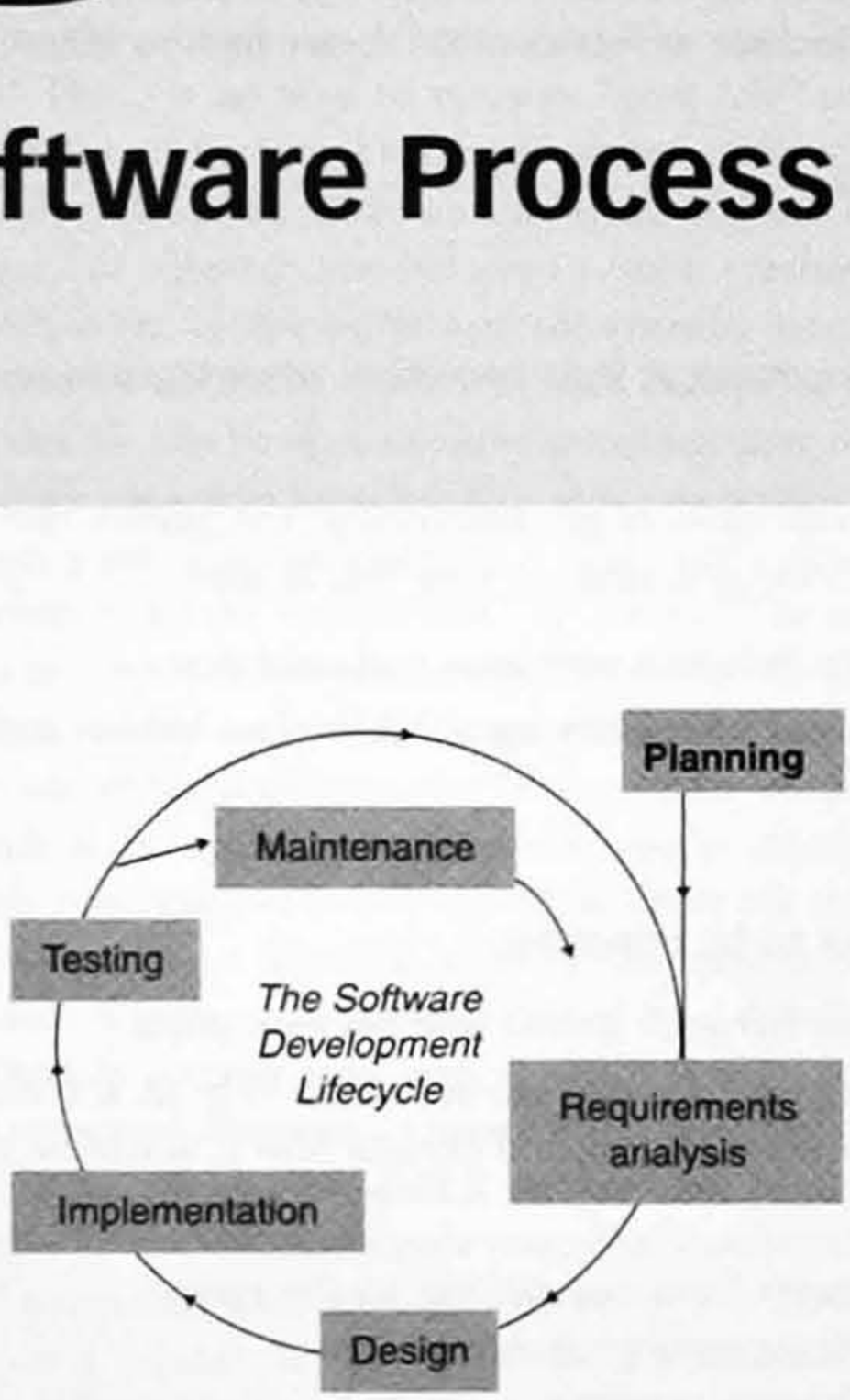
3

Software Process



- What are the main activities of software processes?
- What are the main software process types?
- How would a team such as a student team go about selecting a process?

Figure 3.1 The context and learning goals for this chapter

A software project progresses through a series of activities, starting at its conception and continuing even beyond its release to customers. There are numerous ways for individuals and teams to organize these activities. Typically, a project is organized into *phases*, each with a prescribed set of activities conducted during that phase. A *software process* prescribes the interrelationship among the phases by expressing their order and frequency, as well as defining the deliverables of the project. It also specifies criteria for moving from one phase to the next. Specific software processes, called software process *models*—or *life cycle* models—are described. This chapter is organized to first describe the individual activities of software processes, and then to describe process models—the various ways in which these activities can be pursued.

In addition to the activities prescribed by process models, there is a set of generic activities, called *umbrella activities*, that are implemented throughout the life of a project. For instance, projects contain certain

risks that if they come to pass can affect the successful outcome of the software. Risks need to be identified, monitored, and managed throughout the life of a project. This is an umbrella activity known as risk management. Other umbrella activities include project management, configuration management, and quality management. Project management is covered in Part III, configuration management in Chapter 6, and quality management throughout, but especially in Part VII.

People sometimes associate process with overhead, unnecessary paperwork, longer schedules, and so on. In other words, they feel that process doesn't add value, or worse, that it adversely affects the success of a project. When implemented incorrectly, software processes can indeed lead to undesirable results. However, as we explain in this chapter, if processes are applied with a degree of flexibility and adaptability, they are of great benefit and invaluable to the successful outcome of projects.

In certain situations, a version of working software containing a subset of the overall product functionality is required early in the overall schedule. This version of the emerging product, called a *prototype*, typically implements functionality that is deemed a risk to the project. Typical reasons to build prototypes include proving technical feasibility and validating the usability of a user interface. Prototypes are covered in detail in Section 3.2.3.1.

To reinforce the contents of this chapter and to provide guidance to students at the same time, this chapter ends with a section devoted to getting student teams started with a term project.

3.1 THE ACTIVITIES OF SOFTWARE PROCESS

Most software process models prescribe a similar set of phases and activities. The difference between models is the order and frequency of the phases. Some process models, such as the waterfall, execute each phase only once. Others, such as iterative models, cycle through multiple times. This section describes the phases that are prevalent in most software process models, and is summarized in Figure 3.2. The phases in Figure 3.1 and Figure 3.2 are identical, except that Figure 3.1 combines inception and planning. The parts are explained below. Specific process models and how they implement the phases are covered in Section 3.2.

---

1. **Inception**
   Software product is conceived and defined.

2. **Planning**
   Initial schedule, resources and cost are determined.

3. **Requirements Analysis**
   Specify what the application must do; answers *"what?"*

4. **Design**
   Specify the parts and how they fit; answers *"how?"*

5. **Implementation**
   Write the code.

6. **Testing**
   Execute the application with input test data.

7. **Maintenance**
   Repair defects and add capability.

Figure 3.2 The main phases of a software project, showing inception as a separate phase

Inception

This is the phase where initial product ideas are formulated and a vision of the software is conceived. Whether it is a brand new product or an improvement to existing software, every project starts with an idea of what is to be built. Major functionality and project scope is defined. Target customers and market segments are identified. Customers and stakeholders may be consulted for high-level input into software functionality. However, stakeholder involvement at this stage is different from that during the subsequent requirements analysis phase. During inception, the goal is to gather very high level feedback. As an example, suppose a company is considering building a video store application. A market analysis may be conducted to determine the existence of competing video store applications. A summary of their features and an analysis of their strengths and weaknesses is compiled. The commercial success of the proposed application is determined by identifying potential new customers and contacting them for information concerning:

- Their satisfaction with their current application
- Ideas for needed functionality and features
- Their likelihood of adopting a new product

As an example, suppose that a currently deployed system requires custom hardware, but potential customers would prefer using standard Windows PCs. Based on this analysis and feedback, the need for and viability of the proposed application is determined. If it is deemed promising, high-level functionality for the application is defined based on the feedback received, and the project proceeds to the planning phase.

Planning

Once a high-level idea for the application is developed, a plan is formulated to produce it. Since this is still very early in the overall life cycle and detailed information is not yet available, the plan will be a rough estimate. However, some plan must be in place to understand the scope and cost of the project. A project plan that identifies the high-level activities, work items, schedule, and resources is developed. From this information a cost estimate can be developed. This step is very important to determine the feasibility of a project. For example, if the cost is deemed to be too high, a reduction in features may necessary. If the product release date is too late, more resources may be added. It is critical that these types of problems be identified as early as possible so corrective action can be taken.

The results of this phase are typically captured in a Software Project Management Plan (SPMP). The SPMP and project management are covered in Part IV. Because the plan is only a rough one at this stage, it is necessary to modify and adapt it throughout the life of the project. For example, suppose that during subsequent requirements analysis potential new customers are identified and additional functionality is requested. Using the video store application as an example, suppose a request is received to add additional workstations to stores so customers can retrieve detailed movie information, such as release date, movie studio, cast, director, producer, genre, and so on. This new functionality may require additional project resources and changes to the schedule. The SPMP would be updated as a result. Some life cycle models, such as the spiral model described below, prescribe specific points in the process where planning information is revisited and updated when required. Agile projects, as will be seen, keep planning to a minimum.

In addition to the activities described above, a plan is developed for managing project artifacts such as technical specifications and source code. This is known as *configuration management*, and includes tasks such as tracking changes to artifacts and handling multiple versions. Configuration management is planned at this early project stage, before the first project artifacts are generated. Configuration management is covered in detail in Chapter 6.

Requirements Analysis

During this phase, detailed information regarding customer wants and needs, and problems the software is intended to solve are gathered. Information gathered and documented is in much greater detail than it was in the inception phase. During inception, only enough information is required to start planning the project. During requirements analysis, specific product functions and features are defined along with requirements such as performance, reliability, and usability. Requirements are generated in a form that is completely readable and understandable by customers. High-level requirements in particular are typically expressed in ordinary English (or the local language). Various techniques are used to obtain this information, including customer interviews and brainstorming sessions. Requirements describe *what* the application is intended to accomplish. They are specified in sufficient detail so they can be used as input for the subsequent design phase, which defines *how* the software will be built. The results of the analysis are typically captured in a formal Software Requirements Specification (SRS), which serves as input to the next phase.

Software Design

The purpose of software design is to define how the software will be constructed to satisfy the requirements. That is, the internal structure of the software is defined. The two main levels of software design are software architecture and detailed design. Software architecture is analogous to the overall blueprints of a house as a whole. Blueprints specify the number of rooms and their layout, door and window locations, number of floors, and so on. Software architecture specifies how the software is broken into subsystems or modules and the software interfaces between them. For example, the architecture for a video store application may consist of components such as a user interface module, a problem domain module, and a database module. Agile projects generally perform design, implementation, and much testing in an interweaved fashion rather than calling out design as a separate phase.

The detailed design is analogous to the details contained in a house blueprint. In house plans, details such electrical wiring and plumbing are specified. In software, details such as algorithms and data structures are specified. Other aspects of software design include user interface design and database design. The output of software design is specified in a Software Design Document (SDD) and is used as input to the implementation phase. Software design is covered in detail in Part V.

Implementation

This phase consists of programming, which is the translation of the software design developed in the previous phase to a programming language. It also involves integration, the assembly of the software parts. The output consists of program code that is ready to be tested for correctness. For agile techniques, design implementation and much testing are performed in tandem.

Testing

In this phase, the code produced during the implementation phase is tested for correctness. Testing is performed at three levels. First, individual modules are tested by developers. Second, modules are integrated and tested to ensure that they interface properly. Third, once all the modules have been integrated, the entire system is tested to ensure that it meets the user requirements. System testing is typically conducted by an independent quality assurance (QA) team. Recall from Chapter 2 that this testing process is called *validation*. Once system testing has been completed, several levels of customer testing are conducted. During *beta testing* the software is as close to the final release as possible, and is given to part of the customer community with the understanding that they report any defects they find. The goal is to uncover a set of problems that could only be discovered with this type of "real-world" testing. Once beta testing is complete, *acceptance testing* is

conducted on the "final" release of software. Acceptance tests are comprised of a subset of system tests, and are conducted by either the customer or a representative of the customer to ensure that it meets the customer's criteria for release. Sometimes a round of acceptance testing is conducted prior to beta testing, to make sure that the system meets certain criteria before it is given to customers for beta testing.

Maintenance

After the software is officially released to customers, the maintenance phase commences. During maintenance, modifications are made to the software that arise through one of the following:

- The repair of software defects that are discovered during normal customer use of the system
- Customer requests for enhancements
- A desire to improve attributes of the system such as performance or reliability

Modifications to the software are bundled together, and new versions of the software with these modifications are released. Maintenance is discussed in detail in Chapter 29.

Figure 3.3 shows examples of artifacts produced during each phase for our example video store application.

- **Inception**

  " . . . An application is needed to keep track of video rentals . . . "

- **Planning** (Software Project Management Plan)

  " . . . The project will take 12 months, require 10 people and cost $2M . . . "

- **Requirements Analysis** (Product: Software Requirements Spec.)

  " . . . The clerk shall enter video title, renter name and date rented. The system shall . . . "

- **Design** (Software Design Document: Diagrams and text)

  " . . . classes *DVD, VideoStore,* . . . , related by . . . "

- **Implementation** (Source and object code)

  . . . class DVD{ String title; . . . } . . .

- **Testing** (Software Test Documentation: test cases and test results)

  " . . . Ran test case: *Rent "The Matrix" on Oct 3, rent "SeaBiscuit" on Oct 4, return "The Matrix" on Oct 10* . . .
  Result: *"SeaBiscuit" due Oct 4, 2004 balance of $8. (correct)* . . . "

- **Maintenance** (Modified requirements, design, code, and text)

  Defect repair: "Application crashes when balance is $10 and attempt is made to rent *"Gone With the Wind"* . . . "
  Enhancement: "Allow searching by director."

Figure 3.3 The main phases applied to a video store application

3.2 SOFTWARE PROCESS MODELS

Software process models define the order and frequency of phases in a project. The following sections describe the most important process models, starting with the classical waterfall process.

3.2.1 The Waterfall Process Model

One of the oldest software process models was defined by Royce [1] and is called the *waterfall process*. Despite its many weaknesses (described later in the section), the waterfall process is still in widespread use today and is the basis for many other more effective processes.

Figure 1.6 in Chapter 1 illustrates the phases of the waterfall process. Note that Royce's model begins with the requirements phase—he did not include inception and planning phases. In practice, organizations implementing a waterfall process would typically start with these phases. The waterfall executes in a sequential manner through each phase, concluding with maintenance. The output from one phase is used as input for the next, implying that a phase is not started until the previous one has completed. It is accepted in waterfall processes that there is a small overlap between adjacent phases. As an example, some personnel will be performing the last part of requirements analysis while others will have already started the design phase.

The waterfall process is characterized by documents generated by the time each phase is completed. These include requirements specification and design specification, for example. Also, there are usually entrance and exit criteria between phases to ensure that a phase is completed successfully before moving on.

In practice, an iterative relationship between successive phases is usually inevitable. For example, after the requirements are completed, unforeseen design difficulties may arise. Some of these issues may result in the modification or removal of conflicting or non-implementable requirements. This may happen several times, resulting in looping between requirements and design. Another example of feedback is between maintenance and testing. Defects are discovered by customers after the software is released. Based on the nature of the problems, it may be determined there is inadequate test coverage in a particular area of the software. Additional tests may be added to catch these types of defects in future software releases. A general guideline, often accepted to still be within the waterfall framework, is that feedback loops should be restricted to adjacent phases. This minimize potentially expensive rework if the feedback were to span multiple phases. An modified model illustrating this iterative relationship is shown in Figure 3.4.

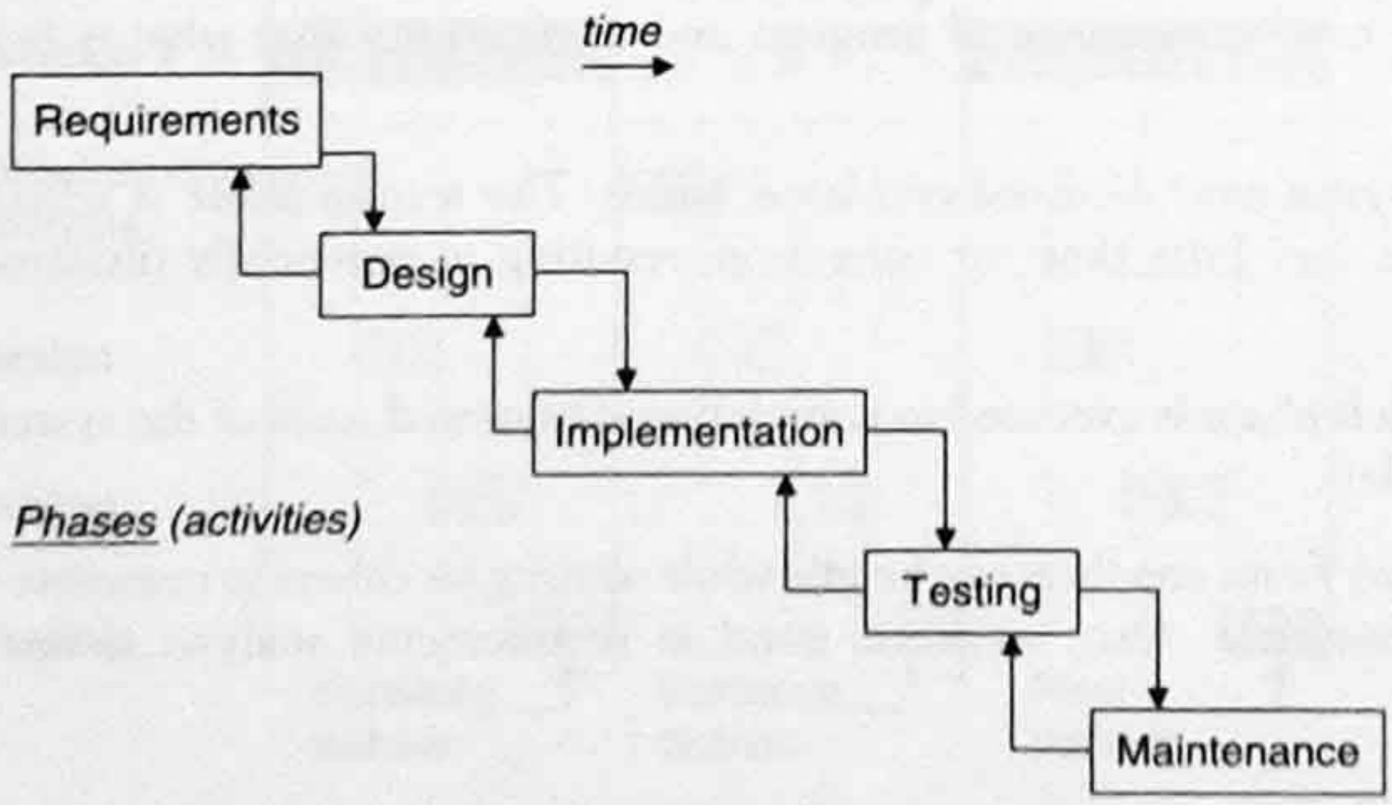


Figure 3.4 The waterfall software development process in practice: feedback is inevitable

A major limitation of the waterfall process is that the testing phase occurs at the end of the development cycle—the first time the system is tested as a whole. Major issues such as timing, performance, storage, and so on can be discovered only then. Even with thorough up-front analysis, these kinds of factors are difficult to predict until encountered during testing. Solutions may require either complex design changes or modifications to the requirements that the design is based on, necessitating iteration beyond adjacent phases. Discovering and repairing these types of defects so late in the development cycle can jeopardize the project schedule.

