Chapter 2

Software Life-Cycle Models

Learning Objectives

After studying this chapter, you should be able to

- Describe how software products are developed in practice.
- Understand the evolution-tree life-cycle model.
- Appreciate the negative impact of change on software products.
- Utilize the iterative-and-incremental life-cycle model.
- Comprehend the impact of Miller's Law on software production.
- Describe the strengths of the iterative-and-incremental life-cycle model.
- Realize the importance of mitigating risks early.
- Describe agile processes, including extreme programming.
- Compare and contrast a variety of other life-cycle models.

Chapter 1 describes how software products would be developed in an ideal world. The theme of this chapter is what happens in practice. As will be explained, there are vast differences between theory and practice.

2.1 Software Development in Theory

In an ideal world, a software product is developed as described in Chapter 1. As depicted schematically in Figure 2.1, the system is developed from scratch; $\varnothing$ denotes the empty set. (See Just in Case You Wanted to Know Box 2.1 if you want to know the origin of the term *from scratch*.) First the client's Requirements are determined, and then the Analysis

Just in Case You Wanted to Know — Box 2.1

The term *from scratch*, meaning "starting with nothing," comes from 19th century sports terminology. Before roads (and running tracks) were paved, races had to be held on open ground. In many cases, the starting line was a scratch in the sand. A runner who had no advantage or handicap had to start from that line, that is, "from [the] scratch."

The term *scratch* has a different sporting connotation nowadays. A "scratch golfer" is one whose golfing handicap is zero.

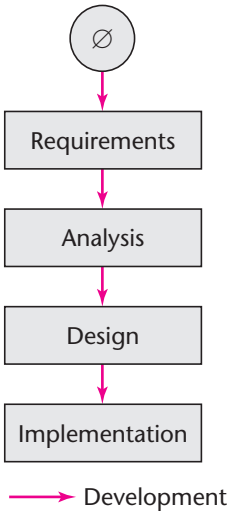


FIGURE 2.1 Idealized software development.

is performed. When the analysis artifacts are complete, the Design is produced. This is followed by the Implementation of the complete software product, which is then installed on the client's computer.

However, software development is considerably different in practice for two reasons. First, software professionals are human and therefore make mistakes. Second, the client's requirements can change while the software is being developed. In this chapter, both these issues are discussed in some depth, but first we present a mini case study, based on the case study in [Tomer and Schach, 2000], that illustrates the issues involved.

Mini Case Study

2.2 Winburg Mini Case Study

To reduce traffic congestion in downtown Winburg, Indiana, the mayor convinces the city to set up a public transportation system. Bus-only lanes are to be established, and commuters will be encouraged to "park and ride"; that is, to park their cars in suburban parking lots and then take buses from there to work and back at a cost of one dollar per ride. Each bus is to have a fare machine that accepts only dollar bills. Passengers insert a bill into the slot as they enter the bus. Sensors inside the fare machine scan the bill, and the software in the machine uses an image recognition

algorithm to decide whether the passenger has indeed inserted a valid dollar bill into the slot. It is important that the fare machine be accurate because, once the news gets out that any piece of paper will do the trick, fare income will plummet to effectively zero. Conversely, if the machine regularly rejects valid dollar bills, passengers will be reluctant to use the buses. In addition, the fare machine must be rapid. Passengers will be equally reluctant to use the buses if the machine spends 15 seconds coming to a decision regarding the validity of a dollar bill—it would take even a relatively small number of passengers many minutes to board a bus. Therefore, the requirements for the fare machine software include an average response time of less than 1 second and an average accuracy of at least 98 percent.

Episode 1 The first version of the software is implemented.

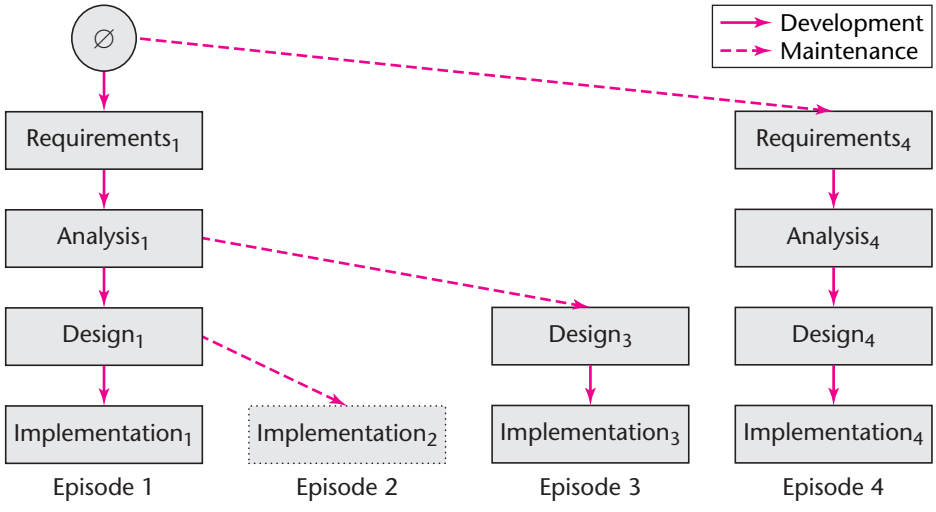
Episode 2 Tests show that the required constraint of an average response time of 1 second for deciding on the validity of a dollar bill is not achieved. In fact, on average, it takes 10 seconds to get a response. Senior management discovers the cause. It seems that, to get the required 98 percent accuracy, a programmer has been instructed by her manager to use double-precision numbers for all mathematical calculations. As a result, every operation takes at least twice as long as it would with the usual single-precision numbers. The result is that the program is much slower than it should be, resulting in the long response time. Calculations then show that, despite what the manager told the programmer, the stipulated 98 percent accuracy can be attained even if single-precision numbers are used. The programmer starts to make the necessary changes to the implementation.

Episode 3 Before the programmer can complete her work, further tests of the system show that, even if the indicated changes to the implementation were made, the system would still have an average response time of over 4.5 seconds, nowhere near the stipulated 1 second. The problem is the complex image recognition algorithm. Fortunately, a faster algorithm has just been discovered, so the fare machine software is redesigned and reimplemented using the new algorithm. This results in the average response time being successfully achieved.

Episode 4 By now, the project is considerably behind schedule and way over budget. The mayor, a successful entrepreneur, has the bright idea of asking the software development team to try to increase the accuracy of the dollar bill recognition component of the system as much as possible, to sell the resulting package to vending machine companies. To meet this new requirement, a new design is adopted that improves the average accuracy to over 99.5 percent. Management decides to install that version of the software in the fare machines. At this point, development of the software is complete. The city is later able to sell its system to two small vending machine companies, defraying about one-third of the cost overrun.

Epilogue A few years later, the sensors inside the fare machine become obsolete and need to be replaced by a newer model. Management suggests taking advantage of the change to upgrade the hardware at the same time. The software professionals point out that changing the hardware means that new software also is needed. They suggest reimplementing the software in a different programming language. At the

FIGURE 2.2 The evolution-tree life-cycle model for the Winburg mini case study. (The rectangle drawn with a dotted line denotes the implementation that was not completed.)



time of writing, the project is 6 months behind schedule and 25 percent over budget. However, everyone involved is confident that the new system will be more reliable and of higher quality, despite "minor discrepancies" in meeting its response time and accuracy requirements.

Figure 2.2 depicts the **evolution-tree life-cycle model** of the mini case study. The leftmost boxes represent Episode 1. As shown in the figure, the system was developed from scratch (∅). The requirements (Requirements_1), analysis (Analysis_1), design (Design_1), and implementation (Implementation_1) followed in turn. Next, as previously described, trials of the first version of the software showed that the average response time of 1 second could not be achieved and the implementation had to be modified. The modified implementation appears in Figure 2.2 as Implementation_2. However, Implementation_2 was never completed. That is why the rectangle representing Implementation_2 is drawn with a dotted line.

In Episode 3, the design had to be changed. Specifically, a faster image recognition algorithm was used. The modified design (Design_3) resulted in a modified implementation (Implementation_3).

Finally, in Episode 4, the requirements were changed (Requirements_4) to increase the accuracy. This resulted in modified specifications (Analysis_4), modified design (Design_4), and modified implementation (Implementation_4).

In Figure 2.2, the solid arrows denote development and the dashed arrows denote maintenance. For example, when the design is changed in Episode 3, Design_3 replaced Design_1 as the design of Analysis_1.

The evolution-tree model is an example of a **life-cycle model** (or **model**, for short), that is, the series of steps to be performed while the software product is developed and maintained. Another life-cycle model that can be used for the mini

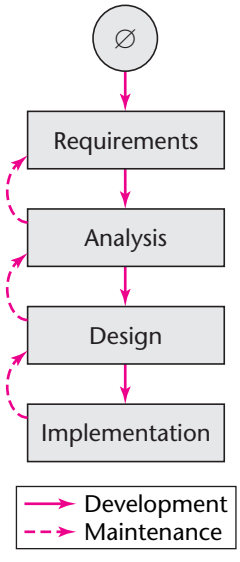


FIGURE 2.3
A simplified version of the waterfall life-cycle model.

case study is the **waterfall life-cycle model** [Royce, 1970]; a simplified version of the waterfall model is depicted in Figure 2.3. This classical life-cycle model can be viewed as the linear model of Figure 2.1 with feedback loops. Then, if a fault is found during the design that was caused by a fault in the requirements, following the dashed upward arrows, the software developers can backtrack from the design up to the analysis and hence to the requirements and make the necessary corrections there. Then, they move down to the analysis, correct the specification document to reflect the corrections to the requirements, and in turn, correct the design document. Design activities can now resume where they were suspended when the fault was discovered. Again, the solid arrows denote development; the dashed arrows, maintenance.

The waterfall model can certainly be used to represent the Winburg mini case study, but, unlike the evolution-tree model of Figure 2.2, it cannot show the order of events. The evolution-tree model has a further advantage over the waterfall model. At the end of each episode we have a **baseline**, that is, a complete set of artifacts (recall that an **artifact** is a constituent component of a software product). There are four baselines in Figure 2.2. They are

At the end of Episode 1: Requirements_1, Analysis_1, Design_1, Implementation_1
At the end of Episode 2: Requirements_1, Analysis_1, Design_1, Implementation_2
At the end of Episode 3: Requirements_1, Analysis_1, Design_3, Implementation_3
At the end of Episode 4: Requirements_4, Analysis_4, Design_4, Implementation_4

The first baseline is the initial set of artifacts; the second baseline reflects the modified (but never completed) Implementation_2 of Episode 2, together with the unchanged requirements, analysis, and design of Episode 1. The third baseline is the same as the first baseline but with the design and implementation changed. The fourth baseline is the complete set of new artifacts shown in Figure 2.2. We revisit the concept of a baseline in Chapters 5 and 16.

2.3 Lessons of the Winburg Mini Case Study

The Winburg mini case study depicts the development of a software product that goes awry for a number of unrelated causes, such as a poor implementation strategy (the unnecessary use of double-precision numbers) and the decision to use an algorithm that was too slow. In the end, the project was a success. However, the obvious question is, Is software development really as chaotic in practice? In fact, the mini case study is far less traumatic than many, if not the majority of, software projects. In the Winburg mini case study, there were only two new versions of the software because of faults (the inappropriate use of double-precision numbers; the utilization of an algorithm that could not meet the response time requirement), and only one new version because of a change made by the client (the need for increased accuracy).

Why are so many changes to a software product needed? First, as previously stated, software professionals are human and therefore make mistakes. Second, a software product is a model of the real world, and the real world is continually changing. This issue is discussed at greater length in Section 2.4.

Mini Case Study

2.4 *Teal Tractors Mini Case Study*

Teal Tractors, Inc., sells tractors in most areas of the United States. The company has asked its software division to develop a new product that can handle all aspects of its business. For example, the product must be able to handle sales, inventory, and commissions paid to the sales staff, as well as providing all necessary accounting functions. While this software product is being implemented, Teal Tractors buys a Canadian tractor company. The management of Teal Tractors decides that, to save money, the Canadian operations are to be integrated into the U.S. operations. That means that the software has to be changed before it is completed:

1. It must be modified to handle additional sales regions.
2. It must be extended to handle those aspects of the business that are handled differently in Canada, such as taxes.
3. It must be extended to handle two different currencies, U.S. dollars and Canadian dollars.

Teal Tractors is a rapidly growing company with excellent future prospects. The takeover of the Canadian tractor company is a positive development, one that may well lead to even greater profits in future years. But, from the viewpoint of the software division, the purchase of the Canadian company could be disastrous. Unless the requirements, analysis, and design have been performed with a view to incorporating possible future extensions, the work involved in adding the Canadian sales regions may be so great that it might be more effective to discard everything done to date and start from scratch. The reason is that changing the product at this stage is similar to trying to fix a software product late in its life cycle (see Figure 1.6). Extending the software to

handle aspects specific to the Canadian market, as well as Canadian currency, may be equally hard.

Even if the software has been well thought out and the original design is indeed extensible, the design of the resulting patched-together product cannot be as cohesive as it would have been if it had been developed from the very beginning to cater to both the United States and Canada. This can have severe implications for future maintenance.

The software division of Teal Tractors is a victim of the **moving-target problem**. That is, while the software is being developed, the requirements change. It does not matter that the reason for the change is otherwise extremely worthwhile. The fact is that the takeover of the Canadian company could well be detrimental to the quality of the software being developed.

In some cases, the reason for the moving target is less benign. Sometimes a powerful senior manager within an organization keeps changing his or her mind regarding the functionality of a software product being developed. In other cases, there is **feature creep**, a succession of small, almost trivial, additions to the requirements. But whatever the reason may be, frequent changes, no matter how minor they may seem, are harmful to the health of a software product. It is important that a software product be designed as a set of components that are as independent as possible, so that a change to one part of the software does not induce a fault in an apparently unrelated part of the code, a so-called **regression fault**. When numerous changes are made, the effect is to induce dependencies within the code. Finally, there are so many dependencies that virtually any change induces one or more regression faults. At that time, the only thing that can be done is to redesign the entire software product and reimplement it.

Unfortunately, there is no known solution to the moving-target problem. With regard to positive changes to requirements, growing companies are always going to change, and these changes have to be reflected in the mission-critical software products of the company. As for negative changes, if the individual calling for those changes has sufficient clout, nothing can be done to prevent the changes being implemented, to the detriment of the further maintainability of the software product.

2.5 Iteration and Incrementation

As a consequence of both the moving-target problem and the need to correct the inevitable mistakes made while a software product is being developed, the life cycle of actual software products resembles the evolution-tree model of Figure 2.2 or the waterfall model of Figure 2.3, rather than the idealized chain of Figure 2.1. One consequence of this reality is that it does not make much sense to talk about (say) "*the* analysis phase." Instead, the operations of the analysis phase are spread out over the life cycle. Similarly, Figure 2.2 shows four different versions of the implementation, one of which (Implementation_2) was never completed because of the moving-target problem.

Consider successive versions of an artifact, for example, the specification document or a code module. From this viewpoint, the basic process is iterative. That is, we produce the first version of the artifact, then we revise it and produce the second version, and so on. Our

intent is that each version is closer to our target than its predecessor and finally we construct a version that is satisfactory. **Iteration** is an intrinsic aspect of software engineering, and iterative life-cycle models have been used for over 30 years [Larman and Basili, 2003]. For example, the waterfall model, which was first put forward in 1970, is iterative (but not incremental).

A second aspect of developing real-world software is the restriction imposed on us by **Miller's Law**. In 1956, George Miller, a professor of psychology, showed that, at any one time, we humans are capable of concentrating on only approximately seven chunks (units of information) [Miller, 1956]. However, a typical software artifact has far more than seven chunks. For example, a code artifact is likely to have considerably more than seven variables, and a requirements document is likely to have many more than seven requirements. One way we humans handle this restriction on the amount of information we can handle at any one time is to use **stepwise refinement**. That is, we concentrate on those aspects that are currently the most important and postpone until later those aspects that are currently less critical. In other words, every aspect is eventually handled but in order of current importance. This means that we start off by constructing an artifact that solves only a small part of what we are trying to achieve. Then, we consider further aspects of the problem and add the resulting new pieces to the existing artifact. For example, we might construct a requirements document by considering the seven requirements we consider the most important. Then, we would consider the seven next most important requirements, and so on. This is an incremental process. **Incrementation** is also an intrinsic aspect of software engineering; incremental software development is over 45 years old [Larman and Basili, 2003].

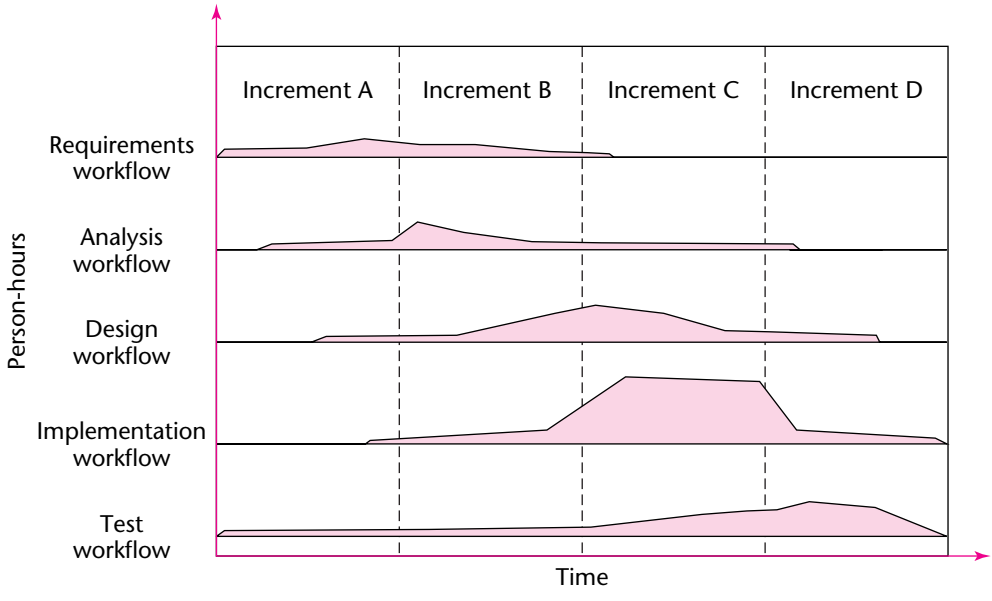
In practice, iteration and incrementation are used in conjunction with one another. That is, an artifact is constructed piece by piece (incrementation), and each increment goes through multiple versions (iteration). These ideas are illustrated in Figure 2.2, which represents the life cycle for the Winburg mini case study (Sections 2.2 and 2.3). As shown in that figure, there is no single "requirements phase" as such. Instead, the client's requirements are extracted and analyzed twice, yielding the original requirements ($\mathsf{Requirements}_1$) and the modified requirements ($\mathsf{Requirements}_4$). Similarly, there is no single "implementation phase," but rather four separate episodes in which the code is produced and then modified.

These ideas are generalized in Figure 2.4, which reflects the basic concepts underlying the **iterative-and-incremental life-cycle model** [Jacobson, Booch, and Rumbaugh, 1999]. The figure shows the development of a software product in four increments, labeled Increment A, Increment B, Increment C, and Increment D. The horizontal axis is time, and the vertical axis is person-hours (one person-hour is the amount of work that one person can do in 1 hour), so the shaded area under each curve is the total effort for that increment.

It is important to appreciate that Figure 2.4 depicts just one possible way a software product can be decomposed into increments. Another software product may be constructed in just 2 increments, whereas a third may require 14. Furthermore, the figure is not intended to be an accurate representation of precisely how a software product is developed. Instead, it shows how the emphasis changes from iteration to iteration.

The sequential phases of Figure 2.1 are artificial constructs. Instead, as explicitly reflected in Figure 2.4, we must acknowledge that different **workflows** (activities) are performed over the entire life cycle. There are five **core workflows**, the **requirements workflow**, **analysis workflow**, **design workflow**, **implementation workflow**, and **test workflow**, and, as stated in the previous sentence, all five are performed over the life

FIGURE 2.4 The construction of a software product in four increments.



cycle of a software product. However, there are times when one workflow predominates over the other four.

For example, at the beginning of the life cycle, the software developers extract an initial set of requirements. In other words, at the beginning of the iterative-and-incremental life cycle, the requirements workflow predominates. These requirements artifacts are extended and modified during the remainder of the life cycle. During that time, the other four workflows (analysis, design, implementation, and test) predominate. In other words, the requirements workflow is the major workflow at the beginning of the life cycle, but its relative importance decreases thereafter. Conversely, the implementation and test workflows occupy far more of the time of the members of the software development team toward the end of the life cycle than they do at the beginning.

Planning and documentation activities are performed throughout the iterative-and-incremental life cycle. Furthermore, testing is a major activity during each iteration, and particularly at the end of each iteration. In addition, the software as a whole is thoroughly tested once it has been completed; at that time, testing and then modifying the implementation in the light of the outcome of the various tests is virtually the sole activity of the software team. This is reflected in the test workflow of Figure 2.4.

Figure 2.4 shows four increments. Consider Increment A, depicted by the column on the left. At the beginning of this increment, the requirements team members determine the client's requirements. Once most of the requirements have been determined, the first version of part of the analysis can be started. When sufficient progress has been made with the analysis, the first version of the design can be started. Even some coding is often done during this first increment, perhaps in the form of a proof-of-concept prototype to test the feasibility of part of the proposed software product. Finally, as previously mentioned,

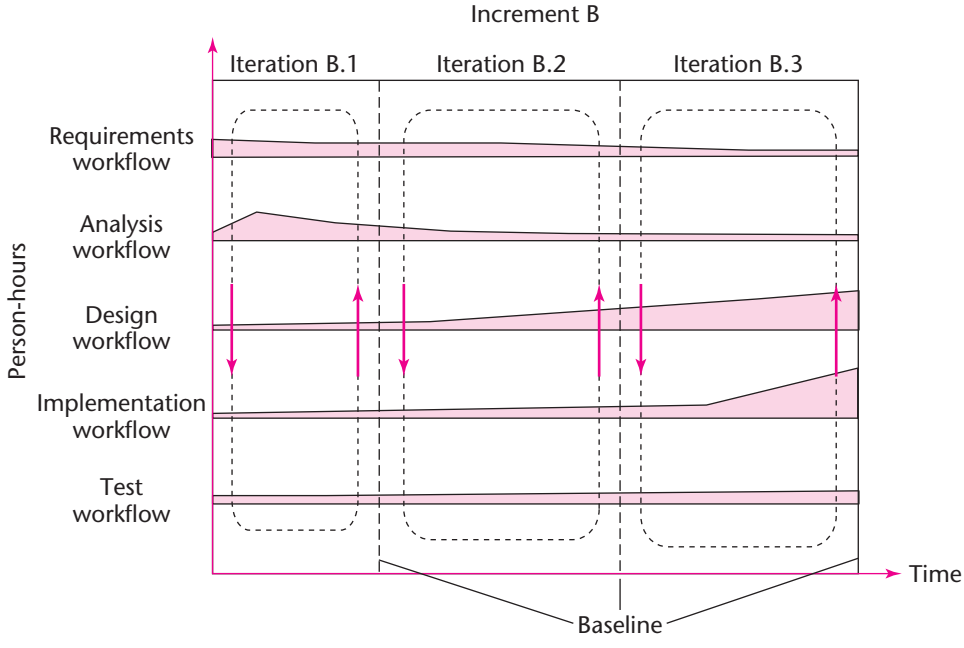


FIGURE 2.5
The three iterations of Increment B of the iterative-and-incremental life-cycle model of Figure 2.4.

planning, testing, and documentation activities start on Day One and continue from then on, until the software product is finally delivered to the client.

