2 Chapter

Software Process and Methodology

Key Takeaway Points

- A software process defines the phases of activities or what need to be performed to construct a software system.
- A software methodology details the steps or how to perform the activities of a software process. A methodology is an implementation of a process.
- Software development needs a software process and a methodology.

Writing programs that consist of a few thousand lines of code might be an unprecedented experience and a challenge for many undergraduate and graduate students. However, in the software industry, software development for systems that consist of millions of lines of code is a common practice. The difference between academic programming projects and real-world software development projects is not limited to the number of lines of code. Systems development in the real world has to overcome many other challenges. To overcome these challenges, a disciplined approach to systems development is required. This is commonly called a *software process*. The waterfall process presented in Chapter 1 is such a process or *process model*. Many software development organizations adopt the waterfall process partly due to its simplicity. However, experiences show that the waterfall process is associated with a number of problems. To overcome these problems, many other software process models have been proposed. The software industry then discovered that software development requires not only a process but also a methodology. While a process specifies the activities to be performed to develop a software system, a methodology defines the steps and how to perform the steps to carry out the activities of a process. In other words, a methodology is an implementation of a process.

In summary, this chapter presents the following:

- Challenges of system development in the real world.
- Merits and limitations of the waterfall process.
- Other well-known software process models.

- The theory of wicked problems and how it relates to software development.
- Agile processes and agile methods.
- An overview of the process and methodology presented in this book.

2.1 CHALLENGES OF SYSTEM DEVELOPMENT

Developing software systems in the lab is quite different than developing them in the real world. Real-world projects are much larger and much more complex. Besides these obvious challenges, there are others. An understanding of some of the challenges is useful for the study of software engineering. That is, it helps to understand why computer science alone is not enough, and why we need software processes and methodologies. Furthermore, it helps us to know how software processes and methodologies help overcome the challenges. First, there are project challenges:

- **Project Reality 1.** Many system development projects have long development durations, which typically range from one year to several years. Real-world projects must meet schedule and budget constraints.
- **Project Challenge 1.** How do we plan, schedule, and manage the project work without a complete knowledge of what the customer wants, and what will happen in the next several years?
- **Project Reality 2.** Many system development projects require collaboration of multiple departments or development teams. For example, many large embedded systems involve both hardware and software components and require the software engineering department to cooperate with hardware departments.
- **Project Challenge 2.** How do we divide the work and assign the interdependent pieces to the departments and teams, and be able to smoothly integrate the components produced by the different departments and teams?
- **Project Reality 3.** Different departments or teams may use different development processes, methods, and tools. The departments or teams may be located at different places.
- **Project Challenge 3.** How do we ensure proper communication and coordination among the departments and teams?

Besides project challenges, there are product challenges. Some examples are:

- **System Reality 1.** Many real-world systems have to satisfy a large number of requirements and constraints including stringent real-time constraints. For example, a mail-handling system has hundreds of requirements and constraints. It is required to scan and process more than 40,000 mail-pieces per hour or 12 mail-pieces per second.
- **System Challenge 1.** How do we develop the system to ensure that these requirements and constraints are met?
- **System Reality 2.** Requirements and constraints may change from time to time.

requirements had to change every week during the first three months due to change requests from the customer.

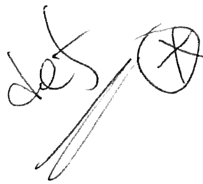
- **System Challenge 2.** How do we design the processes and the products to cope with changes that are often inevitable and the exact changes are impossible to predict?
- **System Reality 3.** The system will evolve for many many years, creating a maintenance challenge. For example, changes are not documented or poorly documented, changes to the system may introduce bugs as well as deteriorate the structure of the system. All these make the system more and more difficult to understand, test, and maintain.
- **System Challenge 3.** How do we design the system so that changes can be made relatively easily and without much impact to the rest of the system?

- **System Reality 4.** The system may consist of hardware and software components, use third party components and multiple implementation languages, and run on multiple platforms and machines located at different places.
- **System Challenge 4.** How do we design the system to hide the hardware, platform, and implementation peculiarities so that changes to these will not affect the rest of the system?

The list is not complete. There are many more challenges in the real world that are not included. However, the list is enough to show that a scientific approach is not adequate to tackle the challenges. We need an engineering approach. Two important components are software process and software methodology.

2.2 SOFTWARE PROCESS

Advances in computer science, especially nonnumeric computation and relational database systems in the 1960s, led to a rapid expansion of computer applications in the business sector. The existing ad hoc development processes could not satisfy the needs of such development projects. The notion of a software development process or software process was introduced and defined as follows:



Definition 2.1 A *software process* is a series of phases of activities performed to construct a software system. Each phase produces some artifacts which are the input to other phases. Each phase has a set of entrance criteria and a set of exit criteria.

For example, the waterfall process, presented in Chapter 1, exhibits a strict sequence of development activities. The requirements phase produces the requirements specification, the design phase produces the design, and so on. Frequently, the entrance

2.3 MERITS AND PROBLEMS OF THE WATERFALL PROCESS

The conventional waterfall process defines a straight sequence of activities such as requirements analysis, design, implementation, testing, and maintenance. The process has been used by many companies for many decades. Before discussing its problems, it would be fair to point out that the process does have a number of merits. These merits explain why the process is still in use in the software industry. First, the simple, straight sequence of phases of the waterfall process greatly simplifies project planning, project scheduling, and project status tracking. In this sense, it is deemed a "predictable" process. Second, the straight sequence of functional activities allows function-oriented project organization. In such an organization, the functional teams are specialized in different functional areas such as requirements analysis, design, and implementation. Projects are carried out by the functional teams in a pipeline manner. The third merit of the waterfall process is that the phased approach is appropriate for some large, complex, long-lasting, embedded systems. Examples of such systems include mail-sorting and routing systems, process control systems, and many other types of computerized equipment. Such systems need to respond to numerous hardware-generated events, process huge amounts of incoming data, and control the behaviors of hardware devices. Usually, the capabilities or requirements of such systems are jointly defined by the equipment manufacturer and the customer. In many cases, the system to be developed is merely a major enhancement of the functionality, performance, and security of an existing system. The vendor has experience and good knowledge of the application and what the customer wants; and hence, major changes to the requirements are rare. Once the purchase order is issued, the customer does not have access to the equipment until the acceptance testing stage, although the customer participates in reviews and prototype demonstrations. The phased approach allows each phase to be performed rigorously to ensure that the system runs reliably and satisfies performance and timing constraints.

The waterfall process has a number of serious drawbacks. First, the one strict sequence of phases and related milestones makes it difficult to respond to requirements change. However, change to requirements is needed in many circumstances, such as increased business competition, new regulations or standards, or inability to identify all requirements. For many applications, the long development duration is unacceptable because the requirements were identified long ago and business needs have changed dramatically. Third, the users cannot experiment with the system to provide feedback until it is released. Experiences show that early user feedback helps in detecting misconception of business needs and problems in user interface design. Another drawback is that the customer cannot reap the benefits of the new system during the long development period. Finally, the customer has to accept the risk of low return on investment if the project is canceled; that is, the design documents and partly tested code are not worth the investment.

2.4 SOFTWARE DEVELOPMENT IS A WICKED PROBLEM

Properties of a Tame Problem	Properties of a Wicked Problem
1. A tame problem can be completely specified.	1. A wicked problem does not have a definite formulation.
2. For a tame problem, the specification and the solution can be separated.	2. For a wicked problem, the specification is the solution and vice versa.
3. For tame problems, there are stopping rules.	3. There is no stopping rule for a wicked problem—you can always do it better.
4. A solution to a tame problem can be evaluated in terms of correct or wrong.	4. Solutions to wicked problems can only be evaluated in terms of good or bad, and the judgment is subjective.
5. Each step of the problem-solving process has a finite number of possible moves.	5. Each step of the problem-solving process has an infinite number of choices—everything goes as a matter of principle.
6. There is a definite chain of cause-and-effect reasoning.	6. Cause-and-effect reasoning is premise-based, leading to varying actions, but hard to tell which one is the best.
7. The solution can be tested immediately; once tested, it remains correct forever.	7. The solution cannot be tested immediately and is subject to lifelong testing.
8. The solution can be adapted for solving similar problems.	8. Every wicked problem is unique.
9. The solution process is a scientific process.	9. The solution process is a political process.
10. If the problem is not solved, simply try again.	10. The problem solver has no right to be wrong because the consequence is disastrous.

FIGURE 2.1 Properties of tame and wicked problems

of California, Berkeley. A wicked problem exhibits a number of wicked-problem properties or wicked properties for short. These properties imply that wicked problems are difficult to solve. In contrast, tame problems such as solving mathematical equation systems, developing chess-playing programs, compiler construction, and operations research exhibit nice properties. Figure 2.1 compares the properties of these two types of problem.

Unfortunately, *application software development in general is a wicked problem.* For a tame problem such as solving an equation system, the problem has a definite formulation. The specification and the solution can be separated. The equation system is the specification; an assignment of values to the variables is a solution. This is not true for many application software development projects. For example, the requirements specification may not specify the real requirements. This is why many projects fail because the delivered system does not meet users' expectations. Prototypes are often constructed to help identify real requirements. In this case, the prototype is both the specification and the implementation because it not only specifies the features but also how to implement the features.

The number of steps to solving an equation system is finite. Each step has only a finite number of possible moves. This is not the case for software design—the number of possible design alternatives is limited only by the knowledge and creativity of the designer. The process to solving an equation system stops when a solution is found. But software projects terminate because the team has run out of time, money, or patience. A solution to an equation system can be evaluated immediately and objectively as correct or wrong. But a software system can only be evaluated in terms

personal preference of the evaluator. In comparison, many software systems are subject to lifelong testing. For example, a computerized sell-off on May 6, 2010, sent the Dow Jones Industrials to a loss of nearly 1,000 points or 10% of its value at one time. Procter & Gamble, a stable blue chip stock, dropped almost 37% to a seven-year low. These were caused by a simple typographical error that should have been prevented. The software system has been in use for decades before the incident occurred. This and many other software-related incidents explain why software systems need lifelong testings. While the solution to a tame problem could be adapted to solve a similar problem, every application software development project is unique. This is why application software is developed or custom made, not manufactured.

Software development is not a scientific process—in other words, many decisions are not made scientifically but politically and economically. For example, a good-enough algorithm is chosen instead of an optimal one because it is more economical to implement, use, and maintain the good-enough algorithm. Sometimes, the choice to use a specific programming language or DBMS is a political decision. In one project that the author was contracted to develop, the client requested not to use a certain product because the client had a bad experience with its vendor. Finally and importantly, software failures could result in millions of dollars of property damages and loss of human lives. Therefore, software developers have no right to be wrong. Tame problems do not share this property—if the result is incorrect, then the software developer can simply try again.

2.5 SOFTWARE PROCESS MODELS

Problems with the waterfall process have led researchers and funding agencies to find a better process that considers the wicked properties of software development. Tons of money and effort have been poured into this research. As a result, many software process models have been proposed. Most models adopt an iterative, rather than a strictly sequential, process of activities. This section reviews some of these process models.