Major advantages and disadvantages of the waterfall process are summarized below.

Advantages

- *Simple and easy to use*: Phases are executed and completed serially, with specific entrance and exit criteria for moving between phases. Orderly execution of phases is easy to comprehend.
- *Practiced for many years and people have much experience with it*: The process is well understood, and many people are comfortable with its execution.
- *Easy to manage due to the rigidity of the model*: Each phase has specific deliverables and a review process.
- *Facilitates allocation of resources* (due to sequential nature of phases): Distinct phases facilitate allocation of personnel with distinct skills.
- *Works well for smaller projects where requirements are very well understood*: It isn't necessary to add complexity of iteration if requirements are well known up front.

Disadvantages

- *Requirements must be known up front*: It's difficult to imagine every detail in advance. Most projects start out with some uncertainty, and more details are learned as the project progresses.
- *Hard to estimate reliably*: To gain confidence in an estimate, there may be the need to design and implement parts, especially riskier ones. Estimates become more precise as the project progresses.
- *No feedback of system by stakeholders until after testing phase*: The process does not facilitate intermediate versions. Stakeholders often need reassurance of progress and confirmation that what is being developed meets requirements.
- *Major problems with system aren't discovered until late in process*: The testing phase is where these problems are found, but it leaves very little time for correction, resulting in potentially disastrous effects on project schedule and cost.
- *Lack of parallelism*: Each phase is executed to completion. Disjointed parts of the system could otherwise be completed in parallel.
- *Inefficient use of resources*: Team members can be idle while waiting for others to complete their dependent tasks or for phases to complete. Also, someone good at requirements analysis is not necessarily good at programming.

Because of factors like these, alternative (but related) processes are often employed, and these are covered in subsequent sections.

3.2.2 Iterative and Incremental Development

The waterfall process is characterized by a sequential execution through the phases. It is generally agreed that the order of the phases dictated by the waterfall is fundamental: gather requirements, create a software design to realize the requirements, implement the design, and test the implementation. The problem arises, however, when this is scaled up to gather *all* the requirements, do *all* of the design, implement *all* of the code, and test *all* of the system in a linear fashion [2]. Except for the smallest of projects this is impractical. As the system is developed and refined, more is learned and a need arises to revisit each of the phases. That is, software is more naturally developed in a cyclical manner, where a part of the system is developed and tested, feedback is gathered, and based on the feedback more of the system is developed. This reflects the fact that not everything is understood at the start of a project. *Iterative* processes accept this cyclical nature and are discussed in the rest of this section. Figure 3.1 is drawn in a way that reflects this.

An *iterative* process is typified by repeated execution of the waterfall phases, in whole or in part, resulting in a refinement of the requirements, design, and implementation. An iterative process is *incremental* if each iteration is relatively small. At the conclusion of such an iteration, a piece of operational code is produced that supports a subset of the final product functionality and features. Project artifacts such as plans, specifications, and code evolve during each phase and over the life of the project. Artifacts are considered complete only when the software is released.

As defined by Cockburn [3], incremental development is "a scheduling and staging strategy that allows pieces of the system to be developed at different times or rates and integrated as they are completed." This implies that iterations need not be executed serially, but can be developed in parallel by separate teams of people. Successive increments implement an increasing set of functionality and features until the final iteration, which produces the final product. Figure 3.5 illustrates this concept for a project with three iterations. Note that the output of each testing phase is a working subset of the final product that incrementally contains more functionality.

An iteration other than an incremental one is sometimes defined as "a self-contained mini-project, with a well-defined outcome: a stable, integrated and tested release" [2]. That is, each iteration is a self-contained

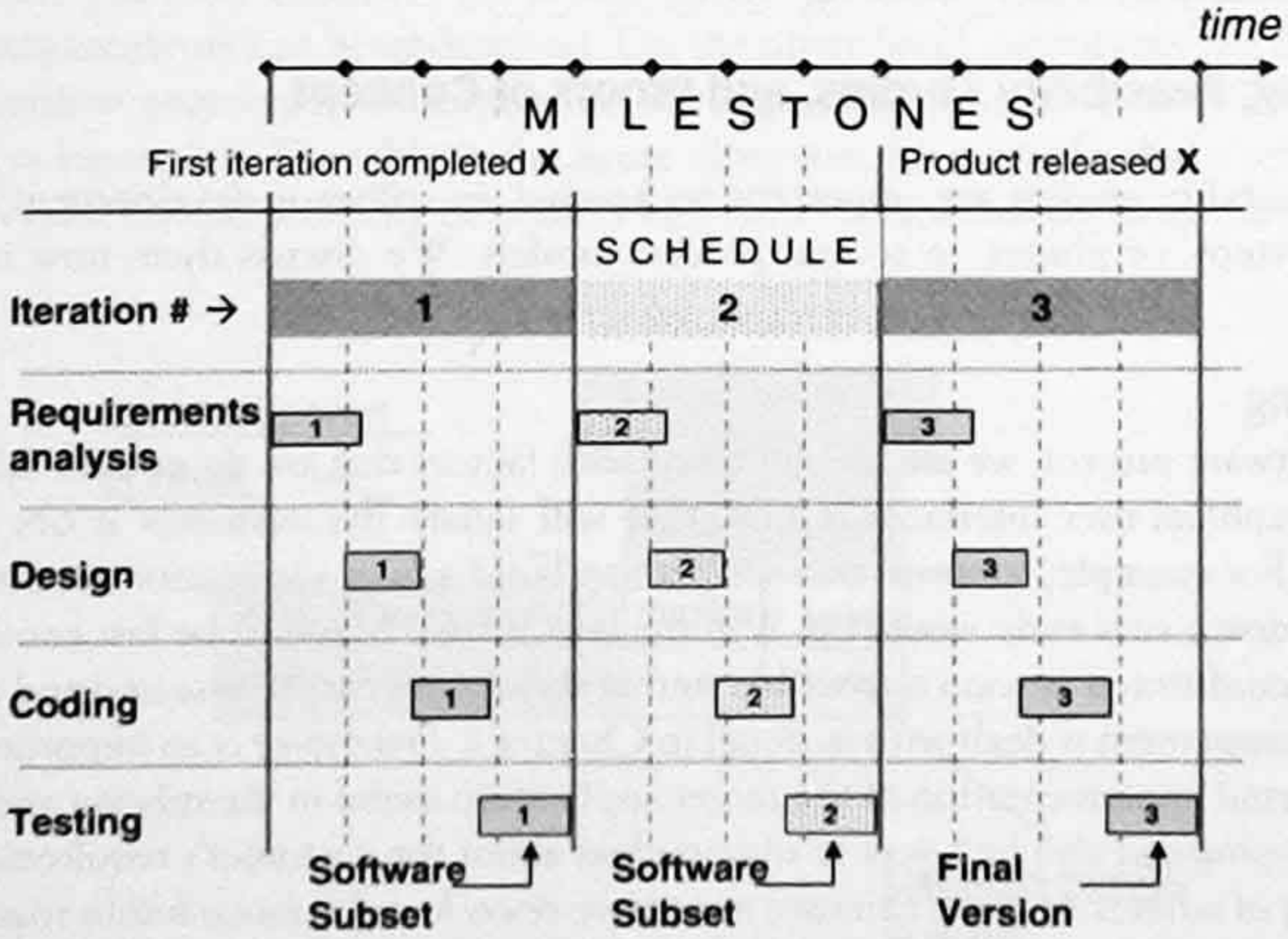


Figure 3.5 The iterative software development process: an example with three iterations

Source: Cockburn, Alistair. "Unraveling Incremental Development," January 1993, http://alistair.cockburn.us/Unraveling+incremental+development.

project with its own set of activities, plan, objectives, and measurable evaluation criteria. The "release" is a working version of software that is either used internally by the project team or externally by stakeholders and customers. Types of releases can be [2]:

- A *proof of concept*, or *feasibility study*, that is used to demonstrate or investigate feasibility of a particular aspect of the software. This includes producing software or simulations. These are covered in Section 3.2.3.2.
- A *prototype* that is a working version of software demonstrating a particular capability that is deemed high risk. Prototypes are covered in Section 3.2.3.1.
- An "internal" release that is only used by the development team and is used to ensure that development is on track, elicit feedback, and provide a basis for further development and additional capabilities.
- An "external" release that is shipped to customers for evaluation.

Treating each iteration as a self-contained project allows clear and manageable objectives to be set, and reduces overall project complexity by breaking it down into smaller pieces. By producing working software at the end of each iteration, project progress can more easily be monitored and planned, and feedback can be elicited to ensure that its capabilities are meeting stakeholder requirements. Typically, early iterations generate software releases that address aspects of the project with the highest risk. In this way, as the project progresses the overall project risk level is reduced.

An example of an iterative and incremental process that is used within parts of Microsoft is reported by Cusumano and Selby [4], to which they give the name "synch-and-stabilize." Product features are defined at a high level during the initial phase of a project, with the idea that many will evolve as product development proceeds. The product is divided into parts, each consisting of a set of features, with small teams assigned to each part. The project is also divided into parts, or iterations, each with its own completion milestone. Each iteration includes several features. Feature teams employ incremental synchronization by combining their work and stabilizing the resulting system on a daily or weekly basis. In this way the evolving software application is continually kept in a "working" state.

3.2.3 Prototyping, Feasibility Studies, and Proofs of Concept

Prototypes and feasibility studies are important techniques in software development, and are explicitly included as formal steps, or phases, in several process models. We discuss them now in more detail.

3.2.3.1 Prototyping

On beginning a software project, we are usually faced with factors that we do not yet fully understand. The look and feel of graphical user interfaces (GUIs) that will satisfy the customer is one common example. Timing is another. For example, suppose that we plan to build a Java application to control an airplane. A critical issue to pin down very early would be: Will the Java Virtual Machine be fast enough? Each unknown factor or *risk* is best confronted as soon as possible so that its severity can be assessed and a plan developed to deal with it. Risk management is dealt with in detail in Chapter 8. *Prototyping* is an important risk management technique. It is a partial implementation of the target application useful in identifying and retiring risky parts of a project. Prototyping can also be a way to obtain ideas about the customer's requirements. An increase in one's understanding of what is to come can save expensive rework and remove future roadblocks before they occur. Agile processes contain some of the benefits of prototyping because they deal at all times with working code. However, they do not have all of the benefits, since prototypes proceed in parallel with the main thread of the project, whether agile or not.

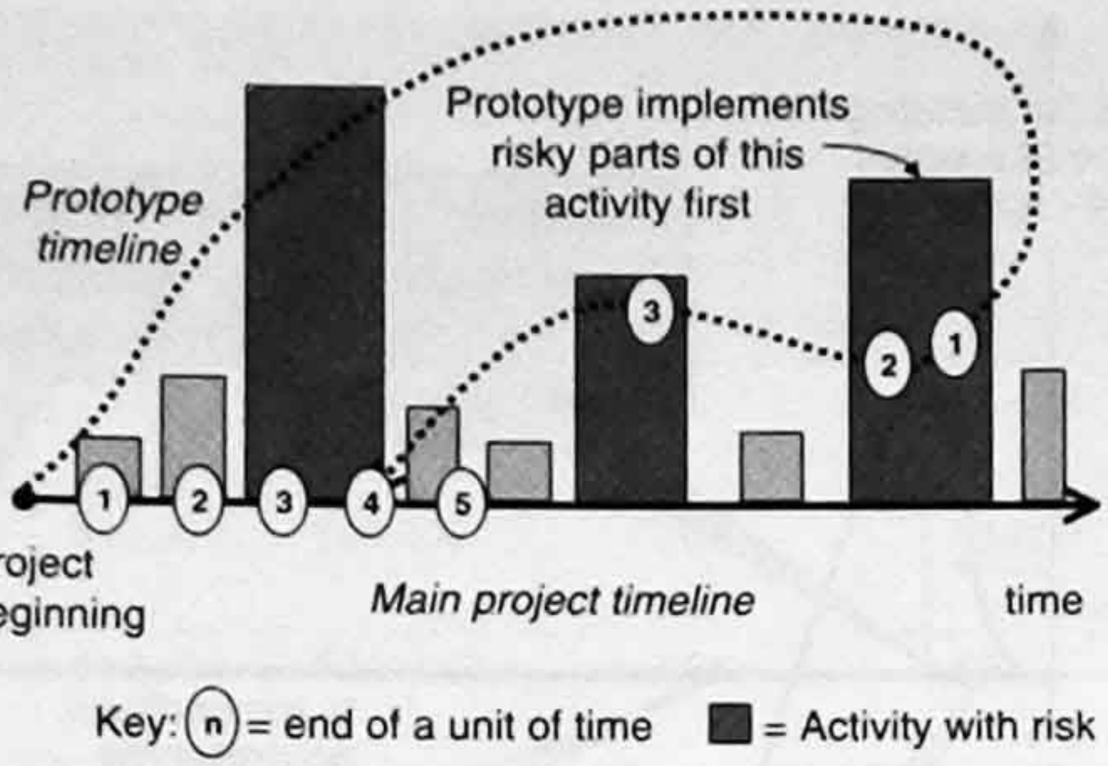


Figure 3.6 An illustration of prototyping in the context of a development project

As Figure 3.6 shows, work on a prototype typically progresses in parallel with regular work on the project. The risks are prioritized and the prototype is geared toward as many of the most important ones as time allows. In the example shown in Figure 3.6, the prototype ignores the large risk near the beginning of the project because it will be dealt with early in the ordinary course of the project in any case.

Large programs, such as billion dollar defense projects, utilize extensive prototyping to retire risks and to guide the requirements and design of the real thing. For example, before the U.S. Navy built the Aegis generation of shipboard systems, it built an entire scaled-back version, complete with software and hardware, installed on a ship for the purpose. This prototype served to indicate to the Navy what the main problems might be with the eventual systems. It also helped the Navy to develop the requirements and design of the eventual system.

Simple graphics showing GUIs using paint-type tools may be sufficient if the prototype's goal is to envision the interfaces. The more extensive the prototype, the more risks can be retired with it and the more easily a customer's requirements can be understood. On the other hand, prototypes are themselves software applications, so extensive prototypes are expensive. A rough assessment as to whether or not to build a prototype is shown in Figure 3.7. The table in the figure illustrates, for example, that a relatively inexpensive prototype with high value should probably be built. "High value" means that building the prototype helps the

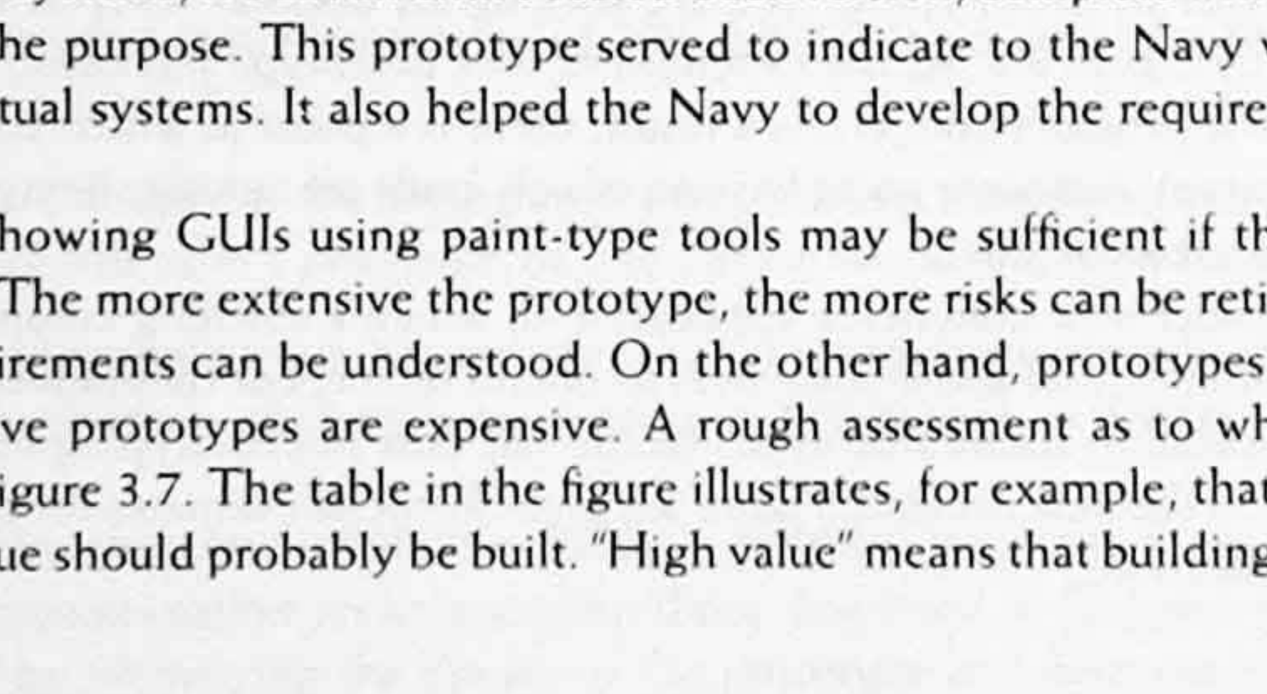


Figure 3.7 A rough calculation of whether developing a prototype would be worth it