Similarly, the primary concentration during Increment B is on the requirements and analysis workflows, and then on the design workflow. The emphasis during Increment C is first on the design workflow, and then on the implementation workflow and test workflow. Finally, during Increment D, the implementation workflow and test workflow dominate.

As reflected in Figure 1.4, about one-fifth of the total effort is devoted to the requirements and analysis workflows (together), another one-fifth to the design workflow, and about three-fifths to the implementation workflow. The relative total sizes of the shaded areas in Figure 2.4 reflect these values.

There is iteration during each increment of Figure 2.4. This is shown in Figure 2.5, which depicts three iterations during Increment B. (Figure 2.5 is an enlarged view of the second column of Figure 2.4.) As shown in Figure 2.5, each iteration involves all five workflows but again in varying proportions.

Again, it must be stressed that Figure 2.5 is not intended to show that every increment involves exactly three iterations. The number of iterations varies from increment to increment. The purpose of Figure 2.5 is to show the iteration within each increment and repeat that all five workflows (requirements, analysis, design, implementation, and testing, together with planning and documentation) are carried out during almost every iteration, although in varying proportions each time.

As previously explained, Figure 2.4 reflects the incrementation intrinsic to the development of every software product. Figure 2.5 explicitly displays the iteration that underlies incrementation. Specifically, Figure 2.5 depicts three consecutive iterative steps, as opposed to one large incrementation. In more detail, Iteration B.1 consists of requirements,

analysis, design, implementation, and test workflows, represented by the leftmost dashed rectangle with rounded corners. The iteration continues until the artifacts of each of the five workflows are satisfactory.

Next, all five sets of artifacts are iterated in Iteration B.2. This second iteration is similar in nature to the first. That is, the requirements artifacts are improved, which in turn triggers improvements to the analysis artifacts, and so on, as reflected in the second iteration of Figure 2.5, and similarly for the third iteration.

The process of iteration and incrementation starts at the beginning of Increment A and continues until the end of Increment D. The completed software product is then installed on the client's computer.

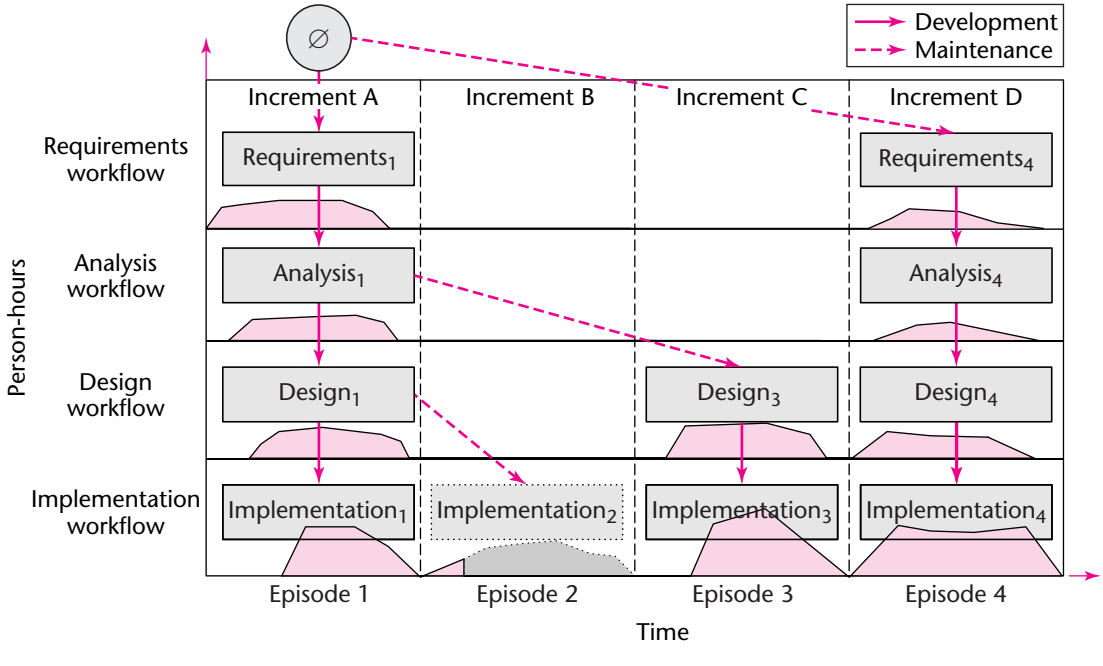
Mini Case Study

2.6 *Winburg Mini Case Study Revisited*

Figure 2.6 shows the evolution-tree model of the Winburg mini case study (Figure 2.2) superimposed on the iterative-and-incremental model (the test workflow is not shown because the evolution-tree model assumes continual testing, explained in Section 1.7). Figure 2.6 sheds additional light on the nature of incrementation:

- Increment A corresponds to Episode 1, Increment B corresponds to Episode 2, and so on.

FIGURE 2.6 The evolution-tree life-cycle model for the Winburg mini case study (Figure 2.2) superimposed on the iterative-and-incremental life-cycle model.



- From the viewpoint of the iterative-and-incremental model, two of the increments do not include all four workflows. In more detail, Increment B (Episode 2) includes only the implementation workflow, and Increment C (Episode 3) includes only the design workflow and the implementation workflow. The iterative-and-incremental model does not require that every workflow be performed during every increment.
- Furthermore, in Figure 2.4 most of the requirements workflow is performed in Increment A and Increment B, whereas in Figure 2.6 it is performed in Increment A and Increment D. Also, in Figure 2.4 most of the analysis is performed in Increment B, whereas in Figure 2.6 the analysis workflow is performed in Increment A and Increment D. This indicates that neither Figure 2.4 nor Figure 2.6 represents the way every software product is built. Instead, each figure shows the way that one particular software product is built, highlighting the underlying iteration and incrementation.
- The small size and abrupt termination of the implementation workflow during Increment B (Episode 2) of Figure 2.6 shows that Implementation_2 was not completed. The gray piece reflects the part of the implementation workflow that was not performed.
- The three dashed arrows of the evolution-tree model show that each increment constitutes maintenance of the previous increment. In this example, the second and third increments are instances of corrective maintenance. That is, each increment corrects faults in the previous increment. As previously explained, Increment B (Episode 2) corrects the implementation workflow by replacing double-precision variables with the usual single-precision variables. Increment C (Episode 3) corrects the design workflow by using a faster image recognition algorithm, thereby enabling the response time requirement to be met. Corresponding changes then have to be made to the implementation workflow. Finally, in Increment D (Episode 4) the requirements are changed to stipulate improved overall accuracy, an instance of perfective maintenance. Corresponding changes are then made to the analysis workflow, design workflow, and implementation workflow.

2.7 Risks and Other Aspects of Iteration and Incrementation

Another way of looking at iteration and incrementation is that the project as a whole is divided into smaller mini projects (or increments). Each mini project extends the requirements, analysis, design, implementation, and testing artifacts. Finally, the resulting set of artifacts constitutes the complete software product.

In fact, each mini project consists of more than just extending the artifacts. It is essential to check that each artifact is correct (the test workflow) and make any necessary changes to the relevant artifacts. This process of checking and modifying, then rechecking and remodifying, and so on, is clearly iterative in nature. It continues until the members of the

development team are satisfied with all the artifacts of the current mini project (or increment). When that happens, they proceed to the next increment.

Comparing Figure 2.3 (the waterfall model) with Figure 2.5 (view of the iterations within Increment B) shows that each iteration can be viewed as a small but complete waterfall model. That is, during each iteration the members of the development team go through the classical requirements, analysis, design, and implementation phases on a specific portion of the software product. From this viewpoint, the iterative-and-incremental model of Figures 2.4 and 2.5 can be viewed as a consecutive series of waterfall models.

The iterative-and-incremental model has many strengths:

1. Multiple opportunities are offered for checking that the software product is correct. Every iteration incorporates the test workflow, so every iteration is another chance to check all the artifacts developed up to this point. The later faults are detected and corrected, the higher is the cost, as shown in Figure 1.6. Unlike the classical waterfall model, each of the many iterations of the iterative-and-incremental model offers a further opportunity to find faults and correct them, thereby saving money.
2. The robustness of the underlying architecture can be determined relatively early in the life cycle. The **architecture** of a software product includes the various component artifacts and how they fit together. An analogy is the architecture of a cathedral, which might be described as Romanesque, Gothic, or Baroque, among other possibilities. Similarly, the architecture of a software product might be described as object-oriented (Chapter 7), pipes and filters (UNIX or Linux components), or client–server (with a central server providing file storage for a network of client computers). The architecture of a software product developed using the iterative-and-incremental model must have the property that it can be extended continually (and, if necessary, easily changed) to incorporate the next increment. Being able to handle such extensions and changes without falling apart is called **robustness**. Robustness is an important quality during development of a software product; it is vital during postdelivery maintenance. So, if a software product is to last through the usual 12, 15, or more years of postdelivery maintenance, the underlying architecture has to be robust. When an iterative-and-incremental model is used, it soon becomes apparent whether or not the architecture is robust. If, in the course of incorporating (say) the third increment, it is clear that the software developed to date has to be drastically reorganized and large parts reimplemented, then it is clear that the architecture is not sufficiently robust. The client must decide whether to abandon the project or start again from scratch. Another possibility is to redesign the architecture to be more robust, and then reuse as much of the current artifacts as possible before proceeding to the next increment. Another reason why a robust architecture is so important is the moving-target problem (Section 2.4). It is all but certain that the client's requirements will change, either because of growth within the client's organization or because the client keeps changing his or her mind as to what the target software has to do. The more robust the architecture, the more resilient to change the software will be. It is not possible to design an architecture that can cope with too many drastic changes. But, if the required changes are reasonable in scope, a robust architecture should be capable of incorporating those changes without having to be drastically restructured.

3. The iterative-and-incremental model enables us to mitigate risks early. **Risks** are invariably involved in software development and maintenance. In the Winburg mini case study, for example, the original image recognition algorithm was not fast enough; there is an ever-present risk that a completed software product will not meet its time constraints. Developing a software product incrementally enables us to mitigate such risks early in the life cycle. For example, suppose a new local area network (LAN) is being developed and there is concern that the current network hardware is inadequate for the new software product. Then, the first one or two iterations are directed toward constructing those parts of the software that interface with the network hardware. If it turns out that, contrary to the developers' fears, the network has the necessary capability, the developers can proceed with the project, confident that this risk has been mitigated. On the other hand, if the network indeed cannot cope with the additional traffic that the new LAN generates, this is reported to the client early in the life cycle, when only a small proportion of the budget has been spent. The client can now decide whether to cancel the project, extend the capabilities of the existing network, buy a new and more powerful network, or take some other action.
4. We always have a working version of the software. Suppose a software product is developed using the classical life-cycle model of Figure 2.1. Only at the very end of the project is there a working version of the software product. In contrast, when the iterative-and-incremental life-cycle model is used, at the end of each iteration, there is a working version of part of the overall target software product. The client and the intended users can experiment with that version and determine what changes are needed to ensure that the future complete implementation meets their needs. These changes can be made to a subsequent increment, and the client and users can then determine if further changes are needed. A variation on this is to deliver partial versions of the software product, not only for experimentation but also to smooth the introduction of the new software product in the client organization. Change is almost always perceived as a threat. All too often, users fear that the introduction of a new software product within the workplace will result in them losing their jobs to a computer. However, introducing a software product gradually can have two benefits. First, the understandable fear of being replaced by a computer is diminished. Second, it is generally easier to learn the functionality of a complex software product if that functionality is introduced stepwise over a period of months, rather than as a whole.
5. There is empirical evidence that the iterative-and-incremental life cycle works. The pie chart of Figure 1.1 shows the results of the report from the Standish Group on projects completed in 2006 [Rubenstein, 2007]. In fact, this report (the so-called CHAOS Report—see Just in Case You Wanted to Know Box 2.2) is produced every 2 years. Figure 2.7 shows the results for 1994 through 2006. The percentage of successful products increased steadily from 16 percent in 1994 to 34 percent in 2002, but then decreased to 29 percent in 2004. In both the 2002 [Softwaremag.com, 2004] and 2004 [Hayes, 2004] reports, one of the factors associated with the successful projects was the use of an iterative process. (The reasons given for the decrease in the percentage of successful projects in 2004 included: more large projects than in 2002, use of the waterfall model, lack of user involvement, and lack of support from senior executives [Hayes, 2004].) Then, the percentage of successful projects increased again in the 2006 study to 35 percent. The president of the Standish Group, Jim Johnson, attributed this increase to three factors: better project management, the emerging Web infrastructure, and (again) iterative development [Rubenstein, 2007].

Just in Case You Wanted to Know Box 2.2

The term *CHAOS* is an acronym. For some unknown reason, the Standish Group keeps the acronym top secret. They state [Standish, 2003]:

> Only a few people at The Standish Group, and any one of the 360 people who received and saved the T-shirts we gave out after they completed the first survey in 1994, know what the CHAOS letters represent.

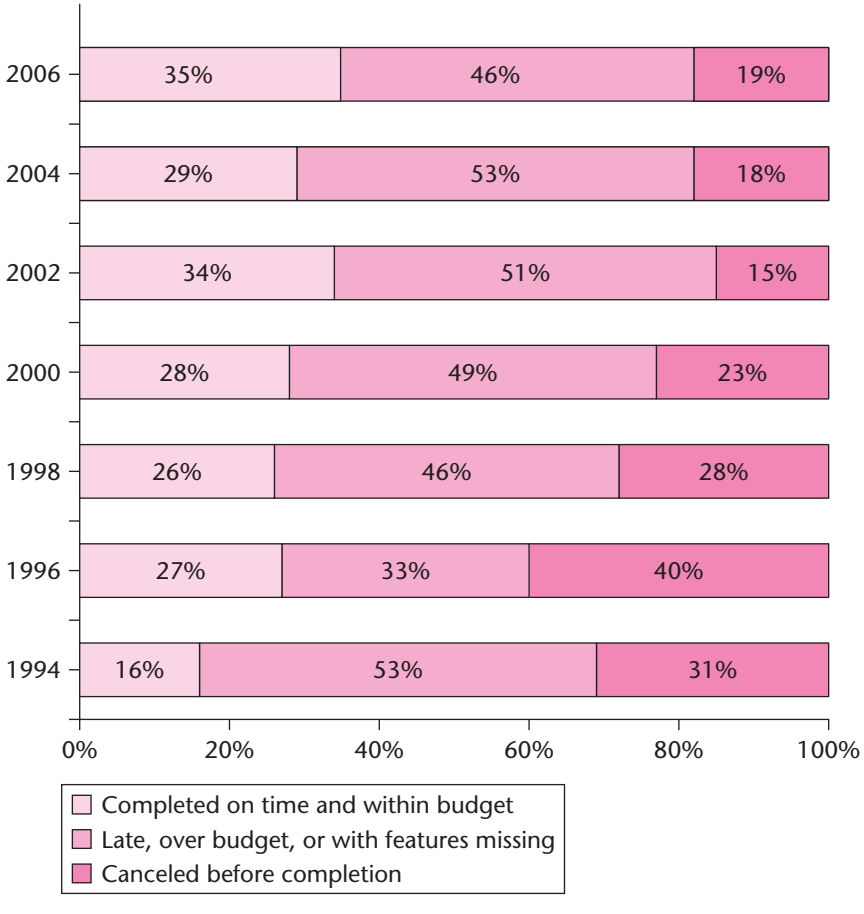


FIGURE 2.7 Results of the Standish Group CHAOS Report from 1994 to 2006.

2.8 Managing Iteration and Incrementation

At first glance, the iterative-and-incremental model of Figures 2.4 and 2.5 looks totally chaotic. Instead of the orderly progression from requirements to implementation of the waterfall model (Figure 2.3), it appears that developers do whatever they like, perhaps some coding in the morning, an hour or two of design after lunch, and then half an hour of specifying before going home. That is *not* the case. On the contrary, the iterative-and-incremental model is as regimented as the waterfall model, because as previously pointed out, developing a software product using the iterative-and-incremental model is nothing more or less than developing a series of smaller software products, all using the waterfall model.

In more detail, as shown in Figure 2.3, developing a software product using the waterfall model means successively performing the requirements, analysis, design, and implementation phases (in that order) on the software product as a whole. If a problem is encountered, the feedback loops of Figure 2.3 (dashed arrows) are followed; that is, iteration (maintenance) is performed. However, if the same software product is developed using the iterative-and-incremental model, the software product is treated as a set of increments. For each increment in turn, the requirements, analysis, design, and implementation phases (in that order) are repeatedly performed *on that increment* until it is clear that no further iteration is needed. In other words, the project as a whole is broken up into a series of waterfall mini projects. During each mini project, iteration is performed as needed, as shown in Figure 2.5. Therefore, the reason the previous paragraph stated that the iterative-and-incremental model is as regimented as the waterfall model is because the iterative-and-incremental model *is* the waterfall model, applied successively.

2.9 Other Life-Cycle Models

We now consider a number of other life-cycle models, including the spiral model and the synchronize-and-stabilize model. We begin with the infamous code-and-fix model.

2.9.1 Code-and-Fix Life-Cycle Model

It is unfortunate that so many products are developed using what might be termed the **code-and-fix life-cycle model**. The product is implemented without requirements or specifications, or any attempt at design. Instead, the developers simply throw code together and rework it as many times as necessary to satisfy the client. This approach is shown in Figure 2.8, which clearly displays the absence of requirements, specifications, and design. Although this approach may work well on short programming exercises 100 or 200 lines long, the code-and-fix model is totally unsatisfactory for products of any reasonable size. Figure 1.6 shows that the cost of changing a software product is relatively small if the

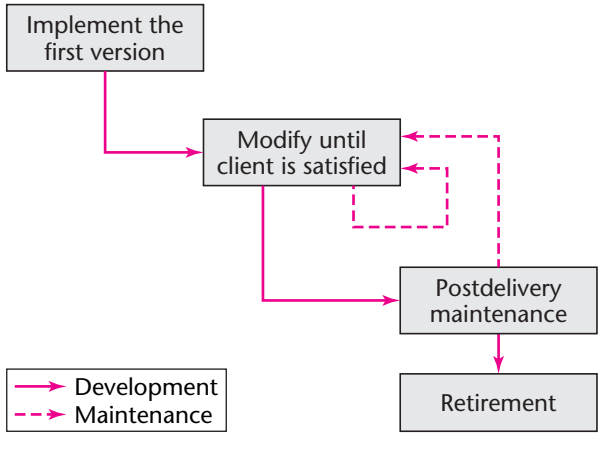


FIGURE 2.8 The code-and-fix life-cycle model.

change is made during the requirements, analysis, or design phases but grows unacceptably large if changes are made after the product has been coded or, worse, if it has already been delivered and installed on the client's computer. Hence, the cost of the code-and-fix approach is actually far greater than the cost of a properly specified and meticulously designed product. In addition, maintenance of a product can be extremely difficult without specification or design documents, and the chances of a regression fault occurring are considerably greater. Instead of the code-and-fix approach, it is essential that, before development of a product begins, an appropriate life-cycle model be chosen.

Regrettably, all too many projects use the code-and-fix model. The problem is particularly acute in organizations that measure progress solely in terms of lines of code, so members of the software development team are pressured into churning out as many lines of code as possible, starting on Day One of the project. The code-and-fix model is the easiest way to develop software—and by far the worst way.

A simplified version of the waterfall model was presented in Section 2.2. We now consider that model in more detail.

2.9.2 Waterfall Life-Cycle Model

The waterfall life-cycle model was first put forward by Royce [1970]. Figure 2.9 shows the feedback loops for maintenance while the product is being developed, as reflected in Figure 2.3, the simplified waterfall model. Figure 2.9 also shows the feedback loops for postdelivery maintenance.

A critical point regarding the waterfall model is that no phase is complete until the documentation for that phase has been completed and the products of that phase have been approved by the software quality assurance (SQA) group. This carries over into modifications; if the products of an earlier phase have to be changed as a consequence of following

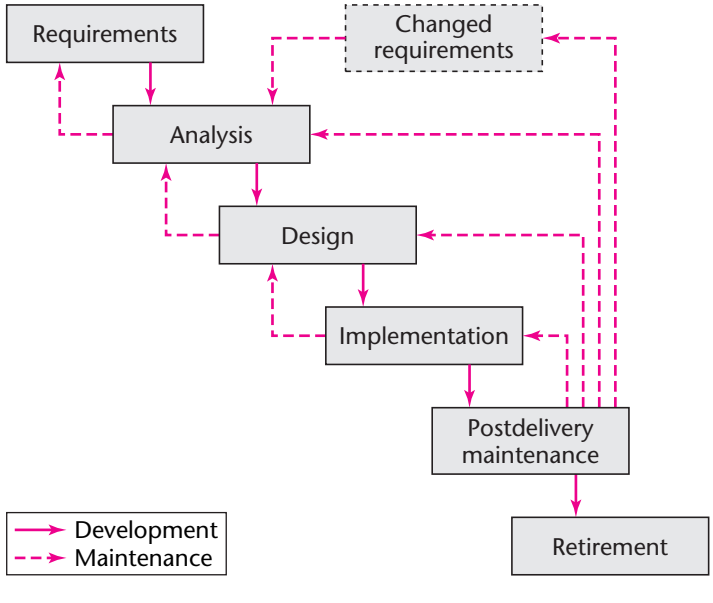


FIGURE 2.9 The full waterfall life-cycle model.

a feedback loop, that earlier phase is deemed to be complete only when the documentation for the phase has been modified and the modifications have been checked by the SQA group. Inherent in every phase of the waterfall model is testing. Testing is not a separate phase to be performed only after the product has been constructed, nor is it to be performed only at the end of each phase. Instead, as stated in Section 1.7, testing should proceed continually throughout the software process. In particular, during maintenance, it is necessary to ensure not only that the modified version of the product still does what the previous version did—and still does it correctly (regression testing)—but that it also satisfies any new requirements imposed by the client.

The waterfall model has many strengths, including the enforced disciplined approach—the stipulation that documentation be provided at each phase and the requirement that all the products of each phase (including the documentation) be meticulously checked by SQA. However, the fact that the waterfall model is documentation driven can also be a weakness. To see this, consider the following two somewhat bizarre scenarios.

First, Joe and Jane Johnson decide to build a house. They consult with an architect. Instead of showing them sketches, plans, and perhaps a scale model, the architect gives them a 20-page single-spaced typed document describing the house in highly technical terms. Even though both Joe and Jane have no previous architectural experience and hardly understand the document, they enthusiastically sign it and say, "Go right ahead, build the house!"

Another scenario is as follows: Mark Marberry buys his suits by mail order. Instead of mailing him pictures of their suits and samples of available cloths, the company sends Mark a written description of the cut and the cloth of their products. Mark then orders a suit solely on the basis of a written description.

