**THE SYSTEMS DEVELOPMENT LIFE CYCLE**

from **chapter 1** DWT 5th ed

A methodology is a formalized approach to implementing the SDLC (i.e., it is a list of steps and deliverables). There are many different systems development methodologies, and each one is unique, based on the order and focus it places on each SDLC phase. Some methodologies are formal standards used by government agencies, whereas others have been developed by consulting firms to sell to clients. Many organizations have internal methodologies that have been honed over the years, and they explain exactly how each phase of the SDLC is to be performed in that company.

There are many ways to categorize methodologies. One way is by looking at whether they focus on business processes or the data that support the business. A process-centered methodology emphasizes process models as the core of the system concept. In Figure 1-1, for example, process-centered methodologies would focus first on defining the processes (e.g., assemble sandwich ingredients). Data-centered methodologies emphasize data models as the core of the system concept. In Figure 1-1, data-centered methodologies would focus first on defining the contents of the storage areas (e.g., refrigerator) and how the contents were organized.2 By contrast, object-oriented methodologies attempt to balance the focus between process and data by incorporating both into one model. In Figure 1-1, these methodologies would focus first on defining the major elements of the system (e.g., sandwiches, lunches) and look at the processes and data involved with each element.

Another important factor in categorizing methodologies is the sequencing of the SDLC phases and the amount of time and effort devoted to each.3 In the early days of computing, programmers did not understand the need for formal and well-planned life-cycle methodologies. They tended to move directly from a very simple planning phase right into the construction step of the implementation phase — in other words, from a very fuzzy, not-well-thought-out system request into writing code. This is the same approach that you sometimes use when writing programs for a programming class. It can work for small programs that require only one programmer, but if the requirements are complex or unclear, you might miss important aspects of the problem and have to start all over again, throwing away part of the program (and the time and effort spent writing it). This approach also makes teamwork difficult because members have little idea about what needs to be accomplished and how to work together to produce a final product. In this section, we describe three different classes of system development methodologies: structured design, rapid application development, and agile development.

**Structured Design**

The first category of systems development methodologies is called structured design. These methodologies became dominant in the 1980s, replacing the previous ad hoc and

undisciplined approach. Structured design methodologies adopt a formal step-by-step approach to the SDLC that moves logically from one phase to the next. Numerous pro¬cess-centered and data-centered methodologies follow the basic approach of the two struc¬tured design categories outlined next.

**Waterfall Development**

The original structured design methodology (still used today) is waterfall development. With waterfall development-based methodologies, the analysts and users proceed in sequence from one phase to the next (see Figure 1-2). The key deliverables for each phase are typically very long (often hundreds of pages in length) and are presented to the project sponsor for approval as the project moves from phase to phase. Once the sponsor approves the work that was conducted for a phase, the phase ends and the next one begins. This methodology is referred to as waterfall development because it moves forward from phase to phase in the same manner as a waterfall. Although it is possible to go backward in the SDLC (e.g., from design back to analysis), it is extremely difficult (imagine yourself as a salmon trying to swim upstream against a waterfall, as shown in Figure 1-2).

Structured design also introduced the use of formal modeling or diagramming techniques to describe the basic business processes and the data that support them. Traditional structured design uses one set of diagrams to represent the processes and a separate set of diagrams to represent data. Because two sets of diagrams are used, the systems analyst must decide which set to develop first and use as the core of the system: process-model diagrams or data-model diagrams.

The two key advantages of the structured design waterfall approach are that it identifies system requirements long before programming begins and it minimizes changes to the requirements as the project proceeds. The two key disadvantages are that the design must be completely specified before programming begins and that a long time elapses between the completion of the system proposal in the analysis phase and the delivery of the system (usually many months or years). If the project team misses important requirements, expensive post-implementation programming may be needed (imagine yourself trying to design a car on paper; how likely would you be to remember interior lights that come on when the doors open or to specify the right number of valves on the engine?). A system can also require significant rework because the business environment has changed from the time when the analysis phase occurred.

**Parallel Development**

Parallel development methodology attempts to address the problem of long delays between the analysis phase and the delivery of the system. Instead of doing design and implementation in sequence, it performs a general design for the whole system and then divides the project into a series of distinct subprojects that can be designed and implemented in parallel. Once all subprojects are complete, the separate pieces are integrated and the system is delivered (see Figure 1-3).

The primary advantage of this methodology is that it can reduce the time to deliver a system; thus, there is less chance of changes in the business environment causing rework. However, sometimes the subprojects are not completely independent; design decisions made in one subproject can affect another, and the end of the project can require significant integration efforts.

