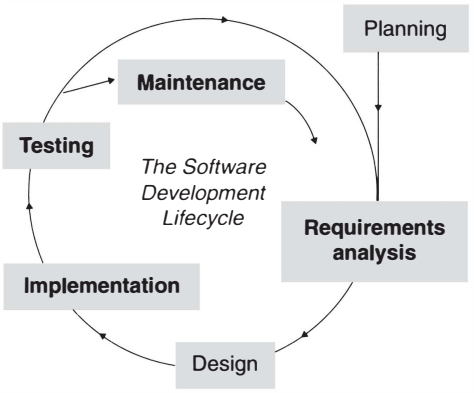
4

Agile Software Processes



- How did agile methods come about?
- What are the principles of agility?
- How are agile processes carried out?
- Can agile processes be combined with non-agile ones?

Figure 4.1 The context and learning goals for this chapter

In the 1990s, agile software development came into being as an alternative to the existing classical approaches to software engineering that were perceived to be too "process-heavy." These classical approaches emphasize the need to plan projects in advance, express requirements in writing, provide written designs satisfying the requirements, write code based on these designs satisfying the written requirements, and finally to test the results. As we discussed in Chapters 1 and 3, however, many projects following these steps exhibit major problems. A primary reason is that stakeholders do not usually know at the inception of a project entirely what they require. Agile processes address this shortcoming. This chapter defines what is meant by agile development, describes several specific software processes that adhere to agile principles and are thus considered agile processes, and discusses how agile and non-agile processes can be combined.

4.1 AGILE HISTORY AND THE AGILE MANIFESTO

A group of industry experts met in 2001 to discuss ways of improving on the then current software development processes that they complained were documentation driven and process heavy. Their goal was to produce a set of values and principles to help speed up development and effectively respond to change. Calling themselves the Agile Alliance, the group's goal was, in essence, to produce a development framework that was efficient and adaptable. During the 1990s, various iterative software methodologies were beginning to gain popularity. Some were used as the basis for the agile framework. These methodologies had different combinations of old and new ideas, but all shared the following characteristics [1].

- Close collaboration between programmers and business experts
- Face-to-face communication (as opposed to documentation)
- Frequent delivery of working software
- Self-organizing teams
- Methods to craft the code and the team so that the inevitable requirements churn was not a crisis

As a result of their meeting, the Agile Alliance produced the Agile Manifesto [1] to capture their thoughts and ideas, and it is summarized in Figure 4.2.

Note that the Agile Manifesto is not anti-methodology. Instead, its authors intended to restore balance. For example, they embrace design modeling as a means to better understand how the software will be built, but not producing diagrams that are filed away and seldom used. They embrace documentation, but not hundreds of pages that cannot practically be maintained and updated to reflect change [2].

The four points of the Agile Manifesto form the basis of agile development. The first part of each statement specifies a preference. The second part specifies something that, although important, is of lower priority. Each of the four points is described next.

Individuals and Interactions (over processes and tools)

For decades, management practice has emphasized the high value of communications. Agile practices emphasize the significance of highly skilled individuals and the enhanced expertise that emerges from interactions among them. Although processes and tools are important, skilled people should be allowed to

---

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **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.

Figure 4.2 The Agile Manifesto

adapt the process and modify the tools as appropriate to get their job done as efficiently as possible. As suggested in [3], agile methods offer generative values rather than prescriptive rules: a minimum set of values, observed in all situations, that generate appropriate practices for special situations. Individuals and teams use these rules when problems arise as a basis for generating solutions that are appropriate for the project. Creativity is emphasized as a major means for problem solving. This is in contrast to more rigid software processes, which prescribe a set of predetermined rules and force teams to adapt themselves to these rules. Agile practices suggest that the latter approach is not effective and actually adds to the risk of project failure.

Working Software (over comprehensive documentation)

Working software is considered the best indicator of project progress and whether goals are being met. Teams can produce pages of documentation and supposedly be on schedule, but these are really promises of what they expect to produce. Agile practices emphasize producing working code as early as possible. As a project progresses, software functionality is added in small increments such that the software base continues to function as an operational system. In this way team members and stakeholders always know how the real system is functioning.

Although significant, working software is of greatly diminished value without reasonable documentation. Agile practices emphasize that project teams determine for themselves the level of documentation that is absolutely essential.

Customer Collaboration (over contract negotiation)

This statement emphasizes the fact that development teams are in business to provide value to customers. Keeping as close as possible to your customer is a long-established maxim of good business practice. Many programmers are disconnected from the customer by organizational layers and intermediaries; it is highly desirable to remove this barrier. All stakeholders, including customers, should work together and be on the same team. Their different experiences and expertise should be merged with goodwill that allows the team to change direction quickly as needed to keep projects on track and produce what is needed. Contracts and project charters with customers are necessary, but in order to adapt to inevitable change, collaboration is also necessary [3].