2.5.1 Prototyping Process

The prototyping process model recognizes the mismatch between the newly constructed software system and users' expectations, and the challenge to deliver the capabilities within the time and budget constraints. As a solution, a prototype of the system is constructed and used to acquire and validate requirements. Prototypes are also used in feasibility studies as well as design validation. A prototype can be very simple or very sophisticated. A simple prototype shows only the look and feel and a sequence of screen shots to illustrate how the system would interact with a user. A sophisticated prototype may implement many of the system functions. Prototypes are generally classified into throwaway prototypes and evolutionary prototypes. A throwaway prototype is constructed quickly and economically—just enough to serve its purpose. A throwaway prototype could be reused in unit or integration testings as a reference implementation to check whether the implementation produces the correct

2.5.2 Evolutionary Process

Prototypes help requirements acquisition, requirements validation, feasibility study, and validation of design ideas. However, throwaway prototypes imply that much effort is wasted. This is true when sophisticated prototypes are needed for feasibility study and design validation of large, real-time embedded systems. The *evolutionary process model* is aimed at solving this problem by letting the prototype evolve. It lets the users experiment with an initial prototype, constructed according to a set of preliminary requirements. Feedback from the users is used to evolve the prototype. This process is repeated until no more extensions are required. For this reason, it is also referred to as the *evolutionary prototyping process model.* Since the prototype is not a throwaway prototype, it is constructed with operational functionality and needed robustness.

The evolutionary process is most suitable for exploratory types of projects, where the exact requirements and algorithms are to be discovered from working with the system. Many real-world projects belong to this category, including intelligent systems, research software that aims to discover unknown things like gene sequencing, as well as embedded systems that actively interact with the environment. The evolutionary process is not suitable for projects that require a predictable schedule of progress.

2.5.3 Spiral Process

The spiral process proposed by Barry Boehm [32] is known for its unique feature for risk management. As its name implies, the development process looks like a spiral, as shown in Figure 2.2. Each cycle of the spiral is aimed at enhancing a certain aspect of the system under development, for example, functionality, performance, or quality. In this sense, the system evolves incrementally as the model iterates the spiral cycles. Each cycle of the spiral selectively executes some of the following steps:

1. *Determine the objectives, alternatives, and constraints for the current cycle* (the northwest corner of the spiral). The objectives, the alternative approaches to accomplish the objectives, and the constraints that must be satisfied are identified in this step. Project risk items are identified and prioritized.
2. *Evaluate alternatives; identify and resolve risks* (the northeast corner of the spiral). The alternatives to accomplish the objectives within the constraints are evaluated. Risks of not achieving the objectives are identified and ways to resolve the risks are developed. Prototyping, simulation, modeling, and benchmarking are some of the techniques for risk resolution. Depending on the outcome of risk analysis and resolution, the next step may be one of the following:

   - If there are remaining risks, then the subsequent steps would be the southwest corner, that is, planning for the next level of prototyping, followed by a new prototyping cycle.
   - If the previous cycles have resolved the major known risks, then the subsequent steps could proceed like the waterfall, as shown in the southeast corner of the model.
   - If the prototype produced during the previous rounds are operational and robust enough to evolve into a final system, then the subsequent steps would

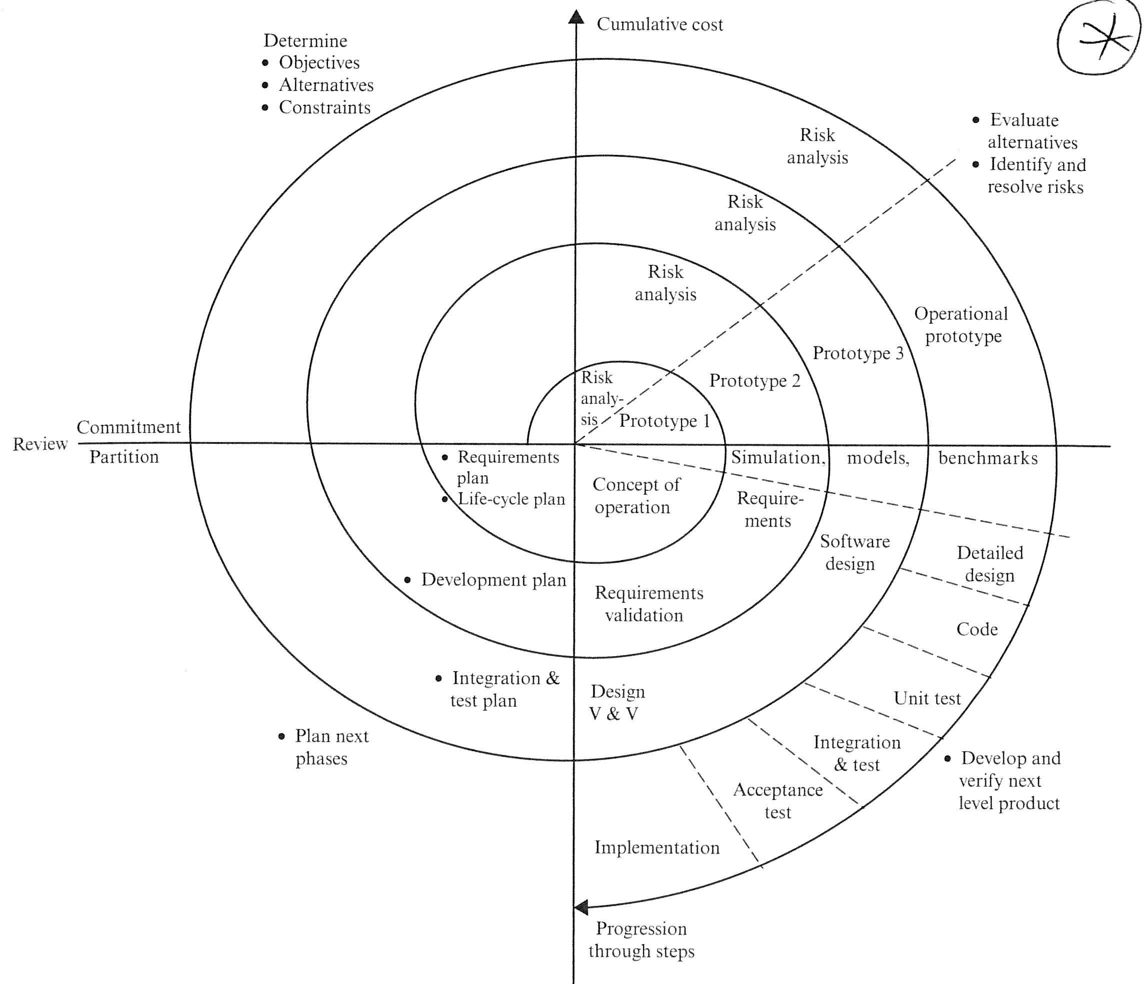


FIGURE 2.2 The spiral process

3. *Develop and verify next level system* (the southeast corner). This step proceeds like the waterfall model, as shown in Figure 2.2.
4. *Plan next phases* (the southwest corner). For either a new initiative or a continuing project, this step defines the requirements, the life-cycle activities, and the integration and test plan for the next phase.

2.5.4 The Unified Process

The Rational Unified Process or Unified Process (UP) [91], as shown in Figure 2.3, consists of a series of cycles. Each cycle concludes with a release of the system. Each cycle has several iterations. The iterations are grouped into four phases—inception, elaboration, construction, and transition. Each phase ends with a major milestone, at

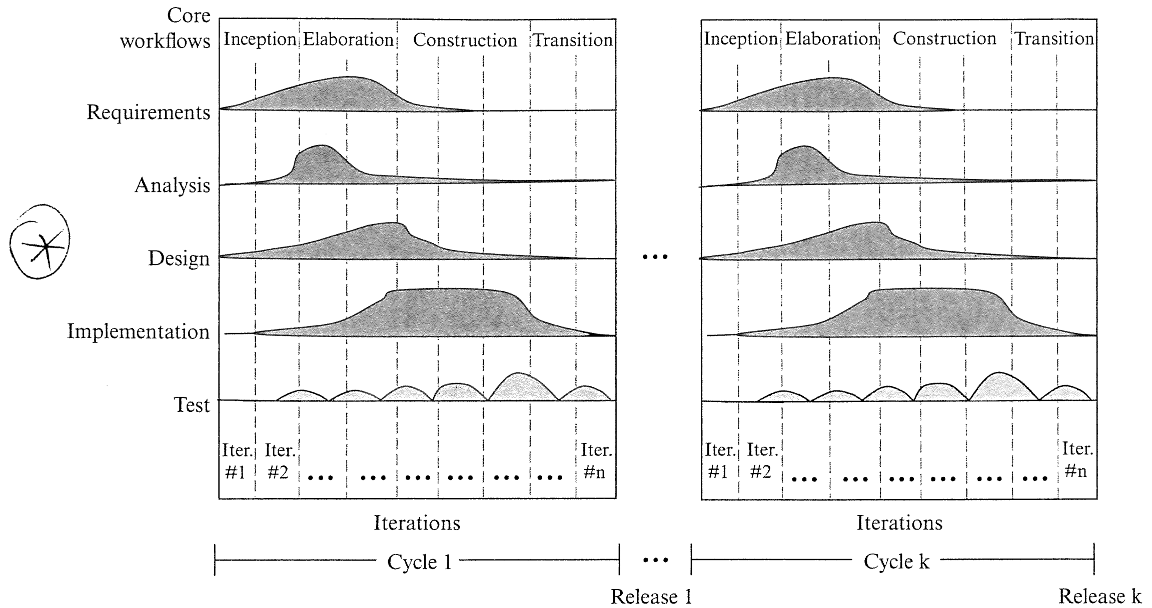


FIGURE 2.3 Illustration of the Unified Process

requirements analysis, design, implementation, and testing. The gray areas in Figure 2.3 are rough indications of the extent of the workflow activities that are carried out during the phases. The focuses of the four phases are described below.

Inception. The first one or two iterations constitute the inception phase. This phase produces a simplified use case model, a tentative software architecture, and a project plan. In simple terms, a use case models a business process of the application for which the system is being developed. Verb-noun phrases are used to describe use cases. For example, *Deposit Money, Withdraw Money*, and *Check Balance* are use cases for an automatic teller machine (ATM). The use case model contains the most critical use cases of the software system.

Elaboration. The elaboration phase consists of the next several iterations. During this phase, most use cases are specified, and UML diagrams representing the architectural design are produced. The most critical use cases of the software system are designed and implemented.

Construction. During the construction phase, the remaining use cases are iteratively developed and integrated into the system. The system can be incrementally installed in the target environment.

Transition. During the transition phase, activities are performed to deploy the software system. These include user training, beta testing, defect correction, and functional enhancement.

The UP focuses on identifying use cases and uses them to plan the iterations

driven. The UP determines the architecture or the overall structure of the system early in the life cycle, and uses it to guide the development of the software system. For this reason, the UP is said to be *architecture-centric*. The other two features of the UP are that it is *iterative* and *incremental* because the system is developed and deployed iteratively and incrementally.