The preceding two scenarios are highly unlikely. Nevertheless, they typify precisely the way software is often constructed using the waterfall model. The process begins with the specifications. In general, specification documents are long, detailed, and, quite frankly, boring to read. The client is usually inexperienced in the reading of software specifications, and this difficulty is compounded by the fact that specification documents are usually written in a style with which the client is unfamiliar. The difficulty is even worse when the specifications are written in a formal specification language like Z [Spivey, 1992] (Section 12.9). Nevertheless, the client proceeds to sign off on the specification document, whether properly understood or not. In many ways there is little difference between Joe and Jane Johnson contracting to have a house built from a written description that they only partially comprehend and clients approving a software product described in terms of a specification document that they only partially understand.

Mark Marberry and his mail-order suits may seem bizarre in the extreme, but that is precisely what happens when the waterfall model is used in software development. The first time that the client sees a working product is only after the entire product has been coded. Small wonder that software developers live in fear of the sentence, "I know this is what I asked for, but it isn't really what I wanted."

What has gone wrong? There is a considerable difference between the way a client understands a product as described by the specification document and the actual product. The specifications exist only on paper; the client therefore cannot really understand what the product itself will be like. The waterfall model, depending as it does so crucially on written

specifications, can lead to the construction of products that simply do not meet the client's real needs.

In fairness it should be pointed out that, just as an architect can help a client understand what is to be built by providing scale models, sketches, and plans, so the software engineer can use graphical techniques, such as data flow diagrams (Section 12.3) or UML diagrams (Chapter 17) to communicate with the client. The problem is that these graphical aids do not describe how the finished product will work. For example, there is a considerable difference between a flowchart (a diagrammatic description of a product) and the working product itself. In this book, two solutions are put forward for solving the problem that the specification document generally does not describe a product in a way that enables the client to determine whether the proposed product meets his or her needs. The object-oriented solution is described in Chapters 11 and 13. The classical solution is the rapid-prototyping model, described in Section 2.9.3.

2.9.3 Rapid-Prototyping Life-Cycle Model

A **rapid prototype** is a working model that is functionally equivalent to a subset of the product. For example, if the target product is to handle accounts payable, accounts receivable, and warehousing, then the rapid prototype might consist of a product that performs the screen handling for data capture and prints the reports, but does no file updating or error handling. A rapid prototype for a target product that is to determine the concentration of an enzyme in a solution might perform the calculation and display the answer, but without doing any validation or reasonableness checking of the input data.

The first step in the **rapid-prototyping life-cycle model** depicted in Figure 2.10 is to build a rapid prototype and let the client and future users interact and experiment with the rapid prototype. Once the client is satisfied that the rapid prototype indeed does most of

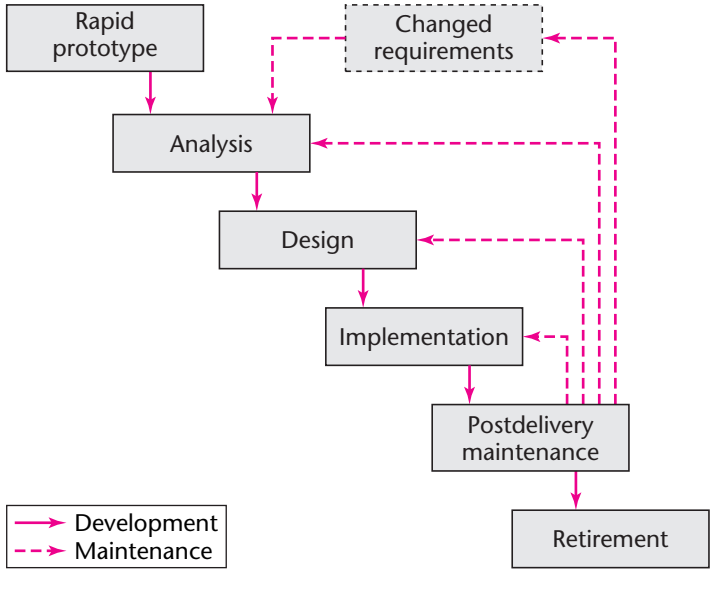


FIGURE 2.10 The rapid-prototyping life-cycle model.

what is required, the developers can draw up the specification document with some assurance that the product meets the client's real needs.

Having produced the rapid prototype, the software process continues as shown in Figure 2.10. A major strength of the rapid-prototyping model is that the development of the product is essentially linear, proceeding from the rapid prototype to the delivered product; the feedback loops of the waterfall model (Figure 2.9) are less likely to be needed in the rapid-prototyping model. There are a number of reasons for this. First, the members of the development team use the rapid prototype to construct the specification document. Because the working rapid prototype has been validated through interaction with the client, it is reasonable to expect that the resulting specification document will be correct. Second, consider the design. Even though the rapid prototype has (quite rightly) been hurriedly assembled, the design team can gain insight from it—at worst it will be of the "how not to do it" variety. Again, the feedback loops of the waterfall model are less likely to be needed here.

Implementation comes next. In the waterfall model, implementation of the design sometimes leads to design faults coming to light. In the rapid-prototyping model, the fact that a preliminary working version of the software product has already been built tends to lessen the need to repair the design during or after implementation. The prototype has given some insights to the design team, even though it may reflect only partial functionality of the complete target product.

Once the product has been accepted by the client and installed, postdelivery maintenance begins. Depending on the specific maintenance task that has to be performed, the cycle is reentered either at the requirements, analysis, design, or implementation phase.

An essential aspect of a rapid prototype is embodied in the word *rapid*. The developers should endeavor to construct the rapid prototype as rapidly as possible to speed up the software development process. After all, the sole use of the rapid prototype is to determine what the client's real needs are; once this has been determined, the rapid prototype implementation is discarded but the lessons learned are retained and used in subsequent development phases. For this reason, the internal structure of the rapid prototype is not relevant. What is important is that the prototype be built rapidly and modified rapidly to reflect the client's needs. Therefore, speed is of the essence.

Rapid prototyping is discussed in greater detail in Chapter 11.

2.9.4 Open-Source Life-Cycle Model

Almost all successful **open-source software** projects go through two informal phases. First, a single individual has an idea for a program, such as an operating system (Linux), a Net browser (Firefox), or a Web server (Apache). He or she builds an initial version, which is then made available for distribution free of charge to anyone who would like a copy; nowadays, this is done via the Internet, at sites like SourceForge.net and FreshMeat.net. If someone downloads a copy of the initial version and thinks that the program fulfills a need, he or she will start to use that program.

If there is sufficient interest in the program, the project moves gradually into informal phase two. Users become co-developers, in that some users report defects and others suggest ways of fixing those defects. Some users put forward ideas for extending the program,

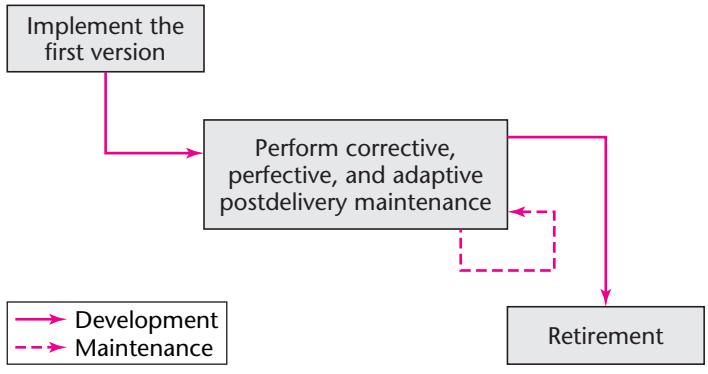


FIGURE 2.11
The open-source life-cycle model.

and others implement those ideas. As the program expands in functionality, yet other users port the program so that it can run on additional operating system/hardware combinations. A key aspect is that individuals usually work on an open-source project in their spare time on a voluntary basis; they are not paid to participate.

Now look more closely at the three activities of the second informal phase:

1. Reporting and correcting defects is corrective maintenance.
2. Adding additional functionality is perfective maintenance.
3. Porting the program to a new environment is adaptive maintenance.

In other words, the second informal phase of the open-source life-cycle model consists solely of postdelivery maintenance, as shown in Figure 2.11. In fact, the term *co-developers* in the second paragraph of this section should rather be *co-maintainers*.

There are a number of key differences between closed-source and open-source software life-cycle models:

- Closed-source software is maintained and tested by teams of employees of the organization that owns the software. Users sometimes submit defect reports. However, these are restricted to **failure reports** (reports of observed incorrect behavior); users have no access to the source code, so they cannot possibly submit **fault reports** (reports that describe where the source code is incorrect and how to correct it).

  In contrast, open-source software is generally maintained by unpaid volunteers. Users are strongly encouraged to submit defect reports. Although all users have access to the source code, only the minority have the inclination and the time, as well as the necessary skills, to peruse the source code and submit fault reports ("fixes"); most defect reports are therefore failure reports. There is generally a **core group** of dedicated maintainers who take responsibility for managing the open-source project. Some members of the **peripheral group**, that is, the users who are not members of the core group, choose to submit defect reports from time to time. The members of the core group are responsible for ensuring that these defects are corrected. In more detail, when a fault report is submitted, a core group member checks that the fix indeed solves the problem and modifies the source code appropriately. When a failure report is submitted, a member of the core group will either personally determine the fix or assign that task to another volunteer,

often a member of the peripheral group who is eager to become more involved in the open-source project. Again, the power to install the fix in the software is restricted to members of the core group.

- New versions of closed-source software are typically released roughly once a year. Each new version is carefully checked by the software quality assurance group before release; a wide variety of test cases are run.

  In contrast, a dictum of the open-source movement is "Release early. Release often" [Raymond, 2000]. That is, the core group releases a new version of an open-source product as soon as it is ready, which may be a month or even only a day after the previous version was released. This new version is released after minimal testing; it is assumed that more extensive testing will be performed by the members of the peripheral group. A new version may be installed by literally hundreds of thousands of users within a day or two of its release. These users do not run test cases as such. However, in the course of utilizing the new version on their computer, they encounter failures, which they report via e-mail. In this way, faults in the new version (as well as deeper faults in previous versions) come to light and are corrected.

Comparing Figures 2.8, 2.10, and 2.11, we see that the open-source life-cycle model has features in common with both the code-and-fix model and the rapid-prototyping model. In all three life-cycle models, an initial working version is produced. In the case of the rapid-prototyping model, this initial version is discarded, and the target product is then specified and designed before being coded. In both the code-and-fix and open-source life-cycle models, the initial version is reworked until it becomes the target product. Accordingly, in an open-source project, there are generally no specifications or design.

Bearing in mind the great importance of having specifications and designs, how have some open-source projects been so successful? In the closed-source world, some software professionals are more skilled and some are less skilled (see Section 9.2). The challenge of producing open-source software has attracted some of the finest software experts. In other words, an open-source project can be successful, despite the lack of specifications or design, if the skills of the individuals who work on that project are so superb that they can function effectively without specifications or design.

The open-source life-cycle model is restricted in its applicability. On the one hand, the open-source model has been exceedingly successfully used for certain infrastructure software projects, such as operating systems (Linux, OpenBSD, Mach, Darwin), Web browsers (Firefox, Netscape), compilers (gcc), Web servers (Apache), or database management systems (MySQL). On the other hand, it is hard to conceive of open-source development of a software product to be used only in one commercial organization. A key to open-source software development is that the members of both the core group and the periphery are users of the software being developed. Consequently, the open-source life-cycle model is inapplicable unless the target product is viewed by a wide range of users as useful to them.

At the time of writing, there are about 350,000 open-source projects at SourceForge.net and FreshMeat.net. About half them have never even attracted a team to work on the project. Of those where work has started, the overwhelming preponderance have never been completed and are unlikely to ever progress much further. But when the open-source model

has worked, it has sometimes been incredibly successful. The open-source products listed in parentheses in the previous paragraph are widely used; most of them are utilized on a regular basis by literally millions of users.

Explanations for the success of the open-source life-cycle model are presented in Chapter 4 within the context of team organizational aspects of open-source software projects.

2.9.5 Agile Processes

Extreme programming [Beck, 2000] is a somewhat controversial new approach to software development based on the iterative-and-incremental model. The first step is that the software development team determines the various features (**stories**) the client would like the product to support. For each such feature, the team informs the client how long it will take to implement that feature and how much it will cost. This first step corresponds to the requirements and analysis workflows of the iterative-and-incremental model (Figure 2.4).

The client selects the features to be included in each successive build using cost–benefit analysis (Section 5.2), that is, on the basis of the duration and the cost estimates provided by the development team as well as the potential benefits of the feature to his or her business. The proposed build is broken down into smaller pieces termed **tasks**. A programmer first draws up test cases for a task; this is termed **test-driven development** (TDD). Two programmers work together on one computer (**pair programming**) [Williams, Kessler, Cunningham, and Jeffries, 2000], implementing the task and ensuring that all the test cases work correctly. The two programmers alternate typing every 15 or 20 minutes; the programmer who is not typing carefully checks the code while it is being entered by his or her partner. The task is then integrated into the current version of the product. Ideally, implementing and integrating a task should take no more than a few hours. In general, a number of pairs will implement tasks in parallel, so integration is essentially continuous. Team members change coding partners daily, if possible; learning from the other team members increases everyone's skill level. The TDD test cases used for the task are retained and utilized in all further integration testing.

Some drawbacks to pair programming have been observed in practice [Drobka, Noftz, and Raghu, 2004]. For example, pair programming requires large blocks of uninterrupted time, and software professionals can have difficulty in finding 3- to 4-hour blocks of time. In addition, pair programming does not always work well with shy or overbearing individuals, or with two inexperienced programmers.

A number of features of extreme programming (XP) are somewhat different from the way in which software is usually developed:

- The computers of the XP team are set up in the center of a large room lined with small cubicles.
- A client representative works with the XP team at all times.
- No individual can work overtime for two successive weeks.
- There is no specialization. Instead, all members of the XP team work on requirements, analysis, design, code, and testing.

- There is no overall design step before the various builds are constructed. Instead, the design is modified while the product is being built. This procedure is termed **refactoring**. Whenever a test case will not run, the code is reorganized until the team is satisfied that the design is simple, straightforward, and runs all the test cases satisfactorily.

Two acronyms now associated with extreme programming are YAGNI (you aren't gonna need it) and DTSTTCPW (do the simplest thing that could possibly work). In other words, a principle of extreme programming is to minimize the number of features; there is no need to build a product that does any more than what the client actually needs.

Extreme programming is one of a number of new paradigms that are collectively referred to as **agile processes**. Seventeen software developers (later dubbed the Agile Alliance) met at a Utah ski resort for two days in February 2001 and produced the *Manifesto for Agile Software Development* [Beck et al., 2001]. Many of the participants had previously authored their own software development methodologies, including Extreme Programming [Beck, 2000], Crystal [Cockburn, 2001], and Scrum [Schwaber, 2001]. Consequently, the Agile Alliance did not prescribe a specific life-cycle model, but rather laid out a group of underlying principles that were common to their individual approaches to software development.

Agile processes are characterized by considerably less emphasis on analysis and design than in almost all other modern life-cycle models. Implementation starts much earlier in the life cycle because working software is considered more important than detailed documentation. Responsiveness to changes in requirements is another major goal of agile processes, and so is the importance of collaborating with the client.

One of the principles in the *Manifesto* is to deliver working software frequently, ideally every 2 or 3 weeks. One way of achieving this is to use **timeboxing** [Jalote, Palit, Kurien, and Peethamber, 2004], which has been used for many years as a time management technique. A specific amount of time is set aside for a task, and the team members then do the best job they can during that time. Within the context of agile processes, typically 3 weeks are set aside for each iteration. On the one hand, it gives the client confidence to know that a new version with additional functionality will arrive every 3 weeks. On the other hand, the developers know that they will have 3 weeks (but no more) to deliver a new iteration without client interference of any kind; once the client has chosen the work for an iteration, it cannot be changed or increased. However, if it is impossible to complete the entire task in the timebox, the work may be reduced ("descoped"). In other words, agile processes demand fixed time, not fixed features.

Another common feature of agile processes is to have a short meeting at a regular time each day. All team members have to attend the meeting. Making all the participants stand in a circle, rather than sit around a table, helps to ensure that the meeting lasts no more than the stipulated 15 minutes. Each team member in turn answers five questions:

- What have I done since yesterday's meeting?
- What am I working on today?
- What problems are preventing me from achieving this?
- What have we forgotten?
- What did I learn that I would like to share with the team?

The aim of the **stand-up meeting** is to raise problems, not solve them; solutions are found at follow-up meetings, preferably held directly after the stand-up meeting. Like timeboxing, stand-up meetings are a successful management technique now utilized

within the context of agile processes. Both timeboxed iterations and stand-up meetings are instances of two basic principles that underlie all agile methods: communication and satisfying the client's needs as quickly as possible.

Agile processes have been successfully used on a number of small-scale projects. However, agile processes have not yet been used widely enough to determine whether this approach will fulfill its early promise. Furthermore, even if agile processes turn out to be good for small-scale software products, that does not necessarily mean that they can be used for medium- or large-scale software products, as will now be explained.

To appreciate why many software professionals have expressed doubts about agile processes within the context of medium- and especially large-scale software products [Reifer, Maurer, and Erdogmus, 2003], consider the following analogy by Grady Booch [2000]. Anyone can successfully hammer together a few planks to build a doghouse, but it would be foolhardy to build a three-bedroom home without detailed plans. In addition, skills in plumbing, wiring, and roofing are needed to build a three-bedroom home, and inspections are essential. (That is, being able to build small-scale software products does not necessarily mean that one has the skills for building medium-scale software products.) Furthermore, the fact that a skyscraper is the height of 1000 doghouses does not mean that one can build a skyscraper by piling 1000 doghouses on top of one another. In other words, building large-scale software products requires even more specialized and sophisticated skills than those needed to cobble together small-scale software products.

A key determinant in deciding whether agile processes are indeed a major breakthrough in software engineering will be the cost of future postdelivery maintenance (Section 1.3.2). That is, if the use of agile processes results in a reduction in the cost of postdelivery maintenance, XP and other agile processes will become widely adopted. On the other hand, refactoring is an intrinsic component of agile processes. As previously explained, the product is not designed as a whole; instead, the design is developed incrementally, and the code is reorganized whenever the current design is unsatisfactory for any reason. This refactoring then continues during postdelivery maintenance. If the design of a product when it passes its acceptance test is open-ended and flexible, then perfective maintenance should be easy to achieve at a low cost. However, if the design has to be refactored whenever additional functionality is added, then the cost of postdelivery maintenance of that product will be unacceptably high. As a consequence of the newness of the approach, there are still essentially no data on the maintenance of software developed using agile processes. However, preliminary maintenance data indicate that refactoring can consume a large percentage of the overall cost [Li and Alshayeb, 2002].

Experiments have shown that certain features of agile processes can work well. For example, Williams, Kessler, Cunningham, and Jeffries [2000] showed that pair programming leads to the development of higher-quality code in a shorter time, with greater job satisfaction. However, an extensive experiment to evaluate pair programming within the context of software maintenance described in Section 4.6 [Arisholm, Gallis, Dybå, and Sjøberg, 2007] came to the same conclusion as an analysis of 15 published studies comparing the effectiveness of individual and pair programming [Dybå et al., 2007]: It depends on both the programmer's expertise and the complexity of the software product and the tasks to be solved.

The *Manifesto for Agile Software Development* essentially claims that agile processes are superior to more disciplined processes like the Unified Process (Chapter 3). Skeptics respond that proponents of agile processes are little more than hackers. However, there is a middle ground. The two approaches are not incompatible; it is possible to incorporate proven features

of agile processes within the framework of disciplined processes. This integration of the two approaches is described in books such as the one by Boehm and Turner [2003].

In conclusion, agile processes appear to be a useful approach to building small-scale software products when the client's requirements are vague. In addition, some of the features of agile processes can be effectively utilized within the context of other life-cycle models.

2.9.6 Synchronize-and-Stabilize Life-Cycle Model

Microsoft, Inc., is the world's largest manufacturer of COTS software. The majority of its packages are built using a version of the iterative-and-incremental model that has been termed the **synchronize-and-stabilize life-cycle model** [Cusumano and Selby, 1997].

The requirements analysis phase is conducted by interviewing numerous potential clients for the package and extracting a list of features of highest priority to the clients. A specification document is now drawn up. Next, the work is divided into three or four builds. The first build consists of the most critical features, the second build consists of the next most critical features, and so on. Each build is carried out by a number of small teams working in parallel. At the end of each day, all the teams **synchronize**; that is, they put the partially completed components together and test and debug the resulting product. **Stabilization** is performed at the end of each of the builds. Any remaining faults that have been detected so far are fixed, and they now **freeze** the build; that is, no further changes will be made to the specifications.

The repeated synchronization step ensures that the various components always work together. Another advantage of this regular execution of the partially constructed product is that the developers obtain early insight into the operation of the product and can modify the requirements if necessary during the course of a build. The life-cycle model can be used even if the initial specification is incomplete. The synchronize-and-stabilize model is considered further in Section 4.5, where team organizational details are discussed.

The spiral model has been left to last because it incorporates aspects of all the other models described in Section 2.9.

2.9.7 Spiral Life-Cycle Model

As stated in Section 2.5, an element of risk is always involved in the development of software. For example, key personnel can resign before the product has been adequately documented. The manufacturer of hardware on which the product is critically dependent can go bankrupt. Too much, or too little, can be invested in testing and quality assurance. After spending hundreds of thousands of dollars on developing a major software product, technological breakthroughs can render the entire product worthless. An organization may research and develop a database management system, but before the product can be marketed, a lower-priced, functionally equivalent package is announced by a competitor. The components of a product may not fit together when integration is performed. For obvious reasons, software developers try to minimize such risks wherever possible.

One way of minimizing certain types of risk is to construct a prototype. As described in Section 2.9.3, one approach to reducing the risk that the delivered product will not satisfy the client's real needs is to construct a rapid prototype during the requirements phase. During subsequent phases, other sorts of prototypes may be appropriate. For example, a telephone company may devise a new, apparently highly effective algorithm for routing calls through a long-distance network. If the product is implemented but does not work as expected, the telephone company will have wasted the cost of developing the product. In addition, angry or

inconvenienced customers may take their business elsewhere. This outcome can be avoided by constructing a **proof-of-concept prototype** to handle only the routing of calls and testing it on a simulator. In this way, the actual system is not disturbed; and for the cost of implementing just the routing algorithm, the telephone company can determine whether it is worthwhile to develop an entire network controller incorporating the new algorithm.

A proof-of-concept prototype is not a rapid prototype constructed to be certain that the requirements have been accurately determined, as described in Section 2.9.3. Instead, it is more like an engineering prototype, that is, a scale model constructed to test the feasibility of construction. If the development team is concerned whether a particular part of the proposed software product can be constructed, a proof-of-concept prototype is constructed. For example, the developers may be concerned whether a particular computation can be performed quickly enough. In that case, they build a prototype to test the timing of just that computation. Or they may be worried that the font they intend to use for all screens will be too small for the average user to read without eyestrain. In this instance, they construct a prototype to display a number of different screens and determine by experiment whether the users find the font uncomfortably small.

The idea of minimizing risk via the use of prototypes and other means is the idea underlying the **spiral life-cycle model** [Boehm, 1988]. A simplified way of looking at this life-cycle model is as a waterfall model with each phase preceded by risk analysis, as shown in Figure 2.12. Before commencing each phase, an attempt is made to **mitigate** (control) the risks. If it is impossible to mitigate all the significant risks at that stage, then the project is immediately terminated.

