

Object-Oriented Software Engineering

An Agile Unified Methodology

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

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

2.5.7 Agile Processes

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

Agile Manifesto

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

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

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

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

¹See www.agilemanifesto.org.

A Team Software Process Script

Purpose		To guide a team through developing a software product
Entry Criteria		<ul style="list-style-type: none"> • An instructor guides and supports one or more five-student teams. • The students are all PSP trained. • The instructor has the needed materials, facilities, and resources to support the teams. • The instructor has described the overall product objectives.
General		<p>The PSP process is designed to support three team modes. Follow the scripts that apply:</p> <ol style="list-style-type: none"> 1. Develop a small- to medium-sized software product in two or three development cycles. 2. Develop a smaller product in a single cycle. 3. Produce a product element, such as a requirements, design, or a test plan, in part of one cycle.
Week	Step	Activities
1	Review	<ul style="list-style-type: none"> • Course introduction and PSP review. • Read preface, introduction, and this chapter, focus on the PSP section.
2	LAU1	<ul style="list-style-type: none"> • Review course objectives and assign student teams and roles. • Read TSP and overview of the agile unified methodology in this chapter.
	STRAT1	<ul style="list-style-type: none"> • Produce the conceptual design, establish the development strategy, make size estimates, and assess risk. • Apply a software architectural design style (in most cases the N-tier architecture). • Read architectural design, and project management chapter, focus on estimation and risk management sections.
3	PLAN1, REQ1	<ul style="list-style-type: none"> • Define and inspect requirements, focus on high-priority requirements. • Derive use cases from the requirements, produce use case diagrams and traceability matrix, specify high-level use cases. • Allocate the use cases to the cycles, produce allocation matrix. • Review the use cases, use diagrams, high-level use case specifications, and matrices. • Read system engineering, software requirements elicitation, and quality assurance chapters, focus on requirements related sections, read deriving use cases chapter.
4	REQ1, DES1	<ul style="list-style-type: none"> • Perform cycle 1 domain modeling (brainstorming, domain concept classification, and domain model visualization). • Specify cycle 1 expanded use cases, produce use case based test cases. • Review domain model, expanded use cases, and use case based test cases. • Read domain modeling, actor-system interaction modeling, and software testing chapters (use case based testing).
5	DES1	<ul style="list-style-type: none"> • Produce and review cycle 1 scenarios, scenario tables, and sequence diagrams. • Derive and inspect cycle 1 design class diagram (DCD), and user interface design. • Read object interaction modeling, deriving design class diagram, and user interface design chapters.
6	IMP1	<ul style="list-style-type: none"> • Conduct cycle 1 test driven development (maybe combined with pair-programming) to fulfill 100% branch coverage. • Review unit test cases and code. • Read implementation, quality assurance, and software testing chapters.
7	TEST1	<ul style="list-style-type: none"> • Build, and integrate cycle 1, run use case based test cases. • Demonstrate cycle 1 software to the customer and users, solicit and record feedback. • Produce user documentation for cycle 1.
8	PM1	<ul style="list-style-type: none"> • Conduct a postmortem and write the cycle 1 final report. • Produce role and team evaluations for cycle 1.
	LAU2	<ul style="list-style-type: none"> • Re-form teams and roles for cycle 2.
	STRAT2, PLAN2, REQ2	<ul style="list-style-type: none"> • Produce the strategy and plan for cycle 2, assess risks. • Update and review requirements, domain model, use cases, traceability matrix, and allocation matrix.
9	DES2	<ul style="list-style-type: none"> • Apply GRASP patterns, and update and review cycle 1 sequence diagrams. • Produce and inspect cycle 2 expanded use cases and use case based test cases. • Apply GRASP, and produce and review cycle 2 scenarios, scenario tables and sequence diagrams. • Read applying responsibility assignment patterns chapter.

FIGURE 2.10 TSP development script

10	IMP2	<ul style="list-style-type: none"> • Test driven develop and inspect cycle 2, accomplish 100% branch coverage. • Review unit test cases and code.
	TEST2	<ul style="list-style-type: none"> • Build, integrate, and test cycle 2, demonstrate cycle 2 software to the customer and users, solicit and record feedback. • Produce user documentation for cycle 2.
11	PM2	<ul style="list-style-type: none"> • Conduct a postmortem and write the cycle 2 final report. • Produce role and team evaluations for cycle 2.
	LAU3	<ul style="list-style-type: none"> • Reform teams and roles for cycle 3.
12	STRAT3, PLAN3, REQ3	<ul style="list-style-type: none"> • Produce the strategy and plan for cycle 3, assess risks. • Update and review requirements, domain model, use cases, traceability matrix, and allocation matrix.
	DES3	<ul style="list-style-type: none"> • Apply situation specific or Gang of Four patterns, update cycle 1 and cycle 2 design diagrams. • Produce and inspect cycle 3 expanded use cases and use case based test cases. • Produce and review cycle 3 sequence diagrams (situation-specific patterns are applied). • Read applying situation-specific patterns chapter.
13	IMP3	<ul style="list-style-type: none"> • Test driven develop and inspect cycle 3, accomplish 100% branch coverage. • Review unit test cases and code.
	TEST3	<ul style="list-style-type: none"> • Build, integrate, and test cycle 3, demonstrate finished product to the customer and users, solicit and record feedback. • Produce and review user's manual for the finished product.
14	PM3	<ul style="list-style-type: none"> • Conduct a postmortem and write the cycle 3 final report. • Produce role and team evaluations for cycle 3. • Review the product produced and the processes used, identify lessons learned and propose process improvements.
Exit Criteria		<ul style="list-style-type: none"> • Completed product or product element and user documentation. • Completed and updated project notebook. • Documented team evaluations and cycle reports.

FIGURE 2.10 (Continued)

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

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

2. Agility values working software over comprehensive documentation.

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

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

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

3. Agility values customer collaboration over contract negotiation.

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

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

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

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

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

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

Agile Principles

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

1. *Active user involvement is imperative.*

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

Active user involvement means that representatives from the user community interact with the development team closely and as frequently as needed. For example, a couple of knowledgeable user representatives are assigned to the project. They stay and work with the team or visit the team regularly several times a week. These arrangements greatly improve the communication and understanding between the team and the users. These, in turn, ensure that requirement misconceptions are corrected early, users' feedback is addressed properly and timely, and decisions about the system are made with the users. All these imply that real requirements are identified and prioritized, and the system is built to meet users' expectations.

