield developers from non-business distractions. Facilitate sprint reviews for gathering feedback. Facilitate interactions outside sprint reviews.

Area of Product Management	How Product Owners Self-Manage	How Development Teams Self-Manage	How Scrum Masters Self-Manage
Quality	Add and clarify acceptance criteria to requirements. Ensure that the development team correctly understands and interprets requirements. Provide development teams with feedback about the product from the organization and from the marketplace. Accept functionality as done during each sprint.	Commit to providing technical excellence and good design. Test their work throughout the day and comprehensively test all development each day. Inspect their work and adapt for improvements at sprint retrospective meetings at the end of each sprint.	Help facilitate the sprint retrospective. Help ensure face-to-face communication between scrum team members, which in turn helps ensure quality work. Help create a sustainable development environment so that the development team can perform at its best.
Risk	Look at overall product risks as well as risks to their ROI commitment. Prioritize high-risk items on the product backlog near the top to address them sooner rather than later.	Identify and develop the risk mitigation approach for each sprint. Alert the scrum master to roadblocks and distractions. Use information from each sprint retrospective to reduce risk in future sprints. Embrace cross-functionality to reduce risk if one member unexpectedly leaves the team. Commit to delivering shippable functionality at the end of each sprint, reducing overall product risk.	Help prevent roadblocks and distractions. Help remove roadblocks and identified risks. Facilitate development team conversations about possible risks.

REMEMBER

All in all, people developing products using agile techniques tend to find a great deal of job satisfaction. Self-management speaks to a deeply rooted human desire for autonomy — to control our own destiny — and allows people this control on a daily basis.

The next section discusses another reason that people who use agile product development techniques are happy: the servant-leader.

Supporting the team: The servant-leader

The scrum master serves as a servant-leader, someone who leads by removing obstacles, preventing distractions, and helping the rest of the scrum team do its job to the best of its ability. Agile leaders help find solutions rather than assign tasks. Scrum masters coach, trust, and challenge the scrum team to manage itself.

Other members of the scrum team can also take on servant leadership roles. While the scrum master helps get rid of distractions and roadblocks, the product owner and members of the development team can also help where needed. The product owner can lead by proactively providing important details about the product needs and quickly providing answers to questions from the development team. Development team members can teach and mentor one another as they become more cross-functional. Each person on a scrum team may act as a servant-leader at some point during product development. The servant leadership mindset permeates the entire team.

Larry Spears identified ten characteristics of a servant-leader in his paper, "The Understanding and Practice of Servant-Leadership" (Servant Leadership Roundtable, School of Leadership Studies, Regent University, August 2005). Here are those characteristics, along with our additions for how each characteristic can benefit the team dynamics.

- **Listening:** Listening closely to other members of the scrum team will help the people on the scrum team identify areas to help one another. A servant-leader may need to listen to what people are saying, as well as what people are *not* saying, in order to remove obstacles.
- **Empathy:** A servant-leader tries to understand and empathize with people on the scrum team, and to help them understand one another.
- **Healing:** Healing can mean undoing the damage of non-people-centric processes. These are processes that treat people like equipment and other replaceable parts. Many traditional project management approaches can be described as being non-people-centric.
- **Awareness:** The people on the scrum team may need to be aware of activities on many levels to best serve the scrum team.
- **Persuasion:** Servant-leaders rely on an ability to convince, rather than on top-down authority. Strong persuasion skills, along with organizational clout or influence, will help a scrum master advocate for the scrum team to the company or organization. A servant-leader can also pass along persuasion skills to the rest of the scrum team, helping maintain harmony and build consensus.

- **Conceptualization:** Each member of a scrum team can use conceptualization skills. The changing nature of agile lifecycles encourages the scrum team to envision ideas beyond those at hand. A servant-leader will help nurture the scrum team's creativity, both for the development of the product and for team dynamics.
- **Foresight:** Scrum teams gain foresight with each sprint retrospective. By inspecting its work, processes, and team dynamics on a regular basis, the scrum team can continuously adapt and understand how to make better decisions for future sprints.
- **Stewardship:** A servant-leader is the steward of the scrum team's needs. Stewardship is about trust. Members of the scrum team trust one another to look out for the needs of the team and the product as a whole.
- **Commitment to the growth of people:** Growth is essential to a scrum team's ability to be cross-functional. A servant-leader will encourage and enable a scrum team to learn and grow.
- **Building community:** A scrum team is its own community. A servant-leader will help build and maintain positive team dynamics within that community.

Servant leadership works because it positively focuses on individuals and interactions, a key tenet of agile product development. Much like self-management, servant leadership requires trust and respect.

TECHNICAL STUFF

The concept of servant leadership is not specific to agile product development. If you have studied management techniques, you may recognize the works of Robert K. Greenleaf, who started the modern movement for servant leadership — and coined the term *servant-leader* — in an essay in 1970. Greenleaf founded the Center for Applied Ethics, now known as the Greenleaf Center for Servant Leadership, which promotes the concept of servant leadership worldwide.

Another servant-leader expert, Kenneth Blanchard, co-wrote with Spencer Johnson the *One Minute Manager* (published by William Morrow), wherein he describes characteristics that make great managers of high-functioning people and teams. (The book has since been updated as *The New One-Minute Manager*, published by Harper Collins India.) The reason the managers Blanchard studied were so effective is because they focused on ensuring that the people doing the work had direction, resources, and protection from noise to do their job as quickly as possible.

The next two sections largely relate to team factors for scrum team success: the dedicated team and the cross-functional team.

Working with a dedicated team

Having a dedicated scrum team provides the following important benefits:

- **Keeping people focused on one goal at a time helps prevent distractions.** Dedication to one goal, such as a sprint goal, increases productivity by reducing *task-switching* — moving back and forth between different tasks without really completing any of them.
- **Dedicated scrum teams have fewer distractions — and fewer distractions mean fewer mistakes.** When a person doesn't have to meet the demands of more than one initiative, that person has the time and clarity to ensure his or her work is the best it can be. Chapter 17 discusses ways to increase product quality in detail.
- **When people work on dedicated scrum teams, they know what they will be working on every day.** An interesting reality of behavioral science is that when people know what they will be working on in the immediate future, their minds engage those issues consciously at work and unconsciously outside the work environment. Stability of tasks engages your mind for much longer each day, enabling better solutions and higher quality products.
- **Dedicated scrum team members are able to innovate more.** When people immerse themselves in a product without distractions, they can come up with creative solutions for product functionality.
- **People on dedicated scrum teams are more likely to be happy in their jobs.** By being able to concentrate on one goal, a scrum team member's job is easier. Many, if not most, people enjoy producing quality work, being productive, and being creative. Dedicated scrum teams lead to higher satisfaction.
- **When you have a dedicated scrum team working the same amount of time each week, you can accurately calculate velocity — the team's development speed.** In Chapter 15, we talk about determining a scrum team's velocity at the end of each sprint and using velocity to determine long-term timelines and costs. Because velocity relies on comparing output from one sprint to the next, using velocity to forecast time and cost works best if the scrum team's work hours are constant. If you are unable to have a dedicated scrum team, at least try to have team members allocated to your development effort for the same amount of time each week.

TECHNICAL STUFF

The idea of the productive multitasker is a myth. In the past 25 years, and especially in the last decade, a number of studies have concluded that task-switching reduces productivity, impairs decision-making skills, and results in more errors.

To have a dedicated scrum team, you need strong commitment from your organization. Many companies ask employees to work on multiple objectives or goals at

one time, under the mistaken assumption that they will save money by hiring fewer people. When companies start to embrace a more agile mindset, they learn that the least expensive approach is to reduce defects and raise development productivity through focus.

WARNING

Work in progress is expensive inventory and creates no value. Scrum teams continually seek to reduce work in progress through focus and dedication.

Each member of the scrum team can help ensure dedication:

- If you're a product owner, make sure that the company knows that a dedicated scrum team is a good fiscal decision. You are responsible for product return on investment, so be willing to fight for your product's success.
- If you're a member of the development team and anyone requests that you do work outside the current sprint goal, you can push back and involve the product owner or scrum master, if necessary. A request for outside work, regardless of how benign, is a potentially expensive distraction.
- If you're a scrum master, as the expert on agile approaches, you can educate the company on why a dedicated scrum team means decreased work in progress and increased productivity, quality, and innovation. A good scrum master should also have the organizational clout to keep the company from poaching people from the scrum team for other development efforts. See Chapter 8 to learn more about the importance of stable, long-lived, and even permanent teams.

Another characteristic of scrum teams is that they are cross-functional.

Working with a cross-functional team

Cross-functional development teams are also important. The development team doesn't just include programmers — people writing software code; it includes all the people who will have a job during development to transform a product requirement into something that is valuable and shippable. For non-software scrum teams, there won't be any programmers, but there will be people with the range of skills needed to create the product.

For example, a scrum team developing software would include people with programming, database, quality assurance, usability, graphics, design and infrastructure skills. While each person has specialties, being cross-functional means that everyone on the team is willing to pitch in on different parts of development, as much as possible. The same is true for non-software products.

As a member of a development team, you continuously ask yourself two questions: "What can I contribute today?" and "How can I expand my contribution in the future?" Everyone on the development team will use his or her current skills and specialties in each sprint. Cross-functionality gives development team members the opportunity to learn new skills by working on areas outside of their expertise. Cross-functionality also allows people to share their knowledge with their fellow development team members. You don't need to be a jack-of-all-trades to work on a development team, but you should be willing to learn new skills and help with all kinds of tasks. To learn more about building individual and team capability, see the T-shaped, pi-shaped and M-shaped models discussed in Chapter 7.

TECHNICAL STUFF

Although task-switching decreases productivity, cross-functionality works because you're not changing the context of what you are working on; you're looking at the same problem from a different perspective. Working on different aspects of the same problem increases knowledge depth and your ability to do a better job.

The biggest benefit of a cross-functional development team is the elimination of single points of failure. If you have worked on a project before, how many times have you experienced delays because a critical member of the team is on vacation, out sick, or, worse, has left the company? Vacations, illness, and turnover are facts of life, but with a cross-functional development team, other team members can jump in and continue work with minimal disruption. Even if an expert leaves the team unexpectedly and abruptly, other development team members will know enough about the work to keep it progressing.

WARNING

Development team members go on vacation or catch the flu. Don't sabotage your team by having only one person know a skill or functional area.

Cross-functionality takes strong commitment from the development team, both as individual members and as a group. The old phrase, "There is no *i* in *team*" is especially true with agile development. Working on a development team is about skills, rather than titles.

TIP

Development teams without titles are more merit-based because team seniority and status is based on current knowledge, skills, and contribution.

Letting go of the idea that you're a "senior quality assurance tester" or a "junior developer" can require a new way of thinking about yourself. Embracing the concept of being part of a cross-functional development team may take some work, but it can be rewarding as you learn new skills and develop a rhythm of teamwork.

TIP

When developers also test, they create code that is test-friendly.

Having a cross-functional development team also requires commitment and support from your organization. Some companies eliminate titles or keep them

intentionally vague (you might see something like "application development") to encourage teamwork. Other techniques for creating a strong cross-functional development team from an organizational standpoint include offering training, recognizing scrum teams as a whole, and being willing to make changes if a particular person does not fit in with a team environment. When hiring, your company can actively look for people who will work well in a highly collaborative environment, who want to learn new tasks, and who are willing to work on all areas of development.

Both the physical environment and the cultural environment of an organization are important keys to success. The next section shows you how.

Reinforcing openness

As we explain in other chapters, a collocated scrum team is ideal. The Internet has brought people together globally, but nothing — not the best combination of emails, instant messages, videoconferencing, phone calls, and online collaboration tools — can replace the simplicity and effectiveness of a face-to-face conversation. Figure 16-1 illustrates the difference between an email exchange and a conversation in person.

The idea of scrum team members working in the same physical location and being able to talk in person, instantly, is important to team dynamics. You find more details on communication later in this chapter. Also, Chapter 6 provides details on how to set up both physical and virtual environments for a scrum team to communicate effectively.

Having a cultural environment of openness, which is conducive to scrum team growth, is another success factor. Everyone on a scrum team should be able to

- Feel safe.
- Speak his or her mind in a positive way.
- Challenge the status quo.
- Be open about challenges without being penalized.
- Request resources that will make a difference.
- Make mistakes and learn from them.
- Suggest change and have other scrum team members seriously consider those changes.
- Respect fellow scrum team members.
- Be respected by other members of the scrum team.

FIGURE 16-1: Email versus face-to-face conversation.

Trust, openness, and respect are fundamental to team dynamics.

TIP

Some of the best product and process improvements come from novices asking "silly" questions.

Another facet of scrum team dynamics is the concept of the size-limited team.

Limiting development team size

An interesting psychological aspect of team dynamics is the number of people on a development team. Scrum development teams usually have between three and nine people. An ideal size is somewhere in the middle.

Limiting development team size to this range provides a team with enough diverse skills to take a requirement from paper to production while keeping communication and collaboration simple. Development team members can easily interact with one another and make decisions by consensus.

When you have development teams with more than nine people, the people on those teams tend to break into subgroups and build silos. This is normal social human behavior, but subgroups can be disruptive to a development team striving to be self-managing. It is also more difficult to communicate with larger development teams; there are more communication channels and opportunities to lose or misconstrue a message.

Development teams with fewer than nine people, on the other hand, tend to naturally gravitate to an agile approach. However, development teams that are too small may find working cross-functionally difficult because there may not be enough people with varying skills on the team.

TIP

If your product development requires more than nine development team members or you think you need to create a position to improve intra-team communication, consider breaking up the work between multiple scrum teams instead. Find details on how to work with multiple scrum teams in Chapters 15 and 19.

Managing product development with dislocated teams

As we say throughout the book, a collocated scrum team is ideal for agile product development. However, sometimes it isn't possible for a scrum team to work together in one place. *Dislocated teams* (teams with people who work in different physical locations), as we all witnessed during the COVID-19 pandemic, exist for many reasons and in different forms.

In some companies, people with the right skills for a team may work in different offices, and the company may not want the cost of bringing those people together for the duration of development. Some organizations work jointly with other organizations on development, but may not want or be able to share office space. Some people may telecommute, especially contractors, live long distances from the company they work with, and never visit that company's office. Some companies work with offshore groups and create products with people from other countries.

TIP

If you need to offshore, then offshore with both feet. Collocate entire scrum teams with the product owner, development team, and scrum master together. You may have one scrum team in one physical location, and another at a different physical location, but organize your scrum teams to be together.

The good news is that you can still develop with a dislocated scrum team or teams. If you have to work with a dislocated team, we've found that an agile approach allows you to see working functionality much sooner and limits the risk of

inevitable misunderstandings that a dislocated team will experience. Dislocated teams are often more effective taking an agile approach by using scrum than without it.

Table 16-3, from Ambysoft's "Agile Adoption Rate Survey Results" in 2008, shows a comparison of success rates for products developed with collocated scrum teams against those with geographically dispersed scrum teams. Even for teams separated by large distances, with scrum they still have high success rates.

TABLE 16-3 **Success of Collocated and Dislocated Scrum Teams**

Team Location	Success Percentage
Collocated scrum team (everyone on the team is in the same physical location)	83%
Dislocated but physically reachable (team members work in different physical locations but can travel to be face to face)	72%
Distributed across geographies (team members are separated significantly, for example, by time zones)	60%

"Agile Adoption Rate Survey Results" (Scott W. Ambler, Ambysoft, Copyright © 2008)

How do you successfully develop a product with a dislocated scrum team? We have three words: communicate, communicate, and communicate. Because daily in-person conversations are not possible throughout the entire day, dislocated scrum teams require unique efforts by everyone working on the product. Here are some tips for successful communication among non-collocated scrum team members:

- **Use videoconferencing technology to simulate face-to-face conversations.** The majority of interpersonal communication is visual, involving facial cues, hand gestures, and even shoulder shrugs. Videoconferencing enables people to see one another and benefit from nonverbal communication as well as a discussion. Use videoconferencing liberally throughout the day, not just for sprint meetings. Make sure team members are ready for impromptu video chats and have the necessary equipment, such as sufficient bandwidth, microphones, headphones and multiple monitors, to make videoconferencing successful.
- **If possible, arrange for the scrum team members to meet in-person in a central location at least once at the beginning of the development, and preferably multiple times throughout.** The shared experience of meeting in-person, even once or twice, can help build teamwork among dislocated team members. Working relationships built through face-to-face visits are stronger and carry on after the visit ends.

- **Use an online collaboration tool.** Some tools simulate whiteboards and user story cards, track conversations, and enable multiple people to update artifacts at the same time.
- **Include scrum team members' pictures on online collaboration tools, or even in email address signature lines.** Humans respond to faces more than written words alone. A simple picture can help humanize instant messages and emails.
- **Be cognizant of time zone differences.** Put multiple clocks showing different time zones on the wall so you don't accidentally call someone's cellphone at 3 a.m. and wake up that person — or wonder why he or she isn't answering.
- **Be flexible because of time zone differences as well.** You may need to take video calls or phone calls at odd hours from time to time to help keep work moving. For drastic time zone differences, consider trading off on times you are available. One week, Location A can be available in the early morning. The next, Location B can be available later in the evening. That way, no one always has an inconvenience.
- **If you have any doubt about a conversation or a written message, ask for clarification by phone or video.** It always helps to double-check when you're unsure of what someone meant. Follow up with a call to avoid mistakes from miscommunication. Additional communication effort is required for a successful dislocated team.
- **Be aware of language and cultural differences between scrum team members, especially when working with groups in multiple countries.** Understanding colloquialisms and pronunciation differences can increase the quality of your communication across borders. It helps to know about local holidays, too. We've been blindsided more than once by closed offices outside our region, which is another reason to meet in-person in each other's physical locations.
- **Make an extra attempt to discuss non-work topics sometimes.** Discussing non-work topics helps you grow closer to scrum team members, regardless of location.

With dedication, awareness, and strong communication, distributed agile development can succeed.

The unique approaches to team dynamics is part of what make agile product development successful. Communication is closely related to team dynamics, and agile communication methods have big differences from traditional projects, as you see in the following section.

What's Different about Agile Communication?

Communication, in project management terms, is the formal and informal ways the people on the project team convey information to each other. As with traditional projects, good communication is a necessity for agile product development.

However, the agile principles set a different tone, emphasizing simplicity, directness, and face-to-face conversations. The following agile principles relate to communication:

4. Business people and developers must work together daily throughout the project.
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.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The Agile Manifesto also addresses communication, valuing working software over comprehensive documentation. Although documentation has value, working functionality is more important.

Table 16-4 shows some differences between communication on traditional projects and agile development.

TIP

The question of how much documentation is required is not a volume question but an appropriateness question. Why do you need a specific document? How can you create it in the simplest way possible? You can use poster-sized sticky sheets to put on the wall and make information digestible. This can also work best for visually conveying artifacts such as the vision statement, the definition of done, the impediments log, and important architectural decisions. Pictures truly are worth a thousand words.

The following sections show how to take advantage of the agile framework's emphasis on in-person communication, focus on simplicity, and value of working functionality as a communication medium.

TABLE 16-4 **Traditional versus Agile Communication**

Communication Management with Traditional Approaches	Communication Management with Agile Approaches
Team members might make no special effort for in-person conversations.	Agile approaches value face-to-face communication as the best way to convey information.
Traditional approaches place high value on documentation. Teams may create a large number of complex documents and status reports based on process, rather than considering actual need.	Agile documents, or *artifacts*, are intentionally simple and provide information that is barely sufficient. Agile artifacts only contain essential information and can often convey status at a glance. Teams use the *show, don't tell* concept, showing working functionality to communicate progress on a regular basis in the sprint review.
Team members may be required to attend a large number of meetings, whether or not those meetings are useful or necessary.	Meetings are, by design, as quick as possible and include only people who will add to the meeting and benefit from the meeting. Agile meetings provide all the benefits of face-to-face communication without wasting time. The structure of agile meetings is to enhance, not reduce, productivity.

Managing Agile Communication

To manage communication with agile product development, you need to understand how different agile communication methods work and how to use them together. You also need to know why status is different and how to report progress to stakeholders. The following sections show you how.

Understanding agile communication methods

You can communicate through artifacts, meetings, and informally.

Face-to-face conversations are the heart and soul of agile product development. When scrum team members talk with one another about the product throughout every day, communication is easy. Over time, scrum team members understand each other's personality, communication style, and thought processes, and will be able to communicate quickly and effectively.

Figure 16-2, from Alistair Cockburn's presentation *Software Development as a Cooperative Game*, shows the effectiveness of face-to-face communication versus other types of communication.

Copyright © Humans and Technology, Inc.

FIGURE 16-2: Comparison of communication types.

In previous chapters, we describe a number of artifacts and meetings that fit with agile development. All the agile artifacts and meetings play a role in communication. Agile meetings provide a format for communicating in a face-to-face environment. Agile meetings have a specific purpose and a specific amount of time so that the development team can work, rather than sit in meetings. Agile artifacts provide a format for written communication that is structured but not cumbersome or unnecessary.

Table 16-5 provides a view of the different communication channels with agile product development.

TABLE 16-5 Agile Communication Channels

Channel	Type	Role in Communication
Release planning and sprint planning	Meetings	Planning meetings have specific desired outcomes or are focused on a specific scope of work driven by a business goal, and concisely communicate the purpose and details of the release, and the sprint to the scrum team. Learn more about planning meetings in Chapters 9 and 10.
Product vision statement	Artifact	The product vision statement communicates the end goal of the product to the team and the organization. Find out more about the product vision in Chapter 9.
Product roadmap	Artifact	The product roadmap communicates a long-term view of the features that support the product vision and are likely to be part of the product. Find out more about the product roadmap in Chapter 9.

Channel	Type	Role in Communication
Product backlog	Artifact	The product backlog communicates the scope of the product as a whole to the team. Find out more about the product backlog in Chapters 9 and 10.
Release plan	Artifact	The release plan communicates the goal and timing for a specific release. Find out more about the release plan in Chapter 10.
Sprint backlog	Artifact	When updated daily, the sprint backlog provides immediate sprint and development status to anyone who needs that information. The burndown chart on the sprint backlog provides a quick visual of the sprint progress. Find out more about the sprint backlog in Chapters 10 and 11.
Task board	Artifact	Using a task board visually radiates the status of the current sprint or release to anyone who walks by the scrum team's work area. As a team member moves a completed task, everyone knows it's time for the next task. Find out more about the task board in Chapter 11.
Daily scrum	Meeting	The daily scrum provides the scrum team with a verbal, face-to-face opportunity to coordinate the priorities of the day and identify any challenges. Find out more about daily scrum meetings in Chapter 11.
Face-to-face conversations	Informal	Face-to-face conversations are the most effective mode of communication.
Sprint review	Meeting	The sprint review is the embodiment of show, don't tell, philosophy. Displaying working functionality to the entire team conveys progress in a more meaningful way than a written report or a conceptual presentation ever could. Find out more about sprint reviews in Chapter 12.
Sprint retrospective	Meeting	The sprint retrospective allows the scrum team to communicate with one another specifically for improvement. Find out more about sprint retrospectives in Chapter 12.
Meeting notes	Informal	Meeting notes are an optional, informal communication method. Meeting notes can capture action items from a meeting to ensure that people on the scrum team remember them for later. Notes from a sprint review are updated product backlog items. Notes from a sprint retrospective are action items added to the product backlog for consideration in a future sprint and remind the scrum team of plans for improvement.
Collaborative solutions	Informal	Whiteboards, sticky notes, and electronic collaboration tools all help the scrum team communicate. Ensure that these tools augment, rather than replace, face-to-face conversations. Capturing and saving collaboration results is a low-fidelity way to remind the team of decisions made for immediate and future consideration. Digital versions of these can be effective.

REMEMBER

Artifacts, meetings, and more informal communication channels are all tools. Keep in mind that even the best tools need people to use those tools correctly to be effective. Agility is about people and interactions; tools are secondary to success.

The next section addresses a specific area of agile communication: status reporting.

Status and progress reporting

All products have stakeholders, people outside the immediate scrum team who have a vested interest in the product. At least one of the stakeholders is the person responsible for paying for your development (the sponsor). It is important for stakeholders, especially those responsible for budgets, to know how development is progressing. This section shows how to communicate your status.

Status for scrum teams is a measure of the features that the team has completed. Using the definition of done from Chapters 2, 10, 12, and 17, a feature is complete if the scrum team has developed, tested, integrated, and documented that feature, per the agreement between the product owner and the development team.

If you've worked on a traditional project, how many times have you been in a status meeting and reported that the project was, say, 64 percent complete? If your stakeholders had replied, "Great! We would like that 64 percent now; we ran out of funds," you and the stakeholders alike would be at a loss, because you didn't mean that 64 percent of your features were ready to use. You meant that each one of the product features was only 64 percent in progress, you had no working functionality, and you still had a lot of work to do before anyone could use the product.

Working functionality that meets the definition of done is the primary measure of progress. You can confidently say that product features are complete. Because scope changes constantly, you would not express status as a percentage. Instead, a list of potentially shippable features would be more interesting for stakeholders to see as it grows.

Track the progress of your sprint and release daily. Your primary tools for communicating status and progress are the task board, sprint backlog, product backlog, release and sprint burndown charts, and the sprint review.

TIP

The sprint review is where you demonstrate working software to your stakeholders. Resist creating slides or handouts; the key to the sprint review is showing your stakeholders progress as a demonstration, rather than only telling them what you completed. Principle 7 says, "Working software is the primary measure of progress." Show, don't tell.

Strongly encourage anyone who may have an interest in your product to come to your sprint reviews. When people see the working functionality in action, especially on a regular basis, they get a much better sense of the work you've completed.

WARNING

Don't get sucked in to *double work agile*. Companies and organizations that are starting out using agile techniques may expect to see traditional status reports, in addition to agile artifacts. These organizations may also want members of the scrum team to attend regular status meetings, outside the daily scrums and other agile meetings. This is called *double work agile* because you are doing twice as much work as necessary. Double work agile is one of the top pitfalls for agile adoptions. Scrum teams will burn out quickly if they try to meet the demands of two drastically different development approaches. You can avoid double work agile by educating your company about why agile artifacts and events are a better replacement for old documents and meetings. Insist on experimenting with agile artifacts and events.

The sprint backlog is a report of the daily status of your current sprint. The sprint backlog contains the sprint's user stories and their related tasks and estimates. The sprint backlog also often has a burndown chart that visually shows the status of the work the development team has completed and the remaining work to complete the requirements in the sprint. The development team is in charge of updating the sprint backlog at least once a day by updating the number of hours of work remaining for each task. The sprint backlog provides the daily status, so you don't need to spend time on status in your daily scrum. Your daily scrum is for coordinating work for the day, not for status.

WARNING

If you're a project manager now, or if you study project management in the future, you may come across the concept of *earned value management* (EVM), as a way of measuring project progress and performance. Some agile practitioners try to use an agile-like version of EVM, but we avoid EVM for agile product development. EVM assumes that your project has a fixed scope, which is antithetical to an agile approach. Instead of trying to change agile approaches to fit into old models, use the tools here — they work.

The burndown chart quickly shows, rather than tells, status. When you look at a sprint burndown chart, you can instantly see whether the sprint is going well or might be in trouble. In Chapter 11, we show you an image of sample burndown charts for different sprint scenarios; here it is again in Figure 16-3.

If you update your sprint backlog every day, you'll always have an up-to-date status for your stakeholders. You can also show them the product backlog so that they know which features the scrum team has completed to date, which features will be part of future sprints, and the priority of the features.

FIGURE 16-3: Profiles of burndown charts.

REMEMBER

The product backlog will change as you add and reprioritize features. Make sure that people who review the product backlog, especially for status purposes, understand this concept.

TIP

A task board is a great way to quickly show your team the status of a sprint, release, or even of the entire product. Task boards have sticky notes with user story titles in at least four columns: To Do, In Progress, Accept, and Done. If you display your task board in the scrum team's work area, anyone who walks by can see a high-level status of which product features are done and which features are in progress. The scrum team always knows where development stands, because the team sees the task board every day. See an example of a task board in Chapter 6.

Always strive for simple, low-fidelity information radiators to communicate status and progress. The more you can make information accessible and on-demand, the less time you and your stakeholders will spend preparing and wondering about status.

IN THIS CHAPTER

- Learning how agile quality approaches reduce risk
- Discovering ways to ensure quality product development
- Taking advantage of automated testing for better productivity
- Understanding how agile development approaches reduce risk

Chapter 17

Managing Quality and Risk

Quality and risk are closely related parts of product development. In this chapter, you find out how to deliver quality products using agile methods. You understand how to take advantage of agile approaches to manage product risk. You see how quality has historically affected risk, and how agile quality management fundamentally reduces risk.

What's Different about Agile Quality?

Quality refers to whether a product works, and whether it fulfills the stakeholders' and customers' needs. Quality is an inherent part of agile product management. All 12 Agile Principles that we list in Chapter 2 promote quality either directly or indirectly. Those principles follow:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

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

These principles emphasize creating an environment where teams are able to produce valuable, working functionality. Agile approaches encourage quality both in the sense of products working correctly and meeting the needs of stakeholders.

Table 17-1 shows some differences between quality management on traditional projects and agile product development.

BUGS. BUGS? BUGS!

Why do we call computer problems *bugs?* The first computers were large, glass-encased machines that took up entire rooms. In 1947, one of these behemoth computers, the Mark II Aiken Relay Calculator at Harvard University, had problems with one of its circuits. Engineers traced the issue to a moth — a literal bug — in the machine. After that, the team's running joke was that any issue with the computer had to be a bug. The term stuck, and people still use *bug* today to describe hardware problems, software problems, and sometimes even problems outside of the computer science realm. The engineers at Harvard even taped the moth to a logbook. That first bug was on display for years at the Smithsonian National Museum of American History.

TABLE 17-1 **Traditional versus Agile Quality**

Quality Management with Traditional Approaches	Quality Dynamics with Agile Approaches
Testing is the last phase of a project before product deployment. Some features are tested months after they were created.	Testing is a daily part of each sprint and is included in each requirement's definition of done. You use automated testing, allowing quick and robust testing every day.
Quality is often a reactive practice, with the focus mostly on product testing and issue resolution.	You address quality both reactively, through testing, and proactively, encouraging practices to set the stage for quality work. Examples of proactive quality approaches in software development include face-to-face communication, pair programming, test-driven development (also known as test-first development), and established coding standards.
Problems are riskier when found at the end of a project. Sunk costs are high by the time teams reach testing.	You can create and test riskier features in early sprints, when sunk costs are still low.
Problems or defects, sometimes called *bugs* in software development, are hard to find at the end of a project, and fixes for problems at the end of a project are costly.	Problems are easy to find when you frequently and incrementally test smaller amounts of shippable work. Fixes are easier when you fix something you just created, rather than something you created months earlier.
Sometimes, to meet a deadline or save money, teams cut the testing phase short.	Testing is assured because it is part of every sprint to satisfy the team's definition of done.

At the start of this chapter, we state that quality and risk are closely related. The agile approaches in Table 17-1 greatly reduce the risk and unnecessary cost that usually accompany deferred quality management.

Another difference about quality is the multiple quality feedback loops throughout development. In Figure 17-1, you see the different types of product feedback a scrum team receives in the course of development. The development team can immediately incorporate this feedback into the product, increasing product quality on a regular basis.

FIGURE 17-1: Quality feedback cycles.

REMEMBER

In Chapter 16, we tell you that development teams include everyone who works on a product. Development teams must include people who are experts in creating and executing tests and ensuring quality. Development team members are cross-functional; that is, every team member may do different jobs at different times during development. Cross-functionality extends to quality activities such as preventing issues, testing, and fixing bugs.

In the next section, you see how to use agile techniques to increase quality.

Managing Agile Quality

Agile development teams have the primary responsibility for quality. The responsibility for quality is an extension of the responsibilities and freedoms that come with self-management. When the development team is free to determine its development methods, the development team is also responsible for ensuring that those methods result in quality work.

TECHNICAL STUFF

Organizations often refer to quality management as a whole as *quality assurance,* or *QA.* You may see QA departments, QA testers, QA managers, QA analysts, and all other flavors of QA-prefixed titles to refer to people who are responsible for quality activities. QA is also sometimes used as shorthand for testing, as in "we performed QA on the product" or "now we are in the QA phase." QA usually refers to whether the product meets the business or customer need for which it was created. *Quality control (QC)* is also a common way to refer to quality management, but it refers more directly to the technical quality of the product.

The other members of the scrum team — the scrum master and the product owner — also play parts in quality management. Product owners provide clarification on requirements and also accept those requirements as being done throughout each sprint. Scrum masters help ensure development teams have a work environment where the people on development teams can work to the best of their abilities.

Luckily, agile approaches have several ways to help scrum teams create quality products. In this section, you see how testing in sprints increases the likelihood of finding defects and reduces the cost of fixing them. You gain an understanding of the many ways agile development proactively encourages quality product development. You see how inspecting and adapting on a regular basis addresses quality. Finally, you find out how automated testing is essential to delivering valuable products continuously throughout agile development.

Quality and the sprint

Quality management is a daily part of agile product development. Scrum teams run development in sprints, short development cycles that last one to four weeks. Each cycle includes activities from the different phases of a traditional project for each user story in the sprint: requirements, design, development, testing, and integration for deployment. Find out more about working in sprints in Chapters 10, 11, and 12.

TIP

Here's a quick riddle: Is it easier to find a quarter on a table or in a stadium? Obviously, the answer is a table. Just as obvious is that it is easier to find a defect in a single backlog item's worth of development than in an entire product. Iterative development makes quality product development easier.

Scrum teams test throughout each sprint. Figure 17-2 shows how testing fits into sprints. Notice that testing begins in the first sprint, as soon as the development team starts creating the first requirement.

When development teams test throughout each sprint, they can find and fix defects very quickly. Development teams implement product requirements, immediately test those requirements, and fix any problems immediately before considering the work done. Instead of trying to remember how to fix something they created weeks or months ago, development teams are, at the most, fixing the requirement they worked on one or two days earlier.

Testing every day is a great way to ensure product quality. Another way to ensure product quality is to create a better product from the start. The next section shows you different ways that agile product development helps you avoid errors and create an excellent product.

Proactive quality

An important and often-neglected aspect of quality is the idea of preventing problems. A number of agile approaches allow and encourage scrum teams to proactively create quality products. These practices include

- An emphasis on technical excellence and good design
- Incorporation of quality-specific development techniques into product creation
- Daily communication between the development team and the product owner
- Acceptance criteria built into user stories
- Face-to-face communication and collocation
- Sustainable development
- Regular inspection and adaption of work and behavior

FIGURE 17-2: Testing within sprints.

The following sections provide a detailed look at each of these proactive quality practices.

REMEMBER

Quality means both that a product works correctly and that the product does what the stakeholders need it to do.

Continuous attention to technical excellence and good design

Scrum teams focus on technical excellence and good design because these traits lead to valuable products. How do development teams provide great technical solutions and designs?

One way that development teams provide technical excellence is through self-management, which provides them with the freedom to innovate technically. Traditional organizations may have mandatory technical standards that may or may not make sense for a given project. Self-organizing development teams have the freedom to decide whether a standard will provide value in creating a product, or if a different approach will work better. Innovation can lead to good design, technical excellence, and product quality.

Self-management also provides development teams with a sense of product ownership. When people on development teams feel a deep responsibility for the product they're creating, they often strive to find the best solutions and execute those solutions in the best way possible.

TIP

Nothing is more sophisticated than a simple solution.

Organizational commitment also plays a role in technical excellence. Some companies and organizations, regardless of their product management approaches, have a commitment to excellence. Think about the products that you use every day and associate with quality; chances are those products come from companies that value good technical solutions. If you're working for a company that believes in and rewards technical excellence, enacting this agile principle will be easy.

Other companies may undervalue technical excellence; scrum teams at these companies may struggle when trying to justify training or tools that will help create better products. Some companies do not make the connection between good technology, good products, and profitability. Scrum masters and product owners may need to educate their companies on why good technology and design are important and may need to lobby to get development teams what they need to create a great product.

WARNING

Don't confuse technical excellence with using new technologies for the sake of using something new or trendy. Your technology solutions should efficiently support the product needs, not just add to a resume or a company skills profile.

By incorporating technical excellence and good design into your everyday work, you create a quality product that you are proud of.

Quality development techniques

During the past several decades of software development, the motivation to be more adaptive and agile has inspired a number of agile development techniques that focus on quality. This section provides a high-level view of a few extreme programming (XP) development approaches that help ensure quality proactively. For more information on XP practices, see Chapter 5.

TIP

Many agile quality management techniques were created with software development in mind. You can adapt some of these techniques when creating other types of products, such as hardware products or even building construction. If you're going to work on a non-software development effort, read about the development methods in this section with adaptability in mind:

- **Test-driven development (TDD):** This development method begins with a developer creating a test for the requirement he or she wants to create. The developer then runs the test, which should fail at first because the functionality does not yet exist. The developer develops until the test passes, and then refactors the code — takes out as much code as possible, while still having the test pass. With TDD, you know that the newly created functionality of a requirement works correctly because you test while you create the functionality and develop the functionality until the test passes.
- **Pair programming:** With *pair programming,* developers work in groups of two. Both developers sit at the same computer and work as a team to execute one development task. The developers take turns at the keyboard to collaborate. Usually, the one at the keyboard takes a direct tactical role, while the observing partner takes a more strategic or navigating role, looking ahead and providing in-the-moment feedback. Because the developers are literally looking over one another's shoulder, they can catch errors quickly. Pair programming increases quality by providing instant error checks and balances.
- **Peer review:** Sometimes called *peer code review,* a *peer review* involves members of the development team reviewing one another's work as soon as it is completed. Like pair programming, peer reviews have a collaborative nature; when developers review each other's finished products, the developers work together to provide solutions for any issues they find. If development

teams don't practice pair programming, they should at least practice peer reviews, which increase quality by allowing development experts to look for structural problems within the product.

- **Collective code ownership:** In this approach, everyone on the development team can create, change, or fix any part of the code. Collective code ownership can speed up development, encourage innovation, and with multiple pairs of eyes on the code, help development team members quickly find defects.
- **Continuous integration:** This approach involves the creation of integrated code builds one or more times each day. Continuous integration allows members of the development team to check how the user story it is creating works with the rest of the product. Continuous integration helps ensure quality by allowing the development team to check for conflicts regularly. Continuous integration is essential to automated testing; you need to create a code build with each check-in before running automated tests. Find out more about automated testing later in this chapter.

REMEMBER

With agile product development, the development team decides which tools and techniques will work best for the sprints, the product, and the team.

Many agile software development techniques help ensure quality, and there is a lot of discussion and information about these techniques in the community of people who use agile approaches. We encourage you to learn more about these, especially if you're a developer. Entire books are dedicated to some of these techniques, such as test-driven development. The information we provide here is at the tip of the iceberg. See Chapter 24 for more recommendations.

The product owner and development team

Another aspect of agile product development that encourages quality is the close relationship between the development team and the product owner. The product owner is the voice of business needs for the product. In this role, the product owner works with the development team every day to ensure that the functionality meets those business needs.

During planning stages, the product owner's job is to help the development team understand each requirement correctly. During the sprint, the product owner answers questions that the development team has about requirements and is responsible for reviewing functionality and accepting them as done. When the product owner accepts requirements, he or she ensures that the development team correctly interpreted the business need for each requirement, and that the new functionality performs the task that it needs to perform.

In waterfall projects, feedback loops between developers and business owners are less frequent, so a development team's work typically strays from the original product goals set in the product vision statement.

A product owner who reviews requirements daily catches misinterpretations early. The product owner can then set the development team back on the right path, avoiding wasted time and effort.

REMEMBER

The product vision statement not only articulates the product's goals but also communicates how your product supports the company's or organization's strategies. Chapter 9 explains how to create a product vision statement.

User stories and acceptance criteria

Another proactive quality measure with agile development is the acceptance criteria you build into each user story. In Chapter 9, we explain that a user story is one format for describing product requirements. User stories increase quality by outlining the specific actions the user will take to correctly meet business needs. Figure 17-3 shows a user story and its acceptance criteria.

Title Transfer money between accounts

As Carol,

I want to transfer funds between accounts

so that each account has the correct amount of funds

Value	Author	Estimate
	Jennifer	

When I do this:	This happens:
When I view my account balances,	I see an option to transfer funds.
When I select the transfer option,	I choose between which accounts I want to transfer funds.
When I select the "transfer from" option,	I see a list of my available accounts and balances.
When I select the "transfer to" option,	I see a list of my available accounts and balances.

FIGURE 17-3: A user story and acceptance criteria.

Even if you don't describe your requirements in a user story format, consider adding validation steps to each of your requirements. Acceptance criteria don't just help the product owner review requirements; they help the development team understand how to create the product in the first place.

Face-to-face communication

Have you ever had a conversation with someone and known, just by looking at that person's face, that he or she didn't understand you? In Chapter 16, we explain that face-to-face conversations are the quickest, most effective form of communication. This is because humans convey information with more than just words; our facial expressions, gestures, body language, and even where we are looking contribute to communicating and understanding one another.

Face-to-face communication helps ensure quality because it leads to better interpretation of requirements, roadblocks, and discussions between scrum team members.

Sustainable development

Chances are, at some point in your life, you've found yourself working or studying long hours for an extended period of time. You may have even pulled an all-nighter or two, getting no sleep at all for a night. How did you feel during this time? Did you make good decisions? Did you make any silly mistakes?

Unfortunately, many teams on traditional projects find themselves working long, crazy hours, especially toward the end of a project, when a deadline is looming and it seems like the only way to finish is to spend weeks working extra-long days. Those long days often mean more problems later, as team members start making mistakes — some silly, some more serious — and eventually burn out.

With agile development, scrum teams help ensure that they do quality work by creating an environment where members of the development team sustain a constant working pace indefinitely. Working in sprints helps sustain a constant working pace; when the development team chooses the work it can accomplish in each sprint, it shouldn't have to rush at the end.

The development team can determine what sustainable means for itself, whether that means working a regular 40-hour workweek, a schedule with more or fewer days or hours, or working outside a standard nine-to-five time frame.

TIP

If your fellow scrum team members start coming to work with their shirts on inside out, you might want to double-check that you're maintaining a sustainable development environment.

Keeping the development team happy, rested, and able to have a life outside of work can lead to fewer mistakes, more creativity and innovation, and better overall products.

Being proactive about quality saves you a lot of headaches in the long run. It is much easier and more enjoyable to work on a product with fewer defects to fix. The next section discusses an agile approach that addresses quality from both a proactive and a reactive standpoint: inspect and adapt.

Quality through regular inspecting and adapting

The agile tenet of inspect and adapt is a key to creating quality products. Throughout agile development, you look at both your product and your process (inspect)

and make changes as necessary (adapt). Chapters 9 and 12 have more information about this tenet.

In the sprint review and sprint retrospective meetings, scrum teams regularly step back and review their work and methods and determine how to make adjustments for a better product. We provide details on the sprint review and sprint retrospective in Chapter 12. Following is a quick overview of how these meetings help ensure quality.

In a sprint review, scrum teams review requirements completed at the end of each sprint. Sprint reviews address quality by letting stakeholders see working requirements and provide feedback on those requirements throughout the course of development. If a requirement doesn't meet stakeholder expectations, the stakeholders tell the scrum team immediately. The scrum team can then adjust the product in a future sprint. The scrum team can also apply its revised understanding of how the product needs to work on other product requirements.

In a sprint retrospective, scrum teams meet to discuss what worked and what might need adjusting at the end of each sprint. Sprint retrospectives help ensure quality by allowing the scrum team to discuss and immediately fix problems. Sprint retrospectives also allow the team to come together and formally discuss changes to the product, the development process, or work environment that might increase quality.

The sprint review and sprint retrospective aren't the only opportunities for inspecting and adapting for quality. Agile approaches encourage reviewing work and adjusting behavior and methods throughout each workday. Daily inspecting and adapting everything you do on the product help ensure quality.

Another way to manage and help assure quality is to use automated testing tools. The next section explains why automated testing is important to agile software development and how to incorporate automated testing into your product.

Automated testing

Automated testing is the use of software to test your product. If you want to quickly create functionality that meets the definition of done — designed, developed, tested, integrated, and documented — you need a way to quickly test each piece of functionality as it's created. Automated testing means quick and robust testing on a daily basis. Scrum teams continually increase the frequency with which they automatically test their system so they can continually decrease the time it takes them to complete and deploy new valuable functionality to their customers.

TIP

Teams won't become agile without automated testing. Manual testing simply takes too long.

Throughout this book, we explain how scrum teams embrace low-tech solutions. Why, then, is there a section in this book about automated testing, a rather high-tech quality management technique? The answer to this question is efficiency. Automated testing is like the spell-check feature in word-processing programs. As a matter of fact, spell-checking is a form of automated testing. In the same way, automated testing is a much quicker and often more accurate — thus, more efficient — method of finding defects than manual testing.

To develop a product using automated testing, development teams develop and test using the following steps:

1. **While implementing user stories, write and execute automated tests in support of user stories.**
2. **Whenever something is added, run automated tests as part of the organization's continuous integration and continuous deployment pipeline.**
3. **Give feedback to developers immediately when tests fail, ensuring that the errors are corrected immediately.**

TIP

Automated testing allows development teams to have rapid create-test-fix cycles. Also, automated testing software can often test requirements quicker and with more accuracy and consistency than a person testing those requirements.

Today's market has a lot of automated testing tools. Some automated testing tools are open-source and free; others are available for purchase. The development team needs to review automated testing options and choose the tool that will work best.

Automated testing changes the work for people in quality roles on the development team. Traditionally, a large part of a quality management person's work involved manually testing products. The tester on a traditional project would use the product and look for problems. With automated testing, however, quality activities largely involve creating tests to run on automated testing software. Automated testing tools augment, rather than replace, people's skills, knowledge, and work.

WARNING

It is still a good idea to have humans periodically check that the requirements you're developing work correctly, especially when you first start using an automated testing tool. Any automated tool can have hiccups from time to time. By manually double-checking (sometimes called smoke-testing) small parts of automated tests, you help avoid getting to the end of a sprint and finding out that your product doesn't work like it should.

You can automate almost any type of product test. If you're new to product development, you may not know that there are many different types of testing. A small sample includes the following:

- **Unit testing:** Tests individual units, or the smallest parts, of product code.
- **Regression testing:** Tests an entire product from start to finish, including requirements you have tested previously.
- **User acceptance testing:** Product stakeholders or even some of the product's end users review a product and accept it as complete.
- **Functional testing:** Tests to make sure the product works according to acceptance criteria from the user story.
- **Integration testing:** Tests to make sure the product works with other parts of the product.
- **Enterprise testing:** Tests to make sure the product works with other products in the organization, as necessary.
- **Performance testing:** Tests how a product performs given different scenarios.
- **Load testing:** Tests how well a product handles different amounts of concurrent activity.
- **Security testing:** Tests for product vulnerabilities, nefarious threats, and weaknesses that can be exploited.
- **Smoke testing:** Tests on small but critical parts to help determine if the product as a whole is likely to work.
- **Static testing:** Focuses on checking standards, rather than working product.

Automated testing works for these tests as well as many other types of product tests.

As you may understand by now, quality is an integral part of agile product development. Quality is just one factor, however, that differentiates risk with agile product development from traditional projects. In the next sections, you see how risk on traditional projects compares to risk with agile development.

What's Different about Agile Risk Management?

Risk refers to the factors that contribute to a project's success or failure. With agile product development, risk management doesn't have to involve formal risk documentation and meetings. Instead, risk management is built into scrum roles, artifacts, and events. In addition, consider the following agile principles that support risk management:

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.
7. Working software is the primary measure of progress.

The preceding principles, and any practice that demonstrates those principles, significantly mitigate or eliminate many risks that frequently lead to project challenges and failure.

According to the Standish Group's "2015 Chaos Report," a study of 10,000 software projects, small agile development efforts are 30 percent more likely to succeed than traditional projects. See Figure 17-4. Medium-sized projects are four times (400 percent) more likely to succeed with an agile approach than a traditional approach, and large, complex projects are six times (600 percent) more likely to succeed with an agile approach. In other words, the bigger the development effort, the higher the chance of success using agile approaches.

Table 17-2 shows some of the differences between risk on traditional projects and agile development.

With agile approaches, risk declines as development progresses. Figure 17-5 shows a comparison of risk and time between waterfall projects and agile product development.

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

FIGURE 17-4: Standish Group's "2015 Chaos Report."

The resolution of all software projects from FY2011-2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

TABLE 17-2 **Traditional versus Agile Risk**

Risk Management with Traditional Approaches	Risk Dynamics with Agile Approaches
Large numbers of projects fail or are challenged.	Risk of catastrophic failure — spending large amounts of money with nothing to show — is almost eliminated.
The bigger, longer, and more complex the project, the more risky it is. Risk is highest at the end of a project.	Product value is gained immediately and incrementally every sprint, rather than sinking costs into a product for months or even years with the growing chance of failure.
Conducting all the testing at the end of a project means that finding serious problems can put the entire project at risk.	Testing occurs while you develop. If a technical approach, a requirement, or even an entire product is not feasible, the development team discovers this in a short time, and you have more time to course correct. If correction is not possible, stakeholders spend less money on failed development.
Projects are unable to accommodate new requirements mid-project without increased time and cost because extensive sunk cost exists in even the lowest-priority requirements.	Change for the benefit of the product is welcomed. Agile techniques accommodate new high-priority requirements without increasing time or cost by removing a low-priority requirement of equal time and cost.

Risk Management with Traditional Approaches	Risk Dynamics with Agile Approaches
Traditional projects require precise time and cost estimates at the project start, when teams know the least about the project. Estimates are often inaccurate, creating a gap between expected and actual project schedules and budgets.	Time and cost is estimated using the scrum team's actual performance, or velocity. You refine estimates throughout development, because the longer you work, the more you learn about the product, the requirements, and the scrum team.
When stakeholders don't have a unified goal, they can end up confusing the project team with conflicting information about what the product should achieve.	A single product owner is responsible for creating a vision for the product and represents the stakeholders to the team.
Unresponsive or absent stakeholders can cause project delays and result in products that do not achieve the right goals.	The product owner is responsible for providing information about the product immediately. In addition, the scrum master helps remove impediments on a daily basis.

FIGURE 17-5: Agile product development's declining risk model.

All projects have some risk, regardless of your approach. However, with agile product management, the days of catastrophic project failure — spending large amounts of time and money with no return on investment (ROI) — are over. The elimination of large-scale failure is the biggest difference between risk on traditional projects and agile product development. In the next section, you see why.

Managing Agile Risk

In this section, you examine key structures of agile development that reduce risk over the life of the product. You find out how to use agile tools and events to find risks at the right time and how to prioritize and mitigate those risks.

Reducing risk inherently

Agile approaches, when implemented correctly, inherently reduce risk in product development. Developing in sprints ensures a short time between product investment and proof that the product works. Sprints also provide the potential for a product to generate revenue early. The sprint review, the sprint retrospective, and the product owner's involvement during each sprint provide constant product feedback to the development team. Ongoing feedback helps prevent deviations between product expectations and the completed product.

Three especially important factors in risk reduction with agile development are the definition of done, self-funding, and the idea of failing fast. You find out more about each of these factors in this section.

Risk and the definition of done

In Chapter 12, we discuss when a requirement is done. To consider a requirement complete and ready to demonstrate at the end of a sprint, that requirement must meet the scrum team's definition of done. The product owner and the development team agree upon the details of the definition; definitions of done usually include the following categories:

- **Developed:** The development team must fully create the working product requirement in an environment that reflects the one where the customer will use it.
- **Tested:** The development team must have tested that the product works correctly and is defect-free.
- **Integrated:** The development team must have ensured that the requirement works with the whole product and any related systems.
- **Documented:** The development team must have created notes about how it created the product and the rationale behind key technical decisions made.

Figure 17-6 shows a sample definition of done, with details.

FIGURE 17-6: Sample definition of done.

The product owner and the development team may also create a list of acceptable risks. For example, they may agree that end-to-end regression testing or performance testing is overkill for the sprint definition of done. Or, with cloud computing, load testing may not be as crucial because additional capacity can be easily and quickly added on demand at nominal costs. Acceptable risks allow the development team to concentrate on the most important activities.

The definition of done drastically changes the risk factor with agile approaches. By creating a product that meets the definition of done in every sprint, you end each sprint with a working, usable, and shippable product increment. Even if outside factors cause development to end early, stakeholders will always see some value and have working functionality to use now and build upon later.

Self-funding development

Agile development efforts can mitigate financial risk in a unique way that traditional projects cannot: self-funding development. Chapter 15 includes examples of self-funding development. If your product is an income-generating product, you could use that income to help fund the rest of your product.

In Chapter 15, we show you two different ROI models. Here they are again, in Tables 17-3 and 17-4. The development efforts in both tables create identical products.

In Table 17-3, the product created $100,000 in income after six months of development. Now compare the ROI in Table 17-3 to the ROI in Table 17-4.

TABLE 17-3 Income from a Traditional Project with a Final Release after Six Months

Month	Income Generated	Total Project Income
January	$0	$0
February	$0	$0
March	$0	$0
April	$0	$0
May	$0	$0
June	$0	$0
July	$100,000	$100,000

TABLE 17-4 Income from Agile Development with Monthly Releases and a Final Release after Six Months

Month/Release	Income Generated	Total Income
January	$0	$0
February	$15,000	$15,000
March	$25,000	$40,000
April	$40,000	$80,000
May	$70,000	$150,000
June	$80,000	$230,000
July	$100,000	$330,000

In Table 17-4, the product generated income with the very first release. By the end of six months, the product had generated $330,000 — $230,000 more than the project in Table 17-3.

The capability to generate income in a short amount of time has a number of benefits for companies and teams. Self-funding agile development make good financial sense for almost any organization, but they can be especially useful to organizations that may not have the funds to create a product up front. For groups short on cash, self-funding can enable product features that would otherwise not be feasible.

Self-funding also helps mitigate the risk that development will be cancelled due to lack of funds. A company emergency may dictate diverting a traditional project's budget elsewhere, delaying or cancelling the project—with nothing tangible to show for the time spent to that point. However, agile product development that generates additional revenue with every release has a good chance of continuing during a crisis.

Finally, self-funding helps sell stakeholders in the first place; it's hard to argue with a product that provides continuous value and pays for at least part of the product costs from the start.

Failing fast

All product development efforts carry some risk of failure. Testing within sprints introduces the idea of *failing fast:* Instead of sinking costs into a long effort for requirements, design, and development, and then finding problems that will prevent the product from moving forward during the testing phase, development teams identify critical problems within a few sprints. This quantitative risk mitigation can save organizations large amounts of money.

Tables 17-5 and 17-6 illustrate the difference in sunk costs for a failed waterfall project and a failed agile development effort. Both tables are for identical products with identical costs.

In Table 17-5, the stakeholders spent 11 months and close to a million dollars to find out that a product idea would not work. Compare the sunk cost in Table 17-5 to that in Table 17-6.

By testing early (daily during each two-week sprint), the development team from Table 17-6 determined that the product would not work by the end of February, spending less than one-sixth of the time and money spent in the project in Table 17-5.

REMEMBER

Because of the definition of done, product development that fails early still leaves you with something tangible that you may leverage or improve. For example, the failed development effort in Table 17-5 would have provided working functionality in the first two sprints.

The concept of failing fast can apply beyond technical problems with a product. You can also use development within sprints and fast failure to see if a product will work in the marketplace, and to end the product development early if it looks like customers won't buy or use the product. By releasing small parts of the product and testing the product with potential customers early in development, you get a good idea of whether your product is commercially viable, and save large amounts of money if you find that people will not buy the product. You also discover important changes you might make to the product to better meet customer needs.

TABLE 17-5 **Cost of Failure on a Waterfall Project**

Month	Phase and Issues	Sunk Project Cost	Total Sunk Project Cost
January	Requirements phase	$80,000	$80,000
February	Requirements phase	$80,000	$160,000
March	Requirements phase	$80,000	$240,000
April	Requirements phase	$80,000	$320,000
May	Design phase	$80,000	$400,000
June	Design phase	$80,000	$480,000
July	Development phase	$80,000	$560,000
August	Development phase	$80,000	$640,000
September	Development phase	$80,000	$720,000
October	QA phase: Large-scale problem uncovered during testing.	$80,000	$800,000
November	QA phase: Development team attempted to resolve problem to continue development.	$80,000	$880,000
December	Project cancelled; product not viable.	0	$880,000

TABLE 17-6 **Cost of Failure with Agile Techniques**

Month	Sprint and Issues	Sunk Cost	Total Sunk Cost
January	Sprint 1: No issues. Sprint 2: No issues.	$80,000	$80,000
February	Sprint 3: Large-scale problem uncovered during testing resulted in failed sprint. Sprint 4: Development team attempted to resolve problem to continue development; sprint ultimately failed.	$80,000	$160,000
Final	Development cancelled; product not viable.	0	$160,000

Finally, failing fast does not necessarily mean cancellation. If you find catastrophic issues when sunk costs are low, you may have the time and budget to determine a completely different approach to create a product.

The definition of done, self-funding development, and the idea of failing fast, along with the foundation of agile principles, all help lower risk. In the next section, you see how to actively use agile tools to manage risk.

Identifying, prioritizing, and responding to risks early

Although the structure of agile product development inherently reduces many traditional risks, development teams still should be aware of the problems that can arise during development. Scrum teams are self-managing; in the same way that they are responsible for quality, scrum teams are responsible for trying to identify risks and ways to prevent those risks from materializing.

TIP

Scrum teams prioritize the highest-value and highest-risk requirements first.

Instead of spending hours or days documenting all of the potential risks, the likelihood of those risks happening, the severity of those risks, and ways to mitigate those risks, scrum teams use existing agile artifacts and meetings to manage risk. Scrum teams also wait until the last responsible minute to address risk, when they know the most about the product and problems that are more likely to arise. Table 17-7 shows how scrum teams can use the different agile tools to manage risk at the right time.

TABLE 17-7 Agile Risk Management Tools

Artifact or Meeting	Role in Risk Management
Product vision	The product vision statement helps unify the team's definition of product goals, mitigating the risk of misunderstandings about what the product will need to accomplish. While creating the product vision, the team might think of risks on a very high level, based on marketplace and customer feedback, and inline with organizational strategy. Find out more about the product vision in Chapter 9.
Product roadmap	The product roadmap provides a visual overview of the product's requirements and priorities. This overview allows the team to quickly identify gaps in requirements and incorrectly prioritized requirements. Find out more about the product roadmap in Chapter 9.
Product backlog	The product backlog is a tool for accommodating change in the product. Being able to add changes to the product backlog and reprioritize requirements regularly helps turn the traditional risk associated with scope changes into a way to create a better product. Keeping the requirements and the priorities in the product backlog current helps ensure that the development team can work on the most important requirements at the right time. Find out more about the product backlog in Chapters 9 and 10.

(continued)

TABLE 17-7 ***(continued)***

Artifact or Meeting	Role in Risk Management
Release planning	During release planning, the scrum team discusses risks to the release and how to mitigate those risks. Risk discussions in the release planning meeting should be high-level and relate to the release as a whole. Address risks with individual requirements in the sprint planning meetings. Find out more about release planning in Chapter 10.
Sprint planning	During each sprint-planning meeting, the scrum team discusses risks to the specific requirements and tasks in the sprint and how to mitigate those risks. Risk discussions during sprint planning can be done in depth, but should only relate to the current sprint. Find out more about sprint planning in Chapter 10.
Sprint backlog	The burndown chart on the sprint backlog provides a quick view of the sprint status. This quick view helps the scrum team manage risks to the sprint as they arise and minimize their effect by addressing problems immediately. Find out more about sprint backlogs and how burndown charts show status in Chapter 11.
Daily scrum	During each daily scrum, development team members discuss roadblocks. Roadblocks, or impediments, are sometimes risks. Talking about roadblocks every day gives the development team and the scrum master the chance to mitigate those risks immediately. Find out more about the daily scrum in Chapter 11.
Task board	The task board provides an unavoidable view of the sprint status, allowing the scrum team to catch risks to the sprint and manage them right away. Find out more about task boards in Chapter 11.
Sprint review	During the sprint review, the scrum team regularly ensures that the product meets stakeholders' expectations. The sprint review also provides opportunities for stakeholders to discuss changes to the product to accommodate changing business needs. Both aspects of the sprint review help manage the risk of getting to the end of development with the wrong product. Find out more about sprint reviews in Chapter 12.
Sprint retrospective	During the sprint retrospective, the scrum team discusses issues with the past sprint and identifies which of those issues may be risks in future sprints. The development team needs to determine ways to prevent those risks from becoming problems again. Find out more about sprint retrospectives in Chapter 12.

The artifacts and meetings discussed in this section systematically help scrum teams manage risk with agile development by addressing risk by the responsible roles at appropriate times. The larger and more complex the product, the higher the likelihood that an agile approach can eliminate the risk of failure.

5 Ensuring Success

IN THIS PART . . .

Build a foundation through organizational and individual commitment to becoming more agile.

Choose your first agile product development opportunity and create an environment that will optimize agile transition success.

Simplify agile techniques across multi-team product development, enabling alignment and autonomy.

Become a change agent in your organization and help avoid both organizational and leadership pitfalls in agile transitions.

IN THIS CHAPTER

- Obtaining organizational and individual commitment
- Assembling teams with the necessary skills and abilities
- Establishing an appropriate environment
- Investing in training
- Securing initial and ongoing organizational support

Chapter 18
Building a Foundation

To successfully move from traditional project management to agile processes, you must start with a good foundation. You need commitment, both from your organization and from people as individuals, and you need to find a good team for your first pilot of agile techniques, providing the team an environment conducive to agile approaches. You want to find the right training for your team, and sustainably support your organization's agile approach so that it can grow beyond your first development effort.

In this chapter, we show you how to build a strong agile foundation within your organization.

Organizational and Individual Commitment

Commitment to agile product development means making an active, conscious effort to work with new methods and to abandon old habits. Commitment at both an individual level and at an organizational level is critical to agile transition success.

Without organizational support, even the most enthusiastic scrum team members may find themselves forced back into old project management processes. Without the commitment of individual team members, a company that embraces agile approaches may encounter too much resistance, or even sabotage, to be able to become an agile organization.

The following sections provide details on how organizations and people can support an agile transition.

Organizational commitment

Organizational commitment plays a large role in agile transition. When a company and the groups in that company embrace agile principles, the transition can be easier for the team members.

Organizations can commit to an agile transition by doing the following:

- Training leaders in skills specific to leading the transformation internally
- Engaging an experienced agile expert to create a realistic transition plan based on the current state and to guide the company through that plan
- Investing in employee training, starting with the members of the company's first scrum team and the leadership at all levels who support them
- Allowing scrum teams to abandon waterfall processes, meetings, and documents in favor of streamlined agile approaches
- Ensuring all scrum team members necessary are dedicated: An empowered product owner, a cross-functional development team of multi-skilled people, and an influential servant-leader scrum master
- Enabling development teams to continuously increase their skill sets
- Providing automated testing tools and a continuous integration framework
- Logistically supporting scrum team collocation for effective and real-time collaboration
- Ensuring that distributed teams are organized in similar time zones and making an investment in appropriate virtual tools and training when rare circumstances prevent physical collocation, such as the COVID-19 pandemic
- Allowing scrum teams to manage themselves
- Empowering product owners to make business prioritization decisions, development teams to own technical excellence decisions, and scrum masters to challenge the status quo by breaking down organizational constraints to agility

- Encouraging learning by giving scrum teams the time and freedom to go through a healthy trial-and-error process
- Revising employee performance reviews to emphasize team performance
- Encouraging scrum teams and celebrating successes

Organizational support is also important beyond the agile transition. Companies can ensure that agile processes continue to work by hiring with scrum teams in mind and by providing agile training to new employees. Organizations can also engage the ongoing support of an agile mentor, who can guide teams as they encounter new and challenging situations.

Organizations, of course, are made up of individuals. Organizational commitment and individual commitment go hand in hand.

Individual commitment

Individual commitment has an equal role to organizational commitment in agile transitions. When each person on a scrum team works at adopting agile practices, the changes become easier for everyone on the team.

People can individually commit to an agile transition by using these methods:

- Attending training and conferences and being willing to learn about agile methods
- Being open to change, willing to try new processes, and making an effort to adapt new habits
- Letting go of ego and working as part of a team, especially across traditional hierarchal and departmental boundaries
- Resisting the temptation to fall back on old processes
- Acting as a peer coach for team members who are less experienced in agile techniques
- Allowing themselves to make mistakes and learn from those mistakes
- Reflecting on each sprint honestly in the sprint retrospective and committing to improvement efforts
- Actively becoming multi-skilled development team members
- Taking responsibility for successes and failures as a team
- Taking the initiative to be self-managing
- Being active and present throughout

Like organizational commitment, individual commitment is important beyond the agile transition period. The people on the first pilot will become a reference model as well as change agents throughout the company, setting the stage and exemplifying for other teams how to effectively work with agile methods.

Getting commitment

Commitment to agile methods may not be instant. You'll need to help people in your organization overcome the natural impulse to resist change.

A good early step in an agile transition is to find an *agile champion*, a senior-level manager or executive who can help ensure organizational change. The fundamental process changes that accompany agile transitions require support from the people who make and enforce business decisions. A good agile champion will be able to rally the organization and its people around process, structure, and mindset changes.

Another important way to get commitment is to identify challenges with the organization's current development practices and provide potential solutions with agile approaches. Agile values, principles, and frameworks (such as scrum) can help address many problems, including issues with product quality, customer satisfaction, team morale, budget and schedule overruns, funding, portfolio management, and overall product issues.

Finally, highlight benefits of agile approaches to product development. Some of the real and tangible benefits that drive shifts from traditional methods of project management to agile methods include the following:

- **Happier customers:** Agile approaches often have higher customer satisfaction because scrum teams produce working products quickly, can respond to change, and collaborate with customers as partners.
- **Profit benefits:** Agile approaches allow teams to deliver functionality to market quicker than with traditional approaches. Agile organizations can realize higher return on investment, often resulting in self-funded teams.
- **Defect reduction:** Quality is a key part of agile approaches. Proactive quality measures, continuous integration and testing, and continuous improvement all contribute to higher-quality products.
- **Improved morale:** Agile practices, such as sustainable development and self-managing development teams, can mean happier employees, improved efficiency, and less company turnover.

You can find more benefits of agile product development in Chapter 21.

Can you make the transition?

You've established many valuable reasons for moving to an agile approach, and your case looks good. But will your organization be able to make the transition? Here are some key questions to consider:

- **What are the organizational roadblocks?** Does your organization have a value-delivery culture or a risk-management culture? Does it support coaching and mentorship alongside management? Is there support for training? How does the organization define success? Does it have an open culture that will embrace a high visibility of product progress?
- **How are you doing business today?** How is product development planned at the macro level? Is the organization fixated on fixed scope? How engaged are business representatives? Do you outsource development?
- **How do your teams work today, and what will need to shift under agile methods?** How ingrained is waterfall? Does the team have a strong command-and-control mentality? Can good ideas come from anywhere? Is early learning from failure accepted? Is there trust in the team? Are people shared across teams? What do you need to ask for to secure a shift? Can you get people, tools, space, and commitment to pilot the change?
- **What are the regulatory challenges?** Are there processes and procedures that relate to regulatory requirements? Are these requirements imposed upon you from externally or internally adopted regulations and standards? Will you need to create additional documentation to satisfy regulatory requirements? Are you likely to be audited for compliance, and what would be the cost of noncompliance?

As you review your analysis of the roadblocks and challenges, you may uncover the following concerns:

- **Agile approaches reveal that the organization needs to change.** As you compare agile practices and results with what you have done traditionally, you may reveal that performance has not been all it could have been. You need to tackle this head on. Your organization has been operating within a framework of how product development was expected to be run. Your organization has done its best to produce a result, often in the face of extreme challenges. For all parties involved, you have to acknowledge their efforts and introduce the potential of agile processes to allow them to produce yet greater results.
- **Leaders may misinterpret agile techniques as insufficient.** Often the values and principles of the Agile Manifesto are misinterpreted to mean agile frameworks involve insufficient planning and documentation and attempt to disregard generally accepted project management standards. Experienced

project managers may view some of that value slipping away in a transition to agile processes. Take every opportunity to clarify what agile values and principles support and do not support. Show how each principle addresses the same challenges that traditional project management attempts to resolve but in a different way.

- **Negative previous experience with attempted agile transformation.** Employees may have had a previous experience of a failed agile transformation and are turned off by the idea of trying it again. This situation can be an opportunity to return to agile values and principles. Proper training, an experienced agile mentor, and a focus on agile fundamentals are essential for transformation success.
- **Moving from a leadership to a service model can be challenging.** Agile leaders are service oriented. Command and control gives way to facilitation. Servant leadership is a big shift for many teams and functional managers. Demonstrate how the shift provides more effective outcomes for everyone. You can read more about servant leadership in Chapter 16.