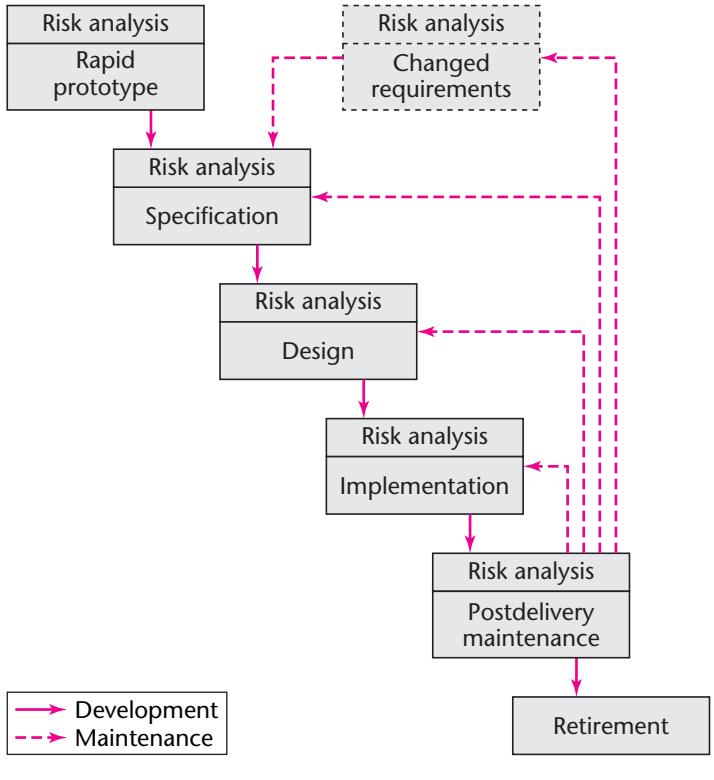


FIGURE 2.12
A simplified version of the spiral life-cycle model.

Prototypes can be used effectively to provide information about certain classes of risk. For example, timing constraints can generally be tested by constructing a prototype and measuring whether the prototype can achieve the necessary performance. If the prototype is an accurate functional representation of the relevant features of the product, then measurements made on the prototype should give the developers a good idea as to whether the timing constraints can be achieved.

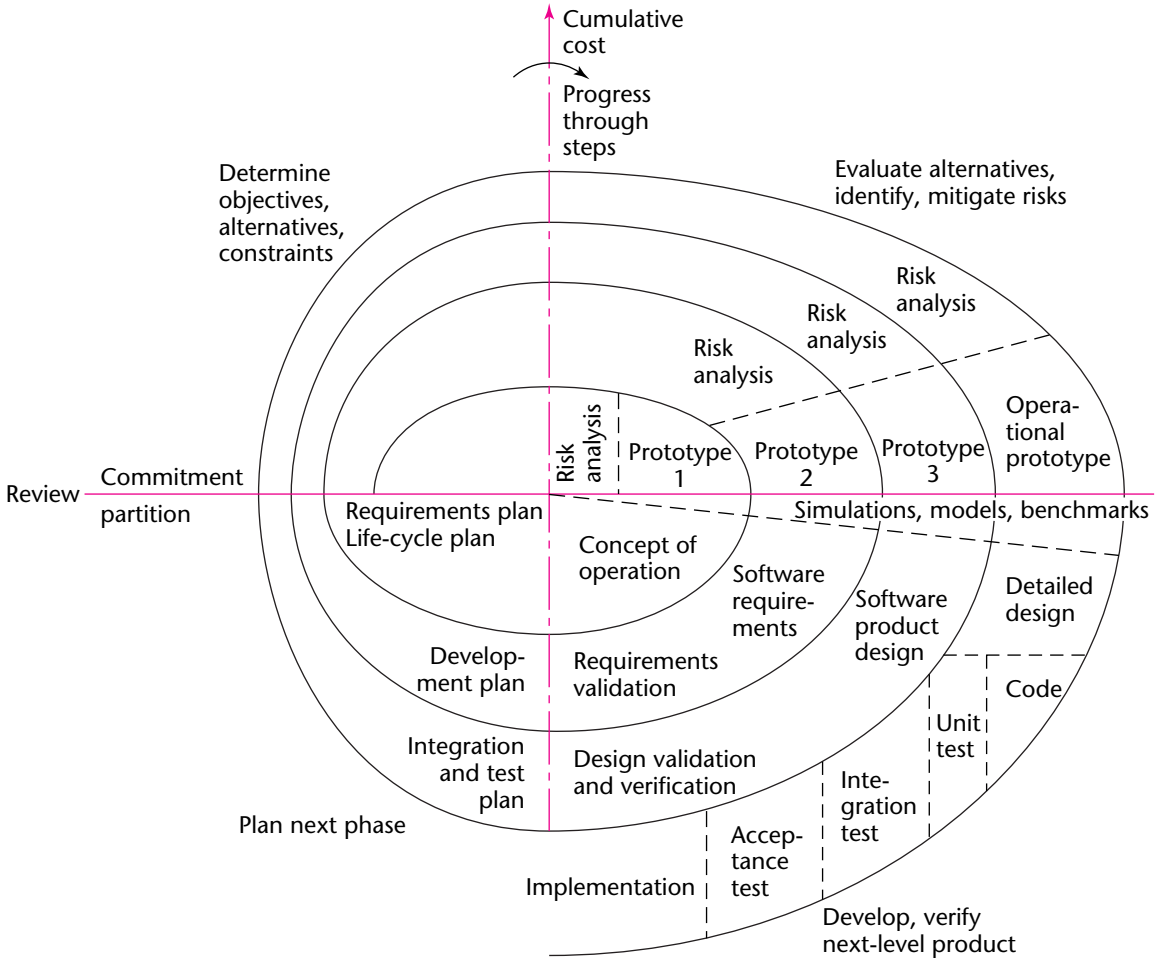
Other areas of risk are less amenable to prototyping, for example, the risk that the software personnel necessary to build the product cannot be hired or that key personnel may resign before the project is complete. Another potential risk is that a particular team may not be competent enough to develop a specific large-scale product. A successful contractor who builds single-family homes would probably not be able to build a high-rise office complex. In the same way, there are essential differences between small-scale and large-scale software, and prototyping is of little use. This risk cannot be mitigated by testing team performance on a much smaller prototype, in which team organizational issues specific to large-scale software cannot arise. Another area of risk for which prototyping cannot be employed is evaluating the delivery promises of a hardware supplier. A strategy the developer can adopt is to determine how well previous clients of the supplier have been treated, but past performance is by no means a certain predictor of future performance. A penalty clause in the delivery contract is one way of trying to ensure that essential hardware is delivered on time, but what if the supplier refuses to sign an agreement that includes such a clause? Even with a penalty clause, late delivery may occur and eventually lead to legal action that can drag on for years. In the meantime, the software developer may have gone bankrupt because nondelivery of the promised hardware caused nondelivery of the promised software. In short, whereas prototyping helps reduce risk in some areas, in other areas it is at best a partial answer, and in still others it is no answer at all.

The full spiral model is shown in Figure 2.13. The radial dimension represents cumulative cost to date, and the angular dimension represents progress through the spiral. Each cycle of the spiral corresponds to a phase. A phase begins (in the top left quadrant) by determining objectives of that phase, alternatives for achieving those objectives, and constraints imposed on those alternatives. This process results in a strategy for achieving those objectives. Next, that strategy is analyzed from the viewpoint of risk. Attempts are made to mitigate every potential risk, in some cases by building a prototype. If certain risks cannot be mitigated, the project may be terminated immediately; under some circumstances, however, a decision could be made to continue the project but on a significantly smaller scale. If all risks are successfully mitigated, the next development step is started (bottom right quadrant). This quadrant of the spiral model corresponds to the classical waterfall model. Finally, the results of that phase are evaluated and the next phase is planned.

The spiral model has been used successfully to develop a wide variety of products. In one set of 25 projects in which the spiral model was used in conjunction with other means of increasing productivity, the productivity of every project increased by at least 50 percent over previous productivity levels and by 100 percent in most of the projects [Boehm, 1988]. To be able to decide whether the spiral model should be used for a given project, the strengths and weaknesses of the spiral model are now assessed.

The spiral model has a number of strengths. The emphasis on alternatives and constraints supports the reuse of existing software (Section 8.1) and the incorporation of

FIGURE 2.13 Full spiral life-cycle model [Boehm, 1988]. (© 1988 IEEE.)



software quality as a specific objective. In addition, a common problem in software development is determining when the products of a specific phase have been adequately tested. Spending too much time on testing is a waste of money, and delivery of the product may be unduly delayed. Conversely, if too little testing is performed, then the delivered software may contain residual faults, resulting in unpleasant consequences for the developers. The spiral model answers this question in terms of the risks that would be incurred by not doing enough testing or by doing too much testing. Perhaps most important, within the structure of the spiral model, postdelivery maintenance is simply another cycle of the spiral; there is essentially no distinction between postdelivery maintenance and development. Therefore, the problem that postdelivery maintenance is sometimes maligned by ignorant software professionals does not arise, because postdelivery maintenance is treated the same way as development.

There are restrictions on the applicability of the spiral model. Specifically, in its present form, the model is intended exclusively for internal development of large-scale software [Boehm, 1988]. Consider an internal project, that is, one where the developers and client are members of the same organization. If risk analysis leads to the conclusion that the project should be terminated, then in-house software personnel can simply be reassigned to a different project. However, once a contract has been signed between a development organization and an external client, an attempt by either side to terminate that contract can lead to a breach-of-contract lawsuit. Therefore, in the case of contract software, all risk analysis must be performed by both client and developers before the contract is signed, not as in the spiral model.

A second restriction on the spiral model relates to the size of the project. Specifically, the spiral model is applicable to only large-scale software. It makes no sense to perform risk analysis if the cost of performing the risk analysis is comparable to the cost of the project as a whole, or if performing the risk analysis would significantly affect the profit potential. Instead, the developers should first decide how much is at risk and then how much risk analysis, if any, to perform.

A major strength of the spiral model is that it is risk driven, but this can also be a weakness. Unless the software developers are skilled at pinpointing the possible risks and analyzing the risks accurately, there is a real danger that the team may believe that all is well at a time when the project, in fact, is headed for disaster. Only if the members of the development team are competent risk analysts should management decide to use the spiral model.

Overall, however, the major weakness of the spiral model, as well as the waterfall model and the rapid-prototyping model, is that it assumes that software is developed in discrete phases. In reality, however, software development is iterative and incremental, as reflected in the evolution-tree model (Section 2.2) or the iterative-and-incremental model (Section 2.5).

2.10 Comparison of Life-Cycle Models

Nine different software life-cycle models have been examined with special attention paid to some of their strengths and weaknesses. The code-and-fix model (Section 2.9.1) should be avoided. The waterfall model (Section 2.9.2) is a known quantity. Its strengths are understood, and so are its weaknesses. The rapid-prototyping model (Section 2.9.3) was developed as a reaction to a specific perceived weakness in the waterfall model, namely, that the delivered product may not be what the client really needs. However, there is still insufficient evidence that this approach is superior to the waterfall model in other respects. The open-source life-cycle model has been incredibly successful in a small number of cases when used to construct infrastructure software (Section 2.9.4). Agile processes (Section 2.9.5) are a set of controversial new approaches that, so far, appear to work, but for only small-scale software. The synchronize-and-stabilize model (Section 2.9.6) has been used with great success by Microsoft, but as yet there is no evidence of comparable success in other corporate cultures. Yet another alternative is to use the spiral model (Section 2.9.7), but only if the developers are adequately trained in risk analysis and risk resolution. The evolution-tree model (Section 2.2) and the iterative-and-incremental model (Section 2.5) are closest to the way that software is produced in the real world. An overall comparison appears in Figure 2.14.

Each software development organization should decide on a life-cycle model that is appropriate for that organization, its management, its employees, and its software process

FIGURE 2.14
Comparison of life-cycle models described in this chapter, including the section in which each is defined.

Life-Cycle Model	Strengths	Weaknesses
Evolution-tree model (Section 2.2)	Closely models real-world software production Equivalent to the iterative-and-incremental model	
Iterative-and-incremental life-cycle model (Section 2.5)	Closely models real-world software production Underlies the Unified Process	
Code-and-fix life-cycle model (Section 2.9.1)	Fine for short programs that require no maintenance	Totally unsatisfactory for nontrivial programs
Waterfall life-cycle model (Section 2.9.2)	Disciplined approach Document driven	Delivered product may not meet client's needs
Rapid-prototyping life-cycle model (Section 2.9.3)	Ensures that the delivered product meets the client's needs	Not yet proven beyond all doubt
Open-source life-cycle model (Section 2.9.4)	Has worked extremely well in a small number of instances	Limited applicability Usually does not work
Agile processes (Section 2.9.5)	Work well when the client's requirements are vague	Appear to work on only small-scale projects
Synchronize-and-stabilize life-cycle model (Section 2.9.6)	Future users' needs are met Ensures that components can be successfully integrated	Has not been widely used other than at Microsoft
Spiral life-cycle model (Section 2.9.7)	Risk driven	Can be used for only large-scale, in-house products Developers have to be competent in risk analysis and risk resolution

and should vary the life-cycle model depending on the features of the specific product currently under development. Such a model incorporates appropriate aspects of the various life-cycle models, utilizing their strengths and minimizing their weaknesses.

Chapter Review

There are significant differences between the way that software is developed in theory (Section 2.1) and the way it is developed in practice. The Winburg mini case study is used to introduce the evolution-tree model (Section 2.2). Lessons of this mini case study, especially that requirements change, are presented in Section 2.3. Change is discussed in greater detail in Section 2.4, where the moving-target problem is presented using the Teal Tractors mini case study. In Section 2.5, the importance of iteration and incrementation in real-world software engineering is stressed, and the iterative-and-incremental model is presented. The Winburg mini case study is then re-examined in Section 2.6 to illustrate the equivalence of the evolution-tree model and the iterative-and-incremental model. In Section 2.7, the strengths of the iterative-and-incremental model are presented, particularly that it enables us to resolve risks early. Management of the iterative-and-incremental model is discussed in Section 2.8. A number of different life-cycle models are now described, including the code-and-fix life-cycle model (Section 2.9.1), waterfall life-cycle model (Section 2.9.2), rapid-prototyping life-cycle model (Section 2.9.3), open-source life-cycle model (Section 2.9.4), agile processes (Section 2.9.5), synchronize-and-stabilize life-cycle model (Section 2.9.6), and spiral life-cycle model (Section 2.9.7). In Section 2.10, these life-cycle models are compared and suggestions are made regarding the choice of a life-cycle model for a specific project.

For Further Reading

The waterfall model was first put forward in [Royce, 1970]. An analysis of the waterfall model is given in the first chapter of [Royce, 1998].

The synchronize-and-stabilize model is outlined in [Cusumano and Selby, 1997] and described in detail in [Cusumano and Selby, 1995]. The spiral model is explained in [Boehm, 1988], and its application to the TRW Software Productivity System appears in [Boehm et al., 1984].

Extreme programming is described in [Beck, 2000]; refactoring is the subject of [Fowler et al., 1999]. The *Manifesto for Agile Software Development* may be found at [Beck et al., 2001]. Books have been published on a variety of agile methods, including [Cockburn, 2001] and [Schwaber, 2001]. Agile methods are advocated in [Highsmith and Cockburn, 2001], [Boehm, 2002], [DeMarco and Boehm, 2002], and [Boehm and Turner, 2003], whereas the case against agile methods is presented in [Stephens and Rosenberg, 2003]. Refactoring is surveyed in [Mens and Tourwe, 2004]. The use of XP in four mission-critical projects is described in [Drobka, Noftz, and Raghu, 2004]. Issues that can arise when introducing agile processes within an organization that currently is using traditional methodologies are discussed in [Nerur, Mahapatra, and Mangalaraj, 2005] and in [Boehm and Turner, 2005].

A number of papers on extreme programming appear in the May–June 2003 issue of *IEEE Software*, including [Murru, Deias, and Mugheddu, 2003] and [Rasmusson, 2003], both of which describe successful projects developed using extreme programming. The June 2003 issue of *IEEE Computer* contains several articles on agile processes. The May–June 2005 issue of *IEEE Software* has four articles on agile processes, especially [Ceschi, Sillitti, Succi, and De Panfilis, 2005] and [Karlström and Runeson, 2005]. The extent to which agile methods are used in the software industry is analyzed in [Hansson, Dittrich, Gustafsson, and Zarnak, 2006]. A survey of the critical success factors in agile software products is presented in [Chow and Cao, 2008]. Approaches to assist in the transition to agile methods are given in [Qumer and Henderson-Sellers, 2008]. Refactoring poses problems for software configuration management tools; a solution is put forward in [Dig, Manzoor, Johnson, and Nguyen, 2008].

Agile testing of a large-scale software product is described in [Talby, Keren, Hazzan, and Dubinsky, 2006]. The effectiveness of test-driven development is discussed in [Erdogmus, Morisio, and Torchiano, 2005]. The May–June 2007 issue of *IEEE Software* has a variety of articles on test-driven development, including [Martin, 2007].

Risk analysis is described in [Ropponen and Lyttinen, 2000], [Longstaff, Chittister, Pethia, and Haimes, 2000], and [Scott and Vessey, 2002]. Managing risks in offshore software development is presented in [Sakthivel, 2007] and in [Iacovou and Nakatsu, 2008]. Risk management when software is developed using COTS components is described in [Li et al., 2008].

A major iterative-and-incremental model is described in detail in [Jacobson, Booch, and Rumbaugh, 1999]. However, many other iterative-and-incremental models have been put forward over the past 30 years, as recounted in [Larman and Basili, 2003]. The use of an incremental model to build an air-traffic control system is discussed in [Goth, 2000]. An iterative approach to re-engineering legacy systems is given in [Bianchi, Caivano, Marengo, and Visaggio, 2003]. A tool for supporting incremental software development while ensuring that the artifacts evolve consistently is described in [Reiss, 2006].

Many other life-cycle models have been put forward. For example, Rajlich and Bennett [2000] describe a maintenance-oriented life-cycle model. The July–August 2000 issue of *IEEE Software* has a variety of papers on software life-cycle models, including [Williams, Kessler, Cunningham, and Jeffries, 2000] which describes an experiment on pair programming, one component of agile methods.

Rajlich [2006] goes further and suggests that many of the topics of this chapter have led us to a new paradigm for software engineering.

The proceedings of the International Software Process Workshops are a useful source of information on life-cycle models. [ISO/IEC 12207, 1995] is a widely accepted standard for software life-cycle processes.

Key Terms

agile process *60*
analysis workflow *44*
architecture *49*
artifact *41*
baseline *41*
code-and-fix life-cycle model *52*
core group *57*
core workflow *44*
design workflow *44*
evolution-tree life-cycle model *40*
extreme programming *59*
failure report *57*
fault report *57*
feature creep *43*
freeze *62*
implementation workflow *44*
incrementation *44*
iteration *44*
iterative-and-incremental life-cycle model *44*
life-cycle model *40*
Miller's Law *44*
mitigate risk *63*
model *40*
moving-target problem *43*
open-source software *56*
pair programming *59*
peripheral group *57*
proof-of-concept prototype *63*
rapid prototype *55*
rapid-prototyping life-cycle model *55*
refactoring *60*
regression fault *43*
requirements workflow *44*
risk *50*
robustness *49*
spiral life-cycle model *63*
stabilize *62*
stand-up meeting *60*
stepwise refinement *44*
story *59*
synchronize *62*
synchronize-and-stabilize life-cycle model *62*
task *59*
test-driven development *59*
test workflow *44*
timeboxing *60*
waterfall life-cycle model *41*
workflow *44*

Problems

2.1 Represent the Winburg mini case study of Sections 2.2 and 2.3 using the waterfall model. Is this more or less effective than the evolution-tree model? Explain your answer.

2.2 Assume that the programmer in the Winburg mini case study had used single-precision numbers from the beginning. Draw the resulting evolution tree.

2.3 What is the connection between Miller's Law and stepwise refinement?

2.4 Does stepwise refinement correspond to iteration or incrementation?

2.5 How are a workflow, an artifact, and a baseline related?

2.6 What is the connection between the waterfall model and the iterative-and-incremental model?

2.7 Suppose you have to build a product to determine the cube root of 9384.2034 to four decimal places. Once the product has been implemented and tested, it will be thrown away. Which life-cycle model would you use? Give reasons for your answer.

2.8 You are a software engineering consultant and have been called in by the vice-president for finance of a corporation that manufactures tires and sells them via its large chain of retail outlets. She wants your organization to build a product that will monitor the company's stock, starting with the purchasing of the raw materials and keeping track of the tires as they are manufactured, distributed to the individual stores, and sold to customers. What criteria would you use in selecting a life-cycle model for the project?

2.9 List the risks involved in developing the software of Problem 2.8. How would you attempt to mitigate each risk?

2.10 Your development of the stock control product for the tire company is so successful that your organization decides that it must be reimplemented as a package to be sold to a variety of different organizations that manufacture and sell products via their own retailers. The new product must therefore be portable and easily adapted to new hardware and/or operating systems. How would the criteria you use in selecting a life-cycle model for this project differ from those in your answer to Problem 2.8?

2.11 Describe the sort of product that would be an ideal application for open-source software development.

2.12 Now describe the type of situation where open-source software development is inappropriate.

2.13 Describe the sort of product that would be an ideal application for an agile process.

2.14 Now describe the type of situation where an agile process is inappropriate.

2.15 Describe the sort of product that would be an ideal application for the spiral life-cycle model.

2.16 Now describe the type of situation where the spiral life-cycle model is inappropriate.

2.17 Describe a risk inherent in using the waterfall life-cycle model.

2.18 Describe a risk inherent in using the code-and-fix life-cycle model.

2.19 Describe a risk inherent in using the open-source life-cycle model.

2.20 Describe a risk inherent in using agile processes.

2.21 Describe a risk inherent in using the spiral life-cycle model.

2.22 (Term Project) Which software life-cycle model would you use for the Chocoholics Anonymous product described in Appendix A? Give reasons for your answer.

2.23 (Readings in Software Engineering) Your instructor will distribute copies of [Rajlich, 2006]. Do you agree that software engineering has embarked on a new paradigm? Explain your answer.

References

[Arisholm, Gallis, Dybå, and Sjøberg, 2007] E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," *IEEE Transactions on Software Engineering* **33** (February 2007), pp. 65–86.

[Beck, 2000] K. Beck, *Extreme Programming Explained: Embrace Change,* Addison-Wesley Longman, Reading, MA, 2000.

[Beck et al., 2001] K. Beck, M. Beedle, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, and A. van Bennekum, *Manifesto for Agile Software Development*, agilemanifesto.org, 2001.

[Bianchi, Caivano, Marengo, and Visaggio, 2003] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio, "Iterative Reengineering of Legacy Systems," *IEEE Transactions on Software Engineering* **29** (March 2003), pp. 225–41.

[Boehm, 1988] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* **21** (May 1988), pp. 61–72.

[Boehm, 2002] B. W. Boehm, "Get Ready for Agile Methods, with Care," *IEEE Computer* **35** (January 2002), pp. 64–69.

[Boehm and Turner, 2003] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley Professional, Boston, MA, 2003.

[Boehm and Turner, 2005] B. Boehm and R. Turner, "Management Challenges to Implementing Agile Processes in Traditional Development Organizations," *IEEE Software* **22** (September–October 2005), pp. 30–39.