2. *The team must be empowered to make decisions.*

Agile development values individuals and interactions over processes and tools. This principle realizes this. That is, team members are required and encouraged to make decisions and take responsibility and ownership. To be able to do this, the team members are required to work as a team and interact with each other and the users throughout the project.

3. *Requirements evolve but the timescale is fixed.*

Unlike conventional approaches that freeze the requirements, agile processes are designed to welcome change. This principle means that the scope of work is allowed to evolve to cope with requirements change, but the agreed time frame and budget are fixed. This means that new or modified requirements are accommodated at the expense of other requirements. That is, the extra effort is compensated by giving up other requirements that are not mission critical.

4. *Capture requirements at a high level; lightweight and visual.*

Agile development values working software over comprehensive documentation. After all, the bottom line is to deliver the working system, not the analysis and design documentation. To accomplish this, agile methods capture barely enough of the requirements with user stories, features, or use cases written on small-size story cards. These are visualized using storyboards or sequences of screen shots, sketches, or other visual means to show how the user would interact with the system. These techniques avoid heavy documentation and make it easy to

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

5. *Develop small, incremental releases and iterate.*

Agile projects develop and deploy the system in bite-size increments to deliver the use cases, user stories, or features iteratively. This arrangement has several advantages: project progress is visible, the users only need to learn a few new features at a time, it is easier for the users to provide feedback, and small increments reduce risks of project failure.

6. *Focus on frequent delivery of software products.*

Before agile development, there are iterative approaches such as the spiral process and the unified process. Agile processes differ from their predecessors in frequent delivery of the software system in small increments. Different agile methods suggest different iteration lengths, which range from daily to three months. For example, Dynamic Systems Development Method (DSDM) suggests two to six weeks while Extreme Programming (XP) uses one to four weeks. An iteration in Scrum is called a sprint and is usually set to 30 days. The iteration duration of the methodology presented in this book can range from two weeks to three months.

7. *Complete each feature before moving on to the next.*

This principle means that each feature must be 100% implemented and thoroughly tested before moving onto the next. The challenge here is that how do we know that the feature is thoroughly tested? Test-driven development (TDD) and test coverage criteria provide a solution. TDD requires that tests for each feature must be written before implementation. Test coverage criteria define the coverage requirements that the tests must satisfy. For example, the 100% branch coverage criterion is used by many companies. It requires that each branch of each conditional statement of the source code must be tested at least once.

8. *Apply the 80-20 rule.*

This is also referred to as the "good enough is enough" rule. The rule is based on the belief that 80% of the work or result is produced by 20% of the system functionality. Therefore, priority should be given to the 20% of the requirements that will produce the 80% of work or result. This principle advises the development team to direct the customer and users to identify and prioritize such requirements. The rule also reminds team members of the diminishing return associated with the final extra miles. This applies to features that are nice to have, and performance optimization that is not really needed, and so forth. For example, an optimal algorithm may not be worth the extra implementation effort if a simpler algorithm is fast enough for the data to be processed.

9. *Testing is integrated throughout the project life cycle; test early and often.*

This principle and principles 5–7 complement each other. That is, testing is an integral part of frequent delivery of completely implemented small increments of the system. This principle is supported by test tools such as JUnit, a Java class unit testing and regression testing tool. Using such a tool, a programmer needs to specify how to invoke the feature to be tested and how to evaluate the test result. The tool will generate the tests, run the tests, and check the test result, all automatically. The tests can be run as often as desired.

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

2.6 SOFTWARE DEVELOPMENT METHODOLOGY

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

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

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

2.6.1 Difference between Process and Methodology

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

2.7 AGILE METHODS

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

2.7.1 Dynamic Systems Development Method

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

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

AUM*	DSDM	SCRUM	FDD	XP
Key Features <ul style="list-style-type: none"> • A cookbook for teamwork using UML • For beginners and seasoned developers • Suitable for agile or plan-driven, large or small team projects 	<ul style="list-style-type: none"> • A framework that works with Rational Unified Process and XP • Base on 80-20 principle • Suitable for agile or plan-driven projects 	<ul style="list-style-type: none"> • Include Scrum Master, Product Owner, and Team Roles • 15-minute daily status meeting to improve communication • Team retrospective to improve process 	<ul style="list-style-type: none"> • Feature-driven and model-driven • Configuration management, review and inspection, and regular builds • Suitable for agile or plan-driven projects 	<ul style="list-style-type: none"> • Anyone can change any code anywhere at any time • Integration and build many times a day whenever a task is completed • Work ≤ 40 hours a week
Life-Cycle Activities Preplanning <ol style="list-style-type: none"> 1. Acquire and prioritize requirements 2. Derive use cases 3. Assign use cases to increments 4. Architectural design During Each iteration: <ol style="list-style-type: none"> 1. Accommodating change 2. Domain modeling 3. System-actor interaction modeling 4. Behavior modeling 5. Design class diagram 6. Test-driven development/pair programming 7. Integration testing 8. Deployment 	Feasibility Study <ol style="list-style-type: none"> 1. Assess suitability of DSDM for project 2. Identify risks Business Study <ol style="list-style-type: none"> 1. Produce prioritized requirements 2. Architectural design 3. Risk resolution Functional Model Iteration <ol style="list-style-type: none"> 1. Identify prototype functionality 2. Build, review, and approve prototype Design and Build Iteration <ol style="list-style-type: none"> 1. Build system 2. Conduct beta test Implementation <ol style="list-style-type: none"> 1. Deploy system 2. Assess impact to business 	Release Planning Meeting <ol style="list-style-type: none"> 1. Identify and prioritize requirements (<i>product backlog</i>) 2. Identify top-priority requirements that can be delivered within an increment called a sprint 3. Identify sprint development activities Sprint Iteration: <ol style="list-style-type: none"> 1. Sprint planning meeting to determine what and how to build next 2. Daily Scrum meeting for team members to report status Sprint Review Meeting <ol style="list-style-type: none"> 1. Increment demo 2. Team retrospective Deployment	Develop Overall Model <ol style="list-style-type: none"> 1. System walkthrough 2. Develop small group models 3. Derive overall model 4. Produce model description Build Feature List <ol style="list-style-type: none"> 1. Identify business activities to be automated 2. Identify features of business activities Plan by Feature <ol style="list-style-type: none"> 1. Schedule development of business activities 2. Assign business activities to chief programmers 3. Assign classes to team members Design by Feature <ol style="list-style-type: none"> 1. Produce sequence diagrams to show the interaction of objects using features 2. Encapsulate features to form classes to be implemented Build by Feature Implement the classes Deployment	Exploration <ol style="list-style-type: none"> 1. Collect information about the application 2. Conduct feasibility study Planning <ol style="list-style-type: none"> 1. Determine the stories for the next release 2. Plan for the next release Iterations to First Release <ol style="list-style-type: none"> 1. Define/modify architecture 2. Select and implement stories for each iteration 3. Perform functional tests by customer Productionizing <ol style="list-style-type: none"> 1. Evaluate and improve system performance 2. Certify and test system for production use Maintenance <ol style="list-style-type: none"> 1. Improve the current release 2. Repeat process with each new release Death <ol style="list-style-type: none"> 1. Produce system documentation if project is done, or 2. Replace system if maintenance is too costly