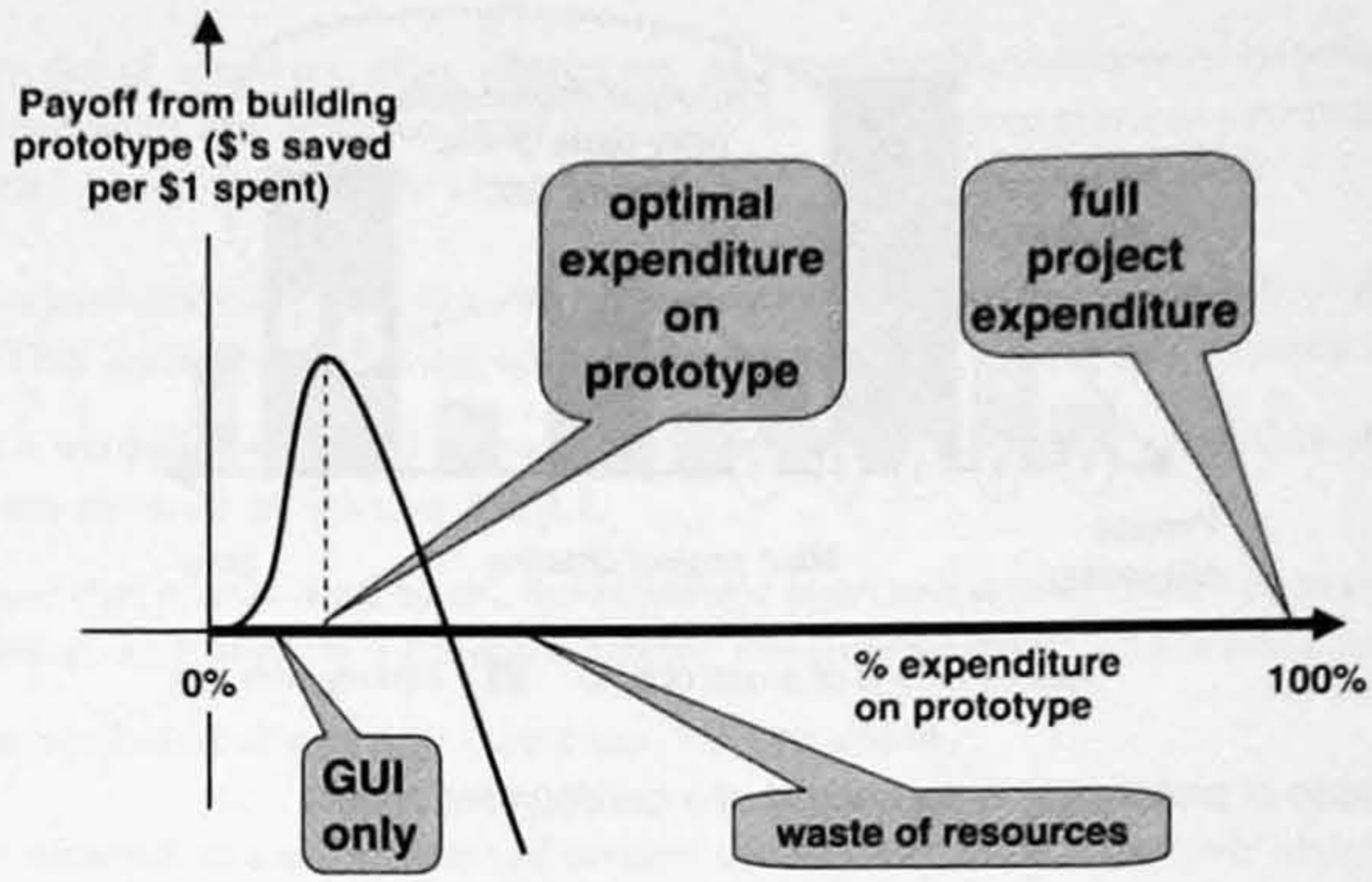


Figure 3.8 Concept of when it is worthwhile to build a prototype (near the beginning)

customer understand better what kind of product is likely to emerge, helps the engineers understand better what kind of product should emerge, and/or retires a development risk.

Many cases fall into the "maybe" category of the table in Figure 3.7, and more detailed analysis is required to assess the value of prototyping. We are seeking an optimal level of effort to be spent on a prototype, as suggested by Figure 3.8. As the expenditure on a prototype increases, its usefulness increases, but so does its drain on the project's budget. As a result, there is a point at which the payoff is optimal (the maximum point for the curve), and some point beyond which funds are actually being squandered (where the curve drops below the horizontal axis).

As an example, consider an e-commerce application in which a clothing company wants to sell goods online, retain customer profiles, and allow customers to obtain pictures of themselves wearing clothing from the catalog. A typical calculation about prototypes factors the cost of prototyping features and the potential for using prototype code in the final product. Figure 3.9 gives financial estimates for prototyping four parts of the clothing vendor application:

1. GUI screenshots
2. Transaction security
3. Complete transaction
4. Customer tries on clothing

For each of the four application features considered for the prototype, several estimates can made: the cost of building the feature, the percentage of the feature's implementation that will be reused in the application itself (i.e., not discarded), and the "gross benefit" from the effort. The gross benefit here estimates the gain from prototyping the feature, excluding reuse of the code and excluding all expenses. For example, as shown in Figure 3.9, we have estimated that if the "Customer tries on clothing" feature were to be prototyped, it would save a minimum $20,000 in development costs. This estimate is based on factors such as the following:

	Estimated cost	Gross Benefit excluding code reuse		Percentage of prototype code reused in application	Net Payoff		
		min	max		min	max	average
Prototype feature	*B*	*D*	*E*	*C*	*D*-(1-*C*)*B*	*E*-(1-*C*)*B*	
1. GUI screenshots	$10,000	$10,000	$80,000	50%	$5,000	$75,000	$40,000
2. Transaction security	$50,000	$10,000	$300,000	80%	$0	$290,000	$145,000
3. Complete transaction	$80,000	$10,000	$400,000	50%	−$30,000	$200,000	$85,000
4. Customer tries on clothing	$120,000	$20,000	$140,000	30%	−$64,000	$56,000	−$4,000

Figure 3.9 Payoff calculation example for building a prototype for a clothing application

- Preventing time wasted on proposed requirements that the prototype shows are not really needed (e.g., minimum of three unneeded requirements out of 100; $300,000 budgeted for the requirements phase = $9000 saved)
- Implementing a software design for the "trying on clothes" feature, thereby retiring some development risks (e.g., estimate that this will save a minimum of one person-week of design time = $2000)
- Rework that would have resulted from the customer changing requirements only after seeing the developed product (e.g., rework minimum of three requirements at $3000 each = $9000)

The minimum savings therefore is $9000 + $2000 + $9000 = $20,000. Estimating the cost of building the prototype can use approximation techniques like those described in Chapter 8. An estimation of code reuse can be performed by identifying the classes of the prototype and determining which are likely to be usable in the actual application.

This type of estimation often consists of adding up the estimation of smaller parts, which is often more feasible. Bracketing each estimate between a minimum and a maximum can help to make this process a little easier for the estimator.

Once these estimates are made, the best- and worst-case scenarios for each feature can be computed. This is shown in Figure 3.9. The minimum payoff value is obtained by taking the most pessimistic combination: the highest costs, the lowest gross benefits, and the lowest reuse percentages. The maximum payoff is calculated correspondingly. For example, the maximum payoff (the most optimistic alternative) for the "GUI screenshot" prototype feature is as follows:

$$
\begin{aligned}
&[\text{maximum estimated benefit}] - [\text{ minimum estimated costs}] \\
&= \$80,000 - [(\text{minimum estimated cost}) \times (\text{percent not reusable})] \\
&= \$80,000 - [\ \$10,000 \times 50\%\ ] = \$75,000
\end{aligned}
$$

Averaging is one way to deal with the spread between best and worst cases. The result suggests a positive payoff for all proposed prototype features except for the "trying on clothes" feature, which projects −$4000: an overall waste of $4000. The latter negative result is due to relatively low payoff, high development cost, and low reuse.

It may be advisable for the prototype to evolve into the application itself, but this should be planned for, not accidental. By their nature, prototypes are rapidly constructed and rarely documented. They can be implemented using languages that get results quickly but may be unsuitable for the application itself.

3.2.3.2 Feasibility Studies

It is sometimes uncertain whether proposed requirements can actually be implemented in practice. In other words, an entire project is at risk rather than a few specific requirements. In addition, the project would not be feasible if the risk were to be realized. In such cases, *feasibility studies* may be advisable. These are partial implementations or simulations of the application. For example, consider the feasibility of a Java Internet-based Encounter video game, and let's say we suspect performance will be so slow that the game would be of negligible interest to anyone. A feasibility study could consist of setting up a message-passing simulation at the anticipated rate from a number of players, but with dummy content. Delays could then be estimated by clocking the simulation.

Simulations can be expensive to create since they are applications in themselves, sometimes requiring software engineering artifacts of their own such as a requirements specification! The author was once involved with a simulation of a large system under development. The simulation grew into a large program, which was needed while engineers developed the real system. No one took the requirements for the simulation seriously because it was not "the real thing." As a result, even though the development community relied on the simulation, the cost of maintaining and using it became astronomical. Making changes required tracking down an employee who "knew the system." Feasibility simulations are common in large defense programs that involve extensive software and hardware.

3.2.4 Spiral Model

One of the earliest and best known iterative processes is Barry Boehm's Spiral Model [5]. It is called a spiral because Boehm conceptualized development iterating as an outward spiral, as shown in Figure 3.10.

Boehm's spiral model is a risk-driven process in which iterations have the specified goals shown in Figure 3.10. (Recall that risks are potential events or situations that can adversely affect the success of a project.) A project starts at the center, as it were, and each cycle of the spiral represents one iteration. The goal of each cycle is to increase the degree of system definition and implementation, while decreasing the degree of risk. Risk management is built into the process in that very early in each iteration, project risks are identified and analyzed and a plan for the iteration is created that includes mitigating some or all of the risks. As an example, suppose that at the beginning of a cycle a risk is identified with the screen layout of a portion of the user interface. Software could be developed to implement part of the user interface so feedback can be elicited from stakeholders. Doing this mitigates risk early in a project and leaves ample time to implement necessary changes. Thus the overall project risk reduces the further along you are in the process. After the major risks are addressed and mitigated, the project transitions to a waterfall model, as shown in the outer spiral of Figure 3.10.

Each iteration in Boehm's spiral model consists of the following steps:

1. Identification of critical objectives and constraints of the product.
2. Evaluation of project and process alternatives for achieving the objectives.

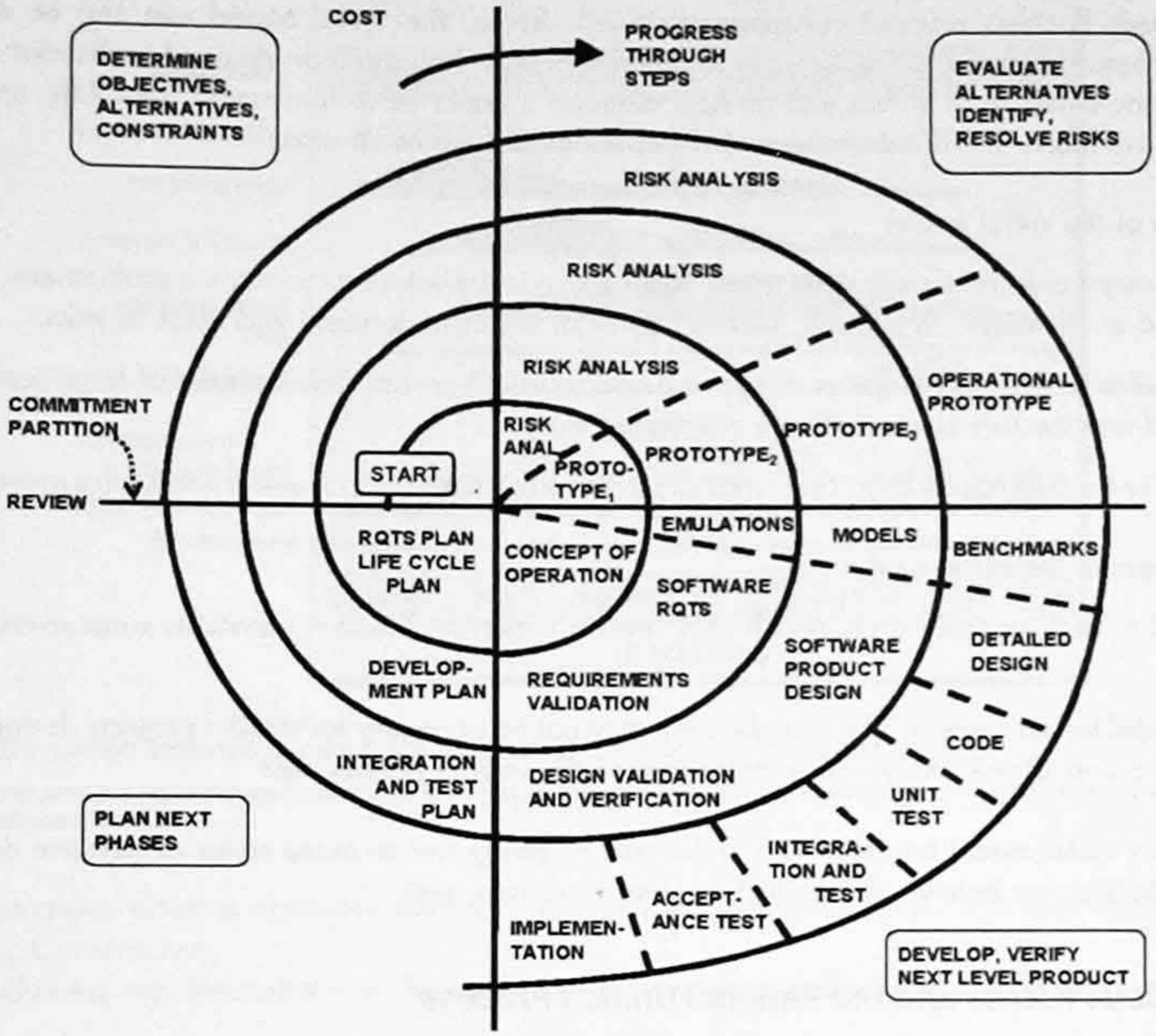


Figure 3.10 Boehm's spiral model for software development

Source: Boehm, B. W., "A Spiral Model of Software Development and Enhancement." IEEE Computer, Vol. 21, No. 5, May 1988, pp. 61–72.

3. Identification of risks.
4. Cost-effective resolution of a subset of risks using analysis, emulation, benchmarks, models, and prototypes.
5. Development of project deliverables including requirements, design, implementation, and testing.
6. Planning for next and future cycles—the overall project plan is updated, including schedule, cost, and number of remaining iterations.
7. Stakeholder review of iteration deliverables and their commitment to proceed based on their objectives being met.

Each traversal of the spiral results in incremental deliverables. Early iterations produce either models, emulations, benchmarks, or prototypes, and later iterations follow a more waterfall-like process and incrementally produce more complete versions of the software. This implies that early iterations may exclude step 5, and later iterations may exclude step 4. Although Figure 3.10 shows four iterations, the process does not prescribe a set number of cycles. The number is dictated by the size of a project, the number of risks identified, and their rate of retirement.

Although Boehm's original conception was risk-driven, the Spiral Model can also be driven by a sequence of functionality sets. For example, iteration 1 for an online video-on-demand application could be to implement the database of videos and its API; iteration 2 could be to implement the GUIs, and so on.

Key advantages and disadvantages of the spiral model are listed next.

Advantages of the spiral model

- *Risks are managed early and throughout the process*: Risks are reduced before they become problematic, as they are considered at all stages. As a result, stakeholders can better understand and react to risks.
- *Software evolves as the project progresses*: It is a realistic approach to the development of large-scale software. Errors and unattractive alternatives are eliminated early.
- *Planning is built into the process*: Each cycle includes a planning step to help monitor and keep a project on track.

Disadvantages of the spiral model

- *Complicated to use*: Risk analysis requires highly specific expertise. There is inevitably some overlap between iterations.
- *May be overkill for small projects*: The complication may not be necessary for smaller projects. It does not make sense if the cost of risk analysis is a major part of the overall project cost.

Boehm's spiral model has been very influential in giving rise to many styles of iterative development models, including, we believe, the unified process, discussed next.

3.2.5 Unified Process and the Rational Unified Process

The Unified Software Development Process (USDP) was first described by Jacobson, Booch, and Rumbaugh in 1999 [6] and is an outgrowth of earlier methodologies developed by these three authors—namely, Jacobson's "Objectory methodology," the "Booch methodology" [7], and Rumbaugh et al.'s Object Modeling Technique [8]. The USDP is generically referred to as the *Unified Process* (UP). IBM's Rational Software division has developed a detailed refinement to the UP called the *Rational Unified Process* (RUP), which is a commercial product providing a set of process guidelines and tools to help automate the process.

The UP is a "use-case driven, architecture-centric, iterative and incremental" software process [6]. Iterations are grouped into four "phases," shown on the horizontal axis of Figure 3.11: *Inception*, *Elaboration*, *Construction*, and *Transition*. The UP's use of the term "phase" is different from the common use of the term. In fact, referring to the figure, the term "discipline" is the same as the common use of "phase" that we use in this book.