Keep in mind that some resistance will arise; change can't happen without opposition. Be ready for resistance, but don't let it discourage you.

Timing the transition

Organizationally, you can start your initiative to move to an agile approach at any time. You might consider a few optimal times:

- **When you need to prove that agile product development is necessary:** Use the end of a large project, when you can see clearly what did not work (for example, during a sunset review). You'll be able to demonstrate clearly the issues with whatever failed in your approach, and you'll gain a springboard for your first agile pilot.
- **When your challenge is doing accurate budgeting:** Run your first agile development effort in the quarter before the start of the annual budget year (namely, one quarter before the end of the current budget cycle). You'll get metrics from your first development effort that will allow you to be more informed when planning next year's budget.
- **When you're starting a new product development effort:** Moving to agile processes when you have a new objective to accomplish lets you start fresh without the baggage of old approaches.

- **When you're trying to reach a new market or industry:** Agile techniques allow you to deliver quick innovation to help your organization create products for new types of customers.
- **When you have new leadership:** Management changes are great opportunities for setting new expectations with agile approaches.

Although you can take advantage of any of these opportunities to start using agile processes, they're not required. The best time to become more agile is . . . today!

Choosing the Right Pilot Team Members

Determining the right people to work with, especially in the early stages, is important to success. Here are things to think about when choosing people for the different roles in your organization's first agile pilot.

The agile champion

At the beginning of an agile transition, the agile champion will be a key person in helping ensure that the team can succeed. This person should be able to effectively and quickly influence each level of the organization that affects the pilot teams' chances for success. A good agile champion should be able to do all these tasks:

- Be passionate about agility and the organizational and market issues agile approaches will address.
- Make decisions about company processes. If there is a status quo, the agile champion should be able to influence a change.
- Get the organization excited about what's possible with agile processes.
- Regularly and directly collaborate with and support the team as it goes through the steps to establish agile processes.
- Acquire the team members necessary for success, both for the first pilot as well as for all teams in the long term.
- Be an escalation point to remove unnecessary distractions and non-agile processes.

When choosing an agile champion, look for someone who has influence in the organization — whose voice is respected and who has led change initiatives successfully in the past.

The agile transition team

As important as the agile champion is, one person can't do everything. The agile champion should work together with other organizational leaders whom the scrum team relies on for support in the transition. Together, the agile transition team removes organizational impediments to ensure the success of the pilot team and future scrum teams.

TIP

Although the agile transition team addresses organizational impediments in support of pilot teams, it makes changes to optimize not just a few local teams but the entire organizational system. This process is called *systems thinking.* The issues raised by the one pilot team inform the agile transition team of system-wide issues, and the transition team addresses those issues in ways that will benefit the entire organization.

The agile transition team should

- Be committed to organizational success through the continuous support of pilot teams.
- Establish a clear vision and roadmap for how the organization will become more agile.
- Be organized like a scrum team, with a product owner (agile champion), development team (leaders who can make organizational changes in support of the pilot scrum teams), and a scrum master (an organizational leader who can focus on helping the agile transition team adopt agile principles and coach the rules of scrum).
- Operate as a scrum team, holding all five scrum events and implementing all three scrum artifacts.

Figure 18-1 illustrates how the agile transition team's and pilot scrum team's sprint cadences are aligned. Impediments identified in the sprint retrospective of the pilot team become backlog items for the transition team to resolve as process improvements for the pilot team.

Not only does the agile transition team provide systematic support for the pilot scrum team, but the organizational leadership also becomes more agile by using scrum alongside the pilot team.

FIGURE 18-1: Alignment of the agile transition team and the pilot scrum team cadences.

The product owner

With an agile champion and an agile transition team in place, the focus turns to pilot scrum teams. The pilot scrum team product owners should come from the business side of the organization, aligning the business with technology. During the first agile development effort, the product owner may need to acclimate to working on the product daily with the development team. A good product owner should

- Be decisive.
- Be an expert about customer requirements and business needs.
- Have the business authority and be empowered to prioritize and reprioritize product requirements.
- Be organized enough to manage ongoing changes to the product backlog.
- Be committed to working with the rest of the scrum team and to being available to the development team daily throughout development.
- Have the ability to obtain product funding and other resources.

When choosing a product owner for your first agile pilot, find someone who can provide product expertise and commitment to the product. See Chapter 7 for more information about the product owner role.

The development team

With agile product development, the self-managing development team is central to the success of the product. The development team determines how to go about the work of accomplishing the product goals. Good development team members should be able to do the following:

- Be versatile in skills.
- Be willing to work cross-functionally.
- Plan a sprint and self-manage around that plan.
- Understand the product requirements and provide effort estimates.
- Provide technical advice to the product owner so that he or she can understand the complexity of the requirements and make appropriate decisions.
- Respond to circumstances and adjust processes, standards, and tools to optimize performance.

Intellectually curious developers, eager to learn new things and contribute to product goals in a variety of ways, are more likely to thrive in an agile environment. When choosing a development team for the pilot, select people who are open to change, enjoy a challenge, like to be in the forefront of new developments, and are willing to do whatever it will take to ensure success, including learning and using new skills outside their existing skill set. To learn more about the development team role, see Chapter 7.

The scrum master

The scrum master on a company's first agile product development effort may need to be more sensitive to potential development team distractions than on later efforts. A good scrum master should

- Have influence (clout).
- Have enough organizational influence to remove outside distractions that prevent the team from successfully using agile methods.
- Know enough about agile product development to be able to help the team uphold agile processes throughout development.

- Have the communication and facilitation skills to guide the development team in reaching consensus.
- Trust enough to step back and allow the development team to organize and manage itself.

When determining the scrum master for a company's first team, you want to select someone who is willing to be a servant-leader. At the same time, the scrum master will need to have a strong enough temperament to help thwart distractions, challenge the status quo, and uphold agile processes in the face of organizational and individual resistance. See Chapter 7 for details on the scrum master role.

The stakeholders

On an organization's first agile pilot, good stakeholders should

- Be involved.
- Defer to the product owner for final product decisions.
- Attend sprint reviews and provide product feedback.
- Understand agile principles. Sending stakeholders to the same training as the rest of the team will help them be more comfortable with new approaches, processes, and techniques.
- Receive product information in agile formats, such as sprint reviews, product backlogs, and sprint backlogs.
- Provide details when the product owner and development team have questions.
- Work collaboratively with the product owner and the rest of the team.

The stakeholders should be trustworthy, cooperative, and active contributors to a product.

The agile mentor

An agile mentor, sometimes called an agile coach, is key to keeping teams and organizations on track while learning scrum and beginning to establish a more agile environment. A good agile mentor should

- Be experienced (diversity in industry, business function, and role experience)
- Be an expert at agile processes, especially in the agile approaches your organization chooses.

- Be familiar with development efforts of different sizes, large and small.
- Help teams self-manage, ask questions to help them learn for themselves, and provide useful advice and support without taking over.
- Guide the team through its first sprint at the beginning and be available to answer questions as needed throughout development.
- Work with and relate to the product owner, the development team members, and the scrum master.
- Be a person from outside a department or organization. Internal agile mentors often come from a company's traditional project management group or center of excellence. If the agile mentor comes from inside the organization, he or she should be able to put aside political considerations when making suggestions and providing advice.

A number of organizations offer agile strategy, planning, and mentorship, including our company, Platinum Edge.

Creating an Environment That Enables Agility

When you're laying the foundation for adjusting your approach from traditional methods to agile methods, create an environment where agile product development can be successful and teams can thrive. An agile environment refers to not only physical environments, such as the one we describe in Chapter 6, but also a good organizational environment. To create a good agile environment, you should have the following:

- **Good use of agile processes:** This may seem obvious, but using proven agile frameworks and techniques from the beginning. Use the Roadmap to Value in Figure 18-2, using scrum and the other key agile practices to increase your chances of success. Start with the basics; build on them only when the product and your knowledge progress. Changing processes for the sake of perceived progress doesn't lead to perfection. Remember, practice doesn't make perfect; practice makes permanent. Start out correctly.
- **Unfettered transparency:** Be open about development status and upcoming process changes. People on the team and throughout the organization should be privy to product development details.

- **Frequent inspection:** Use the regular feedback loop opportunities that scrum provides to see firsthand how development is going.
- **Immediate adaptation:** Follow up on inspection by making necessary changes for improvement throughout development. Take opportunities to improve today; don't wait until the end of a release or end of development.
- **A dedicated scrum team:** The product owner, development team, and scrum master should be fully allocated to the development effort.
- **A collocated scrum team:** For best results, the product owner, development team, and scrum master should sit together, in the same area of the same office. Although collocation is preferred, consider equipment and training to make a virtual environment effective when collocation is not possible (as was the case during the COVID-19 pandemic, as discussed in Chapter 6).
- **A well-trained product team:** When the members of the team work together to learn about agile values and principles and experiment with agile techniques, they have shared understanding and common expectations about where they're headed as an agile organization.

FIGURE 18-2: The Roadmap to Value.

Luckily, many opportunities for training in agile processes are available. You can find formal certification programs as well as non-certification agile courses and workshops. Available agile certifications include the following:

- From the Scrum Alliance:
  - Certified ScrumMaster (CSM)
  - Advanced Certified ScrumMaster (A-CSM)
  - Certified Scrum Product Owner (CSPO)
  - Advanced Certified Scrum Product Owner (A-CSPO)
  - Certified Scrum Developer (CSD)
  - Certified Scrum Professional (CSP) for ScrumMasters (CSP-SM), Product Owners (CSP-PO), and Developers (CSP)
  - Certified Team Coach (CTC)
  - Certified Enterprise Coach (CEC)
  - Certified Scrum Trainer (CST)
  - Certified Agile Leadership (CAL)
- From Scrum.org:
  - Professional Scrum Master (PSM I, II, III)
  - Professional Scrum Product Owner (PSPO I, II, III)
  - Professional Scrum Developer (PSD I)
  - Professional Agile Leadership (PAL I)
- From the International Consortium for Agile (ICAgile): Various professional and expert tracks in agile facilitation, coaching, engineering, training, business agility, delivery management, DevOps, enterprise, agility, and value management
- From Kanban University
  - Team Kanban Practitioner (TKP)
  - Kanban Management Professional (KMP I, II)
  - Kanban Coaching Professional (KCP)
  - Accredited Kanban Consultant (AKC) and Trainer (AKT)

- The Project Management Institute Agile Certified Practitioner (PMI-ACP) accreditation
- Numerous university certificate programs

With a good environment, you have a good chance at success.

Support Agility Initially and Over Time

When you first launch into agile processes, give your agile transition every chance for success by paying attention to key success factors:

- **Choose a good pilot.** Select a pilot that's important enough to get everyone's support. At the same time, set expectations: Although the pilot will produce measurable improvements, the results will be modest while the team is learning new methods and will improve over time.
- **Get an agile mentor.** Use a mentor or coach to increase your chances of setting up a good agile environment and maximizing your chances of great performance.
- **Communicate — a lot.** Keep talking about agile principles at every level of the organization. Support your agile champion to encourage progress through the pilot and toward more extensive agile adaptation.
- **Prepare to move forward.** Keep thinking ahead. Consider how you'll take the lessons from the pilot to new development efforts and teams. Also think about how you'll enable agile principles and techniques from a single product to many products, including those with multiple teams.

IN THIS CHAPTER

- Identifying when and why to scale across multiple teams
- Understanding the basics of scaling
- Exploring scaling challenges

Chapter 19

De-Scaling across Teams

Depending on the schedule, scope, and required skills, many small and medium-sized product development efforts can be accomplished with a single scrum team. Larger efforts, however, may require more than one scrum team to achieve the product vision and release goals in a reasonable go-to-market time frame. When more than one scrum team is required, the teams need effective inter-team collaboration, communication, and synchronization. Regardless of the development effort's size, if interdependencies exist between multiple teams working together on the same product, or even across a collection of products, you may need to scale. Be aware, however, that scaling is an anti-pattern to agility.

With agile techniques, you decompose requirements into the most simple, independent channels of value, which enables the team to deliver working product early and continuously. Scaling is the opposite of decomposition, introducing dependencies. Rather than introducing overhead to deal with these dependencies, the goal is to simplify the breaking down of dependencies.

TIP

Scale only if you have to. Even though you may have the talent and resources available to enable multiple teams work on your product, multiple teams don't automatically ensure higher quality and faster time to market. Always look for ways to implement the tenth agile principle, "Simplicity — the art of maximizing the amount of work not done — is essential." Less is more.

As an agile framework, scrum helps teams organize their work and expose progress effectively, whether your product comprises one scrum team or one thousand scrum teams. Scaling brings new challenges, however, so you want to implement techniques for coordination and collaboration across teams that not only support agile values and principles, but also address the specific challenges facing your product and organization.

In this chapter, we discuss some of the issues to address when you need multiple teams for your product development effort. We also provide overviews of some common agile scaling frameworks and approaches that address the challenges of scaling.

Multi-Team Agile Development

Organizations determine the need for multiple scrum teams when the product backlog and release plan require a faster speed of development than a single scrum team can achieve.

With agile product development, cross-functional teams work together during every sprint, doing the same types of work each sprint and implementing requirements from the product backlog into completed, working, shippable functionality. When multiple teams work from the same product backlog, however, you have new challenges to address.

Common challenges with more than one scrum team working on the same product include the following:

- **Product planning:** Agile planning is collaborative, from the beginning. Collaboration for large groups is different than for single scrum teams. Establishing a vision with the broader team (all scrum teams and stakeholders) and building a product roadmap with collaborative input from all parties involved requires a different approach than with a single-team product.
- **Release planning:** Similar to the challenge of product planning, releases involve more specific planning of scope and release timing. Coordinating who will work on what and who will need help from whom throughout the release cycle is even more critical to ensure that dependencies, scope gaps, and talent allocation match the needs across the product.
- **Decomposition:** To break down larger requirements in the same product backlog, multiple teams may need to be involved in research and refinement discussions and activities. Who initiates these discussions? Who facilitates?

- **Sprint planning:** Although not the last opportunity to coordinate planning and execution between scrum teams, sprint planning is when scrum teams lock in a certain amount of scope from the product backlog to execute. At this stage, dependencies between scrum teams become reality. If the preceding activities of developing the product roadmap and release plan have not exposed dependencies, how can the scrum teams expose and address them at sprint planning?
- **Daily coordination:** Even after effective planning and collaboration from initiation through sprint planning, scrum teams can and should collaborate each day. Who participates and what can be done while teams are in execution mode?
- **Sprint review:** With so many teams demonstrating their product increments and seeking feedback, how can stakeholders participate with their limited schedules? How can product owners update the product backlog with all that was learned across multiple scrum teams? How do development teams know what was accomplished by other development teams?
- **Sprint retrospective:** Multiple scrum teams working together make up a broader product team. How do they identify opportunities for improvement and implement those improvements across the program?
- **Integration:** All product increments need to work together in an integrated environment. Who does the integration? Who provides the infrastructure to the teams? Who ensures the integrations work?
- **Architecture decisions:** Who oversees the architecture and technical standards? How can these decisions be decentralized to enable teams to be self-organizing and work as autonomously as possible?

These are some examples; you might be able to identify others based on your experience. Whatever your situation, select solutions to your scaling challenges that address your specific challenge.

TIP

Some scaling frameworks offer solutions to challenges you may not have. Be careful not to bloat your framework fixing things that are not broken.

Since the first scrum teams in the mid-1990s, there have been agile products requiring multiple scrum teams collaborating effectively. Following are overviews of various scaling frameworks and techniques addressing many of these challenges.

Making Work Digestible through Vertical Slicing

One of the simplest scaling approaches is known as vertical slicing, which provides a straightforward solution for dividing the work across teams so they can incrementally deliver and integrate functionality at every sprint. If your scaling challenge is breaking down the work across teams, vertical slicing is a solution.

REMEMBER

The concept of vertical slicing applies to single team product development efforts, too. Development teams consist of people who collectively possess all skills required to turn a requirement into completed, shippable functionality. The development team swarms on one requirement at a time, which is a vertical slice of the product backlog, touching all aspects of the technology and skills required.

Highly aligned and autonomous scrum teams do what needs to be done to meet customer needs. Development teams working on the same product are each cross-functional enough to work on any item from the product backlog. Each team pulls items from the same product backlog and implements on the same aligned cadence. If one team lacks a skill required for a requirement, they work with teams who do to expand their capability, reducing constraining dependencies.

During each sprint, each team integrates their work with the work of other teams, ideally through automation to make integrations less costly and troublesome. At the end of each sprint, all product backlog items selected during the sprint are elaborated, designed, developed, tested, integrated, documented, and approved.

Figure 19-1 illustrates how a product can be vertically sliced to enable multiple teams to work on a single product.

With a common baseline definition of done for all scrum teams, vertical slicing enables each team to be self-organizing in their development approach, yet aligned with the overall release goal. The product vision and roadmap become the glue keeping everyone together.

Scrum of scrums

Vertical slicing illustrates how scrum teams coordinate and integrate technically. How do these different scrum teams coordinate with each other daily? The *scrum of scrums* model is one way of facilitating effective integration, coordination, and collaboration among the people that make up scrum teams. Most scaling frameworks we show you in this chapter use scrum of scrums to enable daily coordination between scrum teams.

FIGURE 19-1: Vertical slices of product features implemented by multiple scrum teams.

Figure 19-2 illustrates each role of each team coordinating daily with people of the same role in other teams regarding priorities, dependencies, and impediments that affect the broader program team. The scrum of scrums for each role is facilitated by someone designated and empowered by the group involved. Thorough integration and release efforts establish a consistent and regular scrum of scrums model.

FIGURE 19-2: Scrum of scrums for coordinating between scrum teams.

Each day, individual scrum teams hold their own daily scrums at approximately the same time, in separate locations. Following these daily scrums, the scrum of scrums meetings described next occur.

Product owner scrum of scrums

Each day or as often as needed, following the individual scrum teams' daily scrums, the product owners from each scrum team meet for no longer than 15 minutes. They address the priorities at play and make any adjustments based on the realities uncovered during the individual scrum team's daily scrum. Each product owner may address the following:

- The business requirements that each has accepted or rejected since the last time they met
- The requirements that should be accepted by the time they meet again
- Which requirements are impeded and need help from other teams to resolve (such as "John, we won't be able to do requirement 123 until you complete requirement xyz from your current sprint backlog.")

TIP

In effect, the product owners form a band of aligned, self-organizing product owners and, similar to their own scrum teams, may huddle around a high-level summary view of what each scrum team is working on. Figure 19-3 is an example of what the product owners scrum of scrums board may look like.

FIGURE 19-3: Scrum of scrums task board.

Development team scrum of scrums

Each day following the individual scrum teams' daily scrums, one development team member representative from each scrum team attends the development teams scrum of scrums for no longer than 15 minutes to discuss the following:

- Their team's accomplishments since the last time they met
- Their team's planned accomplishments between now and the next meeting, and how their collective work will be integrated
- Technical concerns with which they need help
- Technical direction decisions that the team has made and what anyone should be aware of to prevent potential issues

TIP

Consider rotating development team members from the individual scrum teams who attend the scrum of scrums, either daily or for each sprint, to ensure that everyone stays tuned in to the efforts of the overall product development.

Scrum master scrum of scrums

The scrum masters from each scrum team also meet with the other scrum masters for no longer than 15 minutes to address the impediments that each team is dealing with. Each scrum master addresses the following:

- The individual team-level impediments resolved since the last time they met and how they were resolved, in case other scrum masters run into the issue
- New impediments identified since last time they met and any unresolved impediments
- Which impediments they need help resolving
- Potential impediments that everyone should be aware of

Impediments needing escalation are discussed and addressed after the daily scrum of scrums. Similar to the product owners and development team members, the scrum masters form their own band of aligned and self-organizing scrum masters.

REMEMBER

Your organization should have standards that guide and empower individual teams to make tactical decisions. This way, each team doesn't have to reinvent the wheel.

Vertical slicing is a simple way to maintain the autonomy of each scrum team to deliver valuable functionality within a wider program context. It is also effective at helping teams have timely and relevant conversations about constraints and progress.

Multi-Team Coordination with LeSS

Large-scale scrum (LeSS) is another way to scale scrum across massive product development efforts. *LeSS* is based on principles that support keeping scrum simple when putting multiple scrum teams together to work on the same product backlog. LeSS focuses on system optimization so that each scrum team can be versatile in their ability to contribute effectively to the overall product backlog. It also presents a variety of options and approaches for addressing each scaling challenge. In this section, we present an overview and then cover a few options that stand out.

LeSS defines two framework sizes: LeSS and LeSS Huge. The difference lies in the size of the total teams involved.

LeSS, the smaller framework

Figure 19-4 illustrates the basic LeSS framework, using three scrum teams as an example. LeSS recommends no more than eight scrum teams follow the basic model.

FIGURE 19-4: Basic LeSS framework.

Used with permission, Craig Larman and Bas Vodde

LeSS outlines how scrum teams work together one sprint at a time, starting with sprint planning, followed by sprint execution and daily scrums, and ending with sprint review and sprint retrospective. Although much of LeSS remains true to the scrum framework, the following significant differences exist:

» In LeSS, scrum masters typically work with one to three teams, and there is only one product owner for up to eight teams.

REMEMBER

Dedicating scrum team members to one team eliminates the overhead of frequent cognitive demobilization and remobilization due to context switching. Always be aware of the risks of splitting the focus of team members across multiple teams.

- Sprint planning (Part 1) does not require all developers to attend, but at least two members per scrum team, along with the product owner, attend. The representative team members then go back and share their information with their teams.
- Independent sprint planning (Part 2) and daily scrum meetings occur, and members from different teams can attend each other's meeting to facilitate information sharing.
- Sprint reviews are usually combined across all teams.
- Overall sprint retrospectives are held in addition to individual team retrospectives. Scrum masters, product owners, and representatives from development teams inspect and adapt the overall system of the product, such as processes, tools, and communication.

LeSS Huge framework

With LeSS Huge, a few thousand people could work on one product development effort. But the structure remains simple.

The scrum teams are grouped around major areas of customer requirements, called requirement areas. For each requirement area, you have one area product owner and between four and eight scrum teams (a minimum of four teams in each requirement area prevents too much local optimization and complexity). An overall product owner works with several area product owners, forming a product owner team for the product. Figure 19-5 illustrates LeSS Huge.

FIGURE 19-5: The LeSS Huge framework.

Used with permission, Craig Larman and Bas Vodde

As in scrum at a single team level, as well as in basic LeSS, you have one product backlog, one definition of done, one potentially shippable product increment, one product owner, and one sprint cadence across teams. LeSS Huge is simply a stacking of multiple parallel LeSS implementations for each requirement area.

To enable these teams to work together effectively across the requirement areas

- The product owner regularly coordinates with the area product owners.
- Requirement areas are flagged in the product backlog to identify who is planning to work on which parts of the product.
- A set of parallel sprint meetings is needed per requirement area. Overall sprint reviews and retrospectives involving all teams are necessary to enable continuous inspection and adaptation beyond single teams. These multi-team events help coordinate the overall work and process across the program.

With the exception of limiting opportunities for developers to work closely with business people (the product owner) on a daily basis, LeSS provides a simple way for scaling scrum across the product development effort.

REMEMBER

LeSS, just like single-team scrum, relies heavily on development teams with business domain knowledge and access to customers and business stakeholders empowered to elaborate requirements in collaboration directly with customers and the product owner. This collaboration is even more crucial in LeSS, where multiple teams share access with only one product owner.

We also find the flexibility of coordination techniques suggested in LeSS to be effective for teams addressing their specific multi-team coordination challenges. In addition to a scrum of scrums (discussed earlier in this chapter) and continuous integration (see Chapter 5), LeSS suggests several options for scrum teams coordinating with other scrum teams, a few of them highlighted in the following sections.

Sprint review bazaar

Multiple teams work toward the same product increment in each sprint, so all teams have something to demonstrate, and all teams need stakeholder feedback for updating their portion of the product backlog. Because all scrum teams are on the same cadence, even a LeSS basic organization would involve a lot of sprint review meetings for stakeholders to attend on the same day.

LeSS recommends a diverge-converge pattern to the sprint review, similar to a science fair or bazaar format. Each scrum team sets up in one part of a room large

enough to accommodate all scrum teams. Each scrum team demonstrates what it did during the sprint, collecting feedback from the stakeholders visiting its area. Stakeholders visit their areas of interest. Scrum teams may loop through their demonstrations a few times to accommodate stakeholders visiting multiple teams. This approach also allows scrum team members to see demonstrations of other scrum teams. Note that combined sprint reviews can be held in other ways.

Combining sprint reviews increases transparency and collaboration culture across scrum teams.

Observers at the daily scrum

Although daily scrums are conducted so that the scrum team can coordinate their work for the day, anyone is invited to listen. Transparency is key for agility. The scrum of scrums model described previously in this chapter is a participatory model — developers attending the integration scrum team daily scrum participate in the discussion. However, sometimes other scrum team members just need to be aware of what other teams are doing.

A representative of the development team from one team may attend the daily scrum of another team, observe, and then report back to his or her own team to determine any action to take. This can be a non-disruptive way for other scrum teams to be involved without extra meeting time overhead.

Component communities and mentors

LeSS takes a vertical slicing approach also to dividing up the product backlog across teams, so multiple teams may "touch" the same system or technology components. For instance, multiple teams may work in a common database, user interface, or automated testing suite. Setting up a community of practice (CoP) around these areas gives these people a chance to collaborate informally on the component areas where they spend most of their time.

A CoP usually organized by someone from one of the scrum teams who has the knowledge and experience to teach people how the component works, monitor the component long-term, and engage the community in regular discussions, workshops, and reviews of work being done in the component area.

Multi-team meetings

Similar to the combined sprint review model, LeSS scrum teams may benefit from meeting together for other scrum planning events and activities. Product backlog

refinement, sprint planning part two, and other design workshops are some examples. LeSS recommends similar formats for each situation, common elements of which include the following:

- An overall session first, shared among all teams, to identify which teams are likely to take on which product backlog items.
- Representatives of each team attend overall sessions (all can attend, but attendance is not required).
- Team-level sessions follow overall sessions to dive into details.
- Multi-team breakouts follow overall sessions, as needed, with just those teams involved.

The key to these sessions is that they are face to face, in the same room, allowing for real-time collaboration to break down dependencies. For distributed LeSS groups (one team in one geographic location and other teams in other locations), videoconferencing is key.

Travelers

The more versatile your development team, the fewer bottlenecks your scrum team will experience. Traditional organizations have specialists in technical areas, and there are not enough of them to go around to all the scrum teams when starting an agile transition. To begin bridging skill gaps across teams, technical experts can become travelers, joining scrum teams to coach and mentor in their area of expertise through pairing (see Chapter 5), workshops, and teaching sessions.

As this expertise is shared, the expert mentor continues to lead and grow the skills across the organization (as a CoP organizer). In addition, scrum teams increase their cross-functionality and can develop more efficiently. The traveler is careful to ensure that accountability for the product remains with the development team.

Aligning through Roles with Scrum@Scale

Agile scaling models vary in complexity and simplicity. The Scrum@Scale approach for two or hundreds of scrum teams working together is a form of basic scrum of scrums model for scrum masters and product owners, coordinating communication, impediment removal, priorities, requirement refinement, and planning. The scrum of scrums model for the scrum master and product owner roles makes this daily synchronization possible between teams across programs of varying sizes.

WARNING

According to the Scrum@Scale guide, if an organization can't scrum, it can't scale. Building on a solid foundation of empowered, small, self-organizing teams, the agile values and principles, and scrum values and scrum itself enables the most effective scaling.

The scrum master cycle

Scrum@Scale groups at most five scrum teams into a scrum of scrums that work together to deliver a product. With more complex products requiring additional scrum teams, more than one scrum of scrums may be needed, with at most five scrum teams each. Daily representatives from each scrum team (at least the scrum masters) attend a scaled daily scrum meeting. It mirrors the daily scrum for individual scrum teams, inspecting the collective progress and surfacing and removing impediments.

With Scrum@Scale, reducing the number of people involved in a scrum of scrums limits the communication complexities for effective cross-team collaboration. It provides better visibility into what the scrum teams are working on and how their work might affect each other.

Figure 19-6 illustrates the Scrum@Scale scrum of scrums model.

FIGURE 19-6: Scrum@Scale scrum of scrums model.

With product development efforts of more than five scrum teams, Scrum@Scale implements a scrum of scrums of scrums, where a representative of each scrum of scrums attends with four other scrum of scrums representatives to surface and remove impediments at the scrum of scrums of scrums level.

Figure 19-7 illustrates the Scrum@Scale scrum of scrums of scrums model.

FIGURE 19-7: Scrum@Scale scrum of scrums of scrums model.

©1983-2020 Jeff Sutherland & Scrum, Inc.

The executive action team (EAT), shown in Figure 19-8, fulfills the scrum master role for the entire organization, providing an agile ecosystem for enabling the scrum teams to function optimally. The focus of the EAT is ensuring that agile values and scrum are implemented effectively and the organization is optimized for it.

FIGURE 19-8: Scrum@Scale executive action team (EAT).

©1983-2020 Jeff Sutherland & Scrum, Inc.

The product owner cycle

Each scrum of scrums has a common shared product backlog. The product owners organize in a similar and aligned way as the scrum masters in the scrum of scrums, only instead of labeling them as scrum of scrums, they are a product owner team.

A product owner team — including a chief product owner (CPO) and the product owners of each scrum team — is responsible for ordering the product backlog and ensuring that individual scrum teams align with the priorities. The CPO may be one of the individual product owners or someone else dedicated to it.

Each scrum team has a product owner and focuses on prioritization of their scrum team's sprint backlog. The product owner team communicates the overarching product vision.

Figure 19-9 illustrates the Scrum@Scale product owner team.

©1983-2020 Jeff Sutherland & Scrum, Inc.

FIGURE 19-9: Scrum@Scale product owner team.

To coordinate product development for the entire organization, CPOs meet with executives and key stakeholders in an *executive metascrum (EMS)* event, which is the forum for collaborating and alignment on priorities, budget and maximizing delivery of value.

Figure 19-10 illustrates the Scrum@Scale executive metascrum (EMS).

FIGURE 19-10: Scrum@Scale executive metascrum (EMS).

©1983-2020 Jeff Sutherland & Scrum, Inc.

Synchronizing in one hour a day

In an hour or less per day, an entire organization can align priorities for the day and accomplish effective coordination of impediment removal. For instance, at 8:30 a.m., each individual scrum team holds their daily scrums separately. At 8:45 a.m., the scrum masters hold their scrum of scrums and the product owner team associated with each scrum of scrums meet. At 9:00 a.m., if a scrum of scrums of scrums exists, scrum of scrums masters meet, and the product owners meet to coordinate priorities at the next level. Finally, at 9:15 a.m., EAT and EMS meet to coordinate and address any issues that need to be resolved for the scrum teams.

Joint Program Planning with SAFe

Scaled Agile Framework (SAFe®) is used to scale scrum and agile principles across multiple layers of an IT and software or systems development organization. Figure 19-11 shows the full SAFe 5.0 big picture.

Organizations can choose from four SAFe configurations:

- **Full SAFe** is the most complex configuration, supporting the building of large integrated solutions requiring hundreds of people or more. Refer to Figure 19-11.

FIGURE 19-11: SAFe for Lean Enterprises 5.0.

Reproduced with permission from ©2011-2020 Scaled Agile, Inc. All rights reserved. SAFe and Scaled Agile Framework are registered trademarks of Scaled Agile, Inc.

- **Portfolio SAFe** adds portfolio and investment funding, portfolio operations, and lean governance across the organization.
- **Large Solution SAFe** is for building larger and more complex solutions but does not require portfolio-level constructs.
- **Essential SAFe** is a basic starting point for smaller organizations and consists of the minimal elements necessary. See Figure 19-12.

FIGURE 19-12: Essential SAFe configuration.

Reproduced with permission from ©2011-2020 Scaled Agile, Inc. All rights reserved. SAFe and Scaled Agile Framework are registered trademarks of Scaled Agile, Inc.

SAFe is focused around seven core competencies of the lean enterprise:

- **Lean Agile leadership:** Leadership skills to drive and sustain organizational change by empowering people and teams to reach their potential
- **Team and technical agility:** Driving team agile behaviors as well as sound technical practices including quality and development practices
- **Agile product delivery:** Building high-performing collaborative teams of teams that use design thinking and customer focus to provide continuous flow of valuable products with various automation toolsets
- **Enterprise solution delivery:** Building and sustaining the world's largest enterprise cyber-solutions, software products, and networks
- **Lean portfolio management:** Executing portfolio vision and strategy formulation, chartering portfolios, prioritizing, and roadmapping
- **Organizational agility:** Aligning strategy and execution by applying lean and systems thinking to strategy and investment funding, portfolio operations, and governance
- **Continuous learning culture:** Continually increasing knowledge and competency and performance by becoming an organization committed to learning and innovating

Although other scaling frameworks have tactical differences, they also have many similarities, such as the following:

- Development is done by development teams.
- Teams are aligned in sprint length and cadence.
- A scrum of scrums for coordination.

We don't go into all details of SAFe here, but we do highlight a few practices that address some of the scaling challenges discussed previously in this chapter.

SAFe introduces several new roles, titles, and structure names. In all SAFe configurations, you'll find at the team of teams level the *agile release train (ART) model*, which is a team of multiple agile teams (50–125 people in total) delivering incremental releases of value. The ART provides a fixed cadence with which the teams align and synchronize. The rest of the organization, knowing this cadence, can also reliably plan its work around this known release train schedule.

ARTs are driven by a release train engineer (RTE), product management, and a system architect/engineer. The RTE acts as the scrum master of the release train.

Product management leadership provides product ownership guidance. The system architect/engineer provides technical leadership for the release train. The ART works at a cadence of five iterations by default to create what is known as a *program increment (PI)*, which is similar to the product increment concept in scrum.

WARNING

Scaled frameworks have the potential to allow your organization to remain unchanged in the spirit of reducing organizational disruption. The reality may be that the organization needs to change to better meet the needs of your customer. Choose from the scaled frameworks what makes the most sense for your organization.

Joint program increment planning

Joint program increment (PI) planning unifies agile teams across an ART. In PI planning, teams plan their work for the next PI together, face-to-face, in the same room at the same time.

PI planning includes the following:

- Setting business context for the PI by a senior executive or business owner.
- Communicating program vision by product management, and supporting features from the backlog.
- Presenting the system architecture vision and any agile-supportive changes to development practices (such as test automation).
- Outlining the planning process by the RTE.
- Setting up team breakout sessions to determine capacity and backlog items that they will work on in support of the program vision.
- Reviewing draft plans with all teams, with each team presenting key planning outputs, potential risks, and dependencies. Product management and other stakeholders provide input and feedback.
- Reviewing draft plans by management to identify any issues with scope, talent allocation constraints, and dependencies. Facilitated by the RTE.
- Breaking out of teams to adjust planning based on all feedback.
- Reviewing the final plan, facilitated by the RTE.

The magic of PI planning is that dependencies are identified and coordinated in the moment during the event. If one team identifies a dependency in one of its own requirements during the team breakouts, that team sends a team member to another team to discuss the dependency right there and then. No back-and-forth occurs.

Although no amount of planning can identify every issue up front, this type of collaboration addresses most issues ahead of time. In addition, it establishes an open line of communication throughout the program increment execution, ensuring teams synchronize and address issues immediately and more effectively than if they had planned as separate teams, sharing documentation without discussion.

Clarity for managers

In Chapters 3 and 16, we discuss ways management changes to enable teams to be more agile and adaptive in nature. For larger organizations, SAFe provides structure for middle management's involvement with agile teams. The portfolio and large solution configurations outline roles and functions not fulfilled by individual team members, providing some clarity to how functional, technical, and other leadership types can clear the way and enable the individual teams to be as effective and efficient as possible, as well as connect strategy to execution.

WARNING

If layers of leadership are added to your product development effort, be careful to protect the role of the product owner and his or her proximity to the customer. Many scaled teams have inadvertently fallen into the trap of disempowering their scrum teams.

Disciplined Agile Toolkit

Project Management Institute's (PMI) Disciplined Agile (DA) is based on the premise that true business agility comes from freedom, not frameworks. DA is a toolkit with hundreds of practices that guide teams and organizations to help improve the way they work.

Disciplined Agile is a learning-oriented hybrid approach to information technology (IT) solution delivery. It claims to be focused on delivering both risk and value, enterprise aware, and scalable.

Disciplined Agile's foundation layer outlines the principles (including agile and lean), guidelines, roles, and teams and how they work together.

Disciplined DevOps makes up the second layer where streamlining of software development and IT operations activities occurs. We discuss DevOps in Chapter 10.

Layer 3 is defined as value streams, based on the FLEX workflow, which tie the organization's strategies and enabling decisions for improving each part of the

organization in the context of the entire system. The goal is not just to be innovative but also to increase value realization through the following:

- Research and development
- Business operations
- Portfolio management
- Product management
- Strategy
- Governance
- Sales and marketing
- Continuous improvement

Disciplined Agile Enterprise (DAE), layer 4 of DA, is about the ability to sense and respond to changes in the marketplace through organizational culture and structure. The DAE layer focuses on the rest of the organization's enterprise activities that support its value streams, including

- Enterprise architecture
- People management
- Asset management
- Finance
- Vendor management
- Legal
- IT
- Transformation

Disciplined Agile is a hybrid toolkit that builds on the foundation of other methods and frameworks, such as DevOps, extreme programming (XP), scrum, SAFe, kanban, and others. The frameworks provide the process bricks while DA provides the mortar to hold the bricks together.

As mentioned at the beginning of this chapter, scaling is an anti-pattern. The best way to prevent the need for scaling is to enable your teams to become highly aligned and highly autonomous. Break down work and features into the smallest and most valuable increments.

Conway's Law (by Melvin Conway, who introduced the law in 1967) states that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." Similarly, your product is a reflection of your organization, so organize as simply as possible.

Cultural changes such as the adoption of agile techniques require organization-wide commitment to long-term changes in mindset and structure, which we discuss next in Chapter 20.

IN THIS CHAPTER

- Understanding change management issues and common change management models
- Following steps for agile adoption in your organization
- Avoiding common problems in adopting agile principles
- Leading change by example

Chapter 20

Being a Change Agent

If you're contemplating the idea of introducing agile product development to your company or organization, this chapter can help get you started on those changes. Introducing agility means learning and practicing a new mindset, culture, organizational structures, frameworks, and techniques. In this chapter, you learn key principles and steps to implementing agile product development techniques. We also introduce common change models, including the model we use at our company, Platinum Edge. We also cover leading change by example, as well as common pitfalls to avoid in your agile transition.

Becoming Agile Requires Change

Traditional approaches to project management focus significantly on processes, tools, comprehensive documentation, contract negotiation, and following a plan. Although agile product development remains dedicated to addressing each of these, the focus shifts to individuals, interactions, working functionality, customer collaboration, and responding to change.

Waterfall organizations didn't get where they are overnight and won't change overnight. For some organizations, decades of forming habits, establishing and protecting fiefdoms, and reinforcing a traditional mindset are engrained. The

organizational structure will require some type of change, the leadership will need to learn a new way of looking at developing people and empowering them to do their work, and those doing the work will have to learn to work together and manage themselves in ways that may feel foreign.

Why Change Doesn't Happen on Its Own

Change is about people more than it is about defining a process. People resist change, and that resistance is based on personal experience, emotion, and fear. David Rock developed a brain-based model for identifying five key domains that influence our behavior in social situations, often involving change. It's called the SCARF model, which stands for

- Status: Our relative importance to those around us
- Certainty: Our ability to predict what's coming
- Autonomy: Our sense of control
- Relatedness: How safe we feel around others
- Fairness: How fair we perceive the exchanges between people

Each of these domains activates a threat or reward response in our brain, which is what we rely on for survival and is reflected in our behavior. You might be able to identify which of these domains influences you to feel threatened. Change is difficult for each of us, for a myriad of reasons.

We see these reactions firsthand as we help organizations make these changes. As consultants and trainers, often our first exposure to an organization is when it asks for formal classroom training to learn what it means to be agile and how scrum works. After a two-day class, the level of excitement about implementing this more modern way of thinking and working usually increases, and our students consistently express how much it makes sense. Excitement isn't enough for change.

Scrum is simple. Agile values and principles resonate with almost everyone. But none of it is easy. Scrum for developing products and services is like playing a new game, with new positions, new rules, and a different playing field. Imagine that an American football coach came to his team one day and said, "We're going to learn how to play futbol (American soccer) today. Meet me out on the pitch in 15 minutes with your gear, and we'll get right to work." What would happen? Everyone might know how to play futbol based on what he or she had seen on TV or experienced as a youth, but the team wouldn't be ready to make the change.

A lot of confusion would ensue. Old rules, techniques, training, and thinking would have to be unlearned for the team to learn the new stuff and come together to compete effectively. Immediately, you'd hear questions from the players, such as the following:

- When can I use my hands?
- How many timeout calls do we get?
- Am I on offense or defense during this play?
- Where do I line up at kick-off?
- Who holds the ball when we kick a goal?
- Where's my helmet?
- These shoes make it hard to kick sometimes.

Transitioning to agile techniques won't happen overnight, but it will happen if you and your organization's leadership take a change management approach to your agile transition. For existing waterfall organizations, agile transformation takes at least one to three years from the time management commits to it. Commitment doesn't just mean writing a check for the training and coaching. Commitment means that leaders start learning how to lead the change instead of outsourcing it to a consultant. It's an ongoing journey, not a destination.

Strategic Approaches to Implementing and Managing Change

Organizational change initiatives typically fail without a strategy and discipline. Here, we define *failure* as not reaching the desired end state goal of what the organization will look like after the change. Failure is often due to being unclear as to the goal or because the change plan doesn't address the highest risk factors and challenges impeding the desired change.

Various approaches exist to managing change. We show you several here, including ours (Platinum Edge), so you know what to expect as you embark on your own change initiative.

Lewin

Kurt Lewin was an innovator in social and organizational psychology in the 1940s and established a cornerstone model for understanding effective organizational change. Most modern change models are based on this philosophy, which is unfreeze — change — refreeze, as illustrated in Figure 20-1.

FIGURE 20-1: Lewin's unfreeze, change, refreeze change philosophy.

If you want to change the shape of a cube of ice, you first have to change it from its existing frozen state to liquid so that it can be changed or reshaped, then mold the liquid into the new shape you want, then put it through a solidification process to form the new shape. Unfreezing is implied between the first two states in the figure, and the changes made are implied during the unfrozen state.

Unfreeze

The first stage represents the preparation needed before change can take place — challenging existing beliefs, values, and behaviors. Reexamination and seeking motivation for a new equilibrium is what leads to participation and buy-in for meaningful change.

Change

The next stage involves uncertainty and resolving that uncertainty to do things a new way. This transitional stage represents the formation of new beliefs, values, and behaviors. Time and communication are the keys to seeing the changes begin to take effect.

Refreeze

As people embrace new ways, confidence and stability increase, and the change starts to take shape into a solid new process, structure, belief system, or set of behaviors.

This simple pattern provides the foundation for most change management tools and frameworks, including those we discuss in this chapter.

ADKAR's five steps to change

Prosci is one of the leading organizations in change management and benchmarking research. One of Prosci's change management tools, ADKAR, is an acronym for the five outcomes (awareness, desire, knowledge, ability, and reinforcement) individuals and organizations need to achieve for successful change. It is a goal-oriented model for individuals, and a focus model for the discussions and actions organizations need to take together.

Organizational changes still require change for individuals, so the secret to success is affecting change for everyone involved.

ADKAR outlines the individual's successful journey through change. The five steps of the journey also each align with organizational change activities. Typically, these steps are completed in the order listed, but a non-linear approach is realistic in our experience. You may need to readdress previous steps multiple times as you progress through each step.

Awareness

Humans find change difficult. When change initiatives come top-down in an organization, people may verbally agree to them, but their actions tell a different story. Mismatch of actions and words is usually innocent and natural. Without awareness, or an understanding of the factors influencing management's desire to change, or especially without a recognition that something should change, individuals will not be motivated to change. Informing the individuals in an organization, helping them have a shared understanding of the challenges that exist, and then assessing whether awareness is common constitute the first step to successful, lasting change. It is the basis, without which the initiative won't make progress.

Desire

Based on their awareness of a challenge needing to be addressed, individuals will have an opinion on whether or not change is necessary or desired to address it. Making the connection between the awareness of an issue and what could or should be done about it is the next step. Once desire exists for the individuals in an organization, there is motivation to move together to change.

Knowledge

Desire is key, but it won't result in change by itself. Knowledge of how to make the change and where each individual fits into the change make up the next crucial part of the change process. Individuals throughout the organization need to understand what the changes mean for them, and leadership needs to facilitate education and actions in a cooperative way across the organization. Knowledge often comes from expanding understanding and skills through training and coaching.

Ability

With new knowledge of how to change, implementation requires acquiring skills, redefining roles, and clearly defining new performance expectations. Other commitments may need to be delayed or replaced with new behaviors or responsibilities. Continued coaching and mentoring may be required, and leadership needs to be clear that this reprioritization of commitments is expected and encouraged.

Reinforcement

Changes don't stick after one successful iteration. New behaviors, skills, and processes must be reinforced through continued corrective action and coaching to ensure that old habits don't return.

The ADKAR model surrounds these steps with assessments and action plans to guide leaders and individuals through their change journey. ADKAR should be used iteratively, using scrum, inspecting and adapting each step.

Kotter's eight steps for leading change

John Kotter's eight-step process for leading change identifies eight common but preventable reasons why organizations fail at their change initiatives, and addresses each with actions that should be taken to successfully lead change.

- **Create a sense of urgency.** The leadership action is to create a sense of urgency to pull people out of complacency. People get used to the status quo, and learn to deal with it. Helping others see the need for change requires the creation of a sense of urgency for change. Leaders must communicate the importance of immediate action.
- **Build a guiding coalition.** The leadership action is to build a guiding coalition. Successful change will require more than just one active supporter, even if that one person is at the highest level of the organization. Executives, directors, managers, and even informal social leaders with influence need to be unified in the need for and vision of a change. This coalition must be formed and drive the change.
- **Form a strategic vision and initiatives.** Kotter estimates that leadership undercommunicates the vision for change by as much as 1,000 times. Even if people are unhappy with the status quo, they won't always make sacrifices for a change unless they believe in the proposed benefits and that change is possible. As a change coalition, clearly define how the future is different from the past and present, as well as the steps to make that future a reality. We discuss visions and roadmaps for products and services in Chapter 9 — change management also needs to begin with a clear vision of where you're headed.

- **Enlist a volunteer army.** The leadership action is to enlist a volunteer army. Change will accelerate and last if massive numbers of people buy in and are internally driven. As a result of leadership's effective communication of vision and need, people should rally around a cause they come to believe in. If they don't rally, reevaluate your messaging, tone, and frequency.
- **Enable action by removing barriers.** The leadership action is to remove barriers to action. Some obstacles may be only perceived, but others are real. However, both must be overcome. One blocker in the "right" place can be the single reason for failure. Many people tend to avoid confronting obstacles (processes, hierarchies, working across silos), so leadership must act as servant-leaders to identify and remove impediments that are reducing the empowerment of individuals implementing the changes on the front lines.
- **Generate short-term wins.** The leadership action is to generate short-term wins. The end transformation goal usually can't be achieved in the short term, so fatigue can set in for everyone involved if successes and progress go unrecognized along the way. Evidence of change should be highlighted and exposed early and regularly. This reinforcement increases morale through difficult times of change, and motivates and encourages continued efforts and progress.
- **Sustain acceleration.** The leadership action is to sustain acceleration. Celebrating short-term wins sets a false sense of security that change is complete. Each success should build on the previous success. Push on, and push on harder after each success, with increased confidence and credibility. Continue to overcommunicate the vision throughout the transformation.
- **Institute change.** The leadership action is to institute change. Leadership will have the opportunity throughout the change process to connect successes and new behaviors with the culture's evolution and growing strength to keep old habits from returning. These connections should be recognized openly and made visible to everyone as soon as successes and new behaviors are realized.

Platinum Edge's Change Roadmap

Throughout this book, we highlight the fact that agile processes are different from traditional project management. Moving an organization from waterfall to an agile mindset is a significant change. Through our experience guiding companies

through this type of change, we've identified the following important steps to successfully become an agile organization.

Figure 20-2 illustrates our agile transition roadmap for successful agile transformation.

FIGURE 20-2: Platinum Edge agile transition roadmap.

Step 1: Conduct an agile audit to define an implementation strategy with success metrics

An *agile audit* of your organization is

- A three- to five-week review of the existing project management, product development, corporate structure, objectives, and culture
- Identification of opportunities to improve efficiency, effectiveness, and agility
- Creation and presentation of an implementation strategy and roadmap

An *implementation strategy* is a plan that outlines the following:

- Your current strengths to build on as you transition
- The challenges you'll face based on your current structure
- Action items for how your organization will transition to agile product development

TIP

Implementation strategies are most effectively performed by external agile experts in the form of an assessment or a current state audit.

Whether you engage with a third party or conduct the assessment yourself, make sure the following questions are addressed:

- **Current processes:** How does your organization develop products today? What does it do well? What are its problems?
- **Future processes:** How can your company benefit from agile approaches? What agile methods or frameworks will you use? What key changes will your organization need to make? What will your transformed company look like from a team and process perspective?
- **Step-by-step plan:** How will you move from existing processes to agile processes? What will change immediately? In six months? In a year or longer? This plan should be a roadmap of successive steps getting the company to a sustainable state of agile maturity.
- **Benefits:** What advantages will the agile transition provide for the people and groups in your organization and the organization as a whole? Agile techniques are a win for most people; identify how they will benefit.

- **Potential challenges:** What will be the most difficult changes? What training will be required? What departments or people will have the most trouble with agile approaches? Whose fiefdom is being disrupted? What are your potential roadblocks? How will you overcome these challenges?
- **Success factors:** What organizational factors will help you while switching to agile processes? How will the company commit to a new approach? Which people or departments will be agile champions?

A good implementation strategy will guide your company through its move to agile practices. A strategy can provide supporters with a clear plan to rally around and support, and it can set realistic expectations for your organization's agile transition.

For your first agile product development effort, identify a quantifiable way to recognize success. Using metrics will give you a way to instantly demonstrate success to stakeholders and your organization. Metrics provide specific goals and talking points for sprint retrospectives and help set clear expectations for the team.

TIP

Metrics related to people and performance work best when related to teams rather than to individuals. Scrum teams manage themselves as a team, succeed as a team, fail as a team — and should be evaluated as a team.

Keeping track of success measurements can do more than help you improve throughout your work. Metrics can provide clear proof of success when you move past your first product and start to scale agile practices throughout your organization.

Step 2: Build awareness and excitement

After you have a roadmap showing you the "how" of your agile transition, you need to communicate the coming changes to people in your organization. Agile approaches have many benefits; be sure to let all individuals in your company know about those benefits and get them excited about the coming changes. Here are some ways to build awareness:

- **Educate people.** People in your organization may not know much — or anything — about agile product development. Educate people about agile principles and approaches and the change that will accompany the new approaches. You can create an agile wiki, hold lunchtime learning sessions, and even have hot-seat discussions (face-to-face discussions with leadership where people can talk safely about concerns and get their questions answered about changes and agile topics) to address concerns with the transition.

- **Use a variety of communication tools.** Take advantage of communication channels such as newsletters, blogs, intranets, email, and face-to-face workshops to get the word out about the change coming to your organization.
- **Highlight the benefits.** Make sure people in your company know how an agile approach will help the organization create high-value products, lead to customer satisfaction, and increase employee morale. Chapter 21 has a great list of the benefits of agile product development for this step.
- **Share the implementation plan.** Make your transition plan available to everyone. Talk about it, both formally and informally. Offer to walk people through it and answer questions. We often print the transition roadmap on posters and distribute it throughout the organization.
- **Involve the initial scrum team.** As early as you can, let the people who may work on your company's first agile pilot know about the upcoming changes. Involve the pilot scrum team members in planning the transition to help them become enthusiastic agile practitioners.
- **Be open.** Drive the conversation about new processes. Try to stay ahead of the company rumor mill by speaking openly, answering questions, and quelling myths about the agile transition. Structured communications like the hot-seat sessions we mention earlier are a great example of open communication.

Building awareness will generate support for the upcoming changes and alleviate some of the fear that naturally comes with change. Communication will be an important tool to help you successfully implement agile processes.

Step 3: Form a transformation team and identify a pilot

Identify a team in your company that can be responsible for the agile transformation at the organization level. This agile transition team, which is described in Chapter 18, is made up of executives and other leaders who will systematically improve processes, reporting requirements, and performance measurements across the organization. Selecting people for the team who are passionate about and committed to helping the organization become more adaptive and resilient is paramount.

The agile transition team will create organizational changes within sprints, just like the development team creates product features within sprints. The transition team will focus on the highest-priority changes supporting agility in each sprint and will demonstrate its implementation, when possible, during a sprint review with all stakeholders, including the pilot scrum team members.

Starting your agile transition with just one pilot is a great way to establish a reference model of what a scrum team can look like and to demonstrate the benefits of an agile approach. Having a reference model allows you to figure out how to work with agile methods with little disruption to your organization's overall business. Concentrating on one pilot to start also lets you work out some of the kinks that inevitably follow change. Figure 20-3 shows the types of development efforts that benefit most from the agile approach.

FIGURE 20-3: Product development efforts that can benefit from agile techniques.

When selecting your first agile pilot to establish a reference model for future scrum teams, look for an endeavor with these qualities:

- **Appropriately important:** Make sure the product you choose is important enough to merit interest within your company. However, avoid the most important product coming up; you want room to make and learn from mistakes. See the note on the blame game in the later section "Avoiding Transformation Pitfalls."
- **Sufficiently visible:** Your pilot should be visible to your organization's key influencers, but don't make it the most high-profile item on the agenda. You will need the freedom to adjust to new processes; critical product development efforts may not allow for that freedom on the first try of a new approach.
- **Clear and containable:** Look for a product with clear requirements and a business group that can commit to defining and prioritizing those requirements. Try to choose a product that has a distinct end point, rather than one that can expand indefinitely.

- **Not too large:** Select a pilot that you can complete with no more than two scrum teams working simultaneously to prevent too many moving parts at once. A single-team pilot is preferred.
- **Tangibly measurable:** Choose an endeavor that you know can show measurable value within sprints.

TECHNICAL STUFF

People need time to adjust to organizational changes of any type, not just agile transitions. Studies have found that with large changes, companies and teams will see dips in performance before they see improvements. *Satir's Curve,* shown in Figure 20-4, illustrates the process of teams' excitement, chaos, and finally adjustment to new processes.

FIGURE 20-4: Satir's Curve.

After you've successfully used agile techniques on one agile product development, you'll have a reference model and foundation for future successes.

Step 4: Build an environment for success

One of the agile principles states, "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

We outline what it means to create an environment to enable success in Chapter 6. Study the 4 agile values and the 12 agile principles carefully (see Chapter 2) and seriously determine whether you're creating an environment for success or rationalizing that the status quo is good enough.

Start fixing and improving your physical and cultural environment as early as possible.

Step 5: Train sufficiently and recruit as needed

Training is a critical step when shifting to an agile mindset. The combination of face-to-face training with experienced agile experts and the ability to work through exercises using agile processes is the best way to help the team to absorb and develop the knowledge needed to successfully begin.

Training works best when the members of the team can train and learn together, and then bring the shared experience back to work with them. A common language and understanding exist among them. As agile trainers and mentors, we've had the opportunity to overhear conversations between team members that start, "Remember when Mark showed us how to . . .? That worked when we did it in class. Let's try it and see what happens." If the product owner, development team, scrum master, and stakeholders can attend the same class, they can apply lessons to their work as a team.

Recruiting talent to fill gaps in the roles you need avoids the obvious problems you'll have at the start of the transition. Without a dedicated product owner and his or her clear direction to the team, how likely is your pilot to succeed? How will that affect the team's ability to self-organize? Who will facilitate the many interactions if you're missing a scrum master? What will the first sprint look like if you're missing a key skill on the development team required to minimally achieve the first sprint goal?

Work with your human resources department as early as possible to start the recruiting process. Work with your agile expert advisers to tap into their network of experienced agile practitioners.

Step 6: Kick off the pilot with active coaching

When you have a clear agile implementation strategy, an excited and trained team, a pilot product with a product backlog, and clear measures for success, congratulations! You're ready to run your first sprint.

Don't forget, though — agile approaches are new to the pilot team. Teams need coaching to become high performing. Engage with agile experts for agile coaching to start your pilot right.

TIP

Practice doesn't make perfect. Practice makes permanent. Start off right.

As the scrum team plans its first sprint, it should not bite off too many requirements. Keep in mind that you're just starting to learn about a new process and a new product. New scrum teams often take on a smaller amount of work than they think they can complete in their first sprints. A typical progression follows.

REMEMBER

After you establish overall goals through the product's vision statement, product roadmap, and initial release goal, your product backlog needs only enough user-story level requirements (see Chapter 10) for one sprint for the scrum team to start development.

- **In sprint 1,** scrum teams take on 25 percent of the work they think they can complete during sprint planning.
- **In sprint 2,** assuming sprint 1 was a success, scrum teams take on 50 percent of the work they think they can complete during sprint planning.
- **In sprint 3,** scrum teams take on 75 percent of the work they think they can complete during sprint planning.
- **In sprint 4 and beyond,** scrum teams take on 100 percent of the work they think they can complete during sprint planning.

By sprint 4, the scrum team will be more comfortable with new processes, will know more about the product, and will be able to estimate tasks with more accuracy. High-performance patterns — such as teams that finish early accelerate faster — can be learned earlier by using shorter sprints.

WARNING

You can't plan away uncertainty. Don't fall victim to analysis paralysis; set a direction and go!

Throughout the first sprint, be sure to consciously stick with agile practices. Think about the following during your first sprint:

- Have your daily scrum meeting, even if you feel like you didn't make any progress and especially if anyone is feeling stuck. Remember to state roadblocks, too!
- The development team may need to remember to manage itself and not look to the product owner, the scrum master, or anywhere besides the sprint backlog for task assignments.
- The scrum master may have to remember to protect the development team from outside work and distractions, especially while other members of the organization get used to having a dedicated scrum team around.

- The product owner may have to become accustomed to working directly with the development team, being available for questions, and reviewing and accepting completed requirements immediately.

In the first sprint, expect the road to be a little bumpy. That's okay; agile processes are about learning and adapting.

In Chapter 10, you can see how the scrum team can plan the sprint. Chapter 11 provides the day-to-day details on running the sprint.

Step 7: Execute the Roadmap to Value

When you've chosen your pilot, don't fall into the trap of using a plan from an old methodology or set of habits. Instead, use agile processes from the start.

We outline the Roadmap to Value throughout this book, introducing it in Chapter 9 and leading you through each of the seven stages in Chapters 9 through 12.

Step 8: Gather feedback and improve

You'll naturally make mistakes at first. No problem. At the end of your first sprint, you gather feedback and improve with two important events: the sprint review and the sprint retrospective.

In your first sprint review, it will be important for the product owner to set expectations about the format of the meeting, along with the sprint goal and completed product functionality. The sprint review is about product demonstration — fancy presentations and handouts are unnecessary overhead. Stakeholders may initially be taken aback by a bare-bones approach. However, those stakeholders will soon be impressed as they find a working product increment replacing the fluff of slides and lists. Transparency and visibility — show, rather than tell.

The first sprint retrospective may require setting some expectations as well. It will help to conduct the meeting with a preset format, such as the one in Chapter 12, both to spark conversation and avoid a free-for-all complaining session.

In your first sprint retrospective, pay extra attention to the following:

- Keep in mind how well you met the sprint goal, not how many user stories you completed. (Focus on outcomes achieved over outputs produced.)
- Go over how well you completed requirements to meet the definition of done: Designed, developed, tested, integrated, and documented.

- Discuss how you met your success metrics or desired outcomes.
- Talk about how well you stuck with agile principles. We start the journey with principles.
- Celebrate successes, even small gains, as well as examine problems and solutions.
- Remember that the scrum team should manage the meeting as a team, gain consensus on how to improve, and leave the meeting with a plan of action.

You can find more details about both sprint reviews and sprint retrospectives in Chapter 12.

Step 9: Mature and solidify improvements

Inspecting and adapting enables scrum teams to grow as a team and to mature with each sprint.

Agile practitioners sometimes compare the process of maturing with the martial arts learning technique of *Shu Ha Ri*, a Japanese term that can be translated to "maintain, detach, transcend" as introduced in Chapter 8. The term describes three stages in which people learn new skills:

- **In the *Shu* stage,** new scrum teams may work closely with an agile coach or mentor to follow processes correctly.
- **In the *Ha* stage,** scrum teams will find that the sprint retrospective is a valuable tool for talking about how their improvisations worked or did not work. In this stage, scrum team members may still learn from an agile mentor, but they may also learn from one another, from other agile professionals, and from starting to teach agile skills to others.
- **In the *Ri* stage,** scrum teams can customize processes, knowing what works in the spirit of the agile values and principles.

At first, maturing as a scrum team can take a concentrated effort and commitment to using agile processes and upholding agile values. Eventually, however, the scrum team will be humming along, improving from sprint to sprint, and inspiring others throughout the organization.

With time, as scrum teams and stakeholders mature, entire companies can mature into successful agile organizations.

Step 10: Progressively expand within the organization

Completing a successful pilot is an important step in moving an organization to agile product development. With metrics that prove the success of your pilot and the value of agile methodologies, you can garner commitment from your company to support new opportunities for applying agile techniques.

To progressively expand the agile footprint across an organization, start with the following:

- **Support new teams.** An scrum team that has reached maturity — the people who worked on the first agile pilot — should now have the expertise and enthusiasm to become agile ambassadors in the organization. These people can become part of a guild to help new teams to learn and grow. See Chapter 8 to learn more about guilds.
- **Redefine metrics.** Identify measurements for success, across the organization, with each new scrum team and with each new product.
- **Expand methodically.** It can be exciting to produce great results, but companywide improvements require significant process changes. Don't move faster than the organization can handle. Check out Chapter 19 for different ways of working across multiple teams.
- **Identify new challenges.** Your first agile pilot may have uncovered roadblocks that you didn't consider in your original implementation plan. Update your strategy and maturity roadmap as needed.
- **Continue learning.** As you roll out new processes, make sure that new team members have the proper training, mentorship, and resources to effectively use agile techniques.

The preceding steps work for successful agile product development transitions. Use these steps and return to them as you expand, and you can enable agile principles to thrive in and drive your organization's success.

Leading by Example

Although the ten steps for successful agile transition are extremely helpful in agile transformations, an aspect that is imperative to every successful transformation is executive ownership. Consultants and coaches advise, but it is the organization that needs to do the work. Leaders have a responsibility to "walk the walk and talk the talk" of the new culture they're trying to create. Agile leaders

understand that every eye is watching them always, and learning to be an agile and adaptive leader is crucial.

The role of a servant-leader in an agile organization

We refer to *servant leadership* frequently throughout this book. In agile transformations, leaders thoughts shift from asking "What's wrong with this team?" to "How can I help?" Cultural shifts are more successful when people know that their leaders are alongside them, experiencing change and giving up old ways.

Instead of practicing command and control management tendencies, agile leaders set a clear vision but allow people to figure out how to accomplish that vision on their own. After people start sprinting toward the new vision, the leader is nearby giving encouragement and support and bulldozing blockers. Here are some keys for effective servant leadership.

High-performing teams are transparent — the pilot teams will work to make *everything* transparent. They do this so they can make necessary adjustments as part of their empirical process control. Some information, such as velocity, may seem like a manager's opportunity for improving team performance. Avoid the trap! Velocity is not a goal to be pushed and the velocity number is meaningless as a productivity metric — team outcomes are what matters! (We discuss how to use velocity appropriately as a planning tool in Chapter 15.) As a leader, retrospective results shared by the team may provide insight for organizational impediments that you can remove. Serve the team by removing blockers impeding their ability to reach their full potential.

TIP

Teams thrive when they feel psychologically safe. Harvard's Dr. Amy Edmondson defines psychological safety as the belief that you won't be punished or humiliated for sharing your ideas, questions, and concerns or for admitting when you made a mistake. Studies show that psychological safety enables moderate risk-taking, speaking your mind, and creativity. Psychologically safe people are comfortable being themselves.

Keys for successful servant leadership

Following are keys for successful servant leadership:

- **Build individual and organizational capability.** The more capable the teams become, the more effective they become. Teams get better at delivering value when individual team members get better at contributing to the

team's success. Invest in training so that individuals are able to contribute to their team's success in more than one way. Encourage team members to select tasks with which they may be unfamiliar so that they can stretch their skills and capability as they perform unfamiliar work. Build not only individual capability but also team capability. Build T-, pi-, and M-shaped individuals and pi- and M-shaped teams. (For more on T, pi, and M-shapes, see in Chapter 7.)

- **Manage by exception.** Serve where you're needed. If a team is thriving, stretching, and accomplishing, stay out of their way (but continue to provide support). Focus your attention on the teams that need you most but, again, don't get in the way. Offer support by asking how you can help. Reviewing sprint burndown charts will provide a clear insight into progress and is an excellent opportunity to watch for trends. Burndown charts can highlight where you might be able to offer help. (See Chapter 11 to learn more about burndown charts.)
- **Adjust the manager-to-creator ratio.** Agile organizations are self-organizing and self-managing. Rather than manager support, scrum teams need servant leadership support. The manager-to-creator ratio can be helpful for evaluating if you have the right ratio of managers to people developing your products. Agile organizations have low manager-to-creator ratios because the creators are empowered to do what they do well, and the managers redirect their time from micromanaging to providing the support and trust creators need. The organization will have more empowered team members supported by servant-leaders.

Avoiding Transformation Pitfalls

Organizations can make a number of common but serious mistakes when implementing agile practices. Table 20-1 provides an overview of some typical problems and ways to turn them around.

As you may notice, many of these pitfalls are related to a lack of organizational support, the need for training, and falling back on old project management practices. If your company supports positive changes, if the team is trained, and if the scrum team makes an active commitment to upholding agile values, you'll have a successful agile transition.

TABLE 20-1 **Common Agile Transition Problems and Solutions**

Problem	Description	Potential Solution
Faux agile or double work agile or both	Sometimes organizations will say that they are "doing agile." They may go through some of the practices used on agile development efforts, but they haven't embraced agile principles and continue creating waterfall deliverables and products. This is sometimes called *faux agile* and is a sure path to avoiding the benefits of agile techniques. See Chapter 23 to learn more about faux or fake agile. Trying to complete agile processes in addition to waterfall processes, documents, and meetings is double the work. *Double work agile* results in quick team burnout. If you're doing twice the work, you aren't adhering to agile principles.	Insist on following one process — an agile process. Garner support from management to avoid non-agile principles and practices.
Lack of training	Investment in a hands-on training class will provide a quicker, better learning environment than even the best book, video, blog, or white paper. Lack of training often indicates an overall lack of organizational commitment to agile practices. Keep in mind that training can help scrum teams avoid many of the mistakes on this list.	Build training into your implementation strategy. Giving teams the right foundation of skills is critical to success and necessary at the start of your agile transition.
Ineffective product owner	The product owner role is non-traditional. Scrum teams need a product owner who is an expert on business needs and priorities and can work well with the rest of the scrum team on a daily basis. An absent or indecisive product owner will quickly sink an agile product development effort.	Start the pilot with a person who has the time, expertise, and temperament to be a good product owner. Ensure the product owner has proper training. The scrum master can help coach the product owner on how to leverage agile principles and approaches, such as scrum, to build great products, and may try to clear roadblocks preventing the product owner from being effective. If removing impediments doesn't work, the scrum team should insist on replacing the ineffective product owner with a product owner who can make product decisions and help the scrum team be successful.
Lack of automated testing	Without automated testing, it may be impossible to fully complete and test work within a sprint. Manual testing requires time that fast-moving scrum teams don't have.	You can find many low-cost, open-source testing tools on the market today. Look into the right tools and make a commitment as a development team to using those tools.