* AUM: the agile unified methodology described in this book

FIGURE 2.12 Summary of some agile methods

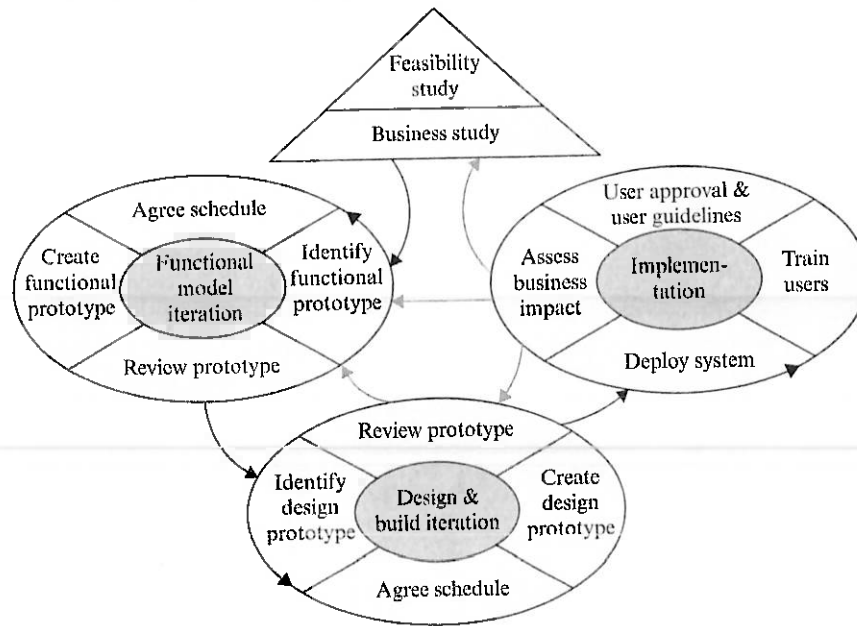


FIGURE 2.13 Process of the Dynamic Systems Development Method

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

2.7.2 Scrum

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

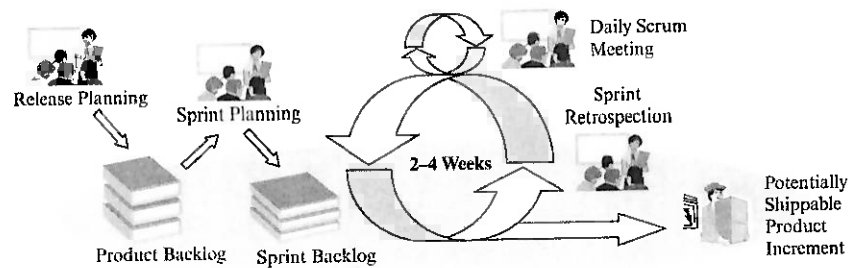


FIGURE 2.14 Scrum development activities

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

2.7.3 Feature Driven Development

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

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

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

2.7.4 Extreme Programming

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

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

2.7.5 Agile or Plan-Driven

Although agile methods are getting increasingly popular in the software industry, that does not mean that they do not have limitations. In [35], Boehm and Turner point out that agile methods and plan-driven approaches "have home grounds where one clearly