Responding to Change (over following a plan)

Producing a project plan forces team members to think through a project and develop contingencies. However, change is inevitable, and agile practitioners believe that change should not only be planned for but also embraced. Very good project managers plan to respond to change, and this is a requirement for teams operating at the most effective levels, as you will see when we discuss the highest capability levels of the CMMI in Section III of this book. As changes to the plan occur, the team should not stay focused on the outdated plan but deal instead with the changes by adapting the plan as necessary [3]. Agile practices rely on short iterations of one to six weeks to provide timely project feedback and information necessary to assess project progress and respond as necessary.

4.2 AGILE PRINCIPLES

In addition to the values described in Figure 4.2, the authors of the Agile Manifesto outlined a set of guiding principles that support the manifesto. These are quoted from [1] (bold added).

- "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."

Many of these principles are implemented in practice by the agile methods described in the next section.

4.3 AGILE METHODS

This section describes some of the methods by which many agile processes practice the principles in the Agile Manifesto.

Figure 4.3 shows the manner in which agile methods implement the Agile Manifesto, as follows. Agile processes commonly employ small, close-knit teams; periodic customer requirements meetings; a code-centric approach; documentation on an as-needed basis (e.g., high-level requirements statements only); customer representatives working within the team; refactoring; pair programming; continual unit-testing; and acceptance tests as a means of setting customer expectations.

We next elaborate on the topics not already explained above.

Pair programming is a form of continual inspection by one team member of the work of a teammate. Typically, while one programs, the other inspects and devises tests. These roles are reversed for periods of time that the pair determines.

Documenting on an as-needed basis usually involves writing some high-level requirements but not detailed requirements. These are frequently collected in the form of *user stories*. A user story is a significant task that the user wants to accomplish with the application. According to Cohn [4], every user story consists of the following:

- A written description
- Conversations with the customer that establish a mutual understanding of its purpose and content
- Tests intended to validate that the user story has been implemented

MANIFESTO → / RESPONSES:	1. Individuals and interactions over processes and tools	2. Working software over comprehensive documentation	3. Customer collaboration over contract negotiation	4. Responding to change over following a plan
a. Small, close-knit **team** of **peers**	y			y
b. Periodic **customer** requirements **meetings**	y		y	y
c. **Code-centric**		y		y
d. **High-level** requirements statements only			y	y
e. **Document as** needed			y	y
f. **Customer reps** work within team	y			y
g. **Refactor**				y
h. **Pair** programming and no-owner code	y			
i. Unit-test-intensive; Acceptance-**test-driven**		y	y	
j. **Automate** testing		y	y	

Figure 4.3 Ways to address the principles of the Agile Manifesto

Examples of user stories for a video store application are as follows:

- "The user can search for all DVDs by a given director."
- "The user can establish an account that remembers all transactions with the customer."
- "The user can view all available information on any DVD."

Continual interaction and contact with the customer is achieved in two ways. First, the work periods (1–6 weeks, usually) in which the each batch of requirements are to be fulfilled are specified with a team that involves the customer. Second, a customer representative is encouraged to be part of the team.

The emphasis on *working software* is realized by means of coding versions of the application and showing them to the customer. These are usually closely tied to corresponding tests. Indeed, *test-driven development*, an agile approach, actually has developers write tests even before developing the code.

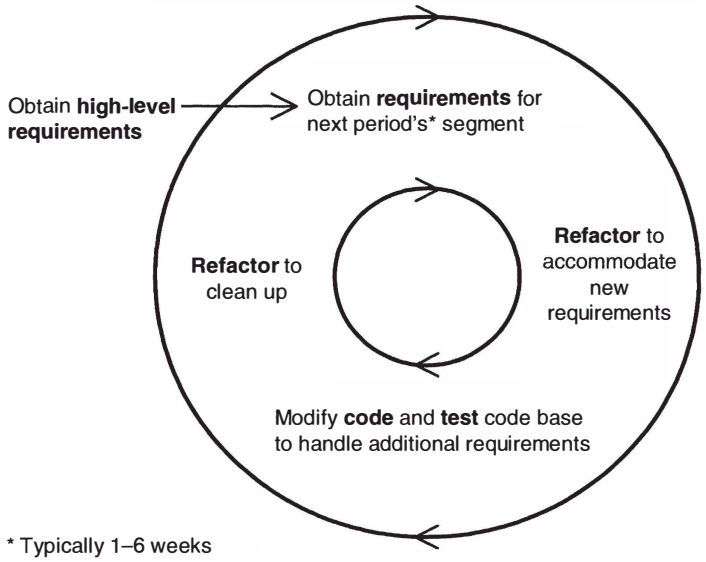


Figure 4.4 A typical agile development iteration

Refactoring is a process of altering the form of a code base while retaining the same functionality. The usual goal of a refactoring is to make the design amenable to the addition of functionality, thereby satisfying the agile desire to respond well to change. The very fact that the discipline of refactoring has been developed is a major factor making agile methods possible. This book covers refactoring in Chapter 24. Although refactoring is discussed later in the book, much of it will be useful before then and can be referred to throughout the book.