[Boehm et al., 1984] B. W. Boehm, M. H. Penedo, E. D. Stuckle, R. D. Williams, and A. B. Pyster, "A Software Development Environment for Improving Productivity," *IEEE Computer* **17** (June 1984), pp. 30–44.

[Booch, 2000] G. Booch, "The Future of Software Engineering," keynote address, International Conference on Software Engineering, Limerick, Ireland, May 2000.

[Ceschi, Sillitti, Succi, and De Panfilis, 2005] M. Ceschi, A. Sillitti, G. Succi, and S. De Panfilis, "Project Management in Plan-Based and Agile Companies," *IEEE Software* **22** (May–June 2005), pp. 21–27.

[Chow and Cao, 2008] T. Chow and D.-B. Cao, "A Survey Study of Critical Success Factors in Agile Software Projects," *Journal of Systems and Software* **81** (June 2008), pp. 961–71.

[Cockburn, 2001] A. Cockburn, *Agile Software Development*, Addison-Wesley Professional, Reading, MA, 2001.

[Cusumano and Selby, 1995] M. A. Cusumano and R. W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press/Simon and Schuster, New York, 1995.

[Cusumano and Selby, 1997] M. A. Cusumano and R. W. Selby, "How Microsoft Builds Software," *Communications of the ACM* **40** (June 1997), pp. 53–61.

[DeMarco and Boehm, 2002] T. DeMarco and B. Boehm, "The Agile Methods Fray," *IEEE Computer* **35** (June 2002), pp. 90–92.

[Dig, Manzoor, Johnson, and Nguyen, 2008] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective Software Merging in the Presence of Object-Oriented Refactorings," *IEEE Transactions on Software Engineering* **34** (May–June 2008), pp. 321–35.

[Drobka, Noftz, and Raghu, 2004] J. Drobka, D. Noftz, and R. Raghu, "Piloting XP on Four Mission-Critical Projects," *IEEE Software* **21** (November–December 2004), pp. 70–75.

[Dybå et al., 2007] T. Dybå, E. Arisholm, D. I. K. Sjøberg, J. E. Hannay, and F. Shull, "Are Two Heads Better than One? On the Effectiveness of Pair Programming," *IEEE Software* **24** (November–December 2007), pp. 12–15.

[Erdogmus, Morisio, and Torchiano, 2005] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Transactions on Software Engineering* **31** (March 2005), pp. 226–37.

[Fowler et al., 1999] M. Fowler with K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading, MA, 1999.

[Goth, 2000] G. Goth, "New Air Traffic Control Software Takes an Incremental Approach," *IEEE Software* **17** (July–August 2000), pp. 108–11.

[Hansson, Dittrich, Gustafsson, and Zarnak, 2006] C. Hansson, Y. Dittrich, B. Gustafsson, and S. Zarnak, "How Agile Are Industrial Software Development Practices?" *Journal of Systems and Software* **79** (September 2006), pp. 1217–58.

[Hayes, 2004] F. Hayes, "Chaos Is Back," *Computerworld*, www.computerworld.com/managementtopics/management/project/story/0,10801,97283,00.html, November 8, 2004.

[Highsmith and Cockburn, 2001] J. Highsmith and A. Cockburn, "Agile Software Development: The Business of Innovation," *IEEE Computer* **34** (September 2001), pp. 120–22.

[Iacovou and Nakatsu, 2008] C. L. Iacovou and R. Nakatsu, "A Risk Profile of Offshore-Outsourced Development Projects," *Communications of the ACM* **51** (June 2008) pp. 89–94.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[Jacobson, Booch, and Rumbaugh, 1999] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Jalote, Palit, Kurien, and Peethamber, 2004] P. Jalote, A. Palit, P. Kurien, and V. T. Peethamber, "Timeboxing: A Process Model for Iterative Software Development," *Journal of Systems and Software* **70** (February 2004), pp. 117–27.

[Karlström and Runeson, 2005] D. Karlström and P. Runeson, "Combining Agile Methods with Stage-Gate Project Management," *IEEE Software* **22** (May–June 2005), pp. 43–49.

[Larman and Basili, 2003] C. Larman and V. R. Basili, "Iterative and Incremental Development: A Brief History," *IEEE Computer* **36** (June 2003), pp. 47–56.

[Li and Alshayeb, 2002] W. Li and M. Alshayeb, "An Empirical Study of XP Effort," *Proceedings of the 17th International Forum on COCOMO and Software Cost Modeling*, Los Angeles, October 2002, IEEE.

[Li et al., 2008] J. Li, O. P. N. Slyngstad, M. Torchiano, M. Morisio, and C. Bunse, "A State-of-the-Practice Survey of Risk Management in Development with Off-the-Shelf Software Components," *IEEE Transactions on Software Engineering* **34** (March–April 2008), pp. 271–86.

[Longstaff, Chittister, Pethia, and Haimes, 2000] T. A. Longstaff, C. Chittister, R. Pethia, and Y. Y. Haimes, "Are We Forgetting the Risks of Information Technology?" *IEEE Computer* **33** (December 2000), pp. 43–51.

[Martin, 2007] R. C. Martin, "Professionalism and Test-Driven Development," *IEEE Software* **24** (May–June 2007), pp. 32–36.

[Mens and Tourwe, 2004] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering* **30** (February 2004), pp. 126–39.

[Miller, 1956] G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review* **63** (March 1956), pp. 81–97; reprinted in: www.well.com/user/smalin/miller.html.

[Murru, Deias, and Mugheddu, 2003] O. Murru, R. Deias, and G. Mugheddu, "Assessing XP at a European Internet Company," *IEEE Software* **20** (May–June, 2003), pp. 37–43.

[Nerur, Mahapatra, and Mangalaraj, 2005] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of Migrating to Agile Methodologies," *Communications of the ACM* **48** (May 2005), pp. 72–78.

[Qumer and Henderson-Sellers, 2008] A. Qumer and B. Henderson-Sellers, "A Framework to Support the Evaluation, Adoption and Improvement of Agile Methods in Practice," *Journal of Systems and Software* **81** (November 2008), pp. 1899–1919.

[Rajlich, 2006] V. Rajlich, "Changing the Paradigm of Software Engineering," *Communications of the ACM* **49** (August 2006), pp. 67–70.

[Rajlich and Bennett, 2000] V. Rajlich and K. H. Bennett, "A Staged Model for the Software Life Cycle," *IEEE Computer* **33** (July 2000), pp. 66–71.

[Rasmusson, 2003] J. Rasmusson, "Introducing XP into Greenfield Projects: Lessons Learned," *IEEE Software* **20** (May–June, 2003), pp. 21–29.

[Raymond, 2000] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates, Sebastopol, CA, 2000; also available at www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

[Reifer, Maurer, and Erdogmus, 2003] D. Reifer, F. Maurer, and H. Erdogmus, "Scaling Agile Methods," *IEEE Software* **20** (July–August 2004), pp. 12–14.

[Reiss, 2006] S. P. Reiss, "Incremental Maintenance of Software Artifacts," *IEEE Transactions on Software Engineering* **32** (September 2006), pp. 682–97.

[Ropponen and Lyttinen, 2000] J. Ropponen and K. Lyttinen, "Components of Software Development Risk: How to Address Them? A Project Manager Survey," *IEEE Transactions on Software Engineering* **26** (February 2000), pp. 96–111.

[Royce, 1970] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *1970 WESCON Technical Papers, Western Electronic Show and Convention*, Los Angeles, August 1970, pp. A/1-1–A/1-9; reprinted in: *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, IEEE, pp. 328–38.

[Royce, 1998] W. Royce, *Software Project Management: A Unified Framework*, Addison-Wesley, Reading, MA, 1998.

[Rubenstein, 2007] D. Rubenstein, "Standish Group Report: There's Less Development Chaos Today," www.sdtimes.com/content/article.aspx?ArticleID=30247, March 1, 2007.

[Sakthivel, 2007] S. Sakthivel, "Managing Risk in Offshore Systems Development," *Communications of the ACM* **50** (April 2007), pp. 69–75.

[Schwaber, 2001] K. Schwaber, *Agile Software Development with Scrum*, Prentice Hall, Upper Saddle River, NJ, 2001.

[Scott and Vessey, 2002] J. E. Scott and I. Vessey, "Managing Risks in Enterprise Systems Implementations," *Communications of the ACM* **45** (April 2002), pp. 74–81.

[Softwaremag.com, 2004] "Standish: Project Success Rates Improved over 10 Years," www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish, January 15, 2004.

[Spivey, 1992] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, New York, 1992.

[Standish, 2003] Standish Group International, "Introduction," www.standishgroup.com/chaos/introduction.pdf, 2003.

[Stephens and Rosenberg, 2003] M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case against XP*, Apress, Berkeley, CA, 2003.

[Talby, Keren, Hazzan, and Dubinsky, 2006] D. Talby, A. Keren, O. Hazzan, and Y. Dubinsky, "Agile Software Testing in a Large-Scale Project," *IEEE Software* **23** (July–August 2006), pp. 30–37.

[Tomer and Schach, 2000] A. Tomer and S. R. Schach, "The Evolution Tree: A Maintenance-Oriented Software Development Model," in: *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000)*, Zürich, Switzerland, February/March 2000, ACM, pp. 209–14.

[Williams, Kessler, Cunningham, and Jeffries, 2000] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair Programming," *IEEE Software* **17** (July–August 2000), pp. 19–25.

Chapter 3

The Software Process

Learning Objectives

After studying this chapter, you should be able to

- Explain why two-dimensional life-cycle models are important.
- Describe the five core workflows of the Unified Process.
- List the artifacts tested in the test workflow.
- Describe the four phases of the Unified Process.
- Explain the difference between the workflows and the phases of the Unified Process.
- Appreciate the importance of software process improvement.
- Describe the capability maturity model (CMM).

The software process is the way we produce software. It incorporates the methodology (Section 1.11) with its underlying software life-cycle model (Chapter 2) and techniques, the tools we use (Sections 5.6 through 5.12), and most important of all, the individuals building the software.

Different organizations have different software processes. For example, consider the issue of documentation. Some organizations consider the software they produce to be self-documenting; that is, the product can be understood simply by reading the source code. Other organizations, however, are documentation intensive. They punctiliously draw up specifications and check them methodically. Then they perform design activities painstakingly, check and recheck their designs before coding commences, and give extensive descriptions of each code artifact to the programmers. Test cases are preplanned, the result of each test run is logged, and the test data are meticulously filed away. Once the product has been delivered and installed on the client's computer, any suggested change must be proposed in writing, with detailed reasons for making the change. The proposed change can be made only with written authorization, and the modification is not integrated into the product until the documentation has been updated and the changes to the documentation approved.

Just in Case You Wanted to Know Box 3.1

Why does the software process vary so drastically from organization to organization? A major reason is lack of software engineering skills. All too many software professionals simply do not keep up to date. They continue to develop software Ye Olde Fashioned Way, because they know no other way.

Another reason for differences in the software process is that many software managers are excellent managers but know precious little about software development or maintenance. Their lack of technical knowledge can result in the project slipping so badly behind schedule that there is no point in continuing. This frequently is the reason why many software projects are never completed.

Yet another reason for differences among processes is management outlook. For example, one organization may decide that it is better to deliver a product on time, even if it is not adequately tested. Given the identical circumstances, a different organization might conclude that the risk of delivering that product without comprehensive testing would be far greater than taking the time to test the product thoroughly and consequently delivering it late.

Intensity of testing is another measure by which organizations can be compared. Some organizations devote up to half their software budgets to testing software, whereas others feel that only the user can thoroughly test a product. Consequently, some companies devote minimal time and effort to testing the product but spend a considerable amount of time fixing problems reported by users.

Postdelivery maintenance is a major preoccupation of many software organizations. Software that is 10, 15, or even 20 years old is continually enhanced to meet changing needs; in addition, residual faults continue to appear, even after the software has been successfully maintained for many years. Almost all organizations move their software to newer hardware every 3 to 5 years; this, too, constitutes postdelivery maintenance.

In contrast, yet other organizations essentially are concerned with research, leaving development—let alone maintenance—to others. This applies particularly to university computer science departments, where graduate students build software to prove that a particular design or technique is feasible. The commercial exploitation of the validated concept is left to other organizations. (See Just in Case You Wanted to Know Box 3.1 regarding the wide variation in the ways different organizations develop software.)

However, regardless of the exact procedure, the software development process is structured around the five workflows of Figure 2.4: requirements, analysis (specification), design, implementation, and testing. In this chapter, these workflows are described, together with potential challenges that may arise during each workflow. Solutions to the challenges associated with the production of software usually are nontrivial, and the rest of this book is devoted to describing suitable techniques. In the first part of this chapter, only the challenges are highlighted, but the reader is guided to the relevant sections or chapters for solutions. Accordingly, this part of the chapter not only is an overview of the software process, but a guide to much of the rest of the book. The chapter concludes with national and international initiatives to improve the software process.

We now examine the Unified Process.

3.1 The Unified Process

As stated at the beginning of this chapter, methodology is one component of a software process. The primary object-oriented methodology today is the **Unified Process**. As explained in Just in Case You Wanted to Know Box 3.2, the Unified "Process" is actually a methodology, but the name Unified Methodology already had been used as the name of the first version of the **Unified Modeling Language** (UML). The three precursors of the Unified Process (OMT, Booch's method, and Objectory) are no longer supported, and the other object-oriented methodologies have had little or no following. As a result, the Unified Process is usually the primary choice today for object-oriented software production. Fortunately, as will be demonstrated in Part B of this book, the Unified Process is an excellent object-oriented methodology in almost every way.

The Unified Process is not a specific series of steps that, if followed, will result in the construction of a software product. In fact, no such single "one size fits all" methodology could exist because of the wide variety of types of software products. For example, there are many different application domains, such as insurance, aerospace, and manufacturing. Also, a methodology for rushing a COTS package to market ahead of its competitors is different from one used to construct a high-security electronic funds transfer network. In addition, the skills of software professionals can vary widely.

Instead, the Unified Process should be viewed as an adaptable methodology. That is, it is modified for the specific software product to be developed. As will be seen in Part B, some features of the Unified Process are inapplicable to small- and even medium-scale software. However, much of the Unified Process is used for software products of all sizes. The emphasis in this book is on this common subset of the Unified Process, but aspects of the Unified Process applicable to only large-scale software also are discussed, to ensure that the issues that need to be addressed when larger software products are constructed are thoroughly appreciated.

3.2 Iteration and Incrementation within the Object-Oriented Paradigm

The object-oriented paradigm uses modeling throughout. A **model** is a set of UML diagrams that represent one or more aspects of the software product to be developed. (UML diagrams are introduced in Chapter 7.) Recall that UML stands for *Unified Modeling Language*. That is, UML is the tool that we use to represent (model) the target software product. A major reason for using a graphical representation like UML is best expressed by the old proverb, a picture is worth a thousand words. UML diagrams enable software professionals to communicate with one another more quickly and more accurately than if only verbal descriptions were used.

The object-oriented paradigm is an iterative-and-incremental methodology. Each workflow consists of a number of steps, and to carry out that workflow, the steps of the workflow are repeatedly performed until the members of the development team are satisfied that they have an accurate UML model of the software product they want to develop. That is, even the most experienced software professionals iterate and reiterate until they are finally satisfied that the UML diagrams are correct. The implication is that software engineers, no

Until recently, the most popular object-oriented software development methodologies were object modeling technique (OMT) [Rumbaugh et al., 1991] and Grady Booch's method [Booch, 1994]. OMT was developed by Jim Rumbaugh and his team at the General Electric Research and Development Center in Schenectady, New York, whereas Grady Booch developed his method at Rational, Inc., in Santa Clara, California. All object-oriented software development methodologies essentially are equivalent, so the differences between OMT and Booch's method are small. Nevertheless, there always was a friendly rivalry between the supporters of the two camps.

This changed in October 1994, when Rumbaugh joined Booch at Rational. The two methodologists immediately began to work together to develop a methodology that would combine OMT and Booch's method. When a preliminary version of their work was published, it was pointed out that they had not developed a methodology but merely a notation for representing an object-oriented software product. The name *Unified Methodology* was quickly changed to *Unified Modeling Language* (UML). In 1995, they were joined at Rational by Ivar Jacobson, author of the Objectory methodology. Booch, Jacobson, and Rumbaugh, affectionately called the "Three Amigos" (after the 1986 John Landis movie *Three Amigos!* with Chevy Chase and Steve Martin), then worked together. Version 1.0 of UML, published in 1997, took the software engineering world by storm. Until then, there had been no universally accepted notation for the development of a software product. Almost overnight UML was used all over the world. The Object Management Group (OMG), an association of the world's leading companies in object technology, took the responsibility for organizing an international standard for UML, so that every software professional would use the same version of UML, thereby promoting communication among individuals within an organization as well as companies worldwide. UML [Booch, Rumbaugh, and Jacobson, 1999] is today the unquestioned international standard notation for representing object-oriented software products.

An orchestral score shows which musical instruments are needed to play the piece, the notes each instrument is to play and when it is to play them, as well as a whole host of technical information such as the key signature, tempo, and loudness. Could this information be given in English, rather than a diagram? Probably, but it would be impossible to play music from such a description. For example, there is no way a pianist and a violinist could perform a piece described as follows: "The music is in march time, in the key of B minor. The first bar begins with the A above middle C on the violin (a quarter note). While this note is being played, the pianist plays a chord consisting of seven notes. The right hand plays the following four notes: E sharp above middle C . . ."

It is clear that, in some fields, a textual description simply cannot replace a diagram. Music is one such field; software development is another. And for software development, the best modeling language available today is UML.

Taking the software engineering world by storm with UML was not enough for the Three Amigos. Their next endeavor was to publish a complete software development methodology that unified their three separate methodologies. This unified methodology was first called the *Rational Unified Process* (RUP); *Rational* is in the name of the methodology not because the Three Amigos considered all other approaches to be irrational, but because at that time all three were senior managers at Rational, Inc. (Rational was bought by IBM in 2003). In their book on RUP [Jacobson, Booch, and Rumbaugh, 1999], the name *Unified Software Development Process* (USDP) was used. The term *Unified Process* is generally used today, for brevity.

matter how outstanding they may be, almost never get the various work products right the first time. How can this be?

The nature of software products is such that virtually everything has to be developed iteratively and incrementally. After all, software engineers are human, and therefore subject to Miller's Law (Section 2.5). That is, it is impossible to consider everything at the same time, so just seven or so chunks (units of information) are handled initially. Then, when the next set of chunks is considered, more knowledge about the target software product is gained, and the UML diagrams are modified in the light of this additional information. The process continues in this way until eventually the software engineers are satisfied that all the models for a given workflow are correct. In other words, initially the best possible UML diagrams are drawn in the light of the knowledge available at the beginning of the workflow. Then, as more knowledge about the real-world system being modeled is gained, the diagrams are made more accurate (iteration) and extended (incrementation). Accordingly, no matter how experienced and skillful a software engineer may be, he or she repeatedly iterates and increments until satisfied that the UML diagrams are an accurate representation of the software product to be developed.

Ideally, by the end of this book, the reader would have the software engineering skills necessary for constructing the large, complex software products for which the Unified Process was developed. Unfortunately, there are three reasons why this is not feasible.

1. Just as it is not possible to become an expert on calculus or a foreign language in one single course, gaining proficiency in the Unified Process requires extensive study and, more important, unending practice in object-oriented software engineering.
2. The Unified Process was created primarily for use in developing large, complex software products. To be able to handle the many intricacies of such software products, the Unified Process is itself large. It would be hard to cover every aspect of the Unified Process in a textbook of this size.
3. To teach the Unified Process, it is necessary to present a case study that illustrates the features of the Unified Process. To illustrate the features that apply to large software products, such a case study would have to be large. For example, just the specifications typically would take over 1000 pages.

For these three reasons, this book presents most, but not all, of the Unified Process.

The five **core workflows** of the Unified Process (requirements workflow, analysis workflow, design workflow, implementation workflow, and test workflow) and their challenges are now discussed.

3.3 The Requirements Workflow

Software development is expensive. The development process usually begins when the client approaches a development organization with regard to a software product that, in the opinion of the client, is either essential to the profitability of his or her enterprise or somehow can be justified economically. The aim of the **requirements workflow** is for the development organization to determine the client's needs. The first task of the development team is to acquire a basic understanding of the **application domain** (**domain** for short), that is, the specific environment in which the target software product is to operate. The domain could be banking, automobile manufacturing, or nuclear physics.

At any stage of the process, if the client stops believing that the software will be cost effective, development will terminate immediately. Throughout this chapter the assumption is made that the client feels that the cost is justified. Therefore, a vital aspect of software development is the **business case**, a document that demonstrates the cost-effectiveness of the target product. (In fact, the "cost" is not always purely financial. For example, military software often is built for strategic or tactical reasons. Here, the cost of the software is the potential damage that could be suffered in the absence of the weapon being developed.)

At an initial meeting between client and developers, the client outlines the product as he or she conceptualizes it. From the viewpoint of the developers, the client's description of the desired product may be vague, unreasonable, contradictory, or simply impossible to achieve. The task of the developers at this stage is to determine exactly what the client needs and to find out from the client what constraints exist.

- A major constraint is almost always the **deadline**. For example, the client may stipulate that the finished product must be completed within 14 months. In almost every application domain, it is now commonplace for a target software product to be mission critical. That is, the client needs the software product for core activities of his or her organization, and any delay in delivering the target product is detrimental to the organization.
- A variety of other constraints often are present, such as **reliability** (for example, the product must be operational 99 percent of the time, or the mean time between failures must be at least 4 months). Another common constraint is the size of the executable load image (for example, it has to run on the client's personal computer or on the hardware inside the satellite).
- The **cost** is almost invariably an important constraint. However, the client rarely tells the developers how much money is available to build the product. Instead, a common practice is that, once the specifications have been finalized, the client asks the developers to name their price for completing the project. Clients follow this bidding procedure in the hope that the amount of the developers' bid is lower than the amount the client has budgeted for the project.

The preliminary investigation of the client's needs sometimes is called **concept exploration**. In subsequent meetings between members of the development team and the client team, the functionality of the proposed product is successively refined and analyzed for technical feasibility and financial justification.

Up to now, everything seems to be straightforward. Unfortunately, the requirements workflow often is performed inadequately. When the product finally is delivered to the user, perhaps a year or two after the specifications have been signed off on by the client, the client may say to the developers, "I know that this is what I asked for, but it isn't really what I wanted." What the client asked for and, therefore, what the developers thought the client wanted, was not what the client actually *needed*. There can be a number of reasons for this predicament. First, the client may not truly understand what is going on in his or her own organization. For example, it is no use asking the software developers for a faster operating system if the cause of the current slow turnaround is a badly designed database. Or, if the client operates an unprofitable chain of retail stores, the client may ask for a financial management information system that reflects such items as sales, salaries, accounts payable, and accounts receivable. Such a product will be of little use if the real reason for the losses

is shrinkage (theft by employees and shoplifting). If that is the case, then a stock control system rather than a financial management information system is required.

But the major reason why the client frequently asks for the wrong product is that software is complex. If it is difficult for a software professional to visualize a piece of software and its functionality, the problem is far worse for a client who is barely computer literate. As will be shown in Chapter 11, the Unified Process can help in this regard; the many UML diagrams of the Unified Process assist the client in gaining the necessary detailed understanding of what needs to be developed.