**Rapid Application Development (RAD)**

A second category of methodologies includes rapid application development (.RADj-based methodologies. These are a newer class of systems development methodologies that emerged in the 1990s. RAD-based methodologies attempt to address both weaknesses of structured design methodologies by adjusting the SDLC phases to get some part of the system developed quickly and into the hands of the users. In this way, the users can better understand the system and suggest revisions that bring the system closer to what is needed.4

Most RAD-based methodologies recommend that analysts use special techniques and computer tools to speed up the analysis, design, and implementation phases, such as computer-aided software engineering (CASE) tools, joint application design (JAD) sessions, fourth-generation or visual programming languages that simplify and speed up programming, and code generators that automatically produce programs from design specifications. The combination of the changed SDLC phases and the use of these tools and techniques improves the speed and quality of systems development. However, there is one possible subtle problem with RAD-based methodologies: managing user expectations. Owing to the use of the tools and techniques that can improve the speed and quality of systems development, user expectations of what is possible can change dramatically. As a user better understands the information technology (IT), the systems requirements tend to expand. This was less of a problem when using methodologies that spent a lot of time thoroughly documenting requirements.

**Phased Development**

A phased development-based methodology breaks an overall system into a series of versions that are developed sequentially. The analysis phase identifies the overall system concept, and the project team, users, and system sponsor then categorize the requirements into a series of versions. The most important and fundamental requirements are bundled into the first version of the system. The analysis phase then leads into design and implementation—but only with the set of requirements identified for version 1 (see Figure 1-4).

Once version1 is implemented, work begins on version 2. Additional analysis is performed based on the previously identified requirements and combined with new ideas and issues that arose from the users' experience with version 1. Version 2 then is designed and implemented, and work immediately begins on the next version. This process continues until the system is complete or is no longer in use.

Phased development-based methodologies have the advantage of quickly getting a useful system into the hands of the users. Although the system does not perform all the functions the users need at first, it does begin to provide business value sooner than if the system were delivered after completion, as is the case with the waterfall and parallel methodologies. Likewise, because users begin to work with the system sooner, they are more likely to identify important additional requirements sooner than with structured design situations.

The major drawback to phased development is that users begin to work with systems that are intentionally incomplete. It is critical to identify the most important and useful features and include them in the first version and to manage users' expectations along the way.

**Prototyping**

A *prototyping*-based methodology performs the analysis, design, and implementation phases concurrently, and all three phases are performed repeatedly in a cycle until the system is completed. With these methodologies, the basics of analysis and design are performed, and work immediately begins on a *system prototype,* a quick-and-dirty program that provides a minimal amount of features. The first prototype is usually the first part of the system that is used. This is shown to the users and the project sponsor, who provide comments. These comments are used to reanalyze, redesign, and reimplement a second prototype, which provides a few more features. This process continues in a cycle until the analysts, users, and sponsor agree that the prototype provides enough functionality to be installed and used in the organization. After the prototype (now called the “system”) is installed, refinement occurs until it is accepted as the new system (see Figure 1-5).

The key advantage of a prototyping-based methodology is that it *very* quickly provides a system with which the users can interact, even if it is not ready for widespread organizational use at first. Prototyping reassures the users that the project team is working on the system (there are no long delays in which the users see little progress), and prototyping helps to more quickly refine real requirements.

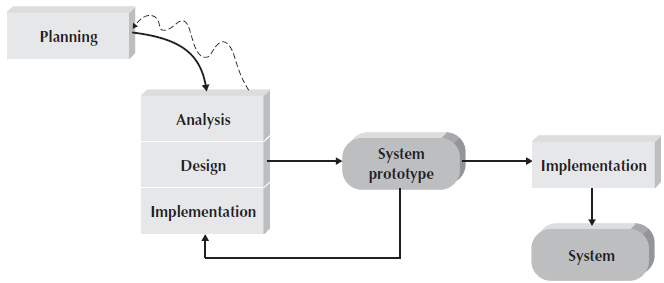


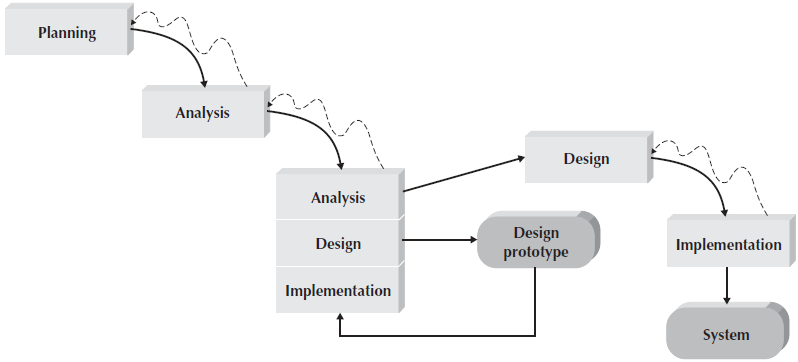
FIGURE 1-5 A Prototyping-Based Methodology