Agile methods employ the development cycle shown in Figure 4.4. Typically, the requirements are expressed in terms of user stories. Past experience in the project allows the team to assess its *velocity*: an assessment of the relative difficulty of stories and the rate at which it is able to implement them.

The schedule effects of agile methods can be seen in Figure 4.5. Planning is attenuated because there is less to plan. Requirements analysis, design, and testing are often confined to high levels. The emphasis is mostly code centered.

4.4 AGILE PROCESSES

"Agile development" isn't itself a specific process or methodology. Instead, it refers to any software process that captures and embraces the fundamental values and principles espoused in the Agile Manifesto. The following sections illustrate and describe three agile processes as representative examples.

- Extreme programming (XP)
- Crystal
- Scrum

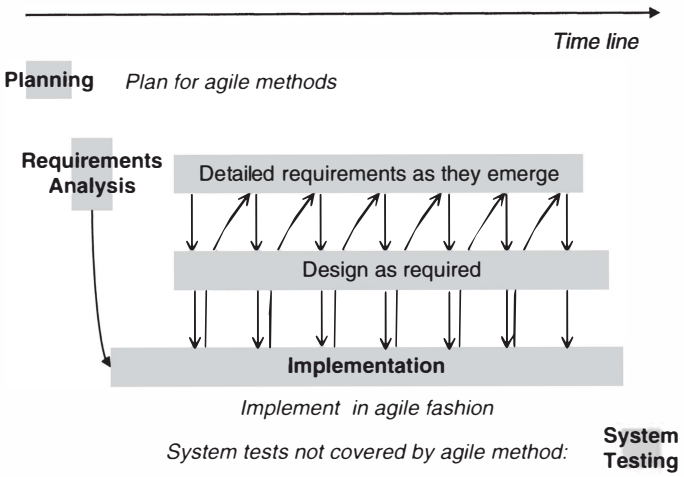


Figure 4.5 The agile schedule

4.4.1 Extreme Programming

In 1996, Kent Beck and colleagues began a project at DaimlerChrysler [5] using an approach to software development that appeared to make matters much simpler and more efficient. The methodology he developed and used became known as *Extreme Programming (XP)* [6].

Beck [6] cites four "values" guiding extreme programming: communication, simplicity, feedback, and courage. These are summarized in Figures 4.6 and 4.7. XP programmers communicate continually with their customers and fellow programmers. They keep their design simple and clean. They obtain feedback by testing their software, starting on day one. They deliver parts of 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 [5].

XP was created to deal effectively with projects in which requirements change. In fact, XP expects requirements to be modified and added. On many projects, customers start with only a vague idea of what they want. As a project progresses and a customer sees working software, specifics of what they want become progressively firm.

1. **Communication**
   - Customer on site
   - Pair programming
   - Coding standards
2. **Simplicity**
   - Metaphor: entity names drawn from common metaphor
   - Simplest design for current requirements
   - Refactoring

Figure 4.6 The "values" of extreme programming, 1 of 2

Source: Beck, Kent, "Extreme Programming Explained: Embrace Change," Addison-Wesley, 2000.

1. **Feedback** always sought
   - Continual testing
   - Continuous integration (at least daily)
   - Small releases (smallest useful feature set)
2. **Courage**
   - Planning and estimation with customer user stories
   - Collective code ownership
   - Sustainable pace

Figure 4.7 The "values" of extreme programming, 2 of 2

Source: Beck, Kent, "Extreme Programming Explained: Embrace Change," Addison-Wesley, 2000.

XP projects are divided into iterations lasting from one to three weeks. Each iteration produces software that is fully tested. At the beginning of each, a planning meeting is held to determine the contents of the iteration. This is "just-in-time" planning, and it facilitates the incorporation of changing requirements.

As code is developed, XP relies on continual integration rather that assembling large, separately developed modules. In the same spirit, releases are modest in added capability. The idea is to bite off a small amount of new capability, integrate and test it thoroughly, and then repeat the process.

Extreme programming recognizes the all-too-frequent breakdown in customer developer relationships, where each party develops a separate concept of what's needed and also how much the features will cost to develop. The result of this mismatch is, all too often, a mad dash for deadlines, including long working hours and an unsustainable pace. In response, extreme programming promotes a modus vivendi that's sustainable in the long term. In addition, it requires every developer to acknowledge up front that all code is everyone's common property, to be worked on as the project's needs require. In other words, no code "belongs" to a programmer. This is in keeping with engineering practice, where a bridge blueprint, for example, is the product of an organization and not the personal property of one designer.