3.4 The Analysis Workflow

The aim of the **analysis workflow** is to analyze and refine the requirements to achieve the detailed understanding of the requirements essential for developing a software product correctly and maintaining it easily. At first sight, however, there is no need for an analysis workflow. Instead, an apparently simpler way to proceed would be to develop a software product by continuing with further iterations of the requirements workflow until the necessary understanding of the target software product has been obtained.

The key point is that the output of the requirements workflow must be totally comprehended by the client. In other words, the artifacts of the requirements workflow must be expressed in the language of the client, that is, in a natural (human) language such as English, Armenian, or Zulu. But all natural languages, without exception, are somewhat imprecise and lend themselves to misunderstanding. For example, consider the following paragraph:

> A part record and a plant record are read from the database. If it contains the letter A directly followed by the letter Q, then calculate the cost of transporting that part to that plant.

At first sight, this requirement seems perfectly clear. But to what does *it* (the second word in the second sentence) refer: the part record, the plant record, or the database?

Ambiguities of this kind cannot arise if the requirements are expressed (say) in a mathematical notation. However, if a mathematical notation is used for the requirements, then the client is unlikely to understand much of the requirements. As a result, there may well be miscommunication between client and developers regarding the requirements, and consequently, the software product developed to satisfy those requirements may not be what the client needs.

The solution is to have two separate workflows. The requirements workflow is couched in the language of the client; the analysis workflow, in a more precise language that ensures that the design and implementation workflows are correctly carried out. In addition, more details are added during the analysis workflow, details not relevant to the client's understanding of the target software product but essential for the software professionals who will develop the software product. For example, the initial state of a statechart (Section 13.6) would surely not concern the client in any way but has to be included in the specifications if the developers are to build the target product correctly.

The specifications of the product constitute a contract. The software developers are deemed to have completed the contract when they deliver a product that satisfies the acceptance criteria of the specifications. For this reason, the specifications should not include imprecise terms like *suitable, convenient, ample*, or *enough*, or similar terms that

sound exact but in practice are equally imprecise, such as *optimal* or *98 percent complete*. Whereas contract software development can lead to a lawsuit, there is no chance of the specifications forming the basis for legal action when the client and developers are from the same organization. Nevertheless, even in the case of internal software development, the specifications always should be written as if they will be used as evidence in a trial.

More important, the specifications are essential for both testing and maintenance. Unless the specifications are precise, there is no way to determine whether they are correct, let alone whether the implementation satisfies the specifications. And it is hard to change the specifications unless some document states exactly what the specifications currently are.

When the Unified Process is used, there is no specification document in the usual sense of the term. Instead, a set of UML artifacts are shown to the client, as described in Chapter 13. These UML diagrams and their descriptions can obviate many (but by no means all) of the problems of the classical specification document.

One mistake that can be made by a classical analysis team is that the specifications are ambiguous; as previously explained, **ambiguity** is intrinsic to natural languages. **Incompleteness** is another problem in the specifications; that is, some relevant fact or requirement may be omitted. For instance, the specification document may not state what actions are to be taken if the input data contain errors. Moreover, the specification document may contain **contradictions**. For example, one place in the specification document for a product that controls a fermentation process states that if the pressure exceeds 35 psi, then valve M17 immediately must be shut. However, another place states that, if the pressure exceeds 35 psi, then the operator immediately must be alerted; only if the operator takes no remedial action within 30 seconds should valve M17 be shut automatically. Software development cannot proceed until such problems in the specifications have been corrected. As pointed out in the previous paragraph, many of these problems can be reduced by using the Unified Process. This is because UML diagrams together with descriptions of those diagrams are less likely to contain ambiguity, incompleteness, and contradictions.

Once the client has approved the specifications, detailed planning and estimating commences. No client authorizes a software project without knowing in advance how long the project will take and how much it will cost. From the viewpoint of the developers, these two items are just as important. If the developers underestimate the cost of a project, then the client pays the agreed-upon fee, which may be significantly less than the developers' actual cost. Conversely, if the developers overestimate what the project costs, then the client may turn down the project or have the job done by other developers whose estimate is more reasonable. Similar issues arise with regard to duration estimates. If the developers underestimate how long completing a project will take, then the resulting late delivery of the product, at best, results in a loss of confidence by the client. At worst, lateness penalty clauses in the contract are invoked, causing the developers to suffer financially. Again, if the developers overestimate how long it will take for the product to be delivered, the client may well award the job to developers who promise faster delivery.

For the developers, merely estimating the duration and total cost is not enough. The developers need to assign the appropriate personnel to the various workflows of the development process. For example, the implementation team cannot start until the relevant design artifacts have been approved by the software quality assurance (SQA) group, and the design team is not needed until the analysis team has completed its task. In other words, the developers have to plan ahead. A software project management plan (SPMP) must be

drawn up that reflects the separate workflows of the development process and shows which members of the development organization are involved in each task, as well as the deadlines for completing each task.

The earliest that such a detailed plan can be drawn up is when the specifications have been finalized. Before that time, the project is too amorphous for complete planning. Some aspects of the project certainly must be planned right from the start, but until the developers know exactly what is to be built, they cannot specify all aspects of the plan for building it.

Therefore, once the specifications have been approved by the client, preparation of the software project management plan commences. Major components of the plan are the **deliverables** (what the client is going to get), the **milestones** (when the client gets them), and the **budget** (how much it is going to cost).

The plan describes the software process in fullest detail. It includes aspects such as the life-cycle model to be used, the organizational structure of the development organization, project responsibilities, managerial objectives and priorities, the techniques and CASE tools to be used, and detailed schedules, budgets, and resource allocations. Underlying the entire plan are the duration and cost estimates; techniques for obtaining such estimates are described in Section 9.2.

The analysis workflow is described in Chapters 12 and 13: classical analysis techniques are described in Chapter 12, and object-oriented analysis is the subject of Chapter 13. A major artifact of the analysis workflow is the software project management plan. An explanation of how to draw up the SPMP is given in Sections 9.3 though 9.5.

Now the design workflow is examined.

3.5 The Design Workflow

The specifications of a product spell out *what* the product is to do; the design shows *how* the product is to do it. More precisely, the aim of the **design workflow** is to refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers.

As explained in Section 1.3, during the classical design phase, the design team determines the internal structure of the product. The designers decompose the product into **modules**, independent pieces of code with well-defined interfaces to the rest of the product. The interface of each module (that is, the arguments passed to the module and the arguments returned by the module) must be specified in detail. For example, a module might measure the water level in a nuclear reactor and cause an alarm to sound if the level is too low. A module in an avionics product might take as input two or more sets of coordinates of an incoming enemy missile, compute its trajectory, and invoke another module to advise the pilot as to possible evasive action. Once the team has completed the decomposition into modules (the **architectural design**), the **detailed design** is performed. For each module, algorithms are selected and data structures chosen.

Turning now to the object-oriented paradigm, the basis of that paradigm is the **class**, a specific type of module. Classes are extracted during the analysis workflow and designed during the design workflow. Consequently, the object-oriented counterpart of architectural design is performed as a part of the object-oriented analysis workflow, and the object-oriented counterpart of detailed design is part of the object-oriented design workflow.

The design team must keep a meticulous record of the design decisions that are made. This information is essential for two reasons.

1. While the product is being designed, a dead end will be reached at times and the design team must backtrack and redesign certain pieces. Having a written record of why specific decisions were made assists the team when this occurs and helps it get back on track.
2. Ideally, the design of the product should be open-ended, meaning future enhancements (postdelivery maintenance) can be done by adding new classes or replacing existing classes without affecting the design as a whole. Of course, in practice, this ideal is difficult to achieve. Deadline constraints in the real world are such that designers struggle against the clock to complete a design that satisfies the original specifications, without worrying about any later enhancements. If future enhancements (to be added after the product is delivered to the client) are included in the specifications, then these must be allowed for in the design, but this situation is extremely rare. In general, the specifications, and hence the design, deal with only present requirements. In addition, while the product is still being designed, there is no way to determine all possible future enhancements. Finally, if the design has to take *all* future possibilities into account, at best it will be unwieldy; at worst, it will be so complicated that implementation is impossible. So the designers have to compromise, putting together a design that can be extended in many reasonable ways without the need for total redesign. But, in a product that undergoes major enhancement, the time will come when the design simply cannot handle further changes. When this stage is reached, the product must be redesigned as a whole. The task of the redesign team is considerably easier if the team members are provided a record of the reasons for all the original design decisions.

3.6 The Implementation Workflow

The aim of the **implementation workflow** is to implement the target software product in the chosen implementation language(s). A small software product is sometimes implemented by the designer. In contrast, a large software product is partitioned into smaller subsystems, which are then implemented in parallel by coding teams. The subsystems, in turn, consist of **components** or **code artifacts** implemented by an individual programmer.

Usually, the only documentation given a programmer is the relevant design artifact. For example, in the case of the classical paradigm, the programmer is given the detailed design of the module he or she is to implement. The detailed design usually provides enough information for the programmer to implement the code artifact without too much difficulty. If there are any problems, they can quickly be cleared up by consulting the responsible designer. However, there is no way for the individual programmer to know if the architectural design is correct. Only when integration of individual code artifacts commences do the shortcomings of the design as a whole start coming to light.

Suppose that a number of code artifacts have been implemented and integrated and the parts of the product integrated so far appear to be working correctly. Suppose further that a programmer has correctly implemented artifact `a45`, but when this artifact is integrated with the other existing artifacts, the product fails. The cause of the failure lies not in artifact `a45` itself, but rather in the way that artifact `a45` interacts with the rest of the product, as

specified in the architectural design. Nevertheless, in this type of situation the programmer who just coded artifact a45 tends to be blamed for the failure. This is unfortunate, because the programmer has simply followed the instructions provided by the designer and implemented the artifact exactly as described in the detailed design for that artifact. The members of the programming team are rarely shown the "big picture," that is, the architectural design, let alone asked to comment on it. Although it is grossly unfair to expect an individual programmer to be aware of the implications of a specific artifact for the product as a whole, this unfortunately happens in practice all too often. This is yet another reason why it is so important for the design to be correct in every respect.

The correctness of the design (as well as the other artifacts) is checked as part of the test workflow.

3.7 The Test Workflow

As shown in Figure 2.4, in the Unified Process, testing is carried out in parallel with the other workflows, starting from the beginning. There are two major aspects to testing.

1. Every developer and maintainer is personally responsible for ensuring that his or her work is correct. Therefore, a software professional has to test and retest each artifact he or she develops or maintains.
2. Once the software professional is convinced that an artifact is correct, it is handed over to the software quality assurance group for independent testing, as described in Chapter 6.

The nature of the **test workflow** changes depending on the artifacts being tested. However, a feature important to all artifacts is traceability.

3.7.1 Requirements Artifacts

If the requirements artifacts are to be testable over the life cycle of the software product, then one property they must have is **traceability**. For example, it must be possible to trace every item in the analysis artifacts back to a requirements artifact and similarly for the design artifacts and the implementation artifacts. If the requirements have been presented methodically, properly numbered, cross-referenced, and indexed, then the developers should have little difficulty tracing through the subsequent artifacts and ensuring that they are indeed a true reflection of the client's requirements. When the work of the members of the requirements team is subsequently checked by the SQA group, traceability simplifies their task, too.

3.7.2 Analysis Artifacts

As pointed out in Chapter 1, a major source of faults in delivered software is faults in the specifications that are not detected until the software has been installed on the client's computer and used by the client's organization for its intended purpose. Both the analysis team and the SQA group must therefore check the analysis artifacts assiduously. In addition, they must ensure that the specifications are feasible, for example, that a specific hardware component is fast enough or that the client's current online disk storage capacity is adequate to handle the new product. An excellent way of checking the analysis artifacts is by means of a review. Representatives of the analysis team and of the client are present.

The meeting usually is chaired by a member of the SQA group. The aim of the review is to determine whether the analysis artifacts are correct. The reviewers go through the analysis artifacts, checking to see if there are any faults. Walkthroughs and inspections are two types of reviews, and they are described in Section 6.2.

We turn now to the checking of the detailed planning and estimating that takes place once the client has signed off on the specifications. Whereas it is essential that every aspect of the SPMP be meticulously checked by the development team and then by the SQA group, particular attention must be paid to the plan's duration and cost estimates. One way to do this is for management to obtain two (or more) independent estimates of both duration and cost when detailed planning starts, and then reconcile any significant differences. With regard to the SPMP document, an excellent way to check it is by a review similar to the review of the analysis artifacts. If the duration and cost estimates are satisfactory, the client will give permission for the project to proceed.

3.7.3 Design Artifacts

As mentioned in Section 3.7.1, a critical aspect of testability is traceability. In the case of the design, this means that every part of the design can be linked to an analysis artifact. A suitably cross-referenced design gives the developers and the SQA group a powerful tool for checking whether the design agrees with the specifications and whether every part of the specifications is reflected in some part of the design.

Design reviews are similar to the reviews that the specifications undergo. However, in view of the technical nature of most designs, the client usually is not present. Members of the design team and the SQA group work through the design as a whole as well as through each separate design artifact, ensuring that the design is correct. The types of faults to look for include logic faults, interface faults, lack of exception handling (processing of error conditions), and most important, nonconformance to the specifications. In addition, the review team always should be aware of the possibility that some analysis faults were not detected during the previous workflow. A detailed description of the review process is given in Section 6.2.

3.7.4 Implementation Artifacts

Each component should be tested while it is being implemented (desk checking); and after it has been implemented, it is run against test cases. This informal testing is done by the programmer. Thereafter, the quality assurance group tests the component methodically; this is termed **unit testing**. A variety of unit-testing techniques are described in Chapter 15.

In addition to running test cases, a code review is a powerful, successful technique for detecting programming faults. Here, the programmer guides the members of the review team through the listing of the component. The review team must include an SQA representative. The procedure is similar to reviews of specifications and designs described previously. As in all the other workflows, a record of the activities of the SQA group are kept as part of the test workflow.

Once a component has been coded, it must be combined with the other coded components so that the SQA group can determine whether the (partial) product as a whole functions correctly. The way in which the components are integrated (all at once or one at a time) and the specific order (from top to bottom or from bottom to top in the component interconnection diagram or class hierarchy) can have a critical influence on the quality of the resulting

product. For example, suppose the product is integrated bottom up. A major design fault, if present, will show up late, necessitating an expensive reimplementation. Conversely, if the components are integrated top down, then the lower-level components usually do not receive as thorough a testing as would be the case if the product were integrated bottom up. These and other problems are discussed in detail in Chapter 15. A detailed explanation is given there as to why coding and integration must be performed in parallel.

The purpose of this **integration testing** is to check that the components combine correctly to achieve a product that satisfies its specifications. During integration testing, particular care must be paid to testing the component interfaces. It is important that the number, order, and types of formal arguments match the number, order, and types of actual arguments. This strong type checking [van Wijngaarden et al., 1975] is best performed by the compiler and linker. However, many languages are not strongly typed. When such a language is used, members of the SQA group must check the interfaces.

When the integration testing has been completed (that is, when all the components have been coded and integrated), the SQA group performs **product testing**. The functionality of the product as a whole is checked against the specifications. In particular, the constraints listed in the specifications must be tested. A typical example is whether the response time has been met. Because the aim of product testing is to determine whether the specifications have been correctly implemented, many of the test cases can be drawn up once the specifications are complete.

Not only must the correctness of the product be tested but its robustness must also be tested. That is, intentionally erroneous input data are submitted to determine whether the product will crash or whether its error-handling capabilities are adequate for dealing with bad data. If the product is to be run together with the client's currently installed software, then tests also must be performed to check that the new product will have no adverse effect on the client's existing computer operations. Finally, a check must be made as to whether the source code and all other types of documentation are complete and internally consistent. Product testing is discussed in Section 15.21. On the basis of the results of the product test, a senior manager in the development organization decides whether the product is ready to be released to the client.

The final step in testing the implementation artifacts is **acceptance testing**. The software is delivered to the client, who tests it on the actual hardware, using actual data as opposed to test data. No matter how methodical the development team or the SQA group might be, there is a significant difference between test cases, which by their very nature are artificial, and actual data. A software product cannot be considered to satisfy its specifications until the product has passed its acceptance test. More details about acceptance testing are given in Section 15.22.

In the case of COTS software (Section 1.11), as soon as product testing is complete, versions of the complete product are supplied to selected possible future clients for testing on site. The first such version is termed the **alpha release**. The corrected alpha release is called the **beta release**; in general, the beta release is intended to be close to the final version. (The terms *alpha release* and *beta release* are generally applied to all types of software products, not just COTS.)

Faults in COTS software usually result in poor sales of the product and huge losses for the development company. So that as many faults as possible come to light as early as possible, developers of COTS software frequently give alpha or beta releases to selected companies, in

the expectation that on-site tests will uncover any latent faults. In return, the alpha and beta sites frequently are promised free copies of the delivered version of the software. Risks are involved for a company participating in alpha or beta testing. In particular, alpha releases can be fault laden, resulting in frustration, wasted time, and possible damage to databases. However, the company gets a head start in using the new COTS software, which can give it an advantage over its competitors. A problem occurs sometimes when software organizations use alpha testing by potential clients in place of thorough product testing by the SQA group. Although alpha testing at a number of different sites usually brings to light a large variety of faults, there is no substitute for the methodical testing that the SQA group can provide.

3.8 Postdelivery Maintenance

Postdelivery maintenance is not an activity grudgingly carried out after the product has been delivered and installed on the client's computer. On the contrary, it is an integral part of the software process that must be planned for from the beginning. As explained in Section 3.5, the design, as far as is feasible, should take future enhancements into account. Coding must be performed with future maintenance kept in mind. After all, as pointed out in Section 1.3, more money is spent on postdelivery maintenance than on all other software activities combined. It therefore is a vital aspect of software production. Postdelivery maintenance must never be treated as an afterthought. Instead, the entire software development effort must be carried out in such a way as to minimize the impact of the inevitable future postdelivery maintenance.

A common problem with postdelivery maintenance is documentation or, rather, lack of it. In the course of developing software against a time deadline, the original analysis and design artifacts frequently are not updated and, consequently, are almost useless to the maintenance team. Other documentation such as the database manual or the operating manual may never be written, because management decided that delivering the product to the client on time was more important than developing the documentation in parallel with the software. In many instances, the source code is the only documentation available to the maintainer. The high rate of personnel turnover in the software industry exacerbates the maintenance situation, in that none of the original developers may be working for the organization at the time when maintenance is performed. Postdelivery maintenance frequently is the most challenging aspect of software production for these reasons and the additional reasons given in Chapter 16.

Turning now to testing, there are two aspects to testing changes made to a product when postdelivery maintenance is performed. The first is checking that the required changes have been implemented correctly. The second aspect is ensuring that, in the course of making the required changes to the product, no other inadvertent changes were made. Therefore, once the programmer has determined that the desired changes have been implemented, the product must be tested against previous test cases to make certain that the functionality of the rest of the product has not been compromised. This procedure is called **regression testing**. To assist in regression testing, it is necessary that all previous test cases be retained, together with the results of running those test cases. Testing during postdelivery maintenance is discussed in greater detail in Chapter 16.

A major aspect of postdelivery maintenance is a record of all the changes made, together with the reason for each change. When software is changed, it has to be regression tested. Therefore, the regression test cases are a central form of documentation.

3.9 Retirement

The final stage in the software life cycle is **retirement**. After many years of service, a stage is reached when further postdelivery maintenance no longer is cost effective.

- Sometimes the proposed changes are so drastic that the design as a whole would have to be changed. In such a case, it is less expensive to redesign and recode the entire product.
- So many changes may have been made to the original design that interdependencies inadvertently have been built into the product, and even a small change to one minor component might have a drastic effect on the functionality of the product as a whole.
- The documentation may not have been adequately maintained, thereby increasing the risk of a regression fault to the extent that it would be safer to recode than maintain.
- The hardware (and operating system) on which the product runs is to be replaced; it may be more economical to reimplement from scratch than to modify.

In each of these instances the current version is replaced by a new version, and the software process continues.

True retirement, on the other hand, is a somewhat rare event that occurs when a product has outgrown its usefulness. The client organization no longer requires the functionality provided by the product, and it finally is removed from the computer.

3.10 The Phases of the Unified Process

Figure 3.1 differs from Figure 2.4 in that the labels of the increments have been changed. Instead of Increment A, Increment B, and so on, the four increments are now labeled Inception phase, Elaboration phase, Construction phase, and Transition phase. In other words, the phases of the Unified Process correspond to increments.

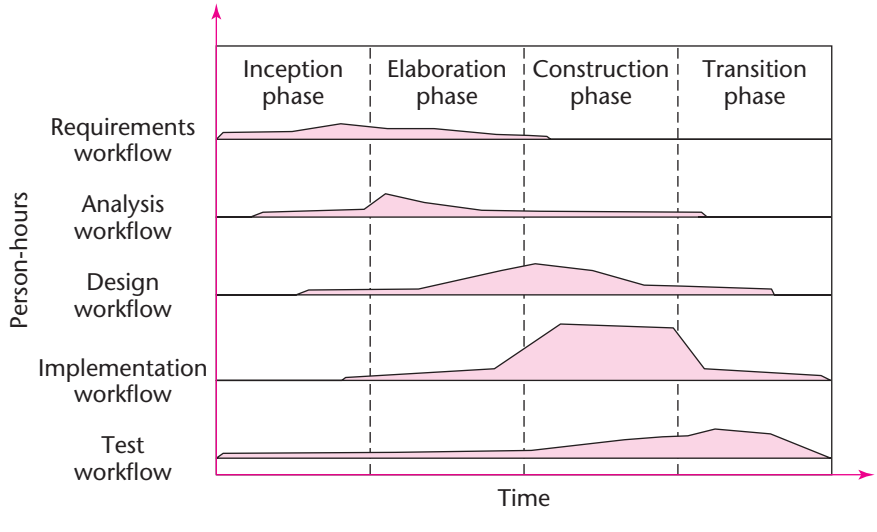


FIGURE 3.1
The core workflows and the phases of the Unified Process.

Although in theory the development of a software product could be performed in any number of increments, development in practice often seems to consist of four increments. The increments or phases are described in Sections 3.10.1 through 3.10.4, together with the deliverables of each phase, that is, the artifacts that should be completed by the end of that phase.

Every step performed in the Unified Process falls into one of five core workflows and *also* into one of four phases, the inception phase, elaboration phase, construction phase, and transition phase. The various steps of these four phases are already described in Sections 3.3 through 3.7. For example, building a business case is part of the requirements workflow (Section 3.3). It is also part of the inception phase. Nevertheless, each step has to be considered twice, as will be explained.

Consider the requirements workflow. To determine the client's needs, one of the steps is, as just stated, to build a business case. In other words, within the framework of the requirements workflow, building a business case is presented within a *technical* context. In Section 3.10.1, a description is presented of building a business case within the framework of the inception phase, the phase in which management decides whether or not to develop the proposed software product. That is, building a business case shortly is presented within an *economic* context (Section 1.2).

At the same time, there is no point in presenting each step twice, both times at the same level of detail. Accordingly, the inception phase is described in depth to highlight the difference between the technical context of the workflows and the economic context of the phases, but the other three phases are simply outlined.

3.10.1 The Inception Phase

The aim of the **inception phase** (first increment) is to determine whether it is worthwhile to develop the target software product. In other words, the primary aim of this phase is to determine whether the proposed software product is economically viable.