(continued)

TABLE 20-1 ***(continued)***

Problem	Description	Potential Solution
Lack of transition support	Making the transition successfully is difficult and far from guaranteed. It pays to do it right the first time with people who know what they are doing.	When you decide to move to agile product development, enlist the help of an agile mentor — ideally externally with a diverse set of experience and impartial perspective — who can support your transition. Process is easy, but people are hard. It pays to invest in professional transition support with an experienced partner who understands behavioral science and organizational change.
Inappropriate physical environment	When scrum teams are not collocated, or otherwise enabled to collaborate in real-time and high fidelity, they lose the advantage of face-to-face communication. Scrum teams work best when they can sit together in the same area or, if distributed, in the same time zone.	If your scrum team is in the same building but not sitting in the same area, move the team together. Consider creating a room or annex for the scrum team to continually collaborate. Try to keep the scrum team area away from distracters, such as the guy who can talk forever or the manager who needs just one small favor. Before starting a pilot with a dislocated scrum team, do what you can to enlist local talent. If you must work with a dislocated scrum team, take a look at Chapter 16 to see how to manage dislocated teams.
Poor team selection	Scrum team members who don't support agile processes, don't work well with others, or don't have capacity for self-management will sabotage a new agile pilot from within.	When creating a scrum team, consider how well potential team members will enact the agile principles. The keys are versatility and a willingness to learn.
Discipline slips	Remember that agile products still need requirements, design, development, testing, and releases. Doing that work in sprints requires discipline to keep from falling into old habits of, for instance, delaying testing until the end of the sprint.	You need more, not less, discipline to deliver working functionality in a short iteration. Progress needs to be consistent and constant. The daily scrum helps ensure that progress is occurring throughout the sprint. Use the sprint retrospective as an opportunity to reset approaches to discipline.
Lack of support for learning	Scrum teams succeed as teams and fail as teams; calling out one person's mistakes (known as the *blame game*) destroys the learning environment and destroys innovation.	The scrum team can make a commitment at the start to leaving room for learning and to accepting success and failures as a group.
Diluting until dead	Watering down agile processes with old waterfall habits erodes the benefits of agile processes until those benefits no longer exist.	When making process changes, stop and consider whether those changes support agile values and principles. Resist changes that aren't compatible. Remember that maximizing work not done is essential.

Avoiding agile leadership pitfalls

Although there are pitfalls to avoid from an organizational transformation perspective, here are a few tips to enable organizational leader success:

- **Hold the right people accountable.** Many leaders hold their direct reports accountable for product delivery. In an agile organization, accountability for what is built and when it's delivered to customers lies with the product owner, and accountability for building quality into the product lies with the development team. Organizational leaders are servant-leaders helping product owners make better decisions and creating opportunities for building capable development teams who can build products to meet customer needs. Having an organizational leader become accountable for something a product owner or development team owns leads to frustration and de-motivation.
- **Be open to what could be.** Many leaders, especially new leaders to the organization, believe the flavor of their agile at their old organization was better than the agile practices used in their new organization. Begin where your new organization is and learn why they develop products the way they do. Perhaps you can sign up to attend the same training received by your teams to better align with the organization's training. Beware of inadvertently undermining the organization's agile trajectory.
- **Get trained.** The most effective transformations begin with leaders who lead by example. Getting trained is the first and most important step leaders can take to start exemplifying the agile culture they want to create. Otherwise, pilot teams will wonder why they are being asked to practice agile techniques when their untrained leaders don't. Avoid this trap by investing time and energy into getting trained.
- **Inspect and adapt.** Use the sprint review as an opportunity to not only show your support for the team but also to inspect and adapt. Many coaching opportunities become exposed during a sprint review. Your absence from a sprint review sends a message that you believe whatever the team is working on is not important. Take advantage of the "olympic stage" performance improvement opportunity discussed in Chapter 12.
- **Practice agility yourself.** Gather your leadership team around you and use scrum. Build a backlog and make it transparent to your organization. Plan your sprint, hold a daily standup, demonstrate your work to your organization in a sprint review, and then hold a team retrospective. If you think scrum is a good idea for your organization, isn't it a good idea for you too? Teams who see their leaders using scrum will be more motivated to improve their scrum practice.

- **Trust others.** Scrum teams fail early and often. Failure is essential for learning. Avoid helicopter-managing and allow teams to solve their own problems as much as possible. As a leader, you know there is no substitute for hard-fought, lesson-learned experience. Let natural consequences lead to unforgettable learning opportunities. Many traditional management tendencies may need to be unlearned to help teams benefit from autonomy, mastery, and purpose, as discussed in Chapter 8.

Signs Your Changes Are Slipping

The following list of questions helps you see warning signs and provide ideas on what to do if problematic circumstances arise:

- Are you doing "scrum, but . . ."?

  "ScrumBut" occurs when organizations partially adopt scrum. Beware of old practices that thwart agile principles, such as finishing sprints with incomplete functionality.

  REMEMBER

  Scrum is three roles, three artifacts, and five events. If you find your team tweaking those basic framework components, ask why. Is scrum exposing something you're not willing to inspect and adapt?

- Are you still documenting and reporting in the old way?

  If you're still burning hours on hefty documentation and reporting, it's a sign that the organization has not accepted agile approaches for conveying status. Help managers understand how to use existing agile reporting artifacts and quit doing double work!

- A team completing 50 story points in a sprint is better than another team doing 10, right?

  No. Keep in mind that story points are relative and consistent within one scrum team, not across multiple scrum teams. Velocity isn't a team comparison metric. It is simply a post-sprint fact that scrum teams use to help them in their own planning. The best way to learn how a team is performing is to attend their sprint review. You can see more about story points and velocity in Chapter 10 and sprint reviews in Chapter 12.

- When will the stakeholders sign off on all the specifications?

  If you're waiting for sign-offs on comprehensive requirements to start developing, you're not following agile practices. You can start development as soon as you have enough requirements for one sprint. Scrum teams we work with start developing as early as day two.

- Are we using offshore to reduce costs?

  Ideally, scrum teams are collocated. The ability for instant face-to-face communication saves more time and money and prevents more costly mistakes than the initial hourly savings you may see with some offshore teams.

  TIP

  Offshoring is common. We're all for it, but when you offshore, offshore with both feet. Create scrum teams in single geographic locations so they can be collocated or at least collaborate all day in similar time zones. Having a fully collocated scrum team on one continent and another fully collocated scrum team on another continent is recommended.

  If you do work with offshore teams, invest in good collaboration tools such as individual video cameras and persistent, virtual team rooms. Short sprints enable frequent inspection and adaptation opportunities, which is particularly helpful for offshore and other vendor engagements.

- Are development team members asking for more time in a sprint to finish tasks?

  The development team may not be working cross-functionally or swarming on priority requirements. Development team members can help one another finish tasks, even if those tasks are outside of a person's core expertise.

  This question can also indicate outside pressures to underestimate tasks and fit more work into a sprint than the development team can handle.

- Are development team members asking what they should do next?

  After a sprint is planned and development work is under way, if the developers are waiting for direction from the scrum master or product owner, they aren't self-organizing. The development team should be telling the scrum master and the product owner what it's doing next, not the other way round.

- Are team members waiting until the end of the sprint to do testing?

  Agile development teams should test every day in a sprint. All development team members are testers.

- Are the stakeholders showing up for sprint reviews?

  If the only people at sprint reviews are the scrum team members, it's time to remind stakeholders of the value of frequent feedback loops. Let stakeholders know that they're missing their chance to review working product functionality regularly, correct course early, and see firsthand how the product development is progressing.

- Is the scrum team complaining about being bossed around by the scrum master?

  Command-and-control techniques are the antithesis of self-management and are in direct conflict with agile principles. Scrum teams are teams of peers — there is no boss on the team. Have a discussion with the agile mentor and act quickly to reset the scrum master's expectations of his or her role.

- Is the scrum team putting in a lot of overtime?

  If the end of each sprint becomes a rush to complete tasks, you aren't practicing sustainable development. Look for root causes, such as pressure to underestimate. The scrum master may need to coach the development team and shield its members from product owner pressure if this is the case. Reduce the story points for each sprint until the development team can get a handle on the work.

- What retrospective?

  If scrum team members start avoiding or cancelling sprint retrospectives, you're on the slide back to waterfall. Remember the importance of inspecting and adapting and be sure to look at why people are missing the retrospective in the first place. If you're not progressing, complacency usually results in sliding backwards. Even if the scrum team has great velocity, development speed can always be better, so keep the retrospective, and keep improving.

6 The Part of Tens

IN THIS PART . . .

Communicate the benefits of agile product development.

Address key factors for agile success.

Common signs and indications that you're *not* agile.

Become an agile professional through learning, networking, and community collaboration, with support from valuable resources.

IN THIS CHAPTER

- Ensuring that products are rewarding
- Making reporting easy
- Improving results
- Reducing risk

Chapter 21

Ten Key Benefits of Agile Product Development

In this chapter, we provide ten important benefits that agile approaches provide to organizations, scrum teams, and products.

REMEMBER

To take advantage of agile product development benefits, you need to trust in agile principles, learn more about different agile practices and approaches, and discover what's best for your team.

Higher Customer Satisfaction

Scrum teams are committed to producing products that satisfy customers. Agile approaches for happier customers include the following:

- Collaborate with customers to collect feedback throughout the process so customers get what they really want.
- Ensure that the product owner is an expert on product requirements and customer needs, or knows where to get that information. (Check out Chapters 7 and 11 for more information about the product owner role.)

- Keep the product backlog updated and prioritized to respond quickly to change. (You can find out about the product backlog in Chapter 10 and its role in responding to change in Chapter 14.)
- Demonstrate working functionality to stakeholders in every sprint review. (Chapter 12 shows you how to conduct a sprint review.)
- Deliver products to market quicker and more often with every release.

Better Product Quality

Customers demand quality products. Agile methods have excellent safeguards to make sure that quality is as high as possible. Scrum teams help ensure quality by doing the following:

- Take a proactive approach to quality to prevent product problems.
- Embrace technological excellence, good design, and sustainable development.
- Define and elaborate requirements just in time so that knowledge of product features is as relevant as possible.
- Build acceptance criteria into user stories so that the development team better understands them and the product owner can accurately validate them.
- Incorporate continuous integration and thorough testing into the development process, allowing the development team to address issues while they're fresh.
- Take advantage of automated testing tools, ensuring that new product increments do not break previous increments.
- Conduct sprint retrospectives, allowing the scrum team to continuously improve processes and work.
- Complete work using the definition of done: Developed, tested, integrated, and documented.

You can find more information about agile quality in Chapter 17.

Reduced Risk

Agile product development techniques virtually eliminate the chance of absolute failure — spending large amounts of time and money with no return on investment. Scrum teams reduce risk by doing the following:

- Develop in sprints, ensuring a short time between initial investment and either failing fast or knowing that a product or an approach will work. Front-load risk by ensuring that the product backlog is stacked with the most valuable and most risky items first. See Chapter 13 to find out about assessing product risks and opportunities.
- Always have a working, integrated product, starting with the first sprint, so that some value is added as shippable functionality every sprint, ensuring the product won't fail completely.
- Develop requirements according to the definition of done in each sprint so that sponsors have completed, usable functionality, regardless of what may happen with the product in the future.
- Provide constant feedback on products and processes through the following:
  - Daily scrum meetings and constant development team communication
  - Regular daily clarification about requirements and review and acceptance of features by the product owner
  - Sprint reviews, with stakeholder and customer input about completed product functionality
  - Sprint retrospectives, where the development team discusses process improvement
  - Releases, where the end user can see and react to new features on a regular basis
- Generate revenue early with self-funding products, allowing organizations to pay for a product with little up-front expense.

You can find more information about managing risk in Chapter 17.

Increased Collaboration and Ownership

When development teams take responsibility for products, they can produce great results. Agile development teams collaborate and take ownership of product quality and performance by doing the following:

- Make sure that the development team, the product owner, and the scrum master work closely together on a daily basis.
- Conduct goal-driven sprint planning meetings, allowing the development team to commit to the sprint goal and organize its work to achieve it.
- Hold daily scrum meetings led by the development team, where development team members organize around work completed, future work, roadblocks, and team morale.
- Conduct sprint reviews, where the development team can demonstrate and discuss the product directly with stakeholders.
- Conduct sprint retrospectives, allowing development team members to review past work and recommend better practices with every sprint.
- Work in a collocated environment, allowing for instant communication and collaboration among development team members. If you're on a distributed team, stay connected to your team's videoconference.
- Make decisions by consensus, using techniques such as estimation poker and the fist of five.

You can find out how development teams estimate effort for requirements, decompose requirements, and gain team consensus in Chapter 9. To discover more about sprint planning and daily scrum meetings, see Chapter 11. For more information about sprint reviews and retrospectives, check out Chapter 12.

More Relevant Metrics

The metrics that scrum teams use to estimate time and cost, measure performance, and make decisions are often more relevant and more accurate than metrics on traditional projects. Agile metrics should encourage sustainable team progress and efficiency in a way that works best for the team to deliver value to the customer early and often. With agile product development, you provide metrics by doing the following:

- Determine timelines and budgets based on each development team's actual performance and capabilities.

- Make sure that the development team that will be doing the work, and no one else, provides effort estimates for requirements.
- Use relative estimates, rather than absolute hours or days, to accurately tailor estimated effort to an individual development team's knowledge and capabilities.
- Refine estimated effort, time, and cost on a regular basis, as the development team learns more about the product.
- Update the sprint burndown chart every day to provide accurate metrics about how the development team is performing within each sprint.
- Compare the cost of future development with the value of that future development, which helps teams determine when to end development and redeploy capital to a new investment opportunity.

WARNING

You might notice that velocity is missing from this list. *Velocity* (a measure of development speed, as detailed in Chapter 15) is a tool you can use to determine timelines and costs, but it works only when tailored to an individual team. The velocity of Team A has no bearing on the velocity of Team B. Also, velocity is great for measurement and trending, but it doesn't work as a control mechanism. Trying to make a development team meet a certain velocity number only disrupts team performance and thwarts self-management.

If you're interested in finding out more about relative estimating, be sure to check out Chapter 9. You can find out about tools for determining timelines and budgets, along with information about capital redeployment, in Chapter 15.

Improved Performance Visibility

With agile product development, every member of the team has the opportunity to know how the product development is going at any given time. Teams can provide a high level of performance visibility by doing the following:

- Place a high value on open, honest communication among the scrum team, stakeholders, customers, and anyone else in an organization who wants to know about a product.
- Provide daily measurements of sprint performance with sprint backlog updates. Sprint backlogs can be available for anyone in an organization to review.

- Provide daily insight into the development team's immediate progress and roadblocks through the daily scrum meeting. Although only the scrum team may speak at the daily scrum meeting, any member of the product team may observe or listen.
- Physically display progress by using task boards and posting sprint burndown charts in the development team's work area every day.
- Demonstrate accomplishments in sprint reviews. Anyone within an organization may attend a sprint review.

Improved product development visibility can lead to greater investment control and predictability, as described in the following sections.

Increased Investment Control

Scrum teams have numerous opportunities to control investment performance and make corrections as needed because of the following:

- Adjusting priorities throughout development allows the organization to have fixed-time and fixed-price products while accommodating change.
- Embracing change allows the team to react to outside factors such as market demand.
- Daily scrum meetings allow the scrum team to quickly address issues as they arise.
- Daily updates to sprint backlogs mean sprint burndown charts accurately reflect sprint performance, giving the scrum team the opportunity to make changes the moment it sees problems.
- Face-to-face conversations remove roadblocks to communication and issue resolution.
- Sprint reviews let stakeholders see working products and provide input about the products before release.
- Sprint retrospectives enable the scrum team to make informed course adjustments at the end of every sprint to enhance product quality, increase development team performance, and refine processes.
- Self-organizing, self-managing teams possess the potential for self-funding products. (Chapter 15 tells you about self-funding products.)

The many opportunities to inspect and adapt throughout agile product development allow all members of the product team — the product owner, development team, scrum master, and stakeholders — to exercise control and ultimately create better products.

Improved Predictability

Agile product development techniques help the team accurately predict how things will go as product development progresses. Here are some practices, artifacts, and tools for improved predictability:

- Keeping sprint lengths and development team allocation the same throughout development allows the team to know the exact cost for each sprint.
- Using individual development team speed allows the team to predict timelines and budgets for releases, the remaining product backlog, or any group of requirements.
- Using the information from daily scrum meetings, sprint burndown charts, and task boards allows the team to predict performance for individual sprints.

You can find more information about sprint lengths in Chapter 10.

Optimized Team Structures

Self-management puts decisions that would normally be made by a manager or the organization into scrum team members' hands. Because of the limited size of development teams — which consist of three to nine people — agile product development efforts can have multiple scrum teams, if necessary. Self-management and size-limiting mean that agile product development can provide unique opportunities to customize team structures and work environments. Here are a few examples:

- Development teams may organize their team structure around people with specific work styles and personalities. Organization around work styles provides these benefits:
  - Allows team members to work the way they want to work
  - Encourages team members to expand their skills to fit into teams they like
  - Helps increase team performance because people who do good work like to work together and naturally gravitate toward one another

- Scrum teams can make decisions tailored to provide balance between team members' professional and personal lives.
- Because development teams estimate the work they will do, product owners can determine how many scrum teams may be required to accomplish the items on a product backlog.
- Ultimately, scrum teams can make their own rules about whom they work with and how they work.

REMEMBER

The idea of team customization allows agile workplaces to have more diversity. Organizations with traditional management styles tend to have monolithic teams where everyone follows the same rules. Agile work environments are much like the old salad bowl analogy. Just like salads can have ingredients with wildly different tastes that fit in to make a delicious dish, agile product development can have people on teams with very diverse strengths that fit in to make great products.

Higher Team Morale

Working with happy people who enjoy their jobs can be satisfying and rewarding. Agile product development improves the morale of scrum teams in these ways:

- Being part of a self-managing team allows people to be creative, innovative, and acknowledged for their contributions.
- Focusing on sustainable work practices ensures that people don't burn out from stress or overwork.
- Encouraging a servant-leader approach assists scrum teams in self-management and actively avoids command-and-control methods.
- Having a dedicated scrum master, who serves the scrum team, removes impediments, and shields the development team from external interferences.
- Providing an environment of support and trust increases people's overall motivation and morale. People benefit from improved autonomy, mastery, purpose, and belongingness.
- Having face-to-face conversations helps reduce the frustration of miscommunication.
- Working cross-functionally allows development team members to learn new skills and to grow by teaching others.

You can find out more about team dynamics in Chapter 16.

IN THIS CHAPTER

- Ensuring scrum teams have the environment and tools they need
- Filling all roles with the right talent
- Enabling teams with clear direction and support

Chapter 22

Ten Key Factors for Agile Product Development Success

Here are ten key factors that determine whether an agile transition will succeed. You don't need all issues resolved before you begin. You just need to be aware of them and have a plan to address them as early in your journey as possible.

TIP

We have found that the first three are the strongest indicators for success. Get those right and the likelihood of your success increases dramatically.

Dedicated Team Members

In Chapter 8, we describe how products are considered long-term assets requiring stable, dedicated, and even permanent teams. Permanent teams retain knowledge about their product and customers. Their high performance is built over years of retrospectives and hard work. Minor adjustments to the team may need to be made due to career opportunities and the like, but for the most part organizations should work to disrupt team makeup as little as possible.

Further, it's critical for these dedicated team members — product owner, development team members, scrum master — to focus on a single objective at a time. If team members are jumping between contexts hourly, daily, weekly, or even monthly, their effectiveness is minimized due to the increased cost of just trying to keep up with multiple task lists. The time lost due to the continual cognitive demobilization and remobilization involved with task switching is costly.

TIP

If you think you don't have enough people to dedicate to your scrum teams, you definitely don't have enough people to thrash them across multiple priorities simultaneously. The American Psychological Association reports that task switching wastes as much as 40 percent of time.

Variance in equals variance out.

Collocation

The Agile Manifesto lists individuals and interactions as the first value. The way you get this value right is by collocating team members to be able to have clear, effective, and direct communication throughout development.

Challenges associated with dislocated teams include the tendency to rely on written communication due to the inconvenience of communicating face-to-face across geographic or even time zone separations. Written communications, especially those intended to resolve issues, are prone to costly delays and misunderstandings. The lack of real-time, face-to-face interactions results in lower levels of trust and familiarity with how other people think, act, and work.

In Chapter 6, we talk about collocation as the first crucial element of an agile environment. Bell Laboratories showed a fifty-fold improvement in productivity. One of the important factors in their success was collocating, which led to increased customer collaboration, working functionality, and response time. Technologies such as videoconferencing and other digital collaboration tools make dislocated collaboration more effective. However, although technology helps compensate for the challenges of dislocation, it's not as valuable or sustainable as physically working next to your teammates. See Chapter 6 to learn more about creating an environment for team success.

Done Means Shippable

Ending sprints with non-shippable functionality is an anti-pattern to becoming more agile. *Done* means shippable. A sprint that ends without potentially shippable functionality is, by definition, not a sprint.

Development teams get to done by swarming on user stories — working together on one user story at a time until it is complete before starting the next. Developers hold each other accountable by ensuring that all rules for their definition of done, including test automation, are satisfied before starting a new user story. Product owners review completed work against the scrum team's definition of done (as well as the user story's acceptance criteria — see Chapter 10) before approving and moving on to a new user story.

Address What Scrum Exposes

Scrum won't solve any problems for you, but it will expose them. Scrum will expose weaknesses and gaps in process, policies, organizational structures, skillsets, roles, artifacts, meeting effectiveness, transparency, and myriad other topics. What you decide to do with what is exposed is up to you. Scrum provides an iterative inspect and adapt framework for addressing items as they're exposed. Scrum teams should address what is exposed on their own when they have control over the exposed issue. When they don't have control over the exposed issue, there should be a path for escalating those items to those in the organization who own the status quo and who can affect change to support scrum teams getting better and better at delivering customer value. An effective way to resolve escalated items exposed by scrum is to use the agile transition team, as explained in Chapter 20.

Clear Product Vision and Roadmap

Although the product owner owns the product vision and product roadmap, many people affect the clarity of these agile artifacts. Product owners need access to and strong working relationships with stakeholders and customers throughout product development to ensure that the vision and roadmap continually reflect what the customer and market need. The development team must also be crystal clear on the purpose of everything it works on, from the product vision to the individual user story. Purpose-driven development delivers business and customer value and mitigates risk effectively.

Without a clear purpose, people wander and lack ownership. When all team members understand the purpose, they come together. Remember the agile principle, "The best architectures, requirements, and designs emerge from self-organizing teams."

We discuss the mechanics of developing the vision and product roadmap in Chapter 9.

Product Owner Empowerment

The product owner's role is to optimize the value produced by the development team. This product owner responsibility requires that someone be knowledgeable about the product and customer, available to the development team throughout each day, and empowered to make priority decisions and give clarification immediately so that development teams don't wait or make inappropriate decisions about the product's direction.

Organizations with the following product owner characteristics will struggle significantly to deliver valuable and shippable functionality at the end of every sprint:

- The product owner struggles to make tough business decisions.
- The product owner is not accessible to the development team because he or she has too many other things to do besides support the development team and work directly with customers and stakeholders.
- More than one product owner is named by the organization for a single product, confusing the development team as to whom to go to for clarification.
- Stakeholders undermine decisions made by the product owner.

Although all roles on the scrum team are vital and equally important, an unempowered and ineffective product owner usually causes scrum teams to ultimately fail at delivering the value customers need from the team. See Chapter 7 for more on the product owner role.

Developer Versatility

You probably won't start your first agile product development effort with a development team that has the ideal level of skills required for every item in your product backlog. However, the goal should be to achieve skill coverage as soon as

possible. Your team will also be challenged to meet its sprint goal if you have single points of failure in any one skill, including testing.

From day one, you need people on your team with the intellectual curiosity and interest to learn new things, to experiment, to mentor and to receive mentoring, and to work together to get things to done as quickly as possible. This versatility is discussed more in Chapter 7.

Scrum Master Clout

As you depart from command and control leadership to empower the people doing the work to make decisions, servant leadership provides the solution. With formal authority, a scrum master would be viewed as a manager — someone to report to. Scrum masters should not be given formal authority but should be empowered by leadership to work with members of the scrum team, stakeholders, and other third parties to clear the way so that the development team can function unhindered.

If scrum masters have organizational *clout,* which is informal and a socially earned ability to influence, they can best serve their teams to optimize their working environment. In Chapter 7, we talk more about different types of clout. Provide training and mentorship to ensure that your scrum masters develop the soft skills of servant leadership and put off the tendencies of commanding and directing.

Leadership Support for Learning

When organizational leaders decide to become agile, their mindset has to change. Too often we see leadership directives without any follow-through for supporting the learning process needed to implement the changes. Realistically, organizational transformations take from one to three years from the time leadership buys in. Buying in means much more than writing the check for the training or coaching services. It means leaders get involved and learn what they need to do to lead the transformation from within, buying in with their time and effort and actions.

It is unrealistic to expect all the benefits of following agile principles after the first sprint. In Chapter 20, we talk about choosing an appropriate agile pilot, one with leeway to stumble a bit at first as everyone learns a new way of working together.

The bottom line: If support for learning is merely lip service, scrum teams will pick up on it early, will lose motivation to try new things, and will go back to waiting for top-down directives on how to do their job.

Transition Support

Chapter 20 compares an agile transition to a sports team learning to play a different sport. Good coaching at leadership and team levels increases your chances to succeed. Coaching provides support in the following forms:

- In-the-moment course correction when discipline starts to slip or mistakes are made
- Reinforcing training
- One-on-one mentoring for specific role-based challenges
- Executive leadership style and mindset adjustments

See our Platinum Edge agile transition roadmap in Chapter 20 for specific steps to take alongside your trusted agile expert coaches.

IN THIS CHAPTER

- Understanding the organizational benefits from adopting agile techniques
- Recognizing potential signs preventing agility
- Watching out for faux agile

Chapter 23

Ten Signs That You're Not Agile

The journey to become agile is popular, but business agility in itself is not the goal. It's a means to an end. Those who are making the journey, according to the Scrum Alliance, report the following benefits:

- Faster time to market
- More alignment with the business
- Better visibility of product development endeavors
- Increased ability to manage changing requirements

The journey to become agile is never ending, but the agile values and principles will guide you. Along the way, you might encounter some signs that your organization is not becoming more agile, as described in this chapter.

A Non-Shippable Sprint Product Increment

The first sign is easy to spot. If your product team is not able to demonstrate a potentially shippable product increment at the end of the sprint, you're not doing it right. Agile product development builds working product increments into each

and every sprint. The product owner may choose not to release the product increment, but it is potentially shippable nonetheless. Agile product teams adhere to their definition of done to instill confidence that they've built the right product increment in the right way. Not having a potentially shippable product increment at the end of a sprint is a sure sign that the team is struggling and may need help.

Signs to watch for are demonstrations of the product in non-production-like environments, a reluctance to let people use the increment, and focusing on attention-distracting PowerPoint presentations rather than working products. Teams in this situation often suggest that they need more time by extending their sprint length, which is an anti-pattern.

Agile values and principles for navigating this situation follow:

- Value #2: Working software over comprehensive documentation.
- Principle #7: Working software is the primary measure of progress.

Scrum teams value working products and demonstrate valuable increments during each sprint review.

Long Release Cycles

The longer your release cycles, the less likely you're becoming more agile. Long release cycles are typically the result of traditional waterfall practices, in which all the risk and value is saved until the end and then released. Or it can be a symptom of feeling the need to deliver everything before delivering anything. *Short* means different durations to different organizations and industries, but the principle of shorter being better than longer is the constant.

TIP

Long release cycles can be reduced by focusing on the next minimum viable product (MVP). Speed-to-market advantage is indefensible because first-to-market winners

- Get the opportunity to be first to capture market share
- Validate the viability of their product by being first to capture end customer feedback lowering the cost of pivot
- Receive an early return on investment (ROI) realizing a dollar today is more valuable than a dollar in the future (net present value 101)
- Avoid internal or external obsolescence

Agile product development releases early and often. Short feedback cycles provide learning for the team, helping them to incrementally align the product to their customer's needs and reduce risk.

If a team is unable to have short release cycles, here are some warning signs: product owners hesitant to release anything until they have everything, the development team lacking confidence in the quality of its work, an engagement with a vendor requiring a traditional waterfall approach, and most frequently, the phrase, "We can't release until the other pieces of the product are complete."

Scrum teams tackle the most risky and valuable requirements first. Early failure can be a form of success because of the speed of learning. You want to validate assumptions and vet good ideas early. Most organization we work with use one-week sprints. Customers demand more sooner.

Agile principles for navigating this situation follow:

- Principle #1: Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Principle #3: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

The key words are *early, continuous, valuable product, frequently,* and *shorter timescale.*

Disengaged Stakeholders

A key sign you're not agile is when stakeholders or sponsors are disengaged. Stakeholder feedback is essential for ensuring that the product will meet the needs of the customers. Lack of stakeholder feedback puts the team in a precarious situation, essentially sailing a ship without a rudder.

Signs your stakeholders are disengaged are a lack of stakeholder participation during product vision and road-mapping sessions, release planning, and most obviously during sprint reviews. The inspection opportunity provided during sprint reviews is the stakeholders' best opportunity to provide feedback.

Stakeholder engagement is a key responsibility of the product owner. We also encourage scrum masters to "never lunch alone," continuously building relationships and influence throughout the organization to enable effective and timely removal of impediments.

Agile values and principles for navigating this situation follow:

- Value #3: Customer collaboration over contract negotiation.
- Principle #8: Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Principle #5: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Scrum teams who maintain a sustainable pace as they collaborate frequently with sponsors and stakeholders are more effective. Sponsors and stakeholders who give the teams the needed environment and support, and then trust the team to get the job done will see teams become even more motivated to build better products faster.

Lack of Customer Contact

As stated, the relationship between the scrum team and the stakeholders is essential, but another sign you're not agile is when an invisible wall is placed between the development team and the stakeholders or customers. Some organizations make the mistake of distancing the technical team from the business partners. Critical feedback needed by the team is missed when this occurs.

REMEMBER

When a development team gets its information indirectly from secondary sources, the process becomes much like the children's telephone game. The message changes each time one child whispers what he or she heard to the next child, so that when the child who started the chain hears what the last child heard, the message has become totally different. Similarly, customer feedback can get twisted and filtered, losing its meaning and intent.

The closer development teams can be to the customers and the more frequent their interactions, the better they'll understand the customer's problems and challenges. The development team's work comes to life and the motivation for solving problems for real people in real situations makes their work more meaningful.

Agile values and principles for navigating this situation follow:

- Value #1: Individuals and interactions over processes and tools.
- Principle #2: Welcome changing requirements, even late in development. Agile processes harness change for the customers competitive advantage.
- Principle #4: Business people and developers must work together daily throughout the project.

Accurately capturing customer needs occurs when scrum teams focus on individuals and interactions, welcome change, and then harness that change for their customer's competitive advantage and work with business people daily.

Lack of Skill Versatility

Teams with limited skill versatility will become victims of dependencies and constraints. Mature scrum teams (those who have worked together for a long time and have self-organized effectively over time to increase skill diversity across the team) are typically more cross-functional than less mature scrum teams. By eliminating single points of failure in a scrum team, you increase its ability to move faster and produce higher-quality products. (See Chapter 7 to learn more about eliminating single points of failure.)

Over time, as each person increases his or her quantity and level of skills, the constraints and delays due to skill gaps disappear. Scrum teams are about skills, not titles. You want team members who can contribute to the sprint goal each and every day without the risk of single points of failure. Working on a task that requires a skill you haven't yet mastered is the first step to organically increasing the team's capability. Outside coaches with the necessary skills who join the team for short periods until the team learns the new skills can also help. Build M-shaped individuals and M-shaped teams, as discussed in Chapters 7 and 16.

Agile principles for navigating this situation follow:

- Principle #9: Continuous attention to technical excellence and good design enhances agility.
- Principle #11: The best architectures, requirements, and designs emerge from self-organizing teams.

Self-organizing, cross-functional teams who strive for technical excellence build both individual and team capability.

Automatable Processes Remain Manual

Avoidable product defects are another sign of impeded agility. Defects are generally the result of poor testing. Teams who manually test take longer and are less thorough. Manual testing doesn't keep up with the pace of change required in today's environment, and falls short of instilling confidence that everything still works whenever a change is made.

Lack of automation is often another barrier to releasing product increments timely. With software development, continuous integration and continuous deployment (CI/CD) pipelines enable teams to not only automate testing but also automate deployment scripts.

Including test automation and deployment automation in the definition of done is critical for becoming agile. Without automation, scrum teams will not be able to deliver early and often and respond quickly to changes in technology and the marketplace.

Agile values and principles for navigating this situation follow:

- Value #4: Responding to change over following a plan.
- Principle #3: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Principle #9: Continuous attention to technical excellence and good design enhances agility.

Prioritizing Tools over the Work

Although tools (including digital tools) can be helpful to scrum teams, they can also cause unnecessary overhead burdens and anti-patterns. A sign that you're not agile is when maintenance of the team's tool requires more time and effort than the development of the product itself. When the tool becomes the team's focus rather than the product they're building, you're not agile.

TIP

When shopping for tools to enable the product development work, remember that a tool that takes a minute a day to update is a good tool. A tool that takes longer than that to update is suboptimal.

Examples of tool pitfalls include the following:

- Backlog items that encourage teams to write novels instead of having face-to-face conversations
- Taskboard workflows that require the team to follow the tool's product development process rather than the process defined by the team
- Inconvenient access to the tool
- Reporting that encourages top-down or micromanaging behavior by stakeholders and managers
- Lack of ease in updating or progressively elaborating requirements

From our experience, the most agile teams use 3x5 cards and sticky notes. Their taskboards adapt to the work at hand. Their backlogs are on walls with full transparency. Remote teams use digital tools that simulate a similar experience as physical tools. Often the teams we work with find that the cost to purchase certain tools and the complexity to train and administer them outweighs their benefits. Use the tool litmus test in Chapter 6 to better understand how to value individuals and interactions over processes and tools.

Before investing in expensive enterprise tools, consider first whether you've taken advantage of the lightweight approach to transparency and reporting of an agile approach, using scrum:

- Access scrum teams' product backlogs and sprint backlogs, including burndown charts on-demand. (See Chapters 9, 10, and 11 to learn more about product and sprint backlogs and burndown charts.)
- Attend scrum teams' sprint reviews to see exactly what the teams have accomplished during the sprint.
- Attend daily scrums — 15 minutes or less — to hear exactly what the team is working on that day and the impediments.
- Engage with scrum teams about real-time truth and reality rather than waiting for delayed reports with limited context or opportunity to get instant clarification. Large organizations have successfully navigated agile waters with a simple spreadsheet. Refer to Chapter 11 for a simple release and sprint burndown spreadsheet.

When a sign of impeded agility due to tools arises, the following agile values and principles help navigate the situation:

- Value #1: Individuals and interactions over processes and tools.
- Principle #6: The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Principle #10: Simplicity — the art of maximizing the amount of work not done — is essential.

High Manager-to-Creator Ratio

Larger organizations likely have developed a heavy middle layer of managers. Many organizations haven't figured out how to function well without multiple managers handling personnel, training, and technical direction on development issues. However, you need to strike the right balance of managers and individuals who produce product. A high manager-to-creator ratio is another sign you're not agile.

Imagine two professional, rival futbol (American soccer) teams of 11 players each who both train intensively and prepare for a match against each other. Team B beats Team A 1-0.

Both teams go back to train for the next match. Team A's management calls on an analyst to provide a solution. After careful analysis of both teams, he sees that Team B has one player as goalkeeper, and ten spread across the field as defenders, midfielders, and forwards, while Team A plays ten goalkeepers at once and one forward to maneuver the ball down the field to the goal without any team members getting in the way.

Team A's management calls in a consultant to restructure the team. She finds what seems obvious: Team A is playing way too many goalkeepers. The consultant recommends that the team play half as many goalkeepers (five), and play five defenders who can relay instructions to the forward from the goalkeepers who have a view of the entire field. She also suggests doubling the assistant coaching staff to increase training and motivation of the forward to score goals.

At the next match, Team B again beats Team A, but this time 2-0.

The forward gets cut and the assistant coaches and defenders get recognized for their motivation strategy, but management calls for another analysis. As a result of the analysis, they build a more modern practice facility and invest in the latest shoe technology for the next season.

REMEMBER

Every dollar spent on someone who manages organizational processes is a dollar not spent on a product creator.

Empower people to self-organize in meeting the needs of your customers. You searched hard to find capable people through recruiting, training, and interviews. You hired them for their talent and experience. Leverage that investment by trusting them to get the job done. Minimize the investment you're making in people who don't create product.

Agile principles for navigating this situation follow:

- Principle #5: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- Principle #11: The best architectures, requirements, and designs emerge from self-organizing teams.

Highly autonomous and highly aligned teams who are empowered to do what is best for the customer are more effective. Value autonomy, mastery, and purpose over traditional management command and control tendencies. See Chapter 8 to learn the benefits of people given autonomy, mastery, and purpose.

Working around What Scrum Exposes

Another sign you're not agile is if you work around the problems exposed by scrum's transparency rather than addressing them head on. Agile frameworks such as scrum are not problem-solving models; they are exposure models. The problems exposed probably always existed, but with scrum there is clear transparency and the problems stand out like a sore thumb. For the agile mindset to thrive, the problems — not just the symptoms, but the true root causes — must be addressed and removed. See Chapter 4 to learn more about root cause analysis techniques.

For example, Toyota car manufacturing used an andon cord on its manufacturing lines for years. An employee pulled the cord to stop the line immediately whenever a problem was found that needed to be corrected. Workers were not reprimanded for pulling the cord. Employees then worked together to fix the problem at the root so it was not passed further down the line, becoming more and more expensive to resolve. The same is true for addressing what scrum exposes: Address the problem at the root cause and prevent it from affecting anyone else in your organization, ever.

Watch out for the following signs:

- Surface-level symptoms rather than the difficult root causes are addressed.
- Scrum teams identify organizational constraints, keeping them from operating in more agile ways, but they are repeatedly ignored.
- Scrum teams tend to skip sprint retrospectives because they feel that nothing ever changes.

These are clear signs that the leaders in the organization are not committed to the agile transformation change. If the elephant is in the room and everyone knows it, start eating the elephant one bite at a time.

Remembering the following agile principle will help:

- Principle #12: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Practicing Faux Agile

Last but not least is faux agile. Faux agile has the appearance of agile, but it's not.

In 2019, *Forbes*' Steve Denning shared the three laws of agile as a guide for organizations curious about the genuineness of their agile practices:

- **The law of the customer:** An obsession with delivering value to customers as the be-all and end-all of the organization
- **The law of the small team:** A presumption that all work be carried out by small, self-organizing teams, working in short cycles and focused on delivering value to customers
- **The law of the network:** A continuing effort to obliterate bureaucracy and top-down hierarchy to operate as an interacting network of teams, all focused on working together to deliver increasing value to customers

Faux agile, according to Denning, demonstrates the following characteristics:

- **Early-stage agile:** An agile journey never ends, so early on, the transition is incomplete. Results can be realized quickly from implementing a few agile practices or techniques, but time and experience are needed to stretch into more agile practices. The key is to avoid being fooled early on that "you've arrived."
- **Agile in name only:** For these organizations, you have to look beyond what they're saying to how they're operating. Do people demonstrate the agile mindset and move at the speed of a new startup? Or have they simply renamed existing non-agile practices with new agile terms?
- **Agile for software only:** Although the agile movement started with software development, not only people creating software benefit from agile techniques. Don't be deceived in thinking agile is only for software teams. For software teams to be successful, all parts of the organization — from senior leadership to human resources to finance to business development — must become agile.
- **Stalled agile journeys:** When agile collides with unmoving traditional practices, the organization may be forced to practice faux agile. When the tension becomes severe, a leader may dismiss agile leaders and coaches by saying that the organization is already agile. Agile principles and bureaucracy are mutually incompatible: In the end, only one can survive.
- **Branded agile:** From the Agile Manifesto and 12 principles have blossomed many different brands of agile. Beware of consultants and trainers who insist their branded version is the one and only true way.
- **Scaling frameworks:** Scaling up agile is an anti-pattern. Progressive elaboration and decomposition to the most independent and valuable increments for the customer creates agility. Be selective regarding the various frameworks to use what works best for your organization.
- **Agile lite:** Some organizations try to apply agile principles loosely without an agile mindset. Without an agile mindset, you're left with an inert, lifeless set of ceremonies.

In addition to the preceding list, you may see other subtle, yet common signs of faux agile:

- The organization continues to require even their scrum teams to follow a PMO-prescribed system development lifecycle (SDLC) with dated status reports and executive approval gates.

- The organization tries a hybrid approach that combines agile techniques with traditional roles. This situation is like setting sail and dropping anchor at the same time and expecting to move. You have to make a choice. Faux agile organizations continue to support and recruit traditional roles in addition to or instead of the types of roles that enable agile principles. These roles are not the same and require very different skill sets. See Chapter 7 to learn more about scrum roles.
- The organization uses technical product owners in addition to or in the absence of business- or customer-facing product owners.

If any of these ten anti-patterns exist in your organization, address them head on. The benefits of agile approaches to product development are extensive, proven, and worthwhile. Continually use agile values and principles to keep your journey to improved business agility on track.

IN THIS CHAPTER

- Finding support for successful agile transitions
- Getting involved with active agile communities
- Accessing resources for common agile approaches

Chapter 24

Ten Valuable Resources for Agile Professionals

Many organizations, websites, blogs, and companies exist to provide information about and support for agile product development. To help you get started, we compiled a list of ten resources that we think are valuable to support your journey to improved agility.

Agile Project Management For Dummies Online Cheat Sheet

www.dummies.com

You can use our online cheat sheet as a companion to this book as you start implementing the agile values and principles from the Agile Manifesto, as well as models outlined throughout the book. You'll find how-to guides, tools, templates, and other helpful resources for your agile toolkit. To get to the cheat sheet, go to www.dummies.com, and then type *Agile Project Management For Dummies* in the Search box.

Scrum For Dummies

In 2018, we published the second edition of *Scrum For Dummies* (Wiley) as a field guide not only to scrum but also to scrum in industries and business functions outside information technology (IT) and software development. Scrum can be applied in any situation where you want early empirical feedback on what you're building or pursuing.

Learn about scrum in industries such as game software development and tangible goods production (construction, manufacturing, hardware development) and in services such as healthcare, education, and publishing.

Explore scrum's applications in business functions, including operations, portfolio management, human resources, finance, sales, marketing, and customer service.

And in everyday life, see how scrum can help you organize your pursuits of dating, family life, retirement planning, and education.

The Scrum Alliance

`www.scrumalliance.org`

The Scrum Alliance is a nonprofit professional membership organization that promotes the understanding and usage of scrum. The alliance achieves this goal by promoting scrum training and certification classes, hosting international and regional scrum gatherings, and supporting local scrum user communities. To find a scrum user group in your area, search for your location at `www.scrumalliance.org/resources/groups`.

The Scrum Alliance site is also rich in blog entries, white papers, case studies, and other tools for learning and working with scrum. Chapter 18 lists many of the Scrum Alliance certifications.

The Agile Alliance

`www.agilealliance.org`

The Agile Alliance is the original global agile community, with a mission to help advance the 12 Agile Principles and common agile practices, regardless of

approach. The Agile Alliance site has an extensive resources section that includes articles, videos, and presentations. Find an index of independent and local agile community groups across the world at `www.agilealliance.org/communities/`.

International Consortium for Agile (ICAgile)

`icagile.com`

ICAgile is a community-driven organization helping people become agile through education, awareness, and certification. Its learning roadmap provides career path development support in business agility, enterprise and team agile coaching, value management, delivery management, agile engineering, agile testing, and DevOps.

Mind the Product and ProductTank

`www.mindtheproduct.com`

Mind the Product is the world's largest community of people passionate about product. They also founded ProductTank meetups so that product leaders could connect, share, and learn from each other. With over 150,000 members worldwide, they offer blogs, global, regional and local events, local meetups, and training by leading product management experts from all over the world. The resources at ProductTank tend to be high quality, and the content is both unique and relevant to the issues facing agile product development teams. Find a local ProductTank meetup in your area at `www.mindtheproduct.com/producttank`.

Lean Enterprise Institute

`www.lean.org`

Lean Enterprise Institute publishes books, blogs, knowledge bases, news, and events for the broader community of lean thinkers and practitioners. As you pursue agile product development, remember to incorporate lean thinking in all that you do. Lean.org is a good launching pad for you to explore the lean topics relevant to your situation.

Extreme Programming

`http://ronjeffries.com`

Ron Jeffries was one of the originators of the extreme programming (XP) development approach, along with Kent Beck and Ward Cunningham. Ron provides resources and services in support of XP's advancement on his ronjeffries.com site. The "What Is Extreme Programming?" section of the site summarizes the core concepts of XP. Other articles and extreme programming resources are also available in wiki format at `http://wiki.c2.com/?ExtremeProgrammingCorePractices`.

The Project Management Institute Agile Community

`www.projectmanagement.com/practices/agile`

The Project Management Institute (PMI) is the largest nonprofit project management membership association in the world. With nearly 3 million members in most countries throughout the world, PMI supports an agile community of practice and an agile certification, the PMI Agile Certified Practitioner (PMI-ACP).

The PMI website provides information and requirements for PMI-ACP certification along with access to papers, books, and seminars about agile project management.

Platinum Edge

`www.platinumedge.com`

Since 2001, our team at Platinum Edge has been helping companies maximize organizational return on investment (ROI). Visit our blog to get the latest insights on practices, tools, and innovative solutions emerging from our work with Global 1000 companies and the dynamic agile community.

We also provide the following services, which are outlined in more detail in Chapter 20:

- **Agile audits:** Auditing your current organizational structure and processes to create an agile implementation strategy that delivers bottom-line results. We also provide feedback on your current agile transition efforts to help you assess whether the investment you've made is generating the results you hoped for.
- **Recruiting:** With access to the best agile and scrum talent — because we've trained them — we help you find the right people to bootstrap your scrum teams, including scrum masters, scrum product owners, and scrum developers.
- **Training:** Public and private customized corporate agile and scrum training and certification, regardless of your level of knowledge. In addition to custom and non-certified training options, we offer the following certification classes:
  - Certified ScrumMaster (CSM)
  - Advanced Certified ScrumMaster (A-CSM)
  - Certified Scrum Product Owner (CSPO)
  - Certified Scrum Developer (CSD)
  - LeSS, Scrum@Scale, and SAFe approaches to scaling
  - PMI Agile Certified Practitioner (PMI-ACP) test preparation
- **Transformation:** Nothing is a larger factor of future success than proper coaching. We follow up on agile training with embedded agile coaching and mentoring to ensure that the right practices occur in the real world.

Index

Numbers

80/20 rule, 85-86

A

Abbe, Ernest, 37
AC+OC>V formula, 243, 267, 305-306
Accredited Kanban Consultant (AKC) and Trainer (AKT), 370
Achor, Shawn, 156
ADKAR model, 399-400
Advanced Certified Scrum Product Owner (A-CSPO), 370
Advanced Certified ScrumMaster (A-CSM), 370, 453
affinity estimating, 195-196
Agile Alliance, 12, 22, 450-451
agile champion, 360, 363
agile coach. *See* agile mentors
agile concepts, 91-92, 239
Agile Manifesto. *See also* Agile Principles; agile project management
  contracts in, 284
  cross-industry application of, 24
  discussion, 13, 22-23
  documentation in, 27
  focus of, 23
  Four Values of
    customer collaboration, 28, 278
    flexibility, 29-30
    overview, 24-25
    people over process, 25, 219
    working functionality, 26-28
  history of, 21-22
  litmus test, 47
agile mentors
  discussion, 102, 134, 258
  in pilot scrum team, 367-368, 371
  product team and, 124
  scrum team daily responsibilities of, 229
Agile Principles
  customer satisfaction in, 32-34
  discussion, 13, 31
  grouping of, 31
  history of, 22, 30. *See also* Platinum Principles
  litmus test, 47
  procurement and, 278-279
  product development in, 38-42
  quality in, 34-36
  teamwork in, 36-38
agile procurement management
  contracts in
    closing, 286
    cost structures, 282-283
    creating, 283-285
  discussion, 269, 278-279
  needs determination in, 280-281
  traditional versus, 279-280
  vendors in
    selecting, 281-283
    working with, 281, 285-286
agile product development
  adaptation in, 19
  advantages of, 39-41
  blank templates for
    management comparison, 41
    sprint backlog burndown chart, 221, 237
  determining duration in, 290
  discussion, 3, 16
  inspections in, 19
  litmus test, 47
  minimizing scope bloat in, 19
  principles of, 38-39
  success rates of, 18
  superiority of, 18-19
  traditional product management versus
    active products, 17-18
    overview, 16-17
    team permanence, 17
    value maximization, 18
  traditional project management versus, 41
  value cost formula in, 16

agile project management. *See also* risk management
adaptation in, 92
agile principles in
customer satisfaction, 32-34
overview, 31
quality, 34-36
attractive features of
to customers, 65-66
to development teams, 67-68
efficiency, 64-65
to executives, 64-65
greater value, 66-67
higher quality, 66
increased flexibility, 65-66
increased ROI, 65
less waste, 66-67
to managements, 66-67
benefits of
better metrics, 426-427
better product quality, 424
higher customer satisfaction, 423-424
higher team morale, 430
improved predictability, 429
increased collaboration, 426
increased ownership, 426
increased visibility, 427-428
iterative methodology, 50-52
more investment control, 428-429
optimized team structures, 429-430
overview, 49-50, 423
reduced risk, 425
blank templates for
customer satisfaction, 33
human interactions/processes, 26
identifying useful documentation, 27
certification, 104
changes caused by, 45-46
customer collaboration in, 28-29
customer satisfaction in, 53-54
discussion, 1, 7-8, 92-93
empirical control method, 14, 92
evolution of, 46
frameworks, 13
history of, 11-13
human interactions in, 25-26
inspection in, 92
litmus test, 47
methodologies, 13
mixing traditional and, 14
more information on, 2
Platinum Principles in
overview, 42
resisting formality, 42-43
teamwork, 43-44
visualization, 44-45
quality and, 34
superiority of, 23-24
synonyms in, 13-14
techniques, 13
tools, 13
traditional versus
failure detection, 63-64
flexibility, 55-57
overview, 54-55
project metrics, 62-63
quality, 60-61
ROI, 64
speed, 60-61
stability, 55-57
team performance, 61-62
unproductive activities, 57-60
transparency in, 92
waterfall project management versus
overview, 14-15
planning, 162
working functionality documentation in, 26-28
agile release train (ART) model, 390-391
Agile Retrospectives (Derby, Larsen), 249
agile scope management
agile principles related to, 270
change in
artifacts, 277-278
evaluating and prioritizing, 274-276
introducing, 274
overview, 272
understanding, 273-274
discussion, 269
time management and, 297-298
traditional versus, 270-272

agile transition. *See also* change management
committing to
agile champion, 360
individual, 359-360
organizational, 358-359
questions to consider, 361-362
timing, 362-363
creating environment for
overview, 368-369
Roadmap to Value, 369
training programs, 370-371
discussion, 357-358
key factors in successful
addressing exposed issues, 433
clear product vision, 433-434
clear roadmap, 433-434
coaching provided, 436
collocation, 322, 432
dedicated teams, 431-432
definition of done, 433
developer versatility, 434-435
overview, 371, 431
product owner empowerment, 434
servant leadership, 435
support for learning, 435-436
pilot scrum team in
agile champion, 363
agile mentor, 367-368, 371
development team, 366
overview, 364-365, 371
product owner, 365-366
scrum master, 366-367
transition team, 364-365
systems thinking in, 364-365
warning signs in
customer contact lacking, 440-441
disengaged stakeholders, 439-440
faux agile, 446-448
ignoring exposed issues, 445-446
lack of automation, 442
limited skill versatility, 441
long release cycles, 438-439
manager-to-creator imbalance, 444-445
non-shippable product increment, 437-438
overview, 437
prioritizing tools, 442-444
AKC (Accredited Kanban Consultant) and Trainer (AKT), 370
ART (agile release train) model, 390-391
artifacts
for cost management, 299, 306
discussion, 179
in scope management, 277, 288
in scrum approach, 102-103
for time management, 298-299
automated testing, 232, 342-344

B

barely sufficient documentation, 27-28, 40, 181
Barton, Brent, 261
Basex, 136
Beck, Kent, 13, 105, 452
Beedle, Mike, 13
Bennekum, Arie van, 13
big bang development, 50-51
Blanchard, Kenneth, 315
Booch, Grady, 12
budgets, 300-303
bugs, 332
bulletin boards, 112, 115-116
burndown chart
communicating progress with, 328-329
discussion, 206
sprint backlog, 220-222

C

Cagan, Marty, 71
CAL (Certified Agile Leadership), 370
capital expenditures (CapEx), 18, 256
capital redeployment, 305
cause-and-effect diagram, 87-88
CD (continuous deployment), 201, 231
Center for Applied Ethics, 315
Certified Agile Leadership (CAL), 370
Certified Enterprise Coach (CEC), 370
Certified Scrum Developer (CSD), 104, 370, 453
Certified Scrum Product Owner (CSPO), 104, 120, 370, 453
Certified Scrum Professional (CSP) for ScrumMasters (CSP-SM), Product Owners (CSP-PO), and Developers (CSP), 370

Certified Scrum Trainer (CST), 104, 370
Certified ScrumMaster (CSM), 104, 120, 370, 453
Certified Team Coach (CTC), 370
change management. *See also* agile transition
  discussion, 395-396
  executive ownership in
    overview, 412-413
    servant leadership, 413-414
  pitfalls to
    leadership pitfalls, 417-418
    organizational pitfalls, 415-416
    overview, 414
    warning signs, 418-420
  Platinum Edge model for
    overview, 401-402
    Step 1: implementation, 403-404
    Step 2: awareness, 404-405
    Step 3: transition team, 405-407
    Step 4: environment, 407-408
    Step 5: training and recruiting, 408
    Step 6: pilot and coaching, 408-410
    Step 7: Roadmap to Value, 410
    Step 8: gather feedback, 410-411
    Step 9: solidify improvements, 411
    Step 10: progressively expand, 412
  resistance in, 396-397
  SCARF model in, 396
  tools for
    ADKAR model, 399-400
    Kotter model, 400-401
    Lewin model, 398
    overview, 397
cheat sheet, 2, 449
chief product owner (CPO), 387
Christensen, Clayton, 75-77
CI (continuous integration), 35, 201, 231, 339
Cirillo, Francesco, 136
Cockburn, Alistair, 13, 112, 325
collaboration tools, 118, 120-121, 323
collective code ownership, 339
collocation
  key for transition, 432
  success rate of team, 322
  team, 61, 110-112
communication
  agile principles in, 324-325
  agile versus traditional, 324
  among dislocated teams, 111-112, 322-323
  discussion, 307
  effectiveness of different forms of, 112
  face-to-face
    effectiveness, 325-326
    goals, 114
    overview, 111-112, 114, 117
    tools, 115-116
  fidelity, 112
  human interactions and, 26
  inefficiency of email for, 58-59
  methods for, 325-328
  openness and, 137-138
  osmotic, 111-112
  product owner and, 125-126
  progress tracking and, 328-330
  resisting formality in, 42-43
  with stakeholders, 39, 125-126
  team size limits and, 146
  technological tools for
    collaboration tools, 118, 323
    constraints, 121
    overview, 116
    persistent chat, 117
    videoconferencing, 117, 322
    virtual collaboration board, 120-121
  in waterfall project management, 53
  in XP approach, 105
communication fidelity, 112
community of practice (CoP), 158, 383-384
comprehensive documentation, 26-28
context switching. *See* multitasking
continuous deployment (CD), 201, 231
continuous integration (CI), 35, 201, 231, 339
contract negotiation, 28-29
Conway, Melvin, 394
Conway's Law, 394
CoP (community of practice), 158, 383-384
cost management
  agile principles in, 299
  agile versus traditional, 300

discussion, 287, 299
lowering costs in, 304-306
using artifacts for, 299, 306
velocity and, 303-306
COVID-19 pandemic, 109-110, 119, 321
CPO (chief product owner), 387
credentials. See specific certifications
Crossing the Chasm (Moore), 167
Crystal methodology, 13
CSD (Certified Scrum Developer), 104, 370, 453
CSM (Certified ScrumMaster), 104, 120, 370, 453
CSP (Certified Scrum Professional) for ScrumMasters (CSP-SM), Product Owners (CSP-PO), and Developers (CSP), 370
CSPO (Certified Scrum Product Owner), 104, 120, 370, 453
CST (Certified Scrum Trainer), 104, 370
CTC (Certified Team Coach), 370
Cunningham, Ward, 13, 105, 452
customer representative. *See* product owner
customers
attraction to agility, 65-66
collaboration with, 28-29, 278
demands of, 70
determining needs of
customer-focused goals, 82
embracing early failure, 81
liberating structures, 83-84
overview, 79
root cause analysis (RCA), 84-88
scientific method, 79-80
story maps, 83
discussion, 69-70
identifying
empathy maps, 74-75
interviews, 78
job-to-be-done approach, 75-78
journey maps, 74-75
overview, 71
product canvas and, 71-74
product discovery workshops and, 79
involvement of, 53-54
satisfaction of, 32-34, 423-424
stakeholders as, 36
uncertainty and, 70

D

DA (Disciplined Agile) toolkit, 392-394
daily scrum. *See also* Roadmap to Value; shippable functionality; sprints
addressing roadblocks in, 217, 234-235, 354
discussion, 103, 114, 215-216
duration of, 217
guidelines for, 217-218
purpose of, 218
scrum master and, 58, 156
scrum teams and, 164
standing during, 217-218
topics addressed in, 216-217
death march, 40
decomposition of requirements. *See* requirements
definition of done
discussion, 26-27, 36
estimation poker and, 194
product increment and, 240-241
risk reduction in, 348-349
Demarco, Tom, 265
demonstrations, 59
dependencies, 178, 199
Derby, Esther, 249
development operations (DevOps), 201
development teams. *See also* multiple teams; team dynamics
building capability in, 157-158
building knowledge in, 157-158
crucial role of, 54
discussion, 35, 124
human interactions in, 26
long-term
advantages, 149-150
characteristics of effective, 153
knowledge, 150-151
overview, 149
Shu Ha Ri concepts, 153-154
team development phases, 151-153
working agreement, 152, 154-155
members of, 128-130
motivating
alignment, 157
autonomy, 155, 157
mastery, 156

development teams *(continued)*
overview, 155
purpose, 156
philosophy of
cross-functionality, 141-142
dedicated team, 140-141
overview, 139-140
ownership, 147
self-management, 144-145
self-organization, 143-144
size limits, 146-147
product development and, 230-231
project management and, 21-22
quality management by, 334
in scrum approach, 101-102
scrum team daily responsibilities of, 226
sprint backlog and, 102, 224
DevOps (development operations), 201
Digital.ai, 100
Disciplined Agile (DA) toolkit, 392-394
dislocated teams, 111-112, 321-323. *See also* remote work; team dynamics
distractions
common, 113
Pomodoro technique for avoiding, 136
scrum master and, 61-62, 113, 231
velocity and, 295
documentation, 27-28, 59-60
double work agile, 329
Drive (Pink), 155
Drucker, Peter, 82
DSDM approach, 12

E

earned value management (EVM), 329
EAT (executive action team), 386
Economics of Fatigue and Unrest, The (Florence), 37
Economy, Peter, 83
Edmondson, Amy, 413
email, 58-59
empathy maps, 74-75
empirical control method, 14, 17, 80
EMS (executive metascrum), 387-388
end user. *See* customers
epic user stories, 173, 190
Essentialism (McKeown), 262
estimation poker, 192-194
EVM (earned value management), 329
executive action team (EAT), 386
executive metascrum (EMS), 387-388
Extreme Programming Explained (Beck), 105
extreme programming (XP)
continuous deployment (CD) in, 201
continuous integration (CI) in, 35, 201
discussion, 40, 92, 105, 452
key practices of, 106-107, 231
other approaches and, 107-108
principles of, 105-106
quality management in, 338

F

facilitator. *See* scrum master
FDD approach, 12
feature requirements, 173
feedback, 50, 70
Fibonacci sequence, 190, 193
Fifth Discipline, The (Senge), 150
fishbone diagram, 87-88
fist of five, 133
"5 Simple Rules of Agile Portfolio Management, The" (Barton), 261
five why's, 87
flexibility, 29-30
Florence, P. Sargant, 37
formality, 43, 50, 114-115
Fowler, Martin, 13

G

gold-plating documentation, 27-28
Greanleaf, Robert K., 315
Greenleaf Center for Servant Leadership, 315
Grenning, James, 13
Gudith, Daniela, 265

H

hardware, 8
Harvard Business Review, 12-13, 156
Harvard Business School, 75

Highsmith, Jim, 13
human interactions, 25-26
Hunt, Andrew, 13

I

ICAgile (International Consortium for Agile), 370, 451
icons, 2
IEEE (Institute of Electrical and Electronics Engineers), 8
IID techniques, 12
Implementing Lean Software Development (Poppendieck, Poppendieck), 151
information radiator, 115, 198, 235-236, 330
information technology (IT) industry, 21, 201
Inspired (Cagan), 71
Institute of Electrical and Electronics Engineers (IEEE), 8
International Consortium for Agile (ICAgile), 370, 451
INVEST approach, 192
IRR (internal rate of return), 255-256
Ishikawa, Kaoru, 87-88
Ishikawa diagram, 87-88
IT (information technology) industry, 21, 201
iterations. *See also* sprints
  discussion, 14, 32, 51
  history of, 95
  short duration of, 35, 40, 56
  in waterfall project management, 94-95
  Winston Royce on, 94-95
iterative methodology, 12, 50-52

J

Jeffries, Ron, 13, 105, 452
job-to-be-done approach, 75-77
Johnson, Spencer, 315
Jones, Daniel T., 96
journey maps, 74-75
just-in-time (JIT) approach, 96, 161, 164-165. *See also* product roadmap; Roadmap to Value

K

kanban
  board, 115, 117
  discussion, 97-99
  operational support and, 202
  task board and, 223
Kanban Coaching Professional (KCP), 370
Kanban Management Professional (KMP I, II), 370
Kanban University, 370
Kellerman, Gabriella Rosen, 156
Kennedy, John F., 82
Kern, Jon, 13
Ketterin, Charles, 79
Kilburn, Tom, 93
KMP I, II (Kanban Management Professional), 370
Kotter, John, 400

L

large-scale scrum (LeSS), 380-384
Larsen, Diana, 249
law of diminishing returns, 267
Layton, Mark, 11, 21
lean canvas, 72
lean coffee approach, 158
Lean Enterprise Institute, 451
lean product development
  discussion, 92, 95
  JIT elaboration in, 96
  kanban practices in, 97-99
  other approaches and, 107-108
  principles of, 96-97
Lean Software Development (Poppendieck, Poppendieck), 96
Lencioni, Peter, 72
LeSS (large-scale scrum), 380-384
Lewin, Kurt, 398
Lewin model, 398
liberating structures, 83-84, 86
linear methodology, 50-52
Lipmanowicz, Henri, 83

M

Machine that Changed the World, The (Womack, Jones, Roos), 96
macrostructures, 84
Management by Objectives, 82
"Managing the Development of Large Software Systems" (Royce), 8, 12, 93
Manifesto for Agile Software Development, 13. *See also* Agile Manifesto
Marick, Brian, 13

Mark, Gloria, 265
Mark IIAiken Relay Calculator, 332
marshmallow challenge, 50-51
Martin, Robert C., 13
mass-production methods, 95-96
Maxwell, John C., 81
McCandless, Keith, 83
McKeown, Greg, 262
Mehrabian, Albert, 112
Mellor, Steve, 13
microstructures, 84
mind maps, 87
Mind the Product, 451
minimal marketable features, 197-198, 289
minimum viable product (MVP), 81, 83, 438
Moltke, Helmuth von, 162
Moore, Geoffrey, 167-168
multiple teams
  common challenges with, 374-375
  DA toolkit for, 392-394
  discussion, 373-374
  LeSS for
    combined meetings, 383-384
    combined sprint review, 382-383
    communities of practice (CoP), 383-384
    daily scrum, 383
    huge framework, 381-382
    overview, 380
    smaller framework, 380-381
  SAFe for
    configurations, 388-389
    core competencies, 390-391
    joint program PI, 391-392
    managers, 392
  Scrum@Scale approach for
    overview, 384-385
    product owner cycle, 387-388
    scrum master cycle, 385-386
    synchronization, 388
  vertical slicing for
    overview, 376
    scrum of scrums model, 376-379
multitasking, 113, 140, 153, 265-266
MVP (minimum viable product), 81, 83, 438

N

"New New Product Development Game, The" (Takeuchi, Nonaka), 12-13, 153
New One-Minute Manager, The (Blanchard, Johnson), 315
nine why's, 87
Nonaka, Ikujiro, 13, 153
Nordstrum Innovation Lab, 81

O

Ohno, Taiichi, 247
One Minute Manager (Blanchard, Johnson), 315
one-day sprints, 202
operational expenditures (OpEx), 256
osmotic communication, 111-112

P

pair programming, 43, 106, 232, 338
PAL I (Professional Agile Leadership), 370
Pareto, Vilfredo, 85
Pareto rule, 85-86
Patton, Jeff, 83
PDSA (Plan-Do-Study-Act) approach, 11-12
peer review, 232, 338-339
pen-pencil rule, 200
permanent teams. *See* development teams
personas, 187-189, 205
physical environments
  communication and
    face-to-face, 114-117
    information radiator, 115
    technological tools, 116-117
  creating
    dedicated areas, 112
    removing distractions, 113
    team collocation, 110-112
  discussion, 109-110
  tools for
    constraints, 121
    litmus test, 119
    overview, 118
    purpose, 119
    success-encouraging tools, 119-121
    virtual collaboration board, 120-121

PI (program increment) planning, 391-392
Pink, Daniel H., 155
Plan-Do-Study-Act (PDSA) approach, 11-12
planning. *See also* product roadmap; Roadmap to Value
  adaptation stage of, 165
  decomposition of requirements in, 164
  discussion, 161
  excessive, 50
  inspection stage of, 165
  JIT elaboration in, 161-162
planning poker, 192-194
Platinum Edge
  discussion, 21
  forced team dislocation and, 120
  history of, 12
  services offered by, 368, 452-453
Platinum Principles, 30, 42-45
PMI (Project Management Institute), 104, 452
PMI Agile Certified Practitioner (PMI-ACP), 104, 452
PMI-ACP (PMI Agile Certified Practitioner), 104, 452
PMI-ACP (Project Management Institute Agile Certified Practitioner) accreditation, 371
pods, 112
Pomodoro technique, 136
Poppendieck, Mary, 96
Poppendieck, Tom, 96
portfolio management
  agile principles in, 253-255
  CapEx in, 256
  discussion, 253-254
  forecasting investment returns in
    agile mentors, 258
    overview, 256
    product mix, 260-261
    risk-value matrix, 257-258
    short- versus long-term decisions, 259-260
    value and risk prioritization, 257-259
  IRR in, 255-256
  managing
    avoiding multitasking, 265-266
    continuous prioritization, 267-268
    engaging vendors, 264
    incremental funding, 264
    keys to effectiveness, 264-265
    other factors, 262, 264
    overview, 261
    reducing complexity, 261-262
    reducing investments, 266-267
    revising investments, 268
    visualization, 262-263
  multitasking and, 265-266
  OpEx in, 256
  other costs to consider in, 256
  SWOT analysis in, 254
Practice of Management, The (Drucker), 82
processes, 25-26
product backlog. *See also* release planning
  communicating progress with, 328-329
  completing, 180-182
  continually updating, 189
  cost management and, 299
  discussion, 27, 175, 179
  estimates, 182
  Pareto rule with, 86
  product owner and, 224
  product roadmap and, 180
  risk management and, 353
  in scrum approach, 102
  story maps and, 83
  time management and, 299
  user stories and, 182
product canvas, 71-74
product discovery, 79, 127-128
product increment, 102-103, 240-241. *See also* shippable functionality; sprint review
product mix, 260-261
product owner
  characteristics of good, 125-127
  communication and, 114, 125-126
  daily scrum and, 217
  discussion, 32-33, 53-54
  product backlog and, 102, 224
  product development and, 128, 230
  product discovery and, 127-128
  quality management and, 334, 339-340
  responsibilities of, 124-125
  review of user stories, 233
  in scrum approach, 101
  scrum team daily responsibilities of, 225, 236-237
  sprint review meeting and, 245
  team support from, 61

product roadmap. *See also* product backlog; user stories
  cost management and, 299
  creating
    arranging product features, 175-176
    assessing value and risk, 178-179
    defining product requirements, 173-175
    determining time frames, 180
    estimating and prioritizing, 176-180
    identifying stakeholders, 172-173
    overview, 171-172
    saving, 180
    updating, 180
  discussion, 162
  product canvas and, 74
  risk management and, 353
  in scrum approach, 108
  time management and, 299
product scope, 270
product team, 124
product vision statement
  creating
    drafting, 167-169
    finalizing, 170
    overview, 165-166
    product objective, 167
    revising, 169-170
    validating, 169-170
  discussion, 162, 340, 353
  product canvas and, 74
  in scrum approach, 108
ProductTank, 451
Professional Agile Leadership (PAL I), 370
Professional Scrum Developer (PSD I), 370
Professional Scrum Master (PSM I, II, III), 370
Professional Scrum Product Owner (PSPO I, II, III), 370
program increment (PI) planning, 391-392
progressive elaboration of requirements. *See* requirements
project management, 8-12. *See also* agile project management; traditional project management
Project Management Institute Agile Certified Practitioner (PMI-ACP) accreditation, 371
Project Management Institute (PMI), 104, 452
Project Mercury, 12
project scope, 270
Prosci ADKAR model, 399-400
PSD I (Professional Scrum Developer), 370
PSM I, II, III (Professional Scrum Master), 370
PSPO I, II, III (Professional Scrum Product Owner), 370