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

2.8 OVERVIEW OF PROCESS AND METHODOLOGY OF THE BOOK

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

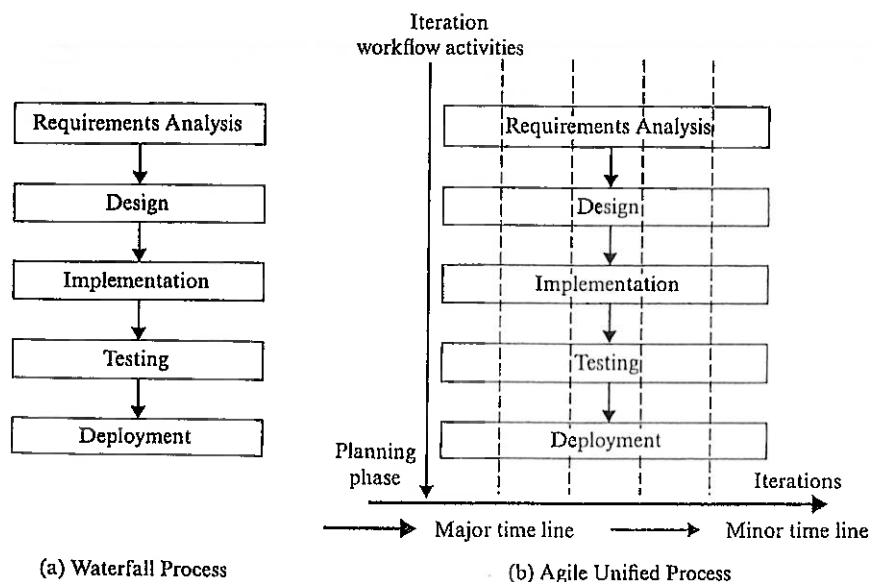
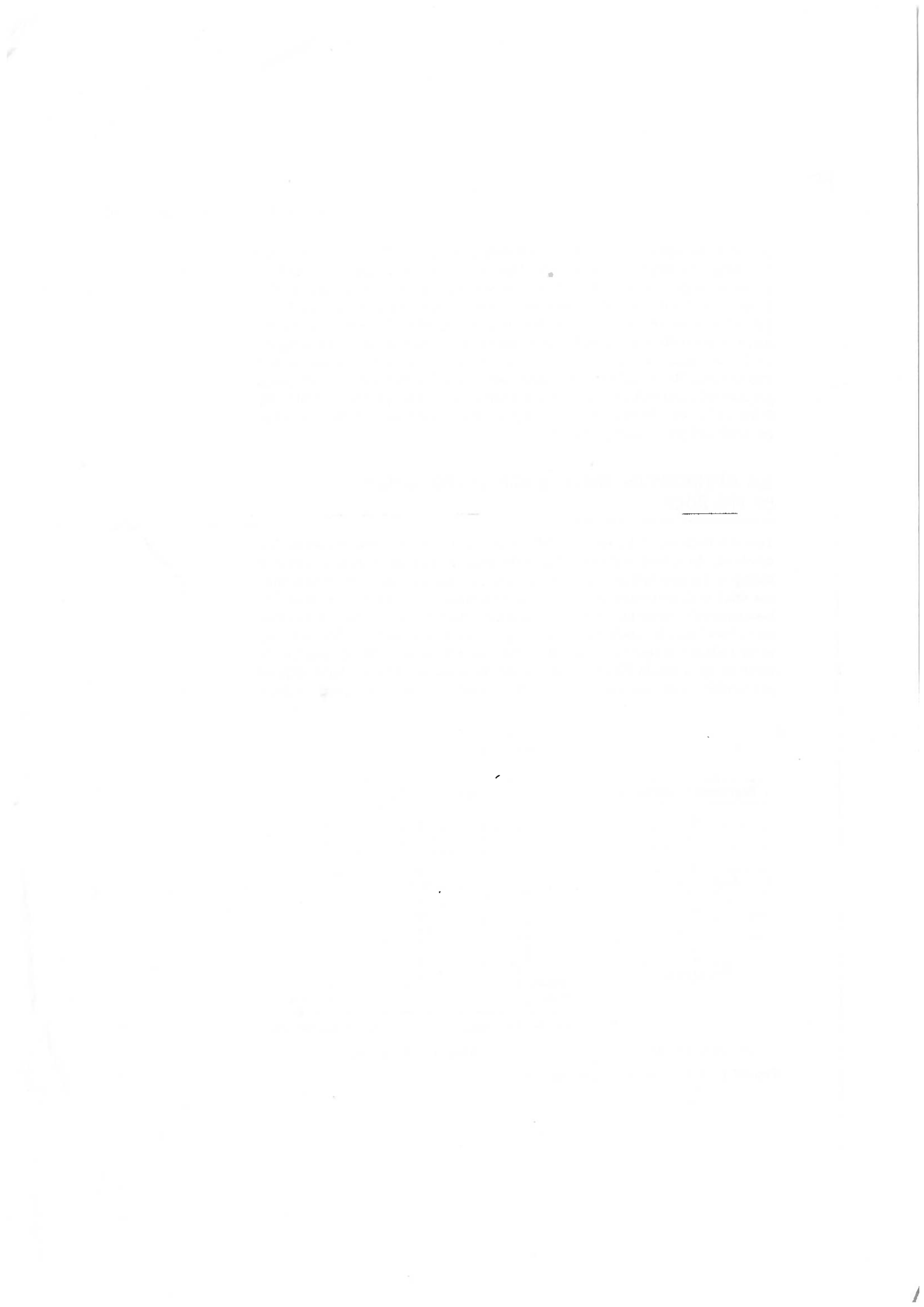
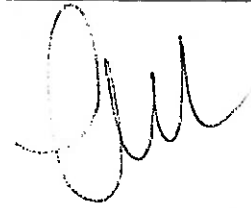


FIGURE 2.15 Waterfall and an agile process



OBJECT-ORIENTED AND CLASSICAL SOFTWARE ENGINEERING

FIFTH EDITION



Stephen R. Schach
Vanderbilt University



Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

PART

1

**INTRODUCTION TO SOFTWARE
ENGINEERING**

The first nine chapters of this book play a dual role. They introduce the reader to the software process, and they also provide an overview of the book. The software process is the way we produce software. It starts with concept exploration and ends when the product is finally decommissioned. During this period, the product goes through a series of phases such as requirements, specification, design, implementation, integration, maintenance, and ultimately, retirement. The software process also includes the tools and techniques we use to develop and maintain software as well as the software professionals involved.

In Chapter 1, "The Scope of Software Engineering," it is pointed out that techniques for software production must be cost effective and promote constructive interaction between the members of the software production team. The importance of objects is stressed throughout the book, starting with this chapter.

"The Software Process" is the title of Chapter 2. Each phase of the process is discussed in detail. Many problems of software engineering are described, but no solutions are put forward in this chapter. Instead, the reader is informed where in the book each problem is tackled. In this way, the chapter serves as a guide to the rest of the book. The chapter concludes with material on software process improvement:

A variety of different software life-cycle models are discussed in detail in Chapter 3, "Software Life-Cycle Models." These include the waterfall model, the rapid prototyping model, the incremental model, extreme programming, the synchronize-and-stabilize model, and the spiral model. To enable the reader to decide on an appropriate life-cycle model for a specific project, the various life-cycle models are compared and contrasted.

Chapter 4 is entitled "Teams." Today's projects are too large to be completed by a single individual within the given time constraints. Instead, a team of software professionals collaborate on the project. The major topic of this chapter is how teams should be organized so that team members work together productively. Various different ways of organizing teams are discussed, including democratic teams, chief programmer teams, synchronize-and-stabilize teams, and extreme programming teams.