XP is unique in using twelve practices that dictate how programmers should carry out their daily jobs. These twelve practices are summarized next [7].

1. **Planning Process.** Requirements, usually in the form of user stories, are defined by customers and given a relative priority based on cost estimates provided by the XP team. Stories are assigned to releases, and the team breaks each story into a set of tasks to implement. We described user stories in Section 4.3.
2. **Small Releases.** A simple system is built and put into production early that includes a minimal set of useful features. The system is updated frequently and incrementally throughout the development process.
3. **Test-Driven Development.** Unit tests are written to test functionality, before the code to implement that functionality is actually written.
4. **Refactoring.** Code is regularly modified or rewritten to keep it simple and maintainable. Changes are incorporated as soon as deficiencies are identified. We introduced refactoring in Section 4.3 and cover it in detail in Chapter 24.
5. **Design Simplicity.** Designs are created to solve the known requirements, not to solve future requirements. If necessary, code will be refactored to implement future design needs.

6. **Pair Programming.** This was described in Section 4.3.
7. **Collective Code Ownership.** All the code for the system is owned by the entire team. Programmers can modify any part of the system necessary to complete a feature they're working on. They can also improve any part of the system.
8. **Coding Standard.** For a team to effectively share ownership of all the code, a common coding standard must be followed. This ensures that no matter who writes a piece of code, it will be easily understood by the entire team.
9. **Continuous Integration.** Code is checked into the system as soon as it's completed and tested. This can be as frequent as several times per day. In this way the system is as close to production quality as possible.
10. **On-Site Customer.** A customer representative is available full-time to determine requirements, set priorities, and answer questions as the programmers have them. The effect of being there is that communication improves, with less hard-copy documentation—often one of the most expensive parts of a software project.
11. **Sustainable Pace.** XP teams are more productive and make fewer mistakes if they're not burned out and tired. Their aim is not to work excessive overtime, and to keep themselves fresh, healthy, and effective.
12. **Metaphor.** XP teams share a common vision of the system by defining and using a common system of describing the artifacts in the project.

4.4.2 Scrum

Scrum is an agile methodology developed in the early 1990s. It is named after the part of a rugby game that, in U.S. football terms, is a cross between the kickoff and a quarterback snap. As defined by the Merriam Webster dictionary, a scrum is "a rugby play in which the forwards of each side come together in a tight formation and struggle to gain possession of the ball using their feet when it is tossed in among them." In other words, scrum is a process that follows "organized chaos." It is based on the notion that the development process is unpredictable and complicated, and can only be defined by a loose set of activities. Within this framework, the development team is empowered to define and execute the necessary tasks to successfully develop software.

The flow of a typical scrum project is shown in Figure 4.8.

A project is broken into teams, or scrums, of no more than 6–9 members. Each team focuses on a self-contained area of work. A scrum master is appointed and is responsible for conducting the daily scrum meetings, measuring progress, making decisions, and clearing obstacles that get in the way of team progress. The daily scrum meetings should last no more than 15 minutes. During the meeting the scrum master is allowed to ask team members only three questions [8]:

1. What items have been completed since the last scrum meeting?
2. What issues have been discovered that need to be resolved?
3. What new assignments make sense for the team to complete until the next scrum meeting?

At the beginning of a project, a list of customer wants and needs is created, which is referred to as the "backlog." The scrum methodology proceeds by means of agile 30-day cycles called "sprints." Each sprint takes on a set of features from the backlog for development. While in a sprint, the team is given complete

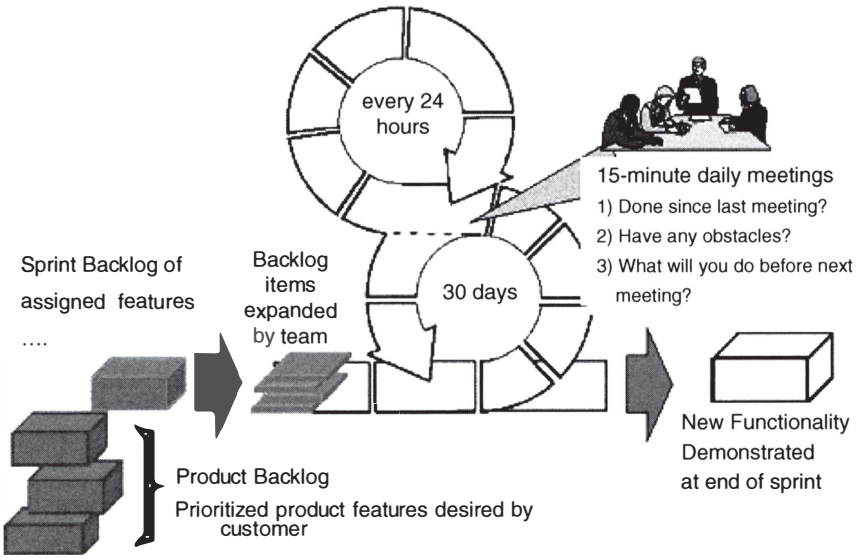


Figure 4.8 The scrum work flow

Source: Quoted and edited from http://www.controlchaos.com/about.