Q

quality management
  agile principles in, 331-332
  in agile project management, 424
  daily testing and, 335
  by development teams, 334
  discussion, 34-36, 331
  feedback loops in, 333
  proactive
    acceptance criteria, 340
    collective code ownership, 339
    continuous integration (CI), 339
    development techniques, 338
    face-to-face communication, 340-341
    good design, 337-338
    organizational commitment, 337
    overview, 335-336
    pair programming, 338
    peer review, 338-339
    role of product owner, 339-340
    sustainable work pace, 341
    technical excellence, 337-338
  risk and, 333
  traditional versus agile, 332-333
  via automated testing, 342-344
  via inspecting and adapting, 341-342

R

R&D (research and development), 80
"Racing in Reverse," 40
RCA (root cause analysis), 84-88
Reece, Andrew, 156
refactoring, 105-106
relative estimating, 13, 108, 178
release planning. *See also* sprints
  cost management and, 299
  creating, 198-200
  discussion, 82, 163
  key activities of, 197-198
  minimal marketable features, 198

minimal marketable features and, 197
product deployment in
market preparation, 204-205
operational support, 201-203
organizational activities, 203-204
overview, 200-201
risk management and, 354
in scrum approach, 108
sprints and, 199-200
time management and, 299
in XP approach, 106
release train engineer (RTE), 390
remote work. *See also* dislocated teams; team dynamics
communication and
overview, 111-112, 116
tools, 117-121
discussion, 109-110
requirements, decomposition of
affinity estimating, 195-196
epic user stories, 190
estimation poker, 193-194
features, 173, 190
guide, 190-191
INVEST approach, 192
overview, 164, 183
during sprints, 190
tasks, 174
themes, 173, 190
user stories, 174
research and development (R&D), 80
return on investment (ROI), 167
Ries, Eric, 81
risk management
agile principles in, 345
in agile project management, 54, 425
agile versus traditional, 345-347
artifacts and meetings for, 353-354
end-of-sprint and, 241
inherent risk reduction in
definition of done, 348-349
failing fast concept, 351-352
overview, 348
self-funding development, 349-351
quality and, 333
in waterfall project management, 53
risk-value matrix, 257-258
roadblocks, 207, 217, 234-235
Roadmap to Value. *See also* product backlog; release planning
adaptation stage in, 165
creating, 180
decomposition of requirements in, 164
discussion, 369
inspection stage in, 165
product roadmap in
arranging product features, 175-176
assessing value and risk, 178-179
creating, 171-172
defining product requirements, 173-175
estimating and prioritizing, 176-180
identifying stakeholders, 172-173
saving, 180
updating, 180
product vision statement in
creating, 166-170
overview, 165-166
release planning in
creating, 198-200
key activities, 197-198
minimal marketable features, 197-198
release sprints, 199-200
scope management and, 273-274
stages of, 162-164
Robichaux, Alexi, 156
Rock, David, 396
ROI (return on investment), 167
role
of agile mentor, 134
of development team members, 128-130
discussion, 123-124
of product owner
overview, 124
product development, 128
product discovery, 127-128
responsibilities, 124-126
of scrum master
characteristics, 131-132
responsibilities, 130-131
of stakeholders, 125, 132-134

Roos, Daniel, 96
root cause analysis (RCA), 84-88
Royce, Winston, 8-9, 12, 93
RTE (release train engineer), 390

S

Scaled Agile Framework (SAFe), 388-392
SCARF model, 396
Schwaber, Ken, 12-13
scientific method, 79
scope bloat, 10
scope creep, 272
Scrum Alliance, 12, 24, 104, 370, 450
scrum approach. *See also* daily scrum
  additional features of, 108
  agile mentors in, 102
  artifacts in, 102-103
  customer satisfaction and, 32
  discussion, 92, 100
  events in, 103
  history of, 12-13
  other approaches and, 107-108
  roles in, 101-102
  sprints in, 100-101
  stakeholders in, 102
  terminology in, 53
Scrum For Dummies (Layton), 11, 450
scrum master
  characteristics of good, 131-132, 144-145
  coaching provided by, 230
  discussion, 53-55, 102
  distractions prevented by, 61-62, 113, 231
  meetings and, 58, 156
  quality management and, 334
  responsibilities of, 130-131
  roadblocks addressed by, 207, 217, 234
  scrum team daily responsibilities of, 227, 236-237
  servant-leader role of, 314-315
  sprint retrospective meetings and, 249-250
scrum room, 112
scrum teams. *See also* daily scrum; multiple teams; physical environments; team dynamics
  communication among dislocated, 111-112
  daily meetings, 164
  discussion, 54-55, 92, 123-124
  philosophy of
    cross-functionality, 141-142
    dedicated team, 140-141
    overview, 139-140
    ownership, 147
    self-management, 144-145
    self-organization, 143-144
    size limits, 146-147
  pilot
    agile mentor, 367-368
    development team, 366
    overview, 364-365
    product owner, 365-366
    scrum master, 366-367
    stakeholders, 367
  product discovery workshops and, 79
  sprint daily responsibilities of
    agile mentors, 228-229
    development team members, 226
    overview, 224-225, 236-237
    product owner, 225
    scrum master, 227
    stakeholders, 228
  sprint reviews and, 103
  stakeholders and, 102
Scrum@Scale approach, 384-388
self-management
  excellence product of, 337
  of scrum master, 144-145
  team dynamics and, 309-313
Senge, Peter, 150
servant leadership, 314, 413-414, 435
shadowing, 43
shared drives, 109-110
Sherwart, Walter, 11-12
shippable functionality. *See also* user stories
  developing in, 230-231
  discussion, 229-230
  elaborating in, 230
  sprint review and, 240-241
  verifying in
    automated testing, 232
    identifying roadblocks, 234-235
    mob programming, 232-233
    overview, 231

pair programming, 232
peer review, 232
product owner review, 233
Shu Ha Ri concepts, 153-154
Sinek, Simon, 166
Slack, 117
Slack (Demarco), 265
SMART goals, 82
software development, 8, 93-94, 201
Software Development as a Cooperative Game (Cockburn), 325
Spears, Larry, 314
Spira, Jonathan, 136
sprint backlog
communicating progress with, 328-329
cost management and, 299
creation of, 206-207
development teams and, 102, 224
discussion, 27
overview, 206
risk management and, 354
in scrum approach, 102
time management and, 299
tracking progress with
burndown chart, 206, 220-222
end of day, 237-238
overview, 219
sprint planning
components of, 206
discussion, 82, 205
duration of, 205
meeting for
create backlog tasks, 211-212
overview, 207-209
risk management, 354
selecting user stories, 209-210
setting goals, 209-210
task guidelines, 212-213
sprint retrospective. *See also* Roadmap to Value
adapting in, 250
discussion, 103, 164, 239, 245
goal of, 246-247
inspecting in, 250, 342
meeting
duration, 248
guidelines, 249
scrum master, 249-250
topics for inspection, 248-249, 354
planning for, 247
sprint review. *See also* Roadmap to Value
communicating progress with, 328-329
determining future investment during, 243
discussion, 84, 164, 239-240
meeting
collecting feedback, 244-245, 342
duration, 242
feedback cycle, 241-242, 354
guidelines, 242-244
overview, 241
product owner tasks, 245
scheduling, 242
preparation for, 240
scrum teams and, 103
shippable functionality and, 240-241
value of, 244
sprints. *See also* release planning; scrum teams; user stories
discussion, 14, 32, 51, 53
planning for, 103, 163
release planning and, 199-200
requirement decomposition during, 190-191
scientific method and, 80
in scrum approach
adaptations, 101
inspections, 100-101
overview, 100
scrum team daily responsibilities and
agile mentors, 229
development team members, 226
end of day, 236-237
overview, 224-225
product owner, 225
scrum master, 227
stakeholders, 228
shippable functionality and
developing, 230-231
elaborating, 230
identifying roadblocks, 234-235
overview, 229-230
verifying, 231-233
short duration of, 35, 40, 56
stakeholders and, 103
testing during daily, 335

stakeholders
communication with, 39, 125-126
creating user stories with, 187-189
as customer, 32
discussion, 132-134
disengaged, 439-440
pilot scrum teams and, 367
product discovery workshops and, 79
product roadmap and, 172-173
product team and, 124
role of, 125
in scrum approach, 102
scrum team daily responsibilities of, 228
scrum teams and, 102
sprints and, 367
user stories and, 186-187
Standish Group
"2015 Chaos Report," 345-346
study on project success/failure rates, 11, 18
study on scope bloat, 10
stay-at-home orders, 109-110, 119
story maps, 83
Sutherland, Jeff, 12-13
swarming, 108, 141-142, 211
SWOTs (strengths, weaknesses, opportunities, and threats), 254
system architect/engineer, 390-391
systems thinking, 364-365

T

Takeuchi, Hirotaka, 13, 153
task board
communicating progress with, 328-330
kanban similar to, 223
risk management and, 354
tracking progress with
components, 222-223
overview, 219, 222
reading, 223-224
task requirements, 174
TDD (test-driven development), 106, 338
team coach. *See* scrum master
team dynamics. *See also* multiple teams
agile principles in, 307-308
agile versus traditional, 308-309, 429-430
collocation and, 61
discussion, 307
managing
cross-functionality, 317-319
dedicated teams, 316-317
dislocated teams, 321-323
limited size, 320-321
openness, 319-320
overview, 309
self-management, 309-313
servant leadership, 314-315
Team Kanban Practitioner (TKP), 370
team room, 112
Teams, 117
teams. *See also* team dynamics
dedicated, 140-141, 431-432
discussion, 26, 36-37
focus, 61-62
in Platinum Principles, 43-44
stakeholders and, 36
strategies for effective, 38
support for, 61
tighter control and, 62-63
visualization techniques and, 71-72
in waterfall project management, 53
technical excellence, 35, 337-338
telepresence robots, 117
test-driven development (TDD), 106, 338
testing
during daily sprints, 335
discussion, 35-36, 51, 60
in XP approach, 105
theme requirements, 173, 190
Thomas, Dave, 13
thrashing. *See* multitasking
time management. *See also* velocity
advantages of agile, 288
agile principles and, 287-288
agile versus traditional, 288
discussion, 287
lowered costs and, 305
multiple teams for, 298
scheduling in, 289
scope changes and, 297-298
using artifacts for, 298-299

TKP (Team Kanban Practitioner), 370
Toyota Production System
  discussion, 247
  just-in-time (JIT) elaboration and, 96
  kanban practices and, 97, 223
traditional procurement management, 279-280
traditional project management
  agile product development versus, 41
  agile versus
    failure detection, 63-64
    flexibility, 55-57
    overview, 54-55
    project metrics, 62-63
    quality, 60-61
    ROI, 64
    speed, 60-61
    stability, 55-57
    team performance, 61-62
    unproductive activities, 57-60
  cost management in, 299-300
  customer involvement in, 28-29
  death march in, 40
  inflexibility of, 23, 30
  modern products and, 50
  planning in, 161
  risk management in, 53
  scope in, 270
  unrealistic nature of, 50-51
traditional scope management, 270-272
Tuckman, Bruce, 151
Tuckman phases to performance, 151-153

U

uniform product developer titles, 44
useful documentation, 27-28
user stories. *See also* shippable functionality; sprints
  creating
    acceptance criteria of, 184
    advantages, 186
    determining product requirements, 189-190
    electronic tools, 185
    identifying stakeholders, 186-187
    identifying users, 187-188
    three C's formula, 185
  decomposition of requirements in, 174
  discussion, 13, 82, 183, 290
  INVEST approach and, 192
  product backlog and, 182
  product owner review of, 233
  requirement decomposition in
    affinity estimating, 195-196
    estimation poker, 192-194
  in scrum approach, 108
User Story Mapping (Patton, Economy), 83

V

value cost formula, 16
values
  discussion, 134-135
  establishing new
    commitment, 135-136
    courage, 138-139
    focus, 136-137
    openness, 137-138
    respect, 138
velocity
  adjusting, 291
  attaining consistent, 296-297
  calculating, 291-292
  cost management and, 303-304
  for determining other costs, 306
  discussion, 199
  estimating timeline using, 292-294
  increasing, 246, 294-295, 304
  for long-range planning, 289-290
  monitoring, 291
  in scrum approach, 108
vertical slicing, 376-379
videoconferencing tools, 117, 120
virtual collaboration board, 120-121
visualization activities
  for determining customer needs, 83
  discussion, 44-45
  for identifying customers
    empathy maps, 74-75
    journey maps, 74-75
  product canvas for, 71-72, 74

W

Wake, Bill, 192
waterfall project management
  agile versus, 14-15, 162, 346
  discussion, 8-9, 12, 14
  failure in agile versus, 351-352
  inflexibility of, 22, 30
  iterations in, 94-95
  major aspects of, 53
  origins of, 93-94
  risk management in, 53
  scope creep in, 283
  unrealistic nature of, 50-51
whiteboards, 111-112, 115-116
Womack, Japes P., 96
working agreement, 154-155
working functionality
  comprehensive documentation versus, 26-28
  discussion, 328
  product increment and, 102-103
Wujec, Tom, 50

X

XP. *See* extreme programming

About the Authors

Mark C. Layton, known globally as Mr. Agile, is an organizational strategist and Scrum Alliance certification instructor with over 20 years in the project and program management field. He is the 2020 president of the Project Management Institute (PMI) Southern Nevada Chapter as well as the Los Angeles chair for the Agile Leadership Network. Mark is the author of the international *Scrum For Dummies* and *Agile Project Management For Dummies* book series, creator of the Agile Foundations Complete Video Course, and is the founder and managing member of Platinum Edge, LLC — an enterprise transformation company that uses organizational design to help businesses with their agile transformation journey.

Prior to founding Platinum Edge in 2001, Mark developed his expertise as a consulting firm executive, a program management coach, and an in-the-trenches project leader. He also spent 11 years as a Cryptographic Specialist for the US Air Force, where he earned both Commendation and Achievement medals for his accomplishments.

Mark holds MBAs from the University of California, Los Angeles, and the National University of Singapore; a B.Sc. (summa cum laude) in Behavioral Science from Pitzer College/University of La Verne; and an A.S. in Electronic Systems from the Air Force's Air College. He is also a Distinguished Graduate of the Air Force's Leadership School, a Certified Scrum Trainer (CST), a certified Project Management Professional (PMP), a recipient of Stanford University's advanced project management certification (SCPM), and a certified Scaled Agile Framework Program Consultant (SAFe SPC).

In addition to his books and videos, Mark is a frequent speaker at major conferences on Lean, Scrum, DevSecOps, and other agile solutions.

Additional information can be found at `https://platinumedge.com`.

Steven J Ostermiller is a trainer, coach, and mentor helping organizations evolve to maximize business value and minimize risk through lean and agile principles and practices. He is the founder and executive director of Utah Agile (in partnership with Agile Alliance, Scrum Alliance, and Silicon Slopes), a non-profit professional community committed to increasing agility for Utah businesses, technology, and individuals. Steve developed and taught the agile project management curriculum for Ensign College in Salt Lake City, Utah and serves on its project management advisory board. He was technical editor of *Scrum For Dummies* and Pearson Education's Agile Foundations Complete Video Course.

Steve's expertise comes from nearly 20 years of successes and failures as a project manager, product manager, operations executive, scrum master, and agile coach,

trainer, and consultant. He has worked with executive leadership and product development teams in a variety of industries on the Fortune lists. He is a Scrum Alliance Certified Scrum Trainer (CST), an ICAgile Certified Professional in Coaching and Facilitation (ICP-ACC, ICP-ATF), and a Project Management Professional, and holds a B.S. in Business Management/Organizational Behavior from the Marriott School of Management at Brigham Young University.

Dean J. Kynaston is an experienced scrum master, coach, and mentor with nearly 20 years of experience empowering leaders, teams, and individuals to become more agile. With Steve, he taught the agile project management curriculum at Ensign College in Salt Lake City, Utah. Dean was also a technical editor for *Scrum For Dummies* and the author of multiple Platinum Edge blog articles.

A Platinum Edge alumni himself and former Project Management Professional (PMP), Dean has worked with multiple organizations and seen much success applying agile values and principles. He has worked with executive leadership and individual teams in both for-profit and non-profit industries, particularly in real estate, construction, automotive, healthcare, and pharmaceuticals. He holds an MBA from Boise State University, is a Certified Scrum Professional (CSP-SM and CSP-PO), and earned a B.S. in Business Management with an emphasis in finance from the Marriott School of Management at Brigham Young University. As a busy father of eight children, Dean finds many opportunities to use scrum with his family team as well.

Dedications

To my special snowflake. I'm so thankful to have you in my life and to be sharing this wild ride with you. The world hasn't seen anything yet. — Mark

To Gwen, my complete and final answer. And to our five littles, who give me every reason to continuously inspect and adapt. — Steve

To my angel mother, an amazing woman and nurturer who led by example in "enduring to the end." — Dean

Authors' Acknowledgments

We'd like to again thank the numerous people who contributed to the previous editions of this book and helped make it a reality. We're also very grateful to those who helped make this third edition a more valuable field guide: Andrew Workmon, for his insight and technical editing; Caroline Patchen for yet again ensuring that these concepts are more easily understood through clear visualization; Craig Larman, Bas Vodde, Jeff Sutherland, and Dean Leffingwell for providing scaling options to the public and for their valuable feedback on the de-scaling chapter; to Susan Pink and Debbye Butler; and to Steve Hayes and the broader John Wiley & Sons team. You are all fantastic professionals; thank you for the opportunity to make this book even better.

And a shout-out to the signers of the Agile Manifesto. Thanks for coming together, finding common ground, and kickstarting the discussion that inspires us to keep becoming more agile.

Publisher's Acknowledgments

Executive Editor: Steve Hayes
Project Editor: Susan Pink
Copy Editor: Susan Pink
Technical Editor: Andrew Workmon
Proofreader: Debbye Butler
Sr. Editorial Assistant: Cherie Case
Production Editor: Mohammed Zafar Ali
Cover Image: © 3d_kot/Shutterstock

O'REILLY®

FREE Full-Length Practice PMI-ACP Exam Inside

Head First Agile

A Brain-Friendly Guide

A Learner's Companion to Understanding Agile and Passing the PMI-ACP® Exam

Load agile concepts right into your brain

Scrum helped Amy keep her

Boss your code around with XP

Unravel th of Lean an

Explore how task boards keep everybody on track

Early Release

RAW & UNEDITED

Andrew Stellman & Jennifer Greene

Praise for *Head First Agile*

Your name could be here! We're looking for early praise from project managers, developers, business anaylsts, and anyone else who's read the early release of our book. Contact us at info@stellman-greene.com.

Praise for *Head First Agile*

Your name could be here! We're looking for early praise from project managers, developers, business anaylsts, and anyone else who's read the early release of our book. Contact us at info@stellman-greene.com.

Praise for other *Head First books*

"With *Head First C#*, Andrew and Jenny have presented an excellent tutorial on learning C#. It is very approachable while covering a great amount of detail in a unique style. If you've been turned off by more conventional books on C#, you'll love this one."

—Jay Hilyard, software developer, coauthor of *C# 3.0 Cookbook*

"I've never read a computer book cover to cover, but this one held my interest from the first page to the last. If you want to learn C# in depth and have fun doing it, this is *the* book for you."

— Andy Parker, fledgling C# programmer

"Going through this *Head First C#* book was a great experience. I have not come across a book series which actually teaches you so well…This is a book I would definitely recommend to people wanting to learn C#"

—Krishna Pala, MCP

"*Head First Web Design* really demystifies the web design process and makes it possible for any web programmer to give it a try. For a web developer who has not taken web design classes, *Head First Web Design* confirmed and clarified a lot of theory and best practices that seem to be just assumed in this industry."

—Ashley Doughty, senior web developer

"Building websites has definitely become more than just writing code. *Head First Web Design* shows you what you need to know to give your users an appealing and satisfying experience. Another great Head First book!"

—Sarah Collings, user experience software engineer

"*Head First Networking* takes network concepts that are sometimes too esoteric and abstract even for highly technical people to understand without difficulty and makes them very concrete and approachable. Well done."

— Jonathan Moore, owner, Forerunner Design

"The big picture is what is often lost in information technology how-to books. *Head First Networking* keeps the focus on the real world, distilling knowledge from experience and presenting it in byte-size packets for the IT novitiate. The combination of explanations with real-world problems to solve makes this an excellent learning tool."

— Rohn Wood, senior research systems analyst, University of Montana

Other related books from O'Reilly

Learning Agile

Beautiful Teams

Applied Software Project Management

Making Things Happen

Practical Development Environments

Process Improvement Essentials

Other books in O'Reilly's *Head First* series

Head First PMP

Head First C#

Head First Java

Head First Object-Oriented Analysis and Design (OOA&D)

Head First HTML with CSS and XHTML

Head First Design Patterns

Head First Servlets and JSP

Head First EJB

Head First SQL

Head First Software Development

Head First JavaScript

Head First Physics

Head First Statistics

Head First Ajax

Head First Rails

Head First Algebra

Head First PHP & MySQL

Head First Web Design

Head First Networking

Head First Agile

Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing ▪ Cambridge ▪ Köln ▪ Sebastopol ▪ Tokyo

Head First Agile

by Andrew Stellman and Jennifer Greene

Copyright © 2016 Andrew Stellman and Jennifer Greene. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Editor: Nan Barber

Design Editor: Louise Barr

Cover Designers: Karen Montgomery, Louise Barr

Production Editors: Melanie Yarbrough

Indexer: Bob Pfahler

Proofreader: Rachel Monaghan

Page Viewers: Quentin the whippet and Tequila the pomeranian

Printing History:

January 2017: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First Agile*, and related trade dress are trademarks of O'Reilly Media, Inc.

PMI-ACP, PMP, and PMBOK are registered marks of Project Management Institute, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: TBD
[LSI] [TBD]

Dedication coming soon...

THANKS FOR BUYING OUR BOOK! WE REALLY LOVE WRITING ABOUT THIS STUFF, AND WE HOPE YOU **GET A KICK OUT OF READING IT**...

...BECAUSE WE KNOW YOU'RE GOING TO **DO GREAT WORK** WITH AGILE!

Andrew

Photo by Nisha Sondhe

Jenny

Andrew Stellman is a developer, architect, speaker, agile coach, project manager, and expert in building better software. Andrew is an author and international speaker, with top-selling books in software development and project management, and world-recognized expert in transforming and improving software organizations, teams, and code. He has architected and built large-scale software systems, managed large international software teams, and consulted for companies, schools, and corporations, including Microsoft, the National Bureau of Economic Research, Bank of America, Notre Dame, and MIT. Andrew's had the privilege of working with some pretty amazing programmers during that time, and likes to think that he's learned a few things from them.

Jennifer Greene is an agile coach, development manager, business analyst, project manager, tester, speaker, and authority on software engineering practices and principles. She's been building software for over twenty years in many different domains including media, finance, and IT consulting. She's worked with teams of excellent developers and testers to tackle tough technical problems and focused her career on finding and fixing the habitual process issues that crop up along the way.

Jenny and Andrew have been building software and writing about software engineering together since they first met in 1998. Their first book, **Applied Software Project Management**, was published by O'Reilly in 2005. They published their first book in the Head First series, **Head First PMP**, in 2009, and their second one, **Head First C#**, in 2009. Both books have gone on to third and, soon, fourth editions.

They founded Stellman & Greene Consulting in 2003—their first project as a consulting company was a really fascinating software project for scientists studying herbicide exposure in Vietnam veterans. And when they're not building software or writing books, they do a lot of speaking at conferences and meetings of software engineers, architects, and project managers.

Check out their website, **Building Better Software**, at **http://www.stellman-greene.com**.

Table of Contents (Summary)

	Intro	xxv
1	What is agile? *Principles and practices*	?
2	Organizations, constraints, and projects: *In good company*	?
3	The process framework: *It all fits together*	?
4	Project integration management: *Getting the job done*	?
5	Scope management: *Doing the right stuff*	?
6	Time management: *Getting it done on time*	?
7	Cost management: *Watching the bottom line*	?
8	Quality management: *Getting it right*	?
9	Human resource management: *Getting the team together*	?

Table of Contents (the real thing)

Intro

Your brain on agile. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing enough to really "get" agile—and maybe even get through the PMI-ACP certification exam?

Who is this book for?	xx
We know what you're thinking	xxi
Metacognition: thinking about thinking	xxiii
Here's what YOU can do to bend your brain into submission	xxii
Read me	xxvi
The technical review team	xxvii
Acknowledgments	xxviii

What is agile?

1 Principles and practices

It's an exciting time to be agile! For the first time, our industry has found a real, sustainable way to solve problems that generations of software development teams have been struggling with. Agile teams use simple, straightforward **practices** that have been proven to work on real-life projects. But wait a minute... if agile is so great, why isn't everyone doing it? It turns out that in the real world, a practice that works really well for one team causes serious problems for another team, and the difference is the team **mindset**. So get ready to change the way you think about your projects!

Do these problems seem familiar?	2
Projects don't have to be this way	4
Your problems…already solved	5
What you need to be a good project manager	6
Understand your company's big picture	10
Your project has value	11
Portfolios, programs, and projects have a lot in common	12
Portfolios, programs, and projects all use charters	13
What a project IS…	17
… and what a project is NOT	17
A day in the life of a project manager	19
How project managers run great projects	21
Project management offices help you do a good job, every time	22
Good leadership helps the team work together	23
Project teams are made of people	24
Operations management handles the processes that make your company tick	26
A t certification is more than just passing a test	30
Meet a real-life PMP-certified project manager	31
Exam Questions	32

In a daily standup meeting, everyone on the team stands for the duration of the meeting. That keeps it short, sweet, and to the point.

But is this guy really paying attention to what his teammates are saying?

2 Agile values and principles

Mindset meets method

There's no "perfect" recipe for building great software.

Some teams have had a lot of success and seen big improvements after adopting agile practices, methods, and methodologies, while others have struggled. We've learned that the difference is the mindset that the people on the team have. So what do you do if you want to get those great agile results for your own team? How do you make sure your team has the right mindset? That's where the **Agile Manifesto** comes in. When you and your team get your head around its **values and principles**, you start to think differently about the agile practices and how they work, and they start to become *a lot more effective*.

A day in Kate's life	38
Kate wants a new job	39
There are different types of organizations	42
Kate takes a new job	47
Stakeholders are impacted by your project	49
More types of stakeholders	50
Your project team has lots of roles too	51
Back to Kate's maintenance nightmare	52
Managing project constraints	54
You can't manage your project in a vacuum	58
Kate's project needs to follow company processes	59
Kate makes some changes…	60
… and her project is a success!	61
Exam Questions	64

Managing projects with Scrum

The rules of Scrum

The rules of Scrum are simple. Using it effectively is not so simple.

3

Scrum is the most common agile methodology, and for good reason: the **rules of Scrum** are straightforward and easy to learn. Most teams don't need a lot of time to pick up **the events, roles, and artifacts** that make up the rules of Scrum. But for Scrum to be really effective, they need to really understand **the values of Scrum** and the Agile Manifesto principles, which help them get into an effective mindset. Because while Scrum seems simple, the way a Scrum team constantly **inspects and adapts** is a whole new way of thinking about projects.

Cooking up a project	70
Projects are like recipes	72
If your project's really big, you can manage it in phases	74
Phases can also overlap	75
Break it down	76
Anatomy of a process	79
Combine processes to complete your project	82
Knowledge areas organize the processes	83
The benefits of successful project management	89
Exam Questions	91

With a new Product Owner, the team should be able to figure out the most valuable features to include in the next sprint.

4 Agile planning and estimation

Generally Accepted Scrum Practices

Agile teams use straightforward planning tools to get a handle on their projects. Scrum teams plan their projects together so that everybody on the team commits to each sprint's goal. To maintain the team's **collective commitment**, planning, estimating and tracking need to be simple and easy for the whole team to do as a group. From **user stories** and **planning poker** to **velocity** and **burndown charts,** Scrum teams always know what they've done and what's left to do. Get ready to learn the tools that keep Scrum teams informed and in control of what they build!

These clients are definitely not satisfied 100
The day-to-day work of a project manager 101
The six Integration Management processes 102
Start your project with the Initiating processes 105
Integration Management and the process groups 106
The Develop Project Charter process 108
Make the case for your project 109
Use expert judgment and facilitation techniques to write your project charter 110
A closer look at the project charter 112
Two things you'll see over and over and over… 115
Plan your project! 118
The Project Management plan lets you plan ahead for problems 119
A quick look at all those subsidiary plans 121
Question Clinic: The "just-the-facts-ma'am" question 124
The Direct and Manage Project Work process 126
The project team creates deliverables 127
Executing the project includes repairing defects 128
Eventually, things WILL go wrong… 130
Sometimes you need to change your plans 131
Look for changes and deal with them 132
Make only the changes that are right for your project 133
Changes, defects, and corrections 134
Decide your changes in change control meetings 134
How the processes interact with one another 135
Control your changes; use change control 136
Preventing or correcting problems 138

5 XP (extreme programming)

Embracing change

Software teams are successful when they build great code. Even really good software teams with very talented developers run into problems with their code. When small code changes "bloom" into a series of **cascading hacks**, or everyday code commits lead to hours of fixing merge conflicts, work that *used to be satisfying* becomes **annoying, tedious, and frustrating**. And that's where **XP** comes in. XP is an agile methodology that's focused on building cohesive teams that **communicate** well, and creating a **relaxed, energized environment**. When teams build code that's **simple**, not complex, they can ***embrace change*** rather than fear it.

Out of the frying pan…	164
…and right back into the fire	165
It looks like we have a scope problem	169
The power of Scope Management	172
The six Scope Management processes	173
Plan your scoping processes	174
Collect requirements for your project	177
Talk to your stakeholders	178
Make decisions about requirements	179
Use a questionnaire to get requirements from a bigger group of people	182
A prototype shows users what your product will be like	183
Now you're ready to write a requirements document	184
Define the scope of the project	187
The project scope statement tells you what you have to do	190
Create the work breakdown structure	196
The inputs for the WBS come from other processes	197
Break it down by project or phase	199
Decompose deliverables into work packages	200
Inside the work package	206
The project scope baseline is a snapshot of the plan	208
The outputs of the Create WBS process	210
Why scope changes	213
The Control Scope process	215
Anatomy of a change	216
A closer look at the change control system	218

Lean / Kanban

6 Eliminating waste and managing flow

Agile teams know that they can always improve they way they work. They use the **Lean mindset** to find out where they are spending time on things that aren't helping them **deliver value**. Then they get rid of the **waste** that's slowing them down. Many teams with a lean mindset use **Kanban** to set **work in progress limits** and create **pull systems** to make sure that people are not getting sidetracked by work that doesn't amount to much. Get ready to learn how to seeing your software development process as a **whole system** can help you build better software!

7 Preparing for the pmi-acp® exam

Check your knowledge

Wow, you sure covered a lot of ground in the last six chapters!

You've delved into the values and principles of the agile manifesto and how they drive an agile mindset, explored how teams use Scrum to manage projects, discovered a higher level of engineering with XP, and seen how teams improve themselves using Lean/Kanban. Now it's time to **take a look back** and drill in some of the most important concepts that you learned. But there's **more to the PMI-ACP® exam** than just understanding agile tools, techniques, and concepts. To really ace the test, you'll need to explore how teams **use them in real-world situations**. So let's give your brain a *fresh look at agile concepts* with a ***complete* set of exercises, puzzles, and practice questions** (along with a little new material) specficially constructed to help prepare you for the PMI-ACP® exam.

8 Professional responsibility

Making good choices

It's not enough to just know your stuff. You need to make good choices to be good at your job. Everyone who has the PMI-ACP credential agrees to follow the **Project Management Institute Code of Ethics and Professional Conduct**, too, because the Code helps you with **ethical decisions**. There may be a few questions based on this material scattered through the PMI-ACP® exam. Luckily, most of what you need to know is **really straightforward**, and with a little review, you'll do great with this material.

Doing the right thing	694
Keep the cash?	696
Fly business class?	697
New software	698
Shortcuts	699
A good price or a clean river?	700
We're not all angels	701
Exam Questions	702

Practice makes perfect

9 Practice PMI-ACP® Exam

Bet you never thought you'd make it this far! It's been a long journey, but here you are, ready to review your knowledge and get ready for exam day. You've put a lot of new information about agile into your brain, and now it's time to see just how much of it stuck. That's why we put together this full-length, 120-question PMI-ACP® practice exam for you. **We followed the exact same PMI-ACP® Examination Content Outline** that the experts at PMI use, so it looks *just like the one you're going to see* when you take the real thing. Now's your time to flex your mental muscle. So take a deep breath, get ready, and let's get started.

Exam Questions 742
Exam Answers 786

Intro

In this section, we answer the burning question:
"So why DID they put that in a book about agile?"

Who is this book for?

If you can answer "yes" to any of these questions:

1. Are you a **developer**, **project manager**, **business analyst**, or other member of a team, and you're looking to improve your projects?
2. Is **your team going agile**, but you're not really sure what that means or how you fit in?
3. Are you thinking about a job search, and want to understand **why employers are asking for agile experience**?
4. Do you prefer **stimulating dinner-party conversation** to **dry**, **dull**, **academic lectures**?

this book is for you.

The PMI-ACP® (Agile Certified Practitioner) is one of the fastest-growing certifications in the world that employers are increasingly demanding.

Are you preparing for the PMI-ACP® exam?

*Then this book is **definitely** for you! We built this book to get the ideas, concepts, and practices of agile into your brain—and we made sure to include 100% coverage of every topic that appears on the exam. We included a lot of exam preparation material, including a full-length simulated exam that's **as close to the real thing** as you can get!*

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Are you **completely new** to working on any kind of teams or working with other people to achieve something?
2. Are you a "go-it-alone" loner who feels that working with other people on a team is always ***waste of time***?
3. Are you **afraid to try something different**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if agile concepts, tools, and ideas are anthropomorphized?

If you've never been on any kind of team before, then many of the ideas in agile will feel foreign. Just to be clear, we're not necessarily talking about a software team—experience on any kind of team will be just fine!

this book is not for you.

[Note from marketing: this book is for anyone with a pulse.]

We know what you're thinking.

"How can *this* be a serious project management book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

And we know what your *brain* is thinking.

Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows…

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, 10 days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never again snowboard in shorts.

And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."

We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

You could pound in a nail with a screwdriver, but a hammer is more fit for the job.

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

THERE HAVE BEEN REPORTS OF BEARS CAUSING PROBLEMS FOR PEOPLE AROUND HERE LATELY. BE CAREFUL OUT HERE.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner-party companion, or a lecture?

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

THIS IS AWFUL... THEY LOST FRANK'S LUGGAGE, AND IT GOT MY WALLET STOLEN!

Get—and keep—the reader's attention. We've all had the "I really want to learn this, but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?", and the feeling of "I rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering *doesn't*.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn about project management. And you probably don't want to spend a lot of time. And since you need to use this on a real project (and especially if you're going to take an exam on it!) you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to think that the material about agile is a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning.

Here's what WE did:

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 80 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

BULLET POINTS

We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

Fireside Chats

And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, and so on, because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

Cut this out and stick it on your refrigerator.

1. **Slow down. The more you understand, the less you have to memorize.**

   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**

   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There are No Dumb Questions"**

   That means all of them. They're not optional sidebars—***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**

   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Drink water. Lots of it.**

   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

6. **Talk about it. Out loud.**

   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

7. **Listen to your brain.**

   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something!**

   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Create something!**

   Apply this to your daily work; use what you are learning to make decisions on your projects. Just do something to get some experience beyond the exercises and activities in this book. All you need is a pencil and a problem to solve…a problem that might benefit from using the tools and techniques you're learning in this book.

Read me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. Once you've read this book, you'll definitely want to keep it on your shelf, so you can revisit useful ideas, tools, and techniques. But the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

The Brain Power exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. ***Don't skip the exercises.*** Even crossword puzzles are important—they'll help get concepts into your brain. But more importantly, they're good for giving your brain a chance to think about the words and terms you've been learning in a different context.

Try the exam questions—even if you're not studying for the exam!

Some of our readers are preparing for the three-hour, 120-question exam PMI-ACP® certification exam. Luckily, the most effective way to prepare for the exam is **to learn agile**. So even if you're not interested in the PMI-ACP® certification, this book is still for you. But you should still try the practice exam questions at the end of each chapter, because answering the exam questions is ***a really effective way*** to get agile concepts into your brain.

The technical review team

Lisa Kellner

Coming soon

Coming soon

Technical reviewers:

We're still working on this book! That's why it's an early release. We'll include pictures and bios for technical reviewers as soon as they're available.

And, as always, we were lucky to have **Lisa Kellner** return to our tech review team. Lisa was awesome, as usual. Thanks so much, guys!

Acknowledgments

Our editor:

We want to thank our editor, **Nan Barber**, for editing this book. Thanks!

The O'Reilly team:

There are so many people at O'Reilly we want to thank that we hope we don't forget anyone.

We'll definitely be thanking the production team in this space. In the meantime, here are a few people we definitely want to acknowledge.

And as always, we love **Mary Treseler**, and can't wait to work with her again! And a big shout out to our other friends and editors, **Mike Hendrickson**, **Laurie Petrycki**, **Tim O'Reilly**, **Andy Oram**, and **Sanders Kleinfeld**. And if you're reading this book right now, then you can thank the greatest publicity team in the industry: **Marsee Henon**, **Sara Peyton**, and the rest of the folks in Sebastopol.

Safari Books Online

Safari Books Online is an on-demand digital library that delivers expert in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of and pricing programs for organizations, government, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

1 what is agile?

Principles and practices

It's an exciting time to be agile! For the first time, our industry has found a real, sustainable way to solve problems that generations of software development teams have been struggling with. Agile teams use simple, straightforward **practices** that have been proven to work on real-life projects. But wait a minute... if agile is so great, why isn't everyone doing it? It turns out that in the real world, a practice that works really well for one team causes serious problems for another team, and the difference is the team **mindset**. So get ready to change the way you think about your projects!

The new features sound great...

Meet Kate. She's a project manager at a successful Silicon Valley startup. Her company builds software that's used by video and music streaming services and internet radio stations to analyze audiences in real time and choose programming suggestions that make their viewers or listeners happy. And now Kate's team has an opportunity to deliver something that will really help the company.

...but things don't always go as expected

Kate's discussion with the project team didn't go nearly as well as she'd hoped. What's she going to tell Ben?

Kate: So if we can get those new advanced audience analytics features into the next release, we'll all get a big bonus.

Mike: Well, that sounds like it would be great.

Kate: Fantastic! So we can count on you guys?

Mike: Hold on! Not so fast. I said it ***sounds*** like it ***would*** be great. But there's no way it's happening.

Kate: Wait, what?! Don't mess with me, Mike.

Mike: Look, if we'd known about this change four months ago when we were designing the audience data analysis service, this would be easy. But now we'd have to rip out a huge chunk of code and replace it with... well, I don't want to get into technical details.

Kate: Good. I don't want to hear them.

Mike: So... are we done here? Because my team's got a lot of work to do.

Agile to the rescue!

Kate's been reading about agile, and she thinks agile might help her get those features into the next release. Agile's gotten really popular with software teams because the ones that have "gone agile" often talk about the great results they get. The software they build is better, which makes a big difference to them *and* their users. Not only that, but when agile teams are effective, they have a much better time at work! Things are more relaxed, and the working environment is a lot more enjoyable.

So why has agile gotten so popular? Lots of reasons:

- When teams go agile, they find that it's a lot easier to meet their deadlines
- They also find that they can really cut down on bugs in their software
- Their code is a lot easier to maintain—adding, extending, or changing their codebase is no longer a headache
- The users are a lot happier, which always makes everyone's lives easier
- And best of all, when agile teams are effective, the team members' lives are better, because they can go home at a reasonable hour and rarely have to work weekends (which, for a lot of developers, is a first!)

A daily standup is a good starting point

One of the most common agile practices that teams adopt is called the **daily standup**, a meeting that happens every day, during which team members talk about what they're working on and their challenges. The meeting's kept short by making everyone stand for the duration. A lot of teams have ben successful adding a daily standup to their projects, and it's often the first step in going agile.

In a daily standup meeting, everyone on the team stands for the duration of the meeting. That keeps it short, sweet, and to the point.

But is this guy really paying attention to what his teammates are saying?

Kate tries to hold a daily standup

To Kate's surprise, not everyone on Mike's team is as excited about this new practice as she is. In fact, one of his developers is angry that she's even suggesting that they add a new meeting, and seems to feel insulted by the idea of attending a meeting every day where he's asked prying questions about his day-to-day work.

Kate thinks Mike and his team are being irrational, but maybe they have a point. What do you think?

BRAIN POWER

So what's going on here? Is Mike being irrational? Is Kate being too demanding? Why is this simple, well-accepted practice causing a conflict?

Different team members have different attitudes

Kate ran into problems adopting agile almost as soon as she started—and she's not alone.

The truth is that a lot of teams simply haven't had as much success with agile as they'd hoped they would. Did you know that well over half of companies that build software have experimented with agile? Despite the success stories—and there are many of them!—a lot of teams try agile, but end up with results that they're not particularly happy with. In fact, they feel a little ripped off! It seemed like agile came with a promise of big changes, but trying to get agile working on their own projects never seemed to work out.

And that's what's happening to Kate. She created a plan all on her own, and now she wants to get status updates from her team. So she's started dragging a reluctant team to her daily standup meeting. She's able to get them in the room. But will it really make a difference? She's worried about how people are deviating from her plan, so she'll concentrate on getting a status update from each person. Mike and his developers, on the other hand, want the meeting to end as quickly as possible so they can get back to "real" work.

In Kate's less-than-effective daily standup, each person tunes out everyone else while they're waiting to give their updates, then says as little as possible when it's time to speak. She still gets some useful information, but at the cost of conflict and boredom.

Two people can see the same practice very differently. If they don't both feel like they're getting something out of it, it can make the practice much less effective.

THAT'S JUST HOW SOFTWARE PROJECTS ARE, RIGHT? THINGS THAT WORK IN TEXTBOOKS DON'T REALLY WORK OUT IN REAL LIFE. THERE'S NOTHING WE CAN DO ABOUT IT, RIGHT?

No! The right mindset makes practices more effective.

Let's be clear: the way Kate is running her standups is how many daily standups are run. And while it's not optimal, a daily standup that's run this way *will still produce results*. Kate will find out about problems with her plan, and Mike's team will benefit in the long run because those problems that do affect them can be taken care of sooner rather than later. The whole thing doesn't take much time every day, and that makes it worth doing.

But there's a big difference between an agile team that goes through the motions and one that gets great results. The key to that difference is the **mindset** the team brings to each project. Believe it or not, the attitude that each person takes towards a practice can make it much more effective!

The attitude that each team member brings to a practice like the daily standup makes a huge difference in how effective it is. But even when everyone tunes out, the meeting is still effective enough to make it worth doing, even if it's boring.

A better mindset makes the practice work better

What would happen if Kate and Mike had a different mindset? What if each person on the team approached the daily standup *with an entirely different attitude*?

For example, what would happen if Kate felt like everyone on the team ***worked together*** to plan the project? Then she would genuinely listen to every single developer. If Kate changes her attitude towards the standup, she stop'll trying to figure out how they've deviated from *her* plan so that *she* can correct them. The focus of the meeting changes for her: now it's about understanding the plan that everyone on the team worked together to create, and her job is about helping the whole team do their work more effectivly.

That's a very different way of thinking about planning, one that Kate was never taught in any of her project management training courses. She was always taught that it was her job to come up with a project plan and basically dictate it to the team. She had tools that let her measure how well the team followed her plan, and strict processes that she would enforce to make changes to it.

Now things are totally different for her. She realized that the only way she could make the daily standup work is if **she puts effort into working with the team** to figure out the best approach to the project. Then the daily standup turns into a way for the whole team to work together to make sure everyone is making solid decisions, and that the project is on track.

Kate used to get really frustrated when she discovered changes to her project plan, because it was usually too late for the team to effectively change course.

Now that the daily standup is in place, the whole team works with her every day to find those changes, so they can make them much earlier. That's a lot more effective!

And what if Mike felt like this meeting wasn't just about giving status updates, but about *understanding how the project is going*, and coming together every day to find ways that everyone can work better? Then the daily standup becomes important to him.

A good developer almost always has opinions not just about his own code, but about the whole direction of the project. The daily standup becomes his way to make sure that the project is run in a sensible, efficient way—and Mike knows that in the long run that will make his job of coding more rewarding for his team, because the rest of the project is being run well. And he knows that when he brings up a problem with the plan during the meeting, ***everyone will listen***, and the project will run better because of it.

So what is agile, anyway?

Agile is a set of **methods and methodologies** that are optimized to help with specific problems that software teams run into, and kept simple so they're relatively straightforward to implement.

These methods and methodologies address all of the areas of traditional software engineering, including project management, software design and architecture, and process improvement. Each of those methods and methodologies consists of **practices** that are streamlined and optimized to make them as easy as possible to adopt.

I SPENT ALL THIS TIME COMING UP WITH **MY** PLAN, BUT THE TEAM KEEPS DEVIATING FROM IT. I CAN USE THE DAILY STANDUP TO MAKE SURE THEY DO **EVERYTHING** I TELL THEM TO DO.

Mindset versus methodology

Agile is *also* a **mindset**, and that's a new idea for a lot of people who haven't worked with agile before. It turns out that each team member's attitude towards the practices they use *can make a huge difference* in how effective those practices are. The agile mindset is focused on helping people share information with each other, which makes it much easier for them to make important project decisions (rather than just relying on a boss or project manager to make those decisions). It's about opening up planning, design, and process improvement to the *entire* team. To help everyone get into an effective mindset, each agile methodology has its own set of **values** that team members can use as a guide.

IF WE **ALL WORK TOGETHER** TO PLAN OUR PROJECT, THEN WE CAN USE THE DAILY STANDUP TO MAKE COURSE CORRECTIONS ALONG THE WAY.

BRAIN POWER

What happens if one team member checks out during the daily standup and doesn't really listen to his or her teammates?

Sharpen your pencil

Here are a few problems that Kate, Ben, and Mike brought up during a daily standup. Across from them, we've written down the names of a few different practices that you'll often see used on agile teams. Don't worry if you haven't run across some of them—you'll learn a lot more about them later in the book, so we included a brief description of each practice to help you out. See if you can match each problem with a practice that might help fix it.

"WE JUST WASTED HOURS GRINDING THROUGH SPAGHETTI CODE TO FIND THAT BUG!"

A **retrospective** is a meeting in which everyone talks about how the last part of the project went, and talks about what lessons can be learned.

"OKAY, WE'VE GONE OVER THE USER STORIES. NOW LET'S FIGURE OUT HOW THEY FIT TOGETHER SO WE CAN PLAN OUT THE NEXT FEW WEEKS OF WORK."

A **user story** is a way to express one very specific need that a user has, usually written out as a few sentences on a sticky note or index card.

"WE ALWAYS SEEM TO RUN INTO THE SAME KIND OF PROBLEMS OVER AND OVER AGAIN WITH EVERY RELEASE."

A **task board** is an agile planning tool in which user stories are attached to a board and categorized into columns based on their status.

"I JUST DEMOED A NEW FEATURE TO ONE OF OUR VIDEO STREAMING USERS. SHE SAID IT WON'T ACTUALLY FIX THE PROBLEM IT'S SUPPOSED TO ADDRESS FOR HER."

A **burndown chart** is a line chart, updated daily, that tracks the amount of work left on the project, "burning" down to zero when the work is done.

"I THOUGHT WE'D BE DONE UPDATING THE SONG DATABASE CODE BY FRIDAY. NOW YOU'RE TELLING ME THAT IT'LL BE THREE MORE WEEKS?"

Developers fix code problems by constantly **refactoring** their code, or improving the code structure without changing its behavior.

It's okay if you haven't run across these practices yet. You'll learn more about them in the next few chapters.

Scrum is the most common agile methodology

There are many ways that teams can be agile, and there's a long list of methods and methodologies that agile teams use. But there have been many surveys done over the years that have found that the most common approach to agile is **Scrum**, a methodology focused on project management.

When a team uses Scrum, every project follows the same pattern. There are three main roles on a Scrum project: the **Product Owner** (like Ben) works with the team to maintain a **product backlog**, the **Scrum Master** helps guide the team past roadblocks, and the **team members** (everyone else on the team). The project's divided into **sprints**, or cycles of equal length (often 30 days) that follow the Scrum pattern. At the start of a sprint, the team does **sprint planning** to determine which features from the product backlog they'll build. This is called the **sprint backlog**, and the team works throughout the sprint to build all of the features in it. Every day the team holds a short meeting called the **Daily Scrum**. At the end of the sprint, working software is demonstrated to the product owner and stakeholders in the **sprint review**, and the team holds a **retrospective** to figure out lessons they've learned.

We'll cover Scrum in depth in Chapters 3 and 4, and we'll not only teach you how it helps teams build better software and run more successful projects, but we'll also use it to explore important concepts and ideas shared by all agile teams.

XP and Lean/Kanban

While Scrum is the most popular agile methodology, many teams take other approaches. The next most popular methodology is **XP**, a methodology focused on software development and programming that's often used in combination with Scrum. Other teams approach agile with **Lean** and **Kanban**, a mindset that gives you tools to understand the way you build software today and a method that helps you evolve to a better state tomorrow. You'll learn about XP and Lean/Kanban in Chapters 5 and 6.

Getting a little overwhelmed with new vocabulary?

We'll highlight new vocabulary words in **boldface** when we first introduce them. We did that a lot on this page—and if there are a few that you're not familiar with, that's okay! Seeing new ideas in context now will help them sink into your brain when you learn about them in detail later. That's part of the Head First neuroscience that makes this book brain-friendly!

Sharpen your pencil Solution

Teams can avoid making the same mistakes over and over again by looking back at the project and talking about what went right and what could be improved.

"ONE OF MY PROGRAMMERS JUST WASTED HOURS GRINDING THROUGH SPAGHETTI CODE TO FIND A BUG!"

A task board is a great way for everyone on the team to see the same big-picture view of the project.

"OKAY, WE'VE GONE OVER THE USER STORIES. NOW LET'S FIGURE OUT HOW THEY FIT TOGETHER SO WE CAN PLAN OUT THE NEXT FEW WEEKS OF WORK."

"WE ALWAYS SEEM TO RUN INTO THE SAME KIND OF PROBLEMS OVER AND OVER AGAIN WITH EVERY RELEASE."

"I JUST DEMOED A NEW FEATURE TO ONE OF OUR VIDEO STREAMING USERS. SHE SAID IT WON'T ACTUALLY FIX THE PROBLEM IT'S SUPPOSED TO ADDRESS FOR HER."

When everyone on the team understands the users and what they need, they do a better job of building software that users love.

"I THOUGHT WE'D BE DONE UPDATING THE SONG DATABASE CODE BY FRIDAY. NOW YOU'RE TELLING ME THAT IT'LL BE THREE MORE WEEKS?"

A **retrospective** is a meeting in which everyone talks about how the last part of the project went, and talks about what lessons can be learned.

A **user story** is a way to express one very specific need that a user has, usually written out as a few sentences on a sticky note or index card.

A **task board** is an agile planning tool in which user stories are attached to a board and categorized into columns based on their status.

A **burndown chart** is a line chart, updated daily, that tracks the amount of work left on the project, "burning" down to zero when the work is done.

This is an XP practice. Some project managers are surprised the first time they discover agile practices that are focused on code, and not just on planning and executing the project.

Developers fix code problems by constantly **refactoring** their code, or improving the code structure without changing its behavior.

there are no Dumb Questions

Q: It sounds like Scrum, XP, and Lean/Kanban are very different from each other. How can they all be agile?

A: Scrum, XP, and Lean/Kanban focus on very different areas. Scrum is mainly focused on project management: what work is getting done, and making sure that it's in line with what the users and stakeholders need. XP is focused on software development: building high-quality code that's well-designed and easy to maintain. Lean/Kanban is a combination of the Lean mindset and the Kanban method, and teams use it to focus on continually improving the way that they build software.

In other words, Scrum, XP, and Lean/Kanban are focused on three different areas of software engineering: project management, design and architecture, and process improvement. So it makes sense that they would have different practices—that's how they differ.

In the next chapter you'll start learn about what they have in common: **shared values and principles** that help teams adopt an agile mindset.

Q: Isn't this all just stuff I know already, only with a new name? Like, Scrum sprints are really just milestones and project phases, right?

A: When you come across an agile methodology like Scrum for the first time, it's really common to look for the parts of it that are similar to things you already know—and that's a good thing! If you've been working on teams for a while, then a lot of agile *should* feel familiar. Your team builds something, and you and your teammates are almost certainly doing a lot of things well that you don't want to change (yet!).

However, it's very easy to fall into the trap of thinking that a familiar-seeming part of agile is exactly the same thing as something you already know. For example, Scrum sprints are *not the same thing as project phases*. There are many differences between phases or milestones in traditional project management and sprints in Scrum.

For example, in a typical project plan, all of the project phases are planned at the beginning of the project; in Scrum, only the next sprint is planned in detail. This difference can feel very strange to a team used to traditional project management.

You'll learn a lot more about how Scrum planning works, and how it may be different from what you're used to. In the meantime, keep an open mind—and try to catch yourself when you have the thought, "This is just like something I already know!"

BULLET POINTS

- Many teams that want to go agile start with the **daily standup**, a meeting with the whole team where everyone stands in order to keep it short
- Agile is a set of **methods and methodologies**, but it's *also* a **mindset**, or an attitude that's shared by everyone on the team
- The daily standup meeting is much more effective when everyone on the team has the right **mindset**—everyone listens to each other, and they all work together to make sure the project is on track
- Every agile methodology comes with a set of **values** to help the team get into the most effective mindset
- When team members follow shared **principles** and share the same set of **values**, it can make the methodology they follow <u>*much more effective*</u>
- **Scrum**, a methodology focused on project management, is the most common agile methodology
- In a Scrum project, the team breaks the work into **sprints**, or cycles of equal length (often 30 days) that follow the Scrum pattern
- Every sprint starts with a **sprint planning** session to determine what they'll build
- During the sprint the team works on the project, and every day they hold short meeting called a **daily scrum**
- At the end of the sprint, the team holds a **sprint review** with the stakeholders to demo the working software that they built
- To finish the sprint, the team holds a **retrospective** to look back at how the sprint went and discuss ways that they can improve together as a team

Watch it!

Don't just dismiss the idea that mindset matters!

A lot of people—especially hardcore developers—tune out as soon as they start hearing words like mindset, values, and principles. That's especially true of coders who have a habit of locking themselves in a room and never talking to anyone. If you're starting to think this way, really try to give these ideas the benefit of the doubt. After all, a lot of great software has been built this way, so there has to be something to it... right?

Sharpen your pencil

Which of these scenarios are examples of applying practices, and which are examples of applying principles? Don't worry if you haven't seen some of these practices yet, just use the context to try to figure out the right answer. (That's a good skill to work on for taking a certification exam!)

1. Kate knows that the most effective way to communicate important information about the project to her team is with a **face-to-face conversation**.

☐ Principle ☐ Practice

2. Mike and his team know that the users will probably change their minds later, and those changes can wreak havoc with their code, so they use **incremental design** to make sure the code that they build is easy to change later.

☐ Principle ☐ Practice

3. Ben uses a **persona** to model a typical user because knows that the more the team understands the users, the better job they'll do of building software.

☐ Principle ☐ Practice

4. Mike always makes sure that his team is working on something that he can demonstrate to Kate and Ben, because he knows that **working software** is the best way to show the team's progress.

☐ Principle ☐ Practice

5. Kate wants to improve the way that the team builds software, so she gets them all to **improve collaboratively and evolve experimentally** by coming up with changes that they can make to their process together, and using data to figure out if those changes made things better.

☐ Principle ☐ Practice

6. Mike and his team **embrace change** by building code that's easy to change down the road.

☐ Principle ☐ Practice

Answers on page 20

Kate: This project's gone so much better than ones in the past. And all because of one little meeting every day!

Mike: Well, I wouldn't say that.

Kate: Come on, Mike! Don't be such a pessimist.

Mike: No, seriously. Look, you didn't really think you're the first person to try to solve our project problems by adding meetings, did you?

Kate: Well, I... um...

Mike: We got some really good results, so I'll be honest with you here. When you started holding those daily standups, almost everyone on the team was unhappy.

Kate: Really?

Mike: Yeah. Don't you remember how for the first week and a half, most of us just stared at our phones the whole time?

Kate: Well, sure. I guess that wasn't particularly useful. If I'm honest with myself, I was actually thinking about calling the whole thing off.

Mike: But then one of my coders brought up that serious architecture issue. Everyone listened because she's really good, and they all respect her opinion.

Kate: Right. We had to make a major change, and I cut two of the features out of the release to make room for it.

Mike: Yes! That was really important. Normally when we run into a problem like that, we have to work late nights to deal with the aftermath. Like when we found out that the listener feedback analysis algorithm had a serious flaw in it.

Kate: Ugh, that was awful. I usually find out about problems like that after we've all promised things we couldn't deliver. This time we caught the problem early, and I could work with Ben to manage our users' expectations and get you guys the time you needed to come up with a new approach.

Mike: We'll definitely bring up problems like that every time they come up.

Kate: Wait—what? That kind of problem happens a lot?!

Mike: Are you kidding? I've never had a project that didn't run into at least one nasty surprise like that once we started coding. That's how software projects work in the real world, Kate.

Looks like Kate discovered that software projects are a lot less clean and simple in real life than they are on paper. Before, she could just build her plan and then force the team to work it... and when things went wrong, it was their fault, not hers.

On the other hand, this project went a lot better than her last ones. She had to work a lot harder to deal with problems, but she got better results!

Agilecross

Solve this crossword and get these agile ideas to stick in your brain!
How many words you can get without looking back at the chapter?

Across

1. A daily _____ can be valuable, but it really works best if everyone on the team has the right mindset
3. In the Kanban method, teams improve collaboratively and _____ experimentally
5. Kanban is an agile method focused on _____ improvement
7. Holds the features that haven't been built yet
9. The Scrum team always demos _____ software
10. Who the Scrum team does the demo for
11. What Scrum teams do every day
16. Helps the team understand their users' needs
17. The Scrum _____ guides the team past roadblocks and helps them implement Scrum
19. Agile planning tool
20. The _____ Owner on a Scrum team maintains a backlog
21. The most effective way to communicate
22. _____ design helps XP teams make code easy to change
24. Scrum teams get together to do this at the start of the project
25. Scrum team's demo at the end of the project

Down

1. How Scrum teams divide up their projects chronologically
2. Helps the team understand who their users are
4. These help teams understand the mindset of a methodology
6. Methodology focused on project management
8. Makes a big difference when adopting practices
12. When the team gets together to figure out what lessons they've learned
13. Chart that tracks the amount of work left on a project
14. What XP teams do constantly to improve their code structure
15. What XP teams do with change
18. A tool or technique used by a team
23. Methodology focused on code and software design

The PMI-ACP certification can help you be more agile

The Agile Certified Practitioner (PMI-ACP)® certification was created by the Project Management Institute to meet the needs of project managers who have increasingly found themselves working with agile methods, methodologies, practices, and techniques. And just like with the PMP certification, PMI has constructed an exam based on real-world tasks, tools, and practices used by agile teams every day.

The PMI-ACP certification is for anyone who works on agile teams, or in an organization that's moving towards adopting agile.

The exam doctor is here to help you get in the best shape possible for taking the exam.

IF YOU'RE PLANNING ON TAKING THE EXAM, WE'LL HELP YOU PREPARE FOR IT BY INCLUDING PRACTICE EXAM QUESTIONS TO REVIEW THE MATERIAL THAT YOU LEARNED IN EACH CHAPTER.

The exam is focused on real-world agile knowledge.

The PMI-ACP exam is designed to reflect the way teams work in the real world. It covers the most common methods and methodologies, including Scrum, XP, and Lean/Kanban. The exam questions are based on knowledge and real-world tasks that teams use every day.

That's why this book is, first and foremost, **built to teach you agile**: because understanding agile methods, methodologies, practices, values, and ideas is the most effective way to prepare for the PMI-ACP certification.

In addition to teaching you all about agile, we will also spend some time focusing specifically on exam material. This book has **100% coverage of the PMI-ACP exam content**, and includes many practice questions, test-taking tips, and exam preparation exercises, including a complete, full-length practice exam that mimics the real thing.

AND EVEN IF YOU AREN'T USING THIS BOOK TO PREPARE FOR THE PMI-ACP CERTIFICATION, THE PRACTICE QUESTIONS WILL GIVE YOU A DIFFERENT WAY OF APPROACHING THE MATERIAL. THAT'S A GREAT WAY TO **SEAL IT INTO YOUR BRAIN!**

Chapters 2 through 7 each end with a set of practice questions. They also include "Question Clinic" sections that break down different types of questions that you'll run across on the exam.

Learning to recognize different kinds of questions that you'll see on the exam is really useful because when you see something familiar it helps your brain relax, which can help the answer come more quickly. We call this one the **"Just the facts, ma'am" question**. It looks like it's just asking for some basic information, but *read all of the answers carefully*! There's often a misleading answer that looks like it might be right, but isn't.

39. Which of the following is used by teams to understand the progress of their project?

A. Refactoring

Some answers will clearly be wrong. Refactoring is about improving code, not understanding project progress.

B. Retrospective

Some answers can be misleading! The retrospective helps the team understand their project better, but it doesn't keep track of progress because it mainly looks back at work that's been done.

C. Burndown chart

Here's the right answer! The burndown chart is a tool that shows how the project has been doing, and how much work is left..

D. Continuous integration

You haven't seen this term yet—it's a practice that XP teams use. Some exam questions will have answers you don't recognize. And that's okay! Just relax, and concentrate on the other answers. In this case, one of the others is correct. But if none of them are, then you can eliminate them and take an educated guess!.

...And we'll help you prepare for the PMI-ACP exam with "Question Clinic" sections that focus on different kinds of exam questions, and give you practice writing questions yourself!

Sharpen your pencil Solution

This is one of the principles behind agile: that a face-to-face conversation is the most effective way to convey information to and within a software team.

1. Kate knows that the most effective way to communicate important information about the project to her team is with a **face-to-face conversation**.

☒ Principle ☐ Practice

Incremental design is an XP practice in which the team grows the codebase incrementally over time.

2. Mike and his team know that the users will probably change their minds later, and those changes can wreak havoc with their code, so they use **incremental design** to make sure the code that they build is easy to change later.

☐ Principle ☒ Practice

A persona is a practice in which the team creates an fictional user with a name (and often a fake photo) to better understand who will be using their software.

3. Ben uses a **persona** to model a typical user because knows that the more the team understands the users, the better job they'll do of building software.

☐ Principle ☒ Practice

An important agile principle is that working software is the primary measure of progress in the project, because it's the most effective way for everyone to gauge exactly what the team has accomplished.

4. Mike always makes sure that his team is working on something that he can demonstrate to Kate and Ben, because he knows that **working software** is the best way to show the team's progress.

☒ Principle ☐ Practice

5. Kate wants to improve the way that the team builds software, so she gets them all to **improve collaboratively and evolve experimentally** by coming up with changes that they can make to their process together, and using data to figure out if those changes made things better.

☐ Principle ☒ Practice

This is one of the core practices of Kanban. The team uses the scientific method to see if their improvements actually work in practice.

6. Mike and his team **embrace change** by building code that's easy to change down the road.

☒ Principle ☐ Practice

An important value shared by all effective XP teams is that they embrace change, rather than try to prevent or resist it.

1 S	T	A	N	D	U	2 P																		
P						E											3 E	4 V	O	L	V	E		
R					5 P	R	O	C	E	S	S				6 S		A							
I						S							7 B	A	C	K	L	O	G					
N			8 M		9 W	O	R	K	I	N	G				R		U							
T			I			N									10 U	S	E	R	S					
			N		11 D	A	I	L	Y	S	C	12 R	U	M			S							
			D									E												
			S		13 B		14 R					T												
15 E			E		16 U	S	E	R	S	T	O	R	Y											
17 M	A	S	T	E	R		F					O							18 P					
B					N		A			19 T	A	S	K	B	O	A	R	D						
R		20 P	R	O	D	U	C	T				P							A					
A					O		T		21 F	A	C	E	T	O	F	A	C	E						
C					W		O					C							T					
E				22 I	N	C	R	E	M	E	N	T	A	L					I					
			23 X				I					I							C					
			24 P	L	A	N	N	I	N	G		V							E					
							G				25 R	E	V	I	E	W								

HEY, CHECK IT OUT! THE "SHARPEN YOUR PENCIL" SOLUTION ON THE PREVIOUS PAGE HAS NOTES WITH ***USEFUL EXPLANATIONS*** ADDED TO IT! I BET THAT'LL BE A GOOD LEARNING TOOL.

2 agile values and principles

Mindset meets method

There's no "perfect" recipe for building great software.

Some teams have had a lot of success and seen big improvements after adopting agile practices, methods, and methodologies, while others have struggled. We've learned that the difference is the mindset that the people on the team have. So what do you do if you want to get those great agile results for your own team? How do you make sure your team has the right mindset? That's where the **Agile Manifesto** comes in. When you and your team get your head around its **values and principles**, you start to think differently about the agile practices and how they work, and they start to become *a lot more effective*.

Something big happened in Snowbird

In the 1990s there was a growing movement throughout the software development world. Teams were growing tired of the traditional way of building software using a **waterfall process**, where the team first defines strict requirements, then draws up a complete design, and builds out all of the software architecture on paper before the code is written.

By the end of the decade, there was a growing consensus that teams needed a more "lightweight" way to build software, and several methodologies—especially Scrum and XP—were gaining popularity as the way to accomplish that.

People on waterfall teams weren't always 100% clear on exactly why they didn't like their process, but there was a lot of agreement that it was somehow too "heavyweight" and cumbersome.

Meeting of the minds

In 2001, a group of seventeen open-minded people got together at the Snowbird ski resort in the mountains outside of Salt Lake City, Utah. The group included thought leaders from all across the new "lightweight" world, including the creators of Scrum and XP. They weren't sure exactly what would come out of the meeting, but there was a strong sense that these new lightweight methods for building software had something in common. They wanted to figure out if they were right, and maybe find a way to write it down.

Leaders from across the industry got together to figure out if there was anything in common between the various and increasingly popular lightweight methods for building software.

The Agile Manifesto

It didn't take long for the group to converge on four values that they all had in common. They wrote those values down in what would come to be known as the **Agile Manifesto**.

Manifesto for Agile Software Development

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.

IF THOSE GUYS WERE SO SMART, WHY DIDN'T THEY JUST COME UP WITH THE ***BEST WAY TO BUILD SOFTWARE***? WHY BOTHER WITH THESE "VALUES" AT ALL?

The idea that there's no "silver bullet" method for building software was first introduced in the 1980's by pioneering software engineer Fred Brooks, in an essay called "No Silver Bullet."

They weren't trying to come up with a "unified" methodology.

One of the fundamental ideas in modern software engineering is that **there's no single "best" way to build software**. That's an important idea that's been around in software engineering for decades. The Agile Manifesto is effective because it ***lays out values that help teams get into an agile mindset***. When everyone on the team genuinely incorporates these values into the way they think, it actually helps them build better software.

Adding practices in the real world can be a challenge

Teams are always looking for ways to improve. We've seen how practices can help. That's especially true of the lightweight practices used by agile teams, which are designed to be simple, straightforward, and easy to adopt. But we've also seen that the team's mindset or attitude can make it much more difficult to adopt them successfully—like when Kate found that the attitude that she, Mike, and the rest of the team had made a big difference when she tried to start holding a daily standup meeting.

Mike and Kate discovered that it's not enough to have a "best" or "correct" practice. If the people don't buy into it, it'll cause conflict and eventually be thrown out.

The four values of the Agile Manifesto guide the team to a better, more effective mindset

The Agile Manifesto contains four "X over y" lines that help us understand what agile teams value. Each of those lines tells us something specific about the values that drive an agile mindset. We can use them to help us understand what it means for a team to be agile.

Let's take a closer look each of those four values →

Individuals and interactions over processes and tools

Agile teams recognize that processes and tools are important. You've already learned about a few practices that agile teams use: daily standups, user stories, task boards, burndown charts, refactoring, and retrospectives. These are all valuable tools that can make a real difference to an agile team.

But agile teams value individuals and interactions even more than processes and tools, because teams always work best when you pay attention to the human element.

You've already seen an example of this—when Kate tried to introduce daily standups, and ended up getting into a conflict with Mike and his development team. That's because a tool that works really well for one team can cause serious problems for another team if the people on the team aren't getting anything out of it, and if it's not directly helping them build software.

Good idea, Kate!

Processes and tools are important to getting the project done, and they can be really valuable. But the **individual people on the team** are even more important, and any tool that you introduce needs to improve their **interactions** with each other and with their users and stakeholders.