2.5.5 Personal Software Process

The *personal software process* (PSP) is a comprehensive framework that is designed to train individual software engineers to improve their personal software processes. PSP consists of a series of scripts, forms, standards, and guidelines that the software engineer can apply to carry out a number of predefined programming exercises. Rather than enforcing a specific development method, the PSP allows the software engineer to choose the development methods. Throughout the training, the PSP helps the software engineer identify areas for improvement. It also helps the software engineer develop abilities that are useful in a teamwork environment, such as developing the ability to estimate more accurately and the ability to meet commitments. As stated above, the PSP is meant to improve the personal software process of a software engineer; it is not meant to be a software development process. That is, after the training, the software engineer is expected to develop and use his own software processes to produce high-quality software. It is believed that quality increase leads to productivity increase because the effort and time spent in testing and debugging is reduced.

The PSP Process Evolution

To facilitate learning, the PSP uses an evolutionary approach. That is, the framework is presented in a series of predefined processes, named PSP0, PSP0.1, PSP1, PSP1.1, PSP2, PSP2.1, and PSP3.0. Each of these processes introduces a couple of good software engineering techniques or practices.

PSP0 and PSP0.1. These two processes introduce process discipline and measurement. In particular, PSP0 introduces the baseline process, time recording, defect recording, and defect type standard. PSP0.1 introduces coding standard, size measurement, and process improvement proposal.

PSP1 and PSP1.1. These two processes introduce estimation and planning. In particular, PSP1 introduces size estimation and test report while PSP1.1 covers planning and scheduling.

PSP2 and PSP2.1. These two processes introduce quality management and design. In particular, PSP2 presents code review and design review, and PSP2.1 introduces a design template.

PSP3.0. This process is designed to guide the development of component-level programs.

PSP Script

In PSP, all processes are described using *process scripts* or scripts for short. Each script

Step	Activities	Description
Purpose		To guide module-level program development
Entry Criteria		• Problem description • PSP0 Project Plan Summary form • Time and Defect Recording logs • Defect Type standard • Stopwatch (optional)
Step	Activities	Description
1	Planning	• Produce or obtain a requirements statement • Estimate development time • Fill the Project Plan Summary form • Complete the Time Recording log
2	Development	• Design the program • Implement the design • Compile the program, fix and log all defects found • Test the program, fix and log all defects found • Complete the Time Recording log
3	Postmortem	Complete the Project Plan Summary form with actual time, defect, and size data
Exit Criteria		• A thoroughly tested program • Complete Project Plan Summary form with estimated and actual data • Completed Time and Defect Recording logs

FIGURE 2.4 PSP activities are described by scripts

exit criteria. For example, the baseline process PSP0 consists of six phases: *planning, design, code, compile, test,* and *postmortem.* This process can be described by the script shown in Figure 2.4. It consists of three steps: (1) planning, (2) development, which encompasses design, code, compile, and test, and (3) postmortem. The entry criteria of a script specify the software artifacts that must be available before the process can begin. The steps list the activities and descriptions of the activities. The exit criteria specify the software artifacts that must be produced when the process is completed.

The postmortem step in Figure 2.4 is a unique feature of the PSP. It requires the software engineer to complete a Project Plan Summary form with the actual time, defect, and size data. This form is described in PSP Forms below. The software engineer completes the form at the end of each programming exercise. The form allows the software engineer to observe his personal software practices, identify areas to improve, and acquire data to use in estimation.

The PSP adopts a recursive view of the development process. That is, a process consists of a series of activities and an activity can be described by a lower-level process. For example, the planning step in Figure 2.4 is a process, which can be described by another script. The script may consist of a requirements step, a resource estimation step, and a scheduling step. Similarly, the development step in Figure 2.4 can be described by a script consisting of design, code, compile, and test steps.

PSP Forms

PSP uses forms to facilitate documentation. Each form specifies the ordinary informa-

programming language. To be consistent with the forms, the following presentation will use the terms *student* and *software engineer* interchangeably. Some of the forms are described below.

The Time Recording Log This form has seven columns as shown in Figure 2.5. The Delta Time is the Stop Date and Time minus Start Date and Time minus Interrupt Time. Each student fills the entries for each step/phase of the process in one line of the form. For example, after completing the baseline process PSP0, the student fills one line of the form for each of the planning, design, code, compile, test, and postmortem steps.

The Defect Recording Log Figure 2.6 shows the Defect Recording Log form. At the top are the 10 defect types, which are explained in Figure 2.7. Each student is required to specify the defects detected and removed during the course of a process. For each

PSP Time Recording Log

Student ______________________ Date __________
Program ______________________ Program# __________
Instructor ______________________ Language __________

Project	Phase	Start Date and Time	Interrupt Time	Stop Date and Time	Delta Time	Comments

FIGURE 2.5 PSP time recording log

Defect Types	
10 Documentation	60 Checking
20 Syntax	70 Data
30 Build, Package	80 Function
40 Assignment	90 System
50 Interface	100 Environment

PSP Defect Recording Log

Student Student 3 Date 1/19
Program Standard Deviation Program# 1
Instructor Brown Language C++

Project	Date	Number	Type	Inject	Remove	Fix Time	Fix Ref.
1	1/19	1	20	Code	Comp.	1	

Description Missing semicolon.

Project	Date	Number	Type	Inject	Remove	Fix Time	Fix Ref.
		2	20	Code	Comp.	1	

Description Missing semicolon.

FIGURE 2.6 PSP defect recording log

Summary of PSP Defect Types

ID	Defect Type	Description
10	Document	Comments, messages, and manuals
20	Syntax	Spelling, punctuation, typos, and instruction formats
30	Build, Package	Change management, library, version control
40	Assignment	Declaration, duplicate names, scope, limits
50	Interface	Procedure calls and references, I/O, user formats
60	Checking	Error messages, inadequate checks
70	Data	Structure, content
80	Function	Logic, pointers, loops, recursion, computation, function defects
90	System	Configuration, timing, memory
100	Environment	Design, compile, test, or other support system problems

FIGURE 2.7 Summary of PSP defect types

defect, the student enters the program number, the date on which the defect was found, the defect identification number, the type of the defect, the phase in which the defect was introduced and removed respectively, the time spent to find and fix the defect, the defect number that during its fix, introduces the current defect, and a brief description of the defect including the error and why it was introduced.

The Project Plan Summary Form The Project Plan Summary form mentioned in the last section is shown in Figure 2.8. It consists of four sections. At the top of the form is the descriptive information, which specifies the student name, date, program name, program number, instructor, and the programming language used. The Time in Phase (in minutes) section specifies the Plan time for the process, and the Actual time, the To Date time, and the To Date % time for each of the phases. For example, if a student spent 30 minutes in the design phase for the first program, and 25 minutes in the design phase for the second program, then the Actual and To Date times for the first program are 30 minutes. The Actual and To Date times for the second program are 25 minutes and 55 minutes, respectively. If the total To Date time for the first program is 117 minutes, then the To Date % time for the first program is 25.6% (i.e., 30 divided by 117). The Defects Injected section has the same columns and rows as the Time in Phase section and records the number of defects by phase. The Defects Removed section is similar to the Defects Injected section and records the number of defects removed by phase.

The PSP also includes methods to help the software engineer in estimation and planning. In addition, PSP presents quality assurance procedures to help the software engineer improve software quality. These are presented in Appendix A.

2.5.6 Team Software Process

The team software process (TSP) is developed by the Software Engineering Institute (SEI) to enable team members who are trained in PSP to work together to carry out a team project. As shown in Figure 2.9, the TSP consists of a series of cycles. Each

PSP0 Project Plan Summary Form

Student	Student 3	Date	1/19
Program	Standard Deviation	Program #	1
Instructor	Brown	Language	C++

Time in Phase (min.)	**Plan**	**Actual**	**To Date**	**To Date %**
Planning		5	5	4.3
Design		30	30	25.6
Code		32	32	27.4
Compile		15	15	12.8
Test		5	5	4.3
Postmortem		30	30	25.9
Total	180	117	117	100.0

Defects Injected	**Actual**	**To Date**	**To Date %**
Planning	0	0	0.0
Design 2 2 28.6	2	2	28.6
Code 5 5 71.4	5	5	71.4
Compile 0 0 0.0	0	0	0.0
Test 0 0 0.0	0	0	0.0
Total Development	7	7	100.0

Defects Removed	**Actual**	**To Date**	**To Date %**
Planning	0	0	0.0
Design	0	0	0.0
Code	0	0	0.0
Compile	6	6	85.7
Test 1 1 14.3	1	1	14.3
Total Development	7	7	100.0
After Development	0	0	

FIGURE 2.8 PSP0 Project Plan Summary form

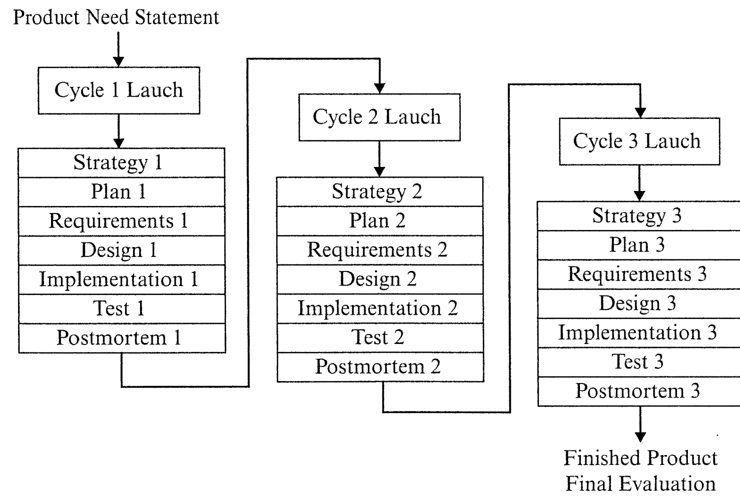


FIGURE 2.9 Team software process

cycle performs a series of activities. A TSP project begins with a TSP launch process to build the team and produce a project plan. The launch process is guided by a trained and qualified TSP coach. The process identifies the customer's needs, assigns roles to team members, produces an initial system concept, a development strategy, and a plan to develop the system. The TSP team also produces a quality plan and a risk management plan. The plans are presented to the management, which may approve or request changes. The last step of each cycle is the postmortem. At the postmortem meeting, the team reviews the launch process, identifies and records improvement suggestions, and assigns follow-up items to team members.

The TSP activities of each cycle are specified in a script. Figure 2.10 illustrates a script that is tailored to use the methodology presented in this textbook. That is, the methodology implements the TSP process. The script shown in Figure 2.10 is designed to fit one semester of teamwork, including learning. It has been tested several times. However, the script can be modified, or tuned to fit different situations. For example, it could run in a shorter period. In this case, there will be only one or two cycles. It could drop some topics, such as applying situation-specific patterns could be moved to another course. Another alternative is running the script to produce only the design but not the implementation.

2.5.7 Agile Processes

The waterfall process works well for tame problems because such problems possess a number of nice properties. Application software development is a wicked problem. It needs a process that is designed to solve wicked problems. Agile processes are such processes. Agile processes emphasize teamwork, joint application development with the users, design for change, and rapid development and frequent delivery of small increments in short iterations. Agile development is guided by agile values, principles, and best practices. All these take into account wicked-problem properties.

Agile Manifesto

According to the Agile Manifesto,[1] agile development values four aspects of software development practices, which are different from their conventional, plan-driven counterparts. These are listed and explained below.