Each UP "phase" consists of one or more iterations, shown across the bottom of Figure 3.11. Iterations are relatively short (e.g., three weeks), the outcome of which is a tested, integrated, and executable partial system. Iterations are built on the work of previous iterations, and thus the final product is constructed incrementally. Each iteration cycles through a set of nine disciplines, which are shown on the vertical axis of Figure 3.11: business modeling, requirements, design, implementation and test activities, plus supporting activities such as configuration management, project management, and environment [9]. For example, during an iteration some requirements may be chosen, the design enhanced to support those requirements, and the requirements implemented and tested. The horizontal "humps" next to each discipline show the relative effort expended on each discipline during iterations. For example, the

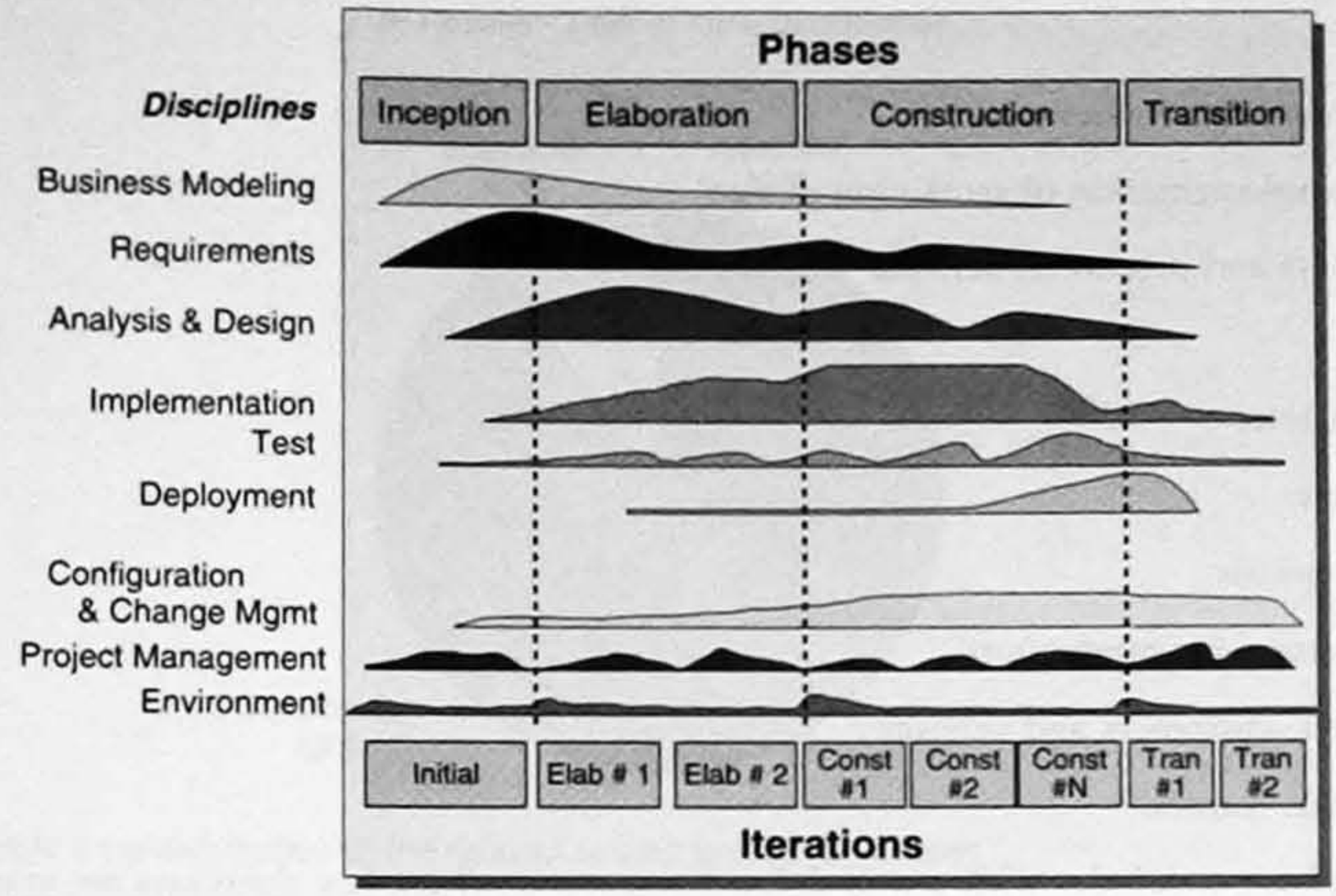


Figure 3.11 The unified software development process: its "phases" vs. traditional phases

Source: Adapted from Ambler, S. W., "A Manager's Introduction to the Rational Unified Process (RUP)." Ambysoft (December 4, 2005)., http://www.ambysoft.com/downloads/managersIntroToRUP.pdf.

largest requirements effort is expended during Inception and Elaboration, and the largest implementation effort during Construction.

The following are descriptions of the work conducted during each of the UP "phases":

Inception

- Establish feasibility
- Make business case
- Establish product vision and scope
- Estimate cost and schedule, including major milestones
- Assess critical risks
- Build one or more prototypes

Elaboration

- Specify requirements in greater detail
- Create architectural baseline
- Perform iterative implementation of core architecture
- Refine risk assessment and resolve highest risk items
- Define metrics
- Refine project plan, including detailed plan for beginning Construction iterations

Construction

- Complete remaining requirements
- Do iterative implementation of remaining design
- Thoroughly test and prepare system for deployment

Transition

- Conduct beta tests
- Correct defects
- Create user manuals
- Deliver the system for production
- Train end users, customers and support
- Conduct lessons learned

Each UP phase concludes with a well-defined milestone, where key decisions are made and specific goals defined. Figure 3.12 shows these milestones and where they are fulfilled in the process.

The typical amount of time spent in each UP phase is shown in Figure 3.13 [10]. However, this varies depending on the type of project. For example, projects that contain only minor enhancements with known requirements may spend more time in the Construction phase, while projects for which very little is known about requirements may spend more time in the Inception and Elaboration UP phases.

The advantages and disadvantages of the unified rocess are summarized below.

Advantages of the unified process

- *Most aspects of a project are accounted for*: The UP is very inclusive, covering most work related to a software development project such as establishing a business case.
- *The UP is mature*: The process has existed for several years and has been quite widely used.

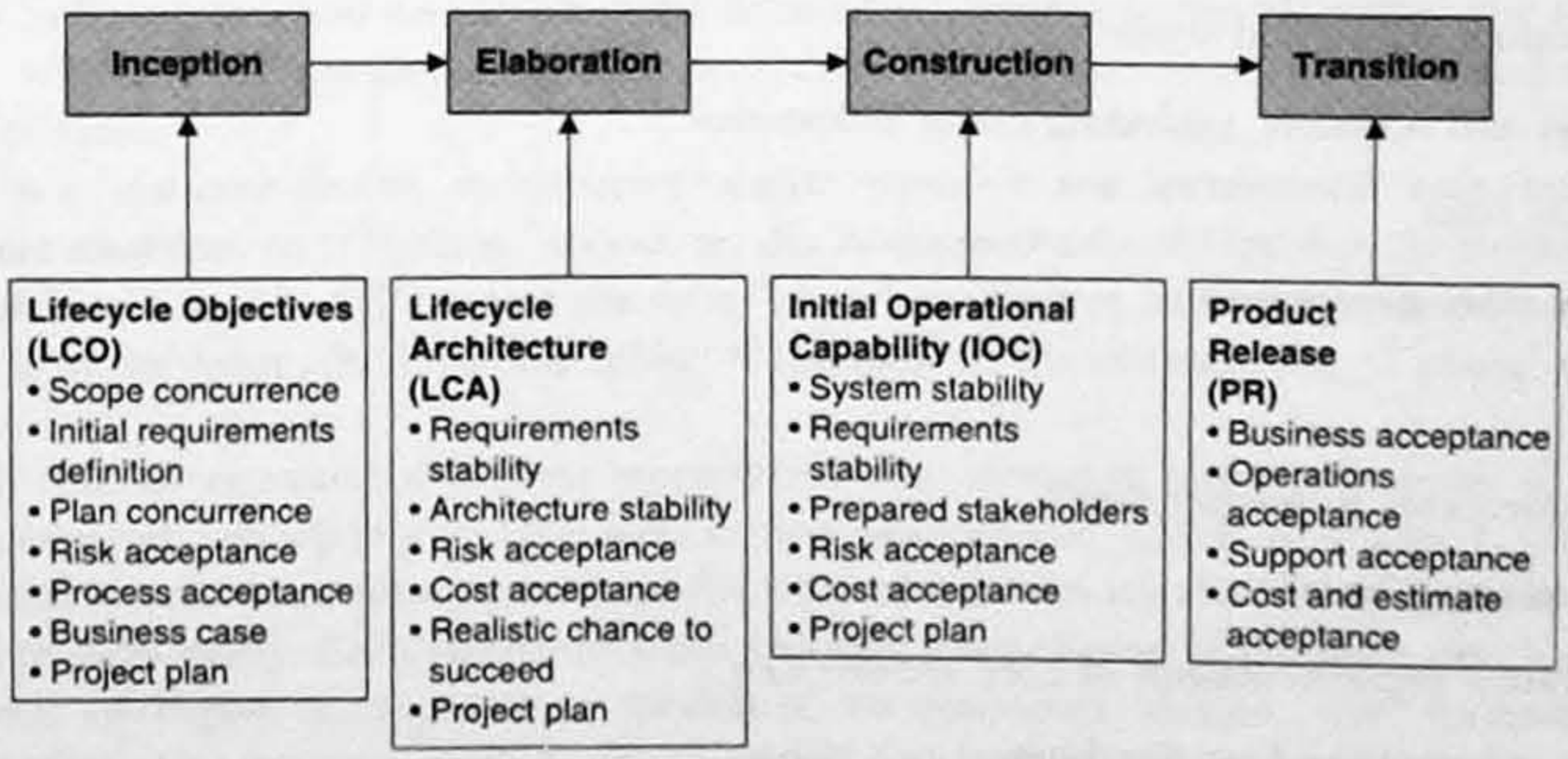


Figure 3.12 Objectives for the Unified process

Source: Adapted from Ambler, S. W., "A Manager's Introduction to the Rational Unified Process (RUP)." Ambysoft (December 4, 2005)., http://www.ambysoft.com/downloads/managersIntroToRUP.pdf.

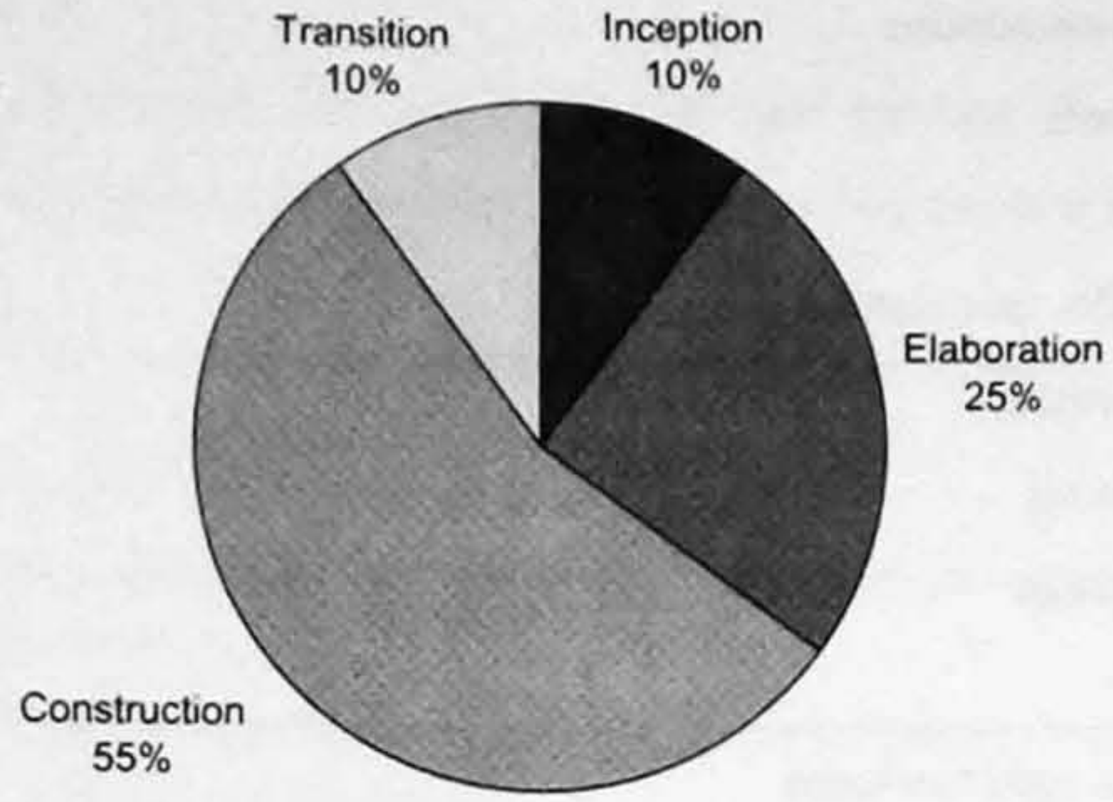


Figure 3.13 Typical time distribution of the rational unified process's "phases"

Source: Adapted from Ambler, S. W., "A Manager's Introduction to the Rational Unified Process (RUP)." Ambysoft (December 4, 2005)., http://www.ambysoft.com/downloads/managersIntroToRUP.pdf.

Disadvantages of the unified process

- *The UP was originally conceived of for large projects*: This is fine, except that many modern approaches perform work in small self-contained phases.
- *The process may be overkill for small projects*: The level of complication may not be necessary for smaller projects.

Practitioners and vendors of the unified process have modified it to be more like an agile process, discussed next.

3.2.6 Agile Processes

This section introduces agile processes. Chapter 4 is devoted entirely to agile methods, and agility is referenced and compared throughout this book.

In 2001, a group of industry experts, disillusioned with some commonly held software engineering beliefs and practices, met to discuss ways to improve software development. Their goal was to produce a set of values and principles to help speed up development and effectively respond to change. The group called themselves the Agile Alliance, in essence to capture their goal or producing a methodology that was efficient and adaptable. The result of their work was the *Manifesto for Agile Software Development* [11], also known as the *Agile Manifesto*, which contains the values and principles they defined. An *agile* software process is one that embraces and conforms to the Agile Manifesto, which is summarized in Figure 3.14.

Agile processes are highly iterative and incremental. They commonly employ the following:

- Small, close-knit teams
- Regular, frequent, disciplined customer requirements meetings
- A code-centric approach, documentation on an as-needed basis (e.g., high-level requirements statements only)

Agile processes 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

Figure 3.14 Main points of the Agile Manifesto

- Customer representatives working within the team
- The use of user stories as the basis for requirements—end-to-end accounts of how users need to accomplish individual tasks
- Refactoring, a kind of disciplined code improvement explained full in Chapter 24
- Pair programming, in which two programmers work at a single workstation
- Continual unit-testing, and acceptance tests as means of setting customer expectations

Figure 3.15 shows how the repeated iterations of an agile process are executed over time. "Story" is a task required of the application as the customer conceives it.

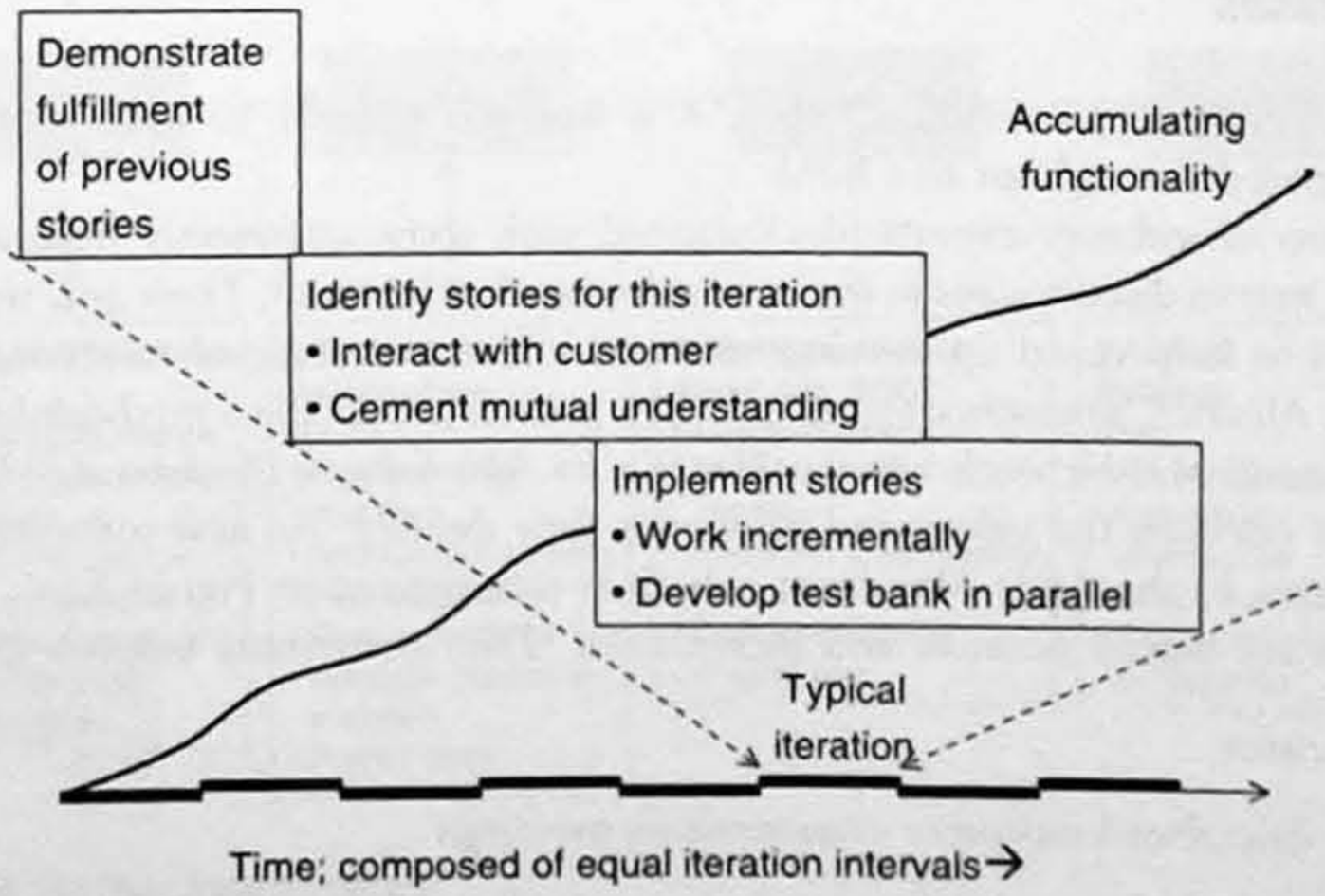


Figure 3.15 The agile process

Key advantages and disadvantages of agile processes are summarized below.