Chapter 5 discusses "The Tools of the Trade." A software engineer needs to be able to use a number of different tools, both theoretical and practical. In this chapter, the reader is introduced to a variety of software engineering tools. One such tool is stepwise refinement, a technique for decomposing a large problem into smaller, more tractable problems. Another tool is cost-benefit analysis, a technique for determining whether a software project is financially feasible. Then, computer-aided software engineering (CASE) tools are

described. A CASE tool is a software product that assists software engineers to develop and maintain software. Finally, to manage the software process, it is necessary to measure various quantities to determine whether the project is on track. These measures (metrics) are critical to the success of a project.

The last two topics of Chapter 5, CASE tools and metrics, are treated in detail in Chapters 10 through 16, which describe the specific phases of the software life cycle. There is a discussion of the CASE tools that support each phase, as well as a description of the metrics needed to manage that phase adequately.

An important theme of this book is that testing is not a separate phase to be carried out just before delivering the product to the client or even at the end of each phase of the software life cycle. Instead, testing is performed in parallel with all software production activities. In Chapter 6, "Testing," the concepts underlying testing are discussed. The consideration of testing techniques specific to individual phases of the software life cycle is deferred until Chapters 10 through 16.

Chapter 7 is entitled "From Modules to Objects." A detailed explanation is given of classes and objects, and why the object-oriented paradigm is proving to be more successful than the structured paradigm. The concepts of this chapter then are utilized in the rest of the book, particularly Chapter 12, "Object-Oriented Analysis Phase," and in Chapter 13, "Design Phase," in which object-oriented design is presented.

The ideas of Chapter 7 are extended in Chapter 8, "Reusability, Portability, and Interoperability." It is important to be able to write reusable software that can be ported to a variety of different hardware and run on distributed architectures such as client-server. The first part of the chapter is devoted to reuse; the topics include a variety of reuse case studies as well as reuse strategies such as object-oriented patterns and frameworks. Portability is the second major topic; portability strategies are presented in some depth. The chapter concludes with interoperability topics such as CORBA and COM. A recurring theme of this chapter is the role of objects in achieving reusability, portability, and interoperability.

The last chapter in Part 1 is Chapter 9, "Planning and Estimating." Before starting a software project, it is essential to plan the entire operation in detail. Once the project begins, management must closely monitor progress, noting deviations from the plan and taking corrective action where necessary. Also, it is vital that the client be provided accurate estimates of how long the project will take and how much it will cost. Different estimation techniques are presented, including function points and COCOMO II. A detailed description of a software project management plan is given. The material of this chapter is utilized in Chapters 11 and 12. When the classical paradigm is used, major planning and estimating activities take place at the end of the specification phase, as explained in Chapter 11. When software is developed using the object-oriented paradigm, this planning takes place at the end of the object-oriented analysis phase (Chapter 12).

chapter

1

THE SCOPE OF SOFTWARE ENGINEERING

A well-known story tells of an executive who received a computer-generated bill for \$0.00. After having a good laugh with friends about “idiot computers,” the executive tossed the bill away. A month later a similar bill arrived, this time marked 30 days. Then came the third bill. The fourth bill arrived a month later, accompanied by a message hinting at possible legal action if the bill for \$0.00 was not paid at once.

The fifth bill, marked 120 days, did not hint at anything—the message was rude and forthright, threatening all manner of legal actions if the bill was not immediately paid. Fearful of his organization’s credit rating in the hands of this maniacal machine, the executive called an acquaintance who was a software engineer and related the whole sorry story. Trying not to laugh, the software engineer told the executive to mail a check for \$0.00. This had the desired effect, and a receipt for \$0.00 was received a few days later. The executive carefully filed it away in case at some future date the computer might allege that \$0.00 was still owing.

This well-known story has a less well-known sequel. A few days later the executive was summoned by his bank manager. The banker held up a check and asked, “Is this your check?”

The executive agreed that it was.

“Would you mind telling me why you wrote a check for \$0.00?” asked the banker.

So the whole story was retold. When the executive had finished, the banker turned to him and she quietly asked, “Have you any idea what your check for \$0.00 did to *our* computer system?”

A computer professional can laugh at this story, albeit somewhat nervously. After all, every one of us has designed or implemented a product that, in its original form, would have resulted in the equivalent of sending dunning letters for \$0.00. Up to now, we have always caught this sort of fault during testing. But our laughter has a hollow ring to it, because at the back of our minds is the fear that someday we will not detect the fault before the product is delivered to the customer.

A decidedly less humorous software fault was detected on November 9, 1979. The Strategic Air Command had an alert scramble when the worldwide military command and control system (WWMCCS) computer network reported that the Soviet Union had launched missiles aimed toward the United States [Neumann, 1980]. What actually happened was that a simulated attack was interpreted as the real thing, just as in the movie *WarGames* some 5 years later. Although the U.S. Department of Defense understandably has not given details about the precise mechanism by which test data were taken for actual data, it seems reasonable to ascribe the problem to a software fault. Either the system as a whole was not designed to differentiate between simulations and reality, or the user interface did not include the necessary checks for ensuring that end users

JUST IN CASE YOU WANTED TO KNOW

In the case of the WWMCCS network, disaster was averted at the last minute. However, the consequences of other software faults sometimes have been tragic. For example, between 1985 and 1987, at least two patients died as a consequence of severe overdoses of radiation delivered by the Therac-25 medical linear accelerator [Leveson and Turner, 1993]. The cause was a fault in the control software.

During the 1991 Gulf War, a Scud missile penetrated the Patriot antimissile shield and struck a barracks near Dhahran, Saudi Arabia. In all, 28 Americans were killed and 98 wounded. The software for the Patriot missile contained a cumulative timing fault. The Patriot was designed to operate for only a few hours at

a time, after which the clock was reset. As a result, the fault never had a significant effect and therefore was not detected. In the Gulf War, however, the Patriot missile battery at Dhahran ran continuously for over 100 hours. This caused the accumulated time discrepancy to become large enough to render the system inaccurate.