Working software over comprehensive documentation

What does "working" software mean? How do you know if your software works? That's actually a harder question to answer than you might think. A traditional waterfall team starts a project by building comprehensive requirements documents to determine what the team will build, reviews that documentation with the users and stakeholders, and then passes it on to the developers to build.

Most professional software developers have had that terrible meeting where the team proudly demonstrates the software they've been working on, only to have a user complain that it's missing an important feature, or that it doesn't work correctly. It often ends in an argument, like the one that Ben had with Mike after he gave a demo of a feature his team had been working on for a few months:

There's an old programmer joke that bugs are just undocumented features. A lot of bugs happen because the programmer thought the software should work one way, but the users expected it to work differently.

A lot of people try to fix this problem with comprehensive documentation, but that can actually make the situation worse. The problem with documentation is that two people can read the same page and come away with two very different interpretations.

That's why agile teams value **working software** over comprehensive documentation—because it turns out that the most effective way for a user to gauge how well the software works is to actually use it.

BRAIN BARBELL

This little puzzle should be familiar to anyone who read Head First PMP!

Lisa is testing the firmware component for the *Black Box 3000™*. The product is only "working" in one of these scenarios. Can you help Lisa figure out which version of the product has "working" firmware?

The Black Box 3000™

Here's a hint: there's an important piece of information that we haven't given you. Without it, this puzzle is really hard to solve.

Some might even say impossible!

Scenario 1

Lisa presses the button, but nothing happens.

SO... HOW DO I KNOW IF THE FIRMWARE IN THE BOX IS WORKING?

Scenario 2

Lisa presses the button and a voice comes out of the box that says, "You pressed the button incorrectly."

Lisa, our tester, is testing the Black Box 3000™ firmware, but she isn't sure what she's supposed to be testing for.

Scenario 3

Lisa presses the button and the box heats up to 628°F. Lisa drops the box and it shatters into hundreds of pieces.

Just in case you don't happen to know the term, **firmware** is software that's programmed into the read-only memory of a piece of hardware.

BRAIN BARBELL

EXPLANATION

Here's that crucial missing piece of information that we didn't give you on the previous page: the *Black Box 3000™* is the heating element from an industrial oven. So Scenario 3 is the scenario that demonstrates "working" software:

The best—and sometimes only!—way to tell if software is working is to put it in the hands of the people who need to use it. If they can use the software to do what they need it to do, it's working. But it's not always easy to know exactly what "working" means, which is why agile teams *also* value comprehensive documentation—they just value working software more.

In this case, literally in Lisa's hands. Let's hope that she was wearing heat-proof gloves!

Here's an example of comprehensive documentation that the team actually found useful: the specification for the Black Box 3000™.

BLACK BOX 3000™
Specification Manual

The BB3K™ is a heating element for an industrial oven.

BB3K™ must heat up to exactly 628°F in 0.8 seconds.

BB3K™ must have a large, easy-to-press button.

Sometimes documentation can be useful—like when it's not clear exactly what "working" software is supposed to do.

LOOKS LIKE SCENARIO #3 IS THE ONE THAT SHOWS WORKING SOFTWARE. GOOD THING I HAD SOME DOCUMENTATION TO TELL ME THAT... BUT IT'S EVEN MORE IMPORTANT TO ME THAT I HAVE THE ***ACTUAL PRODUCT*** IN MY HANDS.

Now that she knows what "working" means for this software, Lisa can actually test it.

Customer collaboration over contract negotiation

No, this *isn't* about consultants or procurement teams who have to deal with contracts!

When people on agile teams talk about contract negotiation, they often mean an attitude that people take towards their users, customers, or people on other teams. When people on a team have a "contract negotiation" mindset, they feel like they have to come to a strict agreement on what the team will build or do before any work can start. Many companies encourage this mindset, asking teams to provide explicit "agreements" (often documented in specifications, and enforced with strict change control procedures) about what it is they will deliver and when.

Contract negotiation is necessary in cases where the customers are unwilling to collaborate. It's very difficult to genuinely collaborate with someone who's being unreasonable—like a customer who routinely changes the scope of the project, but refuses to give the team enough time to make those changes.

Agile teams value customer collaboration over contract negotiation. They recognize that projects change, and that people never have perfect information when starting a project. So instead of trying to nail down exactly what's going to be built before they start, they **collaborate with their users** to try to get the best results.

Scrum teams are especially good at this because they have a **product owner** like Ben who is a true member of the team. He might not have been developing code, but he worked hard on the project by talking to users, understanding what they needed, and working with the rest of the team to help them understand those needs and build working software.

Responding to change over following a plan

Some project managers have a saying: "Plan the work, work the plan." And agile teams recognize that planning is important. But working a plan that has problems will cause the team to build a product with problems.

Traditional waterfall projects have ways of handling changes, but they usually involve strict and time-consuming change control procedures. This reflects a mindset in which changes are the exception, not the rule.

The problem with plans is that they're built at the start of projects, and that's when the team knows the least about the product they're going to build. So agile teams **expect that their plans will change**.

That's why they typically use methodologies that have tools to help them constantly look for changes and respond to them. You've already seen a tool like that: the daily standup.

It's important to plan your project, but it's even more important to recognize that those plans will change once the team starts working on the code.

Once Kate and Mike sorted out their differences, they both realized that the daily standup was a way for everyone to look at the plan every day, and work together to respond to any changes that were needed. When everyone on the team worked together to respond to change, they were able to update the plan that they all built together without introducing chaos into the project.

Responding to change is important to agile teams, but they still value following a plan.

Take another look at the last line of the Agile Manifesto:

> That is, while there is value in the items on the right, we value the items on the left more.

Each of the four values in the Agile Manifesto contains two parts: something (on the right-hand side) that agile teams value, and then something else (on the left-hand side) that agile teams value more.

So when agile teams say they value responding to change over following a plan, that **doesn't** *mean that they don't value planning—in fact, it means the opposite! They absolutely value following a plan. They just value responding to change more.*

In fact, Scrum teams actually do more planning than traditional teams following a waterfall process! But since they're really good at responding to change, it doesn't feel like it to the people on the team.

BULLET POINTS

- The **Agile Manifesto** was created in 2001 by people who came together to find common ground between different "lightweight" methods, methodologies, and approaches to building software
- The Agile Manifesto has **four values** that help agile teams get into the right mindset
- Agile teams **value processes and tools** because they help the team get organized and work effectively
- But they **value people and interactions *more*** because teams work best when you pay attention to the human element
- Agile teams **value comprehensive documentation** because it's an effective way to communicate complex requirements and ideas
- But they **value working software *more*** because it's the most effective way to communicate progress and get feedback from users
- Agile teams **value contract negotiation** because sometimes it's the only way to work effectively in an office culture where mistakes are punished
- But they **value customer collaboration more** because it is much more effective for building software than having a legalistic or antagonistic customer relationship
- Agile teams **value following a plan** because without planning, complex software projects go off the rails
- But they **value responding to change more** because the team that works the wrong plan ends up building the wrong software

Manifesto Magnets

Oops! You had the Agile Manifesto recreated perfectly with refrigerator magnets! But someone slammed the door and they all fell off. Can you recreate the whole thing? See how much of it you can do without flipping back and looking for hints.

Some of the magnets stayed on the fridge. Leave these magnets in place.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

:

Through this work we have come to

over

over

over

over

Don't worry about the order of the values.

The four values in the Agile Manifesto are equally important, so there's no particular order for them. Just make sure that each specific value (X over Y) is matched up.

All of the other magnets fell off the fridge. Can you put them back in the right place?

That is, while there is value in the items on the , we value the items on the more .

following individuals change left customer value tools plan contract processes working and and to comprehensive negotiation right collaboration a documentation software interactions responding

Answers on page 44

there are no Dumb Questions

Q: I'm still not clear on what "waterfall" means.

A: "Waterfall" is the name given to a specific way that software companies have traditionally built software. They divide their projects into phases, usually drawn in a diagram that looks like this:

It was given the name "waterfall" by the software engineering researcher in the 1970s who first described it as a less-than-effective way to build software. The team is often expected to come up with a near-perfect requirements document and design before they start building code, because it takes a lot of time and effort to go back and fix the requirements and design when the team finds problems with them.

The problem is that there's often no way to know if the requirements and design are right until the team starts building code. It's very common for a waterfall team to think they got everything right in the documentation, only to discover serious flaws when the developers start implementing the design.

Q: Then why would anyone ever use a waterfall process?

A: Because it works—or, at least, it *can* work. There have been plenty of teams that used a waterfall process and built great software. It's certainly possible to create the requirements and design first so that there are relatively few changes.

More importantly, there are a lot of companies where the culture really lends itself to a waterfall process. For example, if you work for a boss who will severely punish you if he thinks you made a mistake, then having him personally approve fully fleshed-out requirements and design documents before any code is written can help you keep your job. But no matter how you do it, the effort it takes to figure out who's accountable for each decision in the project takes away from effort that could be spent actually building your product.

Q: So is waterfall good or bad? And how is it less "lightweight" than an agile methodology?

A: Waterfall isn't "good" or "bad," it's just a certain way of doing things. Like any tool, it has its strengths and weaknesses.

However, many teams find a lot more success with agile methodologies like Scrum than they do with a waterfall process. One reason is that they find a waterfall process too "heavyweight" because it imposes a lot of restrictions on how they work: they're required to go through complete requirements and design phases before any code is written. In the next chapter, you'll learn how Scrum teams use their sprints and planning practices to start building the software quickly, which lets them get working software into the hands of their users. This feels a lot more "lightweight" to the team because everything they're doing has an immediate effect on the code that gets built.

Q: Then exactly how should I run my projects? Should my team create documentation or not? Do we work from a complete specification? Should we throw out documentation altogether?

A: Documentation is important to agile teams—but mainly because it can be an effective way to build working software. Documentation is only useful if people read it, and the truth of the matter is that a lot of people just don't read documentation.

When I write a specification requirements document and give it to you and your team to build, the document isn't important. The important thing is that **what's in my head matches what's in your head**, and what's in each of the team members' heads. In some cases, like when there are complex calculations or workflows, *a document can be a really effective* way to the **shared understanding** that leads to great software.

Q: I'm just not sold on this idea that values are important. How do they help me and my team actually write code?

A: The values in the agile manifesto help you and your team get into a mindset that helps you build better software. And you've already learned that your mindset—the attitude that you have towards the practices that you use—can make a big difference.

Think about the example in the last chapter, where Kate and Mike were having trouble with the daily standup meeting. Kate only got mediocre results when she used it as a to dictate her plan to the team and demand status from them. But when everyone had a more collaborative attitude, they got much better results. That's how mindset can have a big effect on real-world results.

Question Clinic: The "which-is-BEST" question

A great way to prepare for the exam is to learn about the different kinds of questions, and then try writing your own. Each of these Question Clinics will look at a different type of question, and give you practice writing one yourself. Even if you're not using this book to prepare for the PMI-ACP certification exam, give this a shot. It can still be a great way to help get these concepts into your brain!

Take a little time out of the chapter for this Question Clinic. It's here to let your brain have a break and think about something different.

A LOT OF EXAM QUESTIONS ASK YOU TO CHOOSE WHICH OF THE ANSWER IS **BEST**. THAT USUALLY MEANS ONE ANSWER IS *PRETTY GOOD*, BUT THERE'S ANOTHER ANSWER THAT'S **BETTER**.

It's not a terrible idea to get senior management involved, but that really ought to be left for resolving serious conflicts. It's better for the team to work with the user and figure things out together, without appealing to authority..

82. A user asks the team for a new feature that's very important, but doing the work will make them miss a committed deadline for a different feature that they plan to work on. Which is the BEST thing the team should do first?

A. Call a meeting with senior management to get an official decision on relative priority

B. Follow the existing plan so they don't miss the deadline, and prioritize the new feature so they work in it next

C. Talk to to the user and figure out if the new feature is important enough for them to change direction

D. Initiate the change control process

Agile teams value following a plan, so this is a good idea... but they value responding to change more. Is there another answer that's more in line with agile values?

This is the BEST answer! Agile teams value responding to change over following a plan. So the best thing to do is respond to the user quickly and get all of the information. Then they can all work together to figure out how the plan is impacted.

If you were studying for the PMP exam, this could be the right answer. And since many agile teams work in companies that have a change control process, they may eventually do this. But the question asked what the team should do first, and this would come later (and maybe not at all!).

THE "WHICH-IS-BEST" QUESTION HAS MORE THAN ONE GOOD ANSWER, BUT ONLY ONE **BEST** ANSWER.

HEAD LIBS

Fill in the blanks to come up with your own "which-is-BEST" question about how agile teams value customer collaboration *over contract negotiation.*

You're a developer on a _____ project. A _____
(an industry) (a type of user)

wants you to _____, but you need to _____.
(something the user wants you to do) (a conflicting thing you want to do)

What is the BEST way to handle this situation?

A. _____
(an obviously wrong answer that has nothing to do with this question at all)

B. _____
(a good answer that is a good idea, but isn't really relevant to this value)

C. _____
(a better answer that's consistent with valuing contract negotiation)

D. _____
(the BEST answer that's consistent with valuing customer collaboration)

LADIES AND GENTLEMEN, WE NOW RETURN YOU TO CHAPTER TWO

They think they've got a hit ...

Mike's been working with the development team for almost a year on the latest killer feature, and he's really excited that they're finally done.

... but it's a flop!

Uh-oh. It looks like Mike and his team wasted a year working on a product that nobody wants. What happened?

THIS WOULD HAVE BEEN GREAT IF WE'D DELIVERED IT A YEAR AGO. BUT THERE'S NO WAY WE CAN USE THIS TODAY!

Mike: What?! We've been working on this for a year. Are you telling me we just wasted our time?

Ben: I have no idea what you've been doing with your time. But I'm telling you right now, I don't see ***any*** of our clients using this.

Mike: But what about that big presentation we gave at that conference last year? Every client we talked to said they would love to figure out exactly who their listeners are and start marketing directly to them.

Ben: Right. And nine months ago, three of those clients were named in a lawsuit for violating privacy laws. Now none of them would touch this.

Mike: But... but this is a huge innovation! You have no idea how many technical problems we had to solve. We even brought in a specialized AI consulting company to help us do advanced customer analysis!

Ben: Look, Mike, I don't know what to tell you. Maybe you can repurpose the code for something else?

Mike: We're going to have to salvage what we can. But I'll tell you right now, **whenever we tear out chunks of code, we always have bugs**.

Ben: Ugh. I wish you'd talked to me about this sooner.

This makes sense! A major source of bugs is rework, or taking code that was already built and modifying it for another purpose.

BRAIN POWER

If agile teams value responding to change but changes often cause rework, then how do they keep that rework from always causing bugs?

The principles behind the Agile Manifesto

The four values in the Agile Manifesto do a really good job of capturing the core of the agile mindset. But while those four values are great at giving you a high-level understanding of what it means to "think agile," there are a lot of day-to-day decisions that every software team needs to make. So in addition to the four values, there are **twelve principles behind the Agile Manifesto** that are there to help you *really* understand the agile mindset.

Manifesto for Agile Software Development

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.

???

```
public object Convert
    (object value, Type targetType,
                object parameter, string language)
{
  double parsedValue;
  if ((value != null)
      && double.TryParse(value.ToString(),
                         out parsedValue)
      && (parameter != null))
      switch (parameter.ToString()) {
          case "Hours":
              return parsedValue * 30;
              case "Minutes":
              case "Seconds":
                  return parsedValue * 6;
          }
      return 0;
}
```

When you're building software, it's not always easy to see the direct connection between the agile values and the day-to-day work. That's where the principles behind the Agile Manifesto come in.

Behind the Scenes

The group at Snowbird came up with the four values pretty quickly, but it took them a few days of deep discussion to agree on the twelve principles behind the Agile Manifesto—and even after they left Utah, they still hadn't finalized the wording. The version they first came up with is a little different. The final version is on the next page (and also at http://www.agilemanifesto.org/, the official website of the Agile Manifesto). But even though the wording changed slightly over the first few years, the ideas laid out in the twelve principles haven't.

Principles behind the Agile Manifesto

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The agile principles help you deliver your product

The first three principles are all about delivering software to your users. And the most effective way to deliver the best software possible is to make sure that it's **valuable**. But what does "value" really mean? How do we make sure that we've got our users, stakeholders, and customers' best interests in mind when we're building software? These principles help us understand those things.

A product owner like Ben works with users and customers to really understand what they need in the software. He can usually spot a feature that the users won't actually use... which happens a lot more often than you might think!

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

So what does that mean, exactly? It means that early delivery and continuous delivery add up to satisfied users:

Early delivery

Getting the first version of the software into users' hands as early as possible so you can get early feedback

+

Continuous delivery

Constantly getting updated versions to the users so they can help the team build software that solves their most important problems

=

Satisfied users

The users help the team stay on track by making sure that the most important features are added first

This is how a lot of teams act when someone points out a major change to the code. It's understandable, because a change that could have been found earlier now requires a lot of work that can be slow and highly annoying.

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

How does the team react when someone points out that they need to make a change that will affect a lot of the code? Every developer has been through this, and it can be a lot of (often difficult) work. So how does the team react? It's natural to resist a big change. But if the team can find a way to not just accept but **welcome** that change, it means that they're ***putting the users' long-term needs ahead of their own short-term annoyance***.

"Requirements" just means what the software is supposed to do... but sometimes the users' needs change, or the programmers misunderstand what's needed, and that can lead to changing requirements.

When the team delivers software early and often to the users, stakeholders, and customers, that gives everyone lots of chances to find changes early, when they're much easier to make.

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Working software is the primary measure of progress.

When developers push back against changes, it's not an irrational response: if they've spent many months working on a feature, changing that code can be a slow, painful, and error-prone process. One reason is that when teams do **rework** (that's when they change existing code to do something new) it almost always leads to bugs, often ones that are really nasty and difficult to track down and fix.

So how does the team avoid rework? **Deliver working software to the users frequently**. If the team is building a feature that isn't useful or does the wrong thing, the users will spot it early, and the team can make the change before too much code is written... and preventing rework prevents bugs.

People actually say stuff like this when they welcome changing requirements. What does it mean for software to be "valuable"?

Principles make the most sense in practice.

Most of us have been on teams that have struggled at one point or another, and the most common way to handle that situation is to adopt a new practice. But some practices that work really well for some teams only get marginal results with other teams—just like we saw with the daily standup in Chapter 1.

So what makes the difference between the team that only gets so-so results with a practice and the one that gets really great results? More often than not, it has a lot to do with *the mindset of the team*, and the attitude they bring to the practice. And that's what these principles are about: helping teams find the best mindset that makes their practices as effective as possible.

You'll see an example on the next page... →

Principles in Practice

The first three principles behind the Agile Manifesto talk about early and continuous delivery of software, welcoming changing requirements, and delivering working software on a short timescale. So how do teams do that in the real world? With great practices, like **iteration** and using a **backlog**.

Iteration: repeatedly performing all of the project activities to continuously deliver working software

The team gets together at the start of the iteration to plan out which features they'll build. They try to include only work that they can actually get done during the iteration.

Iterations are timeboxed, so if you and your team discover partway through an iteration that there's more too much work planned for it, you'll push features to the next one.

DICTIONARY DEFINITION

time-boxed, adjective

setting a hard deadline for an activity to be completed, and adjusting the scope of that activity to meet the deadline

The team couldn't fit all of the requested features into the current ***timeboxed*** *iteration, so they concentrated on the most valuable ones.*

If you read our book "Learning Agile" then you'll recognize these iteration and backlog illustrations!

Backlog: a great way to manage changing requirements

The backlog is a list of features waiting to be built. Any feature that hasn't been included in an iteration yet is fair game for the users and the product owner to change.

When the team plans each iteration, they pull features to include off of the backlog.

HEY, WAIT A MINUTE! WE LEARNED ABOUT HOW SCRUM TEAMS USE A BACKLOG EARLIER. DOES THAT MEAN ***SCRUM SPRINTS*** ARE A FORM OF ***ITERATION***?

Yes! Scrum is an iterative methodology.

The Scrum practice of using sprints is a classic example of how teams use iteration in real life to deliver working software early and frequently. The Scrum team has a Product Owner who works with the users and stakeholders to understand their needs. Everyone learns more with each new version of the working software, and the Product Owner uses that new knowledge to add or remove features from the backlog.

We'll talk a lot more about Scrum in the next chapter.

Fireside Chats

Tonight's talk: **Practice meets principle**

Principle:

I've been looking forward to this debate for a while.

This again? There he goes, back on his "nothing gets done without practices" kick.

Yeah? Okay. Let's talk about those practices for a minute. Like the Daily Scrum, for example—

Yep! And it's not just the Daily Scrum. Let's talk about iteration.

Indeed. But what happens if the people on the team don't really, genuinely believe in the principle of delivering working software frequently?

Yes. But will they *really* deliver **working** software? Or will they cut corners just to push something out the door before the iteration ends? Will they *really* delay a feature until the next iteration because it won't fit? Or will adding iteration to the project make everyone on the team feel like they're just "going through the motions?"

Have you been on a team that tried to go agile but ended up with mediocre results? If so, this might remind you of your own experience.

Practice:

I'm not sure there's going to be much of a debate, if you want to know the truth.

Well, you have to admit, it's a pretty good point. After all, where would a methodology like Scrum be without me? Take away the sprints, backlogs, retrospectives, sprint reviews, Daily Scrum meetings, and sprint planning sessions, and what are you left with? Chaos!

Hold it right there. I already know what you're going to say next. It's that whole thing about the Daily Scrum getting "mediocre" results if the team doesn't "get" the principles.

A fantastic practice, thank you very much.

There will still be iterations! And you know what? It'll be better than it was before they added the practice.

This is true! Even when the team doesn't really understand the principles, adding iterations is still usually an improvement... not much of one, but enough to justify doing it.

Well, at least they'll have *something* to show for their effort. Even if they only get a marginal improvement, it's better than nothing!

Dictionary drill

Here are a bunch of definitions for words that you've seen in this chapter. Can you fill in the words that each definition belongs to?

_____, noun

work done by the team to change previously written code to make it function differently or serve a different purpose, often considered risky by teams due to its increased likelihood of introducing bugs

_____, adjective

setting a hard deadline for an activity to be completed, and adjusting the scope of that activity to meet the deadline

_____, noun

a practice used by teams in which the team works with users, customers, and/or stakeholders to maintain a list of features that will be built in the future, often prioritized with the most valuable features at the top

_____, adjective

a kind of methodology in which teams break the project down into smaller parts, delivering working software at the end of each one, and possibly changing direction based on the feedback from the working software

_____, adjective

a type of model, process, or method for building software in which the entire project is broken down into sequential phases, often including a change control process in which changes force the project into a prior phase

→ Answers on pagepage 45

there are no Dumb Questions

Q: Does each principle match up with exactly one practice? Is there a one-to-one correspondence?

A: Not at all. The first three principles behind the Agile Manifesto emphasize early and continuous delivery of software, welcoming changing requirements, and delivering working software frequently. And we used two practices (iteration and backlog) to help you understand the principles on a deeper level. But that doesn't mean there's a one-to-one relationship between the practices and the principles.

In fact, the opposite is true. You can have principles without practices, and you can have practices without principles.

Q: I'm not sure how that works. What exactly do you mean by "practices without principles?

A: Here's an example of what it looks like when a team puts a practice in place without really understanding or internalizing the agile principles. Scrum teams hold a **retrospective** at the end of each sprint so that they can talk about what went well and what can be improved.

But take another look at the last principle in the list:

> At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

What if the team hasn't really taken this idea to heart? They'll still have the retrospective meeting because the Scrum rules tell them to do it. And they'll probably talk about some of the problems that they had, which can certainly lead to some marginal improvements.

The problem is that while the new meeting did something, it feels like it's somewhat "empty" or superfluous. The people on the team feel like they're taking time away from their "real" jobs to do it. Eventually, they'll start talking about replacing it with something more "efficient," like using an email discussion list or Wiki page. A lot of teams have that experience when they add practices without principles.

Q: Okay, I think I see how you can have practices without really believing in the principles. But how can you have principles without practices?

A: A lot of people have a little trouble with this idea when they first come in contact with the idea of an "agile mindset" driven by principles.

So what does it look like if the team takes very seriously the agile principle of reflecting on how to become more effective, but doesn't have a specific practice for doing that reflection? That's actually pretty common on *very effective* agile teams. Everyone is in the mindset of reflecting often, so when someone feels like it's time to reflect on how the project has gone and make necessary corrections, that person usually grabs a few other team members and has an informal retrospective. If something good comes out of it, they talk it over, and make the necessary correction.

For a team used to a methodology like Scrum that has well-specified rules, that feels disorganized, chaotic, or "loosey-goosey." That's one reason teams like to standardize on a set of practices—so that everyone has common ground rules.

Q: Why did you capitalize "Product Owner" on the bottom of page 25?

A: Because while many people have the job title "product owner," spelling Product Owner with capital letters refers to a specific role with responsibilities specified by the Scrum rules. You'll learn more about that in the next chapter.

When the team adopts practices without the right principle-driven mindset, it often feels "empty" or superfluous, like they're just going through the motions, and they'll start looking for alternatives that take less effort.

Ben: And just as I was feeling good about how things were going. I wish you didn't have to be so negative. What's the bad news now?

Kate: I'm not trying to be negative. I'm really happy about the progress we've made just by starting to use iteration.

Ben: Right! I went back to the users with early builds, and they found all sorts of changes that we could fix early without a lot of rework.

Kate: Yeah, and that's great. But we still have problems.

Ben: Such as...?

Kate: Well, like that meeting we had last Wednesday. We spent all afternoon arguing about documentation.

Ben: Why do you want to bring that up again? You and Mike keep asking for specifications with every tiny detail about what to build.

Kate: Yes, because then I can help the team plan out the project, and Mike and the team know exactly what to build.

Ben: But it's not that simple! These specs are really hard to write. And even when we write a spec for one iteration, it still gets really long.

Kate: Look, if you've got a better idea about how to get the team to build the right software, I'd love to hear it.

BRAIN POWER

A lot of teams seem to have trouble writing and reading highly detailed specifications. Can you think of a more effective way for a product owner to help the team to understand exactly what the users need them to build?

The agile principles help your team communicate and work together

Modern software is built by teams, and while individual people are really important to any team, teams work best when everyone works together—which means developers not just working with each other, but with the users, customers, and stakeholders, too. That's what these next principles are all about.

- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

It's really common for developers to dread or resent meeting with users, because those meetings often uncover changes, which leads to rework that can often be difficult and frustrating. But when the team has a better, more agile mindset, they know that **meeting with users more often** keeps them in sync, and actually prevents those changes.

A more agile mindset would help Mike see that working with his users more often actually prevents those changes.

MEET WITH THE USERS ***MORE OFTEN***?! BUT ALL THEY EVER DO IS ASK US FOR CHANGES.

Teams do their best work when the people on them are **motivated**. Unfortunately, most of us have had bosses or coworkers who seemed determined to drain all of that great motivation. When people feel like they aren't allowed to make mistakes without serious consequences, are pressured to work extremely long hours, and generally feel like they aren't trusted to do their jobs, the quantity and quality of their work plummets. Teams with a more agile mindset know that when everyone is trusted and given a good working environment, they flourish.

I DON'T CARE IF THE TEAM'S BEEN WORKING 70 HOURS A WEEK. FAILURE IS NOT AN OPTION, AND ***MISTAKES WILL BE REPORTED***.

This is a great way to demotivate your whole team and cause them to do lousy work. Ben isn't even the boss! But he can still create an environment of fear and distrust for everyone around him.

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Let's be honest—we don't always read every word in the manual before turning on a new gadget. So why would we expect people to do that with a spec?

Waterfall teams typically build a requirements specification first, and then design the software based on those requirements. The problem is that three people can read the same spec and come away with three very different ideas about what the team is supposed to build. This can be a little surprising—shouldn't specifications be precise enough to give everyone the same idea?

There are two problems with that in the real world: writing technical material is hard, and reading it is even harder. Even if the person writing the spec does a perfect job describing what needs to be built (which rarely happens), the people reading it will very often interpret it differently. So how do you get around this problem?

The answer is surprisingly simple: **face-to-face conversation**. When the team gets together and talks about what they need to build, it *really is* the most efficient and effective way to communicate exactly what needs to be built... and also status, ideas, and any other information.

there are no Dumb Questions

Q: Are you saying that when people are demotivated, they do bad work on purpose?

A: No, not on purpose. But it's very difficult to innovate, create, or do the mentally taxing tasks required to work on a software team when you're in a demotivating environment. And it's surprisingly easy to demotivate a team: your motivation gets drained when you're not trusted to do your job, harshly punished or publicly embarrassed if you make a mistake (everyone makes mistakes!), or held to unreasonable deadlines that you have no input into or control over. Those are all things that have been shown repeatedly to drag down software teams and make them a lot less productive.

Q: Wait, go back to what you were saying about mistakes. We've been talking about welcoming changes. But if you're making a change, doesn't that mean someone made a mistake earlier that has to be changed now?

A: It's dangerous to think of changes as mistakes, especially when you're using iteration. A lot of times, everyone on the team and the users and stakeholders all agree that the software should be built to do something, but when the users get their hands on the working software at the end of the iteration, they realize that it needs to change—not because they made a mistake earlier, but because they now have information they didn't have at the start of the iteration. That's actually a really effective way to build software. But it only works if people feel comfortable making changes, and if they don't call it a mistake or "blame" anyone for finding the change.

Q: Don't we need specifications for more than just communication? What if you need to refer back to the spec in the future? Or if it needs to be distributed to a lot of people?

A: Sure, and those are good reasons to write things down. And that's why agile teams value comprehensive documentation—they just value working software more.

One thing to keep in mind, though, is that if you're writing documentation to refer back to, or to distribute to a wide audience beyond the software team, then a software specification may not be the right kind of document for the job. Documentation is a tool to get a job done, and you always want to use the right tool for the job. The information that teams need in order to build software is usually different than the information a user or manager might need after the software is built, so trying to create a document that serves both purposes might do neither particularly well.

Q: Hey, it looks like the chapter is almost over, and you haven't covered all twelve principles! Why not?

A: Because the agile principles aren't just an isolated topic that teams learn once and then move on from. They're important because they help you understand how agile teams think about the way they work together to build software. That's why the values and principles of the Agile Manifesto are important.

We're not going to stop talking about the agile mindset, values, or principles, even though we're moving on to methodologies in the next chapter. We'll keep coming back to them, because they help you understand the methodologies (for example, Scrum teams are self-organizing, and XP teams value simplicity).

BULLET POINTS

- Software is **valuable** when it does what the users, customers, or stakeholders need it to do
- Teams do that best when they deliver an **early** version to the users, and keep delivering **continuously**
- Agile teams **welcome changing requirements**, and finding those changes early helps prevent rework
- The best way to find those changes early is to get **working software to the users frequently**
- Documents are helpful, but the most the *most effective* way to convey information is **face-to-face conversation**
- Developers on agile teams **work with business people every day**, including users and stakeholders
- **Iteration** is a practice in which teams break the software down into frequent timeboxed deliveries
- A **backlog** is a practice in which teams maintain a list of features that will be built in future iterations

Scrum teams actually maintain two backlogs: one for the current sprint, and one for the whole product.

You'll learn more about that in the next chapter.

Getting into a more agile mindset isn't always easy! Sometimes we get it, but sometimes we need some work. Here are some things we overheard Mike, Kate, and Ben saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the agile principle that it's either compatible or incompatible with.

WHY ARE YOU ASKING ME QUESTIONS? I ALREADY WROTE DOWN EVERYTHING THE USERS ASKED FOR IN THE SPECIFICATION.

COMPATIBLE

INCOMPATIBLE

> Working software is the primary measure of progress.

I JUST FOUND OUT THAT THE AUDIENCE SIZE CALCULATION ALGORITHM WE'RE USING DOESN'T WORK. WE NEED TO PUSH THIS FEATURE TO THE NEXT ITERATION.

COMPATIBLE

INCOMPATIBLE

> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

OKAY, WHICH OF YOU IDIOTS WROTE THIS BUGGY BLOCK OF SPAGHETTI CODE? IT'S YOUR FAULT THAT WE'RE BEHIND.

COMPATIBLE

INCOMPATIBLE

> Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

I'M RUNNING THE LATEST BUILD, BUT I THOUGHT WE'D BE A LOT FURTHER ALONG ON THAT ANALYTICS FEATURE. IS THERE A PROBLEM I DON'T KNOW ABOUT?

COMPATIBLE

INCOMPATIBLE

> Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Answers on page 46

The new product is a hit!

Kate and Mike delivered a great product, and it's been extremely successful.

In fact, it's gone so well that Ben has some fantastic news that everyone will want to hear. Nice work, team!

Mindsetcross

See how well you understand the agile values and principles. Can you solve this without flipping back to the rest of the chapter?

Across

1. When a deadline's been set, and the scope is adjusted to meet it
3. A great way to manage changing requirements
6. How often to deliver
12. When teams repeatedly perform all of the project activities in small chunks
13. What the team does to its behavior after a retrospective
16. An effective way to communicate complex requirements and ideas
19. There's no single "_____" way to build software
22. What we do with customers
23. Something that shouldn't be punished if you want a motivated team
24. What business people and developers must do together daily
25. Very useful for agile teams because they help get the work done
26. At regular _____ the team reflects on how to become more effective
27. The most effective and efficient method of conveying information
28. Teams work best when you pay attention to them

Down

2. When to deliver software
4. The kind of delivery agile teams try to achieve
5. Attitude towards customers or other teams that requires strict agreements before any work can start
7. What agile teams respond to
8. You need to _____ the team to get the job done
9. Traditional but often less-than-effective way to build software
10. The kind of individuals to build projects around
11. Working software is the primary measure of _____
14. Where the original authors of the Agile Manifesto got together
15. What happens to your team if you create a culture of fear
17. Agile teams still follow one
18. The kind of software delivered at the end of every iteration
20. The kind of users that are the highest priority for agile teams
21. Avoid this if you can

→ Answers on page page 47

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. You're a project manager on a team building network firmware for embedded systems. You've called a meeting to give a demo of the latest version of code the team has been working on for a control panel interface to a very technical group of business users and customers. This is the fifth time that you've called a meeting to do a demo like this. And for the fifth time, the users and customers asked for specific changes. The team will now go back and work on a sixth version, and you'll repeat the process again.

Which of the following BEST describes this situation?

A. The team does not understand the requirements
B. The users and customers don't know what they want
C. The project needs better change control and requirements management practices
D. The team is delivering value early and continuously

2. Which of the following is NOT a Scrum role?

A. Scrum Master
B. Team Member
C. Project Manager
D. Product Owner

3. Joaquin is a developer, and his software team is in the process of adopting agile. One of the project's users wrote a brief specification that describes exactly what she wants for a new feature, and Joaquin's manager assigned him to work on that feature. What should Joaquin do next?

A. Demand a meeting with the user, because agile teams recognize that face-to-face conversation is the most efficient and effective method of conveying information
B. Read the specification
C. Ignore the specification, because agile teams value customer collaboration over comprehensive documentation
D. Start writing code immediately, because the team's highest priority is to satisfy the customer through early delivery of valuable software

4. Which of the following is TRUE about working software?

A. It does what the users need it to do
B. It meets the requirements in its specification

Exam Questions

C. Both A and B
D. Neither A nor B

5. Which of the following statements BEST describes the Agile Manifesto?

A. It describes the most effective way to build software
B. It contains practices that many agile teams use
C. It contains values that establish an agile mindset
D. It defines rules for building software

6. Scrum projects are divided into:

A. Phases
B. Sprints
C. Milestones
D. Rolling wave planning

7. You are a developer at a social media company working on a project to build a new feature to create a private site for a corporate client. You need to work with your company's network engineers to determine a hosting strategy, and come up with a set of services and tools that the engineers will use to manage the site. The network engineers want to host all of the services internally on your network, but you and your teammates disagree and feel that the services should be hosted on the client's network. Work on the project has come to a halt while everyone tries to come to an agreement. Which agile value BEST applies to this situation?

A. Individuals and interactions over processes and tools
B. Working software over comprehensive documentation
C. Customer collaboration over contract negotiation
D. Responding to change over following a plan

8. Donald is a project manager on a team that follows separate phases for each project, starting with a requirements phase followed by a design phase. Some work can begin on the code before the requirements and design are finished, but the team typically doesn't consider any work to be complete until those phases are finished. Which term BEST describes Donald's projects?

A. Iterative
B. Rolling wave planning
C. Waterfall
D. Scrum

Exam Questions

9. Keith is the manager of a software team. He's made it clear that mistakes are not to be tolerated. A developer spent several hours building "proof of concept" code to test a possible approach to a complex problem. When he eventually discovered from the experiment that the approach wouldn't work, Keith yelled at him in front of the whole team and threatened to fire him if he did it again.

Which agile principle BEST applies to this situation?

A. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

B. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

C. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

D. Continuous attention to technical excellence and good design enhances agility.

10. What's the highest priority of an agile team?

A. Maximizing the work not done

B. Satisfying the customer by delivering valuable software early and often

C. Welcoming changing requirements, even late in development

D. Using iteration to effectively plan the project

11. Which of the following statements is NOT true about the daily standup?

A. The length is kept short by having everyone stand for the duration of the meeting

B. It's the same thing as a status meeting

C. It is most effective when everyone listens to each other.

D. It's an opportunity for every team member to get involved in planning the project.

12. Which of the following BEST describes the agile mindset with respect to simplicity?

A. Maximizing the work not done

B. Satisfying the customer by delivering valuable software early and often

C. Welcoming changing requirements, even late in development

D. Using iteration to effectively plan the project

13. A'ja is a project manager on a team that is just starting their agile adoption. The first change they made to the way they work was to start holding daily standup meetings. Several team members have approached her to say that they don't like attending. And despite the fact that

Exam Questions

she's getting some valuable information from the team at each standup, A'ja is concerned that the extra lines of communication might not be worth damaging the team cohesion.

What is the BEST thing for A'ja to do?

A. Stop holding the daily standup and find another way to adopt agile.
B. Make and enforce a rule that every attendee must put away his or her phone and pay attention.
C. Follow up with people individually after the meeting to get more detailed status.
D. Work with the team on changing their mindset.

14. You're a developer on a software team. A user has approached your team about building a new feature, and has provided requirements for it in the form of a specification. She is very certain of exactly how the feature will work, and promises there will be no changes. Which agile value BEST applies to this situation?

A. Individuals and interactions over processes and tools
B. Working software over comprehensive documentation
C. Customer collaboration over contract negotiation
D. Responding to change over following a plan

15. Which of the following is NOT a benefit of welcoming changing requirements?

A. It gives the team a way to explain a missed deadline
B. The team builds more valuable software when customers aren't pressured not to change their minds
C. There's more time and less pressure so the team can make better decisions
D. Less code is written before changes happen, which minimizes unnecessary rework

16. Which of the following is NOT part of an agile team's mindset towards working software?

A. It contains the final version of all features
B. It is the primary measure of progress
C. It is delivered frequently
D. It is an effective way to get feedback

17. Which of the following is NOT true about iteration?

A. The team must finish all planned work by the end of an iteration
B. Iterations have a fixed deadline
C. The scope of work performed during an iteration may change by the time it ends
D. Projects typically have multiple sequential iterations

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: D

Did this situation sound negative, like something was going drastically wrong? If it did, you may want to think about your own mindset! This was actually a pretty accurate description of a very successful agile project that uses an iterative methodology. It only sounds like the project is running into problems if you approach it with a mindset that considers change and iteration to be a mistake rather than a healthy activity. If you see the project this way, then you'll be tempted to "blame" the team for not understanding the requirements, or the users for not knowing what they want, or the process for not having adequate controls to prevent and manage changes. Agile teams don't think about things like that. They know that the best way to figure out what the users need is to deliver working software early and frequently.

2. Answer: C

Project managers are very important, but there's no specific role in Scrum called "project manager." Scrum has three roles: Scrum Master, Product Owner, and Team Member. The project manager will fill one of those roles on a project that uses Scrum, but will often still have the "Project Manager" job title.

When your team follows an agile methodology that has specific roles, the role that you fill doesn't always match the title on your business card, especially when your team is just starting to adopt the methodology.

3. Answer: B

It's true that agile teams value customer collaboration, believe face-to-face conversation is the most effective method of conveying information, and place the highest priority on delivering software. However, the user took the time to write the specification, and the information in it could be very helpful in either writing code or having a face-to-face conversation.

When someone takes the time to write down information they think is important, it's very UN-collaborative to ignore it.

4. Answer: D

When agile teams talk about working software, they mean software that they consider "done" and ready to demonstrate to the users. But there's no guarantee that it fulfills the users' needs or that it meets the specific requirements in a specification. In fact, the most effective way to build software that genuinely helps users is to deliver working software frequently. The reason is because the early versions of working software typically **don't fully meet the users' needs**, and the only way for everyone to figure that out is to get it into the hands of the users so they can give feedback about it.

This is why agile teams value early and continuous delivery of working software.

Exam ~~Questions~~ Answers

5. Answer: C

The Agile Manifesto contains the core values shared by effective agile teams. It doesn't define a "best" way to build software or a set of rules that all teams should follow, because people on agile teams know that there's no "one size fits all" approach that works for all teams.

6. Answer: B

Scrum teams work in sprints, typically (but not always) 30 days long. They plan the next 30 days of work (assuming the length is 30 days) at the start of the sprint. At the end of the sprint, they demonstrate working software to the users, and also hold a retrospective to review what went well and find ways to improve.

7. Answer: C

The project is suffering because the team is having trouble collaborating with their customer. In this case, the network engineers are the customer, because they're the ones who will be using the software. This is a situation where it would be easy to take a contract negotiation approach, laying out specific terms and documents to describe what will be built so that software development work can begin. But it's more effective to genuinely collaborate with them and work together to discover the best technical solution.

8. Answer: C

A waterfall project is divided into phases, typically starting with requirements and design phases. Many waterfall teams will begin "pre-work" on code once the requirements and design have reached a stable point, even if they're not yet complete. However, this is definitely not the same thing as iteration, because the team doesn't change the plan based on what they learned building and demonstrating working software.

9. Answer: B

Agile projects are built around motivated team members. Keith is taking actions that undermine the whole team's motivation by undercutting a team member who's taking a good risk and genuinely trying to make the project better.

Exam ~~Questions~~ Answers

10. Answer: B

Flip back and reread the first principle of agile: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." The reason this is the highest priority is because agile teams are focused first and foremost on delivering software that's valuable. All of the other things we do on projects—planning, design, testing, meetings, discussion, documentation—are really, really important, but it's all in service of delivering that valuable software to our customers.

11. Answer: B

While some teams treat the Daily Standup as a status meeting where each team member gives an update to a boss or project manager, that's not really its purpose. It works best when everyone listens to each other, and uses it to plan the project together as a team.

12. Answer: A

Agile teams value simplicity, because simple designs and code are much easier to work with, maintain, and change than complex ones. Simplicity is often called "the art of maximizing the work not done" because—and this is especially true of software—the most effective way to keep something simple is often to simply do less.

13. Answer: D

The reason that the team isn't paying attention during the daily standup is because they don't really care about it or buy into it as an effective tool, and mainly want it to end as quickly as possible so they can get back to their "real" jobs. When teams have this mindset, it's likely that they will eventually stop attending the meeting altogether, and the agile adoption is much less likely to be successful. The daily standup practice will be more effective if the team understands how it helps each of them, both individually and as a team. That mindset shift can only be accomplished through open and honest discussion about what's working and what isn't. That's why working with the team on changing their mindset is the best approach to this situation.

14. Answer: B

It certainly makes sense to read and understand the specification. But the most effective way to truly ascertain whether or not the team really understands what she intended is to deliver working software to her, so that she can see how the requirements she documented were interpreted and work with the team to determine what works well and what needs to change.

Exam ~~Questions~~ Answers

15. Answer: A

There are a lot of great reasons that agile teams welcome changing requirements. When customers are encouraged to change their minds (rather than discouraged from it), they give better information to the team, and that leads to better software. And even when people keep their mouths shut about changes, they almost always eventually get exposed in the end, so when the team gets them early it gives them more time to respond—and the earlier the changes arise, the less code has to be reworked.

However, changes are never an excuse for poor planning or missed deadlines. Effective agile teams generally have an agreement with their users: the teams welcome changing requirements from users, customers, and managers, and in return they aren't blamed for the time it takes to respond to those changes, because everyone recognizes it's still the fastest and most effective way to build software. So nobody really sees welcoming changing requirements as giving the team a way to explain a missed deadline, because the deadlines should already be adjusted to account for the changes.

16. Answer: A

Working software is delivered frequently so that the team can get frequent feedback and make changes early. That's why working software should never be assumed to contain the final version of any requirement. That's why it's "working" software, not "finished" software.

17. Answer: A

Iterations are timeboxed, which means that the deadline is fixed and the scope varies to fit it. The team starts each iteration with a planning meeting to decide what work will be accomplished. But if it turns out that they didn't get the plan right and work takes longer than expected, then any work that didn't get done is pushed to the next iteration.

Manifesto Magnets Solution

Manifesto for Agile Software Development

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 .

Dictionary drill SOLUTION

Here are a bunch of definitions for words that you've seen in this chapter. Can you fill in the words that each definition belongs to?

We used "rework" as a noun earlier in the chapter: "a major source of bugs is rework."

rework, noun

work done by the team to change previously written code to make it function differently or serve a different purpose, often considered risky by teams due to its increased likelihood of introducing bugs

Rework can also be a verb: "We had to rework this bit of code to adapt it to a new purpose."

timeboxed, adjective

setting a hard deadline for an activity to be completed, and adjusting the scope of that activity to meet the deadline

This can also be used as a verb: "Let's timebox the work we're doing on this feature to six hours."

backlog, noun

a practice used by teams in which the team works with users, customers, and/or stakeholders to maintain a list of features that will be built in the future, often prioritized with the most valuable features at the top

iterative, adjective

a kind of methodology in which teams break the project down into smaller parts, delivering working software at the end of each one, and possibly changing direction based on the feedback from the working software

Normally "waterfall" is a noun. But in this case it's actually an adjective that describe a type of process.

waterfall, adjective

a type of model, process, or method for building software in which the entire project is broken down into sequential phases, often including a change control process in which changes force the project into a prior phase

Here's how it's used in a sentence: "Brian used to work at a company that followed a waterfall process, so she's really excited to try an agile process like Scrum."

Getting into a more agile mindset isn't always easy! Sometimes we get it, but sometimes we need some work. Here are some things we overheard Mike, Kate, and Ben saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the agile principle that it's either compatible or incompatible with.

WHY ARE YOU ASKING ME QUESTIONS? I ALREADY WROTE DOWN EVERYTHING THE USERS ASKED FOR IN THE SPECIFICATION.

COMPATIBLE

INCOMPATIBLE

Working software is the primary measure of progress.

It's not fair to ask users for requirements at the start of the project and then refuse to let them change their minds (as long as they understand that the team needs time to make the changes).

When Kate discovered this change, they could have delivered software that didn't work, or delayed the delivery to fix it. But pushing the feature to the next iteration is a better choice, because they'll still deliver working software with the other features.

I JUST FOUND OUT THAT THE AUDIENCE SIZE CALCULATION ALGORITHM WE'RE USING DOESN'T WORK. WE NEED TO PUSH THIS FEATURE TO THE NEXT ITERATION.

COMPATIBLE

INCOMPATIBLE

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

OKAY, WHICH OF YOU IDIOTS WROTE THIS BUGGY BLOCK OF SPAGHETTI CODE? IT'S YOUR FAULT THAT WE'RE BEHIND.

COMPATIBLE

INCOMPATIBLE

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Technical people like Mike are often really blunt. But even if the team has a culture where it's okay to challenge and even insult people, blaming one person for delays or quality issues is really demotivating.

If Kate was only using on her schedule to measure progress, she might think the project was going just fine. Relying on working software as her primary measure of progress helps her spot (and hopefully fix!) problems early.

I'M RUNNING THE LATEST BUILD, BUT I THOUGHT WE'D BE A LOT FURTHER ALONG ON THAT ANALYTICS FEATURE. IS THERE A PROBLEM I DON'T KNOW ABOUT?

COMPATIBLE

INCOMPATIBLE

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Mindsetcross

SOLUTION

		1 T	I	M	E	B	O	X	2 E	D		3 B	A	4 C	K	L	O	G									
									A					O					5 C							
								6 F	R	E	Q	U	E	N	T	L	Y		O							
	7 C			8 T		9 W			L					T					N		10 M					
	H			R		A			Y			11 P		12 I	T	E	R	A	T	I	O	N				
	13 A	D	J	U	S	T	S					R		N					R		T		14 S			
	N			S		E		15 D			16 D	O	C	U	M	E	N	T	A	T	I	O	N			
	G			T		R		E		17 P		G		O					C		V		O			
	E		18 W			F		M		L		R		U		19 B	E	20 S	T		A		W			
21 R		22 C	O	L	L	A	B	O	R	A	T	E		S				A			T		B			
E			R			L		T		N		S			23 M	I	S	T	A	K	E		I			
24 W	O	R	K			L		I				S						I			D		R			
O			I					V						25 T	O	O	L	S					D			
R		26 I	N	T	E	R	V	A	L	S								F								
K			G					T										I								
					27 F	A	C	E	T	O	F	A	C	E			28 P	E	O	P	L	E				
																		D								

3 managing projects with Scrum

The Rules of Scrum

The rules of Scrum are simple. Using it effectively is not so simple.

Scrum is the most common agile methodology, and for good reason: the **rules of Scrum** are straightforward and easy to learn. Most teams don't need a lot of time to pick up **the events, roles, and artifacts** that make up the rules of Scrum. But for Scrum to be really effective, they need to really understand **the values of Scrum** and the Agile Manifesto principles, which help them get into an effective mindset. Because while Scrum seems simple, the way a Scrum team constantly **inspects and adapts** is a whole new way of thinking about projects.

COMING SOON FROM RANCH HAND GAMES

THE TEAM THAT BROUGHT YOU THE BLOCKBUSTER INTERNATIONAL HIT COWS GONE WILD VIDEO GAMES

Meet the Ranch Hand Games team

They're hot off the successful release of *Cows Gone Wild IV: The Milk Man Cometh*, and about to tackle their most ambitious project yet! But while CGW4 may have sold well, but it was far from perfect as a project, and Amy, Brian, and Rick want CGW5 to be an improvement. Agile to the rescue!

Amy: Wow, I'm so glad you said that. I can't take another project like that. Just staying on top of the last-minute artwork changes was practically impossible.

Brian: Oh man, we can't have that argument again. You know they were necessary because the levels kept changing. I had my whole team working nights and weekends just to keep up.

Amy: I know, I know. We all got overwhelmed trying to tackle too many things at the same time.

Brian: No matter how much planning we did, our schedules never seemed long enough.

Rick: Yeah, that was definitely a problem—and I've been doing a lot of research about how to fix that. What do you guys think about using **Scrum**?

Amy: I've been reading about it, too. I think it could help.

Brian: Look, anything that can help make things less insane around here gets my vote.

Amy: But don't the Scrum rules say that we need a Product Owner and a Scrum Master?

Rick: Well, I'm the project manager, so it makes sense for me to be the Scrum Master. Amy, you work with the business side a lot, right? What do you think about being the Product Owner?

Amy: I'll try anything once. I'll start letting the business and PR teams know that I'm the Product Owner.

Brian: Let's do this!

The Scrum events help you get your projects done

Scrum is the most popular agile methodology, and for good reason. The rules of Scrum are simple, and teams all around the world have been able to adopt them and improve their ability to deliver projects. Every Scrum project follows the same pattern, which is defined by a **series of timeboxed events** that always happen in the same order. Here's what the Scrum pattern looks like:

The Scrum events:

- The sprint
- The sprint planning session
- The Daily Scrum
- The sprint review
- The sprint retrospective

This is the single source for requirements and any changes that will be made to the product throughout the project.

Every Scrum project is organized into timeboxed iterations called sprints. Many teams use 30-day sprints, but it's pretty common to see two-week sprints too.

The team uses the product backlog to keep track of the features that they'll build for the whole project.

At the beginning of each sprint, the team gets together for a sprint planning session where they choose what items to include in the sprint.

Every day, the team holds a Daily Scrum, a short meeting where each person gives an update on their progress, what they'll work on next, and any roadblocks they've hit.

The items for the sprint are pulled off of the product backlog and added to the sprint backlog. During the sprint, all of the development work is focused on building the items in the sprint backlog.

When sprint is done, the team holds a meeting called the sprint review, where they meet with the users and give a demo of the working software that they built.

The very last thing the team does in the sprint is meet and hold a retrospective, where they talk about what happened during the sprint so they can reproduce the things that went well and learn from any problems.

If you've read our book "Learning Agile" then you'll recognize this illustration of the basic Scrum pattern!

The Scrum roles help you understand who does what

There are three roles that must be filled on every Scrum team. The first role is the one that's most familiar to us: the **development team**. People on the team may have different areas of expertise, and maybe even different job titles in the company, but they all participate in the Scrum events the same way. There are also two very important roles filled by team members: the **Product Owner** and the **Scrum Master**.

The Product Owner helps the team understand the users' needs so they can build the most valuable product.

The Product Owner works with the team every day to help them understand the features in the product backlog: what items are on it and why the users need them. This is a really important job because it helps the team ***build the most valuable software they can***.

Flip back to the last chapter. Can you find an agile principle that talks about delivering valuable software?

The Scrum rules are clear that the Product Owner and Scrum Master roles are each filled by a person and not a committee.

The Scrum Master helps the team understand and execute Scrum.

Scrum may be simple to describe, but it's not always easy to get right. That's why there's one person on the team, the Scrum Master, whose whole job is to help the development team, the Product Owner, and the rest of the company to do exactly that—get Scrum right.

The Scrum Master is a leader (which is why the word "master" is right there in the name). But he or she demonstrates a very *particular* kind of leadership: the Scrum Master is a **servant leader**. This means that the person in this role spends all of his or her time helping (or "serving") the Product Owner, the development team, and people throughout the organization:

- Helping the product owner find effective ways to manage the backlog
- Helping the development team understand the Scrum events, and facilitating them if needed
- Helping the rest of the organization to understand Scrum and work with the team
- Helping everyone do the best job they can to deliver the most valuable software possible

Watch it!

Scrum teams need at least three members, and they usually max out at nine people.

A Scrum team needs at least three people in order to get a significant amount of work done in a sprint. But once the team size gets up to ten or more people, it becomes really hard for them to coordinate with each other, the Daily Scrum gets too chaotic, and it gets difficult to plan effectively.

The Scrum artifacts keep the team informed

Software projects run on information. The team needs to know about the product that they're working on, what they're building in the current sprint, and how they'll get it built. Scrum teams use three **artifacts** to manage all of this information: the **product backlog**, the **sprint backlog**, and the **increment**.

Here's an example of a product backlog—but there's no rule that says that it has to look like this. Many teams keep spreadsheets, or add entries to a database, or use software tools to manage their backlogs.

Cows Gone Wild 5 Product Backlog

Item #1: The stealth sheep barn level needs to be designed and play tested

Estimated effort: 27 person-days

Value: Builds on the most popular level from CGW4, gamers will be excited about this

Item #2: The milk gun physics and need to be updated to include squirting action

Estimated effort: 4 person-days

Value: Improved gameplay will make the game more fun

Item #3: The mad cow zombie survival level design needs to be finished, including zombie AI and zombie horde attacks

Estimated effort: 16 person-days

Value: Zombies are really popular right now, will be a good selling point

Item #4: The strategic silo assault will have an overhead view of the battlefield, giving the player placement and command/control of turrets, barnyard troops, tractors, feed lots, and snipers

Estimated effort: 19 person-days

Value: Similar to a feature in CGW4, we can add a whole level to the game by reusing code

Page 1 of 7

Every item in the product backlog has four attributes: order, description, estimate, and value.

There's no rule that says that the estimate has to be in person-days. It just needs to be in a unit that everyone on the team understands.

The value can be a description like this, or it can be a relative number, expected dollar value, or another way to measure or express value.

The product backlog is never complete as long as the project is running. The Product Owner will constantly work with users and stakeholders throughout the company to add, remove, change, and reorder the items in the product backlog.

The Product Owner keeps the project on track by continuously grooming the product backlog: staying up to date on what the company needs, and adding, removing, and reordering backlog items.

Here's an example of a sprint backlog (again, there's no rule that says that it has to be formatted exactly like this). The team creates it during sprint planning, but it can evolve based on what the team learns as the sprint unfolds.

Cows Gone Wild 5 Sprint Backlog - Sprint #2

Sprint goal: Create at least one level that can be played from beginning to end

The team crafts the sprint goal, or the objective that will be met by the team by completing the sprint items.

Sprint items:

- Stealth sheep barn level
- Milk gun physics
- Strategic silo assault

In the first part of the sprint planning session, the team determines what can be done in the sprint by choosing which items to include in it.

Sprint plan:

Tasks for the stealth sheep barn level

- Code the two new enemy heifer types
- Create texture maps
- Design the 3D space for the level in the level editor
- Build the udder stealth mode AI

In the second part of the sprint planning session, the team determines how the chosen work will get done by decomposing the sprint items into tasks.

Tasks for the milk gun physics

- Work out the algorithm for impact splatter
- Implement the milk gun class
- Update code for collision detection

Page 1 of 4

The sprint planning session's timebox often expires before the team is done decomposing all of the items into tasks, so they typically start with the items that will get built first and finish the plan along the way.

The increment is the sum of all backlog items that are actually completed and delivered at the end of the sprint

Scrum is an incremental methodology, which means the project is broken into "chunks" that are delivered one after another. Each of those"chunks" is called an **increment**, and each increment represents the result of one complete sprint: the working software that the team demonstrates to the users in the sprint review. That working software typically includes all of the features that they previously delivered —which makes sense, because they're not going to delete them!—so the *product increment is the sum of all of the backlog items completed during this sprint and all of the previous sprints*.

The Scrum Sprint Up Close

30 days

The **sprint** is a ***timeboxed*** iteration. Many teams use a 30-day sprint, but it's common to see two-week sprints as well.

Planning

Daily Scrum

Daily Scrum

Daily Scrum

Development

Daily Scrum

Daily Scrum

Daily Scrum

Sprint Review

Retrospective

The **sprint planning** session is a meeting with the whole team, including the Scrum Master and Product Owner. For a 30-day sprint it's timeboxed to eight hours (shorter sprint lengths will have a shorter planning timebox). It's divided into parts, each timeboxed to half of the meeting length:

- In the first part, the team figures out ***what*** can be done in the sprint. First the team writes down the **sprint goal**, a one- or two-sentence statement that says what they'll accomplish in the sprint. Then they work together to pull items from the product backlog to create the **sprint backlog**, which has everything they'll build during the sprint.
- In the second part, they figure out ***how*** the work will get done. They break down (or **decompose**) each item on the sprint backlog into **tasks** that will take one day or less. This is how they create a ***plan*** for the sprint.

The meeting timebox might expire before the team's done decomposing every item on the sprint backlog, so they concentrate on planning the first week of the sprint.

The **Daily Scrum** is a 15-minute timeboxed meeting. It's held at the same time every day, and everyone on the team (including the Product Owner and Scrum Master) participates. Each person answers three questions:

- What have I done since the last Daily Scrum?
- What will I do between now and the next Daily Scrum?
- What roadblocks are in my way?

In the **sprint review** the whole team meets with key users and stakeholders who have been invited by the Product Owner. The team demonstrates what they built during the sprint, and gets feedback from the stakeholders. They'll also discuss the product backlog, so that everyone knows what will *probably* be on it for the next sprint. For 30-day sprints, this meeting is timeboxed to four hours.

The **sprint retrospective** is a meeting that the team uses to figure out what went well and what can be improved. Everyone on the team participates, including the Scrum Master and Product Owner. By the end of the meeting they'll have written down specific improvements that they can make. It's timeboxed to three hours for a 30-day sprint.

The sprint is over ***when its timebox expires***.

there are no Dumb Questions

Q: Wait, is that all there is to Scrum?

A: That's all of the Scrum rules, but that's definitely not all there is to Scrum. Scrum was designed to be lightweight and simple to understand. But mastering Scrum takes a lot more than just following the rules. You learned in the last chapter how the values of the agile manifesto can make a big difference in how a team uses a practice. That same idea applies to Scrum: **mindset and experience** make the difference between a team with an empty "just following the rules" approach and an effective Scrum team that really "gets" it.

Q: I already break my projects into phases. Aren't sprints the same thing?

A: Not at all. Traditional waterfall projects are often broken into phases, with a complete deliverable at the end of each phase. But those phases are still planned at the beginning of the project. And if there's a change discovered that will affect the next phase, the project needs to be replanned, which usually involves a separate change control process. In other words, the team basically assumes the project plan is mostly correct, and that it's the project manager's job to deal with the relatively few changes that happen along the way.

Scrum is different because it's **iterative**, which is a lot more than just breaking a project down into phases. The team doesn't even plan the next iteration until the current one is complete. The team might discover changes partway through the sprint that affect the next one, or which may even affect the current one. That's why the Product Owner is such an important member of the team: he or she has the authority to make decisions on behalf of the business or customers about what features the team will build, which gives them the power to make those changes immediately.

Q: What's the difference between the product backlog and the sprint backlog?

A: The product backlog contains a list of everything that might be needed in the product. Scrum teams are usually working on ongoing releases of a single product, so the product backlog is never complete. The first version they release typically contains the requirements that are best understood by everyone. As the project goes on, the Product Owner will add and remove items based on what's valuable to the company.

The sprint backlog contains the specific items that the team will build during the sprint, which were moved from the product backlog during sprint planning. The software that the team finishes during the sprint is the **increment**: the team delivers and reviews a complete increment in each sprint. The sprint backlog *also* contains a **plan for delivering the increment**. Team builds that plan during sprint planning by decomposing backlog items into tasks.

Q: What exactly *is* a backlog item?

A: Every item in the product backlog consists of a brief description, a (usually rough) estimate of how long it will take, the business value, and an order. The Product Owner will routinely **groom the product backlog**. This means going through the items in the backlog, removing ones that are no longer useful, re-evaluating the value of each item, and updating the order so the most valuable ones are first.

Q: Wait a minute—in the story, Brian is a team lead. But there are only three roles in Scrum, and "team lead" isn't one of them. What's going on there?

A: Roles and jobs aren't the same thing. As far as Scrum is concerned, Brian is just another member of the development team. His job in the company is team lead, which gives him more authority than the other the developers. Plus, he has skills and his own areas of expertise. Scrum may not have a distinct role for him to play, but *he still plays an important and unique role on the team*. It's just that there are no Scrum-related events or artifacts that are specific to him.

Q: What do you mean by "artifacts"?

A: An artifact is just a by-product specific to a process or methodology. Scrum has **three artifacts**: the product backlog, the sprint backlog, and the increment.

Q: What happens if we get partway through a sprint and discover there's some sort of emergency situation? Do we still have to wait for the timebox to expire before the sprint can end?

A: On ***very rare occasions*** the Product Owner has the authority to cancel the sprint before the timebox is over. When he or she does this, a sprint review is held for any sprint backlog items that are done, and all other items are put back on the product backlog and left for the next sprint planning session. *Be really, really careful about cancelling a sprint.* It's an in-case-of-emergency "break glass" action because it can waste a lot of the team's energy—and more importantly, it causes people in the company to distrust the team, and to lose confidence in the effectiveness of Scrum.

Scrum was designed to be lightweight and simple to understand, but mastering Scrum takes a lot more than just following the rules.

Sharpen your pencil

Write down the name of each of the Scrum events, when the event happens, and the length of the event's timebox. We filled in the first event. Then write down the three Scrum roles and the three Scrum artifacts.

Event names in the order they occur	When the events happen	Length of the event's timebox
sprint		*Assume that the sprint is timeboxed to 30 days*

Write down the Scrum roles

Write down the Scrum artifacts

Answers on page 42

FOUR MONTHS LATER AT A SPRINT RETROSPECTIVE...

I DON'T THINK THIS SPRINT WENT SO WELL.

WHAT TIPPED YOU OFF, SHERLOCK? WAS IT THE FACT THAT WE **BARELY GOT ANYTHING DONE?**

Rick: Hey, there's no need for insults. We're all doing our best, here!

Amy: Sorry. Things have just been really tense with the business lately, and it's got me on edge. I spend so much time with Scrum that I don't have time to do my job.

Rick: What do you mean?

Amy: I mean, you guys constantly ask me to make decisions. Like when we were planning this sprint, and I had to decide whether Brian's team should start the strategic silo assault level or improve the milk grenade mechanics.

Rick: Yeah. You had us get started on the strategic silo assault. Was that wrong?

Amy: Oh man, you have no idea! After the demo I got hauled into the CEO's office and screamed at for an hour. Gamers hated the strategic levels in CGW4, and the last thing our stakeholders want is to see bad reviews for them in CGW5.

Rick: Wait, what? We worked super hard on those. I thought they came out well!

Amy: Me too. But they said that I had no business making the decision to include any strategic levels in the sprint—or any other decision like that.

Rick: But you're the Product Owner! Making those decisions part of your job now.

Amy: Exactly. So I have no idea what to do. Plus, I spend so much time answering the team's questions about features that I barely have time for my real job.

Rick: Well, it's no easier for me. It's getting harder and harder to drag Brian's team members into my Daily Scrums—you know how programmers hate meetings.

Amy: You know what? We're following the Scrum rules, it's kind of helping. I think. Right? Maybe? Um... are we still sure this Scrum stuff is actually worth the trouble?

BRAIN POWER

The team followed the Scrum rules to the letter, but the project still ran into trouble. What went wrong?

The Scrum values make the team more effective

We already saw how agile teams are much more effective when everyone on the team has a mindset that's driven by the values in the Agile Manifesto. Scrum comes with its own set of five **values** that do exactly the same thing for Scrum teams.

Each of the five Scrum values is represented by one word. In this case, the value is "openness".

Openness

You always know what your team members are working on, and you're comfortable that they know what you're working on. If you run into a problem or an obstacle, you can bring it up with the team.

It's not always easy to talk to your teammates when you run into problems. None of us like making mistakes, especially at work. That's why everyone on the team needs to share this value: it's a lot easier to be open with teammates about problems and roadblocks that you run into if they're just as open with you about theirs—and that helps the whole team.

I'M REALLY SORRY, I KNOW I PROMISED I'D BE DONE WITH THE CODE FOR THE SILO SNIPER LEVEL BY NOW, BUT I JUST CAN'T GET THE LOGIC RIGHT.

It was uncomfortable and a little embarrassing for Brian to be open about this roadblock, but it's what was best for the project.

I WON'T LIE, THAT'S GOING TO SET US BACK. BUT WE'LL FIND A WAY TO GET YOU THE TIME YOU NEED.

Respect

You and your teammates have mutual respect for each other, and every person on your team trusts everyone else to do their jobs.

Trust and respect go hand-in-hand. People on Scrum teams listen to each other, and when they disagree they take the time to understand each others' ideas. It's natural for people on teams to disagree on an approach. On an effective Scrum team, you'll listen when your teammates disagree with the approach you're taking, but in the end they'll respect your decisions, and give you the benefit of the doubt if you take an approach they disagree with.

Trust doesn't always come easy to a traditional waterfall team where the project manager does the planning by demanding estimates from the team members. If things go wrong, the project manager can blame the team for underestimating, and the team can blame the project manager for a bad plan.

Courage

Scrum teams have the courage to take on challenges. Individual people on the team have the courage to stand up for their project.

What do you do when the boss demands that you and your team meet an impossible goal? What if he demands that they meet a two-week deadline for a project that can't possibly be done in less than two months? Effective Scrum teams have the courage to stand up and say "no" to impossible goals because they have the planning tools to show what is possible, and users and stakeholders who trust them to deliver the most valuable software that they can.

Focus

This is why the team writes a one- or two-sentence sprint goal at the beginning of the sprint planning meeting. It helps keep them focused during the sprint.

Every team member is focused on the sprint goal, and everything they do in the sprint helps move them towards it. During the sprint, everyone works only on sprint tasks, and does one task at a time until the sprint is done.

During a Scrum sprint, every single person on a Scrum team is focused exclusively on items in the sprint backlog and tasks that they decomposed them into during the planning meeting. Each person works on one item in the sprint backlog at a time, focuses exclusively on one task in the plan, and moves on to the next task only when the current one is done.

WHAT ABOUT *MULTITASKING*? AREN'T TEAMS MORE EFFECTIVE WHEN PEOPLE DO MULTIPLE TASKS AT THE SAME TIME?

People on Scrum teams know that focusing on one task at a time is more effective than trying to multitask.

There's a myth that switching back and forth between multiple tasks many times a day is more effective than concentrating on one at a time. Think about it this way: let's say you have two tasks that will each take one week. If you try to multitask and do them at the same time, the best you'll do is deliver them both at the end of two weeks. But if you do don't start the second task until the first one is done, you'll finish the first task a week earlier.