Advantages of an agile process

- *The project always has demonstrable results*: The end product of each iteration is working software.
- *Developers tend to be more motivated*: Developers prefer to produce working artifacts and tend not to like creating documentation.
- *Customers are able to provide better requirements because they can see the evolving product.*

Disadvantages of an agile process

- *Problematical for large application*: Agile methods are more readily used for smaller projects. There is debate about their utility for large projects.
- *Documentation output is questionable*: Since documentation takes second place, there is a question as to whether necessary documentation will ever be produced.

3.2.7 Open-Source Processes

Open-source software is developed and maintained by people on a volunteer basis. All project artifacts, from requirements documents through source code, are available to anyone. There are several open-source software development Web sites on the Internet that aggregate and reference open source projects, including SourceForge.net and Freshmeat.net. As of 2009, SourceForge.net hosted more than 100,000 projects and had over 1,000,000 registered users. Hosting sites provide source code repositories and utilities for project management, issues discussion, and source control. The case studies in this book illustrate two well known open source projects: Eclipse and OpenOffice.

Open-source projects typically get started when someone develops an application and posts it to a host Web site. Virtually anyone can propose new requirements, and since the source code is freely available, anyone can implement requirements that are not part of the official baseline. There is a process by which proposals and implementations are elevated in priority and a process for accepting new code and capability into the baseline. If defects are found, they are reported and others may work on fixing them. This process is repeated, and the application grows in capability and stability. In this way, open-source projects are developed in an iterative manner. In addition, by exercising an application many times, the open-source community affects a huge testing process.

Some reasons why an individual or company makes a project open source are listed in Figures 3.16 and 3.17, and reasons why they may not are listed in Figure 3.18. A primary reason to make a project open source is to leverage a large number of resources that might not otherwise be available. A principal reason to not make software open source is to keep artifacts hidden from current and prospective competitors. Some (e.g., Ferguson [12] and Kapor) believe that open source may become the dominant process for producing software.

An interesting article written by Alan Joch illustrates how the bank Dresdner Kleinwort Wasserstein (DrKW) turned its internally developed back-end Java integration tool, OpenAdapter, into open source and how it benefited from the decision. The following is an excerpt from that article.

Samolades et al. [13] studied open-source development and compared it with closed-source software. Using common metrics, OSS code quality was found roughly equal to that of CSS, and found to be superior for maintainability, or at worst equal. Some of their results are shown in Figures 3.19 and 3.20.

Open-source users find rewards in collaborative development

By Alan Joch
2/1/2005 http://www.adtmag.com/article.aspx?id=10544

Banks pride themselves on playing things close to the vest. So, when Dresdner Kleinwort Wasserstein's IT group shared the code of OpenAdapter, an internally developed tool, with the development community, it caused a sensation. "It was astonishing," recalls Steve Howe, DrKW's global head of open-source initiatives. "They found it difficult to believe a bank was open sourcing something."

OpenAdapter, a back-end Java integration tool that helps integrate bank apps with little or no custom programming, had become "a famous piece of software within DrKW," Howe says. "Half of the decision to open source it into the wider financial community was to try to replicate that enthusiasm."

DrKW's motive for releasing the tool was hardly altruistic. By releasing the tool to the wider financial community, the investment bank hoped to benefit from the bug fixes and refinements other programmers in Europe and North America might make.

However, Howe and other open-source experts warn that corporations' code-sharing projects require a number of safeguards to protect intellectual property and keep companies from becoming unwitting victims or distributors of destructive code. Some companies believe such risks outweigh the potential benefits and resist open-source business applications. DrKW and other firms believe careful open-source collaboration gives them a competitive advantage.

. . .

Banking on open source

Dresdner Kleinwort Wasserstein (DrKW) believes it found development success with OpenAdapter by creating a variation on the collaborative model the open-source community pioneered. . . . . "We open sourced the tool within DrKW and told people that if they needed to change OpenAdapter slightly, they were free to do that," Howe says. "A lot of developers decided they could make a name for themselves by contributing to the software." Today, more than 100 applications in the bank use the integration tool.

To tap similar expertise outside DrKW, the bank turned to CollabNet, a commercial project-development platform from CollabNet in Brisbane, California. CollabNet provides a number of services, including version control tools and discussion forums common in free open-source clearing-houses such as SourceForge. . . . CollabNet helped the bank establish a separate legal entity called the Software Conservancy, which owns the intellectual rights to OpenAdapter.

The site publishes the latest stable build of the software, which anyone can download. Users can also send in complaints and error reports and suggest bug fixes. So far, DrKW has received input from developers at competing banks and from companies outside the financial industry. Collaboration consists of technical subjects, not information that involves trade secrets or confidential customer data.

Developers interested in becoming more involved in OpenAdapter's evolution can gain access to the source code by signing a legal agreement that assigns the intellectual property rights of their code to the Software Conservancy. About 20 developers now hold such status.

Companies mix and match open-source and proprietary processes according to business needs. Figure 3.21 suggests that the proportion of open-source or proprietary software used is a business decision, taken in the context of expenses and revenues over time. An important factor is the expense of tailoring and maintaining open source.

- ☺ Leveraging large number of resources
- ☺ Professional satisfaction
- ☺ To enable tailoring and integration
- ☺ Academic and research
- ☺ To gain extensive testing
- ☺ To maintain more stably

Figure 3.16 Some reasons *for* making a project open source, 1 of 2

- ☺ To damage the prospects of a competitor's product
- ☺ To gain market knowledge
- ☺ To support a core business
- ☺ To support services

Figure 3.17 Some reasons *for* making a project open source, 2 of 2

- ☹ Documentation inconsistent or poor
- ☹ No guarantee that developers will appear
- ☹ No management control
- ☹ No control over requirements
- ☹ Visibility to competitors

Figure 3.18 Some reasons *against* making a project open source

Project Mnemonic Code	Application Type	Total Code Size (KLOCs)	No, of releases measured	Project Evolution Path
OSSPrA	Operating system application	343	13	OSS project that gave birth to a CSS project while still evolving as OSS
CSSPrA	Operating system application	994	13	CSS project initiated from an OSS project and evolved as a commercial counterpart of OSSPrA

Figure 3.19 Maintainability index comparing OSS and CSS for the same application, 1 of 2

Source: Samoladas, Ioannis, I. Stamelos, L. Angelis, and A. Oikonomou. "Open source software development should strive for even greater code maintainability." Communications of the ACM, Vol. 47, No. 10, October 2004, copyright © 2004. Association for Computing Machinery, Inc. Reprinted by permission.

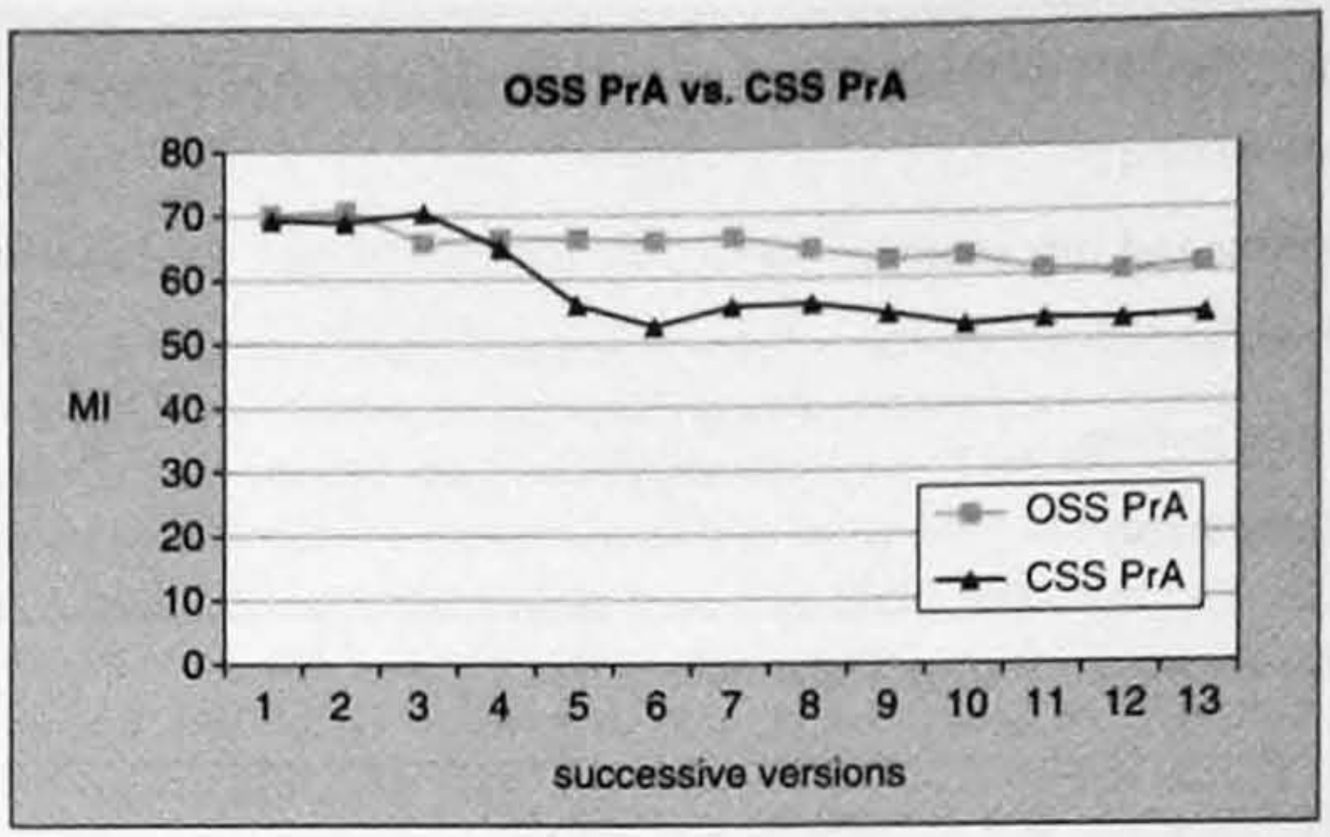


Figure 3.20 Maintainability index comparing OSS and CSS for the same application, 2 of 2

Source: Samoladas, Ioannis, I. Stamelos, L. Angelis, and A. Oikonomou. "Open source software development should strive for even greater code maintainability." Communications of the ACM, Vol. 47, No. 10, October 2004, copyright © 2004. Association for Computing Machinery, Inc. Reprinted by permission.

Various tools and environments are available to facilitate open-source development. Figure 3.22 shows a schematic diagram of one such tool, Collabnet. "Subversion" is an open-source version control system used for many open-source projects.

Key advantages and disadvantages of open-source processes are summarized below.

Advantages of open source

- *The work of many may be obtained free:* For projects that motivate others, work can be obtained from motivated outsiders.
- *Developers tend to be more motivated,* because they choose to work on it.
- *Open-source applications are very well tested,* because they are continually exercised by many and usually have efficient testing processes.

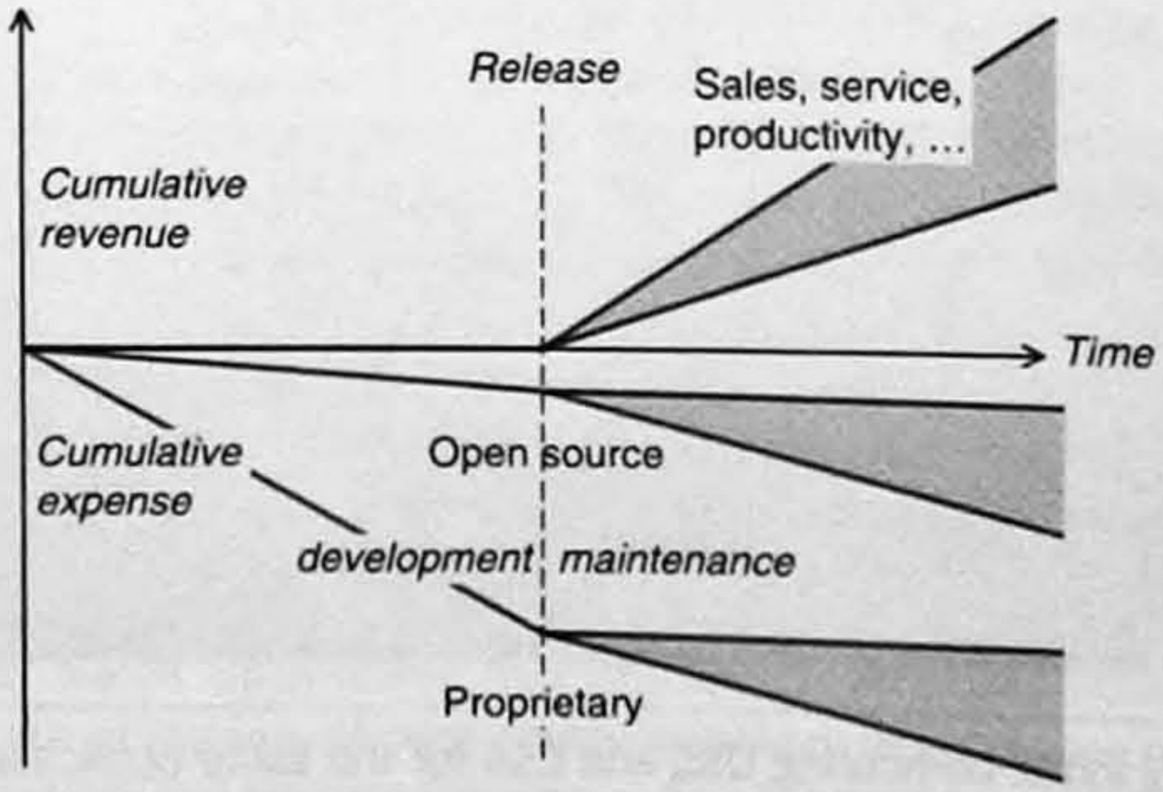


Figure 3.21 Hybrid open-source/proprietary processes

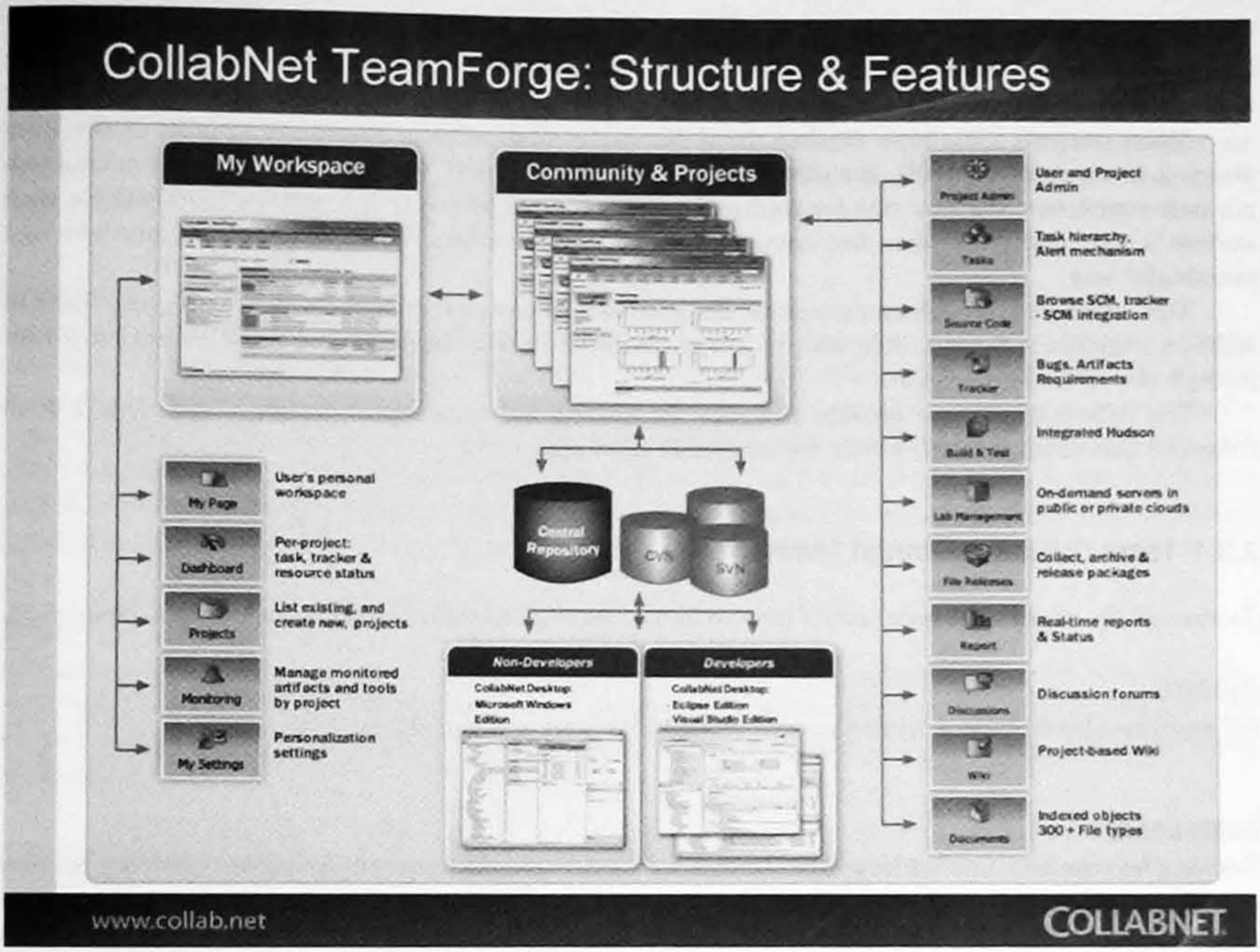


Figure 3.22 CollabNet TeamForge: Structure and Features

Source: CollabNet, http://www.open.collab.net/products/sfee/capabilities.html.