- *Agile development values individuals and interactions over processes and tools.*
- *Agile development values working software over comprehensive documentation.*
- *Agile development values customer collaboration over contract negotiation.*
- *Agile development values responding to change over following a plan.*

1. **Agility values individuals and interactions over processes and tools.**

   Conventional, plan-driven practices believe that a good software process is essential for the success of a software project. One conventional wisdom is that

A Team Software Process Script

<table>
<tr><th colspan="2">Purpose</th><th>To guide a team through developing a software product</th></tr>
<tr><td colspan="2">Entry Criteria</td><td>• An instructor guides and supports one or more five-student teams.
• The students are all PSP trained.
• The instructor has the needed materials, facilities, and resources to support the teams.
• The instructor has described the overall product objectives.</td></tr>
<tr><td colspan="2">General</td><td>The PSP process is designed to support three team modes. Follow the scripts that apply:
1. Develop a small- to medium-sized software product in two or three development cycles.
2. Develop a smaller product in a single cycle.
3. Produce a product element, such as a requirements, design, or a test plan, in part of one cycle.</td></tr>
<tr><th>Week</th><th>Step</th><th>Activities</th></tr>
<tr><td>1</td><td>Review</td><td>• Course introduction and PSP review.
• Read preface, introduction, and this chapter, focus on the PSP section.</td></tr>
<tr><td rowspan="2">2</td><td>LAU1</td><td>• Review course objectives and assign student teams and roles.
• Read TSP and overview of the agile unified methodology in this chapter.</td></tr>
<tr><td>STRAT1</td><td>• Produce the conceptual design, establish the development strategy, make size estimates, and assess risk.
• Apply a software architectural design style (in most cases the N-tier architecture).
• Read architectural design, and project management chapter, focus on estimation and risk management sections.</td></tr>
<tr><td>3</td><td>PLAN1, REQ1</td><td>• Define and inspect requirements, focus on high-priority requirements.
• Derive use cases from the requirements, produce use case diagrams and traceability matrix, specify high-level use cases.
• Allocate the use cases to the cycles, produce allocation matrix.
• Review the use cases, use diagrams, high-level use case specifications, and matrices.
• Read system engineering, software requirements elicitation, and quality assurance chapters, focus on requirements related sections, read deriving use cases chapter.</td></tr>
<tr><td>4</td><td>REQ1, DES1</td><td>• Perform cycle 1 domain modeling (brainstorming, domain concept classification, and domain model visualization).
• Specify cycle 1 expanded use cases, produce use case based test cases.
• Review domain model, expanded use cases, and use case based test cases.
• Read domain modeling, actor-system interaction modeling, and software testing chapters (use case based testing).</td></tr>
<tr><td>5</td><td>DES1</td><td>• Produce and review cycle 1 scenarios, scenario tables, and sequence diagrams.
• Derive and inspect cycle 1 design class diagram (DCD), and user interface design.
• Read object interaction modeling, deriving design class diagram, and user interface design chapters.</td></tr>
<tr><td>6</td><td>IMP1</td><td>• Conduct cycle 1 test driven development (maybe combined with pair-programming) to fulfill 100% branch coverage.
• Review unit test cases and code.
• Read implementation, quality assurance, and software testing chapters.</td></tr>
<tr><td>7</td><td>TEST1</td><td>• Build, and integrate cycle 1, run use case based test cases.
• Demonstrate cycle 1 software to the customer and users, solicit and record feedback.
• Produce user documentation for cycle 1.</td></tr>
<tr><td rowspan="3">8</td><td>PM1</td><td>• Conduct a postmortem and write the cycle 1 final report.
• Produce role and team evaluations for cycle 1.</td></tr>
<tr><td>LAU2</td><td>• Re-form teams and roles for cycle 2.</td></tr>
<tr><td>STRAT2, PLAN2, REQ2</td><td>• Produce the strategy and plan for cycle 2, assess risks.
• Update and review requirements, domain model, use cases, traceability matrix, and allocation matrix.</td></tr>
<tr><td>9</td><td>DES2</td><td>• Apply GRASP patterns, and update and review cycle 1 sequence diagrams.
• Produce and inspect cycle 2 expanded use cases and use case based test cases.
• Apply GRASP, and produce and review cycle 2 scenarios, scenario tables and sequence diagrams.
• Read applying responsibility assignment patterns chapter.</td></tr>
</table>

FIGURE 2.10 TSP development script

10	IMP2	• Test driven develop and inspect cycle 2, accomplish 100% branch coverage. • Review unit test cases and code.
	TEST2	• Build, integrate, and test cycle 2, demonstrate cycle 2 software to the customer and users, solicit and record feedback. • Produce user documentation for cycle 2.
11	PM2	• Conduct a postmortem and write the cycle 2 final report. • Produce role and team evaluations for cycle 2.
	LAU3	• Reform teams and roles for cycle 3.
	STRAT3, PLAN3, REQ3	• Produce the strategy and plan for cycle 3, assess risks. • Update and review requirements, domain model, use cases, traceability matrix, and allocation matrix.
12	DES3	• Apply situation specific or Gang of Four patterns, update cycle 1 and cycle 2 design diagrams. • Produce and inspect cycle 3 expanded use cases and use case based test cases. • Produce and review cycle 3 sequence diagrams (situation-specific patterns are applied). • Read applying situation-specific patterns chapter.
13	IMP3	• Test driven develop and inspect cycle 3, accomplish 100% branch coverage. • Review unit test cases and code.
	TEST3	• Build, integrate, and test cycle 3, demonstrate finished product to the customer and users, solicit and record feedback. • Produce and review user's manual for the finished product.
14	PM3	• Conduct a postmortem and write the cycle 3 final report. • Produce role and team evaluations for cycle 3. • Review the product produced and the processes used, identify lessons learned and propose process improvements.
Exit Criteria		• Completed product or product element and user documentation. • Completed and updated project notebook. • Documented team evaluations and cycle reports.

FIGURE 2.10 *(Continued)*

"the software quality is as good as the software process." Although the conventional wisdom still has its merits, experiences indicate that the abilities of the team members as well as teamwork are more important. After all, it is the team members who carry out the software process. If the team members do not know how to design, or they do not communicate with each other effectively, then the result won't be good. Conventional practices place significant weight on the use of tools. For this reason, many companies invest heavily in development tools and environments. Some tools are good and solve the intended problems. But these can only be accomplished by the right people, who know how. A UML diagram editor won't help if the software engineer does not know how to perform OO design. Although the editor produces nice-looking UML diagrams, these are not necessarily good designs.

Unlike conventional practices, agile methods value individuals and teamwork. This is because software is a conceptual product and the development activities are highly intellectual. If the team members have to work together to jointly build the software product, then the abilities of the team members to interact and contribute to the joint effort is essential to the success of the project. Software processes and tools certainly matter, but individuals and interactions are essential.

2. Agility values working software over comprehensive documentation.

For years or even decades, companies spend tremendous efforts in preparing analysis and design documents. This is partly due to standards audits and partly due to the beliefs that "good software comes from good design documentation, and good design documentation comes from good analysis models." These beliefs are true, but only partly. Many software engineers have experienced that in some cases it is impossible to determine the real requirements, or whether the design works until the code is written and tested. In these cases, comprehensive documentation won't help and might be harmful because it gives the illusion that a working solution has been found. Comprehensive documentation also means less time is available to coding and testing, which are the only means in these cases to identify the real requirements and the needed design.

Consider, for example, a software to optimize the inventory for a large corporation. The inventory consists of textual descriptions of millions of items written by various employees during the last several decades. Numerous acquisition and merger activities significantly increase the number of items, categories of items, and description formats and styles. The software is required to process the inventory descriptions. The objective is to simplify the inventory and reduce inventory costs. Clearly, the requirements for the software are what the software can do to accomplish this objective. However, without implementing the software, nobody knows exactly what the software can accomplish. This is an example of a wicked problem—the specification and the implementation cannot be separated. Suppose that the requirements were somehow identified without needing to implement the software. Then, the design of data structures and algorithms is a grand challenge because it is extremely difficult to know whether the algorithms work and to what extent. This is due to the diversity of the inventory descriptions, inconsistencies, incomplete entries, typos, and abbreviation variations. A trial-and-error approach seems to be more appropriate.

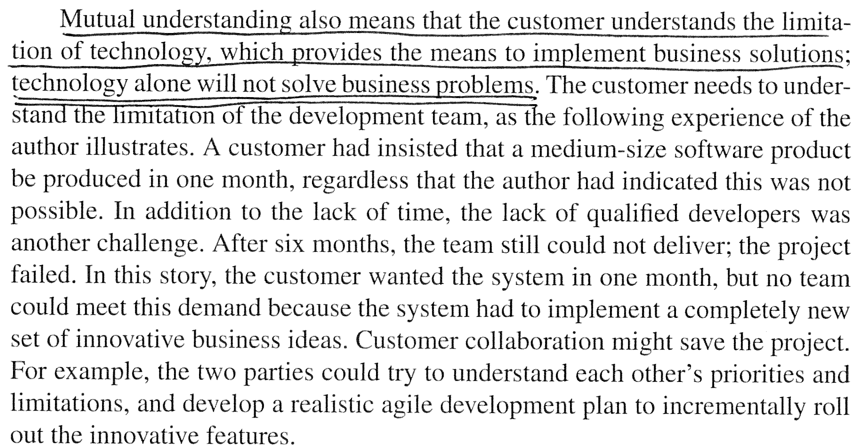
Agile methods value working software because working software is the bottom line. After all, the development team has to deliver the working software to the customer. Only the working software can be tested to ensure that the software system delivers the required capabilities. In this sense, the working software is the requirements and vice versa. The inventory description classification project discussed above illustrates this. However, this discussion must not lead to the conclusion that agile methods do not want analysis and design. On the contrary, agile methods construct analysis and design models. Nevertheless, agile principles advocate *just barely enough modeling* to help understand the problem and communicate the design idea but no more.

3. Agility values customer collaboration over contract negotiation.

Conventional processes involve a contract negotiation phase to identify what the customer wants. A requirements specification is then produced and becomes a part of the contract. During the development process the customer only participates in a couple of design reviews and acceptance testing. Many important design decisions that should be made with the customer are made by the development team. Although the development team is good in making technical

decisions, it may not possess the knowledge and background to make decisions for the customer. For example, a requirement to support more than one DBMS may not specify which DBMSs are to be supported. Technically, the team may know which DBMSs are the best and should be supported. But the customer may consider other factors to be more important. These include the ability of its information technology (IT) staff to maintain the types of DBMSs, costs to introduce such systems, and compatibility with existing systems. If the development team makes such decisions for the customer, then the resulting system may not meet the customer's business needs.

Customer collaboration is essential for the success of a project. It improves communication and mutual understanding between the team and the customer. Improvement in communication helps in identifying real requirements and reducing the likelihood of requirements misconception. Mutual understanding implies risk sharing; and hence, it reduces the probability of project failure. For many projects, the exact outcome of the system, a design decision, or an algorithm is difficult or impossible to predict. In these cases, customer collaboration is extremely important. Mutual understanding means that the development team has a good understanding of the customer's business domain, operations, challenges, and priorities. This enables the team to design and implement the system to meet the customer's business needs.