The major problem with prototyping is that its fast-paced system releases challenge attempts to conduct careful, methodical analysis. Often the prototype undergoes such significant changes that many initial design decisions become poor ones. This can cause problems in the development of complex systems because fundamental issues and problems are not recognized until well into the development process. Imagine building a car and discovering late in the prototyping process that you have to take the whole engine out to change the oil (because no one thought about the need to change the oil until after it had been driven 10,000 miles).

**Throwaway Prototyping** *Throwaway prototyping*-based methodologies are similar to prototyping-based methodologies in that they include the development of prototypes; however, throwaway prototypes are done at a different point in the SDLC. These prototypes are used for a very different purpose than those previously discussed, and they have a very different appearance (see Figure 1-6).

The throwaway prototyping-based methodologies have a relatively thorough analysis phase that is used to gather information and to develop ideas for the system concept. However, users might not completely understand many of the features they suggest, and there may be challenging technical issues to be solved. Each of these issues is examined by analyzing, designing, and building a *design prototype.* A design prototype is not a working system; it is a product that represents a part of the system that needs additional refinement, and it contains only enough detail to enable users to understand the issues under consideration. For example, suppose users are not completely clear on how an order-entry system should work. In this case, a series of mock-up screens *appear* to be a system, but they really do nothing. Or suppose that the project team needs to develop a sophisticated graphics program in Java. The team could write a portion of the program with pretend data to ensure that they could do a full-blown program successfully.

A system developed using this type of methodology relies on several design prototypes during the analysis and design phases. Each of the prototypes is used to minimize the risk associated with the system by confirming that important issues are understood before the real system is built. Once the issues are resolved, the project moves into design and implementation. At this point, the design prototypes are thrown away, which is an important difference between these methodologies and prototyping methodologies, in which the prototypes evolve into the final system.



**FIGURE 1-6** A Throwaway Prototyping-Based Methodology

Throwaway prototyping-based methodologies balance the benefits of well-thought-out analysis and design phases with the advantages of using prototypes to refine key issues before a system is built. It can take longer to deliver the final system as compared to prototyping-based methodologies, but this type of methodology usually produces more stable and reliable systems.

**Agile Development5**

A third category of systems development methodologies is still emerging today: agile development. All agile development methodologies are based on the agile manifesto and a set of twelve principles. The emphasis of the manifesto is to focus the developers on the working conditions of the developers, the working software, the customers, and addressing changing requirements instead of focusing on detailed systems development processes, tools, all-inclusive documentation, legal contracts, and detailed plans. These programming-centric methodologies have few rules and practices, all of which are fairly easy to follow. These methodologies are typically based only on the twelve principles of agile software. These principles include the following:

■ Software is delivered early and continuously through the development process, satis¬fying the customer.

■ Changing requirements are embraced regardless of when they occur in the develop¬ment process.

■ Working software is delivered frequently to the customer.

■ Customers and developers work together to solve the business problem.

■ Motivated individuals create solutions; provide them the tools and environment they need, and trust them to deliver.

■ Face-to-face communication within the development team is the most efficient and effective method of gathering requirements.

■ The primary measure of progress is working, executing software.

■ Both customers and developers should work at a pace that is sustainable. That is, the level of work could be maintained indefinitely without any worker burnout.

■ Agility is heightened through attention to both technical excellence and good design.

■ Simplicity, the avoidance of unnecessary work, is essential.

■ Self-organizing teams develop the best architectures, requirements, and designs.

■ Development teams regularly reflect on how to improve their development processes.

Based on these principles, agile methodologies focus on streamlining the system-development process by eliminating much of the modeling and documentation overhead and the time spent on those tasks. Instead, projects emphasize simple, iterative application development.6 All agile development methodologies follow a simple cycle through the traditional phases of the systems development process (see Figure 1-7). Virtually all agile methodologies are used in conjunction with object-oriented technologies.