Disadvantages of open source

- *They are available to one and all*: This is part of the bargain.
- *Documentation is questionable*: Developers are not consistent in what they produce
- *Design documentation tends to be poor*: It seems that the open-source process has especially great trouble keeping designs and code synchronized, and so design documents are often poor or even nonexistent.

To reinforce the concepts on processes described in this chapter, we continue with case studies.

3.3 CASE STUDY: STUDENT TEAM GUIDANCE

To reinforce the many software engineering concepts and practices introduced in this book, it's best if you are developing a software project in parallel. You will gain the most if the project is a team effort, as this is how most real-world software projects are executed. At the end of most major parts of the book, a section (such as this one) entitled *Team Guidance* can be found, guiding you through the different phases of your group project.

You will be asked to generate specific artifacts, and examples will be provided showing how real and hypothetical teams go about developing applications. We will often use the Encounter video game case study as an example to illustrate team guidance.

Many students look back on their team process as a significant learning adventure. If the team avoids a few common pitfalls, the adventure can be a most enlightening and useful experience. From our past experience, the best size for student groups is four or five. Any less and the workload for each student is too great. More than five doesn't afford all team members the opportunity to contribute in a meaningful way.

Tips distributed throughout the text are designed to help groups maximize the benefit of group work. In addition, exercises at the end of the chapter assign specific artifacts to be developed as the project progresses through the software life cycle.

The following sections provide guidance for holding an initial team meeting, developing a team communication plan, and exercising the communication plan.

3.3.1 Team Guidance—Initial Team Meeting

To start off the project, the team should have an initial meeting and make the decisions shown in Figure 3.23.

Agenda

All meetings should have a written agenda and specific start and end times.

Team Leader

Decide who your first team leader will be (being a team leader provides you with valuable experience but puts additional demands on your time and communication skills). To distribute the benefit of team leadership practice, it can be beneficial to swap team leadership about halfway through the project. Both team leaders can be chosen up front, and they can back each other up in case of emergency.

---

1. Set **agenda** and time limits.
2. Choose the team **leader**.
3. Get everyone's **commitment** to required time.
    - Define an expected average number of hours per week.
    - Gather dates of planned absences.
4. Take a realistic census of **team skills**.
    - Common problem: inflated programming skill claims.
5. Begin forming a **vision** of the application.
6. Decide how team will **communicate**.
7. Take **meeting minutes** with concrete action items.

Figure 3.23 Initial student team meeting—general issues

Time Commitment

A big source of frustration in student teams is a lack of commitment of some team members. It is far better to discuss commitments at the beginning than to try to fix a problem after the project is under way. To reduce the resentment that follows when some team members feel they are contributing much more than others, set in advance an expected number of hours of commitment per week. This also helps to make members work more efficiently.

Team Skills

For student projects there is often a trade-off between producing an impressive-looking product and learning new skills. To produce the most impressive product, team members would specialize along the lines of their greatest strengths. This is a typical industrial mode. They may also be tempted to skimp on documentation in order to demonstrate more features. To learn the most, however, team members need to try activities with which they are inexperienced (e.g., team leadership). They also need to do things right. Teams decide how and when to trade off between goals by specifying when they will specialize and when they will try new roles. Instructors try to establish evaluation criteria that encourage learning.

Vision

All projects start out with a vision for the software to be developed. In industry this is typically initiated by the marketing department, which develops business requirements for the product. For student projects, the product vision includes the purpose of the application, major functionality and operations, and major inputs and outputs.

Communication Plan

Decisions need to be made as early as possible as to how the team will handle communication. The next section covers this in detail.

Meeting Minutes

During team meetings, a member is designated to write the meeting minutes. The purpose of meeting minutes is twofold. First, all important decisions or agreements that are reached are recorded, so they are not forgotten and can be referred back to if necessary. As an example, during the meeting it may be decided that Google Docs will be used for storing all project specifications. This decision is recorded in the meeting minutes. The second purpose of meeting minutes is to record unresolved issues that require follow up action. These are referred to as *action items*. Each action item is assigned to one or more team members with a target date for completion. An example action item might be that Joe is to investigate and recommend a source control tool for managing the project's source code, and is to complete his action in one week. It is a good idea to review open action items at the start of each team meeting.

3.3.2 Team Guidance—Communication Plan

When a software project is initiated in industry, a set of project guidelines, tools, and communication procedures is typically established to ensure that the project is executed as efficiently as possible. This should also be done for your group project.

Many team problems arise from a failure to communicate fully. This problem is not limited to student projects. Real-world teams suffer from it as well. Effective verbal communication means making sure your thoughts are fully understood and listening to what others are saying. This is critical for teams to be

1. Listen to all with concentration
   - Don't pre-judge.
2. Give all team members a turn
   - See the value in every idea.
3. Don't make assumptions.
   - Ask questions to clarify.
4. When in doubt, communicate

Figure 3.24 Key precepts of communication

successful. The precepts shown in Figure 3.24 will help you avoid many communication problems. They sound simple but can be hard to follow, especially during times of stress.

Create *policies* for communicating and sharing information with each other, including guidelines for group editing and merging of documents, setting up and agreeing to meeting times, sharing project artifacts, and so on.

Decide on procedures for how you will generate documents for the project. You probably have to deal with discussing the scope and contents of a document, writing initial drafts, editing the drafts, getting group agreement on edits, merging drafts and producing the final document.

Many teams—including some student teams—are widely distributed geographically, and the means of communication becomes especially important. Large projects are often developed by multiple groups at multiple sites, sometimes in multiple countries. Mergers, acquisitions, "offshoring," dispersed specialists, and joint ventures often result in people at multiple sites working together on projects.

An increasing number of products are available to facilitate group work, including groupware, video conferencing, and instant messaging. Each communication medium has specific strengths. In any case, well-run face-to-face meetings are very hard to beat. If possible, schedule regular face-to-face meetings at least once a week for an hour. It is hard to convene an unscheduled meeting but easy to cancel a scheduled meeting. You can make it a goal to limit actual meetings to less time. However, if the committed time is short—say a half hour —you will find it very difficult to make longer meetings because people will build the short time limit into their schedules. If you think that your team may need to meet additionally, it is advisable to agree on when that would be each week. Set aside the time and aim to avoid a meeting. This provides a positive goal and it avoids the problem of repeatedly trying to find a common time when meetings are needed. When you set meeting times, specify an end time. Meetings typically expand to fill the allotted time. You should always have an agenda for your meetings, otherwise your meetings will be less organized and not as productive as they could be.

E-mail is an essential tool, but can be problematic due to unpredictable delays. Messages can become unsynchronized, damaging the threads of dialogs. This is especially serious near the end of a project when communication is frequent and of immediate importance.

Use a shared Web site, wiki, or chat-type facility. For example, at the time of this writing, a number of free collaboration tools are available from Google.

Do not merely state that you will use a particular tool such as Microsoft Word for word processing. Specify a version number, and exchange a few documents to be sure everyone can read and edit them. Don't change versions during the project without ensuring compatibility first.

Document your decisions in a team *Communication Plan*, as outlined in Figure 3.25.

1. **Meetings**: Team will meet each Monday from . . . to . . . in . . .
   Caveat: do not replace face-to-face meeting with remote meetings unless remote meetings are clearly effective.
2. **Meeting alternative**: Team members should keep Fridays open from . . . to . . . in case an additional meeting is required.
3. **Standards**: Word processor, spreadsheet, compiler, ...
4. **E-mail**: Post e-mails?; require acknowledgement?
   Caveat: e-mail is poor for intensive collaboration
5. **Collaboration**: Tools for group collaboration and discussion —
   e.g. Yahoo Groups, Wiki tool, Google tools, . . .
6. **Other tools: Microsoft Project (scheduling), Group calendar, . . .**

Figure 3.25 Communication planning—forms of communication

3.3.3 Team Guidance—Test Communication Plan

It is important to test the methods specified in your Communication Plan so you can make necessary adjustments as early as possible in the project. This will avoid scrambling for alternatives as the project workload increases.

Search the Web for the latest information on a topic determined by the instructor. Note at least four of its basic goals and at least five of the techniques it uses. Have everyone in the group contribute some information. Create a document containing your group's results, and practice how you will utilize your procedures for group editing and reviews. How will you organize this random activity? How can you obtain a useful result instead of a conglomeration of unconnected text?

3.4 SUMMARY

A software project progresses through a series of activities, starting at its conception and continuing all the way through its release. Projects are organized into phases, each with a set of activities conducted during that phase. A *software process* prescribes the interrelationship among the phases by expressing their order and frequency, as well as defining the deliverables of the project. Specific software processes are called *software process models* or *life cycle models*.

Most software process models prescribe a similar set of phases and activities. The difference between models is the order and frequency of the phases. The phases include planning, requirements analysis, design, implementation, testing, and maintenance.

The *waterfall* process is one of the oldest and best known software process models. Projects following the waterfall process progress sequentially through a series of phases. Work transitions to a phase when work on the previous phase is completed.

Iterative and *incremental* processes are characterized by repeated execution of the waterfall phases, in whole or in part, resulting in a refinement of the requirements, design, and implementation. At the conclusion of an iteration, operational code is produced that supports a subset of the final product's functionality. Project artifacts such as plans, specifications, and code evolve during each phase and over the life of the project. Examples of iterative processes include the spiral model, the unified process, and agile processes.

Agile processes are highly iterative, and emphasize working code throughout, as well as frequent interactions with the customer.

A *prototype* is a partial implementation of the target application useful in identifying and retiring risky parts of a project. It can sometimes be a way to obtain ideas about the customer's requirements. An increase in understanding what is to come can save expensive rework and remove future roadblocks before they occur. Many iterative process models incorporate prototypes in one or more of their iterations.

Sometimes, it is unclear whether certain requirements can be implemented in practice, placing the entire project at risk. In such cases, *feasibility studies* may be advisable, which are partial implementations or simulations of the application.

In open-source processes, software is developed and maintained by people on a volunteer basis. Source code is open to all, and there is a process for virtually anyone to suggest and implement enhancements and submit and repair defects. This process is repeated and the application grows in capability and stability. In this way, open-source projects are developed in an iterative manner. In addition, by exercising an application many times, the open-course community functions as a huge testing process.

3.5 EXERCISES

1. During which process phase(s) would each of the following activities occur?
   a. Creating a project schedule
   b. Determining the need for a bar code reader
   c. Requesting the addition of a file backup capability
   d. Performing a feasibility analysis
   e. Documenting the software interface to an SQL database
   f. Acceptance of the software application by the customer
2. Give an example of a software project that would benefit much more from using the waterfall process than from using most of the alternative processes. Explain your reasoning.
3. Describe the difference between *iterative* and *incremental* development. Describe the ways in which they are related.
4. Give an example of a software project that would benefit more from using an iterative and incremental process than from using most of the alternative processes. Explain your reasoning.
5. a. In your own words, explain how the spiral model utilizes risk analysis and risk mitigation.
   b. Explain why the outer spiral of the spiral model utilizes the waterfall process, and how the spiral model mitigates the inherent disadvantages of the waterfall process.
6. Give an example of a software project that would benefit much more from using the spiral process than from using most of the alternative processes. Write a paragraph explaining your answer.
7. How do the phases of the unified process (UP) differ from the phases usually defined for software processes?
8. Describe the pros and cons of each value listed in the Agile Manifesto (see Figure 3.14).

TEAM EXERCISES

Communication

For the following exercises, consider as a group how you will perform them, check the hints below, then carry out the assignments.

T1. Decide who your team leader(s) will be. Note that being team leader provides you with practice that may be hard to get otherwise.

T2. Decide how your team will communicate, specify your communication tools and methods, and test your communication methods. Be specific: you may change the specifics later.

T3. Search the Web for the latest information on a topic determined by the instructor (e.g., the TSP). Note at least four of its basic goals and at least five of the techniques it uses. Post the references to the course forum or Web site if there is one, and annotate your posting with the name of your group. State individual or group opinions of the topic or issue.

Your team response should be 4–7 pages long.

Hints for Team Exercises

T1 hints: To distribute the benefits of team leadership practice, it can be beneficial to swap team leadership about halfway through the semester. Both team leaders can be chosen up front, and they can back each other up in case of emergency. Such backing up is a good practice in any case, because the probability of a team leader having to quit a project or a class can be high. Note that the second half of a project typically requires the team leader to make decisions more quickly than the first half.

T2 hints: Examples are telephone, meetings, e-mail, forums, chat facilities, and Web sites.

1. Schedule regular face-to-face meetings at least once a week, if possible. It is hard to convene an unscheduled meeting but easy to cancel a scheduled one.
2. E-mail is an essential tool, but can be problematic due to unpredictable delays. Messages can become unsynchronized, damaging the threads (subjects) of dialogs. This is especially serious near the end of a project when communication is frequent and of immediate importance.
3. Use a shared Web site or chat-type facility. Free services are available at http://groups.yahoo.com/, for example.
4. Do not merely state, "We will use Superword for word processing." Specify a version number, and exchange a few messages to be sure. Don't change versions during the project without ensuring compatibility first.
5. Try out all the standards and methods you have chosen.

Throughout this program of study, validate your plans and intentions with practical tests whenever possible. Try to use at least two independent tests. In general, assume that your project will be much more demanding in the future than it is at the beginning.

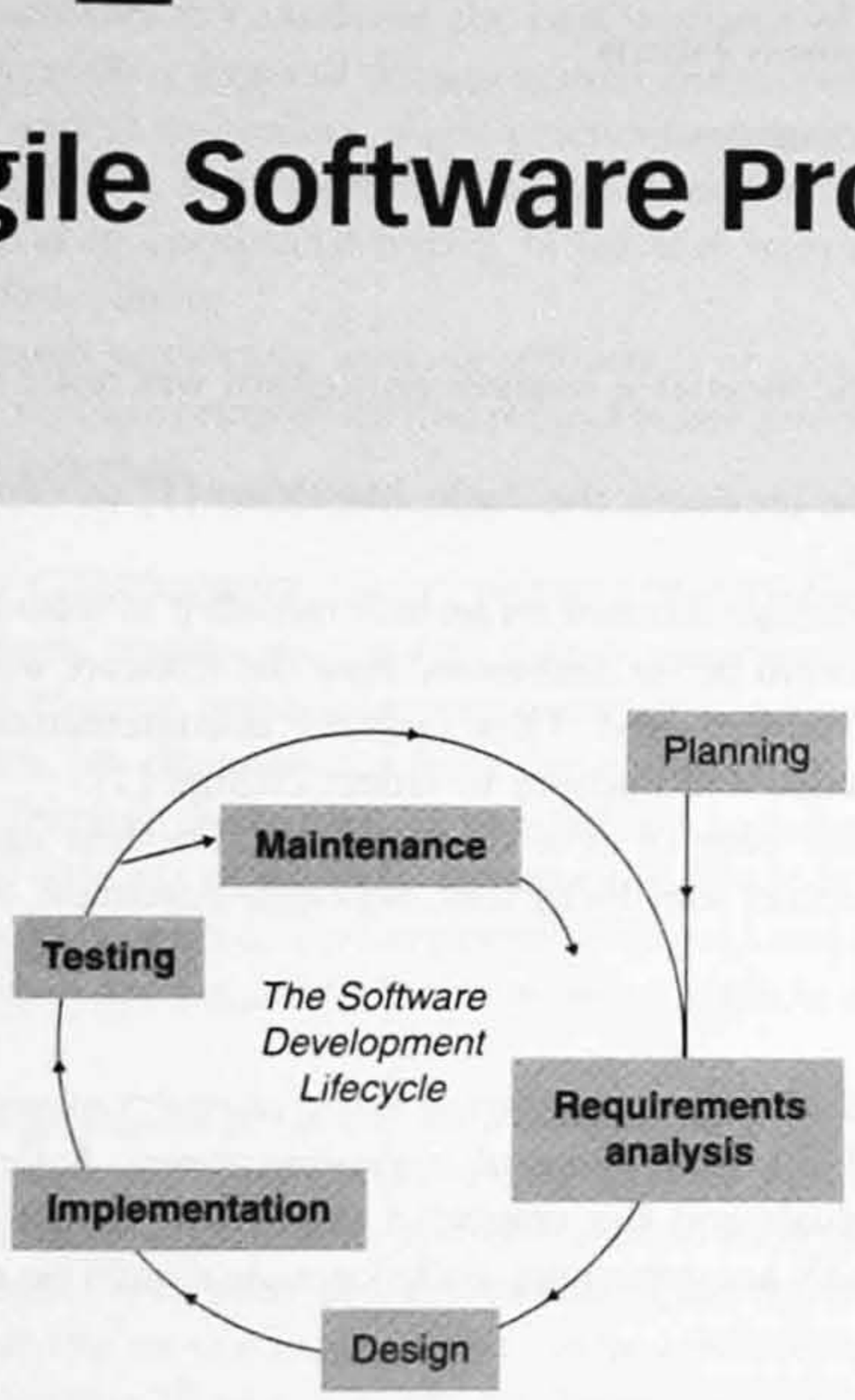
T3 hints: Use this activity to stress your communication system. For example, you may want to set up a Web site to which team members are to add new TSP information. How will you organize this random activity? How can you obtain a useful result instead of a conglomeration of unconnected text?