There's one more Scrum value. Flip the page... →

Commitment

Every person on the team is committed to delivering the most valuable product that they can—and the rest of the company is committed, too.

When the team is committed to the project, it means that the tasks that they planned out to meet the sprint goal are the most important tasks they have to work on. Each team member feels that his or her personal success at the company is tied to the success of the project. Not only that, but every single person on the team them *feels committed to the each item in the increment*, not just the items that they're working on. This is called **collective commitment**.

But what happens if something comes up that's important to the company, but not part of the project? For Scrum to be effective, the boss needs to keep that from happening. In other words, ***the company needs to be fully committed to the project,*** to respecting the team's collective commitment to the sprint goal, and to following the rules of Scrum.

So how does a company express that kind of commitment?

By **giving the team the authority** to determine what features are going to be developed during each sprint, and trusting that's how the team will deliver the most valuable software possible. The company does that by assigning a full-time Product Owner to the team with the authority to decide what features will be built and accept them as done.

Collective commitment means that every team member feels committed to delivering the complete increment, and not just the items he or she is working on.

Story Time

Once upon a time there lived a pig and a chicken who were the very best of friends.

One day the chicken said to the pig, "I have a great idea. Let's open a restaurant together!"

The pig said, "That is a great idea! What should we call it?"

The chicken replied, "How about **Bacon 'n Eggs**?"

The pig thought about it for a minute. Then he said, "You know what? Never mind. You're just involved, but I'm committed."

there are no Dumb Questions

Q: Uh... why are you talking about farm animals?

A: Because the fable of the pig and the chicken is a good way to understand commitment (because the chicken just lays eggs, but the pig has to let himself get eaten—a much greater commitment). In fact, a lot of Scrum teams actually refer to the Scrum team members who are truly committed to the project as "pigs," and those who have an interest but not a true commitment as "chickens." (Sometimes they even call themselves pigs in cultures where "pig" is used as a severe insult!)

The Scrum value of commitment means that when you're on a Scrum team, you genuinely believe that your professional success or failure rests on your ability to deliver valuable software, and that makes you a committed "pig." Your users and stakeholders may have a big interest in the project getting done, which makes them "chickens"—they're very important to the project, but they don't feel the same sense of strong personal commitment as "pigs" do.

Q: What if the team doesn't really "get" some of the Scrum values?

A: An important part of the Scrum Master's job is to help everyone on the team to understand the Scrum values, and to eventually internalize them so that each person has the right mindset for Scrum. Very few teams start out Scrum genuinely believing in all of these values at first. They grow and evolve together over time. The Scrum Master helps by using the Scrum values to help the team understand and deal with project obstacles that come up.

Q: What if my team can't find a Product Owner with enough authority?

A: If the Product Owner doesn't have the authority to decide what the team will build, or to accept each item in the sprint backlog as done, then the company hasn't given the team the authority to do their jobs, and they haven't truly committed to the project—or to Scrum. That can be trouble.

A lot of projects fail because the team **did a great job building the wrong software**. The Product Owner keeps that from happening because his or her ***full-time job*** is to work with the rest of the company and understand what the business needs, decide what features will be built, and help the team to understand those features.

If the Product Owner doesn't have the authority to decide what features the team builds, then the company isn't really committed to delivering the project with Scrum.

BULLET POINTS

- **Scrum** is the most popular agile methodology, and is most successful when used by software teams of at least three and up to nine people
- Scrum includes five timeboxed **events**: the sprint, sprint planning, the Daily Scrum, the sprint review, and the sprint retrospective
- Scrum has three **roles**: the Product Owner, the Scrum Master, and the development team
- Scrum uses three **artifacts**: the product backlog, the sprint backlog, and the increment
- Projects are divided into **sprints**, iterations that are typically timeboxed to 30 days (but can be shorter)
- Each sprint starts with **sprint planning**, a timeboxed meeting in which the team decides which items (such as features) to include in the sprint backlog, and decomposes them into tasks for at least the first week
- The **Daily Scrum** is a meeting where each team member talks about what they've done, what they'll work on next, and if they see any roadblocks ahead
- The Product Owner invites key stakeholders to the **sprint review** where the team demonstrates working software and discusses the next sprint backlog
- At the **sprint retrospective** the team talks about what went well and identifies specific things that can be improved
- Scrum teams have five **values** that help them get into a more effective mindset: openness, respect, courage, focus, and commitment

I'VE GOT GREAT NEWS! I MET WITH A LOT OF PEOPLE ON THE BUSINESS SIDE ABOUT OUR COMMITMENT PROBLEMS, AND THEY AGREED TO ASSIGN ALEX TO OUR TEAM ***AS A FULL-TIME PRODUCT OWNER.***

Alex is a senior manager who's been in the gaming industry for a long time. He meets with users, advertisers, and game reviewers. He knows all about what makes games sell.

GOOD TO MEET YOU GUYS. RICK EXPLAINED HOW SCRUM WORKS, AND THE BOSSES ALL THINK SCRUM IS ***SO IMPORTANT*** THAT THEY ASSIGNED ME TO WORK FULL TIME ON IT.

THERE'S ***NO QUESTION ABOUT AUTHORITY.*** IF I SAY IT GOES INTO THE GAME, IT GOES INTO THE GAME. THE OTHER SENIOR MANAGERS TRUST MY JUDGMENT, AND I TRUST YOU GUYS TO GET THE PROJECT DONE.

PRODUCT OWNER

THAT'S A HUGE RELIEF! NOW I CAN GO BACK TO "JUST" BEING CREATIVE DIRECTOR.

THE DEVELOPERS HAVE A LOT OF QUESTIONS FOR YOU! I'LL INTRODUCE YOU TO THEM.

With a new Product Owner, the team should be able to figure out the most valuable features to include in the next sprint.

Here are some things we overheard Amy, Rick, and Brian saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the Scrum value that it's either compatible or incompatible with.

IT'S MY TURN TO TALK? OK. SINCE THE LAST DAILY SCRUM I KEPT WORKING ON THE SAME FEATURE, AND I'LL DO THE SAME UNTIL THE NEXT SPRINT. NO ROADBLOCKS. WHO'S NEXT?

COMPATIBLE

INCOMPATIBLE

Courage

LOOK, ALEX, I KNOW REDOING ALL OF THE GRAPHICS WILL IMPRESS REVIEWERS, BUT THERE'S ABSOLUTELY NO WAY THE TEAM CAN DO THAT WITHOUT MISSING THE RELEASE DATE.

COMPATIBLE

INCOMPATIBLE

Focus

THE ONLY THING I'M IMPRESSED WITH IS TECHNICAL ABILITY. IF YOU CAN'T CODE, I HAVE NO USE FOR YOU.

COMPATIBLE

INCOMPATIBLE

Openness

I CAN'T WORK ON ANY SPRINT TASKS TODAY. ANOTHER TEAM HAS A REALLY BIG DEADLINE AND THEY NEED TO BORROW ME.

COMPATIBLE

INCOMPATIBLE

Respect

Answers on page 43

Rick: I... uh...

Alex: Yeah, just what I thought.

Rick: Hey, that's not fair. I know he's been working really hard on that feature. It's just taking a lot longer than any of us thought it would.

Alex: So what are you going to do about it?

Rick: Hey, we're all on the same team here. And that includes you, Alex. So I think you meant to ask what are **we** going to do about it?

Alex: Okay, fine. Well, since I'm on the team, let me give you an answer. Other teams build contingency and padding into their schedules so *they never have to tell a senior manager like me* that they're running late.

Is padding the schedule really any different than lying about the schedule or hiding information about it from the boss, except that everyone's decided that it's okay?

Rick: Sure. Yeah. I've done that before on other projects I've managed. I'd add extra tasks to my schedule to account for things taking a lot longer than we anticipated.

Alex: But you're not doing that now?

Rick: Well, no. The Scrum rules don't really give me a way to add contingency, padding, or extra tasks. I don't really see any way to pad the schedule without breaking the Scrum rules.

Alex: Then maybe there's something wrong with Scrum.

BRAIN POWER

What do you do when one of the tasks in the sprint takes a lot longer than the team planned for it to take?

Question Clinic: The "which-comes-next" question

A LOT OF PRACTICES OR EVENTS HAPPEN IN A SPECIFIC ORDER, AND YOU'LL BE ASKED ABOUT THAT ORDER IN A "WHICH-COMES-NEXT" QUESTION. THESE ARE QUESTIONS THAT QUIZ YOU ON HOW PRACTICES FIT TOGETHER IN A METHODOLOGY. THESE QUESTIONS AREN'T USUALLY VERY DIFFICULT, BUT THEY CAN BE A LITTLE MISLEADING.

Most which-comes-next questions describe a situation and ask you what you'd do next. Sometimes it's not immediately obvious that it's asking about the order of events. Watch for any question that lay out a scenario and then asks what comes next, what happens afterwards, or how the team should proceed.

Don't be thrown if the question asks about an industry you don't know much about.

27. You're the Scrum Master for an automotive industry software team working on firmware for an antilock brake system. Your team just finished writing down the objective that will be met when the team completes the sprint items. What is the next thing your team should do?

A. Decompose the sprint backlog items into tasks

B. Review the working software with the users

C. Meet with the business users

D. Decide what items to include in the sprint backlog

The key to a "which-comes-next" question is figuring out what the team is currently doing. So what's another word for "objective that will be met"? That's the definition of the sprint goal! So this must be happening at the beginning of the sprint planning session.

This also happens during the sprint planning session. But when you compare this with the other answers, there's another one that happens before it.

Neither of these answers are things that happen during the sprint planning session.

Aha! If the team is holding a sprint planning session and just finished writing the sprint goal, the next thing they'll do is decide what items from the product backlog will be included in the sprint backlog. This is what comes next!

HEAD LIBS

Fill in the blanks to come up with your own "which-comes-next" question! **Start** by thinking of a Scrum event or activity to be the correct answer, and **then** figure out exactly what the team just finished doing before it—that's what you'll describe in the question!

You're a Product Owner on a _____ project. Your team just finished
(an industry)

_____. A _____ just informed you that
(description of a Scrum activity) (a type of user)

your project _____.
(a problem that came up on the project)

The last part of this question doesn't change the answer at all. A lot of questions are like that.

Which of the following does the team do next?

A. _____
(the correct answer—description of the Scrum activity, tool, or practice that comes next)

B. _____
(description of a different Scrum activity, tool, or practice)

C. _____
(the name of activity, tool, or practice that's part of a different methodology)

D. _____
(description of one of the Scrum values or roles)

THE WHICH-COMES-NEXT QUESTION DOESN'T ALWAYS LOOK LIKE IT'S ASKING ABOUT THE ORDER OF EVENTS, TOOLS, OR PRACTICES! KEEP AN EYE OUT FOR QUESTIONS THAT DESCRIBE SPECIFIC ARTIFACTS THAT GET CREATED OR ACTIONS YOU'LL PERFORM AND THEN ASK YOU WHAT YOU'RE SUPPOSED TO DO NEXT.

A task isn't done until it's "Done" done

When you're on a Scrum team, you work on one backlog item or task at a time—that's what the value of Focus is all about—and you work on it until it's done. But when, exactly, is it done? Is there still some testing left to do? It's easy to think that you're done... except for this one teeny little thing. That's why Scrum teams have a **definition of done** for every item or feature that they add to the backlog. Before an item can go into the sprint backlog, everyone on the team needs to understand and agree about exactly what it means for it to be not just done, but "Done" done. And since every item in the backlog has a definition of done, the ***entire increment has a definition of done***, and the team is committed to delivering a "done" increment at the end of the sprint.

Under the Hood: Sprint Planning

Sprint planning relies on the definition of done

During the first part of the sprint planning meeting, the team decides which items to include in the sprint backlog. But what happens if there's an item that doesn't *really* have a definition of done that everyone on the team understands? Say one person thinks that it means all the code is done, but someone else assumes that it also includes documentation or testing. Even if they don't realize that there's a disagreement about what it means for that item to be done, they'll discover that they can't come to a real agreement on how many other items they can deliver in the sprint. That's why sprint planning only works when every item has a clear definition of done that the whole team can agree on.

Scrum teams adapt to changes throughout the sprint

Project teams have to make decisions every day: Which features will we build in this sprint? What order will we build them in? How will users interact with this feature? What technical approach will we take? Traditional waterfall teams have one answer: all of the planning is done at the beginning of the project. The problem is that at the time the plan is being built, most of those questions don't have answers yet. So the project manager works with the team to make assumptions, and relies on a change control process to change the plan when they guessed incorrectly.

Scrum teams *reject the idea* that every project question can be answered at the beginning of the project, or even at the beginning of a sprint. Instead, they make decisions based on real information as soon as it's discovered. They do this using the **transparency-inspection-adaptation cycle**:

- The cycle starts with **transparency**, where the team decides together what items to include in the sprint, and what it means for each of them to be done. All work done by everyone is visible to the team all the time.
- The whole team meets every day in the Daily Scrum to **inspect** each item that's being worked on.
- If they discover changes, they **adapt** to them (like adding or removing items from the sprint backlog when roadblocks happen).
- The next day **the cycle begins again**, with team members giving complete transparency into what they're doing at the Daily Scrum. This cycle continues every day until the timebox expires and the sprint is complete.
- The team **also inspects and adapts at the other Scrum events**—sprint planning, the sprint review, and the sprint retrospective—by reviewing and modifying the sprint goal, items, tasks, and the way they work.

---

there are no Dumb Questions

Q: This is starting to sound *really* theoretical. Can you tie it back to the real world?

A: Sure. "Transparency" just means that every single person understands all of the features they're building in the sprint, and are open about the work they're doing, what they plan to do next, and what problems they've run into. "Inspection" means that they constantly make sure that knowledge is up to date using the Scrum events (*especially* the Daily Scrum). And "adaptation" means that they're constantly finding ways to change what they plan to do next based on this new information.

Q: If the team plans some of the tasks but not all of them, won't that just cause chaos later on in the project?

A: No. Making all of the decisions at the beginning of the project lets you feel like everything's under control. But very often they aren't, leaving you surprised that somehow your project—which seemed fine yesterday—is suddenly late, and now everyone's in crunch mode. That's why so many teams turn to agile: because their traditionally planned projects keep blowing their schedules, and *something* needs to change. Scrum avoids those pitfalls by recognizing that many important decisions depend on information that **won't be known** until partway through the project.

Q: Can you give me an example of how that would work in real life?

A: Here's one that our Cows Gone Wild team might face. Brian needs to program the behavior for a new tractor vehicle for the player to use, but he can't start until Amy finishes the basic behavior for it. If Rick used traditional project management, he would either have to assume that Amy will finish first (and add padding to the schedule in case she takes too long), or he'd have Brian work on something else first, even if the other task is less important.

Now that they're using Scrum, the team can give themselves more options. They know Brian can't start the coding for the tractor until Amy finishes designing its behavior. But they also know they'll **constantly inspect their progress and adapt the plan along the way**. So instead of deciding *today* whether Brian work on the tractor or on a less important task, they can add ***both*** of them to the sprint backlog—and they'll **put off the decision** until Brian is ready to start the code. If Amy is done with her work, he can start coding. If she's not, he'll pull the other task off of the sprint backlog and come back to the tractor coding when he's done (but only when he's *"Done"* done!).

I STILL DON'T SEE HOW SPRINT PLANNING CAN POSSIBLY WORK. HOW CAN YOU HAVE A TIMEBOXED PLANNING SESSION? YOU SAID THAT THE TIMEBOX OFTEN EXPIRES BEFORE ALL OF THE WORK IS PLANNED. DOESN'T A HALF-BAKED PLAN LEAD TO HALF-FINISHED PROJECTS?

No—because agile teams make decisions as late as possible.

When a lot of people learn about Scrum, they're surprised that the sprint planning session is timeboxed to eight hours for a 30-day sprint (and proportionally less time for shorter sprints), because they're used to having every single task in the project completely planned out before any of the work can begin. But we've already seen that teams following a traditional waterfall process often have trouble when their plans change partway through the project. In fact, changes that would deliver more value often get rejected during the change control process simply because replanning a months-long schedule is too much work.

Scrum teams rarely (if ever!) run into this problem because they don't plan every single task at the very beginning of the project. In fact, usually they don't even plan out all of the tasks at the beginning of the sprint. Instead of planning everything up front, they make decisions at the **last responsible moment**. What that means is that they only need to do the planning that's absolutely necessary to get started with the sprint. If more planning needs to be done, it can be done later in the sprint.

This is a new way of thinking about planning for a lot of teams. Luckily, we have the agile manifesto to help our teams get into a mindset that's really effective.

MINI Sharpen your pencil

The Agile Manifesto is really useful because it helps get into a mindset where concepts like the last responsible moment *really make sense*. One of the twelve Agile Manifesto principles is especially helpful for understanding the last responsible moment. Write down which one you think it is—you can see our answer on page 28.

The Daily Scrum Way Up Close

The "ceremony"

Even though a lot of teams refer to answering the Daily Scrum questions as a "ceremony," every single person is engaged and paying attention.

The whole team gets together at the same time every day, and most teams start with a different person each time. Everyone (including the Product Owner and Scrum Master) answers three questions:

- What did I do **yesterday** to move us towards the sprint goal?
- What will I do **today** to help meet the sprint goal?
- Are there any **impediments** that prevent me from meeting the sprint goal?

Each person's answers are brief and to-the-point because the meeting is timeboxed to 15 minutes.

Inspect and adapt

The reason that each person answers those three questions is to give complete **transparency** into what he or she is doing. But this only works if every other person on the team is listening carefully (which is why it's important to keep the updates brief!). One of the most common things that happens during a Daily Scrum is that one team member realizes his or her teammate is going to do something that doesn't quite make sense: maybe they're starting one sprint backlog item when a different one is more valuable, or taking one approach when there's a better alternative, or has hit a roadblock that other team members can help with.

When this happens, they'll set up a meeting later in the day to talk about it. More often than not, they'll have a discussion that causes them to change the plan: they may take a different approach, or choose another item from the sprint backlog, or have to add more work (and possibly remove a different item from the sprint backlog) to get around the roadblock. This is how the team **adapts** to the change.

THIS IS THE TRANSPARENCY-INSPECTION-ADAPTATION CYCLE, RIGHT? ANSWERING THE QUESTIONS IS THE ***INSPECTION*** PART, AND WHEN THEY MAKE CHANGES THAT'S THE ***ADAPTATION*** PART.

That's right. And it all works because of transparency.

Each team member answers the questions every day, so everyone has current, accurate information about the project. Not to get too theoretical, but this is actually a really good example of **empirical process control theory**, which tells us that when a process (in this case, a software methodology) is based on empiricism, it lets the team optimize the way they work to reduce risk and give them sustainable, predictable results.

DICTIONARY DEFINITION

em-pir-i-cism, noun

the theory that knowledge comes from experience, and that decisions are to be made based on information that is known

The team was driven by **empiricism**, *and rejected the project plan based on guesswork.*

The agile manifesto helps you really "get" Scrum

We learned back in Chapter 2 that the most effective way to approach any agile methodology like Scrum is with a mindset that's driven by the values and principles of the agile manifesto. Let's take a closer look at **three of those principles** that are especially helpful for Scrum teams.

THERE'S NO WAY I CAN ***ACCEPT*** THE HAY THRESHER BATTLE AS ***DONE*** UNTIL YOU GUYS GET THAT MILK GUN WORKING.

> Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

A really important word in this principle is "valuable." Everyone on the team takes it really seriously, and does their best to deliver the most value that they can.

The Product Owner makes sure the team delivers value

That's the whole reason that the team needs a Product Owner with the authority to accept items in the sprint backlog as Done—or, if they're not "Done" done, refuse to accept them. If everyone on the Cows Gone Wild 5 team really "gets" this principle, then they won't be mad at Alex for not accepting this feature as done yet, because they genuinely want to deliver the most **value** that they can. They want to deliver it **early** and **continuously**, and the best way to do that is to finish the current item (which means it's "Done" and ready to demo to the users in the sprint review) so that they can move on to the next one and get as many backlog items done in the sprint as they can.

Under the Hood: The Sprint Review

The sprint review is all about maximizing value

During the sprint review, the team works with key users and stakeholders invited by the Product Owner to inspect the increment and the product backlog, and everyone understands the goal is to maximize value by reviewing the value delivered during the sprint, and maximizing the value of the items for the next sprint. Here's how they do it:

- The Product Owner goes over what was "Done" in the sprint and the team demonstrates the working software.
- The team talks about what went well and what could be improved, and answers user and stakeholder questions.
- The Product Owner walks everyone through the current product backlog, and everyone collaborates on what they think should probably go into the next sprint, so the team learns directly from the users and stakeholders **what they think is going to be most valuable**. This is really important for planning the next sprint.
- They'll have an open and honest discussion of anything that's changed in the marketplace (in case it changes the most valuable thing to do next), the company's timeline and budget, and anything else that's relevant.

Here's Amy's update at the Daily Scrum.

I FINISHED THE CHICKEN COOP AIR ASSAULT ARTWORK, SO THAT'S ANOTHER SPRINT BACKLOG ITEM DONE. THERE ARE A FEW ITEMS I CAN CHOOSE TO DO NEXT. I WAS THINKING I'D WORK ON THE HORSE CORRAL BATTLE LEVEL DESIGN.

I'M NOT SURE THAT'S THE RIGHT THING TO WORK ON NEXT. CAN WE MEET RIGHT AFTER THIS DAILY SCRUM IS OVER AND TALK IT THROUGH?

Sounds like Amy may be going down the wrong track. If Brian hadn't been paying attention, he might not have caught this the potential problem. Now they can figure it out together.

The best architectures, requirements, and designs emerge from self-organizing teams.

Self-organizing means deciding as a team what to work on next

If you've worked on a traditional waterfall team with a project manager, then you've probably worked on tasks that were part of a project plan created by a project manager and assigned to you by either the project manager or your boss. But that's not how Scrum teams work. Scrum teams are **self-organizing**, which is a new way to work for a lot of people.

The whole Scrum team works together to plan the project. There isn't a single person building a plan and telling them what to do. There ***is*** one person—the Product Owner—who decides what the team will deliver at the end of the sprint. But within those constraints, the whole team decides how they'll meet those goals.

But self-organization doesn't just happen during sprint planning. They ***constantly adapt their plan*** by checking in with each other at the Daily Scrum: each person tells the whole team what they're planning on doing next to meet the goals they decided on during sprint planning. If a teammate sees a problem with the approach, they'll work together that day to fix it.

The Daily Scrum is timeboxed, so when two team members discover an issue like this, they'll meet afterwards to work it out (inviting other team members to join if they have input). Brian and Amy will meet and talk about how to handle this situation. Once they've got a new plan, they'll review it at the next Daily Scrum to make sure everyone is on board, and to see if anyone has anything to add.

What did you get for your MINI Sharpen your pencil answer on page 24? Flip the page to find out ours...

MINI Sharpen your pencil Solution

Did you come up with a different answer? This is the agile principle we think is most helpful for understanding the last responsible moment, but other principles can shed light on it too!

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Making decisions at the last responsible moment probably sounds really weird if you've never done it before. But to someone who *genuinely welcomes changing requirements*, it really does feel as normal as riding a bike.

This is a new way of thinking about planning for a lot of teams.

That's part of the mindset shift that we talked about in the last chapter. Here's how it works on a Scrum project in the real world:

- There's **just enough** written down about each item in the product backlog to be able to start sprint planning.
- During sprint planning, the team decomposes enough of the sprint backlog to get everyone started, but they **don't feel like they need to decompose everything**.
- Self-organizing teams don't need to decide the exact order of every single task in painstaking detail when they're planning a sprint. They **trust themselves** to make good decisions when the time comes.
- As the team works through the sprint backlog over the course of the sprint, they discover new tasks and changes and bring them up in the Daily Scrum, and they use that information to work together to ***create a plan* for the next 24 hours**.
- They're **constantly inspecting and adapting**, so they trust themselves to make good decisions in the future.

Scrum teams make decisions at the last responsible moment. They only make project decisions that need to be made now, and leave the rest for later.

THE MAD COW ZOMBIE LEVEL WE BUILT JUST ISN'T AS FUN AS WE THOUGHT IT WOULD BE, AND GAMERS WON'T LIKE IT. LET'S PUSH IT BACK TO THE PRODUCT BACKLOG. MAYBE WE'LL PUT IT OUT IT AS DOWNLOADABLE CONTENT AFTER THE GAME'S RELEASED.

It's a good thing Alex is comfortable making decisions at the last responsible moment! In this case, the only way the team could find out the feature wouldn't be fun (and deliver the value the project needs from it) was to build it. Notice how he's staying positive, not blaming anyone for wasting effort, and keeping the option open to release it later.

Watch it!

The Daily Scrum is timeboxed to 15 minutes, so make sure everyone keeps their updates focused and to the point.

That's easier said than done. Once you and your team start holding a Daily Scrum every day, you'll discover that some people are really uncomfortable talking about their work in front of everyone else on the team, while others just can't seem to stop talking, and will take up the entire 15 minute timebox if you let them.

This is why it's really important that the Scrum Master takes his or her role seriously—especially the responsibility of making sure that the team understands and adheres to the Scrum rules:

- *If someone is reluctant to speak up, the Scrum Master can help that team member to accept the Scrum value of openness and understand how transparency is critical for making Scrum work.*
- *When team members give updates that take up too much of the Daily Scrum, the Scrum Master can show them which facts are relevant and help them manage their Daily Scrum time better.*
- *If a team member gives an update that goes beyond just answering the three questions and raises issues for discussion, the Scrum Master can remind the team to set up a separate meeting for some of the team members to discuss the issue and report back to the whole group.*

BULLET POINTS

- The team decides on the **definition of done** for every item in the sprint backlog during sprint planning.
- The Product Owner doesn't accept an item to include in the sprint review until it's **"Done"** (i.e. it meets the definition of done that the team decided on for it).
- Scrum uses **empirical process control**, which includes a *cycle of transparency, inspection, and adaptation* so that they make decisions based on real, known facts.
- **Transparency** (or visibility) means everyone on the team understands what their teammates are doing.
- **Inspection** means they frequently check with each other on what they're building and how they're building it at the Daily Scrum and the other sprint events.
- The team constantly **adapts** their plan based on what they learn from that inspection.
- Agile teams make decisions at the **last responsible moment**, and plan only those tasks that absolutely need to be planned right now.
- The team delivers value **early** (by working on each backlog item until it's done) and **continuously** (by delivering a complete increment that's "Done" at each sprint review).
- Scrum teams are **self-organizing**: they decide as a team how to meet their goals and who will do the work.
- Scrum teams **welcome changing requirements** more easily than traditional waterfall teams because they're self-organizing and make decisions as late as possible.

Who am I?

A bunch of Scrum artifacts, events, and roles are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down its name, and what kind of thing it is (like whether it's an event, role, etc.).

And watch out—another Scrum concept that's not an event, artifact, or role might just show up and crash the party!

	Name	Kind of thing
I'm a servant leader who guides the team in understanding and implementing Scrum, and helps people outside of the team understand it.	_____	_____
I'm held at the end of the sprint to do an inspection of each item that the team built with users and stakeholders who were invited.	_____	_____
I'm how the team inspects itself, where they look for the things that went well, and make a plan to improve things that didn't.	_____	_____
I'm the sum of all items that the team delivers to the users at the end of the sprint, and I can only be delivered if every item in me is "Done."	_____	_____
I'm the group of professionals who actually do all of the work that's needed to deliver the software to the users and stakeholders.	_____	_____
I'm responsible for deciding what items will go into the product, and I have the authority to accept them as "Done" on behalf of the company.	_____	_____
I'm a 15-minute timeboxed meeting held every day, where team members create a plan for the next 24 hours.	_____	_____
I'm what the Product Owner helps the team optimize and maximize, and the team tries to prioritize the items that have the most of me.	_____	_____
I'm the set of items the team will build during the sprint, along with a plan to build them (usually a set of tasks the items are decomposed into).	_____	_____
I'm a timeboxed meeting where the team comes up with a sprint goal, decides which items they'll deliver, and decomposes them into tasks.	_____	_____
I'm an ordered list of all of the items (with descriptions, estimates, and values) that might be needed in the product at some time in the future.	_____	_____

→ Answers on page 44

there are no Dumb Questions

Q: Is the "Cows Gone Wild" team's story realistic? Can you really use Scrum for something like a video game, which requires a lot of creativity and on-the-fly—and sometimes last-minute—changes with heavy deadline pressure?

A: Not only can Scrum be used for a complex and dynamic project that constantly changes, it's actually better suited to an environment like that than a traditional waterfall process. Scrum teams constantly look for changes and find ways to adapt to them, which makes them much better at dealing with complexity and even chaos. And the timeboxed nature of sprints helps the team meet their deadlines. Alex just showed us a good example of the kind of decision that video game teams make in real life. The team built a feature but it turned out not to be fun enough in its current form, so they shelved it for now and might revisit it to sell later as downloadable content. Scrum gave the team the flexibility to handle this on the fly, while a traditional waterfall team would probably have to go through a lengthy change control process. More importantly, a Scrum team ***sees this change as a victory*** because they'll welcome any change that delivers more value to the product. A traditional waterfall team is likely to see it as a defeat because it "wasted" effort and required a change to the plan.

Q: Does "self-organizing team" mean that there's no boss?

A: Of course there's a boss. If you work in a company and you're not the CEO, then you have a boss. But an effective self-organizing team typically has a manager who doesn't micromanage, and who trusts all of his or her employees to deliver the most valuable software they can. Self-organizing teams are given the authority to decide which features to include in the software, usually by having a Product Owner assigned to the team who's senior enough to make those decisions. They're given the freedom to plan the work so that they can build those features in the way they feel is most effective. And they're given the flexibility to make decisions at the last responsible moment, because that's the most effective time to make important project decisions.

Q: What exactly happens during the sprint retrospective?

A: The sprint retrospective is how the team inspects the sprint that just ended and tries to find ways that they can improve. They look at all sorts of things: the people on the team can improve the processes and tools that they use to do the job, find ways to improve the quality of the software they're building, work on their relationships with others in the organization, and do anything else that might have an impact on the work—especially anything that can make their work more enjoyable or effective. By the time the sprint retrospective ends, the team has put together a plan for improvement. This plan typically consists of a small number of discrete and specific tasks that individual people on the team will carry out. Before the meeting, the Scrum Master helps everyone understand how it works, and makes sure that they all respect the timebox. This happens *before* the meeting because the Scrum Master and Product Owner have to participate in the retrospective as team members, offering their own opinions and ideas.

Q: Hold on—the Product Owner goes to the retrospective too? Does the Product Owner really need to be at *all* of the Scrum events?

A: Absolutely. The Product Owner is a real member of the team, and participates in the Scrum events just like everyone else. In fact, on a lot of Scrum teams the Product Owner does his or her share of the development work. But even in that case, that person still has the authority to make decisions about what items will go into the backlog and how to maximize the value of what the team delivers, and the company still respects his or her decisions.

Q: I'm having trouble finding anyone who has the clout to be the Product Owner but also has enough time to attend all of the Scrum events. Can the Product Owner be a committee instead?

A: Absolutely not. The Product Owner definitely **must** be a person. That person **must** have the authority to make the decisions about what goes into the software and what doesn't, and ***the Product Owner role must be his or her top priority***.

Modifying Scrum like this almost always renders it *much less effective*, usually by removing the parts that make its empirical process control work. Teams often try to "customize" Scrum by bending or breaking any rules that highlight a serious flaw in the way their team is structured. In this case, Scrum makes it really obvious that the team doesn't have the authority to decide what features go into the software. It's **scary** for a manager to say, "The team should build this feature, but not that one." The wrong decision that will cost the company a lot of money, and ***there will be a lot of blame*** if it goes wrong. That's why Scrum requires the team to have a Product Owner who has the authority to make that call.

Teams often try to "customize" Scrum when they don't have a true Product Owner who can make tough decisions about what to build.

Things are looking good for the team

Cows Gone Wild 5 is this year's most anticipated video game release!
Now they just have to get it out the door. (Easier said than done?)

GREAT NEWS! I DEMOED THE HAY THRESHER LEVEL AT THE WISCONSIN GAME CONFERENCE LAST WEEK AND ***WE TOTALLY KILLED IT!***

YEAH! I WAS JUST READING SOME GREAT REVIEWS ON THE TOP TWO GAMER BLOGS FROM PEOPLE WHO ATTENDED. LOOKS LIKE CGW5 IS THE ***MOST ANTICIPATED GAME OF THE YEAR!***

Scrumcross

Here's a great opportunity to seal the Scrum concepts, values, and ideas into your brain. See how many answers you can get without flipping back to the rest of the chapter.

Across

3. The Product Owner is allowed to _____ the sprint, but it wastes the team's energy and damages the company's trust in the team

5. The three Daily Scrum questions give the team complete _____ into what each team member is doing

6. When this is part of your mindset, you don't even consider trying to do two things at once

10. The sprint _____ contains items the team will build during the sprint

11. The most effective time to make a decision is at the last _____ moment

13. Product Owner, Scrum Master, and development team

15. The timeboxed iteration used in Scrum

17. The sprint _____ is an objective crafted by the team when they plan the sprint

18. During sprint _____ the team chooses what items to include in the sprint and builds a plan

19. Responsible for maximizing the value of the product and managing the Product Backlog

22. When this value isn't part of your mindset, you don't trust your teammates, and tend to blame them when things go wrong

23. The team decides how the work will be done during the _____ part of the sprint planning session

25. The theory that knowledge comes from experience

26. The product backlog, the sprint backlog, and the increment

28. The sprint, sprint planning, the Daily Scrum, the sprint review, and the sprint retrospective

29. What that pigs have, chickens don't, and everyone on a Scrum team feels collectively towards the whole project

30. What Product Owners routinely do to keep the product backlog current

Down

1. A self _____ team decides as a team how they'll meet their goals

2. The sprint _____ is how the team inspects itself and creates a plan to improve

4. Each item in the backlog has a description, the business value, an order, and an a rough _____

7. What's missing from the mindset when a team member is uncomfortable bringing up a roadblock or problem during the Daily Scrum

8. The team decides what can be done in the sprint during the _____ part of the sprint planning session

9. The _____ backlog is the single source of requirements and changes for the product

12. The sprint is over when the timebox _____

14. The Proudct Owner makes sure the team maximizes this

16. What the team does to turn sprint backlog items into tasks

20. The sprint _____ is how the team inspects the what they built and adapts the product backlog

21. The team inspects and _____ at each Scrum event by reviewing and modifying artifacts and the way they work

24. What teams do with working software at the sprint review

27. The sprint review is timeboxed to _____ hours for a 30-day sprint

Answers on page 45

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. The Scrum Master is responsible for all of the following except:

A. Helping the team understand what goes on during the Daily Scrum
B. Giving the Product Owner guidance in effectively managing the product backlog
C. Helping the team understand customer requirements
D. Giving the rest of the organization guidance on understanding Scrum and working with the team

2. Which of the following is NOT an attribute of a product backlog item?

A. Status
B. Value
C. Estimate
D. Order

3. Juliette is a Product Owner on a Scrum project in a health care organization. She was called into a meeting with a steering committee made up of her company's senior managers because she decided to include a planned health privacy feature in the most recent sprint. At the meeting, the senior managers told her in the future that she must consult with the whole committee before making business decisions like that.

Which of the following BEST describes Juliette's role?

A. She is in a servant-leadership role
B. She is not committed to the project
C. She needs to concentrate on focus and courage
D. She does not have the authority to adequately fill her Product Owner role

4. When is the increment considered done?

A. When the timebox expires
B. When every item to be delivered meets its definition of done
C. When the team holds the sprint review and demonstrates it to the users and stakeholders
D. When the team holds the sprint retrospective

Exam Questions

5. Which of the following is an example of collective commitment?

A. Everyone on the team feels personally responsible for delivering the entire increment, not just their individual parts of it
B. Everyone on the team always stays late and often works weekends
C. Everyone on the team is responsible for delivering an important part of the project
D. Everyone on the team participates in the sprint planning and retrospective meetings

6. Which of the following is NOT a Scrum event?

A. Sprint review
B. Product backlog
C. Retrospective
D. Daily Scrum

7. Amina is a Scrum Master on a team that is working on adopting Scrum. She wants to make a change to help her team get better at self-organizing. Which of the following is the best area to focus their improvement effort?

A. Daily Scrum
B. Sprint planning
C. Sprint retrospective
D. Product backlog

8. When is a Scrum sprint over?

A. When the team finishes the work
B. When the team completes the sprint retrospective
C. When the timebox expires
D. When the team completes the sprint review

Exam Questions

9. Each person on the team answers all of the following questions during the Daily Scrum except:

A. What roadblocks are in my way?
B. What planned work did I fail to accomplish?
C. What will I do between now and the next Daily Scrum?
D. What have I done since the last Daily Scrum?

10. Barry is a developer at an online retailer. His project manager told him the deadline for the current feature that he's working on is three weeks from now, even though Barry made it clear that he would need four weeks, and there were no specific deadlines or external pressures that require it to be done earlier than that. Barry's team is starting to adopt Scrum. Which of the Scrum values will make the team's Scrum adoption difficult or less effective?

A. Openness
B. Respect
C. Courage
D. Focus

11. Sandeep is a product owner on a Scrum team working on a telecommunications project. The business users let him know about a major regulatory change in one of his regular meetings with them. Handling this regulatory change is now a very high priority for the team, and will need to be the main objective of the next sprint.

Which of the following is used to describe the main objective of the next sprint?

A. The increment
B. The sprint backlog
C. The sprint goal
D. The sprint plan

12. What aspect of empirical process control theory involves frequently examining the different Scrum artifacts and making sure the team is still on track to meet the current goal?

A. Examination
B. Adaptation
C. Transparency
D. Inspection

Exam Questions

13. What is an increment in Scrum?

A. The items from the sprint backlog that the team actually completes during the sprint
B. The items from the product backlog that the team plans to complete during the sprint
C. The result of decomposing the sprint backlog items
D. A statement that describes the objective of the sprint

14. Which of the following helps Scrum teams focus?

A. Multitasking
B. Holding a Daily Scrum
C. Writing a sprint goal
D. Holding a retrospective

15. Danielle is a Product Owner on a Scrum team. She's talking to one of her business users, who gives her a new requirement. Which of the following should Danielle do next?

A. Update the product backlog
B. Hold a sprint planning session
C. Update the sprint backlog
D. Bring up the new requirement at the next Daily Scrum

16. Which of the following BEST describes how the team determines what work needs to be done during the sprint?

A. The Product Owner works with the business users to determine which items go into the product backlog
B. The team decomposes sprint backlog items into tasks
C. The team chooses which product backlog items to include in the sprint backlog
D. The team decides on the sprint goal

17. Which of the following does NOT take place during a sprint review?

A. The product backlog is updated to reflect what will probably be in the next sprint
B. The team collaborates with business users on what they will work on next
C. The working software the team built during the sprint is demonstrated
D. The team looks back at the sprint and creates a plan to improve

~~Exam Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: C

It's the Product Owner's job to help the team understand customer requirements, not the Scrum Master's. The other three answers are good examples of the Scrum Master's servant leadership role.

2. Answer: A

The product backlog does not contain any information about the status of a task. This makes sense—none of the items on in the product backlog are currently being worked on, so they all have the same status of not having been started yet.

3. Answer: D

One of the most common problems that Scrum teams run into is that the person in the Product Owner role does not have the authority to decide on behalf of the company what features the team will build during the sprint, or accept them as done on behalf of the company.

4. Answer: B

The increment is done when every item to be delivered by the team meets its definition of "done." Every item in the sprint backlog has a definition of "done" that the team uses to determine when it's ready to release to the users. The Product Owner can only accept an item on behalf of the company if it meets its definition of "done"—any item that isn't "Done" done when the timebox expires must be pushed to the next sprint.

5. Answer: A

Collective commitment means that everyone on the team feels a personal sense of responsibility to deliver not just the piece that he or she is working on, but to do what it takes to help the team deliver the whole increment at each sprint.

Just because everyone on a team works long hours, that doesn't mean they feel a genuine commitment to it. In fact, they might resent the project and the organization for interfering with their lives, and only work the extra hours out of pressure to keep their jobs.

~~Exam Questions~~ Answers

6. Answer: B

The product backlog is a Scrum artifact, not an event.

7. Answer: A

Self-organizing teams take responsibility for their own planning to meet their objectives, assign work themselves (rather than depending on a single manager or project manager to make those assignments), and fix problems with the plan as they come up. Of all of the practlices listed as answers, the Daily Scrum is the only one that impacts the way the team plans their work and executes that plan.

One reason the Daily Scrum is so important is that it's part of the transparency–inspection–adaptation cycle. The team inspects the plan every day, and adjusts it as they uncover new information about the project.

8. Answer: C

The sprint is over when the timebox expires. The same is true of any timeboxed event. Answer D seems correct because the sprint retrospective is typically the last thing that the team does during the sprint. But if the timebox expires before the team has a chance to hold their retrospective, the sprint still ends. (And that's a good opportunity for the Scrum Master to help them understand how to plan better next time.)

The Product Owner is allowed to cancel the sprint before the timebox is over. But this can waste a lot of the team's energy, and cause people in the company to lose trust in the team, so it should be extremely rare.

9. Answer: B

The purpose of the Daily Scrum questions is to give everyone a good idea of how each person on the team is progressing, so they can help identify problems with the current plan that need to be fixed. But none of the questions are about failures—that could create a negative and possibly embarrassing environment, and detract from the environment of openness.

10. Answer: B

The project manager is having trouble with the Scrum value of respect. Barry gave an honest assessment of the work that needed to be done, but the project manager ignored it and demanded a shorter deadline even though there was no business need to apply the extra pressure. That's disrespectful.

It's also extremely demotivating!

11. Answer: C

When the team holds their sprint planning meeting at the start of the sprint, the first thing they do is to decide on the sprint goal, a brief description of the objective of the sprint that will be met by completing backlog items.

Exam ~~Questions~~ Answers

12. Answer: D

The core of empirical process control theory—the theoretical underpinning for Scrum—is the transparency-inspection-adaptation cycle. In the inspection step, the Scrum team members frequently examine the scrum artifacts, as well as their current progress towards the sprint goal. They try to detect any differences between where they are and where they expected to be, so that they can take action (which is what adaptation is all about).

13. Answer: A

The increment is what the team actually delivered during the sprint. The items that the team intended to complete at the beginning of the sprint often don't exactly match up with the work they actually did. That's a good thing—it means the team used the information they learned along the way to change direction. The increment is the product of what actually happened, and the team doesn't know exactly what the current sprint's increment will contain until they've delivered it.

We learned earlier that Scrum is an incremental methodology. It's the delivery of successive increments that makes it incremental.

14. Answer: B

The sprint goal helps the team focus on the specific objective that they planned on accomplishing during the sprint.

Retrospectives and Daily Scrums can be very useful, but holding meetings is not typically a tool that teams use to help focus.

15. Answer: A

The product backlog is the single source for product requirements, and it's maintained by the Product Owner. When the Product Owner discovers a new requirement, she adds it to the product backlog.

Exam Questions Answers

16. Answer: B

Scrum teams determine what work will be done during the sprint by decomposing backlog items into tasks. The other answers are also things that the team does during sprint planning, but it's not how the team determines what work they're going to do.

17. Answer: D

During the sprint review, the team meets with the business users and customers to review what they've done and collaborate on what the next sprint will accomplish. They'll review the increment, which typically involves demonstrating the working software that they built. They'll also discuss the backlog and update it to show the items that they'll probably work on in the next sprint. The sprint review isn't for looking back at what happened and making improvements—that's what the sprint retrospective is for.

The updated backlog only reflects the probable items that will be worked on during the next sprint. That's not the same thing as committing to build certain items—the team will come up with the sprint backlog during sprint planning, and the Product Owner can make changes to it during the sprint.

DID YOU GET SOME OF THE QUESTIONS WRONG? THAT'S ***ABSOLUTELY OKAY!*** JUST KEEP TRACK OF THEM, AND MAKE SURE TO GO BACK AND RE-READ THE PARTS OF THE CHAPTER THAT COVERED THEM.

When you get a question wrong now, that actually makes it more likely that you'll get a question on the same topic right when you take the exam!

Sharpen your pencil Solution

Here are the five Scrum events, three Scrum roles, and three Scrum artifacts. Remember, each event is timeboxed, but the length of the timebox changes proportionally if the team uses a shorter sprint.

Event names in the order they occur	When the events happen	Length of the event's timebox
sprint	throughout the project	*Assume that the sprint is timeboxed to 30 days*
sprint planning	beginning of the sprint	8 hours
Daily Scrum	every day	15 minutes
sprint review	end of the sprint	4 hours
retrospective	after the sprint review	3 hours

Write down the Scrum roles	Write down the Scrum artifacts
Scrum Master	sprint backlog
Product Owner	product backlog
development team	increment

Here are some things we overheard Amy, Rick, and Brian saying. Draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and then to the Scrum value that it's either compatible or incompatible with.

When it was Brian's turn to talk at the Daily Scrum, he didn't really say anything. Openness means sharing actual details about what you've been working on and what you'll do next.

IT'S MY TURN TO TALK? OK. SINCE THE LAST DAILY SCRUM I KEPT WORKING ON THE SAME FEATURE, AND I'LL DO THE SAME UNTIL THE NEXT SPRINT. NO ROADBLOCKS. WHO'S NEXT?

COMPATIBLE

INCOMPATIBLE

It takes guts to say no to a senior manager, but Rick has the courage to stand up for the project.

LOOK, ALEX, I KNOW REDOING ALL OF THE GRAPHICS WILL IMPRESS REVIEWERS, BUT THERE'S ABSOLUTELY NO WAY THE TEAM CAN DO THAT WITHOUT MISSING THE RELEASE DATE.

COMPATIBLE

INCOMPATIBLE

It's really common for hardcore programmers to dismiss anyone who isn't technical like them, and that's disrespectful.

THE ONLY THING I'M IMPRESSED WITH IS TECHNICAL ABILITY. IF YOU CAN'T CODE, I HAVE NO USE FOR YOU.

COMPATIBLE

INCOMPATIBLE

Focus means that your top priority at work is meeting the sprint goal and working on the sprint backlog.

I CAN'T WORK ON ANY SPRINT TASKS TODAY. ANOTHER TEAM HAS A REALLY BIG DEADLINE AND THEY NEED TO BORROW ME.

COMPATIBLE

INCOMPATIBLE

Courage

Focus

Openness

Respect

Who am I?

Solution

A bunch of Scrum artifacts, events, and roles are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down its name, and what kind of thing it is (like whether it's an event, role, etc.).

And watch out—another Scrum concept that's not an event, artifact, or role might just show up and crash the party!

	Name	Kind of thing
I'm a servant leader who guides the team in understanding and implementing Scrum, and helps people outside of the team understand it.	Scrum Master	role
I'm held at the end of the sprint to do an inspection with users and stakeholders who were invited of each item that the team built.	sprint review	event
I'm how the team inspects itself, where they look for the things that went well, and make a plan to improve things that didn't.	retrospective	event
I'm the sum of all items that the team delivers to the users at the end of the sprint, and I can only be delivered if every item in me is "Done."	increment	artifact
I'm the group of professionals who actually do all of the work that's needed to deliver the software to the users and stakeholders.	development team	role
I'm responsible for deciding what items will go into the product, and I have the authority to accept them as "Done" on behalf of the company.	Product Owner	role
I'm a 15-minute timeboxed meeting held every day, where team members create a plan for the next 24 hours.	Daily Scrum	event
I'm what the Product Owner helps the team optimize and maximize, and the team tries to prioritize the items that have the most of me.	value	concept
I'm the set of items the team will build during the sprint, along with a plan to build them (usually a set of tasks the items are decomposed into).	sprint backlog	artifact
I'm a timeboxed meeting where the team comes up with a sprint goal, decides which items they'll deliver, and decomposes them into tasks.	sprint planning	event
I'm an ordered list of all of the items (with descriptions, estimates, and value) that might be needed in the product at some time in the future.	product backlog	artifact

Scrumcross

SOLUTION

			1 O							2 R						3 C	A	N	C	4 E	L									
		5 T	R	A	N	S	P	A	R	E	N	C	Y								S		6 F	7 O	C	U	S			
8 F			G							T											T			P					9 P	
I		10 B	A	C	K	L	O	G		R				11 R	12 E	S	P	O	N	S	I	B	L	E					R	
R			N						13 R	O	L	E	S		X						M			N		14 V			O	
15 S	P	R	I	N	T		16 D			S					P				17 G	O	A	L		N		A			D	
T			Z				E			18 P	L	A	N	N	I	N	G				T			E		L			U	
			I				C			E					R						E			S		U			C	
			N		19 P	R	O	D	U	C	T	O	W	N	E	20 R				21 A		22 R	E	S	P	E	C	T	
			G				M			T					23 S	E	C	O	N	D										
	24 D				25 E	M	P	I	R	I	C	I	S	M		V				26 A	R	T	I	27 F	A	C	T	S		
	E						O			V						I				P				O						
	M						S			E						28 E	V	E	N	T	S			U						
29 C	O	M	M	I	T	M	E	N	T							W				S			30 G	R	O	O	M			

this page intentionally left blank

4 Agile Planning and Estimation

Generally Accepted Scrum Practices

Agile teams use straightforward planning tools to get a handle on their projects. Scrum teams plan their projects together so that everybody on the team commits to each sprint's goal. To maintain the team's **collective commitment**, planning, estimating and tracking need to be simple and easy for the whole team to do as a group. From **user stories** and **planning poker** to **velocity** and **burndown charts,** Scrum teams always know what they've done and what's left to do. Get ready to learn the tools that keep Scrum teams informed and in control of what they build!

Meanwhile, back at the ranch...

The demo of CGW5 was the most exciting thing at the Wisconsin Game conference. But is the team a victim of their own success? Now it seems like everyone in the gaming world expects it to be the most innovative and fun game of the year. That's a lot of pressure on the team!

How would you solve these problems that have the CGW 5 team concerned about meeting their users' expectations?

Just write down a short sentence for each of these.

1. When they started out, the team thought they'd market the product as a kids' game, but some of the content is kind of violent and the gamers at the conference like the more mature focus.

.....

The demo looks really cool, but it's not exactly integrated with the CGW5 code base just yet.

2. A lot of the features in the demo are limited. To make them work for a full-length game, the team will have to go back and make changes to some pretty old parts of the code.

.....

3. A developer had an idea to add a mini-game as a downloadable add-on. But that feature looks like it's a lot more work than the team initially thought. Is it worth building that downloadable content if it means losing one of the team's best coders for most of the project?

.....

4. The biggest complaint from gamers about the demo is that the player had to stop running in order to change weapons while they were fighting a big battle. That caused them to die all the time and made the game less fun.

.....

Exercise Solution

Here are a few ways that we came up with for the team to handle these situations. Did you come up with different answers? Changes happen all the time in agile teams—the way that you and your team handle them can make the difference between success and failure.

1. When they started out, the team thought they'd market the product as a kids' game, but some of the content is kind of violent and the gamers at the conference like the more mature focus.

Brainstorm real-world users who will use the game and target features to them

You can't build a product that suits your users' needs if you don't know who they are.

2. A lot of the features in the demo are limited. To make them work for a full-length game, the team will have to go back and make changes to some pretty old parts of the code.

Add this work to the product backlog and try to do it early in the project

If the team knows there's work to do that will affect everything else, it's best to do it early in the project.

3. A developer had an idea to add a mini-game as a downloadable add-on. But that feature looks like it's a lot more work than the team initially thought. Is it worth building that downloadable content if it means losing one of the team's best coders for most of the project?

The Product Owner figures out if this feature is valuable and prioritizes it in the backlog

Since the Product Owner knows what the customer wants, he needs to make sure the features like this get the right priority.

4. The biggest complaint from gamers about the demo is that the player had to stop running in order to change weapons while they were fighting a big battle. That caused them to die all the time and made the game less fun.

Meet with the team and talk about how users will want to play the game

Writing down the features they need to build from a user perspective will help the team get it right the first time

So... what's next?

The team had enough backlogged requests to build out a crowd-pleasing demo at the beginning of the project. But now they've got to make sure that the full-length game will make gamers as happy as the demo did.

BRAIN POWER

The CGW team needs to find a way to plan and track their project. Can you think of tools you've used on your own projects that might help? How well do you think they'll work on an agile team?

Introducing GASPs!

When Scrum teams start working on a sprint, they use tools that include the whole team in setting goals and tracking them. While these practices are not part of the core Scrum rules, they've been used by many Scrum teams to plan work and keep everybody on the same page. That's where the **Generally Accepted Scrum Practices** —or "GASPs"—come in. They're not technically part of the Scrum framework, but they're so common among Scrum teams that they're found on almost every Scrum project.

All of these tools help teams share all of the information they gather for planning so that the whole team can plan and track the project together.

❶ User stories and story points

User stories help you to capture what your users need from the software so you can build it out in chunks that they can use. Story Points are a way of saying how much effort will be needed to build a user story.

❸ Task boards

Task boards keep everybody in the team on the same page about the current sprint's progress. It's a quick, visual way to see what everybody is doing.

Task boards keep all the status of stories transparent to the team.

❷ Planning poker

Teams use **planning poker** to get everybody thinking about how big each story is and how they'll build it.

In planning poker, the team explains their estimates while they decide the number of story points for each story, and then they end up coming to an agreement on both the approach and the estimate.

❹ Burndown chart

Everyone on the team can see much work they've done and how much is left to do using **burndown charts**.

This chart shows 7 points burned off of the sprint, which reflects the 2 stories the team's finished so far.

BRAIN POWER

The Y-axis of this chart is labeled story points. How do you think the team will use them?

No more 300-page specs... please?

The team used to use detailed specifications, because writing up all of the requirements for the game seemed like the most efficient way to communicate everything the users needed. But a lot gets lost in translation when one person tries to write down everything and communicate it to a development team. Is there a better, more agile way to write down requirements?

Alex: I was thinking that too. But I need more than just a report of requested features to do that, don't I?

Rick: Huh, I guess you're right. It looks like we need to figure out who our users are first, and then divide the list of features into a list of actions and benefits. Do you feel like you have enough information to do that?

Alex: Actually, when I think about it, I do have all of that. There are three kinds of gamers we target with the CGW series: novices, casual gamers, and experts.

Rick: OK, let's use those roles to write up our stories.

Alex: Yeah, figuring out the actions the users will take is pretty easy. That's usually the meat of the feature request. The benefit they receive is easy to understand too. Actually, writing our requirements this way won't be very hard.

Rick: Great! Let's just get them in the backlog and get started on our first sprint.

Alex: Not so fast, Rick. We still need to figure out how to estimate this stuff. And we don't even know what "this stuff" really is! Seriously... what, exactly, are we going to build?

User stories help teams understand what users need

Software helps people do things. When a user asks a team to build a feature, it's because they need to be able to do something in the future with it that they can't do today. The most efficient way to make sure the team builds the right thing is to keep those needs in mind throughout the development process. A **user story** is a very short description of a specific thing that users need. A lot of teams write them on index cards or sticky notes. By organizing all of their work around user stories, Scrum teams make sure they keep user needs front-and-center in their planning and prioritization process. That way, they stay focused on building what their users need and there are no surprises when the team demonstrates the stories at the end of the sprint.

User Stories

User stories describe how the user will use the software in just a few sentences. Many teams write user stories on note cards following a fill-in-the-blank format:

As a <*type of user*>, I want to <*specific action I'm taking*> so that <*what I want to happen as a result*>.

Because stories are short and modular, they're a good reminder for the team to constantly confirm that they're building the right features. You can think of each story card as a symbol for a conversation that the team is having with users to make sure that they're building features that are useful to them.

The title is just a quick way of summarizing the story

Once the team has discussed the relative size of a feature, they write the story point value on the card

This tells the team the role the user is playing when they are using the software

This is the action the user takes in the software.

This statement tells the team why the user wants the feature.

Story points let the team focus on the relative size of each story

The goal of planning isn't to predict the order features will be done in or their exact completion dates. Instead, the team assigns a point value to each story based on how big it is. That's why most Scrum teams plan their projects using **story points**, which let them compare stories with each other. By focusing on the relative size of features rather than the exact amount of time it will take to develop them, Scrum teams keep the team engaged in planning together and allow for uncertainty in their plans.

How story points work

Story points are simple: the team just picks a number of points that represents the amount of work required for each story, and assigns that number to every story in the sprint backlog. Instead of trying to predict exactly how long it will take to build a feature, the team assigns a point value to each story based on its size relative to other features they've built before. At first, the estimates vary a lot from story to story. But after a while, the team gets used to the scale they're using to estimate and it gets easier to figure out how big each story is.

One way that teams start using story points is to divide stories up into **T-shirt sizes**, and assign a point value to each size. For example, they might decide to use 1 point for extra small features, 2 points for small features, and 3 points for medium features, 4 for large and 5 for extra large. Once they decide on a scale, they just need to decide which category each story fits into. Some teams use the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21...) for story point scales because they think it provides a more realistic weight for bigger features. As long as your team uses a scale consistently, it doesn't matter which one they use.

Whatever doesn't get done in a sprint is moved from that sprint to the next and the total number of story points that are completed in each sprint is tracked as the project's **velocity.** If a team finishes 15 stories totalling 55 story points in a sprint, they track the 55 points as the sprint velocity and that gives them a general idea of roughly how much they can do in the next sprint.

Over time the team gets better and better at assigning story points and more and more consistent in the number of points they deliver in each sprint. That way the team gets a feel for how much they can do in a sprint and takes control of planning together.

Extra Small	Small	Medium	Large	Extra Large	Extra Extra Large
1 point	2 points	3 points	5 points	8 points	13 points

BRAIN POWER

Why could it be better for teams to assign a general size value to each story than an exact date?

THESE USER STORIES ARE GREAT! WE'RE REALLY GETTING A HANDLE ON WHAT THE USERS WANT FROM THE GAME. BUT HOW DO WE FIGURE OUT HOW MUCH WE CAN BUILD IN A SPRINT?

This story applies to all of the novice players, casual players, and expert players.

FIGHT MAD COW ZOMBIES

AS A PLAYER,

I WANT TO SWITCH TO ZOMBIE MODE WHERE THERE ARE TRIPLE THE NUMBER OF ENEMIES AND THEY ARE EASIER TO KILL

SO THAT I CAN REPLAY THE GAME WITH A DIFFERENT MODE

The team needs to agree on how big this story is before we can assign a story point number to it.

Making the game replayable is one of the major goals of this release.

User stories are really simple, which is why Scrum teams find them so valuable. But the most important part of a user story is the discussion that's sparked among the team and with their users and stakeholders.

One reason user stories are so effective is because of this agile principle.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Sharpen your pencil

Re-write the items in this backlog as user stories

Cows Gone Wild 5.2 Product Backlog

Item #1: Stealth chicken coop level

Value: Adds a different play mode for expert users who want to be able to replay levels.

Item #2: Big Bessie's fighting sequence needs to anticipate the hero's attacks and react faster in expert play mode

Value: This will make Bessie harder to defeat.

Item #3: Novice gamers wanted the haymaker gun to include a super-baler that doubles damage when a bale is fired

Value: Will help novice gamers get through harder battles in Easy mode.

Item #4: Users need to be able to switch weapons while sprinting

Value: Users can see their inventory without stopping

Page 1 of 7

Leave the estimates blank for now
AS A
I WANT TO
SO THAT I
AS A
I WANT TO
SO THAT I
AS A
I WANT TO
SO THAT I
AS A
I WANT TO
SO THAT I

Sharpen your pencil Solution

Re-write the items in the product backlog as user stories.

Stealth Chicken Coop Level

AS AN EXPERT PLAYER

I WANT TO PLAY THE CHICKEN COOP LEVEL IN STEALTH MODE

SO THAT I CAN REPLAY THE GAME IN A DIFFERENT MODE

Big Bessie Fight Moves

AS AN EXPERT PLAYER

I WANT TO HAVE BESSIE ANTICIPATE MY MOVES AND REACT FASTER

SO THAT I WILL HAVE MORE FUN FIGHTING BESSIE

Here are the stories we came up with. It's okay if the wording that you used is different, as long as you got practice writing stories.

Haymaker – Super Baler

AS A NOVICE PLAYER

I WANT TO GET DOUBLE DAMAGE BY SHOOTING THE HAYMAKER IN SUPER-BALER MODE

SO THAT I CAN DEFEAT ENEMIES MORE EASILY

Switch Weapons While Sprinting

AS A PLAYER

I WANT TO SWITCH BETWEEN MY EQUIPPED WEAPONS WHILE SPRINTING

SO THAT I DON'T HAVE TO STOP TO SEE MY INVENTORY

Under the Hood: User Stories

User Stories are all about delivering testable software

During the planning for an agile project, the Product Owner will work with end users to identify user stories. Those stories identify the user's needs and their rationale for asking for the feature. But just writing down the user story is only the beginning of the team's understanding of the user's need. Even though the team doesn't know all of the details of what's needed when a user story is first written, they use the card as a reminder that they need to figure out the details and plan the work to develop it. By not delving into the details of each story up front, Scrum teams keep their options open and allow themselves to make decisions about each story at the last responsible moment.

User stories were originally developed as an XP practice (we'll learn more about that in the next chapter), but they're used by many, many Scrum teams. Even though they're much shorter and less-detailed than traditional software requirements, they manage to serve the same purpose while offering teams the flexibility to plan their approach to development as late as possible. Here's how they do it:

- **Card:** First the Product Owner writes down the user story (often using the "As a... I want to... So that..." template we've been talking about), and that card reminds them to understand the details of what needs to be built.
- **Conversation:** When it's time to estimate the story, the team has a conversation with the Product Owner, and sometimes the users, to figure out the details they need to know to estimate the card. Sometimes, the Product Owner will work with designers and users to produce mock-ups, or sometimes the team will produce technical designs that help them flesh out the approach to building out a story.
- **Confirmation:** Next, the team turns their attention to the tests they'll write to make sure the user story has been built. This confirmation is an important feedback loop and the fact that user stories are small and self-contained helps both the team and users agree on the tests to run.

Some teams will write the tests that confirm each user story on the back of the user story card. That helps the team to remember how the story should work when it's done. It also helps the users and team come to an agreement on how the story will behave when the software's ready. These tests are alternately called conditions of satisfaction and acceptance criteria.

The guidelines for writing a good user story can be summed up with the acronym **INVEST**:

I - Independent: user stories should be able to be described apart from one another.

N - Negotiable: all of the features in a product are the product of negotiation

V - Valuable: there's no reason to spend time writing a card that isn't valuable to your users

E - Estimatable: each user story needs to convey a feature that the team can assign as size or effort number to

S - Small: user stories should describe independent interactions, not huge categories of functionality

T- Testable: being able to test each user story is what makes it such an effective feedback loop for Scrum teams

The whole team estimates together

Once the team has a prioritized list of user stories to get started with, they need to figure out how much effort it will take to build them. They usually estimate the story points needed to build each story out as part of the Scrum planning meeting at the start of each sprint. Most often the team knows which stories are the highest priority by looking at the backlog, so they try to commit to as many high priority stories as they can in each sprint. One way the team does this is planning poker.

1 The setup

Each team member has a deck of cards with valid estimation numbers on each card. Usually the Scrum Master moderates the session.

When the team can't be in the same room to use cards, the team will agree on the point scale they're going to use up front and a method for communicating estimates. Many distributed teams will have everybody give their estimates over an instant message system to the moderator instead of using physical cards.

2 Understanding each story

The team goes through each story in the sprint backlog in priority order with the Product Owner and asks questions about the story to figure out what the users need.

3 Assigning a story point value

Once the team has discussed the feature, each person assigns a story point value by choosing a card from the deck and shares that value with the group.

4 Explaining the high and low numbers

If the estimates differ between team members, the high number and the low number explain their estimates.

5 Adjusting the estimates

Once the team has heard the explanations, they have a chance to choose an estimation card again. If the team can't be in the same room, they communicate their estimate using e-mail or IM to the moderator without sharing it out loud to the team.

6 Converging on an estimate

Usually, teams start with a significant range of estimates but that range narrows over the course of explanation and adjustment. After a few iterations through the process, the estimates converge on a number that the team is comfortable with. It usually only takes 2-3 iterations of discussion until the team can unanimously agree on a story point value.

No more detailed project plans

It feels like you've got a really good handle on your project if you create a plan to map out all of the dependencies and figure out who will be doing what from the time you start until the end. Traditional project plans make everybody feel like the there's a guarantee of success because everything has been thought through. More often than not, the information you have at the beginning of the project isn't enough to make a completely accurate detailed plan. But some of the decisions traditional project plans ask you to make in the beginning turn out to be different than the ones you'd make if you were in the middle of the project.

Scrum teams try to make decisions at the last responsible moment and allow for change, because they realize that detailed project plans can lead a team to focus on following the plan rather than responding to the changes that come up naturally. That's why Scrum teams work on prioritizing the backlog and doing the highest priority work first. That way, they're always working on the most important tasks, even when things change.

Rick: How do I know when we'll be done though? Or who's doing what?

Brian: You're right. Assigning a story point value to a user story is a lot different than telling you how many days it will take to do the work.

Rick: So I'm just supposed to tell upper management that we'll do as much work as we can over the next two week sprint?

Brian: Well, as long as we're all working on the highest priority features, we're doing the most valuable work we can be doing. I think it's a combination of us doing high priority work and demoing what we've done in every sprint review that keeps everyone in the loop.

Rick: OK. But without a project plan, how do I even know if everybody is busy working on the right tasks?

Estimation Way Up Close

Here are a few concepts used in estimating software in general that will help you understand how Scrum teams estimate:

Elapsed time

Estimates done in elapsed time predict the date a task will be completed. Estimates like this often require buffers and contingency to set expectations. If a member of the team is going to be on vacation during a project, that person can't be assigned work during that time, and the overall project estimate needs to be adjusted to deal with it. Some projects go as far as trying to predict the total number of hours in a day that each person will be able to work on project work versus the time they spend in meetings or other overhead.

Traditional project management practices attempt to account for all of the possible interruptions and schedule adjustments from the beginning of the project. Projects that are planned in this way attempt to forecast a hard-and-fast end date based on the scope of work and effort estimate.

Ideal time

This is the amount of time necessary to accomplish a task if the person who is assigned to it is able to work without interruption. When an estimate is made in ideal time, it assumes that there is no overhead, no sick days, and no competing priorities that would take the person away from the project work and affect your delivery date. Agile teams estimate in ideal time and they use empirical measurements of how much work each team has delivered in past sprints to set expectations of what can be done in a given time frame.

Story Point

A numeric indicator of the relative size of a feature. Features that require roughly the same amount of effort are given the same story point value. Story points don't require buffers when you assign a story point value to a feature, you assume it's a measure of relative size given all of the normal disruptions and uncertainty. Because they're relative size values, they do not translate to specific time values. You can't say, a story point is equal to one hour of work, for example. You can, however, say that a story point is equal to the effort required to create a button and link it to an action.

Velocity

The number of story points completed in a sprint. This number is averaged over time to predict the amount of work that can be accomplished across multiple sprints. Velocity values normally vary at the start of a project and stabilize as the team gets more comfortable with each other and the work they're doing. When a team has been working with a consistent velocity for some time you can forecast that they will continue to deliver at that rate. Rather than predicting the delivery date for each feature in an agile project, the team focuses on delivering the highest value work possible in each sprint and maintaining a sustainable velocity of work.

Taskboards keep the team informed

Once your team has planned the sprint, they need to get started building it. But Scrum teams generally don't sit down and figure out who will do each task at the beginning of each sprint. They try to make it easy for team members to make decisions at the last responsible moment by giving the whole team constantly updated information about how the sprint is progressing.

Most teams start by marking a whiteboard with three columns: To Do, In Progress, and Done. As a team member starts working on a story, he or she will move it from the To Do column to the In Progress column and to the Done column when it's complete.

1 Sprint Begins

All of the user stories are in the To Do column because no one has started working on them yet.

Task boards keep the status of all stories transparent to the team.

2 **Mid Sprint**

Team members move their tickets to In Progress when they start working on them and to the Done column as they are completed. The team usually agrees to a definition of done up front so that everyone is clear on what it means to be done with a story.

Because the team knows which stories are being worked on, they know what they can take on next to help out.

Now everyone can pick what they should work on next instead of waiting for someone to assign it to them.

3 **Sprint Ends**

If the team estimated well, all of the user stories they put in the backlog have been moved to the Done column. If there are leftover user stories, they're added to the next sprint backlog.

If there are any stories left in the "To Do" column or the "In Progress" column at the end of the sprint, they go back to the product backlog to be evaluated in the next planning session.

The team finished all of the user stories in the backlog for this sprint.

Q: So what's the difference between a user story and requirement? Is it just that it's written on an index card?

A: No. In fact a lot of teams don't write their user stories on cards at all. Often, they're they're created as tickets in an issue tracking system. They can be rows in a spreadsheet, or bullets in a document. The biggest difference between user stories and traditional software requirements is that user stories don't attempt to nail down the specific details about the feature that's being described.