Mutual understanding also means that the customer understands the limitation of technology, which provides the means to implement business solutions; technology alone will not solve business problems. The customer needs to understand the limitation of the development team, as the following experience of the author illustrates. A customer had insisted that a medium-size software product be produced in one month, regardless that the author had indicated this was not possible. In addition to the lack of time, the lack of qualified developers was another challenge. After six months, the team still could not deliver; the project failed. In this story, the customer wanted the system in one month, but no team could meet this demand because the system had to implement a completely new set of innovative business ideas. Customer collaboration might save the project. For example, the two parties could try to understand each other's priorities and limitations, and develop a realistic agile development plan to incrementally roll out the innovative features.

4. Agility values responding to change over following a plan.

Conventional practices emphasize "change control" because change is costly. Once an artifact, such as a requirements specification, is finalized, then subsequent changes must go through a rigorous change control process. The process hinders the team to respond to change requests. Agile methods value responding to change over following a plan because change is the way of life. In today's rapidly changing world, every business has to respond quickly to change in business conditions in order to survive or grow. Thus, change to software is inevitable. Advances in Internet technologies enable as well as require businesses to update their web applications quickly and frequently. The inflexibility of the conventional, plan-driven practice cannot satisfy the needs of such applications. Agile

Agile Principles

The agile values express the emphases of agile processes. To guide agile development, the agile community also develops a set of guiding principles called agile development principles or agile principles for short. These principles are as follows:

1. *Active user involvement is imperative.*
   Active user involvement is required by many agile methods. This is because identifying the real requirements is the hardest part for many software development projects. Conventional approaches spend 15%–25% of the total development effort in requirements analysis. They implement rigid change control to freeze the requirements. These do not seem to solve the problem. It is not the lack of time or effort: It is the inability of human beings to know the real requirements in the early stages of the development process. Moreover, the world is changing so the requirements ought to evolve.

   Active user involvement means that representatives from the user community interact with the development team closely and as frequently as needed. For example, a couple of knowledgeable user representatives are assigned to the project. They stay and work with the team or visit the team regularly several times a week. These arrangements greatly improve the communication and understanding between the team and the users. These, in turn, ensure that requirement misconceptions are corrected early, users' feedback is addressed properly and timely, and decisions about the system are made with the users. All these imply that real requirements are identified and prioritized, and the system is built to meet users' expectations.
2. *The team must be empowered to make decisions.*
   Agile development values individuals and interactions over processes and tools. This principle realizes this. That is, team members are required and encouraged to make decisions and take responsibility and ownership. To be able to do this, the team members are required to work as a team and interact with each other and the users throughout the project.
3. *Requirements evolve but the timescale is fixed.*
   Unlike conventional approaches that freeze the requirements, agile processes are designed to welcome change. This principle means that the scope of work is allowed to evolve to cope with requirements change, but the agreed time frame and budget are fixed. This means that new or modified requirements are accommodated at the expense of other requirements. That is, the extra effort is compensated by giving up other requirements that are not mission critical.
4. *Capture requirements at a high level; lightweight and visual.*
   Agile development values working software over comprehensive documentation. After all, the bottom line is to deliver the working system, not the analysis and design documentation. To accomplish this, agile methods capture barely enough of the requirements with user stories, features, or use cases written on small-size story cards. These are visualized using storyboards or sequences of screen shots, sketches, or other visual means to show how the user would interact with

change and trade off requirements because story cards and storyboards are easy to share and manipulate.

5. *Develop small, incremental releases and iterate.*
   Agile projects develop and deploy the system in bite-size increments to deliver the use cases, user stories, or features iteratively. This arrangement has several advantages: project progress is visible, the users only need to learn a few new features at a time, it is easier for the users to provide feedback, and small increments reduce risks of project failure.
6. *Focus on frequent delivery of software products.*
   Before agile development, there are iterative approaches such as the spiral process and the unified process. Agile processes differ from their predecessors in frequent delivery of the software system in small increments. Different agile methods suggest different iteration lengths, which range from daily to three months. For example, Dynamic Systems Development Method (DSDM) suggests two to six weeks while Extreme Programming (XP) uses one to four weeks. An iteration in Scrum is called a sprint and is usually set to 30 days. The iteration duration of the methodology presented in this book can range from two weeks to three months.
7. *Complete each feature before moving on to the next.*
   This principle means that each feature must be 100% implemented and thoroughly tested before moving onto the next. The challenge here is that how do we know that the feature is thoroughly tested? Test-driven development (TDD) and test coverage criteria provide a solution. TDD requires that tests for each feature must be written before implementation. Test coverage criteria define the coverage requirements that the tests must satisfy. For example, the 100% branch coverage criterion is used by many companies. It requires that each branch of each conditional statement of the source code must be tested at least once.
8. *Apply the 80-20 rule.*
   This is also referred to as the "good enough is enough" rule. The rule is based on the belief that 80% of the work or result is produced by 20% of the system functionality. Therefore, priority should be given to the 20% of the requirements that will produce the 80% of work or result. This principle advises the development team to direct the customer and users to identify and prioritize such requirements. The rule also reminds team members of the diminishing return associated with the final extra miles. This applies to features that are nice to have, and performance optimization that is not really needed, and so forth. For example, an optimal algorithm may not be worth the extra implementation effort if a simpler algorithm is fast enough for the data to be processed.
9. *Testing is integrated throughout the project life cycle; test early and often.*
   This principle and principles 5–7 complement each other. That is, testing is an integral part of frequent delivery of completely implemented small increments of the system. This principle is supported by test tools such as JUnit, a Java class unit testing and regression testing tool. Using such a tool, a programmer needs to specify how to invoke the feature to be tested and how to evaluate the test result. The tool will generate the tests, run the tests, and check the test result, all automatically. The tests can be run as often as desired.

10. *A collaborative and cooperative approach between all stakeholders is essential.* Conventional approaches rely on comprehensive documentation to communicate the requirements to the development team. Agile projects capture requirements at a high level and light weight. Therefore, collaboration and cooperation between the development team and the customer representatives and users are essential. The parties must understand each other and work together throughout the life cycle to identify and evolve the requirements. Because the new system may significantly change or affect the work habit and performance of the users, collaboration and cooperation between the team and users are essential to the success of the project.

2.6 SOFTWARE DEVELOPMENT METHODOLOGY

Software development requires not only a process but also a methodology or development method. Unfortunately, the term "methodology" is often left undefined. This leads to a certain degree of confusion. For example, methodology is often confused with process. Process and methodology are important concepts of software engineering. The two are related but they are not the same. Below is a definition for a software methodology:

Definition 2.2 A *software methodology* defines the steps or how to carry out the activities of a software process.

(*) def.

A process in general specifies only the activities and how they relate to each other. It does not specify how to carry out the activities. It leaves the freedom to the software development organization to choose a methodology, or develop one that is suitable for the organization. The definition means that a methodology is an implementation of a process. Software development needs a process and a methodology.

2.6.1 Difference between Process and Methodology

(*)

Figure 2.11 provides an itemized summary of the differences between a process and a methodology. While a software process defines the phased activities or *what to do* in each phase, it does not specify how to perform the activities. A software methodology defines the detailed steps or *how to carry out* the activities of a process. A software process specifies the input and output of each phase, but it does not dictate the representations of the input and output. A methodology defines the steps, step entrance, and exit criteria, and relationships between the steps. A methodology also specifies, for each step, procedures and techniques, principles and guidelines, step input and output, and representations of the input and output. The representations of the artifacts provided by a methodology depend on the view of the world or the paradigm. For example, the object-oriented paradigm views the world and systems

Process	Methodology
• Defines a framework of phased activities • Specifies phases of WHAT • Usually does not dictate representations of artifacts • Hence, it is paradigm-independent • A phase can be realized by different methodologies	• Amounts to a concrete implementation of a process • Describes steps of HOW • Defines representations of artifacts • Hence, it is paradigm-dependent • Each step describes specific procedures, techniques, and guidelines
Examples: • Waterfall process • Spiral process • Prototyping process • Unified Process • Personal software process • Team software process • Agile processes	**Examples:** • Structured analysis/structured design methodology (SA/SD) • Object Modeling Technique (OMT) • Agile methods such as Scrum, Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD), Extreme Programming (XP), and Crystal Orange

FIGURE 2.11 Software process and methodology contrasted

diagrams are created to model objects. In this sense, methodologies are paradigm dependent. A software methodology can be viewed as a concrete implementation of a software process. This implies that a software process may have more than one software methodology as its implementation.

2.6.2 Benefits of a Methodology

The use of a good software development methodology is associated with a number of benefits, including:

1. A good methodology enables the development team to focus on the important tasks, and know how to perform these tasks correctly to produce the desired software system.
2. A good methodology improves communication and collaboration because:
   - the methodology defines a common language for modeling, analysis, and design.
   - the methodology defines the steps for effectively carrying out a development task that everybody knows and follows.
3. A good methodology improves design quality and software productivity because:
   - the software engineers are empowered to correctly and effectively apply the modeling, analysis, and design concepts and tools to construct the system.
   - the peer-review guidelines or checklists enable software engineers to conduct effective inspection and peer reviews to identify flaws in the requirements specification, design, and source code. These in turn reduce testing, debugging, and maintenance costs.
4. A good methodology forms the basis for process improvement because measurements of software quality, productivity, cost, and time to market can be defined

5. A good methodology forms the basis for process automation because many of the methodological steps can be mechanically carried out, making software automation much easier.
6. A methodology that is easy to learn and use enables beginners to produce quality software. As the beginner becomes familiar with the methodology, he can skip some of the steps in agile development.

Software development methodologies have evolved during the last several decades. The following sections review some of the representative methodologies proposed in this period.

2.6.3 Structured Methodologies

Structured analysis and structured design (SA/SD) methodologies were proposed in the 1970s and reached maturity in the 1980s. They are still in use today. Structured analysis uses data flow diagrams (DFDs) to model the business processes of real-world applications. A DFD is essentially a directed graph, in which the vertexes represent external entities, business processes, and data stores while the directed edges represent data flows between them. Divide-and-conquer is employed during structured analysis to decompose complex business processes into lower-level data flow diagrams.

The steps of structured analysis begin with the construction of a top-level DFD, called the context diagram. It depicts the system as the sole process, which interacts with external entities and external data stores. The next steps repeatedly decompose complex processes into simpler processes. This continues until the leaf-level processes can be easily implemented. The input, output, and their data structures, as well as the algorithms of the processes are specified. The DFD is good for describing the existing as well as the proposed business processes. It is not suitable for depicting the invocation relationships between the software modules. This is because the relationships between the processes of a DFD are data flow relationships while the relationships between the software modules are control flow relationships. The so-called structured design fills the gap.

The steps of structured design converts the data flow diagram analysis model into a structure chart or routine diagram, in which the vertexes represent the subroutines, and the edges represent function calls from high-level subroutines to their subordinates.

2.6.4 Classical OO Methodologies

Before UML, there were classical OO methodologies, with three of them widely known. They are the Booch Method, the Object Modeling Technique (OMT), and the use case driven approach. These three methodologies provide the basis for the UML 1.0. The classical OO methodologies were used by numerous software development organizations and contributed to the bloom of the OO paradigm. But the software industry soon discovered that it was a nightmare to integrate and maintain systems that were developed using different methodologies. It was also very costly to support different tools that use different methodologies. These problems called for a unified modeling method and led to the creation of UML and UP.