BIBLIOGRAPHY

1. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," *IEEE WESCON 1970*, August 1970, pp. 1–9.
2. Bittner, Kurt, and Spence, I., 2007, *"Managing Iterative Software Development Projects,"* Addison-Wesley, Pearson Education, Inc.
3. Cockburn, Alistair. "Unraveling Incremental Development," January 1993. http://alistair.cockburn.us/Unraveling+incremental+development [accessed November 5, 2009].
4. Cusumano, Michael, and R. W. Selby. "How Microsoft Builds Software." *Communications of the ACM*, Vol. 40, No. 6(1997), pp.53–61.
5. Boehm, B. W. "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, Vol. 21, No. 5 (May1988), pp.61–72.
6. Jacobson, Ivar, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*, Addison-Wesley, 1999.
7. Booch, Grady. *Object-Oriented Analysis and Design with Applications*, Addison-Wesley,1994.
8. Rumbaugh, James, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, *"Object-Oriented Modeling and Design,"* Prentice Hall, 1990.
9. Larman, Craig, *"Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,"* Prentice Hall, 2005.
10. Ambler, S. W., *"A Manager's Introduction to the Rational Unified Process (RUP)."* Ambysoft (December 4,2005). http://www.ambysoft.com/unifiedprocess/rupIntroduction.html [accessed November 5, 2009].
11. Beck, Kent, Mike Beedle, Arie van Bennekum, and Alistair Cockburn, "Manifesto for Agile Software Development," Feb 2001. http://agilemanifesto.org/ [accessed November 5, 2009].
12. Ferguson, Charles, "How Linux Could Overthrow Microsoft," *Technology Review* (June2005). http://www.technologyreview.com/computing/14504/ [accessed November 5, 2009].
13. Samoladas, Ioannis, I. Stamelos, L. Angelis, and A. Oikonomou, "Open source software development should strive for even greater code maintainability." *Communications of the ACM*, Vol. 47, No. 10, October 2004.

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 →	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
RESPONSES:				
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.

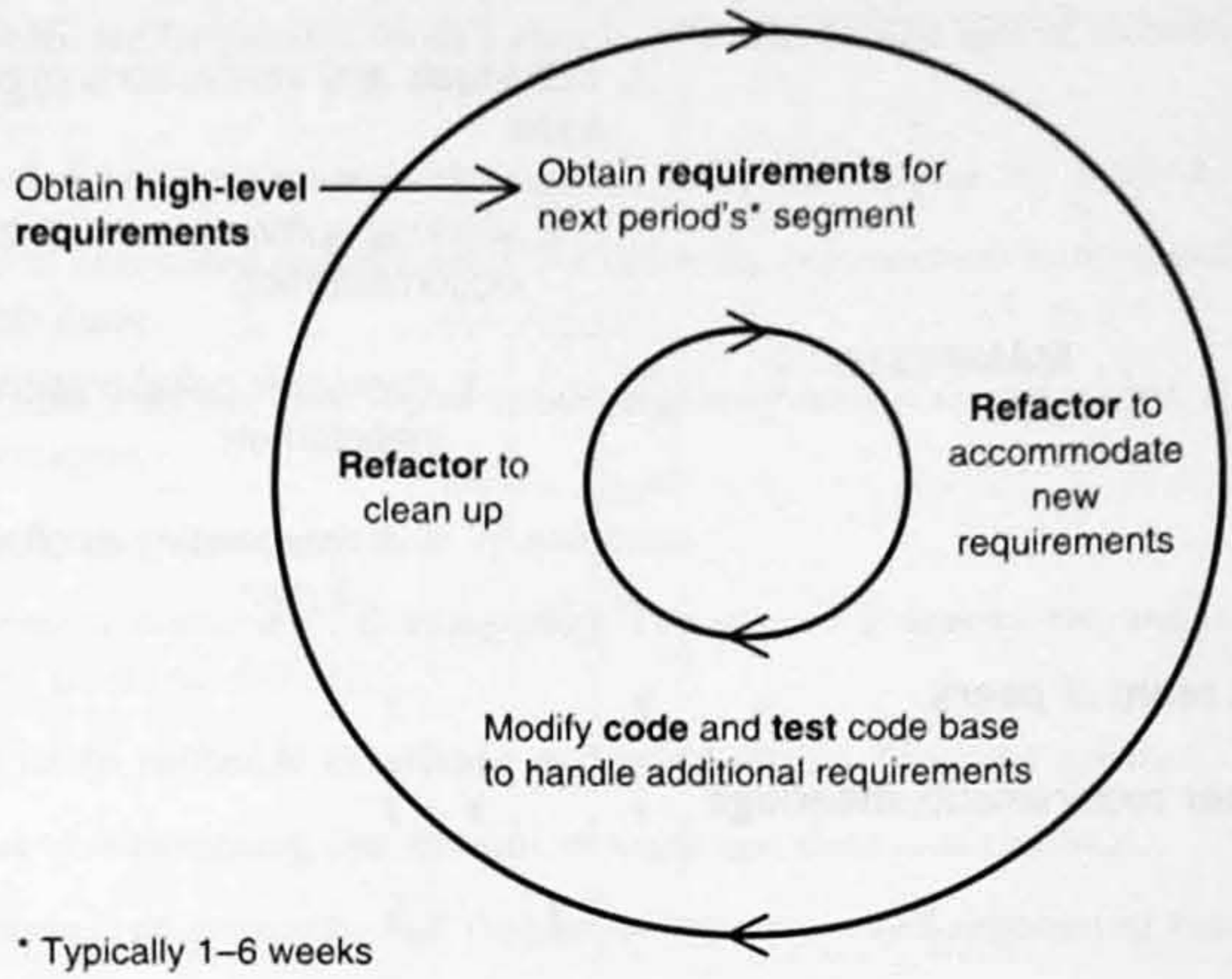


Figure 4.4 A typical agile development iteration

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

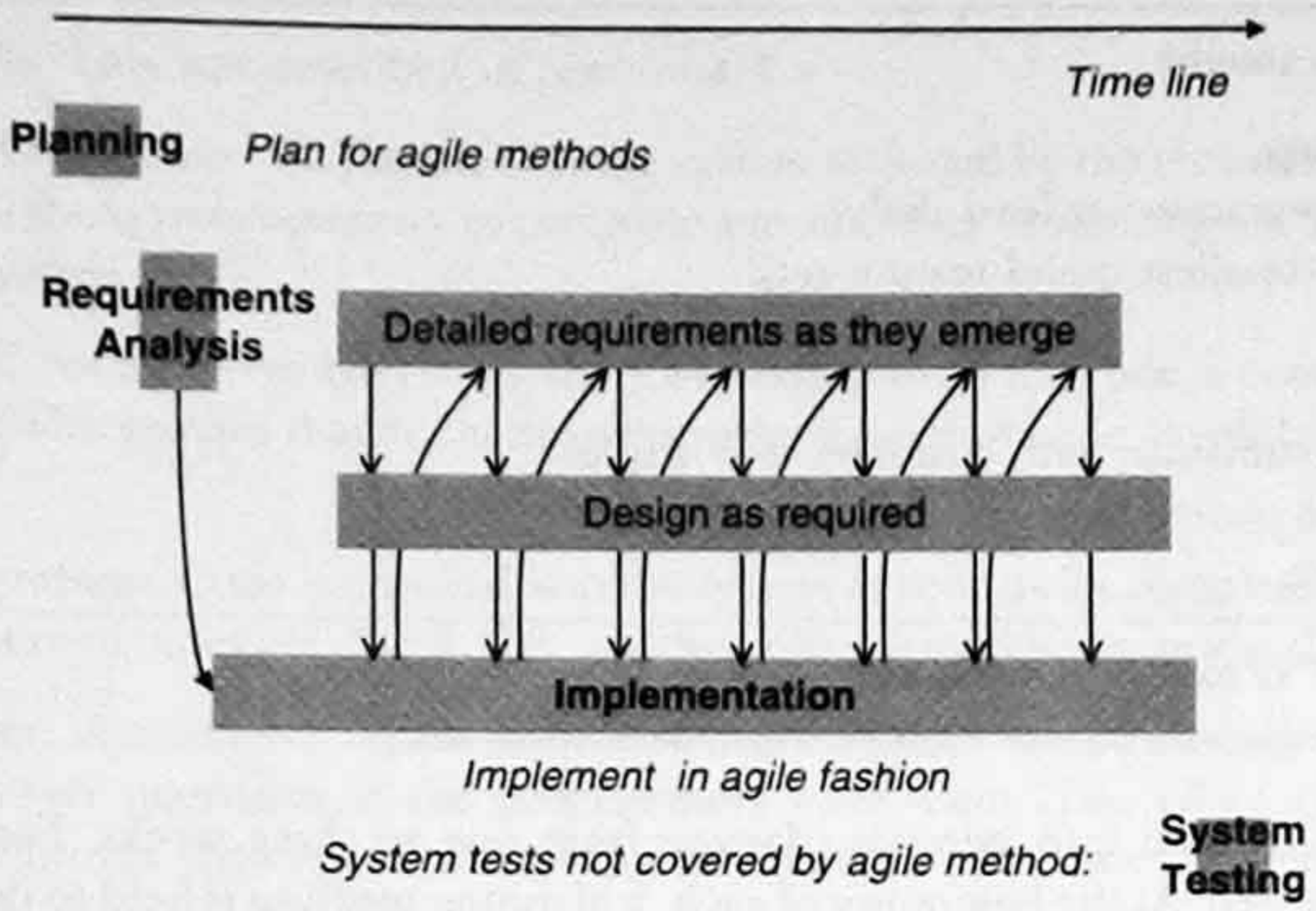


Figure 4.5 The agile schedule

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

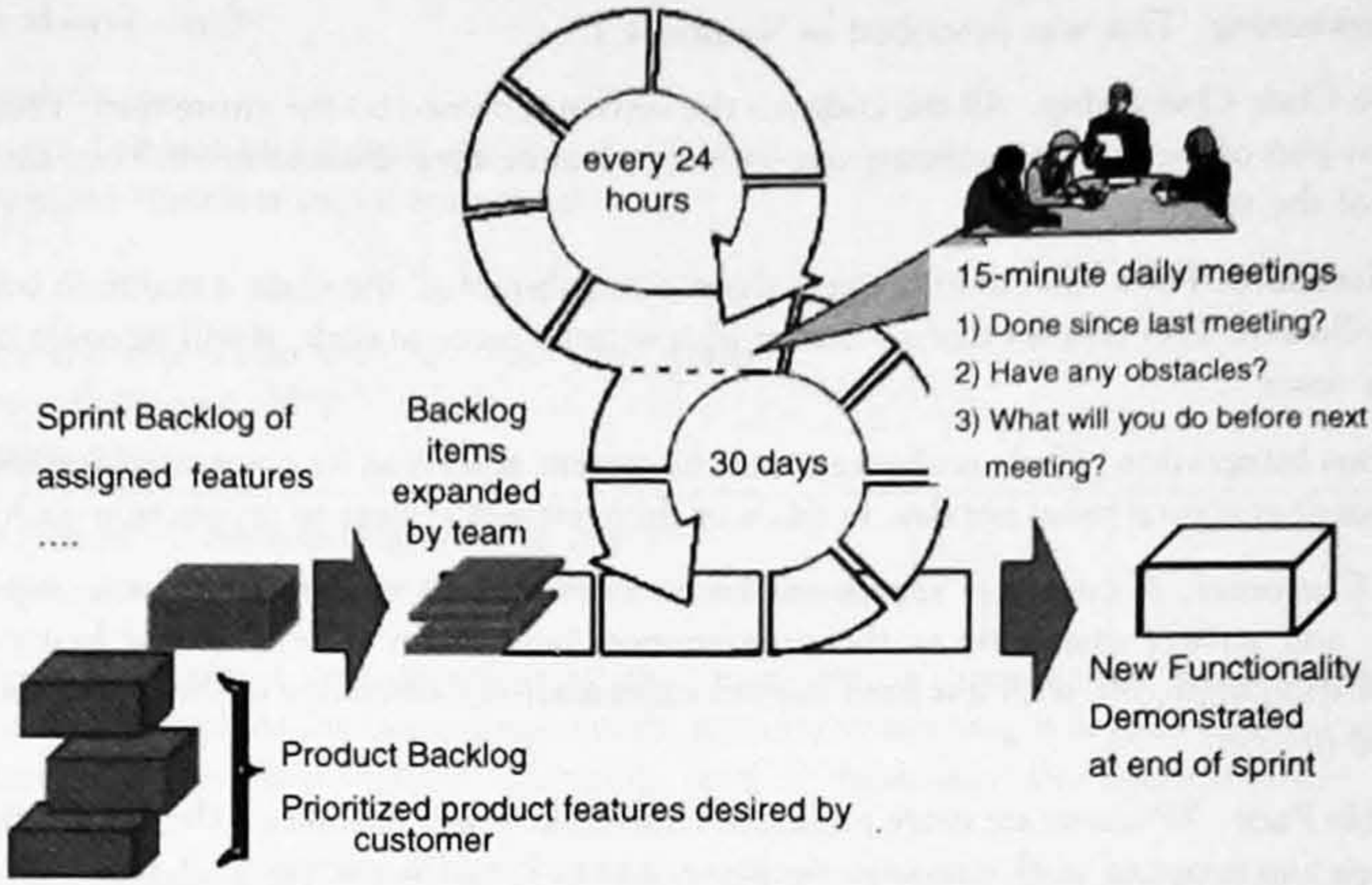


Figure 4.8 The scrum work flow

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

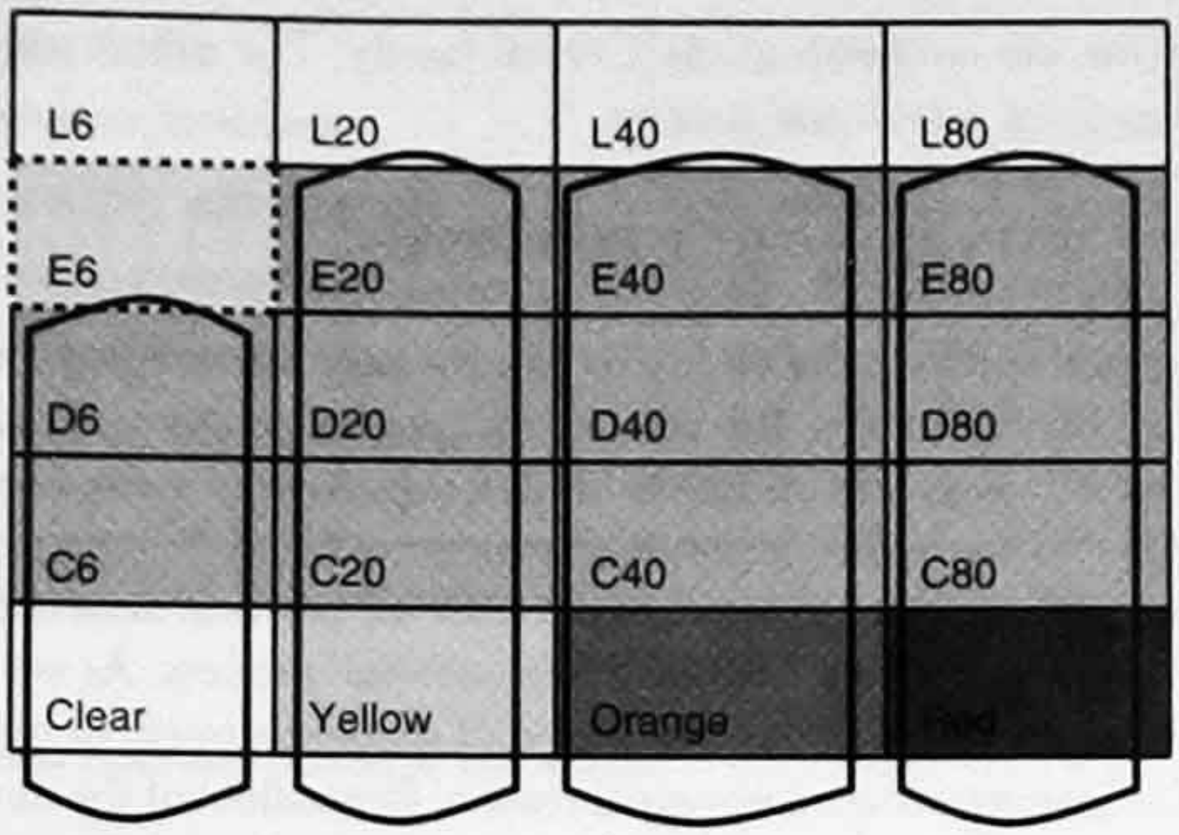


Figure 4.9 Coverage of various Crystal methodologies

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

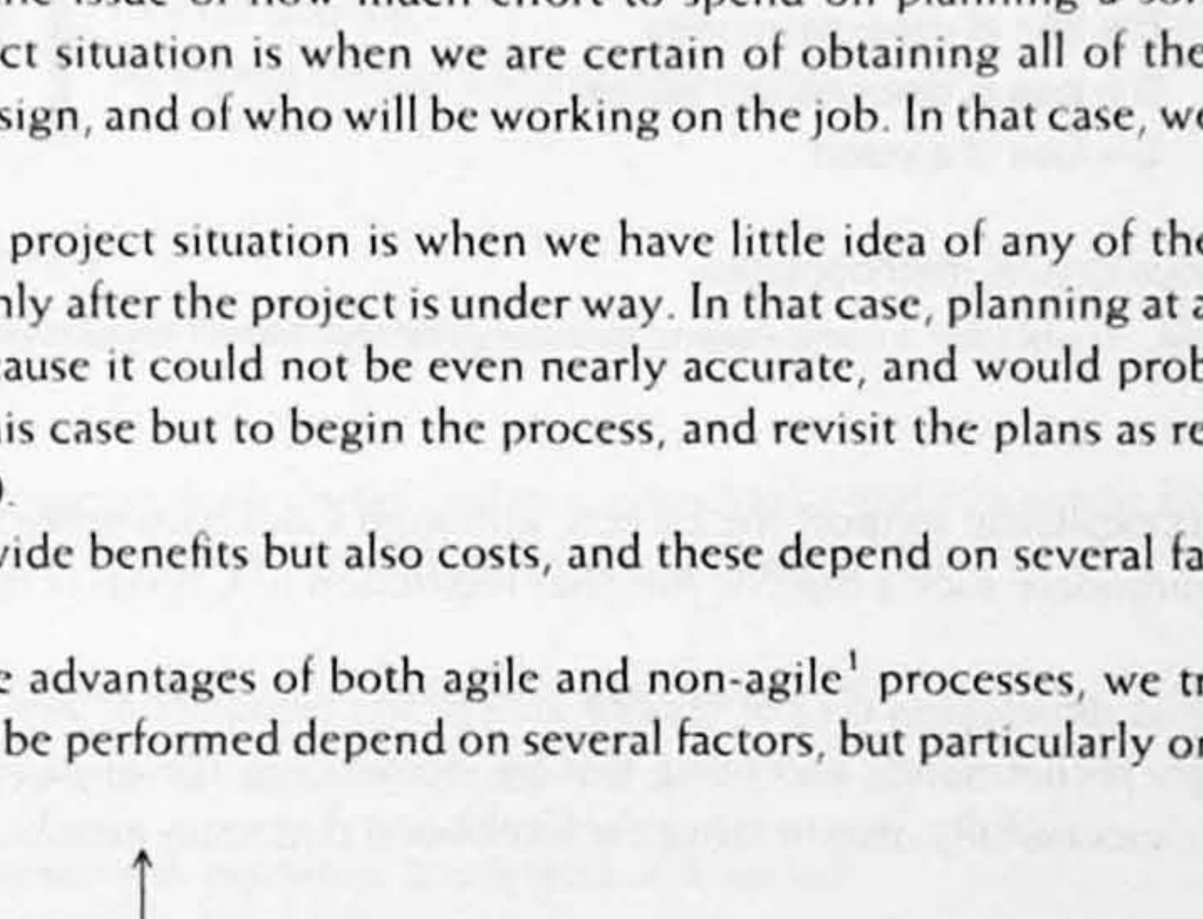


Figure 4.10 How detailed should plans be?