The goal when you're writing a user story is to capture just enough information that everybody can remember who's going to use the feature, what it is, and why users want it. The story itself is a way to make sure that the team talks about the feature and understands it well enough to do the work. Sometimes teams will need to write more documentation once they've confirmed the story with the users. Sometimes just having the conversation is enough, and the team can build the feature without any more documentation.

Q: How do I know how many story points to assign to a story?

A: When a team is starting to use story points for the first time, the first thing they typically do is get together and decide what type of work is worth one story point—usually a simple task that everyone on the team can understand. (For example, a team working on a web application might decide that one story point is equivalent to the effort it takes to add a button with some simple, specific functionality to a web page.) Depending on the kind of work they usually do, they'll choose a scale that makes sense to the team. But once they decide the value of one point, it helps the team understand the rest of the possible point ranges.

there are no Dumb Questions

Some teams use a practice called **T-shirt sizing** to assign all of the stories they're estimating into small, medium, or large categories and assign points that way. (1 point for small, 3 points for medium, 5 points for large). Other teams use broader scales (XS, S, M, L, and XL) with corresponding point values. Other teams assign values using the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21...). As long as the team is consistent in how they assign points to stories, it doesn't matter which approach they use.

Q: What's the point of planning poker? Can't developers just estimate their work on their own?

A: Like most of the other GASPs, planning poker focuses on involving the whole team in planning and tracking progress on your project. Planning poker is all about getting the team to discuss their estimates and agree on the right approach for development. By being transparent about estimates and approach, the team can help each other avoid mistakes and think through the most efficient method for developing each feature together. Planning poker helps teams make their estimation reasoning transparent. When the team decides on the approach and the estimate together, they have a much better chance of catching flaws in their approach earlier and they have a better handle on the work that needs to be done.

Q: What happens if you get the estimate wrong?

A: That happens—and it's okay. You might end up thinking that a feature is worth 3 story points at the beginning of the sprint but realize by the end of it that it should've been a 5. But since story points are used to measure your overall velocity over time, you'll find that the whole team gets better and better at figuring out which features are which as they work with their scale. What's great about planning poker and story points is that they don't expect you to be able to predict the future. Once you assign story points to your sprint backlog, you build your sprint backlog and then track your velocity numbers over time. If you have more story points in the backlog than you can do in the sprint, you move them to the next sprint. As the team estimates and keeps track of what they're delivering, they get better and better at it.

In the beginning, you'll see that the number of story points your team completes per sprint varies a lot. But as the team gets more and more comfortable working together, the number of story points they can accomplish in a sprint becomes more and more predictable.

Rather than focusing on getting each estimate right, GASPs help your team to get a handle on how much work you can actually do. That way you can take on the right amount in each sprint and keep your team working as efficiently as possible.

Planning poker, story points, and velocity get the whole team planning and tracking work together. All of these tools are about the whole team staying accountable for the project's vision and plan.

That's true. Stakeholders who are used to traditional status reports and project plans will need to adjust to these practices.

Traditional project management methods create a plan for delivery up front and then track delivery to that plan. Getting everybody in your organization to give up that feeling that they know exactly how a project will be developed from the very beginning can be a major challenge for agile teams.

Instead of planning each detail up front and holding the team to that plan, Scrum teams promise transparency, the ability to change, and a team focus on building the best product possible the with the time and resources available. By building incrementally and delivering frequently, they often have much happier stakeholders once they get accustomed to working differently.

Question Clinic: The red herring

SOMETIMES A QUESTION WILL GIVE YOU A LOT OF EXTRA INFORMATION THAT YOU DON'T NEED. IT'LL INCLUDE A RAMBLING STORY OR A BUNCH OF EXTRA NUMBERS THAT ARE IRRELEVANT.

Did you read that whole paragraph, only to find out the question had nothing to do with it?

104. You are managing an advertising software project. You have to build an interface for buying space in online publications at an average cost of $75,000 per placement. Your project team consists of a advertising analyst as a Product Owner, and a team of experienced software engineers. Your business case document is complete, and you have met with your stakeholders and sponsor. Your senior managers are now asking you to plan your first sprint. Your team has done four other projects very similar to this one, and you have decided to make your estimate by having the team provide story point values in a group session and converge on an agreed estimate and approach together.

What estimation practice involves having the team provide individual estimates and discuss them until they converge on an agreed value?

A. Planning Poker
B. Planning method
C. Bottom-up
D. Rough order of magnitude

WHEN YOU SEE A RED HERRING QUESTION, YOUR JOB IS TO FIGURE OUT WHAT PART OF IT IS RELEVANT AND WHAT'S INCLUDED JUST TO DISTRACT YOU. IT SEEMS TRICKY, BUT IT'S ACTUALLY PRETTY EASY ONCE YOU GET THE HANG OF IT.

HEAD LIBS

Fill in the blanks to come up with your own red herring question!

You are managing a _____ project.
(kind of project)

You have _____ at your disposal, with _____ . Your
(describe a resource) (how that resource is restricted)

_____ contains _____. The _____
(a project document) (something that document would contain) (a team member)

alerts you that _____, and suggests _____.
(a problem that affected your project) (a suggested solution)

_____?
(a question vaguely related to one of the things in the paragraph above)

A. _____
(wrong answer)

B. _____
(trickily wrong answer)

C. _____
(correct answer)

D. _____
(ridiculously wrong answer)

Rick: It feels like we've almost got it down. Alex keeps the product backlog prioritized. At the start of each sprint we go through the highest priority stories, play planning poker, and assign them story point values. Then we add them into the sprint backlog and get started.

Brian: That all works really well. The team likes getting a chance to talk through the work before we do it. It helps everybody stay on the same page about what needs to get done in a sprint, too.

Rick: It does seem like it's working, but then we get to the end of each sprint and we've always got stories that were bigger than we thought they were. There's always stuff in the sprint backlog that needs to get carried over the next sprint. That's making Alex nervous and making our sprint reviews with the users more tense.

Brian: We need to know whether or not we're on track during the sprint, so we can make adjustments if we need to. We shouldn't commit to stories at the beginning of the sprint if we don't think we can get them done.

Rick: I think it's time for us to start tracking our progress more closely. So... um, how exactly do we do that with Scrum?

Burndown charts show how much work the team has done

Once a team has assigned a story point value to all of the user stories in the sprint backlog, they can use **burndown charts** to get a handle on how the project is progressing. A burndown chart is a simple line chart that shows how many story points are completed each day during the sprint. The burndown chart gives everybody a clear sense of how much work is left to be done at any time. Using a burndown chart, it's clear to everyone on the team how close they are to achieving their sprint goals.

Burndown charts and velocity help the whole team stay in control of the sprint.

Velocity tells you how much your team can do in a sprint

At the end of each sprint, you can count the total number of story points that have been completed. The number of points per sprint is called the **velocity**, and it's a great way to gauge how consistently the team is delivering work. Many teams plot their velocity per sprint as a bar chart so they can see how they did across multiple sprints. Since each team's scale for estimating story points is different, **you can't use to velocity to compare teams to one another**. But you can use it to help figure out how much work your team should commit to based on their past performance.

Sprint velocity

This is a bar chart of the total number of story points completed in each of 4 sprints. If the team is using the same scale for estimation in each sprint, you can use this number to compare how much work has been done from one sprint to the next. To create this chart, the team just adds up the number of story points in the Done column of the task board at the end of each sprint.

Sprint velocity with committed points

This is a bar chart of the total number of story points the team put into the sprint backlog in grey and the number they actually completed in black. To create this chart, the team just adds up the number of story points in the sprint backlog after the planning session and marks that as the committed number. At the end of the sprint they track the velocity number by adding up all of the story points in the Done column of the taskboard.

Sharpen your pencil

Here are Rick's notes from looking at the task board after each day's Daily Scrum. The total estimate for this sprint's backlog is 40 points. Draw the burndown chart.

Day 1: We finished the clean up on the super-baler feature, that's 2 points. We can mark the build script as complete too, that's another 2 points.

Day 2: We can't mark anything complete today.

Day 3: 3 points, we finished the new finishing move for the big Bessie fight.

Day 4:Add two points, we found a refactoring task that has be done to get the haymaker working again.

Day 5: Finished the haymaker refactoring, 2 points.

Day 6: The stealth chicken coop is complete, 8 points.

Day 7: Completed the distribution package script, 5 points.

Day 8: Updated Bessie's AI to react faster, 10 points.

Day 9: Added animations for the super-baler reload, 2 points.

Day 10: Finished the chicken coop refactor, 7 points.

Sharpen your pencil

Here are Rick's notes from looking at the task board after each day's Daily Scrum. The total estimate for this sprint's backlog is 40 points. Draw the burndown chart.

Day 1: We finished the cleanup on the super-baler feature, that's 2 points. We can mark the build script as complete too, that's another 2 points.

Day 2: We can't mark anything complete today.

Day 3: 3 points, we finished the new finishing move for the big Bessie fight.

Day 4:Add two points, we found a refactoring task that has be done to get the haymaker working again.

Day 5: Finished the haymaker refactoring, 2 points..

Day 6: The stealth chicken coop is complete, 8 points.

Day 7: Completed the distribution package script, 5 points.

Day 8: Updated Bessie's AI to react faster, 10 points.

Day 9: Added animations for the super-baler reload, 2 points.

Day 10: Finished the chicken coop refactor, 7 points.

Burn-ups keep your progress and your scope separate from one another

Another way to track your progress during a sprint is to use a burn-up chart. Instead of subtracting the number you've completed from the number you committed to, burn-ups track a cumulative total throughout the sprint and show the total committed scope on a separate line. When stories are added or deleted from the scope it's obvious by looking at the scope line. When stories are put into the "Done" column on the task board, that's easy to see too, by looking at the total number of points burned up in the sprint. Because the scope is tracked on a different line from the number of points accomplished, it's clearer when the scope is changing.

How do we know what to build?

The Product Owner's role on a sprint team is to keep everybody working on the most important thing in every sprint. They're in charge of the order of stories in the sprint backlog and the product backlog. When the team has questions about a user story, the Product Owner is the one who tracks down the answers. Many teams set a time near the end of each sprint to make sure that the backlog is in order before the team starts to plan the next sprint. That meeting is called the **backlog grooming** meeting.

The Product Owner sitss down with as many of the team members as can make it 2 or 3 days before the end of a sprint to take a look at the stories in the backlog in preparation for the next sprint's planning meeting. They use the time to come up with questions that need to be answered before the planning session, and to double-check the priority order of the stories.

Once the backlog grooming meeting is over, the Product Owner has a couple of days before the start of the next sprint to follow up on any open questions and make sure that the priorities make sense to business stakeholders as well.

Some teams refer to grooming the product backlog mid-sprint as product backlog refinement (or "PBR"). In general, teams typically spend less than 10% of their time doing this.

Story maps help you prioritize your backlog

One way to visualize the backlog is to lay it out in a story map. Story maps start by identifying the most core features of your product as its **backbone**. Then that functionality is broken up into the backbone's most important user stories. Those are call the **walking skeleton**. Your first sprints should be focused on delivering as much of the walking skeleton as possible. After that, you can plan your releases to include the features in the prioritized order on the map.

Story maps give teams a way to visualize the release plan

Personas help you get to know your users

A **persona** is a profile of a made-up user that includes personal facts and often a photo, which Scrum teams use to help them understand their users and stakeholders better. Putting a face to each user role and writing down what motivates them will help you make the right choices when you're thinking about how and what to develop. Personas make your user stories more personal. Once the Ranch Hand Games team created them, they started thinking about how each user would react to the features they were building.

Melinda Oglesby

Age: 28
Occupation: IT Consultant
Location: New York, NY
Role: Expert Gamer

Bio: Attends gaming conferences when possible. Owns all of the available consoles. Plays most games more than once. Has built PCs for gaming.

Goals:
Satisfying story
- Multiple play styles / story options
- Challenging puzzles / fights
- Co-operative game play

Frustrations:
Poor quality (bugs, getting stuck, crashes)
- Plot holes
- Poor server performance

To create this persona, Alex interviewed 50 gamers at the conference about how they play CGW games.

These goals and frustrations came up in many of the interviews.

Now that they have a face and a name for their expert gamers, the team often thinks about Melinda's opinion when they're deciding how to design a feature.

BRAIN POWER

How do personas and story maps fit into the ideas of transparency, inspection, and adaptation that make Scrum work?

BULLET POINTS

- **User stories** capture a user need that describes the role of the user, the action they want to accomplish and the benefit that want to achieve.
- User stories are often written using the following template: *As a <role>, I want to <action>, so that <benefit>.*
- **T-shirt sizing** is a method used by many teams to group features into sizes (S, M, L, XL, XXL) based on the amount of effort necessary to build them.
- **Story points** are a way of measuring the size of the effort necessary to build a story. They do not correspond to hours, or calendar time.
- **Planning poker** is a collaborative estimation technique used by Scrum teams to determine the story point values for each story in a sprint by anonymously gathering estimates and adjusting them after listening to the reasoning behind the low and high values team members have given.
- By involving the whole team in planning, GASPs help Scrum teams **adapt** their plan based on what they learn from sprint to sprint.
- **Velocity** is the total number of story points a team accomplishes on average during a sprint.
- Teams use **burn-down charts** to track how many story points they've accomplished on each day during a sprint.
- Product owners hold **backlog grooming** meetings near the end of a sprint to get the backlog ready for the next planning session.

there are no Dumb Questions

Q: So how do I use points to make sure I'm on track for the whole project? Do I estimate the whole backlog?

A: Some teams do. Some teams estimate the entire backlog to come up with a **Release Burndown** chart where they track how far they've come in burning down all of the features initially put into the product backlog. This is how some teams predict their overall release dates for major projects.

But that only works if you're relatively sure that everything that's in the Product Backlog actually needs to be delivered as part of your project. In many cases, there are features in a product's backlog that are so low priority that they will probably never be built. When that's true, it doesn't really make sense to estimate everything in the product backlog and use it to gauge a release date. Instead, many teams focus on building the highest priority functionality possible in every release and releasing software frequently. That way the most important features are always available as soon as possible.

Q: How do I know how many stories to put into a sprint?

A: When your team sits down to plan a sprint, they always know the goals for the project and the highest priority features in the backlog. That should be enough for everyone to have a good idea of which features are most important to the project. Using that knowledge to commit the team to delivering the most valuable product possible in the sprint has been referred to as **commitment-driven** planning by Mike Cohn, one of the most influential thinkers in Agile planning,

The other option teams have is **velocity-driven** planning. This means starting with the team's average velocity, and adding the top stories from the backloig until they reach the average velocity. Cohn prefers commitment-driven planning because it relies on the team members' judgement of what's necessary to build a valuable product.

Q: Are story maps and task boards the same thing?

A: No. Both of them can use whiteboards and story cards to show what's going on in your project, but they're displaying very different information.

You can think of the taskboard as an up-to-date look at the sprint backlog. Checking it will always tell you what's going on with each story in the current sprint.

The story map gives you a similar view of the current plan for all of the stories in the product backlog.

The story map helps everyone on your team have the same vision of where the product is going. Story maps give teams a way to visualize the release plan and understand how stories fit together.

The news could be better...

Now that the team has simple metrics to track their performance, it's easier for them to see when things don't go as planned. When they take a look across a number of sprints, they can tell that they aren't as predictable as they'd hoped they would be.

IT SEEMS LIKE WE'RE DOING A LOT IN SOME SPRINTS AND VERY LITTLE IN OTHERS.

But they overcommitted on the next 3 sprints and it looks like they're having a hard time planning the right amount of work for each sprint.

Looks like the team completed more stories than they planned in this sprint.

YEAH. LOOKING AT THIS VELOCITY, WE'RE A LITTLE ALL OVER THE PLACE. LET'S WORK WITH THE TEAM TO GET IT MORE CONSISTENT.

BRAIN POWER

Can you think of ways to involve the whole team in transparency, inspection, and adaptation around the process the team is using in each sprint?

Retrospectives help your team improve the way they work

At the end of each sprint, the team reflects over the experience they just had and works together to fix any issues that came up. Retrospectives help your team stay aware of how things are going and stay focused on making things better with each sprint. As long as the team is learning from their experience, they'll get better and better at working together as your project progresses. In *Agile Retrospectives: Making Good Teams Great*, Esther Derby and Diana Larsen lay out a simple outline for a retrospective meeting.

1 Set the stage

At the beginning of the meeting, everybody needs to understand the goal and focus of the retrospective. Derby and Larsen also recommend getting each team member to tell the team their overall mood as an opening activity. If everybody gets a chance to talk when the meeting starts, it's more likely they'll feel comfortable sharing their opinions later on.

A team might focus a retrospective on why they're finding more defects in recent sprints, or how to communicate better about design changes.

2 Gather data

During this part of the meeting the team takes a look at all of the events of the last sprint using hard facts. They walk through the timeline and discuss the work that was completed and the decisions that were made. Often, team members are asked to vote on these events and decisions to determine whether they were high or low points for them in the sprint.

?

The vertical "fishbone" lines are categories to help you find and organize the root causes of defects.

Horizontal lines show the root causes you've found for each category.

3 Generate insights

When the team has gathered the data about the sprint, they can zero in on the events that seemed to be the most problematic for the group. During this part of the meeting, the team identifies the root causes of the problems they encountered and spends time thinking about what they could do differently in the future.

Fishbone or Ishikawa diagram

For this example, a team is looking at causes of defects in a sprint.

Teams use fishbone diagrams to understand the root causes of issues.

4 Decide what to do

Now that they've reviewed what happened during the sprint and spent time thinking about what they might do differently, the next step is decide which improvements to implement for the next sprint.

!

Some tools to help you get more out of your retrospectives

One way Scrum teams implement the Agile Manifesto principle of periodically reflecting and improving they way they work is to use these tools in their retrospective meetings:

Tools to help you set the stage:

- ***Check-ins*** are a way to get your team to engage at the beginning of the retrospective. The retrospective leader will often ask team members to go around the room with each person giving a one or two word answer to a question at the start of the meeting.
- ***ESVP*** is a technique where each member of the team is asked to categorize themselves into one of 4 designations: Explorer, Shopper, Vacationer, or Prisoner. **Explorers** want to learn and get as much as they can out of the retrospective. **Shoppers** are looking for one or two improvements out of the retrospectives. **Vacationers** are just happy to be doing something different and away from their desks for the meeting. **Prisoners** would rather be doing something else and feel forced to be part of the retrospective. Asking team members to say which group they think they are a part of helps everybody understand where they're coming from and also helps them to feel more engaged in the meeting.

Tools to help you gather data:

- ***Timeline*** is a way of displaying all of the meaningful activities that happened in a sprint in chronological order. Each member of the team gets an opportunity to add cards to the timeline with the events that were significant to him or her. Once the team creates the first round of cards to go on the timeline, they review it together and add new cards if they can think of events that should be on timeline.
- ***Color Code Dots*** are used to indicate how team members felt about all of the events on the timeline. The moderator might hand out green dots to indicate positive feelings toward an event on the timeline and yellow ones for negative feelings. Then everyone on the team would go through the timeline indicating whether the activities on the timeline were positive or negative to them.

Tools to help you generate insights:

- ***Fishbone*** diagrams are also called **cause and effect** and **Ishikawa** diagrams. They are used to figure out what caused a defect. You list all of the categories of the defects that you have identified and then write the possible causes of the defect you are analyzing from each category. Fishbone diagrams help you **see all of the possible causes** in one place so you can think of how you might prevent the defect in the future.
- ***Prioritize with dots*** is a technique where each team member is given 10 dots to stick on the issues that they want the team to attempt to address first. Then choose the issues that have the most dots to focus on in the "decide what to do" phase of the retrospective.

Tools to help you decide what to do:

- ***Short Subjects*** are a way a categorizing all of the insights the team has come up with into an action plan. Typically the moderator will put short subjects on the top of a whiteboard and the team will work together to put all of their suggestions in the right categories. One common set of short subjects is Stop Doing / Start Doing / Keep Doing. The team takes the time to categorize all of the feedback they've given in the retrospective into the kinds of actions that they can take to make sure they preserve the practices they're doing that are working and change the ones that aren't.

The team had a problem with setting user expectations before they had a planning meeting, so they want to make sure to stop doing that in the next sprint.

Cubicle Conversation

Scrum teams are always focused on improving. At the end of each sprint, everyone on the team takes a look at their burndown chart, their velocity, and the number of stories they've got on the backlog as input to their retrospective. Using the metrics the team has produced through the sprint helps everyone stay on the same page about the root causes of team issues, and that helps the team work together to solve problems that might've come up along the way. Retrospectives are just one more example of how Scrum teams use transparency, inspection, and adaptation to get better and better at building software.

Rick: Now that we've got velocity numbers for the past 4 sprints, it looks like we really need to get a better handle on how much we commit to at the beginning of the sprint.

Alex: But how do we do that? I'm only putting the stuff in the sprint backlog that the users need.

Rick: I think we should bring these velocity numbers to the team retrospective, show them the problem, and figure out some solutions together.

Exercise

Here's what the team had to say about their velocity variance at the most recent sprint retrospective. Match the comment with its short subject.

PROGRAMMERS SHOULDN'T HAVE TO PAY ATTENTION TO VELOCITY NUMBERS. THAT'S THE SCRUM MASTER'S JOB.

Continue Doing

WE SHOULD PROBABLY NOT COMMIT TO MORE STORY POINTS THAN WE DELIVERED IN THE MOST RECENT SPRINT.

Stop Doing

I REALLY LIKE PLANNING POKER, IT HELPS THE TEAM TO FIGURE OUT A DESIGN APPROACH REALLY EFFICIENTLY.

Start Doing

I COULD WAIT TO TALK TO THE STAKEHOLDERS ABOUT OUR GOALS AFTER WE'VE ALL AGREED ON WHAT GOES IN THE SPRINT BACKLOG.

Not Constructive

Exercise

Here's what the team had to say about their velocity variance at the most recent sprint retrospective. Match the comment with its short subject.

PROGRAMMERS SHOULDN'T HAVE TO PAY ATTENTION TO VELOCITY NUMBERS. THAT'S THE SCRUM MASTER'S JOB.

WE SHOULD PROBABLY NOT COMMIT TO MORE STORY POINTS THAN WE DELIVERED IN THE MOST RECENT SPRINT.

I REALLY LIKE PLANNING POKER, IT HELPS THE TEAM TO FIGURE OUT A DESIGN APPROACH REALLY EFFICIENTLY.

I COULD WAIT TO TALK TO THE STAKEHOLDERS ABOUT OUR GOALS AFTER WE'VE ALL AGREED ON WHAT GOES IN THE SPRINT BACKLOG.

- Continue Doing
- Stop Doing
- Start Doing
- Not Constructive

GASPcross

Here's a great opportunity to seal the GASPS into your brain. See how many answers you can get without flipping back to the rest of the chapter.

Across

7. User stories are signifiers for a three-step process that focuses on face-to-face communication between development team members and stakeholders. That process is often described as, card, conversation, and _____

9. ESVP stands for explorers, shoppers,_____, and prisoners

11. Stakeholder needs written using a template (as a <role>, I want to <action>, so that <benefit>) are called user _____

13. _____ diagrams are used to determine the root cause of issues.

14. The y-axis of a burndown chart is labeled story_____

17. When the Product Owner prepares the backlog for a planning session a few days in advance, that's called _____

18. Grouping features in small, medium, large effort categories is called _____ sizing

19. The topmost line in a story map is called the _____

20. An acronym to help you identify a good user story

21. A tool for tracking the current state of all stories in a sprint.

Down

1. _____ _____ are a way of categorizing follow-up actions from retrospectives

2. When estimating effort in story points, some teams use _____ series numbers to appropriately size features

3. Derby and Larsen's basic progression for a retrospective is: set the stage, gather data, _____, decide what to do.

4. Effort estimates that don't take interruptions or other non-working time into account are done in _____ time

5. Estimates that tell the date features will be delivered are done in _____ time

6. A Scrum planning technique that depends on the team describing the high and low individual

estimates from team members until the group reaches consensus.

8. The number of story points completed in a given sprint is called _____

10. _____ are a way of assigning a name and personal facts to a made-up user of your system

12. When planning a sprint, some teams use velocity-driven planning to determine how many story points to include. Others use _____-driven planning to do the same thing.

15. The features needed to implement the minimum functionality needed in a product are shown on a story map in the _____ skeleton

16. _____ charts track scope and completed story points on different lines

GASPcross solution

Pizza party!

The Ranch Hand Games team used their retrospectives to find and fix issues with planning and delivering CGW5. Over time, their sprints went more and more smoothly and they found themselves able to demo great new features in every sprint. By the time the team was ready to ship the game, the whole company knew they had a winner on their hands!

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. Burndown charts are used for all of the following except:

A. Helping the team understand how many points have been delivered in a given sprint
B. Helping the team understand how many points are left to be delivered before the end of a sprint
C. How many points each team member has delivered
D. Whether or not the team will deliver everything they committed to in a given sprint

2. The total number of story points delivered in a sprint is called the sprint _____

A. Increment
B. Review
C. Ideal time
D. Velocity

3. Jim is a Scrum master on a Scrum project in a media company. His team has been asked to buld a new advertising presentation component. They've been working together for 5 sprints and have seen increased velocity over the past two sprints. The team gets together on the first day of the sixth sprint for a planning session. In that session they use a method where the team discusses the features that will be built with the Product Owner, provide estimates on cards, and adjust their estimates as a group until they converge on a number they all agree to.

Which of the following BEST describes the practice they are using?

A. Planning poker
B. Convergence planning
C. Sprint planning
D. Analogous estimation

Exam Questions

4. What acronym can be used to describe good user stories?

A. INSPECT
B. ADAPT
C. INVEST
D. CONFIRM

5. Velocity can be used for all of the following except:

A. To measure team productivity over multiple sprints
B. To compare teams to each other and find out who's more productive
C. To understand how much a team can do when they're estimating a sprint
D. To understand if the team is committing to too much or too little

6. Which tool is used to visualize scope changes?

A. Velocity Bar Charts
B. Burn-up Charts
C. Cumulative Flow Charts
D. Scope Histograms

7. How are user stories commonly written?

A. As a <persona> I want to <action>, so that <benefit>
B. As a <resource>, I want to <goal>, so that <rationale>
C. As a <role> I want to <action>, so that <benefit>
D. None of the above

8. Which of the following BEST describes a taskboard?

A. The Scrum Master uses them to see if the team is following the plan
B. They are used to identify new tasks during a sprint
C. They show the total number of points accomplished in a sprint
D. They show every task in the sprint backlog and its current state

Exam Questions

9. Which of the following BEST describes Derby and Larsen's retrospective method:

A. Set the stage, gather information, decide what to do, document decisions
B. Check in, create timelines, interpret the data, decide where to focus, measure
C. Set the stage, gather data, generate insights, decide what to do
D. ESVP, Color Code Dots, Short Subjects

10. What can you tell about this sprint by looking at the burndown chart above?

A. The sprint is ahead of schedule
B. The sprint is behind schedule
C. The project is in trouble
D. The velocity is too low

11. What is the difference between a burndown and a burn-up chart?

A. Burndown charts subtract story points form the total number committed while burn-up charts start at 0 and add the story points to the total as they're completed
B. Burndown charts have a line for scope that tells you how much is added or deleted as you go
C. Burn-up charts have a trend line to show you the constant rate of completion
D. Burn-up charts and burndown charts are the same

Exam Questions

12. Which of the following is the BEST tool for determining the root cause of a problem?

A. Personas
B. Velocity
C. Fishbone diagrams
D. Short Subjects

13. A Scrum team for medical software company took all of the user stories in their product backlog and arranged them on the wall according to how important the functionality is to a successful product. Then they used that information to determine which features to work on first. What term best describes the practice they were using?

A. Release planning
B. A walking skeleton
C. Velocity planning
D. Story mapping

14. The process of identifying requirements based on user stories is often referred to as

A. Card, Call, Confession
B. Story, Conversation, Product
C. Card, Conversation, Confirmation
D. Card, Test, Documentation

15. ESVP stands for

A. Executive, Student, Vice President
B. Explorer, Student, Vacationer, Prisoner
C. Explorer, Shopper, Vacationer, Practitioner
D. Explorer, Shopper, Vacationer, Prisoner

16. Your Scrum team began measuring velocity over the past 3 sprints and recorded the following numbers: 30, 42, 23. What can you tell about the team from these measurements?

A. The team is still determining its story point scale
B. The team is becoming less productive and actions must be taken to correct this
C. The velocity is evening out over multiple sprints
D. The velocity has not been measured correctly

Exam Questions

17. Which of the following BEST describes a tool for identifying a representative software user and describing his or her needs and motivations?

A. Ishikawa diagrams
B. User Identification Matrices
C. Personas
D. Story Mapping

18. Scrum Planning tools help Scrum teams make project decisions...

A. As early as possible
B. Just in time
C. At the last responsible moment
D. Responsively

19. What can you learn about this project from looking at the velocity bar chart above?

A. The project has too much velocity
B. The team is delivering more story points as the project goes on
C. Too many scope changes are happening
D. The project is running behind

Exam Questions

20. What can you learn about this project by reading the burn-up chart above?

A. Some story points were added to the project scope on day 4 and some were removed on day 7

B. The team is adding stories to the scope each day of the sprint

C. The team is not making progress

D. Stories were added to the project on day 4 and that caused a delay on day 8

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: C

Burndown charts help the whole team see how much they've accomplished and how much is left to do. They do not show individual team member productivity.

2. Answer: D

Teams measure the total number of points they deliver in each sprint as velocity. Velocity can be measured across multiple sprints to help teams get better at estimating and committing to work. Velocity is often used to show the effect of process changes as well.

3. Answer: A

The team is planning using planning poker. They are doing sprint planning, but since the question was specific about how they were planning, that's not the best answer. They're also doing analogous estimation, but that's not the BEST answer for this question because it is not a generally accepted Scrum pratice. Convergence planning is a made-up name, so don't be confused by counterfeit practice names like these.

4. Answer: C

The acronym INVEST stands for Independent, Negotiable, Valuable, Estimatable, Small, and Testable. A good user story should be all of these things.

5. Answer: B

Since velocity is the sum of all of the story points estimated in a given sprint, it can only be used within one team. Another team would have a different scale since their story point values would come from their sprint planning discussions.

Exam ~~Questions~~ Answers

6. Answer: B

Burn-up charts show scope as a separate line on the chart and make it easy to see when scope is added or removed.

7. Answer: C

The correct template for a user story is As a <role>, I want to <action> so that <benefit>. Although the other answers were close, there's a big difference between resources, personas and roles. Roles help you to identify the various perspectives that will need to be accounted for in the application.

8. Answer: D

Taskboards show everyone on the team the status of each task that's in the sprint backlog. It's a way to make sure that everyone has the same information about what's available to work on, what's in progress, and what the team has been completed.

9. Answer: C

Retrospectives start by setting the stage and making sure that the whole team is included in the conversation. Then the team reviews the information they can gather from the sprint. Once everybody agrees on the facts, they use that inforamation to generate insights on what might be causing problems for the team, Once they've identified the problem, they can figure out what they want to do to try to fix the issue.

10. Answer: A

The dotted line shows a constant burn rate for the sprint. It's normal for the number of points to fluctuate and be to the left of the line at some points and to the right of it at others. In this case, the actual completion line is far to the left of the dotted line and that indicates that team is burning story points faster than the constant rate necessary for on time completion.

11. Answer: A

Burndown and burn-up charts track the same information, the rate at which the team is completing story points. Burndowns track that rate by subtracting completed points from the total each day. Burn-ups track it by adding the number of points to the total each day.

Exam ~~Questions~~ Answers

12. Answer: C

Ishikawa diagrams are tools that are used to categorize common root causes for defects and issues in projects and help you to determine which issues fit into which categories. They're often used to help you find out where you might improve the way you work so that you can fix process problems.

13. Answer: D

The team was mapping their stories so that they could determine the best sequence for delivering them.

14. Answer: C

Card, Conversation, Confirmation is a good way to remember that user stories cards are just reminders to talk to the people who have the information you need for building a story. This is one way Scrum teams value face-to-face communication over comprehensive documentation. They try to write down only what they need out of the conversations they have about each user story card.

15. Answer: D

ESVP is used as a means of checking in with each team member at the start of a retrospective. By asking each team member to tell if they are approaching the retrospective as an explorer, shopper, vacationer, or prisoner, the team can engage each team member in the conversation and let everybody know each person's mindset from the beginning of the discussion.

16. Answer: A

It's very common for teams to have a lot of variance when they're first figuring out the scale they'll use for estimation. It's important not to be alarmed with velocity numbers vary. The goal of measuring velocity is to make the team aware of how much they are doing with each sprint so they get better at figuring out how much to take on in future planning sessions.

17. Answer: C

Personas are fake users that the team creates to help them understand how a user might be feeling when they use the software they're building.

Exam ~~Questions~~ Answers

18. Answer: C

Scrum teams know that making too many decisions up front, when you don't know as much about the situations that will come up in a project, can cause more problems than it solves. That's why they focus on making decisions at the last responsible moment.

19. Answer: B

This team is delivering more points with each successive sprint. That's a great trend to observe over a project. It can mean that the team is continuously improving as they work.

20. Answer: A

Since burn-up charts show the scope line separately from the burn-up line, it's easy to see when stories are re-estimated, added, or taken away from scope (as opposed to being completed as part of daily work).

5 xp (extreme programming)

Embracing change

Software teams are successful when they build great code.

Even really good software teams with very talented developers run into problems with their code. When small code changes "bloom" into a series of **cascading hacks**, or everyday code commits lead to hours of fixing merge conflicts, work that *used to be satisfying* becomes **annoying, tedious, and frustrating**. And that's where **XP** comes in. XP is an agile methodology that's focused on building cohesive teams that **communicate** well, and creating a **relaxed, energized environment**. When teams build code that's **simple**, not complex, they can ***embrace change*** rather than fear it.

Meet the team behind CircuitTrak

CircuitTrak is a fast-growing startup that builds software that gyms, yoga studios, and martial arts dojos use to keep track of classes and attendance.

Gary's the founder and CEO

He's a former college football player who went on to coach high school and, later, college teams. He started the company in his garage two years ago. It's been successful almost since day one, and they just moved into an office downtown. Gary's proud of the company he built, and wants to keep it growing.

Gary always looks for employees who have an athletic background because he knows they're naturally motivated do whatever it takes to get the job done.

His employees always call him "Coach" because he's as much their coach as he is their boss.

The new downtown office has the latest furnishings, and even exercise equipment so the team doesn't have to leave in order to work out.

Gyms, yoga studios, and martial arts dojos use CircuitTrak's software to manage their schedules and track their customers' attendance using a website or a mobile app.

Ana and Ryan are the lead engineers

There wouldn't be a CircuitTrak without Ana and Ryan. They were his first two employees, and have been with the company since the beginning, when it was just the three of them working out of Gary's garage. The company's up to nine people now—in the last year they hired four other engineers and two sales people—but Ryan and Ana are still the core of the team.

Ana was Gary's first hire. She played lacrosse, softball, and soccer in high school, and got a softball scholarship to put herself through college where she majored in computer science. Not long after Gary hired her, she recommended that he also hire her college classmate, Ryan. He graduated from the same computer science program a year after she did, and was also a college athlete.

Ryan set his high school swim team's 400 meter freestyle record the same week that he hacked the rival high school's website. (Nobody ever figured out that he did it.)

Ana was the only student at her college who made an All-American team and the computer science department's honor roll at the same time.

Late nights and weekends lead to code problems

Ryan and Ana built the first two versions of CircuitTrak by working 90-hour weeks filled with grit, determination, and lots of caffeine. Now that sales are growing and they're working on version 3.0, they thought they'd be able to relax a little—but they still work plenty of nights and weekends... and Ryan's worried about the effect it's having on the code.

YOU KNOW I DON'T MIND HARD WORK, BUT I THOUGHT WE'D EVENTUALLY BE ABLE TO STOP WORKING NIGHTS AND WEEKENDS... BUT WE HAVEN'T, AND IT'S CAUSING US TO **BUILD WORSE CODE.**

Ana: What's the problem? And let's make this fast, I need to get back to coding.

Ryan: That's my point! We're always up against some deadline.

Ana: Well, it's a startup. What do you expect?

Ryan: I expected this for the first year, maybe. But we've got customers now. We're growing the team. We shouldn't be scrambling like this.

Ana: That's just how software projects are, right?

Ryan: Maybe. But look at what it does to the code.

Ana: What do you mean?

Ryan: Like, look at what we did here. Remember when we had to change the way we stored group identifiers in the trainer management service?

Ana: Yeah, that was ugly. We still needed the old ID in some parts of the code.

Ryan: Right, so some code uses the old format, and some uses the new format.

Ana: Wait, weren't we supposed to clean that up?

Ryan: Add it to the **long list of things** we were "supposed to" clean up.

Ana: Well, it's basically working, right? If we clean it up now, we'll fall behind.

Ryan: So... what? We'll keep adding lousy code, and never get to clean it up?

Ana: I don't know what to tell you. I think that's just how it is.

It seems like Ryan and Ana don't have enough time to build the code as well as they want, so it's littered with scary "TODO" comments describing cleanup work they never actually have time to do.

```
public class TrainerContact
{
   // TODO: Need to clean up the ugly hack in getTrainer() when
   // we switched from integer group identifiers to GUIDs

   public Object getTrainerByOldId(String oldId)
      throws TrainerException {
         UUID trainerGroup = GroupManager.convertGuidToId(oldId);
         if (trainerGroup != null) {
```

XP brings a mindset that helps the team and the code

XP (or E**x**treme **P**rogramming) is an agile methodology that's been popular with software teams since the 1990s. XP is focused not just on project management (like Scrum is), but also on how teams actually build code. Like Scrum, XP has **practices** and **values** that help them get into an effective mindset. XP's mindset helps the team become more cohesive, communicate better, and do a better job planning—which gives everyone enough time to build the code right.

OKAY, MAYBE RYAN'S GOT A POINT. IT'S LIKE THE GREAT UCLA COACH JOHN WOODEN SAID, "BE QUICK, BUT DON'T HURRY." CAN WE FIND A WAY TO KEEP THE SCHEDULE PRESSURE OFF OF YOU GUYS?

THAT WOULD BE GREAT, COACH. BUT WE KEEP MAKING CHANGES, AND THOSE CHANGES FORCE US TO MAKE OTHER CHANGES, AND IT LEAVES US WITH CODE THAT'S ***REALLY MISERABLE TO WORK WITH.***

CHANGE, CHANGE, CHANGE. MAYBE WHAT YOU GUYS REALLY NEED TO CHANGE ***IS YOUR ATTITUDE ABOUT CHANGE.***

Gary's right. The team's been burned too many times when they made ugly, hacky changes to the code, only to spend hours doing frustrating work that feels like it could have been avoided. Now they're afraid of making any changes at all. XP can help them reach a new mindset where they're not afraid of change.

BRAIN POWER

Are any of the 12 agile principles especially appropriate here?

Iterative development helps teams stay on top of changes

The second principle behind the agile manifesto is a good description of how XP teams think about change:

> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

But wait a minute... didn't we talk about this principle earlier in the book? Yes, we did—it's also a good way to explain iterative development and the idea of making decisions at the last responsible moment. So it shouldn't be a surprise that XP is an iterative and incremental methodology. XP uses **practices** that should feel very familiar to you now that you've learned about Scrum. XP teams use **stories**, just like Scrum teams do. They plan a backlog using a **quarterly cycle**, which is broken into iterations called the **weekly cycle**. In fact, the *only new planning idea here* is a simple practice called **slack** that XP teams use to add extra capacity to each iteration.

XP teams use stories to track their requirements

It's no surprise that XP teams use **stories** as one of their core practices, because they're a really effective way to keep track of what you're planning to build. They work exactly the same way that they do in Scrum.

Many XP teams use the "As a... I want to... so that" story format, and often write their stories on index cards or sticky notes.

They'll also typically include a rough estimate of how long the story will take. It's not uncommon to see an XP team use planning poker to come up with that estimate.

Here's an example of a user story that Ana wrote on an index card. The team used planning poker to come up with an estimate, which she wrote in the corner of the card.

XP teams plan their work a quarter at a time

The **quarterly cycle** practice makes a lot of sense, because doing long-term planning each quarter feels natural: we divide the year into seasons, and many businesses typically work in quarters. So once a quarter the XP team organizes meetings to do planning and reflection.

- Meet and reflect on what happened in the past quarter
- Talk about the big picture: what the company's focused on, and how the team fits into it
- Plan the **themes** for the quarter to keep track of their long-term goals
- Plan the backlog for the quarter by meeting with users and stakeholders to pick the next quarter's worth of stories that they'll choose from

XP teams use themes to make sure they don't lose sight of the big picture. A theme is just like a sprint goal in Scrum: a sentence or two that describes what they want to accomplish.

XP teams use one-week iterations

The **weekly cycle** practice is a one-week iteration in which the team chooses stories and builds working software that's "Done" at the end the week.

Each cycle starts with a meeting where they demo the working software and plan out what they're going to accomplish by:

- Reviewing the progress they've made so far, and doing a demo of exactly what they did last week
- Working with the customer to pick the stories for this week
- Decomposing the stories into tasks

XP teams sometimes assign the individual tasks when they plan the weekly cycle, but they'll often self-organize by creating a pile of tasks and have each team member pull his or her next task off of the pile.

The weekly cycle starts on the same day each week, usually a Tuesday or Wednesday (but *usually not Monday*, to avoid pressuring the team to work over the weekend), and the planning meeting is typically held at the same time each week. The customer is typically a part of the meeting to help the team select the stories, and to stay on top of the progress.

TASK — 3 HOURS
MODIFY THE DATABASE LIBRARY AND DATABASE TO ALLOW TRAINER CLASSES TO BE REMOVED AND LOG THE ACTION

TASK — 6 HOURS
ADD A "REMOVE CLASS" API CALL TO THE TRAINER SCHEDULE SERVICE

TASK — 2 HOURS
UPDATE THE MOBILE APP UI TO ADD A "REMOVE CLASS" BUTTON TO THE PAGE THAT DISPLAYS A CLASS, HAVE IT CALL THE NEW API CALL TO REMOVE THE CLASS

Ana's idea might improve performance, so that makes for a great "nice-to-have" feature to add as slack.

I HAD AN IDEA FOR OPTIMIZING THE SCHEDULE SERVICE THAT COULD SERIOUSLY IMPROVE PERFORMANCE. I BET WE CAN **ADD IT AS A SLACK STORY** TO THE NEXT WEEKLY CYCLE.

Slack means giving the team some breathing room

Any time the team creates a plan, the team adds **slack**—another XP practice—by including optional or minor items that they can drop if they start to fall behind. For example, the team might include "nice to have" stories in the weekly cycle. Some teams like to block out "hack days" or even "geek weeks" during the quarter, where the team can work on their own work-related projects and follow up on good ideas that may have gotten swept under the rug.

Courage and respect keep fear out of the project

XP, like every agile method, depends on a team that has the right mindset. That's why XP comes with its own set of values. The first two values are **courage** and **respect**. Do those values sound familiar? They should—they're exactly the same values that you learned about earlier in Chapter 3, because Scrum teams also value courage and respect.

Courage

XP teams have the courage to take on challenges. Individual people on the team have the courage to stand up for their project.

THE OUTLOOK CALENDAR INTEGRATION FEATURE ABSOLUTELY MUST BE DONE BY THE END OF THE MONTH.

I UNDERSTAND HOW IMPORTANT THAT FEATURE IS, BUT WHAT YOU'RE ASKING JUST ISN'T POSSIBLE. LET'S FIGURE OUT WHAT WE CAN REALISTICALLY DELIVER.

Ryan doesn't like saying no to the boss, but he has the courage to do what's best for the project—which means not committing to a deadline he can't meet.

Respect

Teammates have mutual respect for each other, and every person on the team trusts everyone else to do their jobs.

Respect starts with listening to ideas and opinions you might not like and genuinely taking them into account.

IT'LL BE A TOUGH SELL, BUT IF WE SHOW UPDATED APPOINTMENTS INSIDE THE APP'S UI, I THINK WE CAN MAKE IT WORK... ***FOR NOW.***

It's easier for Ryan to have courage when everyone—especially Gary—respects his opinion. Respect goes both ways: Ryan thinks of Gary as not just the boss, but also an important member of the team, and respects his opinion and ideas too.

LET ME GET THIS STRAIGHT. XP IS ITERATIVE, JUST LIKE SCRUM. AND IT HAS VALUES JUST LIKE SCRUM, INCLUDING ***THE SAME EXACT VALUES OF RESPECT AND COURAGE***. THIS IS STARTING TO SOUND REALLY REDUNDANT. SO WHY USE XP AT ALL? WHY NOT JUST STICK WITH SCRUM?

XP and Scrum each focus on different aspects of software development.

XP, like Scrum, is an iterative and incremental methodology. But it ***doesn't*** have **the same strong focus on project management** that Scrum has—especially since it doesn't focus on empirical process control, which is a really powerful tool for teams to improve the way they manage their projects. It's also why Scrum teams feel very structured: each sprint starts and ends with their timeboxed meetings, and every day there's another timeboxed meeting at the same time.

The "P" in XP stands for **programming**, and everything in XP is optimized to help a programming team improve the way they work. XP is different from Scrum because it's focused on getting the team to work well together. XP has a lighter focus on project management, and more focus on improving the way the team builds code.

XP is specific to software development. There's nothing in Scrum that's specific to a software team—in fact, a lot of other industries have adopted Scrum to take advantage of its empirical process control.

SO SCRUM HAS A DEEP FOCUS ON PROJECT MANAGEMENT, BUT DOESN'T REALLY GET INTO THE **DAY-TO-DAY DETAILS OF HOW A TEAM ACTUALLY BUILDS CODE**. THAT'S XP'S DOMAIN, SO IT'S LIGHTER ON THE PROJECT MANAGEMENT SIDE.

XP includes enough project management to get the job done.

What ties them together are common ideas and shared values like the ones in the agile manifesto. And since iteration in XP works exactly like it does in Scrum, and XP shares the values of courage and respect with Scrum, **many agile teams use a *hybrid* of Scrum and XP** by combining the empirical process control of Scrum with XP's focus on team cohesion, communication, code quality, and programming.

BRAIN POWER

What can people on a software team do to improve how well they communicate with each other?

Venn Magnets

You spent all night arranging magnets on your fridge into a really useful Venn diagram that shows which practices and values (and even a couple of concepts) are specific to Scrum, which are specific to XP, and which are shared by both... and then someone slammed the fridge door and all the magnets fell off. Can you put them back in the right places?

Put the magnets that have values, practices, and concepts shared by both XP and Scrum in the middle section of the diagram.

Product Backlog

Retrospective

Commitment

Courage

Slack

Respect

Weekly cycle

Timeboxed Iterations

Themes

Quarterly cycle

Stories

Empiricism

Openness

Focus

Venn Magnets Solution

XP teams have a different attitude towards planning than Scrum teams do, and it's reflected in the practices and values that they share and the ones that they don't.

It takes a lot of work to manage and groom a product backlog. That's why Scrum has a full-time Product Owner on the team.

XP teams don't hold one single retrospective meeting that they hold during each iteration. Instead, they constantly talk about ways to improve how they work together as a team.

XP teams don't have a full-time Product Owner. Instead, the whole team meets with users and customers to do quarterly planning.

XP and Scrum teams share the values of respect and courage. They also use stories in exactly the same way.

Openness and commitment are really important, but only Scrum has them as part of the core value system.

Themes and timeboxed iterations are concepts, not necessarily practices or values.

Slack is a really good way to get a sense of the difference between being on a Scrum team and being on an XP team. It's literally just throwing some extra stories or tasks into the weekly cycle: it lacks the structure, empiricism, and experimental approach of Scum. But it's a "good enough" planning tool for a lot of teams.

there are no Dumb Questions

Q: How do stories get estimated?

A: Planning poker is very popular among XP teams, but they also use a lot of other methods for estimation. Early versions of XP included a practice that was called the **planning game** that guided the team through decomposing stories into tasks, assigning them to team members, and turning them into a plan for the iteration, which is still in use on a few XP teams here and there. But for most teams, estimation in XP is no different from estimation in Scrum. Techniques like planning poker are really useful, but in the end estimation is a skill: the results get better as the team gets more practice.

Q: How does a methodology "focus" on one thing or another?

A: Scrum is focused on project management and product development because the practices, values, and ideas of Scrum are specifically aimed at the problems of project management: determining what product will be built, and planning and executing the work. Scrum practices are primarily built to help the team get organized, to manage the expectations of the users and stakeholders, and to make sure everyone is communicating.

XP has a more limited approach to project management. It's still iterative and incremental, and the practices you've seen so far in this chapter—quarterly cycle, weekly cycle, slack, and stories—are an effective way to plan and manage those iterations. But they lack the structure and rigidity of Scrum: there aren't daily meetings, the meetings aren't timeboxed, and everything just feels "loose" compared to Scrum. A lot of teams find that the structure of Scrum works really well for them, so they'll opt for a **Scrum/XP hybrid** where they replace XP's quarterly cycle, weekly cycle, and slack with a ***complete*** implementation of Scrum. That means including all of the events, artifacts, and roles of Scrum.

Q: Wouldn't a "hybrid" of XP and Scrum break the rules of one of them?

A: Yes—but it's okay! If your team adopts a hybrid of XP and Scrum by replacing the planning practices of XP with the ones in Scrum, then obviously you're not performing every single practice in XP. But remember, the rules of a methodology are there to help you run your projects well. A lot of teams run into trouble when they modify an agile methodology because they don't really understand exactly why that methodology works. They often change or remove an element that may seem minor to them, but don't realize that it's one of the pillars that keeps the whole methodology up—like when teams try to replace the Product Owner in Scrum with a committee, which removes a critical piece of Scrum. Luckily, replacing XP's weekly cycle, quarterly cycle, and slack practices with a ***complete and unmodified*** implementation of Scrum impacts only the planning part of XP, and it doesn't remove any of the other pillars that make XP work, which is why so many teams have had success doing it.

Teams build better code when they work together

A software team is more than just a group of people who happen to be working on the same project. When people work together, listen to each other, and help each other solve problems, they write a lot more code (sometimes ten times as much!), and the code they build is much higher quality. XP teams know this, and the **whole team** practice helps to get them there. This practice is about everyone functioning as a team together. It means doing whatever it takes to make people feel like they belong, and helping each person support everyone else on the team.

Everyone on an XP team feels like they're all in it together. They consider building a really supportive environment to be a core practice for the team.

A whole team is built on trust

When people on an XP team encounter obstacles, they all work together to overcome them. When they're facing an important decision that affects the direction of the project, that decision is made together. That's why trust is so important to XP teams. Everyone on the team learns to trust the rest of the team members to figure out which decisions can be made individually, and which decisions need to be brought to the rest of the team.

Ryan knows that he can be open about this mistake, and the rest of the team will understand. But he also feels responsible, and will work hard to clean things up.

Trust means letting your teammates make mistakes

Everyone makes mistakes. When XP teams take the "whole team" practice seriously, people aren't afraid to make those mistakes, because each team member knows that the everyone else will understand that mistakes happen—and that the only way to move forward is to make those inevitable mistakes and learn from them together.

XP teams don't have fixed or prescribed roles

There are a lot of different jobs to do on any software project: building code, writing stories, talking to users, designing user interfaces, engineering the architecture, managing the project, etc. On an XP team, everyone does a little bit of everything—their roles change depending on the skills they bring to the table. That's one reason why XP teams **don't have fixed or prescribed roles**.

Roles can keep people from feeling a sense of belonging on the team. For example, it's not uncommon on a Scrum team for a Product Owner or Scrum Master to feel like they're not *really* part of the day-to-day work, as if their "special" role puts pressure on them to be more interested than committed. (Remember pigs vs. chickens? Sometimes it's almost as if giving someone's role on the team a name can encourage some people to be more "pig" and others to be more "chicken" on the project.)

Teams work best when they sit together

Programming is a highly social activity. Yes, really! Sure, we're all familiar with the image of the lone coder who sits in the dark for hours on end, emerging from his hole after weeks with a complete, finished product. But that's not really how teams build software in the real world. Take another look at this agile principle:

> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

When you're on a software team, you need information all the time: you need to know what you're building, how you and your team plan to build it, how the piece you're working on will fit into the rest of the software, etc. And that means you're going to have a lot of face-to-face conversations.

So what happens if you and your team sit in entirely different parts of the office? This is really common: for example, when the person laying out the office space has the "coder in the dark hole" image of software development, the space gets laid out to give managers a lot of office space, and the programming team is sprinkled into whatever space is left over. This is a really ineffective environment for a software team.

That's why XP teams **sit together**. That's a simple practice where everyone on the team sits in the same part of the office so that it's easy and convenient for each person to find his or her teammates and have a face-to-face conversation.

XP teams sit together because it's a lot easier for programmers to innovate when they can easily talk to each other, and don't have to spend a lot of time walking around in order to get the information they need.

Sit Together Up Close

The **layout of your team space** can have a big impact on how effectively everyone works together. Here's one approach that a lot of teams have found to be particularly effective.

This is a pretty effective way to arrange the team's workspace. It's a variation on a design called "caves and commons."

Each team member has a private cube where he or she can get work done without interruption.

There's a place to meet—in this case, a big table with chairs right in the middle of the team space—that's convenient for the team so they can have group discussions and meetings.

Caves and commons isn't one of the XP practices, but it's a valuable tool that helps teams with the XP "sit together" practice.

BULLET POINTS

- **XP** is an agile methodology focused on team cohesion, communication, code quality, and programming.
- XP has **practices** that help teams improve the way they work and **values** to help them get into the right mindset.
- XP teams use **stories** to track their requirements. They work exactly the same way as they do in Scrum.
- The **quarterly cycle** practice helps teams plan the long-term work by talking about the big picture, choosing **themes** (or overall goals) for the quarter, and selecting stories for the quarter's backlog.
- The **weekly cycle** practice is a one-week iteration that starts with a planning meeting where the team gives a demo of the working software, works with the customer to pick the stories for the iteration, and decomposes them into tasks.
- XP teams add **slack** to each iteration by including optional "nice-to-have" stories that can be left out if the team falls behind so they can concentrate on delivering working software that's "Done".
- Some teams use a **Scrum/XP hybrid** by replacing XP's planning practices with a complete version of Scrum.
- The **whole team** practice is about giving everyone a sense of belonging on a team.
- XP teams **don't have fixed or prescribed roles**; each team member contributes whatever they can for the team.
- Everyone on the team **sits together** in the same space.
- **Caves and commons** is a common team space layout where each person has an area that gives some privacy, and a common area that's central to the space

XP teams value communication

People on XP teams work together. They plan together, collaborate to figure out what they're going to build, and even code together. If you're on an XP team, you *truly believe* that when you're faced with a problem, you'll come up with the best solution if you communicate with your teammates. That's why **communication** is one of the XP values. One way that XP teams improve the way that they communicate is by having an **informative workspace**. This is an XP practice where the team sets up their working environment so that they can't help but constantly absorb information from it.

Communication

These are two more useful tools that help XP teams with the informative workspace practice.

XP teams make their team space informative by adding information radiators. Those are visual tools (like a big task board or burndown chart) posted on a wall that everyone can see from the work area. They "radiate" information because they're placed in a highly visible area of the workspace.

XP teams take advantage of osmotic communication, which is what happens when you absorb information about the project because you're sitting near people who talk about it—almost like it's by osmosis.

A lot of people code best when they wear headphones and tune out the rest of the world, and that's just fine. You don't need to absorb everything that goes on around you all the time.

I TWEAKED THE SCHEDULING SERVICE BY CREATING AN OBJECT THAT CACHES THE DATA IN SOME OF THE TABLES.

Ryan overheard a useful snippet of Ana's conversation and it gave him a good idea. Then he saw the burndown chart for this week's cycle posted on the wall, and realized it made sense to push it to the next iteration. The team's informative workspace improved their project.

I RAN INTO A SIMILAR PROBLEM. I BET I CAN ***REUSE ANA'S OBJECT*** IN MY CODE.

HMM, LOOKS LIKE WE'RE RUNNING A LITTLE BEHIND. I'LL HAVE TO WAIT FOR THE NEXT WEEKLY CYCLE.

Team programming is social. It's not an isolated activity.

Programmers don't like being interrupted—and for good reason. When you're on a roll coding, you get into a kind of "zone," a state of high concentration where work seems to just flow. In fact, a lot of people have a name for this state: **flow**. It's actually pretty much the same thing as when an athlete is "in the zone" (where players feel like the baseball is the size of a watermelon, or the basketball hoop looks like it's ten feet wide). This effect has actually been studied, and those studies found that it can take 15 to 45 minutes for a programmer to reach that state. An interruption, like a phone call or an annoying email, can completely break you out of flow. If you get two phone calls an hour, you can sit at your desk all day and get nothing done.

So wait a minute—doesn't that mean that to achieve maximum flow, the team should work in an environment of absolute silence? No—just the opposite! It's actually hard to concentrate in dead silence, because every time someone coughs or rustles some papers it feels like it's a freight train went by. If there's a little activity around you all the time, it's actually easier to tune it out. (And after all, athletes can get themselves in the zone even in front of screaming fans!)

Watch it!

Don't fall into the "code monkey" trap. Programming is creative and intellectual work, not just rote typing.

*If you haven't spent a lot of time writing code, you might think of programming as a "heads-down" activity: just put a programmer in a dark room in front of a computer for a few hours, and if there are no distractions he or she will just start spewing out lines of code. That's **not** how most professional software teams work. When people work together as a team, they can accomplish a lot more than if they work individually. (That's true of many kinds of teams, not just software teams!)*

Teams work best with relaxed, rested minds

Software teams need to innovate all the time. Every day brings new problems to solve. Programming is a really unique job, because it's a combination of designing new products, implementing new ideas, understanding what people need, solving complex logical problems, and testing what you built. This kind of work requires a relaxed and rested mind. XP's **energized work** practice helps everyone on the team needs to stays sharp and focused every day. Here's you energize your work:

Leave yourself enough time to do the job

Crazy, unrealistic deadlines are the easiest way destroy your team's productivity, as well as their morale and any joy they take from their work. That's one reason why XP teams use iterative development. When the team sees that they can't get all of the work "Done" for this week's cycle, they'll push some of it into the next iteration instead of trying to squeeze it all in by working late.

Get rid of interruptions

What would happen if everyone on the team turned off email notifications and silenced their office phones for two hours a day? Teams that try this find that it's a lot easier to get into flow, that state of deep concentration where you barely notice how much time has passed.

Just make sure everyone knows not to interrupt each other, because one tap on your shoulder can jar you right out of the zone.

Let yourself make mistakes

It's okay to make mistakes! Building software means constantly innovating: designing new features, coming up with new ideas, building code—and *failure is the foundation of innovation*. Every team goes down the wrong path every now and then; it's much more productive to decide as a team to think of it as a learning experience and an opportunity to learn important lessons about the code you're working on.

Work at a sustainable pace

Occasionally having a "crunch" period where you work long hours for a couple of days generally doesn't hurt, but no team can work like that forever. Teams that try find that they actually produce lower quality code, and end writing less code and getting less done than they do under normal circumstances. A **sustainable pace** means working 40 hours a week, without long nights or weekends, because that's actually the best way to get the most productivity out of the team.

This is what the agile principle about sustainable development means.

Remember this principle from Chapter 3? A good work–life balance is part of the agile mindset because it's the most productive way to run a team.

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

Who am I?

A bunch of XP practices and values are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down their names, and what kind of things they are (like whether they're events, roles, etc.).

And watch out—a couple of tools that are not XP practices or values might just show up and crash the party!

	Name	Kind of thing
I help XP teams get into a mindset where they know that they're best at solving problems when they share knowledge with each other.		
I'm a great way to absorb information about the project from discussions happening around me.		
I help you understand your user's needs, and I'm also used by a lot of Scrum teams.		
I'm a team space that does a great job communicating information about the project.		
I help the team works at a sustainable pace, because teams that work super-long hours actually build less code with worse quality.		
I'm how XP teams do their long-term planning, by meeting with the users once a quarter to work on the backlog.		
I help people get into a mindset where they treat each other well, and value each others' input and contributions.		
I'm the reason people on XP teams will tell the truth about the project, even if it's uncomfortable.		
I'm the way that XP teams do iterative development, and teams use me to deliver the next increment of "Done" working software.		
I'm a large burndown chart or task board put up in the team space in a spot where everyone can't help but notice me.		
I make sure that the team has a space where everyone is near their teammates.		
I help give the team some breathing room in each iteration by adding optional stories or tasks.		

Answers on page 66

there are no Dumb Questions

Q: I don't believe this "energized" and "sustainable" stuff. Isn't it just an excuse that programmers use so they don't have to stay late?

A: Absolutely not! Modern workplaces didn't come up with a 40-hour week by accident. There have been many, many studies over the years run across many different industries that found that while teams can work long hours for a short burst, it doesn't take long for their productivity and quality to drop off a cliff. And if you've ever had to work three 7-day, 70-hour weeks in a row, you know exactly why—your brain gets tired, and is in no condition to do the kind of demanding intellectual work needed to build great software. That's why people on XP take work-life balance really seriously: they go home at a reasonable time every day, and have lives and families outside of the job.

Q: No, I still don't buy it. Isn't programming mainly just typing?

A: Programmers may spend the day in front of a keyboard, but building code is a lot more than just typing. A programmer can write anywhere from a dozen to a few hundred lines of code in a day. But if you hand that programmer a piece of paper with a few hundred lines of code on it, it might take ten or fifteen minutes to actually type them into a computer. The "work" of programming isn't the typing, it's figuring out what the code actually needs to do, and making it work correctly and efficiently.

Q: Doesn't osmotic communication interrupt people's work? Isn't it hard to work in a noisy environment?

A: Osmotic communication works best when people on the team are used to some noise. Our ears tend to perk up when someone is talking about something important and relevant—like how you can hear your name in a crowded room—so it's not hard to tune out conversations around you if you're used to them. It doesn't work so well in a "dead quiet" office environment where everyone feels compelled to whisper, or just not talk at all.

Q: I'm still not clear on how planning works in XP. When does the team meet? How do the stories get estimated?

A: The team estimates stories together during the quarterly planning meeting which happens at the beginning of the quarterly cycle, and they talk about those estimates during the weekly planning meeting at the start of the weekly cycle. They'll also discover stories along the way, so they meet up estimate those stories together. As far as how stories are estimated, it's pretty common to see XP teams using planning poker, but they might also just talk about the story and come up with an estimate that makes sense.

Q: When does the team demo the software to the users?

A: At a meeting at the beginning or end of the weekly cycle, where the users see the software and discuss what the team will work on next. The relationship between the team and the users isn't as formal as it is in Scrum, which has a specific role—Product Owner—for a customer representative who can accept the software. XP doesn't have prescribed roles, but XP teams recognize that it's ideal to have real customers involved. Really effective XP teams feel that the "whole team" practice means treating users who help them understand what to build as a true part of the team.

Q: So wait—XP really doesn't have prescribed roles?

A: No, it doesn't. One of the basic ideas in XP is that if there's a job to do, someone will step up and do it. Everyone on the team brings something unique to the table, and each person's individual role on the project will change based on what's needed, and what their expertise is.

Q: I've heard programmers complain about being assigned "maintenance" tasks, like fixing bugs in old systems. Is that really creative or innovative?

A: Actually, maintenance work can be some of the most intellectually challenging work there is for a software team. Think about what "maintenance" actually means: fixing bugs, often in code you didn't even write yourself. That means taking a machine, one that may be really complex, figuring out how it works (often without much documentation and nobody to ask for help), tracking down how it's broken, and figuring out a way to fix it. Programmers often groan about having to do maintenance, and that gives it a reputation as being "grunt" work: it's intellectually demanding and, unlike coming up with a cool new feature, it's rarely rewarded or complimented by your boss or coworkers.

People on XP teams take work-life balance really seriously: they go home at a reasonable time every day so they can keep up a sustainable pace

Yes! XP teams give themselves space to build great code.

Everything we've talked about so far has been about removing things that kill the team's momentum. Iteration, slack, and stories help the team build the right software, and protect them from unnecessary schedule pressure. An energized workspace, a supportive team, and an informative team space give them the best possible environment to get work done. It's not an accident that XP focuses on these things—they're at the root of the vast majority of team problems, and eliminating them gives the team fertile ground for innovation.

So now the stage is set, and the team is ready. **It's time to dig into the code.**

BULLET POINTS

- **Communication**—one of XP's values—is what matters most in a software project.
- The **informative workspace** practice means anyone can walk into the team space and get a sense of how the project is going just by looking around.
- People absorb information via **osmotic communication** when they sit together and overhear useful discussions.
- The **energized work** practice is how the team stays relaxed, rested, and in the best mental shape for work.
- **Information radiators** are large visual tools like task boards or burndown charts that "radiate" information because they're posted in a place that's hard to miss.
- XP teams work at a **sustainable pace** so they don't burn out. This typically means working regular hours.
- An energized team has enough time to do the job, and **freedom to make mistakes**.
- Interruptions can break a developer's concentration and take him or her out of **flow**, a state of high concentration where he or she is "in the zone."

Question Clinic: The "which-is-NOT" question

YOU'LL SEE SOME QUESTIONS ON THE EXAM THAT LIST VALUES, PRACTICES, TOOLS, OR CONCEPTS AND ASK YOU TO DETERMINE WHICH ONE OF THEM IS NOT PART OF THE GROUP. USUALLY, YOU CAN FIGURE THEM OUT BY GOING THROUGH THE ANSWER CHOICES ONE BY ONE AND ELIMINATING THE ONE THAT DOESN'T BELONG.

97. Which of the following is NOT shared by both XP and Scrum?

A. Timeboxed iterations
B. Stories
C. Respect and courage
D. Slack

XP and Scrum are both iterative. XP uses weekly cycles and Scrum uses sprints. So this isn't the right answer.

You definitely find stories on both Scrum and XP teams, so this one's not right either.

The values of respect and courage are shared by both Scrum and XP teams. So the answers include values as well as practices and tools.

D's definitely the right answer: slack is NOT shared by both XP and Scrum. XP teams use slack by including extra stories in their weekly cycles that can be skipped if the other stories take longer than expected. Scrum teams have a much stronger focus on project management, and have much more detailed planning practices and tools.

Keep your eyes out for questions that ask for "all of the following except" (that's just another way of wording a which-is-NOT question).

TAKE YOUR TIME AND THINK YOUR WAY THROUGH IT. ALL OF THEM WILL HAVE SOMETHING IN COMMON BUT ONE. AS LONG AS YOU REMEMBER THE GROUP YOU'RE FITTING THEM INTO, YOU WON'T HAVE ANY TROUBLE.

Take your time answering which-is-NOT questions.

HEAD LIBS

Fill in the blanks to come up with your own "which-is-NOT" question!

Which of the following is NOT a _____ ?
(value, practice, tool, or concept)

A. _____
(value, practice, tool, or concept that is in the group)

B. _____
(the right answer)

C. _____
(value, practice, tool, or concept that is in the group)

D. _____
(value, practice, tool, or concept that is in the group)

LADIES AND GENTLEMEN, WE NOW RETURN YOU TO CHAPTER FIVE

File Edit Window Help XP

WARNING: The rest of this chapter talks about code

The P in XP stands for programming, and for good reason. While the XP practices you've seen so far apply to any team doing creative or intellectual work, the practices in the rest of this chapter are specifically focused on code.

If you're not a programmer, it's still worth reading the chapter! However, some of the material is a little more code-oriented than what you're used to seeing. But if you plan on working with a team that builds software, some familiarity with these ideas can be REALLY VALUABLE to you and your team. Understanding your teammates' perspectives better can help you reach a more agile mindset together.

If you don't have any background in programming, you should feel comfortable skipping over the sections of this chapter that contain snippets of code. Just make sure to read all the text, paying special attention to the words in boldface. If you do the exercises and test your knowledge with the crossword and exam questions at the end of the chapter, that's a very effective way to get the most important parts of XP into your brain.

And if you're using this book to study for the PMI-ACP® exam, don't worry - the exam does NOT require you to have programming knowledge.

LATER, BACK IN THE TEAM'S MEETING SPACE...

Ana: Quit your whining, Ryan.

Ryan: Hey, don't give me that attitude. This affects you, too.

Ana: OK, I'm listening. What's the problem?

Ryan: You won't like this. It's a change to personal trainer schedules on the mobile app.

Ana: Customers getting notifications about personal training sessions. What's the problem?

Ryan: The problem is they don't just want to get notifications. They want to schedule classes from the mobile app too.

Ana: Oh no. No, no, no. That is not going to work with the way we built this.

Ryan: Tell that to Coach. He's been promising that to the customers.

Gary: Did I hear someone mention me?

Ana: You promised customers that we'd let them schedule classes from the app?!

Gary: What's the problem, guys? How hard can it be to add that?

Ana: We're going to have to completely redesign the way data goes into the system.

Ryan: You know what's frustrating? ***If you'd just told us this was coming a few months ago***, we would have built a completely different back end for the last version.