2.7 AGILE METHODS

Like the evolutionary prototyping model and the spiral model, all agile methods adopt an iterative, incremental development process. However, all agile methods follow the agile manifesto presented in Section 2.5.7. Agile processes emphasize short iterations and frequent delivery of small increments. Although they differ in the naming and detail of the phases, all agile methods more or less cover requirements, design, implementation, integration, testing, and deployment activities during each iteration. However, their emphases are different from conventional processes. For example, agile processes value working software over comprehensive documentation. This means barely enough modeling in the requirements and design phases. This section describes several of the most widely used agile methods. Figure 2.12 gives a brief summary of some of the agile methods, which are described in more detail in the next several sections. Each of these methods has a long list of principles, features, values, and best practices. Instead of showing these, Figure 2.12 lists only three of the most unique features of each agile method.

2.7.1 Dynamic Systems Development Method

The DSDM emerged in the early 1990s in the United Kingdom as an alternative to rapid application development (RAD). It is a process framework that different projects can adapt to perform rapid application development. It has been deemed by some authors to be most suited to financial services applications. The DSDM process is an iterative, incremental process guided by a set of DSDM principles, which are similar to the 10 agile principles presented in Section 2.5.7. As shown in Figure 2.13, the DSDM process consists of five phases. The first two phases are performed only once while the other three phases are iterative:

1. *Feasibility study.* During this phase, the applicability of DSDM and the technical feasibility of the project are determined. The end products include a feasibility report, an outline project plan, and optionally a prototype that is built to assess the feasibility of the project. The prototype may evolve into the final system.
2. *Business study.* During this phase, the requirements are identified and prioritized, a preliminary system architecture is sketched. The end products include a business area definition, a system architecture definition, and an outline prototyping plan.
3. *Functional model iteration.* During this phase, a functional prototype is iteratively and incrementally constructed. The end products include a functional model containing the prototyping code and the analysis models, a list of prioritized functions, functional prototype review documents, a list of nonfunctional requirements, and risk analysis for further development. The prototype review documents specify the user's feedback to be addressed in subsequent increments. The functional prototype will evolve into the final system.
4. *Design and build iteration.* During this phase, the system is designed and built to fulfill the functional and nonfunctional requirements, and tested by the users.

AUM*	DSDM	SCRUM	FDD	XP
Key Features				
• A cookbook for teamwork using UML • For beginners and seasoned developers • Suitable for agile or plan-driven, large or small team projects	• A framework that works with Rational Unified Process and XP • Base on 80–20 principle • Suitable for agile or plan-driven projects	• Include Srum Master, Product Owner, and Team Roles • 15-minute daily status meeting to improve communication • Team retrospect to improve process	• Feature-driven and model-driven • Configuration management, review and inspection, and regular builds • Suitable for agile or plan-driven projects	• Anyone can change any code anywhere at any time • Integration and build many times a day whenever a task is completed • Work ≤ 40 hours a week
Life-Cycle Activities				
Preplanning 1. Acquire and prioritize requirements 2. Derive use cases 3. Assign use cases to increments 4. Architectural design **During Each iteration:** 1. Accommodating change 2. Domain modeling 3. System-actor interaction modeling 4. Behavior modeling 5. Design class diagram 6. Test-driven development/pair programming 7. Integration testing 8. Deployment	**Feasibility Study** 1. Assess suitability of DSDM for project 2. Identify risks **Business Study** 1. Produce prioritized requirements 2. Architectural design 3. Risk resolution **Functional Model Iteration** 1. Identify prototype functionality 2. Build, review, and approve prototype **Design and Build Iteration** 1. Build system 2. Conduct beta test **Implementation** 1. Deploy system 2. Assess impact to business	**Release Planning Meeting** 1. Identify and prioritize requirements (*product backlog*) 2. Identify top-priority requirements that can be delivered within an increment called a sprint 3. Identify sprint development activities **Sprint Iteration:** 1. Sprint planning meeting to determine what and how to build next 2. Daily Scrum meeting for team members to report status **Sprint Review Meeting** 1. Increment demo 2. Team retrospection **Deployment**	**Develop Overall Model** 1. System walkthrough 2. Develop small group models 3. Derive overall model 4. Produce model description **Build Feature List** 1. Identify business activities to be automated 2. Identify features of business activities **Plan by Feature** 1. Schedule development of business activities 2. Assign business activities to chief programmers 3. Assign classes to team members **Design by Feature** 1. Produce sequence diagrams to show the interaction of objects using features 2. Encapsulate features to form classes to be implemented **Build by Feature** Implement the classes **Deployment**	**Exploration** 1. Collect information about the application 2. Conduct feasibility study **Planning** 1. Determine the stories for the next release 2. Plan for the next release **Iterations to First Release** 1. Define/modify architecture 2. Select and implement stories for each iteration 3. Perform functional tests by customer **Productionizing** 1. Evaluate and improve system performance 2. Certify and test system for production use **Maintenance** 1. Improve the current release 2. Repeat process with each new release **Death** 1. Produce system documentation if project is done, or 2. Replace system if maintenance is too costly

* AUM: the agile unified methodology described in this book

FIGURE 2.12 Summary of some agile methods

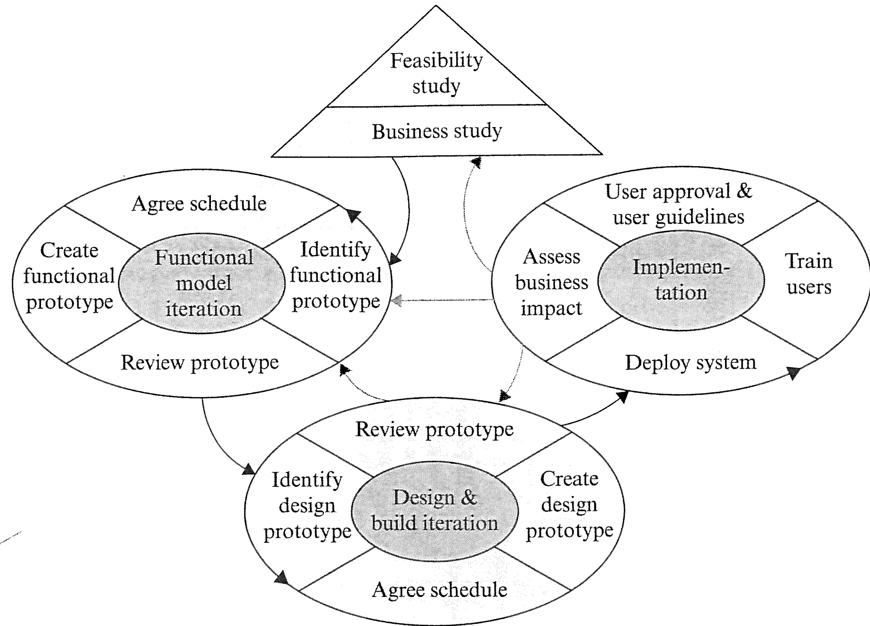


FIGURE 2.13 Process of the Dynamic Systems Development Method

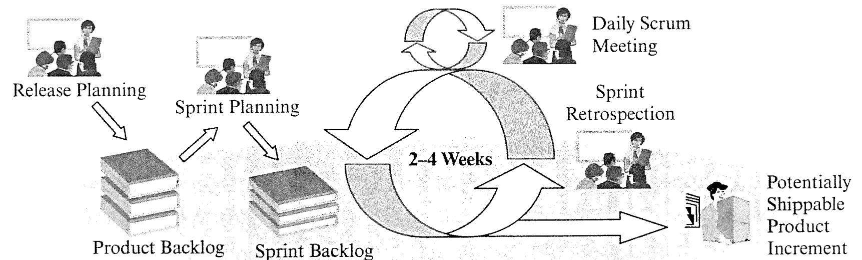
DSDM

5. *Implementation.* During this phase, the system is installed in the target environment and user training is conducted. The end products include a user's manual and a project review report, which summarizes the outcome of the project and what to do in the future.

2.7.2 Scrum

Scrum is a framework that allows organizations to employ and improve their software development practices. It consists of the Scrum teams, the roles within a team, the time boxes, the artifacts, and the Scrum rules. Scrum is an iterative, incremental approach that aims to optimize predictability and control risk. As displayed in Figure 2.14, there

SCRUM



is a release planning meeting. It determines the product backlog and the priorities of the requirements as well as planning for the iterations, called *sprints*. During the sprint iteration phase, the team performs the development activities to develop and deploy increments of the product. Each sprint begins with a sprint planning meeting, at which the team and the product owner determine which items of the product backlog are to be delivered next and how to develop them. Each sprint lasts 30 days, but a shorter or longer time period is allowed. One distinctive feature of the Scrum method is its 15-minute daily Scrum meeting. It allows the team members to exchange progress status to improve mutual understanding. Another distinctive feature of the Scrum method is the team retrospection at the end of each Scrum sprint. This meeting allows the team to improve its practices.

2.7.3 Feature Driven Development

As shown in Figure 2.12, the Feature Driven Development (FDD) method consists of six steps or phases. The first three are performed once and the last three are iterative. The FDD method is considered more suitable for developing mission critical systems by its advocates. The six phases of FDD are briefly described as follows:

1. *Develop overall model.* During this phase, a domain expert provides a walkthrough of the overall system, which may include a decomposition into subsystems and components. Additional walkthroughs of the subsystems or components may be provided by experts in their domains. Based on the walkthroughs, small groups of developers produce object models for the respective domains. The development teams then work together to produce an overall model for the system.
2. *Build a feature list.* During this phase, the team produces a feature list representing the business functions to be delivered by the system. The features of the list may be refined by lower-level features or functions. The list is reviewed with the users and sponsors.
3. *Plan by feature.* During this phase, the team produces an overall plan to guide the incremental development and deployment of the features, according to their priorities and dependencies. The features are assigned to the chief programmers. The chief programmer is the main decision maker of the team. This team organization is referred to as the *chief programmer team organization*. The classes specified in the overall model are assigned to the developers, called *class owners*. A project schedule including the milestones is generated.
4. *Design by feature, build by feature, and deployment.* These three phases are iterative, during which the increments are designed, implemented, reviewed, tested, and deployed. Multiple teams may work on different sets of features simultaneously. Each increment lasts a few days to a few weeks.

The roles and their responsibilities of an FDD project are similar to the common job titles. These include project manager, chief architect, development manager, chief programmer, class owner, domain expert, release manager, toolsmith, system administrator, tester, and technical writer.