[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."

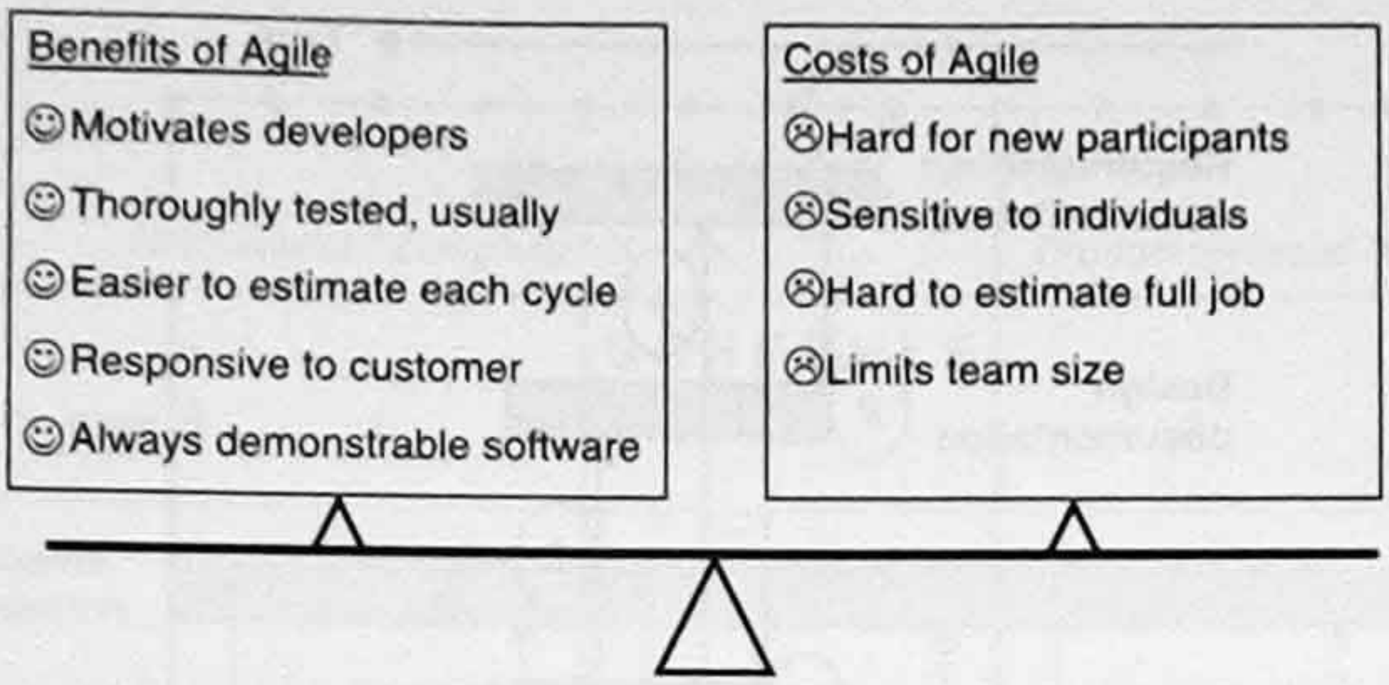


Figure 4.11 Trade-offs between agile and non-agile processes

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

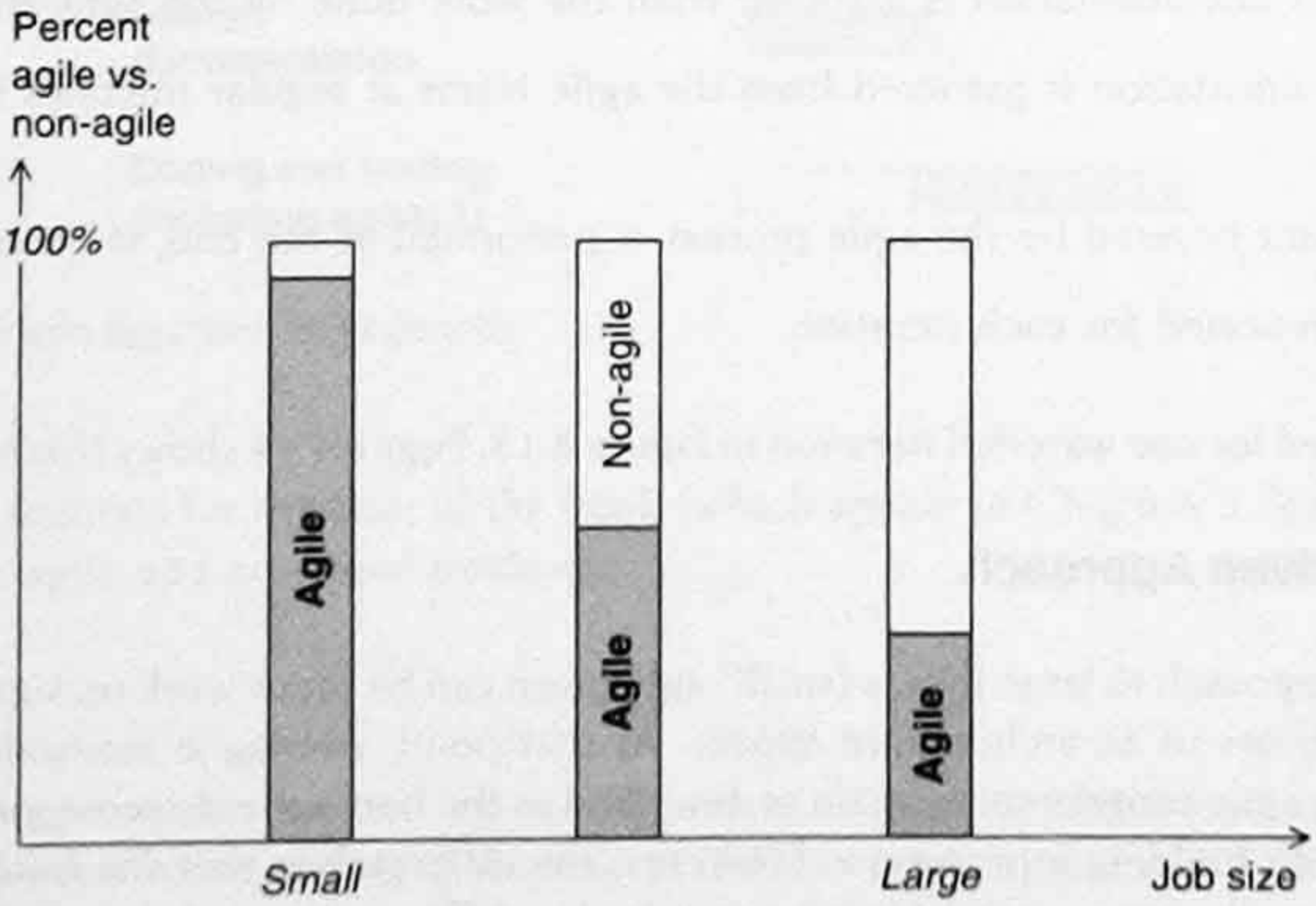


Figure 4.12 Conceptual agile/non-agile combination options: some conventional wisdom, circa 2009

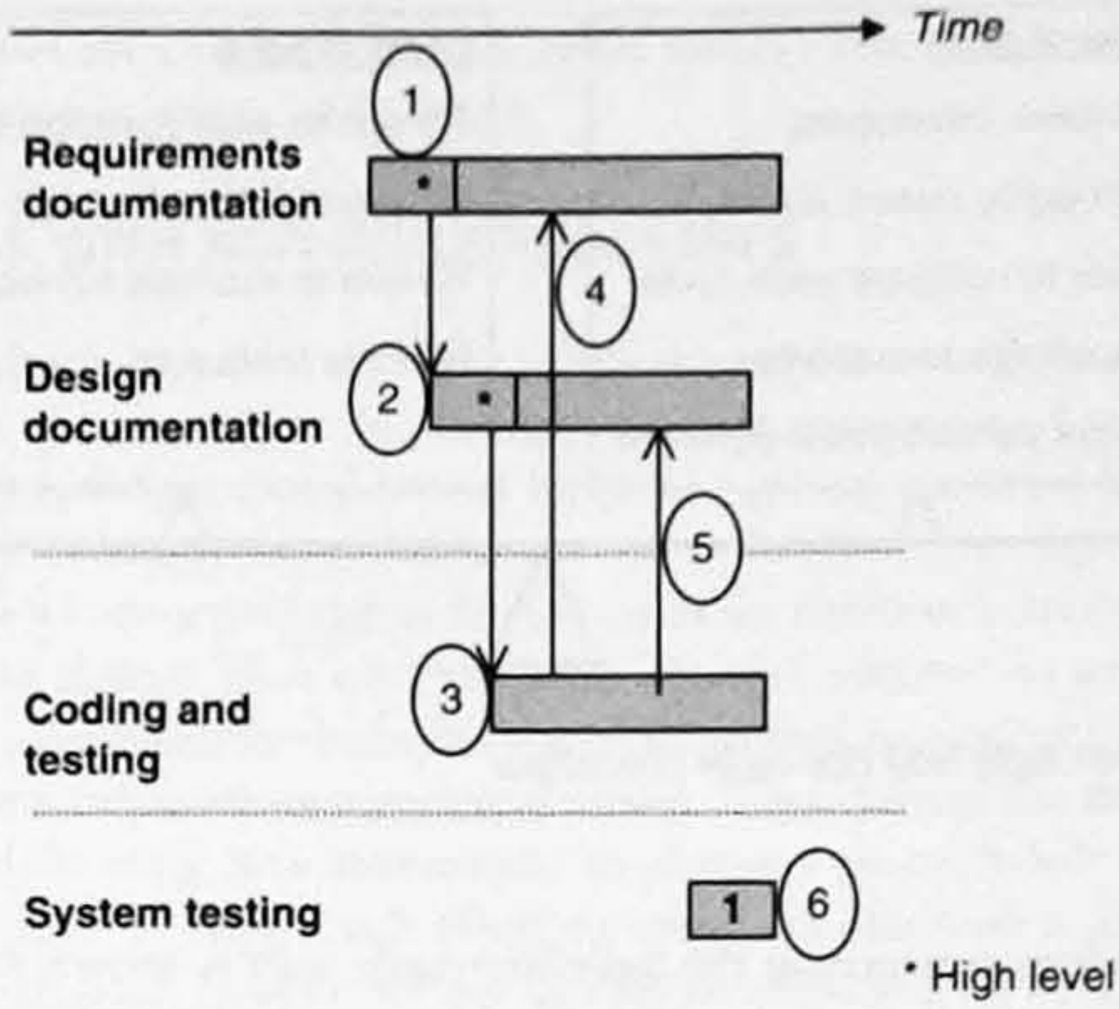


Figure 4.13 Integrating agile with non-agile methods 1: time line for a single iteration

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.

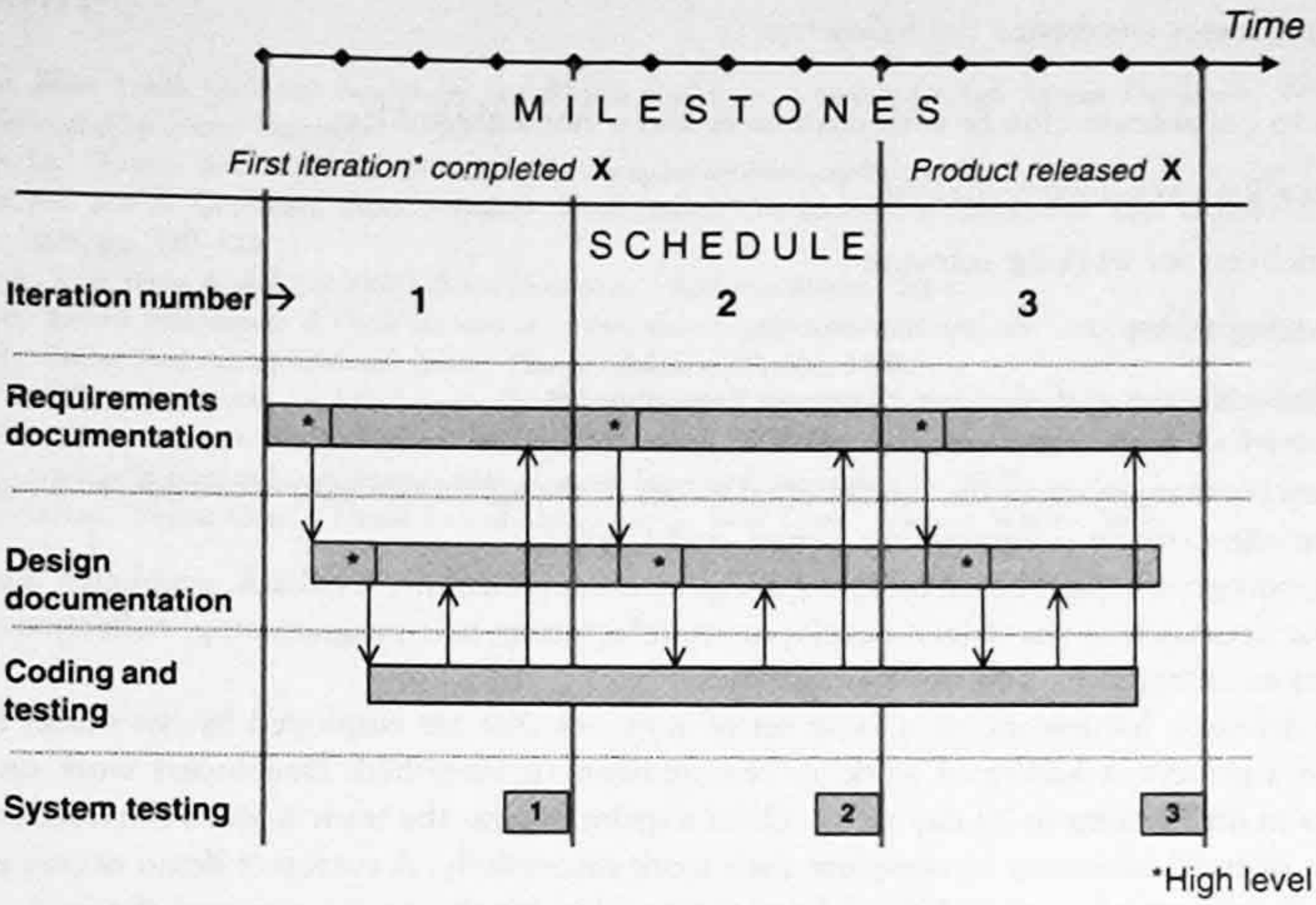


Figure 4.14 Integrating agile with non-agile methods 2: time line for multiple iterations

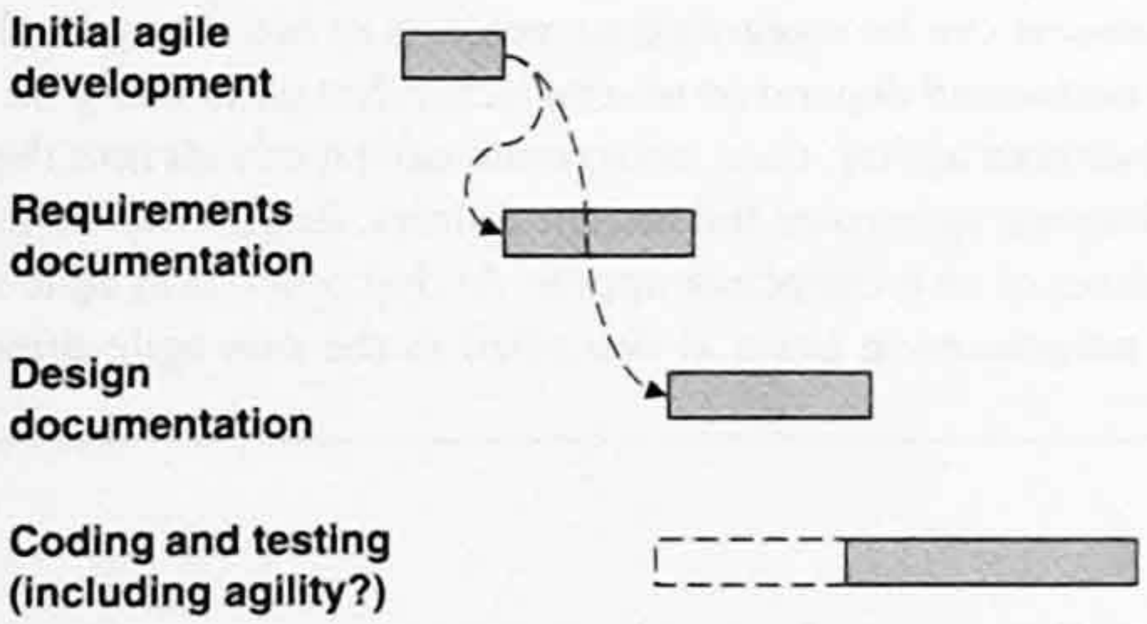


Figure 4.15 An agile-driven approach to large jobs

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.