However, agile methodologies do have critics. One of the major criticisms deals with today's business environment, where much of the actual information systems development is offshored, outsourced, and/or subcontracted. Given agile development methodologies requiring co-location of the development team, this seems to be a very unrealistic assumption. A second major criticism is that if agile development is not carefully managed, and by definition it is not, the development process can devolve into a prototyping approach that essentially becomes a "programmers gone wild" environment where programmers attempt to hack together solutions. A third major criticism, based on the lack of actual documentation created during the development of the software, raises issues regarding the auditability of the systems being created. Without sufficient documentation, neither the system nor the systems-development process can be assured. A fourth major criticism is based on whether agile approaches can deliver large mission-critical systems.

Even with these criticisms, given the potential for agile approaches to address the application backlog and to provide timely solutions to many business problems, agile approaches should be considered in some circumstances. Furthermore, many of the techniques encouraged by attending to the underlying purpose of the agile manifesto and the set of twelve agile principles are very useful in object-oriented systems development. Two of the more popular examples of agile development methodologies are extreme programming (XP) and Scrum.

**Extreme Programming7**

Extreme programming (XP) is founded on four core values: communication, simplicity, feedback, and courage. These four values provide a foundation that XP developers use to create any system. First, the developers must provide rapid feedback to the end users on a continuous basis. Second, XP requires developers to follow the KISS principle.8 Third, developers must make incremental changes to grow the system, and they must not only accept change, they must embrace change. Fourth, developers must have a quality-first mentality. XP also supports team members in developing their own skills. Three of the key principles that XP uses to create successful systems are continuous testing, simple coding performed by pairs of developers, and close interactions with end users to build systems very quickly.

Testing and efficient coding practices are the core of XP. Code is tested each day and is placed into an integrative testing environment. If bugs exist, the code is backed out until it is completely free of errors.

An XP project begins with user stories that describe what the system needs to do. Then, programmers code in small, simple modules and test to meet those needs. Users are required to be available to clear up questions and issues as they arise. Standards are very important to minimize confusion, so XP teams use a common set of names, descriptions, and coding practices. XP projects deliver results sooner than even the RAD approaches, and they rarely get bogged down in gathering requirements for the system. XP adherents claim many strengths associated with developing software using XP. Programmers work closely with all stakeholders, and communication among all stakeholders is improved. Continuous testing of the evolving system is encouraged. The system is developed in an evolutionary and incremental manner, which allows the requirements to evolve as the stakeholders understand the potential that the technology has in providing a solution to their problem. Estimation is task driven and is performed by the programmer who will implement the solution for the task under consideration. Because all programming is done in pairs, a shared responsibility for each software component develops among the programmers. Finally, the quality of the final product increases during each iteration.

For small projects with highly motivated, cohesive, stable, and experienced teams, XP should work just fine. However, if the project is not small or the teams aren't jelled,9 the success of an XP development effort is doubtful. This tends to throw into doubt the whole idea of bringing outside contractors into an existing team environment using XP.10 The chance of outsiders jelling with insiders might simply be too optimistic. XP requires a great deal of discipline, otherwise projects will become unfocused and chaotic. XP is recommended only for small groups of developers—no more than ten developers—and it is not advised for large mission-critical applications. Owing to the lack of analysis and design documentation, there is only code documentation associated with XP, so maintaining large systems built with XP may be impossible. And because mission-critical business information systems tend to exist for a long time, the utility of XP as a business information system development methodology is in doubt. Finally, the methodology needs a lot of on-site user input, something to which many business units cannot commit.11 However, some of the techniques associated with XP are useful in object-oriented systems development. For example, user stories, pair programming, and continuous testing are invaluable tools from which object-oriented systems development could benefit.

**Scrum12**

Scrum is a term that is well known to rugby fans. In rugby, a scrum is used to restart a game. In a nutshell, the creators of the Scrum method believe that no matter how much you plan, as soon as the software begins to be developed, chaos breaks out and the plans go out the window.13 The best you can do is to react to where the proverbial rugby ball squirts out. You then sprint with the ball until the next scrum. In the case of the Scrum methodology, a sprint lasts thirty working days. At the end of the sprint, a system is delivered to the customer.

Of all systems development approaches, on the surface, Scrum is the most chaotic. To control some of the innate chaos, Scrum development focuses on a few key practices. Teams are self-organized and self-directed. Unlike other approaches, Scrum teams do not have a designated team leader. Instead, teams organize themselves in a symbiotic manner and set their own goals for each sprint (iteration). Once a sprint has begun, Scrum teams do not consider any additional requirements. Any new requirements that are uncovered are placed on a backlog of requirements that still need to be addressed. At the beginning of every workday, a Scrum meeting takes place. At the end of each sprint, the team demonstrates the software to the client. Based on the results of the sprint, a new plan is begun for the next sprint.

