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

Just in Case You Wanted to Know Box 3.2

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.

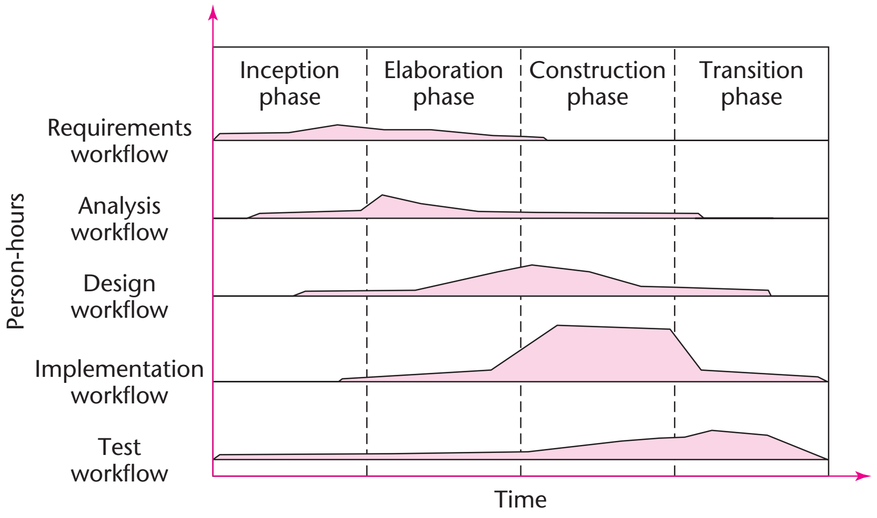

FIGURE 3.1 The core workflows and the phases of the Unified Process.

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

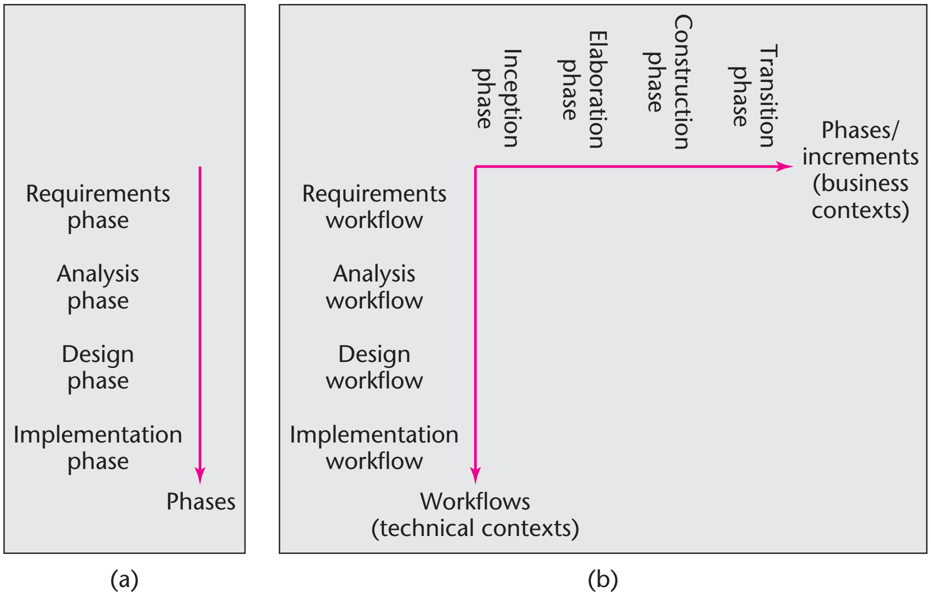

FIGURE 3.2 Comparison of (a) a classical one-dimensional life-cycle model and (b) the two-dimensional Unified Process life-cycle model.

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.

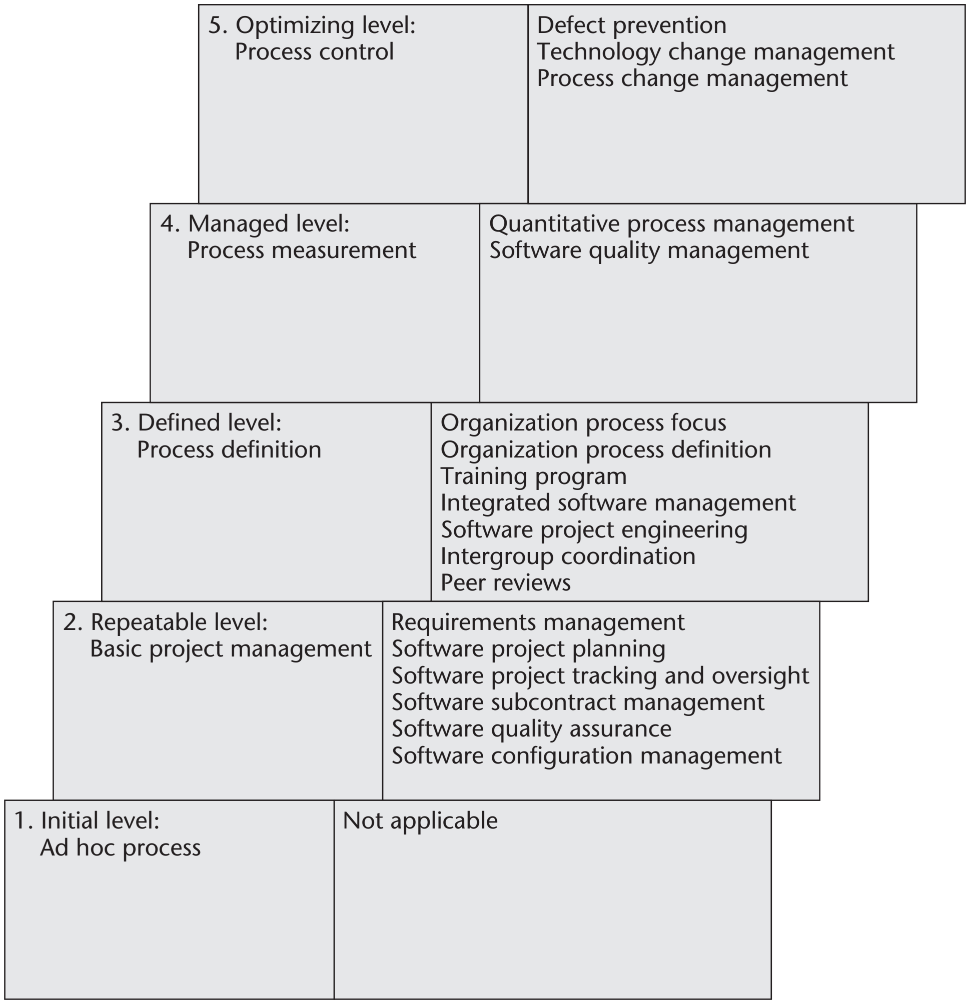
5. Optimizing level:
Process control
Defect prevention
Technology change management
Process change management
4. Managed level:
Process measurement
Quantitative process management
Software quality management
3. Defined level:
Process definition
Organization process focus
Organization process definition
Training program
Integrated software management
Software project engineering
Intergroup coordination
Peer reviews
2. Repeatable level:
Basic project management
Requirements management
Software project planning
Software project tracking and oversight
Software subcontract management
Software quality assurance
Software configuration management
1. Initial level:
Ad hoc process
Not applicable

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*

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 Software* **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.