During the Gulf War, the United States shipped Patriot missiles to Israel for protection against the Scuds. Israeli forces detected the timing problem after only 8 hours and immediately reported it to the manufacturer in the United States. The manufacturer corrected the fault as quickly as it could but, tragically, the new software arrived the day after the direct hit by the Scud [Mellor, 1994].

of the system would be able to distinguish fact from fiction. In other words, a software fault, if indeed the problem was caused by software, could have brought civilization as we know it to an unpleasant and abrupt end. (See the Just in Case You Wanted to Know box above for information on disasters caused by other software faults.)

Whether we are dealing with billing or air defense, much of our software is delivered late, over budget, and with residual faults. Software engineering is an attempt to solve these problems. In other words, software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user's needs. Furthermore, the software must be easy to modify when the user's needs change. To achieve these goals, a software engineer has to acquire a broad range of skills, both technical and managerial. These skills have to be applied not just to programming but to every phase of software production, from requirements to maintenance.

The scope of software engineering is extremely broad. Some aspects of software engineering can be categorized as mathematics or computer science; other aspects fall into the areas of economics, management, or psychology. To display the wide-reaching realm of software engineering, five different aspects now will be examined.

1.1 HISTORICAL ASPECTS

It is a fact that electric power generators fail, but far less frequently than payroll products. Bridges sometimes collapse, but considerably less often than operating systems. In the belief that software design, implementation, and maintenance could be

put on the same footing as traditional engineering disciplines, a NATO study group in 1967 coined the term *software engineering*. The claim that building software is similar to other engineering tasks was endorsed by the 1968 NATO Software Engineering Conference held in Garmisch, Germany [Naur, Randell, and Buxton, 1976]. This endorsement is not too surprising; the very name of the conference reflected the belief that software production should be an engineeringlike activity. A conclusion of the conferees was that software engineering should use the philosophies and paradigms of established engineering disciplines to solve what they termed the *software crisis*; namely, that the quality of software generally was unacceptably low and that deadlines and cost limits were not being met.

Despite many software success stories, a considerable amount of software still is being delivered late, over budget, and with residual faults. That the software crisis still is with us, over 30 years later, tells us two things. First, the software production process, while resembling traditional engineering in many respects, has its own unique properties and problems. Second, the software crisis perhaps should be renamed the *software depression*, in view of its long duration and poor prognosis.

Certainly, bridges collapse less frequently than operating systems. Why then cannot bridge-building techniques be used to build operating systems? What the NATO conferees overlooked is that bridges are as different from operating systems as chalk is from cheese.

A major difference between bridges and operating systems lies in the attitudes of the civil engineering community and the software engineering community to the act of collapsing. When a bridge collapses, as the Tacoma Narrows bridge did in 1940, the bridge almost always has to be redesigned and rebuilt from scratch. The original design was faulty and posed a threat to human safety; certainly, the design requires drastic changes. In addition, the effects of the collapse in almost every instance will have caused so much damage to the bridge fabric that the only reasonable thing is to demolish what is left of the faulty bridge, then completely redesign and rebuild it. Furthermore, other bridges built to the same design have to be carefully inspected and, in the worst case, redesigned and rebuilt.

In contrast, an operating system crash is not considered unusual and rarely triggers an immediate investigation into its design. When a crash occurs, it may be possible simply to reboot the system in the hope that the set of circumstances that caused the crash will not recur. This may be the only remedy if, as often is the case, there is no evidence as to the cause of the crash. The damage caused by the crash usually will be minor: a database partially corrupted, a few files lost. Even when damage to the file system is considerable, backup data often can restore the file system to a state not too far removed from its precrash condition. Perhaps, if software engineers treated an operating system crash as seriously as civil engineers treat a bridge collapse, the overall level of professionalism within software engineering would rise.

Now consider a real-time system, that is, a system able to respond to inputs from the real world as fast as they occur. An example is a computer-controlled intensive care unit. Irrespective of how many medical emergencies occur virtually simultaneously, the system must continue to alert the medical staff to every new emergency without ceasing to monitor those patients whose condition is critical but stable. In general, the failure of a real-time system, whether it controls an intensive care unit, a nuclear

reactor, or the climatic conditions aboard a space station, has significant effects. Most real-time systems, therefore, include some element of fault tolerance to minimize the effects of a failure. That is, the system is designed to attempt an automatic recovery from any failure.

The very concept of fault tolerance highlights a major difference between bridges and operating systems. Bridges are engineered to withstand every reasonably anticipated condition: high winds, flash floods, and so on. An implicit assumption of all too many software builders is that we cannot hope to anticipate all possible conditions that the software must withstand, so we must design our software to try to minimize the damage that an unanticipated condition might cause. In other words, bridges are assumed to be perfectly engineered. In contrast, most operating systems are assumed to be imperfectly engineered; many are designed in such a way that rebooting is a simple operation that the user may perform whenever needed. This difference is a fundamental reason why so much software today cannot be considered to be *engineered*.

It might be suggested that this difference is only temporary. After all, we have been building bridges for thousands of years, and we therefore have considerable experience and expertise in the types of conditions a bridge must withstand. We have only 50 years of experience with operating systems. Surely with more experience, the argument goes, we will understand operating systems as well as we understand bridges and so eventually will be able to construct operating systems that will not fail.

The flaw in this argument is that hardware, and hence the associated operating system, is growing in complexity faster than we can master it. In the 1960s, we had multiprogramming operating systems; in the 1970s, we had to deal with virtual memory; and now, we are attempting to come to terms with multiprocessor and distributed (network) operating systems. Until we can handle the complexity caused by the interconnections of the various components of a software product such as an operating system, we cannot hope to understand it fully; and if we do not understand it, we cannot hope to engineer it.

Part of the reason for the complexity of software is that, as it executes, software goes through discrete states. Changing even one bit causes the software to change state. The total number of such states can be vast, and many of them have not been considered by the development team. If the software enters such an unanticipated state, the result often is software failure. In contrast, bridges are continuous (analog) systems. They are described using continuous mathematics, essentially calculus. However, discrete systems such as operating systems have to be described using discrete mathematics [Parnas, 1990]. Software engineers therefore have to be skilled in discrete mathematics, a primary tool in trying to cope with this complexity.