control of how they are to successfully complete the sprint. At the end of a sprint a customer demonstration is conducted for the customer. It serves several purposes, including [8]:

1. Demonstrating to the customer what has been accomplished.
2. Giving the developers a sense of accomplishment.
3. Ensuring that the software is properly integrated and tested.
4. Ensuring that real progress is made on the project.

At the conclusion of the demonstration, the leftover and new tasks are gathered, a new backlog is created, and a new sprint commences.

4.4.3 Crystal

Crystal is a family of agile methods developed by Alistair Cockburn. Each Crystal method shares common characteristics of "frequent delivery, close communication, and reflective improvement" [9]. Not all projects are the same, so different Crystal methodologies were created to address differences in project size and criticality. Figure 4.9 shows the different methodologies and the size project they are best suited to. Crystal methods are characterized by a color, starting at *clear* for small teams and progressing through to *orange*, *red*, and so on as the number of people increases. For example, Crystal Clear is geared for small teams of approximately 6 people; Yellow for teams of 10–20 people; Orange for teams of 20–40 people, and so on. The other axis defines the criticality of a project, where L is loss of life, E is loss of essential monies, D is loss of discretionary monies, and C is loss of comfort. Note that the row for loss of life is not shaded in Figure 4.9. This is because Cockburn had no experience applying Crystal to these types of projects when he created the

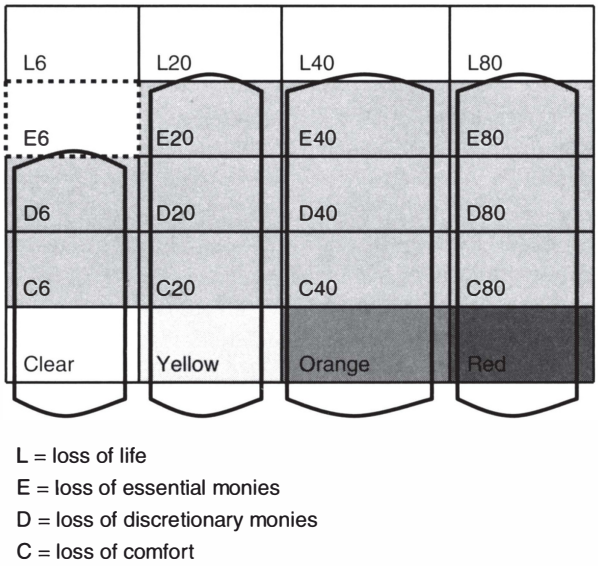


Figure 4.9 Coverage of various Crystal methodologies

Source: Adapted from Cockburn, Alistair, "Crystal Clear: A Human-Powered Methodology for Small Teams," Addison-Wesley, 2005.

chart. Crystal Clear doesn't explicitly support the E6 box, although Cockburn notes that teams may be able to adapt the process to accommodate such projects. Another restriction of Crystal is that it is applicable only to colocated teams.

Cockburn believes that developers do not readily accept the demands of *any* process (documentation, standards, etc). He strongly recommends accepting this by introducing the most limited amount of process needed to get the job done successfully, maximizing the likelihood that team members will actually follow the process.

All Crystal methodologies are built around three common priorities [9]:

1. Safety in the project outcome.
2. Efficiency in development.
3. Habitability of the conventions (i.e., the ability of developers to abide by the process itself).

Crystal projects exhibit seven properties to varying degrees [9]:

1. Frequent delivery.
2. Reflective improvement.
3. Close communication.
4. Personal safety.
5. Focus.
6. Easy access to expert users.
7. Technical environment with automated testing, configuration management, and frequent integration.

The first three properties are common to the Crystal family. The others can be added in any order to increase the likelihood of project safety and success.

4.5 INTEGRATING AGILE WITH NON-AGILE PROCESSES

The advantages of agile methods include the ability to adjust easily to emerging and changing requirements. The disadvantages include awkward roles for design and documentation. Cockburn's Crystal family of methodologies already acknowledges that different kinds of applications must be treated differently, even when the methods are agile.