Ana: Now we have to rip out the database entry code and replace it with a new service.

Gary: I know you guys can do it.

Ryan: Of course we can. But rewriting that much code will leave us with a ***giant mess***.

Ana: You know how they say rework creates bugs? This is a prime example.

Ryan: And that means a lot of ***totally avoidable late nights***. This stinks.

Back in Chapter 2 we learned that rework is a major source of bugs. Does that mean rework always causes bugs?

XP teams embrace change

Here's a basic fact about software projects: they change. A lot. Users ask for changes all the time, and typically have no idea how much work any one change will require. This wouldn't be to bad, but there's a problem: a lot of teams build code that's *difficult to modify*: changes require code modifications that are painful to make, and leave the code in very bad shape. This often leads to teams that push back against those changes. When teams *resist* change, the project suffers. The XP values and practices fix this problem at its source by helping teams to build code that's easier to modify. And when the code is easy to modify, programmers don't feel the need to resist change. That's why **XP has practices and values that are focused on programming**. Because these practices helps teams build code that's easier to modify, XP helps them reach a mindset where they ***embrace change*** instead of resisting it.

People on software teams resist changes when they've had bad experiences with rework causing bugs... but rework doesn't have to do that. XP helps teams embrace change with practices and values that help them to build software that's easier to modify.

Ever heard a programmer complain about spaghetti code? That's when the code's structure is complex and tangled structure (like a pile of spaghetti in a bowl). It's often the result of rework that results in many changes to the same part of the codebase. **Programming doesn't have to be that way!** XP teams have practices and values that make their code easier to modify, so the team can do rework without turning the code into a mess.

Frequent feedback keeps changes small

Talk to a group of programmers, and it often doesn't take long before someone starts to complain about how users always change their minds. "They ask for one thing, but when we build it exactly like they wanted they turn around and tell us they need something totally different. Wouldn't it be easier if we just built the right thing in the first place?"

But ask that same group of programmers how often they designed and built an API, only to find that some of the functions were awkward and difficult to work with. Wouldn't it be easier if we just built the right API in the first place? Obviously. But you don't really know if the interface you designed and built is easy to use until someone writes code that actually uses it.

An API ("application programing interface") is a set of functions that you build into your system so that another programmer can write code to control it.

When you put it that way, programmers recognize that it's really rare to build anything right the first time, so they try to get feedback early and continue to get it frequently. That's why XP teams value **feedback**.

And feedback comes in many forms:

Iteration

You've already seen a really good example of feedback: iteration. Instead of planning six months of work doing one big demo at the end of it, your team will do a small chunk of work and then get feedback from the users. That lets you **continually adjust the plan** as the users learn more about what they need.

Integrating code

When code files on your computer are out of date with the rest of your team, it can lead to frustrating problems. When you **integrate** your new code frequently with your teammates' code, it gives you early feedback. The more frequently you integrate, the earlier you catch conflicts, which makes them a lot easier to resolve.

Feedback

Teammate reviews

Open source teams have an old saying: "Given enough eyeballs, all bugs are shallow." Your team is no different. Getting **feedback from your teammates** helps you catch problems with your code—and it helps them understand what you built so they can work on it later.

That's called Linus's Law, named after the creator of Linux.

Unit tests

One really effective way to get feedback is to build **unit tests**, or automated tests that make sure the code that you built works. Unit tests are typically stored in files along with the rest of the code. When you make a change to your code and it breaks a test, that's some of the most valuable feedback you can get.

Bad experiences cause a rational fear of change

There's very little that's more frustrating to a programmer than being halted in your tracks by an annoying, frustrating problem. Most developers will recognize these very common—and very frustrating—problems that Ryan and Ana are running into.

NOOOOOOOO! SOMEONE JUST COMMITTED THREE WEEKS OF CHANGES ALL AT ONCE, AND NOW **I'M GETTING DOZENS OF CONFLICTS** WHEN I TRY TO COMMIT MY CODE.

The CircuitTrak code is stored in a **version control system**. That's a software tool that teams can use to work on a single set of files without overwriting each others' changes. It keeps track of every code change, letting you see who made the change and see what the code looked like before and after.

IT'S GOING TO TAKE ME HOURS TO SORT OUT ALL THESE CONFLICTS. THIS IS A **NIGHTMARE!**

When you add your latest code changes to a version control system, it's called a **commit**.

Ana's been working on a change that affects a lot of different files. One of her teammates spent the last few weeks modifying many of those files, and just committed his changes. The good news is that when she tried to commit her code, the version control system detected the conflicts and rejected her update. The bad news is that now she has to painstakingly reconcile each of her changes with the ones her teammate made.

This is going to be so much work, Ana's considering just rewriting the whole thing from scratch rather than try to merge her changes.

I STARTED FIXING A BUG, BUT TO GET IT TO WORK I HAVE TO FIX THESE *OTHER TWO* PARTS OF THE CODE...

...AND ONE OF *THOSE* CHANGES AFFECTS *THIS OTHER PIECE*, AND *THAT* REQUIRES *TWO MORE* CHANGES OVER HERE...

Ryan started making one little change to the code, but somehow it rippled out to many other areas of the software. By the time he gets all the way down the chain of fixes, he's practically forgotten what he was supposed to be doing in the first place. This is a really familiar feeling to a lot of programmers, and it's got a name: **shotgun surgery**.

...URGH, SO MANY CHANGES IT **HURTS MY BRAIN!**

How a version control system works

Behind the Scenes

Ana, Ryan, and the rest of the team have been working on CircuitTrak for years, and the project now has thousands of files, including source code, build scripts, database scripts, graphics, and many other files. If they just used a shared folder on the network to hold the files, it would quickly become a mess:

10am: Ana copies the source folder to her computer so she can work on the code

11:30am: Ryan copies the source folder to his computer so he can work code too

1pm: Ana copies updated TrainerContact.java back to the shared folder

3pm: Ryan copies updated TrainerContact.java back to the shared folder

Oops! When Ryan saved his changes, they overwrote Ana's changes. That's going to cause bugs later!

That's why the team uses a version control system, which provides a **repository** that contains not just the latest copies of each file, but also a complete history of changes. It even lets multiple people work on the same file at once:

10am: Ana checks out the source from the repository to a working folder on her computer

11:30: Ryan has the source checked out, so he updates it with the latest changes

1pm: Ana commits her TrainerContact.java changes back to the repository

3pm: Ryan commits changes to a different part of TrainerContact.java

Ryan and Ana changed different lines of the file, so the version control system was able to merge their changes automatically.

Things are a little messy (but still manageable) when two people make **conflicting changes** to the same file.

10am: Ana and Ryan both have working folders updated with the latest source

1pm: Ana commits changes to TrainerContact.java back to the repository

2:30pm: Ryan tries to commit a conflicting change but it's rejected

In this case, there were only a few conflicts, so Ryan was able to resolve them pretty easily and commit the updated code. But when there are a LOT of conflicts, the merge can be really difficult.

When Ryan tried to commit his change, the system found that Ana had already checked in different changes to the same lines in the same file, so it rejected the change and updated his local file to show both sets of changes. Ryan had to resolve the conflicts before he could commit any more code.

XP practices give you feedback about the code

A lot of agile practices are crafted to give the team feedback early and often—like the ones focused on iteration, which give the team feedback about the product they're planning to build and the work involved. Each iteration gives them more information, and they use that information to improve how they plan the next iteration. This is an example of a **feedback loop**: the team learns from each round of feedback, makes adjustments and self-corrects, which changes what they learn about in the next round. These **next four XP practices** are especially good tools because they give the team really good feedback about how they design and build the code.

XP Practice

Pair programming

Two team members sit with each other in front of the same computer and work together to discuss, design, brainstorm, and write the code.

→ **Team members give each other feedback about the code they're building. They switch pairs often, which helps everyone stay on top of how the whole codebase is changing.**

XP Practice

10-Minute Build

The team maintains an automated build that compiles the code, runs the automated tests, and creates the deployable packages. They make sure that it runs in ten minutes or less.

→ **You learn a lot about your code when you try to build it and see what breaks. When the build runs quickly, everyone on the team is comfortable running it as often as they need to.**

Under the Hood: Build Automation

Here's a quick overview of how an automated build works, just in case you haven't used one before.

Automated builds turn source code into packaged binaries

If you're not a programmer, you may not be 100% clear on the mechanics of how software is created. Exactly what do programmers type all day, and how does it turn into software that you can run? Here's what's going on:

- Software typically starts out as a set of **text files that contain code**. This is the source code of the project.
- Programming languages have compilers that **read the source code files and create a binary**, or the executable file the computer's operating system is able to run.
- The binary typically needs to be **packaged** into a single file that contains the binary and any additional files that are needed to run (like an executable installer, a mountable disk image, or a deployable archive file).
- Compiling the source code and packaging it up by hand can be **time-consuming and error-prone**, especially if there are multiple binaries and many files that need to be bundled into a single package.
- This is why teams **automate the compile and package steps** to create the binaries and other files. There are many tools and scripting languages that make it easy for teams to create automated builds.

A system that gives a lot of feedback will often **fail fast**. You want your system to fail quickly so you can fix problems early, before other parts of the system are added which depend on them.

XP Practice

Continuous Integration

Everyone on the team constantly integrates the code in their working folders back into the repository, so that nobody's working folder is more than a few hours out of date.

→ When each person has the latest code in a working folder conflicts show up immediately, and they're a lot easier to fix when they're caught early.

A 10-minute build really helps with both continuous integration and test-driven development because it executes the unit tests, so you find out quickly if you add code that breaks an existing test.

XP Practice

Test-driven development

Before a team member adds new code, the first thing he or she does is to write a unit test that fails, and only after that does he or she write or modify code to make it pass.

→ Unit tests set up a tight feedback loop: build the failing test, write code that makes it pass, learn more about what you're building, write another test, repeat.

DICTIONARY DEFINITION

re-factor, verb

to change the structure of code without changing its behavior

A particularly troublesome block of code that was much less frustrating to work with after Ryan took the time to ***refactor*** *it.*

Developers typically run unit tests using a specialized program (often a plug-in for a build tool or a development environment). The unit test results are usually displayed with color codes: passing tests are green, failed tests are red. Teams that use test-driven development typically follow a cycle of adding failing tests that start off red, making them pass so they turn green, and then refactoring the code. Teams refer to this cycle as red/green/refactor, and consider it a valuable development tool.

Ready to take a closer look at each of these practices? Flip the page! →

XP teams use automated builds that run quickly

There's nothing more frustrating to a programmer than waiting. That's a good thing: a lot of innovation starts with a programmer saying, "I can't stand how long this takes." So it's especially frustrating when it takes a lot of time and effort to build the code—and very few things can kill a team's innovation as quickly as frustration. When something repeatedly takes a long time, the first thing that a good programmer thinks is, "How do I automate that?"

That's where the **10-minute build** practice comes in. The idea is straightforward: the team creates an automated build, usually using a tool or scripting language specifically made for automating builds—and they do it at the beginning of the project. The key here is that they make sure the whole build runs in under ten minutes. That's pretty much the limit of patience most programmers have to wait for the build to finish—and it's long enough so that you can kick off a build, then go grab a cup of coffee and think. By keeping the build under ten minutes, there's no hesitation in running it, which helps find build problems quickly.

When the build requires a lot of manual effort or takes longer than **10** minutes to run, it puts stress on the team and slows down the project.

Continuous integration prevents nasty surprises

When you work on a team building code and committing it to a version control system, your day-to-day work follows a pattern. You do some work, then you update your working folder with the latest changes your teammates have pushed, then you push your own changes back to the version control system. Work, update, push... work, update, push... work, update... ***merge conflict***! Uh-oh—one of your teammates made changes to the same line and committed it since the last time you updated. The version control system had no way of knowing whose change is right, so it modified the code files in your working folder with both sets of changes. Your job is to **resolve the conflict**: you'll look at them, figure out what the code is supposed to do, modify it so that it's correct, and commit the resolved change back to the repository:

When you try to commit a conflicting change, most version control systems add marks like this to the files in your working folder so you can see exactly what conflicts need to be resolved.

```
/**
 * Find students by matching a partial name
 * @param partialName Name of the student to search for
 * @return Student collection with the results of the search
 */
StudentCollection findStudentsByPartialName(String partialName) {
   StudentRecordCollection records = getStudentRecords(searchString);
<<<<<<<
    RecordManager.lookupRecord(records);
    StudentCollection studentsFound = new StudentCollection();
    records.toList(studentsFound);
=======
    StudentCollectionHelper.buildStudentCollection(records);
>>>>>>>
    return studentsFound;
}
```

This code was committed since the last time you updated your working folder.

Here's the conflicting change you tried to add. Resolving the conflict means looking at both changes and figuring out how it needs to work.

Every merge conflict is like a little puzzle, and sometimes those puzzles can be annoying to solve because you're not sure exactly what your teammate was trying to do.

Now flip back to page 30. Do you see why Ana ran into so much trouble?

Her teammate hadn't updated his working folder in weeks. Instead, he just kept making changes to an old version of the code got which more out of date every day, and then committed all of those changes at once. Ana spent the last few hours working on a change to many of those same files. But instead of having one or two little puzzles to solve, now she has to contend with dozens of files marked up with conflicts. Few things are more frustrating to a programmer than resolving many merge conflicts at once.

That's why XP teams use **continuous integration**. It's a really simple practice: every person on the team integrates and tests their changes every few hours, so nobody's working folder is ever out of date. When everyone on the team does continuous integration, they're all working on a current version of the code. There will still be merge conflicts, but they're almost always small and manageable, and never a giant, frustrating monster of a change like the one that Ana has to deal with.

When everyone keeps their working folders up to date, the merge conflicts tend to be small and manageable.

The weekly cycle starts with writing tests

For many developers, XP brings a different way of working. One of the most obvious changes is that the team does **test-driven development** (or TDD). That's a practice where programmers write unit tests before they write the code that it tests. When you're in the habit of writing unit tests first, you think about what it means for your code to work correctly, which helps you write code that's *"Done"* done.

TDD forces you to really think through your code's behavior before you dive into writing it.

OKAY, THIS DOESN'T MAKE SENSE. HOW DO YOU TEST CODE THAT YOU HAVEN'T WRITTEN YET? ***WHY DOES IT MATTER*** IF I WRITE THE TESTS FIRST OR WRITE THE CODE FIRST?

TDD makes it really obvious when you add extra unnecessary dependencies between different parts your code, and those dependencies are exactly what cause the 'shotgun surgery' feeling.

Unit tests change the way the team designs the code

Every programmer's had the experience of writing code, only to wish later that he or she had written it a little differently—maybe you would have chosen different arguments for a function, or used a different data structure, or made other choices. But now the code you wrote is called from five other places, and it will be more work to change it than it will be to just live with the poor decision.

In other words, some of the most annoying code problems happen when you make a bad design choice, then you add other code that depends on it. Do this enough and eventually you get that "shotgun surgery" feeling every time you touch that part of the code.

Unit testing helps prevent that problem. Design problems in your code often become apparent the first time that you write code that uses it. And that's exactly what you're doing when you write a unit test first: you ***use the code that you're about to write***. And you do it in small increments, one bit at a time, smoothing out design problems as you encounter them.

Test-Driven Development Up Close

```
public class ScheduleFactory {
```

```
public class TrainerManager {
```

```
public class UserInterfaceModel {
```

Code is always divided into discrete units

In some languages the units are classes; in others they're functions, modules, procedures... the specific unit varies from language to language, but every programming language works this way. For example, when you write Java code, most of your code goes into "chunks" called classes saved in `.java` files. Those are units of Java code.

Each unit gets its own unit tests

The name "unit testing" is pretty self-explanatory: you write tests for the units of code. For example, in Java unit testing is typically done on a class-by-class basis. Those tests are written in the same language as the rest of the code, and are stored in the same repository. The tests access whatever part of the unit is visible to the rest of the code—for Java classes, that means the public methods and fields—and use them to make sure the unit works.

```
public class ScheduleFactoryTest {
```

```
public class TrainerManagerTest {
```

```
public class UserInterfaceModelTest {
```

The XP mindset helps you think differently about programming, design, and code because its practices give you good habits that keep your code clean, simple, and easy to maintain. TDD is one of those good habits.

This is just like how the Scrum mindset helps you think differently about planning.

Writing the unit tests first forces the developer to think about how the code is going to be used

Every unit of code is used by at least one other unit somewhere in the system—that's how code works. But when you're writing code, there's a paradox: in a lot of cases, you don't really know exactly how the unit you're working on will be used until you actually use it.

Test-driven development helps you catch problems in your code early, when they're much easier to fix. It's surprisingly easy to design a unit that's difficult to use later, and just as easy to "seal" in that poor design by writing additional units that depend on it. But if you write a small unit test every time you make a change to a unit, a lot of those design decisions become obvious.

HMM, I DIDN'T REALIZE HOW WEIRD THIS CLASS WAS UNTIL I STARTED WRITING CODE IN A UNIT TEST THAT USES IT. I'M GLAD I CAN FIX IT NOW **BEFORE ANYTHING ELSE DEPENDS ON IT!**

Agile teams get feedback from design and testing

Agile teams have great design and testing tools that help the teams get more feedback throughout the project. They can use **wireframes** to sketch out user interfaces before building them, **spike solutions** to figure out difficult technical problems, and **usability testing** to make sure they've made effective design choices. All three of these tools are great at generating feedback, which is why XP teams integrate them into their weekly cycles—and rely on them to get feedback to help plan the next weekly cycles.

Wireframes help the team get early feedback about the user interface

Of all the things that software teams build, user interfaces seem to generate the most opinions from users and stakeholders, so they want to get feedback about the UI early and often. That's why teams use wireframes to sketch out user interfaces. There are a lot of different ways to create wireframes. Some are basic sketches of the system's navigation, while others are highly detailed representations of individual screens or pages. It's a lot easier to modify a wireframe than it is to modify code, so teams often review several iterations of each wireframe with the users.

A small change to the user interface can have a huge impact on usability. That's why wireframes and usability testing are so important.

Wireframes are often low fidelity: sketches drawn by hand or with a program that gives them a hand-drawn look. This encourages users to feel more comfortable suggesting changes than if they were more polished. Some users are hesitant to ask for changes if a UI looks really polished because it feels like they're asking the team to do a lot of extra work. Making wireframes look hand-drawn increases the amount of feedback that they generate.

Build spike solutions to get an idea of a feature's technical difficulty

It's not uncommon for a team to have trouble estimating a specific feature because they just don't know enough about what's involved in solving specific technical problems. That's where spike solutions come in handy. A spike solution is code written by a team member specifically meant to figure out a specific technical problem. The only purpose of the spike solution is to learn more about the problem, and the code is usually thrown out after it's done.

When teams talk about usability they're trying to measure how easy it is to learn and use the software. It's really common to discuss the usability of a program's user interface, or the visual interface (like windows or web pages) that users interact with to use the system.

Usability testing means testing your user interface on real users

When you're trying to figure out how effective a user interface is, there's no substitute for getting it in front of real live users and watching how they interact with it. That's what usability testing is all about: sitting users down in front of an early version of the UI that the team has been building and having them use it to perform tasks they'll typically need to use it for. When XP teams do usability testing, it's often done near the end of the weekly cycle so that the information they learned from it can be used in the next one, setting up an extremely valuable feedback loop.

Spike Solutions Up Close

A spike solution helps you solve a tough technical or design problem.

A spike solution is a simple program whose only purpose is to explore solutions to a problem. It's usually timeboxed to a few hours or even a few days, and after the spike is done the code is usually thrown away or set aside (so the team can use it later if they want). This gives the programmer a lot of freedom to focus single-mindedly on solving the problem and ignore the rest of the project. But even though the code is thrown away, the spike is still treated as real project work. The team will typically add a story for it to the weekly cycle.

Architectural spike

When XP teams talk about doing a spike solution, they're usually referring to an **architectural spike**. An architectural spike is used to prove that a specific technical approach works. Teams will often do an architectural spike when they have a few different options for designing a specific technical solution, or if they don't know if a certain approach will work.

Risk-based spike

Sometimes there's a problem that presents a project risk: the developers are pretty sure it will go well, but if it doesn't it could seriously derail the project. That's when the team will do a risk-based spike. It works just like an architectural spike, but with a different goal: it's done specifically to remove a risk from the project.

Spike solutions fail fast: if the programmer discovers that the approach won't work, the spike ends... and the team still considers it to be a successful spike.

Sharpen your pencil

Here are three scenarios that Ryan and Ana are working on that have to do with getting feedback from their project. Write down the name of the tool being used in each scenario.

WE NEED A NEW WAY TO STORE TRAINER SCHEDULES THAT WILL REDUCE MEMORY. I'M BUILDING A PROTOTYPE SO WE CAN GET A SENSE OF HOW MUCH WORK IT WILL BE.

I'M NOT HAPPY WITH HOW THIS CLASS GETS INITIALIZED – IT'S GOING TO BE HARD TO USE. ONCE ITS UNIT TESTS PASS, I'LL MODIFY IT.

I FINISHED DESIGNING THE NEW USER INTERFACE. LET'S MAKE SURE THAT IT WORKS BY GETTING A BUNCH OF USERS IN THE ROOM AND OBSERVING THEM WHILE THEY USE IT.

Answers on page 68.

Pair programming

XP teams use a pretty unique practice called **pair programming**, where two people sit at a single computer and write code together. This is a new experience for people who are used to thinking of programming as a solitary activity. But it can really be an effective tool for building high-quality code very quickly, because many people who do pair programming report that pairs get more work done together than they do when they work separately.

Pair programming keeps everyone focused, helps the team catch bugs, makes it easy to brainstorm, and gets everyone on the team involved in every part of the codebase.

Ryan and Ana help keep each other focused.

When Ryan gets stuck, Ana can jump in and keep things moving, and vice versa.

They're constantly talking about the problems they're working on and brainstorming solutions.

You're a lot less likely to take shortcuts when someone else is working alongside you.

The whole team constantly rotates pairs, so everyone gets experience working with every part of the system.

You don't always know if you have a good handle on an idea until you explain it to someone else.

There are two sets of eyes on every change, and they catch a lot of those little mistakes that would cause headaches later if they were missed.

Sharpen your pencil

The XP practices are useful individually, but when you combine them they're especially effective. We've written down several of the XP practices, and drawn arrows between them. Each arrow has blank lines for you to write on. For each set of blank lines, write down one way that the practice the line is drawn FROM can interact with the practice the line is drawn TO in a way that reinforces and supports it.

We've started you out by filling in this blank to show how "sit together" impacts "informative workspace".

Sharpen your pencil Solution

There are a lot of different ways to solve this, because there are many ways that the XP practices can interact to reinforce and support each other. We've written down ways that we think are important. Did you come up with similar answers to ours?

The XP practices work together to form an ecosystem that leads to better, flexible, more maintainable code.

there are no Dumb Questions

Q: The "sit together" and "pair programming" practices require everyone to be in the same office. Does that mean global or distributed teams can't use XP?

A: Many global and distributed teams use XP. XP teams know that when they sit together, they have more face time, have fewer interruptions from phone calls, and can share an informative workspace. A distributed team where everyone works in different offices and communicates via email and phone can't take advantage of these things. But an important part of the XP mindset is that every practice is about making the team work better. If there are some practices that are simply impossible, they'll work with what they've got.

Q: But doesn't that mean they aren't doing "pure" XP?

A: Really effective teams know that there's ***no such thing as "pure" XP***. XP teams are always looking for ways to improve. There's no "perfect" state that they're trying to get to; they're just trying to get better at what they do together. Mindlessly adhering to practices will de-energize the environment really quickly. And making people feel bad about not being "pure" enough is disrespectful. **Nagging people about XP "purity" is counterproductive.** It makes people feel like you're judging them and their work. That won't make anyone change—it'll just make them resent you, and resent XP.

Q: So does that mean it's okay to throw out practices I don't like?

A: No, that's not okay. The XP practices are carefully designed to work with each other, and when they're used together they help the team integrate XP values into their mindset. For example, teams really start to understand the XP value of communication when they sit together and have an informative workspace. When teams decide to throw out a practice, it's usually because their mindset is incompatible with one of the values, and that causes the practice to feel uncomfortable. When that happens, a really good thing to do is to **make a genuine effort to try the practice**. Often, that helps the team shift their mindset, which in turn helps everyone work better together and build better software.

Q: Doesn't continuous integration just mean setting up a build server?

A: No. A build server is a program that periodically retrieves the latest code from the version control system, runs the automated build, and alerts the team if there are any failures. It's a really good idea, and almost all agile teams use one. But a build server isn't the same thing as continuous integration. Continuous integration means that every person on the team actively (and continuously!) integrates the latest code their teammates wrote into his or her own working folder. The reason this is often given the same name as a build server is that the server is constantly "integrating" code from the version control system into its own repository, and will alert the team any time code is committed that won't compile or causes test failures. But that's no substitute for having each person keep his or her working folder up to date.

Q: I don't get it. If we have a build server that constantly integrates the code, isn't that less work for everyone?

A: It's true that having every member of the team continuously integrate the latest code from the version control into their working folder is more work than just setting up a build server. But if they just rely on email alerts from a build server to tell them when they're out of sync, it often ends badly. For example, you might discover that when you commit code that breaks the build, everyone gets really mad at you, so you commit your code a lot less frequently than you normally would. Or the team might just be so used to "broken build" emails from the build server that they start ignoring them and filing them into folders. On the other hand, if every person feels like they have a responsibility to stop what they're doing every few hours and integrate code from the version control system back into their own working folders, a broken build is rare—and when it happens, the team notices quickly and works together to fix it.

Q: So is doing continuous integration just a matter of making sure the team has enough discipline?

A: Not quite. When a team is really good at using practices like continuous integration, 10-minute builds, or test-driven development, from the outside it looks like they're really disciplined. But it's not really a matter of discipline at all. The team does those things ***because they make sense to everyone***. Everyone on the team simply feels that the work will slow down if, say, they don't take the time to make the build faster, or build a unit test before writing code. They don't need to be nagged, yelled at, or reprimanded—in other words, disciplined—because it wouldn't occur to them *not* to do those things.

Q: I work with a QA team. Does test-driven development mean testers write my unit tests while I write the code?

A: No. You write your own unit tests first, then you write the code to make them pass. The reason that the unit tests should be written by the same person who writes code is that when you write the tests, you learn a lot about the problem that you're working on, which makes the code better.

SOMETHING'S REALLY BUGGING ME. PAIR PROGRAMMING SEEMS LIKE A GIANT WASTE OF TIME. WHEN TWO PEOPLE WORK TOGETHER, DOESN'T IT CAUSE THEM TO **DELIVER CODE HALF AS FAST**?

Pair programming is actually a really efficient way to code.

Pairing up keeps you focused and eliminates a lot of distractions (like popping open a browser or checking your email). And there's always another set of eyeballs to catch bugs early instead of wasting time tracking them down later. But more importantly, it means you're **constantly collaborating with your teammates**. Programming is an intellectual activity: writing code means solving problems and puzzles all day long, one problem or puzzle after another. Talking through those puzzles and problems with one of your teammates is a really effective way to solve them.

That's why even people who are initially resistant to pair programming often find that they really like it after genuinely trying it out for a few weeks.

Is it really fair for us to use the word "irrational"? We think so. Pair programming is a straightforward and—let's face it—unremarkable way to work, and many people do it every day. A very intense and negative emotional reaction to something so mundane is, by definition, not rational.

OKAY, I GET YOUR POINTS. BUT... ARE YOU SURE? HONESTLY, **I JUST DON'T BUY IT**. PAIR PROGRAMMING SIMPLY DOESN'T FEEL RIGHT TO ME.

A practice "just doesn't feel right" when it clashes with your mindset.

Do you think of yourself as a better programmer than everyone around you? Is coding a solitary activity in your mind? If so, then you'll have an irrational dislike of pair programming. Do you think of yourself as a "rock star" surrounded by idiots who couldn't code their way out of a paper bag? Then you'll have an *extremely strong irrational hostility* to pair programming. The key word here is **irrational**: yes, you can think of reasons and rationalizations for disliking pair programming, but at the heart of it what you really have is a *feeling that it's just not right* for you or your team. And that's the definition of irrational: decisions driven by feelings, not reason.

But it turns out that a lot of really good programmers on real-world projects have discovered that not only can their "lesser" teammates (surprisingly!) keep up with them, but when they *genuinely* try pair programming—not just go through the motions, but *really try* to make it work—coding really does go a lot faster. Not only that, but their "slower" teammates start to pick up many of the skills and techniques they've learned, and the whole team improves together.

SORRY, I STILL DON'T BUY IT. PAIR PROGRAMMING IS **BAD**, AND NOTHING YOU SAY IS GOING TO CHANGE MY MIND. DOESN'T THAT MEAN ALL OF XP IS ***WRONG FOR ME AND MY TEAM***?

Then you and your team don't value the same things that XP teams do.

XP teams value focus, respect, courage, and feedback. If you really value these things, pair programming makes a lot of sense. When you value focus, you appreciate how pair programming helps keep you and your teammates on track. When you value respect, you won't have an irrational response to the idea of pairing up with your teammates, because you have respect for them and their abilities. When you value courage, then you'll be willing to look past your own feelings of discomfort and try something could potentially help the team. And when you value feedback, then having two eyes on every line of code that's written feels like a really great idea.

On the other hand, if the last few sentences seem cliché, oversimplistic, overly idealistic, or even stupid, **then you don't share the same values** as effective XP teams.

When you try to adopt a practice that doesn't match your mindset or the culture of your team, it usually doesn't "take" and you just end up going through the motions.

SO WHAT? WHAT HAPPENS IF I DON'T SHARE XP'S VALUES?

Adopting new practices takes work, and shared values motivate everyone to do that work.

When teams try to adopt a methodology with values that don't match the team's culture, it usually doesn't end well. The team will try adding some of the practices, and a few of them may work out temporarily. But eventually it will just feel like you and your team are just "going through the motions" of the practices. They'll feel like a burden without much benefit, and within a few weeks or months the team will go back to the way things were before.

But that doesn't mean there's no hope! It just means that you and your team should *talk about the values* ***before*** *you try the practices*. If you start working on the culture issues from the beginning, it makes XP (or any methodology!) a lot easier to adopt, and gives the whole effort a much better chance of sticking.

Speaking of improving the way the team functions, let's check in on Ryan and Ana →

OUR DEVELOPMENT IS REALLY HUMMING ALONG! I CAN'T BELIEVE HOW MUCH CODE WE'VE WRITTEN IN THE LAST FEW WEEKS.

YEAH, THESE NEW PRACTICES MADE A HUGE DIFFERENCE. BUT... WELL, I'M WONDERING IF IT'S TOO MUCH OF A GOOD THING.

Ryan: Ha ha! Good one! ... um ... wait, you're not joking, are you?

Ana: No, I'm being serious here. We're adding a lot more code, but now we're building some pretty complex stuff.

Ryan: Yes!

Ana: That's not necessarily a good thing.

Ryan: Uh... what?

Ana: Like this centralized common automated build script that you created.

Ryan: How's that a problem? We had a bunch of nearly-identical build scripts. That's a lot of duplicated code. I fixed it.

Ana: Yeah, you saved like 12 lines of code duplicated in eight different build scripts...

Ryan: Okay.

Ana: ... by building this seven hundred line monstrosity that's impossible to debug.

Ryan: Um... okay?

Ana: And now any time I need to modify the build, I have to spend hours trying to debug through that enormous script. It's really painful.

Ryan: But it saves ... well ... okay. It saves like 12 duplicate lines in a couple of scripts. I get your point—duplicate code is usually bad, but in this case it would be a lot easier to maintain a few duplicate lines than keep working with the script I wrote.

Ana: And it's not just the builds. We built this super complex unit test framework.

Ryan: I see where you're going with this. I had to debug through it the other day because I had to update the test data for just one unit test. It took me two hours to do a really simple job that should have taken five minutes.

Ana: You know what? I think adding these new XP practices helped speed up our coding. But I'm starting to think that all of this complexity is starting to slow us down.

Ryan: So what do we do about it?

Complex code is really hard to maintain

As systems grow, they often become big and complicated, and **complex code tends to get more complex** as you work with it. And when code gets more complex, it gets harder to work with, which causes developers to take shortcuts that make the problem worse. That's exactly what happened when Ryan tried to make a change that customers needed:

1. Some of CircuitTrak's yoga studio and martial arts dojo customers want to give the trainer the ability to reserve the entire studio for seminars, meetings, and day-long classes. This is a diagram of one small part of the CircuitTrak codebase that Ryan needs to modify in order to make the change.

2. When Ryan started working on this change, he thought it would just require a simple modification to the TrainerSchedule class.

3. Ryan hadn't realized how complex the TrainerSchedule class had become. It was built so that it's tightly coupled to other classes.

4. Ryan discovered that before he could change TrainerSchedule, he'd have to modify the Customer class. But that would require a change to CustomerTrainers. And before he can do that, he'll need to modify ScheduleUpdater and Trainer, and ... oops, that will require a change to an entirely different part of TrainerSchedule. It'll take days to get this done!

5. So Ryan ended up settling for a hack. He copied all of the code in TrainerSchedule to a new class, StudioSchedule, to handle just this specific situation, made some modifications, and deleted everything in it that he didn't need. Then he added "special case" code ScheduleRenderer and MasterSchedule. It works, but it's ugly. And it makes the whole system a little more complex.

A hack (sometimes called a "kludge") is what programmers call a clumsy, quick-and-dirty solution that technically gets the job done but may cause problems down the road.

Ryan's solution is definitely a hack. The way he implemented it, if they ever need to change the way schedules work, they'll need to remember to change the new StudioSchedule class too—and that will probably cause a "shotgun surgery" cascade of changes.

A big, complex system gets more and more complex one little hack at a time.

When teams value simplicity, they build better code

When you're solving a programming problem, there are an almost infinite number of ways that you can code the solution. Some of those ways are a lot more complex than others. They might have a lot more interconnections between units or add extra layers of logic. Units can grow far too big to understand all at once, or can be written in a way that's too convoluted read and comprehend.

On the other hand, everything works better When your code is simple. It's easier to modify your code to add new behavior, or to modify it to change the way it works. When the code is simple, there are fewer bugs, and they're easier to track down when they happen.

So how do you know if a particular unit—like the `TrainerSchedule` Java class that Ryan was working on, for example—is getting too complex? There's no hard-and-fast rule that governs complexity. That's why instead of a rule, XP teams have a value. Specifically, people on XP teams value **simplicity**. Of the many ways to solve any particular coding problem, someone on an XP team will choose the simplest one that he or she can think of.

Simplicity

Code gets complex when it does too many things.
One of the most common ways that your code can get complex is when one unit does too many things. Units of code tend to be organized by their behavior. When one unit does too many things, one of the most effective ways of reducing complexity is to **separate it into smaller units** that each do just one thing.

Refactor existing code to make it less complex.
There's no single "right" way to build a specific unit of code—there are many right answers, and it's rare to write code optimally the first time. That's why XP teams **refactor** their code as often as they need to. When they refactor (or modify the code to change its structure without altering its behavior), the code almost always ends up less complex than before.

XP team members are always on the lookout for units that are starting to get complex. They know it's worth taking the time to refactor as soon as they see anything at all that can be made more simple.

Great habits are more effective than discipline.
If you try to nag your teammates (or yourself) into using practices like test-driven development or tools like refactoring, they typically won't stick. Instead, people on effective XP teams develop **great habits**. For example, they get into the habit of refactoring every time they see code that can be refactored, just like they get into the habit of writing unit tests first. This is part of the XP mindset.

BRAIN POWER

What kind of habits could help the CircuitTrak team avoid problems like the one Ryan ran into with the studio reservation modification?

Simplicity is a fundamental agile principle

Let's take a closer look at one of the twelve principles behind the Agile Manifesto:

> Simplicity—the art of maximizing the amount of work not done—is essential.

Hmm... "maximizing the amount of work not done" sounds like a philosophical musing, or something the caterpillar said in Alice in Wonderland. What does that really mean?

When units are tightly coupled, it adds complexity to the project

When you're renovating a house, the most damaging thing that you can do is to take a sledgehammer and knock down a wall. That's one way that writing code is different from engineering physical objects: if you delete a bunch of code, it doesn't cause permanent damage to the project—you can easily recover it from the version control system.

If you really want to make your code worse, build some new code, modify a bunch of existing units so that they depend on it, and then modify some additional units so they're all tightly coupled to the ones you modified. That's practically a guarantee you'll spend many frustrating hours jumping from one unit to another trying to track down a problem.

It's tempting to sacrifice simplicity for reusability

Developers love **reusable** code. When you're writing code, you often find that you need to solve the same problem in many different parts of the system. It's a really satisfying "a-ha" moment when you're working on a tough problem and you realize that you can call an existing method or use an object that already exists.

But there's a trap a lot of programmers fall into: optimizing code for reusability, while sacrificing simplicity. That's what Ana was talking about on page 46—Ryan created a very complex build script just to save a few lines of duplicated code, but the new script made it really hard to modify or fix problems in the build. Ryan wanted to avoid duplicate code, but ended up making the project harder to change.

An effective way to maximize the amount of work not done is to only write code for a specific, concrete purpose that you know about right now. Avoid writing code just in case you might need it later.

Every team accumulates technical debt

Little problems with your code add up over time. It happens to every team. All developers—even really good, highly skilled developers—write code that can stand to be improved. This is natural: when we're writing code to solve a problem, we often learn more about that problem as we work on it. It's very natural to write code that works, look at the results, *think about it for a while*, and then realize that there are ways that you can ***improve and simplify it***.

But a lot of times developers don't always go back and improve their code—especially when we feel like we're under enormous pressure to get code out the door as soon as possible, even if it isn't as "done" as it could be. And the longer those "unfixed" design and code problems linger in the codebase, the more problems compound, which leads to complex code that's painful to work with. Teams refer to these lingering design and code problems as **technical debt**.

USE SIMPLICITY AND REFACTORING TO PAY DOWN TECHNICAL DEBT

Technical debt happens to all teams, and one reason that is that it's easy to write code that's a little too complex. Here are a few tips for achieving **greater simplicity** and **avoiding technical debt**:

- **Care:** Simplicity is a **value**, which means that you have to genuinely care about it.
- **Plan:** It takes work to simplify your code—in fact, a lot of people feel that it's harder to write simple code than it is to write complex code. That's why XP teams give themselves time to refactor when they plan their weekly cycles.
- **Search:** It's not always easy to see that something is complex, especially when you're used to looking at it. Sometimes you need to work to find candidates for simplification.
- **Act:** Found some complex code that could be simpler? Now it's time to refactor it!

Watch it!

Don't be afraid to delete your code.

*One of the most common traps that developers fall into is an **unwillingness to delete code** once they've written it. This can lead to **code bloat**: extra behavior, dead code, or other redundancies or inefficiencies that make code worse.*

You should feel comfortable deleting code, because you can always go back and recover it from the version control repository.

XP teams "pay down" technical debt in each weekly cycle

Are you surprised to learn that programmers often don't get code right the first time they write it? It's true! Developers don't just "spit out" code and the move on to the next problem. Just like great artists, craftspeople, and artisans create sketches and preliminary designs and then refine them into finished products, great programmers create initial versions of their code and then refactor that code, often many times.

That's why really effective XP teams make sure that they **add time to every weekly cycle** to "pay down" their technical debt and fix those lingering problems before they start to pile up. And the most effective way to do that is to refactor your code. XP teams have a name for this great habit: **refactor mercilessly**.

I KNOW WE'VE GOT A DEADLINE, BUT IT'S ***WORTH TAKING THE TIME*** TO FIX THIS CODE ***NOW***, BECAUSE THE NEXT CHUNK OF WORK WILL GO A LOT FASTER... AND IT'LL BE *MUCH LESS FRUSTRATING!*

Under the Hood: Refactoring

Let's look at a specific way that developers refactor their code

Refactoring means modifying the structure of your code without changing its behavior, and like most things that agile teams do, it's easy to get started (but it takes time and practice to master the subtleties). Here's an example of a common refactoring called **extract method** that Ana used to simplify her code.

```
for ( StudioSchedule schedule : getStudioSchedules() ) {
    CustomerTrainers trainers = getTrainersForStudioSchedule( schedule );
    if ( trainers.primaryTrainerAvailable() ) {
        ScheduleUpdater scheduleUpdater = new ScheduleUpdater();
        scheduleUpdater.updateSchedule( schedule );
        scheduleUpdater.setTrainer( trainers.getPrimaryTrainer() );
        scheduleUpdater.commitChanges();
    } else if ( trainers.backupTrainerAvailable() ) {
        ScheduleUpdater scheduleUpdater = new ScheduleUpdater();
        scheduleUpdater.updateSchedule( schedule );
        scheduleUpdater.setTrainer( trainers.getBackupTrainer() );
        scheduleUpdater.commitChanges();
    }
}
```

These four lines of code update a class schedule so that the primary trainer is teaching it.

These four lines of code are almost identical. They do the same thing, except for the backup trainer.

Ana refactored the code by moving those four duplicate lines into a new method called `createScheduleUpdaterAndSetTrainer()`

```
for ( StudioSchedule schedule : getStudioSchedules() ) {
    CustomerTrainers trainers = getTrainersForStudioSchedule( schedule );
    if ( trainers.primaryTrainerAvailable() ) {
        createScheduleUpdaterAndSetTrainer( trainers.getPrimaryTrainer() );
    } else if ( trainers.backupTrainerAvailable() ) {
        createScheduleUpdaterAndSetTrainer( trainers.getBackupTrainer() );
    }
}
```

Ana eliminated the duplicate lines of code, which made this code simpler. And now if she needs to do the same thing for other trainers, she can reuse the new method she created.

Incremental design starts (and ends) with simple code

The practices we've talked about are really good for helping everyone on the team to develop habits to help them create small, decoupled units that work independently of each other. As they start to build up those habits, they can start practicing **incremental design**. This XP practice is exactly what it sounds like: the team creates the design for their project in small increments, building only the next bit of design needed for the current quarterly cycle, and concentrating mainly on what's needed in this weekly cycle. They build small, decoupled units, refactoring as they go to remove dependencies, separate units that get too big, and simplify the design of each unit.

When teams do incremental design they discover, uncover, and evolve the design bit by bit—just like when they do incremental development, they discover, uncover, and evolve the plan bit by bit.

When an XP team uses incremental design, the first set of units that they build typically evolve into a small, stable core. As the system grows, they add or modify a small number of units in each weekly cycle. They'll use test-driven development to make sure that each unit has minimal dependencies on the other units, which in turn makes the whole system easier to work with. In each iteration, the team adds only the design needed to build the next set of stories. When units interact in a simple way, it lets the whole system grow organically, bit by bit.

All designs change. Incremental designs are built to change.

Generations of software engineers were taught in school that the design of a system needs to be complete before the team can start coding. This idea is built into the waterfall process: the project needs to complete the design phase before moving on to the development phase. Incremental design works when teams **make design decisions at the last responsible moment**, exactly the same way that teams using iterative development make planning decisions.

Incremental design really does work in the real world. One of the most successful examples is the Unix toolset (the set of Unix shell commands—`cat`, `ls`, `tar`, `gzip`, etc.). These tools weren't all developed at once. Instead, the Unix tools were based on a *philosophy of simplicity*: each tool does one specific, straightforward job, producing output that every other tool can use as input. This allowed thousands of different people over to contribute to the entire toolset over many years. It grew incrementally, with individual tools being added one by one as the need for them arose.

XP teams take a very similar approach, starting with the same idea of embracing the value of simplicity. And just like with the Unix toolset, it's an effective way to work.

SO BECAUSE THE TEAM VALUES SIMPLICITY, IT MAKES SENSE TO THEM TO BUILD ONLY THE UNITS THAT ARE NEEDED FOR THE NEXT SET OF STORIES, AND BECAUSE THE DESIGN IS ALREADY SIMPLE IT'S **EASY TO MODIFY.**

That's right. And software that's designed to be modified makes easy for the team to embrace change.

The whole point of XP is to improve the way that the team writes code, and just as importantly, to improve and energize the working environment. When everyone on the team really "gets" incremental design, the whole system becomes much easier to work with. That makes the work **much more satisfying**: the most tedious parts of the software development job are reduced and often eliminated.

All of this leads to a very positive feedback loop: the weekly cycle and slack give the team enough time to do the work and constantly refactor the code, which lets them incrementally create a simple design, which helps everyone stay energized and approach problems with a fresh and clear mind, which lets them make progress quickly, which gives them success in the company. That success ***lets the team work with the business more effectively***, and that gives them the ability to keep planning the project using weekly cycles and slack.

That feedback loop is what drives an XP team's ability to embrace change.

BULLET POINTS

- XP teams **embrace change** instead of resisting it.
- A **10-minute build** gives the team constant feedback about the build, and reduces frustration from waiting.
- The team uses **continuous integration** by making sure everyone's working folder is no more than a few hours out of date.
- **Test-driven development**, or building unit tests first and then building the code that makes the tests pass, helps teams keep units of code simple and reduce dependencies.
- People who do **pair programming**, with a pair of developers sitting at a single computer, produce better code more quickly than when they work separately.
- If it's not clear whether technical approach will work, **spike solution**, or a small, throwaway program to test a it out, will help the team determine if it's a good approach.
- The XP practices **reinforce each other** to create an ecosystem effect.
- XP teams develop **good habits**, which leads to great software without forcing discipline on the team.
- When a methodology's practices **don't feel right**, it usually points to a clash between the values of the methodology and the mindset or culture of the team.
- Agile teams value **simplicity** because it leads to better code, and helps them to build less code.
- XP teams **don't sacrifice simplicity** for reusability.
- **Incremental design**, or building only the design that's needed for the current iteration, is an effective practice for helping to keep your system from getting complex.

there are no Dumb Questions

Q: Does XP *really* make the job more satisfying?

A: Yes, really! Keeping the workplace energized means everyone watches for signs of exhaustion, boredom, and agitation. Those feelings are often indicators that team members are dealing with avoidable code problems, or are being forced to work late because of irresponsible planning.

Q: How do you know that Ryan's studio schedule change was a hack?

A: There were a few glaring warning signs. The first one was that he copied an entire class, left a bunch of it intact, and just deleted the bits he didn't need. That led to a lot of duplicate code. And then he added "special case" code to other parts of the system. That's code that looks for a particular state—in this case, scheduling a whole studio instead of a single yoga or martial arts class—and performs specific behavior just for that case. Developers try to avoid those things because they make the system more difficult to maintain. There's almost always a more elegant way to solve a problem like that.

Q: Okay, now I'm confused about duplicate code. Ryan shouldn't have made a complex build script just to avoid a few duplicated lines, but he also shouldn't have made that hack with a class that had a lot of duplicate code. So is duplicate code good, or is it bad?

A: There's very little that's more aesthetically unpleasant to a programmer than a block of code that's duplicated in two (or worse, more!) places. It's almost always better to reuse the duplicate code by moving it into its own unit (like a class, function, module, etc.). But sometimes the situation isn't so straightforward. A few lines of duplicate code aren't necessarily particularly easy to reuse. Occasionally it takes a lot of work to extract them into their own unit. When we're coding, we sometimes go to such great lengths to avoid a small block of duplicate code that we end up adding complexity instead of removing it. That's the trap Ryan fell into with his build script.

Q: Hold on... "aesthetically unpleasant?" Since when do aesthetics have anything to do with code?

A: Code aesthetics matter a lot! If you're not a developer, it might seem weird to talk about code being "aesthetically pleasing" or not. But one sign of a great developer is a sense of aesthetics and even beauty in the code that he or she writes. Duplicate code is particularly aesthetically offensive to developers, because it's almost always a sign that something can be simplified.

Q: So how do I know if my code is too complex, or not simple enough? Is there a rule that I can apply?

A: No, there's no rule about how much complexity is too much. That's why ***simplicity is a value, not a rule***. The more experience you have as a programmer on a team that values simplicity, the better you get at making your code simple. That said, there are definitely warning signs that your code might be too complex. For example, you know a block of code is probably too complex if you're afraid to touch it, or if there's a scary comment that says **`Don't edit this!`** at the top. Build scripts and unit tests are too complex if you find yourself avoiding a change because the change itself is easy, but modifying the build script or unit test will be really difficult or annoying.

Q: I still don't get the point about test-driven development and simplicity. Does writing unit tests first really help keep code simple?

A: Yes. A lot of complexity happens when you build units of code that have many dependencies on other parts of the system. If you can avoid adding those dependencies, it makes your whole system a lot easier to maintain, and helps you avoid that "shotgun surgery" feeling when you're working on the code. Unit tests are really good at helping you avoid unnecessary dependencies, because a test for a single unit has to provide all of the input that the unit needs. If that unit has a lot of dependencies, it makes the test extremely annoying to write—and it becomes very obvious exactly which dependencies you really need. Often, that will also show you another part of the system that could be refactored. And it gives you incentive to do that refactoring immediately, because it will make the job at hand less annoying, tedious, or frustrating.

Q: And reducing annoyance and tedium... that's good for the team, right?

A: Yes! One of the best ways to make your team more productive is to make the work everyone is doing **less annoying, tedious, boring, or frustrating**. That's a really effective way to build an energized workplace. It's why people on XP teams really can't imagine working any other way.

Exhaustion, boredom, and agitation can be early indicators of code problems that can be avoided

Here are some things we overheard Ana, Ryan, and Gary saying. Some of them are compatible with XP values, others are incompatbile. Identify the XP value that each of them either compatible or incompatible with. Then draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and another line to the appropriate Scrum value.

THIS JAVA CLASS IS TOO BIG AND DOES TOO MANY THINGS. I'LL REFACTOR IT INTO TWO SEPARATE CLASSES.

COMPATIBLE

INCOMPATIBLE

Respect

I ALWAYS RUN THE BUILD BEFORE I COMMIT MY CODE TO MAKE SURE EVERYTHING COMPILES AND ALL OF THE UNIT TESTS PASS.

COMPATIBLE

INCOMPATIBLE

Communication

YOU'RE ASSIGNING THAT TO THE NEW GUY? HE'S PRETTY YOUNG, MAYBE WE SHOULD GIVE HIM SOME GRUNT WORK UNTIL HE GETS A LITTLE MORE EXPERIENCE.

COMPATIBLE

INCOMPATIBLE

Simplicity

THERE'S A BUG IN MY CODE? ENTER A TICKET FOR IT, I'LL GET TO IT WHEN I HAVE TIME.

COMPATIBLE

INCOMPATIBLE

Feedback

Answers on page 67

FOUR MONTHS LATER...

XPcross

Answers on page 68

Across

1. Scrum has a strong focus on _____ management
3. The XP practices work together and reinforce each other to form this
5. XP teams create automated _____ that run in 10 minutes or less
7. A clumsy, quick-and-dirty solution
8. Everyone on the team continually _____ the code in their working folders back into the version control system
10. The kind of loop that teams use to repeatedly get useful information and make adjustments
12. What a version control system provides for the team to store their code
14. What XP teams do together to help them communicate well
17. When people have this value, they don't mind a little chatter in their office environment
18. What XP teams do with change
19. What teams add when they include optional or minor items
20. Another name for a clumsy, quick-and-dirty solution (rhymes with "stooge")
22. The kind of programming where two people sit at one computer
24. A programmer takes 15 to 45 minutes to reach this state of high concentration
26. What you do when you run across complex code that can be simplified
29. They add complexity to your code
30. Practice used in XP and Scrum to manage requirements
31. How often XP iterations happen
32. This value maximizes the amount of work not done
33. The _____ cycle is how XP teams do mid- to long-term planning

Down

1. Nagging people to achieve this is not only annoying and ineffective, but actually counterproductive
2. Agile teams welcome _____ requirements, even late in development
4. It's a lot less stressful to work with code that's easy to _____
6. All code is broken down into these
7. They're better than discipline for making practices "stick"
9. The pace that XP teams strive for, and the kind of development that agile processes promote
11. If you agree to a deadline you know you won't meet because it's easier to apologize later, you lack this value
13. A burndown chart or task board posted where you can't help but absorb its data is an information _____
14. The kind of solution where you run an experiment by creating a small, throwaway program
15. The kind of design where teams make design decisions at the last responsible moment
16. When XP teams replace their planning practice with a complete and unmodified implementation of Scrum
21. When you try to commit a code change, but find that your teammate already committed a change to the same lines of code
23. XP and Scrum value that helps team members trust each other
24. TDD means writing unit tests _____
25. The kind of communication that happens when you absorb information from conversations all around you
27. A set of changes that you're pushing to a version control system
28. XP teams don't have fixed or prescribed _____

Exam Questions

These practice exam questions will help you review the material in this chapter. You should still try answering them even if you're not using this book to prepare for the PMI-ACP certification. It's a great way to figure out what you do and don't know, which helps get the material into your brain more quickly.

1. Which of the following is NOT true about how XP teams plan their work?

A. XP teams often self-organize by having team members pull their next tasks from a pile of index cards
B. XP teams use week-long iterations
C. XP teams focus on code, so they do very little planning
D. XP is iterative and incremental

2. How do XP's values and practices help teams embrace change?

A. By helping them build code that's easier to modify
B. By placing strict limits on how users request changes
C. By enforcing a change control process
D. By limiting the amount of contact between the business users and the team

3. Amy is a developer on a team that builds mobile apps for commuters. They've adopted XP, but instead of using weekly cycles, quarterly cycles, and slack, they hold a Daily Scrum, do sprint planning, and hold retrospectives. Which of the following BEST describes Amy's team?

A. They do not do adequate planning
B. They are in the process of adopting XP
C. They use a hybrid of Scrum and XP
D. They are transitioning from XP to Scrum

4. Which of the following are NOT common to both XP and Scrum?

A. Roles
B. Iterations
C. Respect
D. Courage

Exam Questions

5. Which of the following is a valid way for XP teams to do estimation?

A. Planning poker
B. The planning game
C. Traditional project estimation techniques
D. All of the above

6. Evan is a project manager on an XP team. He noticed that over the last few weekly cycles, everyone had their headphones on and listened to music all day while coding. Evan is concerned that the lack of osmotic communication is making the workspace less informative. He called a team meeting to explain XP's informative workspace practice, and suggested that they adopt a rule against wearing headphones at work.

Which BEST describes this situation?

A. The team is not performing the informative workspace practice
B. Evan has a responsibility to help the team adopt XP, and is demonstrating servant-leadership
C. Evan needs to improve his understanding of the XP values
D. The team is using a hybrid of Scrum and XP

7. Which of the following is true about test-driven development?

A. Unit tests are written immediately after writing the code that they test
B. Writing unit tests first can have a profound impact on the design of the code
C. Test-driven development is used exclusively by XP teams
D. Writing unit tests causes the whole project to take longer because the team spends more time writing code, but it's worth it for the extra quality.

8. What is involved in continuous integration?

A. Setting up a build server that constantly integrates new code into a working folder and alerts the team on build or test failures
B. Using iteration to continuously produce working software
C. Each person on the team keeps their working folders up to date with the latest code from the version control system
D. Continuously reducing technical debt by improving the structure of the code without modifying its behavior, and integrating those changes back in.

Exam Questions

9. Which of the following is NOT an example of an information radiator?

A. The team sitting together so they can absorb information from conversations that happen around them
B. Posting a burndown chart in a place where everyone can see it
C. Keeping the team's task board on a wall in a common area
D. Maintaining a list of stories the team has finished so far in the weekly cycle on a whiteboard that everyone can see

10. The following practices all establish feedback loops for XP teams except:

A. Test-driven development
B. Continuous integration
C. 10-minute build
D. Stories

11. Why do teams use wireframes that are low fidelity?

A. Users give more feedback when a user interface mock-up looks less polished
B. Agile teams rarely build software that contains detailed audio
C. The team only builds and reviews one set of wireframes per weekly cycle
D. They're only used for less complex user interfaces, and XP teams value simplicity

12. Which of the following promotes sustainable development?

A. Thoroughly planning the next six months of work so that there are no surprises for the team
B. Making sure everyone gets everything right the first time they build it, so no rework is required
C. Making sure everyone leaves on time and nobody feels pressured to work weekends, so the team doesn't burn out
D. Setting tight deadlines, so everyone is motivated to meet them

13. Which of the following is NOT a benefit of pair programming?

A. Everyone on the team gets experience working with every part of the system
B. There are two sets of eyes on every change
C. Pairs help each other stay focused
D. People take turns working, so fatigue is reduced

Exam Questions

14. Joanne is a developer on a team that constantly refactors, does continuous integration, writes unit tests first, and does many other XP practice. What BEST explains this team's culture?

A. They have a strict manager who enforces the rules of XP
B. They have good habits
C. They are highly disciplined
D. They are worried about getting fired if hey don't work this way

15. What happens when a build takes longer than 10 minutes to run?

A. It causes errors in the packaging process
B. Team members run the build infrequently
C. Merge conflicts occur that are difficult to resolve
D. The unit tests fail

16. Joy is a developer working on a team building a mobile operating system. She tried to commit code for a feature that she's been working on, but the version control system won't let her complete the commit until she resolves many conflicts. Which practice will BEST prevent this problem in the future?

A. Sustainable pace
B. Continuous integration
C. 10-minute build
D. Test-driven development

17. Kiah is a developer on an XP project. Her team is doing quarterly planning. One of the features is extremely important, and failing to deliver it will have serious consequences for the project. Kiah is the expert on this part of the project, and she'll be the one doing the programming work. She feels that the design relatively straightforward, and she's pretty sure that she knows exactly how to build it.

What is the BEST action for Kiah and her team to take?

A. Add a story for an architectural spike to an early weekly cycle
B. Build a low-fidelity wireframe to get early feedback
C. Add a story for a risk-based spike to an early weekly cycle
D. Do extra usability testing

Exam ~~Questions~~ Answers

Here are the answers to the practice exam questions in this chapter. How many did you get right? If you got one wrong, that's okay—it's worth taking the time to flip back and re-read the relevant part of the chapter so that you understand what's going on.

1. Answer: C

XP teams might not focus on project management to the extent that Scrum teams do, but XP is still an iterative and incremental methodology that values self-organizing teams. Those reasons are part of why it's an agile methodology.

This also helps keep rework from causing quite so many bugs.

2. Answer: A

It's a lot easier for teams to embrace change when they know those changes won't be a headache for them to make. XP helps with this by including practices and values that help teams build code that's easier to modify.

3. Answer: A

Teams that use a Scrum/XP hybrid have replaced the planning-related XP practices with a complete implementation of Scrum. Amy's team hasn't done that. They adopted some of the Scrum practices, but since they dropped the quarterly cycle but didn't add any sort of product backlog, they've pretty much stopped doing any sort of long-term planning.

They also don't have a Scrum Master or Product Owner. What other Scrum practices have they ignored? What do you think this all of this indicates about the mindset among Amy's team members?

4. Answer: A

XP and Scrum both value respect and courage, and both use timboxed iterations for planning. But XP has no fixed roles, while Scrum teams must always have team members who fill the Product Owner and Scrum Master roles.

The planning game is a practice that was part of an early version of XP. It guided the team through creating an iteration plan by helping them decompose stories into tasks and assign them to team members. It's still in use by a few teams, but planning poker is a lot more popular.

5. Answer: D

XP teams use many different technique for estimating, and there is no specific rule that says the team must use any specific technique. So all of the techniques listed are valid. So is having the team simply meet and talk about how long they think the work will take.

Exam ~~Questions~~ Answers

It may seem weird to talk about feelings at work, but they're actually really important for getting a team to run smoothly. It's really difficult to innovate and do the difficult intellectual and creative work when you're distracted by negative feelings like resentment.

6. Answer: C

Evan has decided that the team is doing something wrong because they are not following his personal interpretation of the XP practices. When he called the team meeting and proposed a rule against wearing headphones, he was ignoring the fact that this is how the team prefers to work. This is very disrespectful, and shows that he doesn't trust them to find an effective way to work. Respect is a core XP value, and when people ignore it, that hurts the whole team by stirring up resentment and other negative feelings.

7. Answer: B

When you write unit tests first, it can have a profound impact on the design of the code. The reason is that when you're writing the tests, awkward constructions and unnecessary coupling between units can become much more apparent. Test-driven development is not exclusive to XP teams—many teams do it, even on waterfall projects. And even though it requires developers to write more code overall, most people who do test-driven development find that it actually saves time overall because it makes fixing bugs and making changes much, much faster.

The total time teams spend writing the extra code for the unit tests is more than made up for by the time saved making changes. This isn't a long-term effect—it's easily noticeable within days or even hours.

8. Answer: C

Continuous integration is a straightforward practice that can have an outsized effect on the project. The team continuously integrates the latest code from the version control system into their working folders every few hours. This prevents them from having to deal with time-consuming and annoying merge conflicts that span many files at the same time.

9. Answer: A

An information radiator is any sort of visual tool or display that conveys useful information about the project and is highly visible so that team members can't help but absorb the information on it as they walk by. The first answer describes osmotic communication.

Osmotic information and information radiators are both tools that can help with the informative workspace practice.

10. Answer: D

Stories are really useful, but they don't really establish a feedback loop the way some practices like test-driven development, continuous integration, or 10-minute builds do. The reason is that most of the time the story doesn't change very much once it's written, so there's no opportunity for feeding information back into it repeatedly. The other three practices establish feedback loops that occur many times over the course of a weekly cycle.

Exam ~~Questions~~ Answers

11. Answer: A

Wireframes are often low fidelity, which means they look like rough sketches or hand-drawn mock-ups. Users are often a lot more willing to give feedback about a sketch that looks like it was easy to draw than they are for a highly polished, accurate mock-up, because it feels intimidating to ask for changes to a design that looks like it required a lot of work. A low fidelity wireframe can still capture all of the detail of a rich user interface, and is no more complex or simple than a mock-up that's highly polished.

Low fidelity wireframes are usually a lot less work than ones that are a lot more polished, which lets teams review several different versions with the users. They can help the team try out several iterations of the same UI in a single weekly cycle.

12. Answer: C

Sustainable development happens when the team works at a pace that they can comfortable manage, which almost always means working normal 40-hour weeks.

A lot of teams have one or two people who make a point of staying late to show how "committed" they are (or to impress the boss). This often puts a lot of pressure on everyone else to stay late, too, which can easily create an unsustainable pace and burn the team out.

13. Answer: D

Pair programming is a very effective and efficient practice because two people working at the same computer keep each other focused, constantly collaborate, catch many problems, and get more done than if they were working alone. But both people are always working together at the same time—they don't take turns working.

14. Answer: B

XP teams use great practices every day because they have great habits. They don't do it out of a sense of discipline, and they certainly don't do it out of fear. Raw discipline and fear can cause temporary, short-term changes to the way teams work, but eventually teams revert back to their habits.

The way to build great habits is to try out the practices, see great results, and use those to slowly change the way you think about your work. That's why adopting the XP practices helps get the team into the XP mindset.

KENT BECK, THE GUY WHO CREATED XP, ONCE SAID, "I'M NOT A GREAT PROGRAMMER. I'M JUST A GOOD PROGRAMMER WITH **GREAT HABITS**."

Exam ~~Questions~~ Answers

15. Answer: B

When an automated build takes a really long time to run, the team runs it a lot less frequently. That means the team gets less frequent feedback about the state of the build.

16. Answer: B

Continuous integration is a simple practice in which the team members keep their working folders up to date with the latest changes in the version control system. That prevents many merge conflicts, which can needlessly waste the team's time and cause a lot of frustration.

17. Answer: C

A risk-based spike is a spike solution that the team undertakes specifically to reduce project risk. In this case, Kiah already knows the technical approach that she will take, so there's no need for an architectural spike. But since the risk for this particular feature is very high, it makes sense to add a risk-based spike to a weekly cycle early in the project. That way the risk will be eliminated early on.

And if it turns out there are unforeseen problems, it's a lot better to discover them early in the project than later.

A bunch of XP practices and values are playing a party game, "Who am I?" They'll give you a clue, and you try to guess who they are based on what they say. Write down their names, and what kind of things they are (like whether they're events, roles, etc.).

And watch out—a couple of tools that are not XP practices or values might just show up and crash the party!

	Name	Kind of thing
I help XP teams get into a mindset where they know that they're best at solving problems when they share knowledge with each other.	communication	value
I'm a great way to absorb information about the project from discussions happening around me.	osmotic communication	tool
I help you understand your user's needs, and I'm also used by a lot of Scrum teams.	stories	practice
I'm a team space that does a great job communicating information about the project.	informative workspace	practice
I help the team works at a sustainable pace, because teams that work super-long hours actually build less code with worse quality.	energized work	practice
I'm how XP teams do their long-term planning, by meeting with the users once a quarter to work on the backlog.	quarterly cycle	practice
I help people get into a mindset where they treat each other well, and value each others' input and contributions.	respect	value
I'm the reason people on XP teams will tell the truth about the project, even if it's uncomfortable.	courage	value
I'm the way that XP teams do iterative development, and teams use me to deliver the next increment of "Done" working software.	weekly cycle	practice
I'm a large burndown chart or task board put up in the team space in a spot where everyone can't help but notice me.	information radiator	tool
I make sure that the team has a space where everyone is near their teammates.	sit together	practice
I help give the team some breathing room in each iteration by adding optional stories or tasks.	slack	practice

Here are some things we overheard Ana, Ryan, and Gary saying. Some of them are compatible with XP values, others are incompatbile. Identify the XP value that each of them either compatible or incompatible with. Then draw a line from each speech bubble to either COMPATIBLE or INCOMPATIBLE, and another line to the appropriate Scrum value.

THIS JAVA CLASS IS TOO BIG AND DOES TOO MANY THINGS. I'LL REFACTOR IT INTO TWO SEPARATE CLASSES.

COMPATIBLE

INCOMPATIBLE

When a unit does two different things, splitting into two separate, independent units makes the code a little less complex.

I ALWAYS RUN THE BUILD BEFORE I COMMIT MY CODE TO MAKE SURE EVERYTHING COMPILES AND ALL OF THE UNIT TESTS PASS.

COMPATIBLE

INCOMPATIBLE

Running the tests before you commit gives you immediate feedback so you can find out if you added a bug that broke another part of the code.

YOU'RE ASSIGNING THAT TO THE NEW GUY? HE'S PRETTY YOUNG, MAYBE WE SHOULD GIVE HIM SOME GRUNT WORK UNTIL HE GETS A LITTLE MORE EXPERIENCE.

COMPATIBLE

INCOMPATIBLE

It's pretty disrespectful to think of someone as a "second-class citizen" on your team and only give them work that's somehow not "good enough" for others.

THERE'S A BUG IN MY CODE? ENTER A TICKET FOR IT, I'LL GET TO IT WHEN I HAVE TIME.

COMPATIBLE

INCOMPATIBLE

When someone wants to talk about a problem in your code, it's always worth taking the time to listen.

Respect

Communication

Simplicity

Feedback

XPcross SOLUTION

1 P	R	O	J	E	2 C	T				3 E	C	O	S	Y	S	T	E	4 M								
U					H													O								
R					A									5 B	6 U	I	L	D	S			7 H	A	C	K	
I				8 I	N	T	E	G	R	A	T	E	9 S		N			I				A				
T					G								U		I			10 F	E	E	D	B	A	11 C	K	
Y					I				12 R	E	P	O	S	I	T	O	13 R	Y				I		O		
	14 S	I	T		N		15 I						T		S		A					T		U		
	P				G		N						A				D					S		R		
	I		16 H				17 C	O	M	M	U	N	I	C	A	T	I	O	N					A		
	K		Y				R						N				A							G		
	18 E	M	B	R	A	C	E				19 S	L	A	C	K		T		20 K	L	U	D	G	E		
			R				M			21 C			B				O									
	22 P	A	I	23 R			E			O		24 F	L	25 O	W		26 R	E	F	A	27 C	T	O	28 R		
			29 D	E	P	E	N	D	E	N	C	I	E	S							O			O		
				S			T			F		R		M							M			L		
				P			A			L		30 S	T	O	R	I	E	S			M			E		
			31 W	E	E	K	L	Y		I		T		T							I			S		
				C						C			32 S	I	M	P	L	I	C	I	T	Y				
33 Q	U	A	R	T	E	R	L	Y		T				C												

Sharpen your pencil

Here are three scenarios that Ryan and Ana are working on that have to do with getting feedback from their project. Write down the name of the tool being used in each scenario.

WE NEED A NEW WAY TO STORE TRAINER SCHEDULES THAT WILL REDUCE MEMORY. I'M BUILDING A PROOF-OF-CONCEPT SO WE CAN GET A SENSE OF HOW MUCH WORK IT WILL BE.

architectural spike

I'M NOT HAPPY WITH HOW THIS CLASS GETS INITIALIZED – IT'S GOING TO BE HARD TO USE. ONCE ITS UNIT TESTS PASS, I'LL MODIFY IT.

red/green/refactor

I FINISHED DESIGNING THE NEW USER INTERFACE. LET'S MAKE SURE THAT IT WORKS BY GETTING A BUNCH OF USERS IN THE ROOM AND OBSERVING THEM WHILE THEY USE IT.

usability testing