A second major difference between bridges and operating systems is maintenance. Maintaining a bridge generally is restricted to painting it, repairing minor cracks, resurfacing the road, and so on. A civil engineer, if asked to rotate a bridge through 90° or to move it hundreds of miles, would consider the request outrageous. However, we think nothing of asking a software engineer to convert a batch operating system into a time-sharing one or to port it from one machine to another with totally different architectural characteristics. It is not unusual for 50 percent of the source

code of an operating system to be rewritten over a 5-year period, especially if it is ported to new hardware. But no engineer would consent to replacing half a bridge; safety requirements would dictate that a new bridge be built. The area of maintenance, therefore, is a second fundamental aspect in which software engineering differs from traditional engineering. Further maintenance aspects of software engineering are described in Section 1.3. But first, economic-oriented aspects are presented.

1.2 ECONOMIC ASPECTS

An insight into the relationship between software engineering and computer science can be obtained by comparing and contrasting the relationship between chemical engineering and chemistry. After all, computer science and chemistry are both sciences, and both have a theoretical component and a practical component. In the case of chemistry, the practical component is laboratory work; in the case of computer science, the practical component is programming.

Consider the process of extracting gasoline from coal. During World War II, the Germans used this process to make fuel for their war machine because they largely were cut off from oil supplies. While the antiapartheid oil embargo was in effect, the government of the Republic of South Africa poured billions of dollars into SASOL (an Afrikaans acronym standing for "South African coal into oil"). About half of South Africa's liquid fuel needs were met in this way.

From the viewpoint of a chemist, there are many possible ways to convert coal into gasoline and all are equally important. After all, no one chemical reaction is more important than any other. But from the chemical engineer's viewpoint, at any one time there is exactly one important mechanism for synthesizing gasoline from coal—the reaction that is economically the most attractive. In other words, the chemical engineer evaluates all possible reactions, then rejects all but that one reaction for which the cost per liter is the lowest.

A similar relationship holds between computer science and software engineering. The computer scientist investigates a variety of ways to produce software, some good and some bad. But the software engineer is interested in only those techniques that make sound economic sense.

For instance, a software organization currently using coding technique CT_{old} discovers that new coding technique, CT_{new} , would result in code being produced in only nine-tenths of the time needed by CT_{old} and, hence, at nine-tenths of the cost. Common sense seems to dictate that CT_{new} is the appropriate technique to use. In fact, although common sense certainly dictates that the faster technique is the technique of choice, the economics of software engineering may imply the opposite.

One reason is the cost of introducing new technology into an organization. The fact that coding is 10 percent faster when technique CT_{new} is used may be less important than the costs incurred in introducing CT_{new} into the organization. It may be necessary to complete two or three projects before recouping the cost of training.

Also, while attending courses on CT_{new} , software personnel are unable to do productive work. Even when they return, a steep learning curve may be involved; it may take months of practice with CT_{new} before software professionals become as proficient with CT_{new} as they currently are with CT_{old} . Therefore, initial projects using CT_{new} may take far longer to complete than if the organization had continued to use CT_{old} . All these costs need to be taken into account when deciding whether to change to CT_{new} .

A second reason why the economics of software engineering may dictate that CT_{old} be retained is the maintenance consequence. Coding technique CT_{new} indeed may be 10 percent faster than CT_{old} , and the resulting code may be of comparable quality from the viewpoint of satisfying the client's current needs. But the use of technique CT_{new} may result in code that is difficult to maintain, making the cost of CT_{new} higher over the life of the product. Of course, if the software developer is not responsible for any maintenance, then, from the viewpoint of just that developer, CT_{new} is a most attractive proposition. After all, use of CT_{new} would cost 10 percent less. The client should insist that technique CT_{old} be used and pay the higher initial costs with the expectation that the total lifetime cost of the software will be lower. Unfortunately, often the sole aim of both the client and the software provider is to produce code as quickly as possible. The long-term effects of using a particular technique generally are ignored in the interests of short-term gain. Applying economic principles to software engineering requires the client to choose techniques that reduce long-term costs.

We now consider the importance of maintenance.

1.3 MAINTENANCE ASPECTS

The series of steps that software undergoes, from concept exploration through final retirement, is termed its *life cycle*. During this time, the product goes through a series of phases: requirements, specification, design, implementation, integration, maintenance, and retirement. Life-cycle models are discussed in greater detail in Chapter 3; the topic is introduced at this point so that the concept of maintenance can be defined.

Until the end of the 1970s, most organizations were producing software using as their life-cycle model what now is termed the *waterfall model*. There are many variations of this model, but by and large, the product goes through seven broad phases. These phases probably do not correspond exactly to the phases of any one particular organization, but they are sufficiently close to most practices for the purposes of this book. Similarly, the precise name of each phase varies from organization to organization. The names used here for the various phases have been chosen to be as general as possible in the hope that the reader will feel comfortable with them. For easy reference, the phases are summarized in Figure 1.1, which also indicates the chapters in this book in which they are presented.

- | |
|---|
| <ol style="list-style-type: none"> 1. Requirements phase (Chapter 10) 2. Specification (analysis) phase (Chapters 11 and 12) 3. Design phase (Chapter 13) 4. Implementation phase (Chapters 14 and 15) 5. Integration phase (Chapter 15) 6. Maintenance phase (Chapter 16) 7. Retirement |
|---|

Figure 1.1 The phases of the software life cycle and the chapters in this book in which they are presented.