Software process, after all, concerns the order in which we perform activities: For example, designing first and then coding from the design. One extreme is the waterfall process. As we have seen, there are many limitations in our ability to thoroughly perform the waterfall sequence once, or even a few times, iteratively. Changeable and unknown requirements are a principal reason. Regardless of the development process we use, we must make trade-offs in deciding how extensively to pursue a phase before moving to another phase. Consider, for example, the issue of how much effort to spend on planning a software enterprise.

One extreme project situation is when we are certain of obtaining all of the requirements of the end date, of the high-level design, and of who will be working on the job. In that case, we can and probably should develop a detailed plan.

The other extreme project situation is when we have little idea of any of these factors, believing that they will become clear only after the project is under way. In that case, planning at a detailed level would be a waste of time at best because it could not be even nearly accurate, and would probably even be misleading. We have no choice in this case but to begin the process, and revisit the plans as required. The extremes are illustrated in Figure 4.10.

Agile methods provide benefits but also costs, and these depend on several factors. Some are shown in Figure 4.11.

In order to gain the advantages of both agile and non-agile[1] processes, we try to integrate them. The means by which this can be performed depend on several factors, but particularly on the size of the job. As of

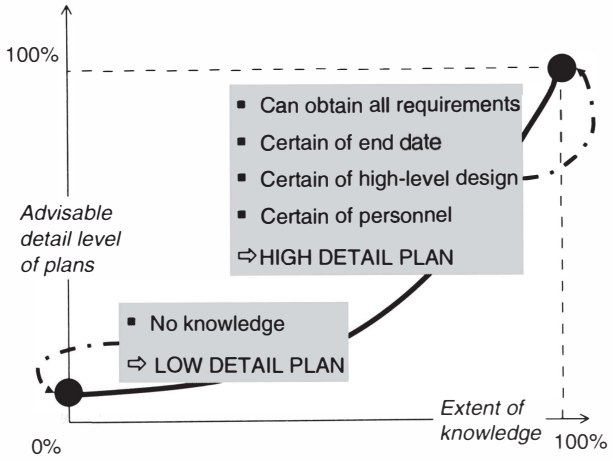


Figure 4.10 How detailed should plans be?

[1] Some authors characterize non-agile processes. For example, one practice is to call them "plan-based." The authors do not believe in a single characterization like this of non-agile processes; hence the blanket term "non-agile."

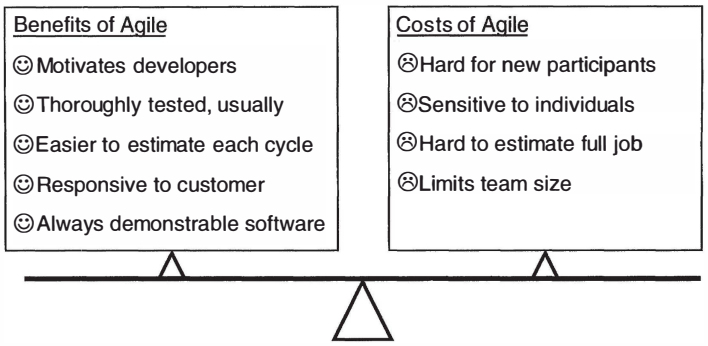


Figure 4.11 Trade-offs between agile and non-agile processes

2009, the conventional wisdom concerning the agile/non-agile split is shown roughly in Figure 4.12: The larger the job, the more non-agile process is required.

Agile processes emphasize code first, whereas non-agile ones advocate coding only from well-documented designs and designing only from well-documented requirements. These approaches appear to be almost contradictory, so combining them requires substantial skill and care. We will concentrate on methods for doing this in large jobs, where the challenges are greatest. We will call the two options *non-agile-driven* and *agile-driven* and will compare them.

4.5.1 A Non-Agile-Driven Approach

A common non-agile-driven approach is to initially approach the job without agility, then fit agile methods into the process after sufficient work has been done to define the agile roles and portions. We develop a plan, create a careful high-level design, and decompose the work into portions that can be implemented by teams in the agile manner. One can superimpose upon each agile team a process by which the accumulating

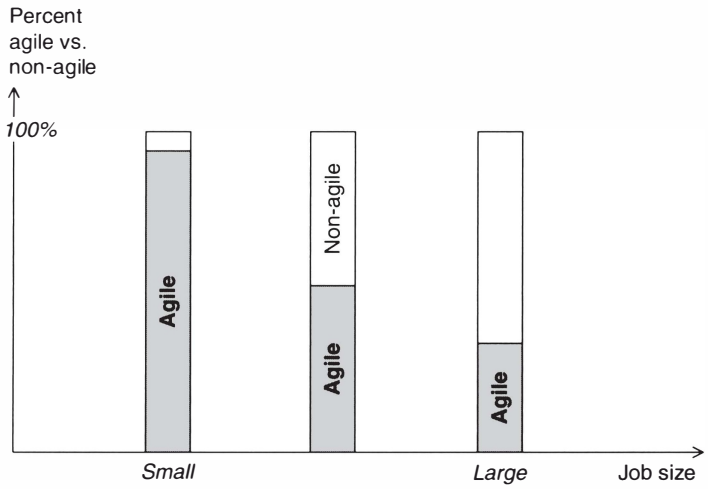


Figure 4.12 Conceptual agile/non-agile combination options: some conventional wisdom, circa 2009