Two steps of the requirements workflow are to understand the domain and build a business model. Clearly, there is no way the developers can give any kind of opinion regarding a possible future software product unless they first understand the domain in which they are considering developing the target software product. It does not matter if the domain is a television network, a machine tool company, or a hospital specializing in liver disease—if the developers do not fully understand the domain, little reliance can be placed on what they subsequently build. Hence, the first step is to obtain domain knowledge. Once the developers have a full comprehension of the domain, the second step is to build a **business model**, that is, a description of the client's business processes. In other words, the first need is to understand the domain itself, and the second need is to understand precisely how the client organization operates in that domain.

Now the scope of the target project has to be delimited. For example, consider a proposed software product for a new highly secure ATM network for a nationwide chain of banks. The size of the business model of the banking chain as a whole is likely to be huge. To determine what the target software product should incorporate, the developers have to focus on only a subset of the business model, namely, the subset covered by the proposed software product. Therefore, delimiting the scope of the proposed project is the third step.

Now the developers can begin to make the initial business case. The questions that need to be answered before proceeding with the project include [Jacobson, Booch, and Rumbaugh, 1999]:

- Is the proposed software product cost effective? That is, will the benefits to be gained as a consequence of developing the software product outweigh the costs involved? How long will it take to obtain a return on the investment needed to develop the proposed software product? Alternatively, what will be the cost to the client if he or she decides not to develop the proposed software product? If the software product is to be sold in the marketplace, have the necessary marketing studies been performed?
- Can the proposed software product be delivered in time? That is, if the software product is delivered late to the market, will the organization still make a profit or will a competitive software product obtain the lion's share of the market? Alternatively, if the software product is to be developed to support the client organization's own activities (presumably including mission-critical activities), what is the impact if the proposed software product is delivered late?
- What risks are involved in developing the software product, and how can these risks be mitigated? Do the team members who will develop the proposed software product have the necessary experience? Is new hardware needed for this software product and, if so, is there a risk that it will not be delivered in time? If so, is there a way to mitigate that risk, perhaps by ordering backup hardware from another supplier? Are software tools (Chapter 5) needed? Are they currently available? Do they have all the necessary functionality? Is it likely that a COTS package (Section 1.11) with all (or almost all) the functionality of the proposed custom software product will be put on the market while the project is under way, and how can this be determined?

By the end of the inception phase the developers need answers to these questions so that the initial business case can be made.

The next step is to identify the risks. There are three major risk categories:

1. *Technical risks*. Examples of technical risks were just listed.
2. *Not getting the requirements right*. This risk can be mitigated by performing the requirements workflow correctly.
3. *Not getting the architecture right*. The architecture may not be sufficiently robust. (Recall from Section 2.7 that the architecture of a software product consists of the various components and how they fit together, and that the property of being able to handle extensions and changes without falling apart is its robustness.) In other words, while the software product is being developed, there is a risk that trying to add the next piece to what has been developed so far might require the entire architecture to be redesigned from scratch. An analogy would be to build a house of cards, only to find the entire edifice tumbling down when an additional card is added.

The risks need to be ranked so that the critical risks are mitigated first.

As shown in Figure 3.1, a small amount of the analysis workflow is performed during the inception phase. All that is usually done is to extract the information needed for the design of the architecture. This design work is also reflected in Figure 3.1.

Turning now to the implementation workflow, during the inception phase frequently no coding is performed. However, on occasion, it is necessary to build a proof-of-concept prototype to test the feasibility of part of the proposed software product, as described in Section 2.9.7.

The test workflow commences at the start of the inception phase. The major aim here is to ensure that the requirements are accurately determined.

Planning is an essential part of every phase. In the case of the inception phase, the developers have insufficient information at the beginning of the phase to plan the entire development, so the only planning done at the start of the project is the planning for the inception phase itself. For the same reason, a lack of information, the only planning that can meaningfully be done at the end of the inception phase is to plan for just the next phase, the elaboration phase.

Documentation, too, is an essential part of every phase. The deliverables of the inception phase include [Jacobson, Booch, and Rumbaugh, 1999]

- The initial version of the domain model.
- The initial version of the business model.
- The initial version of the requirements artifacts.
- A preliminary version of the analysis artifacts.
- A preliminary version of the architecture.
- The initial list of risks.
- The initial use cases (see Chapter 11).
- The plan for the elaboration phase.
- The initial version of the business case.

Obtaining the last item, the initial version of the business case, is the overall aim of the inception phase. This initial version incorporates a description of the scope of the software product as well as financial details. If the proposed software product is to be marketed, the business case includes revenue projections, market estimates, and initial cost estimates. If the software product is to be used in-house, the business case includes the initial cost–benefit analysis (Section 5.2).

3.10.2 The Elaboration Phase

The aim of the **elaboration phase** (second increment) is to refine the initial requirements, refine the architecture, monitor the risks and refine their priorities, refine the business case, and produce the software project management plan. The reason for the name *elaboration phase* is clear; the major activities of this phase are refinements or elaborations of the previous phase.

Figure 3.1 shows that these tasks correspond to all but completing the requirements workflow (Chapter 11), performing virtually the entire analysis workflow (Chapter 13), and then starting the design of the architecture (Section 8.5.4).

The deliverables of the elaboration phase include [Jacobson, Booch, and Rumbaugh, 1999]

- The completed domain model.
- The completed business model.
- The completed requirements artifacts.

- The completed analysis artifacts.
- An updated version of the architecture.
- An updated list of risks.
- The software project management plan (for the remainder of the project).
- The completed business case.

3.10.3 The Construction Phase

The aim of the **construction phase** (third increment) is to produce the first operational-quality version of the software product, the so-called beta release (Section 3.7.4). Consider Figure 3.1 again. Even though the figure is only a symbolic representation of the phases, it is clear that the emphasis in this phase is on implementation and testing the software product. That is, the various components are coded and unit tested. The code artifacts are then compiled and linked (integrated) to form subsystems, which are integration tested. Finally, the subsystems are combined into the overall system, which is product tested. This was described in Section 3.7.4.

The deliverables of the construction phase include [Jacobson, Booch, and Rumbaugh, 1999]

- The initial user manual and other manuals, as appropriate.
- All the artifacts (beta release versions).
- The completed architecture.
- The updated risk list.
- The software project management plan (for the remainder of the project).
- If necessary, the updated business case.

3.10.4 The Transition Phase

The aim of the **transition phase** (fourth increment) is to ensure that the client's requirements have indeed been met. This phase is driven by feedback from the sites at which the beta version has been installed. (In the case of a custom software product developed for a specific client, there is just one such site.) Faults in the software product are corrected. Also, all the manuals are completed. During this phase, it is important to try to discover any previously unidentified risks. (The importance of uncovering risks even during the transition phase is highlighted in Just in Case You Wanted to Know Box 3.3.)

The deliverables of the transition phase include [Jacobson, Booch, and Rumbaugh, 1999]

- All the artifacts (final versions).
- The completed manuals.

3.11 One- versus Two-Dimensional Life-Cycle Models

A classical life-cycle model (like the waterfall model of Section 2.9.2) is a one-dimensional model, as represented by the single axis in Figure 3.2(a). Underlying the Unified Process is a two-dimensional life-cycle model, as represented by the two axes in Figure 3.2(b).

Just in Case You Wanted to Know — Box 3.3

A real-time system frequently is more complex than most people, even its developers, realize. As a result, sometimes subtle interactions take place among components that even the most skilled testers usually would not detect. An apparently minor change therefore can have major consequences.

A famous example of this is the fault that delayed the first space shuttle orbital flight in April 1981 [Garman, 1981]. The space shuttle avionics are controlled by four identical synchronized computers. Also, an independent fifth computer is ready for backup in case the set of four computers fails. Two years earlier, a change had been made to the module that performs initialization before the avionics computers are synchronized. An unfortunate side effect of this change was that a record containing a time just slightly later than the current time was erroneously sent to the data area used for synchronization of the avionics computers. The time sent was sufficiently close to the actual time for this fault not to be detected. About 1 year later, the time difference was slightly increased, just enough to cause a 1 in 67 chance of a failure. Then, on the day of the first space shuttle launch, with hundreds of millions of people watching on television all over the world, the synchronization failure occurred and three of the four identical avionics computers were synchronized one cycle late relative to the first computer.

A fail-safe device that prevents the independent fifth computer from receiving information from the other four computers unless they are in agreement had the unanticipated consequence of preventing initialization of the fifth computer, and the launch had to be postponed. An all too familiar aspect of this incident was that the fault was in the initialization module, a module that apparently had no connection whatsoever with the synchronization routines.

Unfortunately, this was by no means the last real-time software fault affecting a space launch. For example, in April 1999, a Milstar military communications satellite was hurled into a uselessly low orbit at a cost of $1.2 billion; the cause was a software fault in the upper stage of the Titan 4 rocket [*Florida Today*, 1999].

Not just space launches are affected by real-time faults but landings, too. In May 2003, a Soyuz TMA-1 spaceship launched from the international space station landed 300 miles off course in Kazakhstan after a ballistic descent. The cause of the landing problems was, yet again, a real-time software fault [CNN.com, 2003].

The one-dimensional nature of the waterfall model is clearly reflected in Figure 2.3. In contrast, Figure 2.2 shows the evolution-tree model of the Winburg mini case study. This model is two-dimensional and should therefore be compared to Figure 3.2(b).

Are the additional complications of a two-dimensional model necessary? The answer was given in Chapter 2, but this is such an important issue that it is repeated here. During the development of a software product, in an ideal world, the requirements workflow would be completed before proceeding to the analysis workflow. Similarly, the analysis workflow would be completed before starting the design workflow, and so on. In reality, however, all but the most trivial software products are too large to handle as a single unit. Instead, the task has to be divided into increments (phases), and within each increment the developers have to iterate until they have completed the task under construction. As humans, we are limited by Miller's Law [Miller, 1956], which states that we can actively process only seven concepts at a time. We therefore cannot deal with software products as a whole, but instead we have to break those systems into subsystems. Even subsystems can be too large

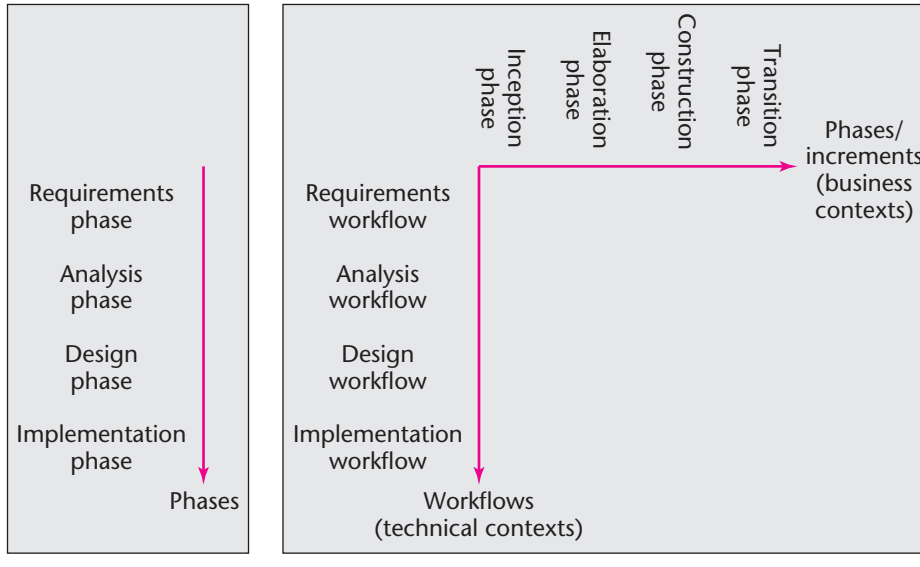


FIGURE 3.2
Comparison of (a) a classical one-dimensional life-cycle model and (b) the two-dimensional Unified Process life-cycle model.

at times—components may be all that we can handle until we have a fuller understanding of the software product as a whole.

The Unified Process is the best solution to date for treating a large problem as a set of smaller, largely independent subproblems. It provides a framework for incrementation and iteration, the mechanism used to cope with the complexity of large software products.

Another challenge that the Unified Process handles well is the inevitable changes. One aspect of this challenge is changes in the client's requirements while a software product is being developed, the so-called moving-target problem (Section 2.4).

For all these reasons, the Unified Process is currently the best methodology available. However, in the future, the Unified Process will doubtless be superseded by some new methodology. Today's software professionals are looking beyond the Unified Process to the next major breakthrough. After all, in virtually every field of human endeavor, the discoveries of today are often superior to anything that was put forward in the past. The Unified Process is sure to be superseded, in turn, by the methodologies of the future. The important lesson is that, based on *today's* knowledge, the Unified Process appears to be better than the other alternatives currently available.

The remainder of this chapter is devoted to national and international initiatives aimed at process improvement.

3.12 Improving the Software Process

Our global economy depends critically on computers and hence on software. For this reason, the governments of many countries are concerned about the software process. For example, in 1987, a task force of the U.S. Department of Defense (DoD) reported, "After two decades of largely unfulfilled promises about productivity and quality gains from

applying new software methodologies and technologies, industry and government organizations are realizing that their fundamental problem is the inability to manage the software process" [Brooks et al., 1987].

In response to this and related concerns, the DoD founded the Software Engineering Institute (SEI) and set it up at Carnegie Mellon University in Pittsburgh on the basis of a competitive procurement process. A major success of the SEI has been the capability maturity model (CMM) initiative. Related software process improvement efforts include the ISO 9000-series standards of the International Organization for Standardization, and ISO/IEC 15504, an international software improvement initiative involving more than 40 countries. We begin by describing the CMM.

3.13 Capability Maturity Models

The **capability maturity models** of the SEI are a related group of strategies for improving the software process, irrespective of the actual life-cycle model used. (The term **maturity** is a measure of the goodness of the process itself.) The SEI has developed CMMs for software (SW–CMM), for management of human resources (P–CMM; the *P* stands for "people"), for systems engineering (SE–CMM), for integrated product development (IPD–CMM), and for software acquisition (SA–CMM). There are some inconsistencies between the models and an inevitable level of redundancy. Accordingly, in 1997, it was decided to develop a single integrated framework for maturity models, capability maturity model integration (CMMI), which incorporates all five existing capability maturity models. Additional disciplines may be added to CMMI in the future [SEI, 2002].

For reasons of space, only one capability maturity model, SW–CMM, is examined here, and an overview of the P–CMM is given in Section 4.8. The SW–CMM was first put forward in 1986 by Watts Humphrey [Humphrey, 1989]. Recall that a software process encompasses the activities, techniques, and tools used to produce software. It therefore incorporates both technical and managerial aspects of software production. Underlying the SW–CMM is the belief that the use of new software techniques in itself will not result in increased productivity and profitability, because our problems are caused by how we manage the software process. The strategy of the SW–CMM is to improve the management of the software process in the belief that improvements in technique are a natural consequence. The resulting improvement in the process as a whole should result in better-quality software and fewer software projects that suffer from time and cost overruns.

Bearing in mind that improvements in the software process cannot occur overnight, the SW–CMM induces change incrementally. More specifically, five levels of maturity are defined, and an organization advances slowly in a series of small evolutionary steps toward the higher levels of process maturity [Paulk, Weber, Curtis, and Chrissis, 1995]. To understand this approach, the five levels now are described.

Maturity Level 1. Initial Level

At the **initial level**, the lowest level, essentially no sound software engineering management practices are in place in the organization. Instead, everything is done on an ad hoc basis. A specific project that happens to be staffed by a competent manager and a good software development team may be successful. However, the usual pattern is time and cost

overruns caused by a lack of sound management in general and planning in particular. As a result, most activities are responses to crises rather than preplanned tasks. In level-1 organizations, the software process is unpredictable, because it depends totally on the current staff; as the staff changes, so does the process. As a consequence, it is impossible to predict with any accuracy such important items as the time it will take to develop a product or the cost of that product.

It is unfortunate that the vast majority of software organizations all over the world are still level-1 organizations.

Maturity Level 2. Repeatable Level

At the **repeatable level**, basic software project management practices are in place. Planning and management techniques are based on experience with similar products; hence, the name *repeatable*. At level 2, measurements are taken, an essential first step in achieving an adequate process. Typical measurements include the meticulous tracking of costs and schedules. Instead of functioning in a crisis mode, as in level 1, managers identify problems as they arise and take immediate corrective action to prevent them from becoming crises. The key point is that, without measurements, it is impossible to detect problems before they get out of hand. Also, measurements taken during one project can be used to draw up realistic duration and cost schedules for future projects.

Maturity Level 3. Defined Level

At the **defined level**, the process for software production is fully documented. Both the managerial and technical aspects of the process are clearly defined, and continual efforts are made to improve the process wherever possible. Reviews (Section 6.2) are used to achieve software quality goals. At this level, it makes sense to introduce new technology, such as CASE environments (Section 5.8), to increase quality and productivity further. In contrast, "high tech" only makes the crisis-driven level-1 process even more chaotic.

Although a number of organizations have attained maturity levels 2 and 3, few have reached levels 4 or 5. The two highest levels therefore are targets for the future.

Maturity Level 4. Managed Level

A **managed-level** organization sets quality and productivity goals for each project. These two quantities are measured continually and corrective action is taken when there are unacceptable deviations from the goal. Statistical quality controls ([Deming, 1986], [Juran, 1988]) are in place to enable management to distinguish a random deviation from a meaningful violation of quality or productivity standards. (A simple example of a statistical quality control measure is the number of faults detected per 1000 lines of code. A corresponding objective is to reduce this quantity over time.)

Maturity Level 5. Optimizing Level

The goal of an **optimizing-level** organization is continuous process improvement. Statistical quality and process control techniques are used to guide the organization. The knowledge gained from each project is utilized in future projects. The process therefore incorporates a positive feedback loop, resulting in a steady improvement in productivity and quality.

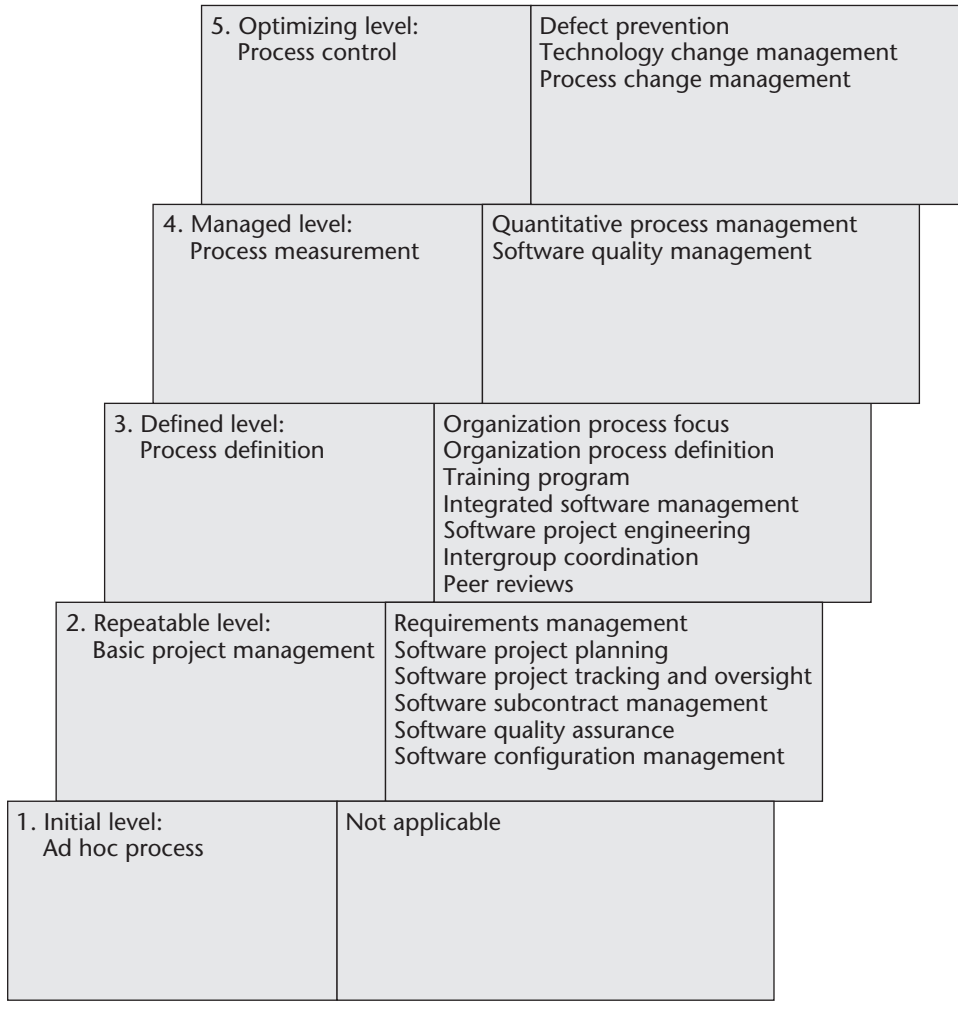


FIGURE 3.3
The five levels of the software capability maturity model and their key process areas (KPAs).

These five maturity levels are summarized in Figure 3.3, which also shows the key process areas (KPAs) associated with each maturity level. To improve its software process, an organization first attempts to gain an understanding of its current process and then formulates the intended process. Next, actions to achieve this process improvement are determined and ranked in priority. Finally, a plan to accomplish this improvement is drawn up and executed. This series of steps is repeated, with the organization successively improving its software process; this progression from level to level is reflected in Figure 3.3. Experience with the capability maturity model has shown that advancing a complete maturity level usually takes from 18 months to 3 years, but moving from level 1 to level 2 can sometimes take 3 or even 5 years. This is a reflection of how difficult it is to instill a methodical approach in an organization that up to now has functioned on a purely ad hoc and reactive basis.

For each maturity level, the SEI has highlighted a series of **key process areas (KPAs)** that an organization should target in its endeavor to reach the next maturity level. For example, as shown in Figure 3.3, the KPAs for level 2 (repeatable level) include configuration management (Section 5.10), software quality assurance (Section 6.1.1), project planning (Chapter 9), project tracking (Section 9.2.5), and requirements management (Chapter 11). These areas cover the basic elements of software management: Determine the client's needs (requirements management), draw up a plan (project planning), monitor deviations from that plan (project tracking), control the various pieces that make up the software product key process area (configuration management), and ensure that the product is fault free (quality assurance). Within each KPA is a group of between two and four related goals that, if achieved, result in that maturity level being attained. For example, one project planning goal is the development of a plan that appropriately and realistically covers the activities of software development.

At the highest level, maturity level 5, the KPAs include fault prevention, technology change management, and process change management. Comparing the KPAs of the two levels, it is clear that a level-5 organization is far in advance of one at level 2. For example, a level-2 organization is concerned with software quality assurance, that is, with detecting and correcting faults (software quality is discussed in more detail in Chapter 6). In contrast, the process of a level-5 organization incorporates fault prevention, that is, trying to ensure that no faults are in the software in the first place. To help an organization to reach the higher maturity levels, the SEI has developed a series of questionnaires that form the basis for an assessment by an SEI team. The purpose of the assessment is to highlight current shortcomings in the organization's software process and to indicate ways in which the organization can improve its process.

The CMM program of the Software Engineering Institute was sponsored by the U.S. Department of Defense. One of the original goals of the CMM program was to raise the quality of defense software by evaluating the processes of contractors who produce software for the DoD and awarding contracts to those contractors who demonstrate a mature process. The U.S. Air Force stipulated that any software development organization that wished to be an Air Force contractor had to conform to SW–CMM level 3 by 1998, and the DoD as a whole subsequently issued a similar directive. Consequently, pressure is put on organizations to improve the maturity of their software processes. However, the SW–CMM program has moved far beyond the limited goal of improving DoD software and is being implemented by a wide variety of software organizations that wish to improve software quality and productivity.