2.7.4 Extreme Programming

Extreme programming or XP is an agile method suitable for small teams facing vague and changing requirements. The driving principle of XP is taking commonsense principles and practices to extreme levels [21]. For example, if frequent build is good, then the teams should perform many builds every day. The XP process consists of six phases:

1. *Exploration.* During this phase, the development team and the customer jointly develop the user stories for the system to the extent that the customer is convinced that there are sufficient materials to make a good release. A user story specifies a feature that a specific user wants from the system. For example, "as a patron, I want to check out documents from the library system." The development team also explores available technologies and conducts a feasibility study for the project. This phase should take no more than two weeks.
2. *Planning.* During this phase, the development team and the customer work together to identify the stories for the next release, including the smallest, most valuable set of stories for the customer. The stories should require about six months of effort to implement. A plan is produced for the next release. This phase should take no more than a couple of days.
3. *Iterations to first release.* During this phase, the overall system architecture is defined. The customer chooses the stories, the team implements them, and the customer tests the functionality. These activities are performed iteratively until the software is good for production use. Each iteration lasts from one to four weeks.
4. *Productionizing.* During this phase, issues such as performance and reliability that are not acceptable for production use are addressed and removed. The system is tested and certified for production use. The system is installed in the production environment.
5. *Maintenance.* According to Beck [21], this phase is really the normal state of an XP project. During this phase, the system undergoes continual change and enhancements, such as major refactoring, adoption of new technology, and functional enhancements with new stories from the customer. The process is repeated for each new release of the system.
6. *Death.* The system evolves during the maintenance phase until the system completely satisfies the customer's business needs and hence no more customer stories are added. When this happens, the project is done and enters the death phase, during which the team produces the system documentation for training, repair, and reference. The project also enters the death state if it cannot live to the customer's expectation.

2.7.5 Agile or Plan-Driven

Although agile methods are getting increasingly popular in the software industry, that does not mean that they do not have limitations. In [35], Boehm and Turner point out

dominates the other." Agile methods work well for small to medium-size projects that face frequent changes in requirements. Plan-driven approaches remain the de facto choice for large, complex systems and equipment manufacturers where predictability is important. Therefore, both approaches are needed. According to Boehm and Turner [35], plan-driven and agile methods "have shortcomings that, if left unaddressed, can lead to project failure. The challenge is to balance the two approaches to take advantage of their strengths and compensate for their weaknesses." That is, we need methods that can adapt to the cultures and circumstances of different software development projects and organizations. Such methods include Crystal Orange [50], DSDM [140], FDD [119], Lean Development [124], lightweight unified process (LUP) [104], and the methodology presented in this book.

2.8 OVERVIEW OF PROCESS AND METHODOLOGY OF THE BOOK

This section presents the agile unified process (AUP) and the methodology used in this book. As shown in Figure 2.15(b), the process could be viewed as a vertical slicing of the waterfall process. Each slice denotes an iteration, which ranges from one week to three months. The process can be viewed as consisting of two axes. The horizontal axis represents the iterations and the vertical axis represents the workflow activities of each iteration. Each iteration performs most of the workflow activities as the waterfall process except that they deal only with the use cases allocated to the iteration. As shown in Figure 2.15(b), before the iterations, there is a brief planning phase, which lasts about a couple of weeks, to identify requirements, derive use cases,

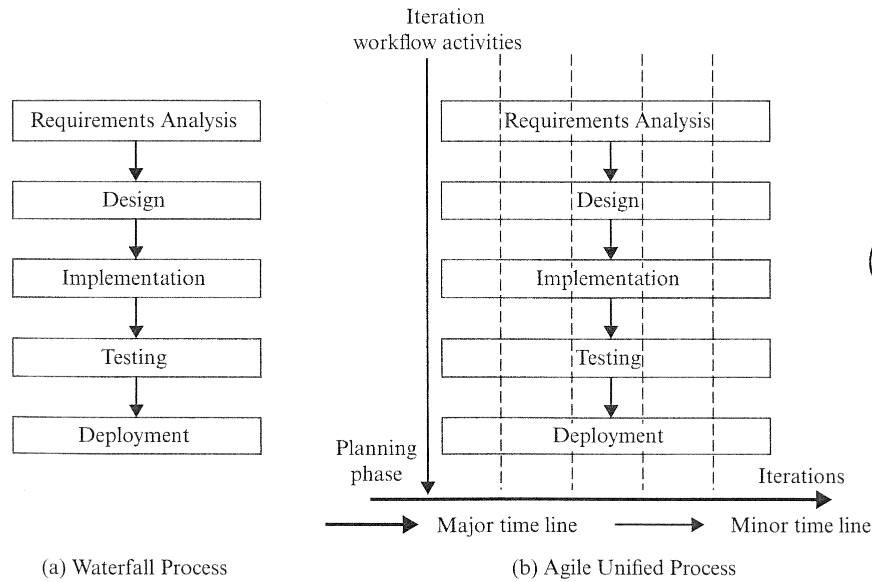
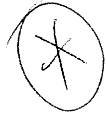


FIGURE 2.15 Waterfall and an agile process



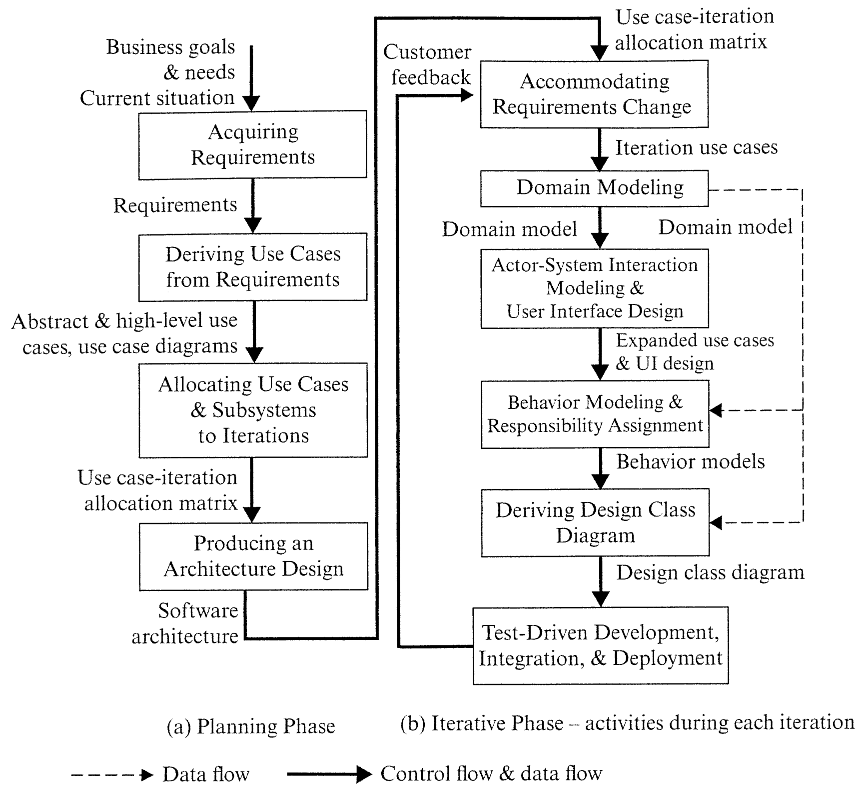


FIGURE 2.16 Overview of the agile unified methodology

and assign them to the iterations. Changes to the requirements, if any, are addressed during the requirements phase of each of the following iterations. Requirements change may lead to changes to the list of use cases and the delivery schedule of the use cases.

Figure 2.16 shows the steps of the agile unified methodology (AUM), along with their input, output, and relationships. The methodology consists of two main phases: (a) the planning phase, and (b) the iterative phase. During the planning phase, the development team meets with the customer representatives and users to identify the requirements and derive use cases from the requirements. The development team also produces an architectural design and a plan to develop and deploy the use cases during the iterative phase. Use case diagrams are produced during the planning phase to show the use cases, and subsystems and components that contain the use cases. Consider, for example, a library information system (LIS). The system may have a few dozen requirements. In illustration, three of them, labeled R1, R2, and R3, are

R1. The LIS must allow patrons to checkout documents.

R2. The LIS must allow patrons to return documents.

R3. The LIS must allow patrons to search for documents using a variety of search

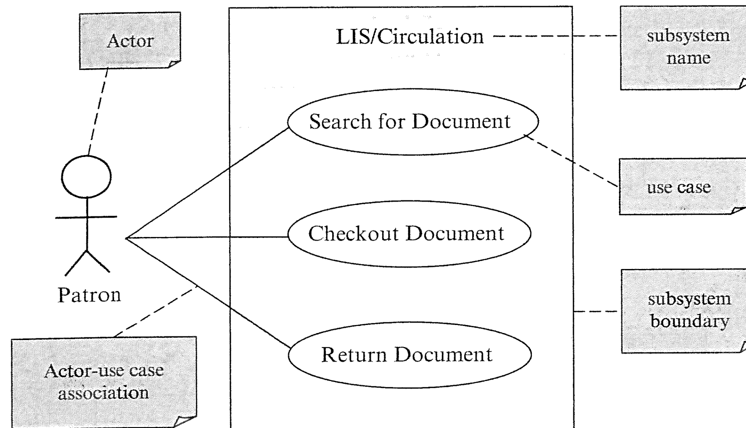


FIGURE 2.17 A use case diagram for a library information system

From these requirements, three use cases are derived—*Checkout Document, Return Document* and *Search for Document*, respectively. The LIS has many other requirements and use cases besides these. A mechanism to partition a large set of use cases is needed. Use case diagrams are useful for organizing the use cases into subsystems so that each subsystem can be dealt with separately. A use case diagram displays the use cases of a system or subsystem, and relationships between the use cases and system users called *actors*. As Figure 2.17 exhibits, the "circulation subsystem" of the LIS has three use cases. Similarly, other use case diagrams could display the use cases of the "cataloging subsystem" and "purchasing subsystem," respectively. These subsystems may be assigned to different development teams to design and implement.

During the iterative phase, the use cases are developed and deployed to the target environment iteratively according to the schedule produced in the planning phase. In particular, each iteration performs the following steps for the use cases assigned to the iteration.

Accommodate Requirements Change

Requirements change is common for many software projects. Such change, if any, is processed at the beginning of each iteration. Change to requirements leads to change to use cases as well as to the plan to develop and deploy the use cases. After making these changes, the development continues with the following steps according to the changed iteration schedule.

Conduct Domain Modeling

Domain modeling is a process to discover important domain concepts. For example, important domain concepts of a library system include user, patron, phone number, document, book, periodical, checkout date, checkout duration, due date, loan,

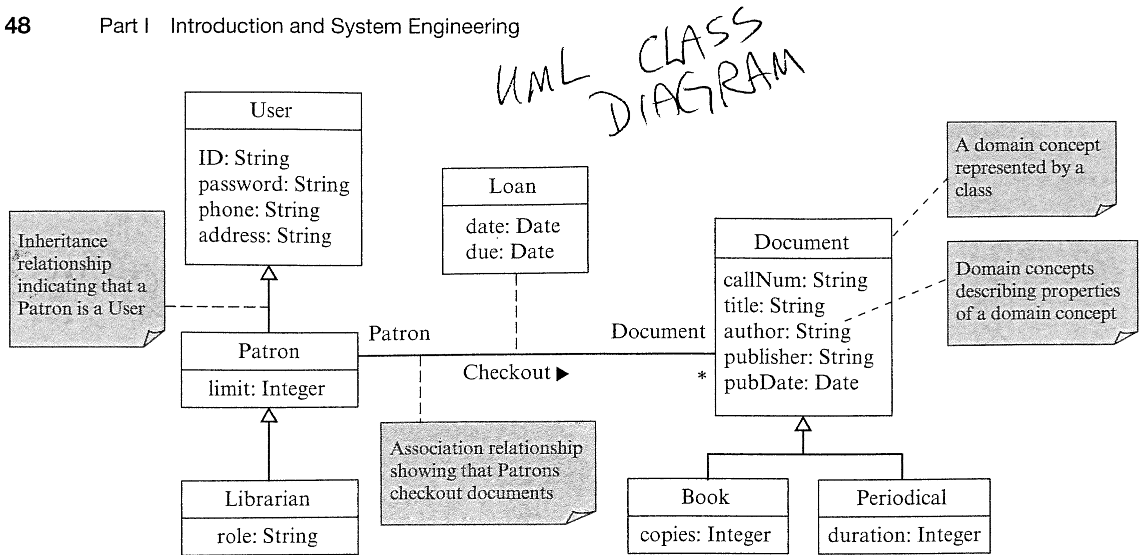


FIGURE 2.18 Sample domain model for a library system

through communication with the customer representatives, users, and domain experts. The domain concepts are classified into classes, attributes of the classes and relationships among the classes. The result is called a *domain model*, visualized by using a UML class diagram. A sample domain model for a library system is shown in Figure 2.18.