Scrum meetings are one of the most interesting aspects of the Scrum development process. The team members attend the meetings, but anyone can attend. However, with very few exceptions, only team members may speak. One prominent exception is management providing feedback on the business relevance of the work being performed by the specific team. In this meeting, all team members stand in a circle and report on what they accomplished during the previous day, state what they plan to do today, and describe anything that blocked progress the previous day. To enable continuous progress, any block identified is dealt with within one hour. From a Scrum point of view, it is better to make a "bad" decision about a block at this point in development than to not make a decision. Because the meetings take place each day, a bad decision can easily be undone. Larman14 suggests that each team member should report any additional requirements that have been uncovered during the sprint and anything that the team member learned that could be useful for other team members to know.

One of the major criticisms of Scrum, as with all agile methodologies, is that it is questionable whether Scrum can scale up to develop very large, mission-critical systems. A typical Scrum team size is no more than seven members. The only organizing principle put forth by Scrum followers to address this criticism is to organize a scrum of scrums. Each team meets every day, and after the team meeting takes place, a representative (not leader) of each team attends a scrum-of-scrums meeting. This continues until the progress of entire system has been determined.

Depending on the number of teams involved, this approach to managing a large project is doubtful. However, as in XP and other agile development approaches, many of the ideas and techniques associated with Scrum development are useful in object-oriented systems development, such as the focus of a Scrum meeting, the evolutionary and incremental approach to identifying requirements, and the incremental and iterative approach to the development of the system

2 The classic modern process-centered methodology is that by Edward Yourdon, Modern Structured Analysis (Englewood Cliffs, NJ: Yourdon Press, 1989). An example of a data-centered methodology is information engi¬neering; see James Martin, Information Engineering, vols. 1-3 (Englewood Cliffs, NJ: Prentice Hall, 1989). A widely accepted standardized non-object-oriented methodology that balances processes and data is IDEF; see FIPS 183, Integration Definition for Function Modeling, Federal Information Processing Standards Publications, U.S. Department of Commerce, 1993.

3 A good reference for comparing systems development methodologies is Steve McConnell, Rapid Development (Redmond, WA: Microsoft Press, 1996).

4 One of the best RAD books is Steve McConnell, Rapid Development (Redmond, WA: Microsoft Press, 1996).

5 Three good sources of information on agile development and object-oriented systems are S. W. Ambler, Agile Modeling: Effective Practices for Extreme Programming and the Unified Process (New York: Wiley, 2002); C. Larman, Agile & Iterative Development: A Manager's Guide (Boston: Addison-Wesley, 2004); R. C. Martin, Agile Software Development: Principles, Patterns, and Practices (Upper Saddle River, NJ: Prentice Hall, 2003).

6 See Agile Alliance, www.agilealliance.com.

7 For more information, see K. Beck, eXtreme Programming Explained: Embrace Change (Reading, MA: Addison-Wesley, 2000); C. Larman, Agile & Iterative Development: A Manager's Guide (Boston: Addison-Wesley, 2004); M. Lippert, S. Roock, and H. Wolf, eXtreme Programming in Action: Practical Experiences from Real World Projects (New York: Wiley, 2002); www.extremeprogramming.org.

8 Keep it simple, stupid.

9 A jelled team is one that has low turnover, a strong sense of identity, a sense of eliteness, a feeling that they jointly own the product being developed, and enjoyment in working together. For more information regarding jelled teams, see T. DeMarco and T. Lister, Peopleware: Productive Projects and Teams (New York: Dorset/House, 1987).

10 Considering the tendency for offshore outsourcing, this is a major obstacle for XP to overcome. For more information on offshore outsourcing, see P. Thibodeau, "ITAA Panel Debates Outsourcing Pros, Cons," Computerworld Morning Update (September 25, 2003); S. W. Ambler, "Chicken Little Was Right," Software Development (October 2003).

11 Many of the observations on the utility of XP as a development approach were based on conversations with Brian Henderson-Sellers.

12 For more information, see C. Larman, Agile & Iterative Development: A Manager s Guide (Boston: Addison-Wesley, 2004); K. Schwaber and M. Beedle, Agile Software Development with Scrum (Upper Saddle River, NJ: Prentice Hall, 2001); R. Wysocki, Effective Project Management: Traditional, Agile, Extreme, 5th Ed. (Indianapolis, IN: Wiley Publishing, 2009).