3.14 Other Software Process Improvement Initiatives

A different attempt to improve software quality is based on the **International Organization for Standardization** (ISO) 9000-series standards, a series of five related standards applicable to a wide variety of industrial activities, including design, development, production, installation, and servicing; ISO 9000 certainly is not just a software standard. Within the ISO 9000 series, standard **ISO 9001** [1987] for quality systems is the standard most applicable to software development. Because of the broadness of ISO 9001, ISO has published specific guidelines to assist in applying ISO 9001 to software: **ISO 9000-3** [1991]. (For more information on ISO, see Just in Case You Wanted to Know Box 1.4.)

ISO 9000 has a number of features that distinguish it from the CMM [Dawood, 1994]. ISO 9000 stresses documenting the process in both words and pictures to ensure consistency and comprehensibility. Also, the ISO 9000 philosophy is that adherence to the standard does not guarantee a high-quality product but rather reduces the risk of a poor-quality product. ISO 9000 is only part of a quality system. Also required are management commitment to quality, intensive training of workers, and setting and achieving goals for continual quality improvement. ISO 9000-series standards have been adopted by over 60 countries, including the United States, Japan, Canada, and the countries of the European Union (EU). This means, for example, that if a U.S. software organization wishes to do business with a European client, the U.S. organization must first be certified as ISO 9000 compliant. A certified registrar (auditor) has to examine the company's process and certify that it complies with the ISO standard.

Following their European counterparts, more and more U.S. organizations are requiring ISO 9000 certification. For example, General Electric Plastic Division insisted that 340 vendors achieve the standard by June 1993 [Dawood, 1994]. It is unlikely that the U.S. government will follow the EU lead and require ISO 9000 compliance for non-U.S. companies that wish to do business with organizations in the United States. Nevertheless, pressures both within the United States and from its major trading partners ultimately may result in significant worldwide ISO 9000 compliance.

ISO/IEC 15504 is an international process improvement initiative, like ISO 9000. The initiative was formerly known as **SPICE**, an acronym formed from Software Process Improvement Capability dEtermination. Over 40 countries actively contributed to the SPICE endeavor. SPICE was initiated by the British Ministry of Defence (MOD) with the long-term aim of establishing SPICE as an international standard (MOD is the UK counterpart of the U.S. DoD, which initiated the CMM). The first version of SPICE was completed in 1995. In July 1997, the SPICE initiative was taken over by a joint committee of the International Organization for Standardization and the International Electrotechnical Commission. For this reason, the name of the initiative was changed from SPICE to ISO/IEC 15504, or 15504 for short.

3.15 Costs and Benefits of Software Process Improvement

Does implementing software process improvement lead to increased profitability? Results indicate that this indeed is the case. For example, the Software Engineering Division of Hughes Aircraft in Fullerton, California, spent nearly $500,000 between 1987 and 1990 for assessments and improvement programs [Humphrey, Snider, and Willis, 1991]. During this 3-year period, Hughes Aircraft moved up from maturity level 2 to level 3, with every expectation of future improvement to level 4 and even level 5. As a consequence of improving its process, Hughes Aircraft estimated its annual savings to be on the order of $2 million. These savings accrued in a number of ways, including decreased overtime hours, fewer crises, improved employee morale, and lower turnover of software professionals.

Comparable results have been reported at other organizations. For example, the Equipment Division at Raytheon moved from level 1 in 1988 to level 3 in 1993. A twofold increase in productivity resulted, as well as a return of $7.70 for every dollar invested in the process improvement effort [Dion, 1993]. As a consequence of results like these, the

FIGURE 3.4 Results of 34 Motorola GED projects (MEASL stands for "million equivalent assembler source lines") [Diaz and Sligo, 1997]. (© 1997, IEEE.)

CMM Level	Number of Projects	Relative Decrease in Duration	Faults per MEASL Detected during Development	Relative Productivity
Level 1	3	1.0	—	—
Level 2	9	3.2	890	1.0
Level 3	5	2.7	411	0.8
Level 4	8	5.0	205	2.3
Level 5	9	7.8	126	2.8

capability maturity models are being applied rather widely within the U.S. software industry and abroad.

For example, Tata Consultancy Services in India used both the ISO 9000 framework and CMM to improve its process [Keeni, 2000]. Between 1996 and 2000, the errors in effort estimation decreased from about 50 percent to only 15 percent. The effectiveness of reviews (that is, the percentage of faults found during reviews) increased from 40 to 80 percent. The percentage of effort devoted to reworking projects dropped from nearly 12 percent to less than 6 percent.

Motorola Government Electronics Division (GED) has been actively involved in SEI's software process improvement program since 1992 [Diaz and Sligo, 1997]. Figure 3.4 depicts 34 GED projects, categorized according to the maturity level of the group that developed each project. As can be seen from the figure, the relative duration (that is, the duration of a project relative to a baseline project completed before 1992) decreased with increasing maturity level. Quality was measured in terms of faults per million equivalent assembler source lines (MEASL); to be able to compare projects implemented in different languages, the number of lines of source code was converted into the number of equivalent lines of assembler code [Jones, 1996]. As shown in Figure 3.4, quality increased with increasing maturity level. Finally, productivity was measured as MEASL per person-hour. For reasons of confidentiality, Motorola does not publish actual productivity figures, so Figure 3.4 reflects productivity relative to the productivity of a level-2 project. (No quality or productivity figures are available for the level-1 projects because these quantities cannot be measured when the team is at level 1.)

Galin and Avrahami [2006] analyzed 85 projects that had previously been reported in the literature as having advanced by one level as a consequence of implementing CMM. These projects were divided into four groups (CMM level 1 to level 2, CMM level 2 to level 3, and so on). For the four groups, the median fault density (number of faults per KLOC) decreased by between 26 and 63 percent. The median productivity (KLOC per person month) increased by between 26 and 187 percent. Median rework decreased by between 34 and 40 percent. The median project duration decreased by between 28 and 53 percent. Fault detection effectiveness (percentage of faults detected during development of the total detected project faults) increased as follows: For the three lowest groups, the median increased by between 70 and 74 percent, and 13 percent for the highest group (CMM level 4 to level 5). The return on investment varied between 120 and 650 percent, with a median value of 360 percent.

Just in Case You Wanted to Know — Box 3.4

There are constraints on the speed of hardware because electrons cannot travel faster than the speed of light. In a famous article entitled "No Silver Bullet," Brooks [1986] suggested that inherent problems exist in software production, and that these problems can never be solved because of analogous constraints on software. Brooks argued that intrinsic properties of software, such as its complexity, the fact that software is invisible and unvisualizable, and the numerous changes to which software is typically subjected over its lifetime, make it unlikely that there will ever be an order-of-magnitude increment (or "silver bullet") in software process improvement.

As a consequence of published studies such as those described in this section and those listed in the For Further Reading section of this chapter, more and more organizations worldwide are realizing that process improvement is cost effective.

An interesting side effect of the process improvement movement has been the interaction between software process improvement initiatives and software engineering standards. For example, in 1995 the International Organization for Standardization published ISO/IEC 12207, a full life-cycle software standard [ISO/IEC 12207, 1995]. Three years later, a U.S. version of the standard [IEEE/EIA 12207.0-1996, 1998] was published by the Institute of Electrical and Electronic Engineers (IEEE) and the Electronic Industries Alliance (EIA). This version incorporates U.S. software "best practices," many of which can be traced back to CMM. To achieve compliance with IEEE/EIA 12207, an organization must be at or near CMM capability level 3 [Ferguson and Sheard, 1998]. Also, ISO 9000-3 now incorporates parts of ISO/IEC 12207. This interplay between software engineering standards organizations and software process improvement initiatives surely will lead to even better software processes.

Another dimension of software process improvement appears in Just in Case You Wanted to Know Box 3.4.

Chapter Review

After some preliminary definitions, the Unified Process is introduced in Section 3.1. The importance of iteration and incrementation within the object-oriented paradigm is described in Section 3.2. Now the core workflows of the Unified Process are explained in detail; the requirements workflow (Section 3.3), analysis workflow (Section 3.4), design workflow (Section 3.5), implementation workflow (Section 3.6), and test workflow (Section 3.7). The various artifacts tested during the test workflow are described in Sections 3.7.1 through 3.7.4. Postdelivery maintenance is discussed in Section 3.8, and retirement in Section 3.9. The relationship between the workflows and the phases of the Unified Process is analyzed in Section 3.10, and a detailed description is given of the four phases of the Unified Process: the inception phase (Section 3.10.1), the elaboration phase (Section 3.10.2), the construction phase (Section 3.10.3), and the transition phase (Section 3.10.4). The importance of two-dimensional life-cycle models is discussed in Section 3.11.

The last part of the chapter is devoted to software process improvement (Section 3.12). Details are given of various national and international software improvement initiatives, including the capability maturity models (Section 3.13), and ISO 9000 and ISO/IEC 15504 (Section 3.14). The cost-effectiveness of software process improvement is discussed in Section 3.15.

For Further Reading

The March–April 2003 issue of *IEEE Software* contains a number of articles on the software process, including [Eickelmann and Anant, 2003], a discussion of statistical process control. Practical applications of statistical process control are described in [Weller, 2000] and [Florac, Carleton, and Barnard, 2000].

With regard to testing during each workflow, an excellent source is [Ammann and Offutt, 2008]. More specific references are given in Chapter 6 of this book and in the For Further Reading section at the end of that chapter.

A detailed description of the original SEI capability maturity model is given in [Humphrey, 1989]. Capability maturity model integration is described in [SEI, 2002]. Humphrey [1996] describes a personal software process (PSP); results of applying the PSP appear in [Ferguson et al., 1997]. The results of an experiment to measure the effectiveness of PSP training are presented in [Prechelt and Unger, 2000]. Extensions needed to the Unified Process for it to comply with CMM levels 2 and 3 are presented in [Manzoni and Price, 2003]. Implementing SW–CMM in small organizations is described in [Guerrero and Eterovic, 2004] and [Dangle, Larsen, Shaw, and Zelkowitz, 2005]. The July–August 2000 issue of *IEEE Software* has three papers on software process maturity, and there are four papers on the PSP in the November–December 2000 issue of *IEEE Software*.

A compendium of the results of many studies of process improvement appears in [Galin and Avrahami, 2006].

Pitterman [2000] describes how a group at Telecordia Technologies reached level 5; a study of how a Computer Sciences Corporation group attained level 5 appears in [McGarry and Decker, 2002]. Insights into the nature of level-5 organizations appear in [Eickelmann, 2003] and [Agrawal and Chari, 2007]. Cost–benefit analysis of software process improvement is described in [van Solingen, 2004]. An empirical investigation of the key factors for success in software process improvement is presented in [Dybå, 2005].

Problems of software product improvement appear in [Conradi and Fuggetta, 2002]. The results of 18 different software process improvement initiatives conducted at Ericsson are described in [Borjesson and Mathiassen, 2004]. A wealth of information on the CMM is available at the SEI CMM website www.sei.cmu.edu. An assessment of the success of the SPICE project can be found in [Rout et al., 2007]. The ISO/IEC 15504 (SPICE) home page is at www.sei.cmu.edu/technology/process/spice/.

A comparison between CMM and IEEE/EIA 12207 is given in [Ferguson and Sheard, 1998], and a comparison between CMM and Six Sigma (another approach to process improvement) appears in [Murugappan and Keeni, 2003]. An approach to implementing both ISO 9001 and CMMI appears in [Yoo et al., 2006]. A repository containing the results of some 400 software improvement experiments is described in [Blanco, Gutiérrez, and Satriani, 2001].

Key Terms

acceptance testing *86*
alpha release *86*
ambiguity *81*
analysis workflow *80*
application domain *78*
architectural design *82*
beta release *86*
budget *82*
business case *79*
business model *89*
capability maturity model (CMM) *95*
class *82*
code artifact *83*
component *83*
concept exploration *79*
construction phase *92*
contradiction *81*
core workflow *78*
cost *79*
deadline *79*
defined level *96*
deliverable *82*
design workflow *82*
detailed design *82*
domain *78*
elaboration phase *91*

implementation workflow *83*
inception phase *89*
incompleteness *81*
initial level *95*
integration testing *86*
International Organization for Standardization (ISO) *98*
ISO 9000-3 *98*
ISO 9001 *98*
ISO/IEC 15504 *99*
key process area (KPA) *98*
managed level *96*
maturity *95*
milestone *82*
model *76*
module *82*
optimizing level *96*
product testing *86*
regression testing *87*
reliability *79*
repeatable level *96*
requirements workflow *78*
retirement *88*
SPICE *99*
test workflow *84*
traceability *84*
transition phase *92*
Unified Modeling Language (UML) *76*
Unified Process *76*
unit testing *85*

Problems

3.1 Define the terms *software process* and *Unified Process*.

3.2 In the software engineering context, what is meant by the term *model*?

3.3 What is meant by a *phase* of the Unified Process?

3.4 Distinguish clearly between an ambiguity, a contradiction, and incompleteness.

3.5 Consider the requirements workflow and the analysis workflow. Would it make more sense to combine these two activities into one workflow than to treat them separately?

3.6 More testing is performed during the implementation workflow than in any other workflow. Would it be better to divide this workflow into two separate workflows, one incorporating the nontesting aspects, the other all the testing?

3.7 "Correctness is the responsibility of the SQA group." Discuss this statement.

3.8 Maintenance is the most important activity of software production and the most difficult to perform. Nevertheless, it is looked down on by many software professionals, and maintenance programmers often are paid less than developers. Do you think that this is reasonable? If not, how would you try to change it?

3.9 Why do you think that, as stated in Section 3.9, true retirement is a rare event?

3.10 Because of a fire at Elmer's Software, all documentation for a product is destroyed just before it is delivered. What is the impact of the resulting lack of documentation?

3.11 You have just purchased Antedeluvian Software Developers, an organization on the verge of bankruptcy because the company is at maturity level 1. What is the first step you will take to restore the organization to profitability?

3.12 Section 3.13 states that it makes little sense to introduce CASE environments within organizations at maturity level 1 or 2. Explain why this is so.

3.13 What is the effect of introducing CASE tools (as opposed to environments) within organizations with a low maturity level?

3.14 Maturity level 1, the initial level, refers to an absence of good software engineering management practices. Would it not have been better for the SEI to have labeled the initial level as maturity level 0?

3.15 (Term Project) What differences would you expect to find if the Chocoholics Anonymous product of Appendix A were developed by an organization at CMM level 1, as opposed to an organization at level 5?

3.16 (Readings in Software Engineering) Your instructor will distribute copies of [Agrawal and Chari, 2007]. Would you like to work in a level-5 organization? Explain your answer.

References

[Agrawal and Chari, 2007] M. Agrawal and K. Chari, "Software Effort, Quality, and Cycle Time: A Study of CMM Level 5 Projects," *IEEE Transactions on Software Engineering* **32** (March 2007), pp. 145–56.

[Ammann and Offutt, 2008] P. Ammann and J. Offutt, *Introduction to Software Testing,* Cambridge University Press, Cambridge, UK, 2008.

[Blanco, Gutiérrez, and Satriani, 2001] M. Blanco, P. Gutiérrez, and G. Satriani, "SPI Patterns: Learning from Experience," *IEEE Software* **18** (May–June 2001), pp. 28–35.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.

[Booch, Rumbaugh, and Jacobson, 1999] G. Booch, J. Rumbaugh, and I. Jacobson, *The UML Users Guide*, Addison-Wesley, Reading, MA, 1999.

[Borjesson and Mathiassen, 2004] A. Borjesson and L. Mathiassen, "Successful Process Implementation," *IEEE Software* **21** (July–August 2004), pp. 36–44.

[Brooks, 1986] F. P. Brooks, Jr., "No Silver Bullet," in: *Information Processing '86*, H.-J. Kugler (Editor), Elsevier North-Holland, New York, 1986; reprinted in *IEEE Computer* **20** (April 1987), pp. 10–19.

[Brooks et al., 1987] F. P. Brooks, V. Basili, B. Boehm, E. Bond, N. Eastman, D. L. Evans, A. K. Jones, M. Shaw, and C. A. Zraket, "Report of the Defense Science Board Task Force on Military Software," Department of Defense, Office of the Under Secretary of Defense for Acquisition, Washington, DC, September 1987.

[CNN.com, 2003] "Russia: Software Bug Made Soyuz Stray," edition.cnn.com/2003/TECH/space/05/06/soyuz.landing.ap/, May 6, 2003.

[Conradi and Fuggetta, 2002] R. Conradi and A. Fuggetta, "Improving Software Process Improvement," *IEEE Software* **19** (July–August 2002), pp. 92–99.

[Dangle, Larsen, Shaw, and Zelkowitz, 2005] K. C. Dangle, P. Larsen, M. Shaw, and M. V. Zelkowitz, "Software Process Improvement in Small Organizations: A Case Study," *IEEE Software* **22** (September–October 2005), pp. 68–75.

[Dawood, 1994] M. Dawood, "It's Time for ISO 9000," *CrossTalk* (March 1994), pp. 26–28.

[Deming, 1986] W. E. Deming, *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge, MA, 1986.

[Diaz and Sligo, 1997] M. Diaz and J. Sligo, "How Software Process Improvement Helped Motorola," *IEEE Software* **14** (September–October 1997), pp. 75–81.

[Dion, 1993] R. Dion, "Process Improvement and the Corporate Balance Sheet," *IEEE Software* **10** (July 1993), pp. 28–35.

[Dybå, 2005] T. Dybå, "An Empirical Investigation of the Key Factors for Success in Software Process Improvement," *IEEE Transactions in Software Engineering* **31** (May 2005), pp. 410–24.

[Eickelmann, 2003] N. Eickelmann, "An Insider's View of CMM Level 5," *IEEE Software* **20** (July–August 2003), pp. 79–81.

[Eickelmann and Anant, 2003] N. Eickelmann and A. Anant, "Statistical Process Control: What You Don't Know Can Hurt You!" *IEEE Software* **20** (March–April 2003), pp. 49–51.

[Ferguson and Sheard, 1998] J. Ferguson and S. Sheard, "Leveraging Your CMM Efforts for IEEE/EIA 12207," *IEEE Software* **15** (September–October 1998), pp. 23–28.

[Ferguson et al., 1997] P. Ferguson, W. S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya, "Results of Applying the Personal Software Process," *IEEE Computer* **30** (May 1997), pp. 24–31.

[Florac, Carleton, and Barnard, 2000] W. A. Florac, A. D. Carleton, and J. Barnard, "Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process," *IEEE Software* **17** (July–August 2000), pp. 97–106.

[*Florida Today*, 1999] "Milstar Satellite Lost during Air Force Titan 4b Launch from Cape," *Florida Today*, www.floridatoday.com/space/explore/uselv/titan/b32/, June 5, 1999.

[Galin and Avrahami, 2006] D. Galin and M. Avrahami, "Are CMM Program Investments Beneficial? Analyzing Past Studies," *IEEE Software* **23** (November–December 2006), pp. 81–87.

[Garman, 1981] J. R. Garman, "The 'Bug' Heard 'Round the World," *ACM SIGSOFT Software Engineering Notes* **6** (October 1981), pp. 3–10.

[Guerrero and Eterovic, 2004] F. Guerrero and Y. Eterovic, "Adopting the SW-CMM in a Small IT Organization," *IEEE Software* **21** (July–August 2004), pp. 29–35.

[Humphrey, 1989] W. S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.

[Humphrey, 1996] W. S. Humphrey, "Using a Defined and Measured Personal Software Process," *IEEE Softwar*e **13** (May 1996), pp. 77–88.

[Humphrey, Snider, and Willis, 1991] W. S. Humphrey, T. R. Snider, and R. R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software* **8** (July 1991), pp. 11–23.

[IEEE/EIA 12207.0-1996, 1998] "IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995," Institute of Electrical and Electronic Engineers, Electronic Industries Alliance, New York, 1998.

[ISO 9000-3, 1991] "ISO 9000-3, Guidelines for the Application of ISO 9001 to the Development, Supply, and Maintenance of Software," International Organization for Standardization, Geneva, 1991.

[ISO 9001, 1987] "ISO 9001, Quality Systems—Model for Quality Assurance in Design/Development, Production, Installation, and Servicing," International Organization for Standardization, Geneva, 1987.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[Jacobson, Booch, and Rumbaugh, 1999] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Jones, 1996] C. Jones, *Applied Software Measurement,* McGraw-Hill, New York, 1996.

[Juran, 1988] J. M. Juran, *Juran on Planning for Quality*, Macmillan, New York, 1988.

[Keeni, 2000] G. Keeni, "The Evolution of Quality Processes at Tata Consultancy Services," *IEEE Software* **17** (July–August 2000), pp. 79–88.

[Manzoni and Price, 2003] L. V. Manzoni and R. T. Price, "Identifying Extensions Required by RUP (Rational Unified Process) to Comply with CMM (Capability Maturity Model) Levels 2 and 3," *IEEE Transactions on Software Engineering* **29** (February 2003), pp. 181–92.

[McGarry and Decker, 2002] F. McGarry and B. Decker, "Attaining Level 5 in CMM Process Maturity," *IEEE Software* **19** (2002), pp. 87–96.

[Miller, 1956] G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review* **63** (March 1956), pp. 81–97. Reprinted in: www.well.com/user/smalin/miller.html.

[Murugappan and Keeni, 2003] M. Murugappan and G. Keeni, "Blending CMM and Six Sigma to Meet Business Goals," *IEEE Software* **20** (March–April 2003), pp. 42–48.

[Paulk, Weber, Curtis, and Chrissis, 1995] M. C. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, MA, 1995.

[Pitterman, 2000] B. Pitterman, "Telecordia Technologies: The Journey to High Maturity," *IEEE Software* **17** (July–August 2000), pp. 89–96.

[Prechelt and Unger, 2000] L. Prechelt and B. Unger, "An Experiment Measuring the Effects of Personal Software Process (PSP) Training," *IEEE Transactions on Software Engineering* **27** (May 2000), pp. 465–72.

[Rout et al., 2007] T. P. Rout, K. El Emam, M. Fusani, D. Goldenson, and H.-W. Jung, "SPICE in Retrospect: Developing a Standard for Process Assessment," *Journal of Systems and Software* **80** (September 2007), pp. 1483–93.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[SEI, 2002] "CMMI Frequently Asked Questions (FAQ)," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, June 2002.

[van Solingen, 2004] R. van Solingen, "Measuring the ROI of Software Process Improvement," *IEEE Software* **21** (May–June 2004), pp. 32–38.

[van Wijngaarden et al., 1975] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, "Revised Report on the Algorithmic Language ALGOL 68," *Acta Informatica* **5** (1975), pp. 1–236.

[Weller, 2000] E. F. Weller, "Practical Applications of Statistical Process Control," *IEEE Software* **18** (May–June 2000), pp. 48–55.

[Yoo et al., 2006] C. Yoo, J. Yoon, B. Lee, C. Lee, J. Lee, S. Hyun, and C.Wu, "A Unified Model for the Implementation of Both ISO 9001:2000 and CMMI by ISO-Certified Organizations," *Journal of Systems and Software* **79** (July 2006), pp. 954–61.