Conduct Actor-System Interaction Modeling

Users interact with the system to obtain the services provided by the system. Actor-system interaction modeling specifies how users interact with the system to carry out the use cases. The specifications are called *expanded use cases*. In illustration, Figure 2.19 shows an expanded use case specification for the Checkout Document

UC1: Checkout Document

Actor: Patron	System: LIS
	0. System displays the main menu.
1. This use case begins with patron clicks the 'Checkout Document' button on the main menu.	2. The system displays the 'Checkout' menu.
3. The patron enters the call number of a document to be checked out and clicks the 'Submit' button.	4. The system displays the document details for confirmation.
5. The patron clicks the 'OK' button to confirm the checkout.	6. The system (updates the database and) displays a checkout complete message.
7. This use case ends when the patron clicks the 'OK' button on the check out complete dialog.	

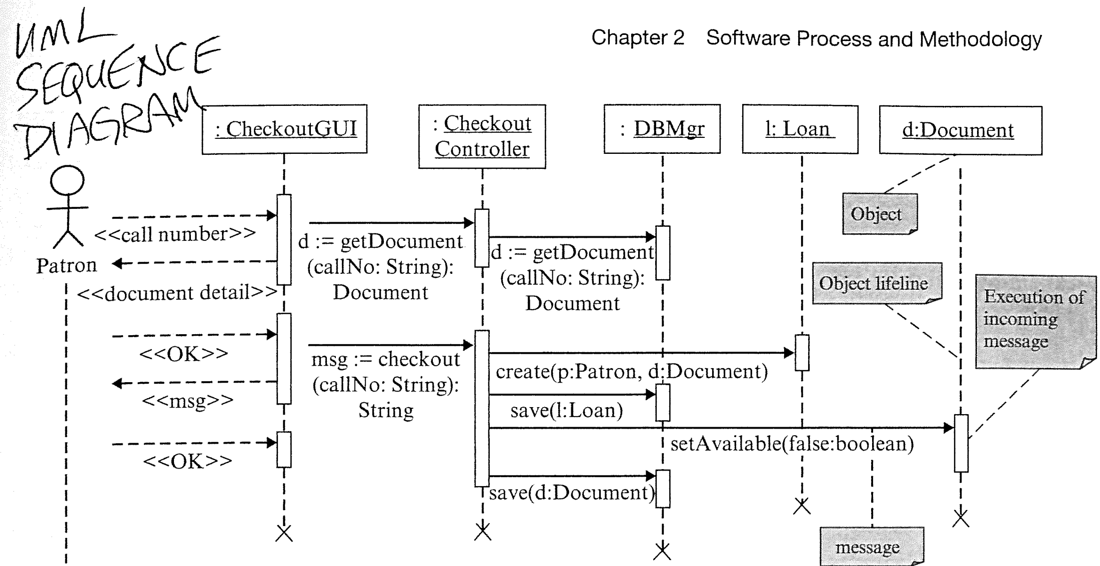


FIGURE 2.20 Object interaction modeling for Checkout Document

use case. The left column of the tabular specification shows the user input and user actions while the right column displays the corresponding system responses.

Perform Behavioral Modeling

Actor-system interaction modeling does not specify the computation or algorithms that process the user requests. This is the concern of behavioral modeling. Consider, for example, in step 3 of Figure 2.19, the patron enters the call number of the document to be checked out. To show the result in step 4, the system has to process the patron request. This involves background processing performed by the software objects. Behavioral modeling describes how the objects interact with each other to process the request. In illustration, the sequence diagram in Figure 2.20 shows how the request is processed. In particular, the rectangles across the top of the sequence diagram represent the objects that participate in the interaction. The vertical dash lines represent the lifelines of the objects. That is, the objects are created and exist in the system. The solid arrow lines represent function calls between the objects. The dashed arrow lines represent the interaction between the actor and the system. The diagram in Figure 2.20 may be interpreted as follows. It begins with the Patron submitting the document's "call number" to the Checkout GUI object, which relays the request by calling the getDocument(...) function of the Checkout Controller object. The latter retrieves the Document object from the database. It also creates a Loan object and saves it to the database. Finally, it updates and saves the Document object.

Derive Design Class Diagram

Behavioral modeling and design produce sequence diagrams, state diagrams, and activity diagrams. These diagrams specify the design of the algorithms that implement the business processes. It is not convenient to implement the classes from these diagrams because a class may appear in several diagrams. For example, the Document class appears in the Checkout Document and Return Document sequence diagrams.

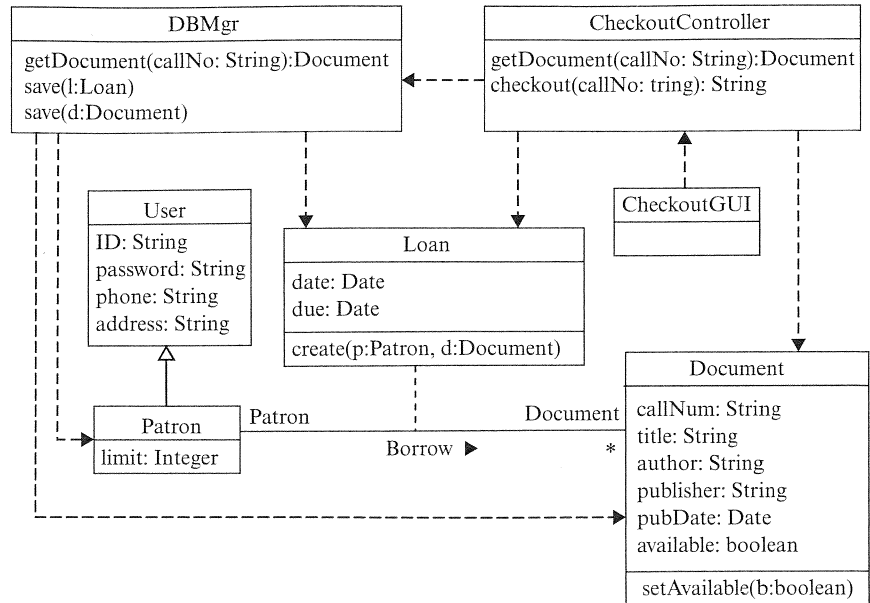


FIGURE 2.21 A design class diagram with implementation order

The software team wants an integrated view of the design. This integrated view is created by using a UML class diagram, called a *design class diagram*. In illustration, Figure 2.21 depicts a design class diagram that is derived from the sequence diagram in Figure 2.20. The dashed arrow lines in the design class diagram represent uses or dependence relationships between the classes.

Implementation, Integration Testing, and Deployment

During test-driven development (TDD), the classes that implement the use cases of the current iteration are implemented and tested. The classes are integrated and tested to ensure that they work with each other. Finally, the software system that implements the use cases of the current iteration is installed and tested in the target environment.

SUMMARY

Software development faces project as well as product challenges. To take on these challenges, a software process is needed. However, the conventional waterfall process is associated with a number of problems. The reason is that it is a process for solving tame problems but application software development in general is a wicked problem. The quest for a better process leads to the creation of a number of software process models with agile processes as the latest member of the club.

While a software process specifies the phased activities, a methodology describes the steps or how to carry out each of the activities of a process. The differences between a process and a methodology are discussed in this chapter. A software methodology is influenced by the software paradigm adopted to develop the software system. This is because a paradigm determines how the development team views the real world and systems. This, in turn, determines the basic building blocks of the software

system and the development methodology. For example, the OO paradigm views the world and systems as consisting of interrelated and interacting objects. This implies that the building blocks of an OO system are objects. Thus, an OO software development methodology must describe how to perform OO modeling, analysis, design, implementation, and testing activities.

The advent of agile processes and methods reflects a significant advance in recognizing the wickedness of software development. The agile manifesto, agile principles, agile processes, and agile methods are all designed to tackle software development as a wicked problem. Although agility is rapidly increasing its popularity in the software industry, conventional plan-driven development approaches will stay for a long period of time because these approaches also have their home grounds in the software industry. Thus, the so-called adaptive approaches, which could be tailored to fit agile, or plan-driven development, are attractive.

FURTHER READING

The spiral model, the personal software process, team software process, and the Unified Process are described in [32, 86, 87, 91], respectively. The classical OO methodology OMT is described in [131], which defines many of the UML notations. The methodology is revised by Blaha and Rumbaught to use UML 2.0 [30]. An excellent presentation of OO analysis and design using UML and patterns is found in [104].

The agile methods presented in this chapter are detailed in [21, 52, 119, 140]. Other agile methods include Crystal Orange [50], Lean Development [124], Adaptive Software Development [80], and agile modeling [11]. Abrahamsson et al. [2] provide an excellent briefing and comparison of the various agile methods. In [15], Avison and Fitzgerald review the history of software development methodologies and suggest a number of future directions. The proceedings of the annual Agile Development Conference publish papers on various aspects of agile development. Other papers of interest include [35, 69, 88, 126, 144, 145].

CHAPTER REVIEW QUESTIONS

1. What is a software process, and what are the process models presented in this chapter?
2. What are the strengths and weaknesses of the waterfall process?
3. What is a software development methodology? What are the differences between a process and a methodology?
4. What are agile processes and agile methods? What are the life-cycle activities of the agile methods presented in this chapter?
5. What are the properties of tame problems and wicked problems, respectively?
6. Why is software development in general a wicked problem?
7. How do agile processes tackle software development as a wicked problem?
8. Will agile development replace the conventional approaches such as the waterfall process?

EXERCISES

2.1 What are the similarities and differences between the conventional waterfall model and the Unified Process model? Identify and explain three advantages

2.2 Write an essay about how a good process and a good methodology help tackling the project and product challenges. Limit the length of the essay to five pages,

2.3 Write a brief essay on the differences between a software process and a software methodology.

2.4 Write an essay that discusses the following two issues:

a. The pros and cons of plan-driven development and agile development processes, respectively.

b. Whether and why agile development will, or will not, replace plan-driven approaches.

2.5 Write a short article that answers the following questions:

a. What are the similarities and differences between the spiral process, the Unified Process, and an agile process.

b. What are the pros and cons of each of these processes.

c. Which types of projects should apply which of these processes?

2.6 Explain in an essay why the waterfall process is a process for solving tame problems.

2.7 Explain in an essay how agile development tackles application software development as a wicked problem.