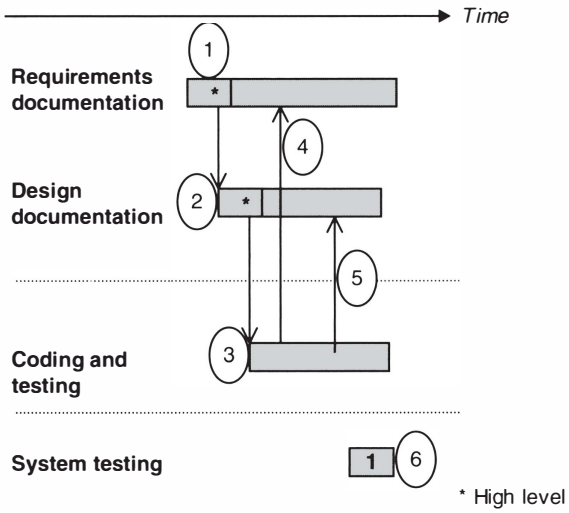


Figure 4.13 Integrating agile with non-agile methods 1: time line for a single iteration

requirements are gathered, and placed within a master requirements document. The same can be done with the accumulating design. Doing this with design is more difficult because design parts may actually be replaced and modified as refactoring takes place. Requirements tend to accumulate as much as they change.

The sequence of events is as follows:

1. High-level requirements are developed for the first iteration.
2. A high-level design is developed based on the high-level requirements.
3. Agile development by teams begins, based on these high-level documents.
4. Full requirements documentation is gathered from the work done by the agile teams as it progresses.
5. The design documentation is gathered from the agile teams at regular intervals to update the design document.
6. System testing not covered by the agile process is performed at the end, if necessary.
7. The process is repeated for each iteration.

This is illustrated for one waterfall iteration in Figure 4.13. Figure 4.14 shows this for multiple iterations.

4.5.2 An Agile-Driven Approach

For an agile-driven approach to large jobs, a (small) agile team can be set to work on significant aspects of the project until the outlines of an architecture appear. At that point, non-agile methods are used. This may involve reintegrating agile programming again as described in the non-agile-driven approach above. This has much in common with building a prototype. However, the difference is that the initial work is performed largely to develop an architecture rather than retire risks. In addition, the work is not planned as throw-away code. One agile-driven approach is shown in Figure 4.15.

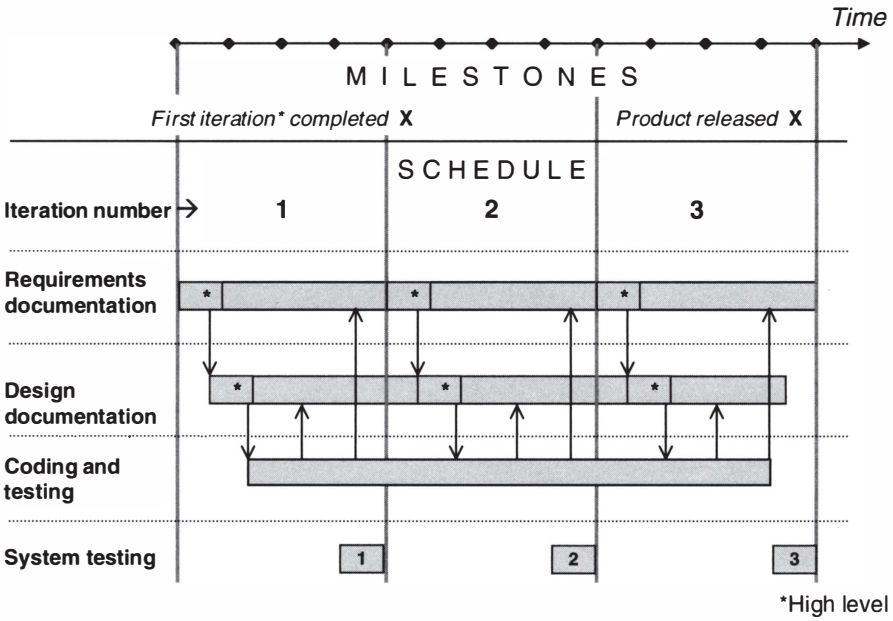


Figure 4.14 Integrating agile with non-agile methods 2: time line for multiple iterations

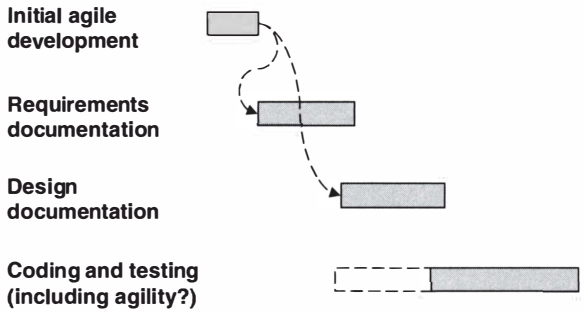


Figure 4.15 An agile-driven approach to large jobs

The case study sections for this part of the book (which appear in Chapters 5 and 6) contain two case studies that combine agile and non-agile methods.

4.6 SUMMARY

Agile software development was created as an alternative to existing plan-based approaches that were perceived to be too process heavy and rigid. A group of industry experts met in 2001 to share their vision for an alternative to these types of processes. They created the Agile Manifesto to capture their thoughts for a process that was adaptable, lean, and agile.

Agile processes emphasize the following:

- The need to collaborate closely with customers and other stakeholders
- Communication over documentation
- Frequent delivery of working software
- Self-organizing teams
- The need to embrace and plan for changing requirements

Any process that embraces the fundamental values of the agile manifesto is considered an agile process. Examples include extreme programming, scrum, and Crystal.

Extreme programming is based on four principles: communication, feedback, simplicity, and courage. It promotes practices such as test-driven development, refactoring, pair-programming, collective code ownership, continuous integration, and on-site customer.

Scrum defines a framework, or a loose set of activities that are employed by the scrum team. At the beginning of a project, a *backlog* of work, or requirements, is identified. Developers work on a subset of requirements in the backlog in 30-day *sprints*. Once a sprint begins, the team is given the freedom to employ any methods deemed necessary to complete their work successfully. A customer demo occurs at the end of each sprint. A new set of work is defined from the backlog for the next sprint and the cycle continues.

Crystal is a family of methodologies that address projects of different size and criticality. Crystal methods share the following characteristics: frequent delivery, reflective improvement, close communication, personal safety, focus, and easy access to expert users, and a technical environment with automated testing, configuration management, and frequent integration.

Agile and non-agile process can be integrated on projects in order to gain the advantages of both. The means by which this can be performed depend on several factors but particularly on the size of the project. One approach is to initiate a job without agility, then incorporate agile methods into the process after enough work has been accomplished in defining agile roles and responsibilities. Another approach is to initiate a job with an agile approach until the outlines of an architecture appear. At that point, non-agile methods are used. This may involve reintegrating agile programming again as described in the non-agile-driven approach above.

4.7 EXERCISES

1. The Agile Manifesto favors working software over extensive documentation. Under what circumstances can this cause problems if taken to an extreme?
2. a. In your own words, explain how agile processes adapt to and embrace changing requirements.
   b. Describe a scenario in which this might be counterproductive.
3. Name three benefits of the XP practice of testing software from "day one," always having working software available.
4. During the daily 15-minute scrum meeting, the leader is only allowed to ask the same three questions. What two additional questions might you want to ask? For each, explain its benefit toward achieving the goals of the meeting.
5. In your own words, explain how Crystal adapts to various types of projects.

BIBLIOGRAPHY

1. Beck, Kent, Mike Beedle, Arie van Bennekum, and Alistair Cockburn, *"Manifesto for Agile Software Development,"* Feb 2001. http://agilemanifesto.org/ [accessed November 5, 2009].
2. Highsmith, Jim, *"History: Agile Manifesto,"* 2001. http://www.agilemanifesto.org/history.html [accessed November 5, 2009].
3. Highsmith, Jim, and A. Cockburn, "Agile Software Development: The Business of Innovation," *IEEE Computer*, Vol. 34, No. 9, September 2001, pp. 120–122.
4. Cohn, Mark, *"User Stories Applied: For Agile Software Development,"* Addison-Wesley, 2004.
5. Wells, Don, *"Extreme Programming: A Gentle Introduction."* http://www.extremeprogramming.org/ [accessed November 15, 2009].
6. Beck, Kent, *"Extreme Programming Explained: Embrace Change,"* Addison-Wesley, 2000.
7. Jeffries, Ron, *"XProgramming.com: An Agile Software Development Resource."* http://xprogramming.com [accessed November 15, 2009].
8. Beedle, Mike, Martine Devos, Yonat Sharon, and Ken Schwaber, *"SCRUM: An extension pattern language for hyperproductive software development."* http://jeffsutherland.com/scrum/scrum_plop.pdf [accessed November 15, 2009].
9. Cockburn, Alistair, *"Crystal Clear: A Human-Powered Methodology for Small Teams,"* Addison-Wesley, 2005.