1. *Requirements phase.* The concept is explored and refined, and the client's requirements are elicited.
2. *Specification (analysis) phase.* The client's requirements are analyzed and presented in the form of the *specification document*, "what the product is supposed to do." This phase sometimes is called the *analysis phase*. At the end of this phase, a plan is drawn up, the *software project management plan*, describing the proposed software development in full detail.
3. *Design phase.* The specifications undergo two consecutive design processes. First comes *architectural design*, in which the product as a whole is broken down into components, called *modules*. Then, each module is designed; this process is termed *detailed design*. The two resulting *design documents* describe "how the product does it."
4. *Implementation phase.* The various components are coded and tested.
5. *Integration phase.* The components of the product are combined and tested as a whole. When the developers are satisfied that the product functions correctly, it is tested by the client (*acceptance testing*). This phase ends when the product is accepted by the client and installed on the client's computer. (We will see in Chapter 15 that the integration phase should be performed in parallel with the implementation phase.)
6. *Maintenance phase.* The product is used to perform the tasks for which it was developed. During this time, it is maintained. Maintenance includes all changes to the product once the client has agreed that it satisfies the specification document (but see the Just in Case You Wanted to Know box on page 10). Maintenance includes *corrective maintenance* (or *software repair*), which consists of the removal of residual faults while leaving the specifications unchanged, as well as *enhancement* (or *software update*), which consists of changes to the specifications and the implementation of those changes. There are, in turn, two types of enhancement. The first is *perfective maintenance*, changes that the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time. The second is *adaptive maintenance*, changes made in response to changes in the environment in which the product operates, such as

new government regulations. Studies have indicated that, on average, maintainers spend approximately 17.5 percent of their time on corrective maintenance, 60.5 percent on perfective maintenance, and 18 percent on adaptive maintenance [Lientz, Swanson, and Tompkins, 1978].

7. *Retirement.* The product is removed from service. This occurs when the functionality provided by the product no longer is of any use to the client organization.

JUST IN CASE YOU WANTED TO KNOW

In the 1970s, software production was viewed as consisting of two distinct activities performed sequentially: *développement* followed by *maintenance*. Starting from scratch, the software product was developed then installed on the client's computer. Any change to the software after installation, whether to fix a residual fault or extend the functionality, constituted classical maintenance [IEEE 610.12, 1990]. Hence, the way that software was developed classically can be described as the *development-then-maintenance* model.

This is a temporal definition; that is, an activity is classified as development or maintenance depending on when it is performed. Suppose that a fault in the software is detected and corrected a day after the software has been installed. This clearly constitutes maintenance. But if the identical fault is detected and corrected the day before the software is installed, in terms of the classical definition, this constitutes development.

There are two reasons why this model is unrealistic today. First, nowadays it is certainly not unusual for construction of a product to take a year or more. During this time, the client's requirements may well change. For example, the client might insist that the product now be implemented on a faster microprocessor that has become available. Alternatively, the client organization may have expanded into Canada while development was under way, and the product now has to be modified so it also can handle sales in Canada. To see how this sort of change in requirements affects the software life cycle, suppose that the client's requirements change while the design is being developed. The software engineering team has to suspend development and modify the specification document to reflect the changed requirements. Furthermore, it then may be necessary to modify the design as well, if the

changes to the specifications necessitate corresponding changes to those portions of the design already completed. Only when these changes have been made can development proceed. In other words, developers have to perform "maintenance" long before the product is installed.

A second problem with the classical development-then-maintenance model arose as a result of the way in which we now construct software. In classical software engineering, a characteristic of development was that the development team built the target product from scratch. In contrast, as a consequence of the high cost of software production today, developers try to reuse parts of existing software wherever possible in the software to be constructed (reuse is discussed in detail in Chapter 8). Therefore, the development-then-maintenance model is inappropriate whenever there is reuse.

A more realistic way of looking at maintenance is to view maintenance as the process that occurs when "software undergoes modifications to code and associated documentation due to a problem or the need for improvement or adaptation" [ISO/IEC 12207, 1995]. By this definition, maintenance occurs whenever a fault is fixed or the requirements change, irrespective of whether this takes place before or after installation of the product.

However, until such time as the majority of software engineers realize that the development-then-maintenance model is outdated, there is little point in trying to change the usage of the word *maintenance*. In this book, I accordingly still refer to maintenance as the activity carried out after development is complete and the product installed. Nevertheless, I hope that the true nature of maintenance soon will be more widely recognized.

Returning to the topic of maintenance, it sometimes is said that only bad software products undergo maintenance. In fact, the opposite is true; bad products are thrown away, whereas good products are repaired and enhanced, for 10, 15, or even 20 years. Furthermore, a software product is a model of the real world, and the real world is perpetually changing. As a consequence, software has to be maintained constantly for it to remain an accurate reflection of the real world.

For instance, if the sales tax rate changes from 6 percent to 7 percent, almost every software product that deals with buying or selling has to be changed. Suppose the product contains the C++ statement

```
const float salesTax = 6.0;
```

or the equivalent Java statement

```
public static final float salesTax = (float) 6.0;
```

declaring that `salesTax` is a floating-point constant initialized to the value 6.0. In this case, maintenance is relatively simple. With the aid of a text editor the value 6.0 is replaced by 7.0 and the code is recompiled and relinked. However, if instead of using the name `salesTax`, the actual value 6.0 has been used in the product wherever the value of the sales tax is invoked, then such a product will be extremely difficult to maintain. For example, there may be occurrences of the value 6.0 in the source code that should be changed to 7.0 but are overlooked or instances of 6.0 that do not refer to sales tax but incorrectly are changed to 7.0. Finding these faults almost always is difficult and time consuming. In fact, with some software, it might be less expensive in the long run to throw away the product and recode it rather than try to determine which of the many constants need to be changed and how to make the modifications.

The real-time real world also is constantly changing. The missiles with which a jet fighter is armed may be replaced by a new model, requiring a change to the weapons control component of the associated avionics system. A six-cylinder engine is to be offered as an option in a popular four-cylinder automobile; this implies changing the on-board computer that controls the fuel injection system, timing, and so on.

Healthy organizations change; only dying organizations are static. This means that maintenance in the form of enhancement is a positive part of an organization's activities, reflecting that the organization is on the move.

But just how much time is devoted to maintenance? The pie chart in Figure 1.2 was obtained by averaging data from various sources, including [Elshoff, 1976; Daly, 1977; Zelkowitz, Shaw, and Gannon, 1979; and Boehm, 1981]. Figure 1.2 shows the approximate percentage of time (= money) spent on each phase of the software life cycle. About 15 years later, the proportion of time spent on the various development phases had hardly changed. This is shown in Figure 1.3, which compares the data in Figure 1.2 with more recent data on 132 Hewlett-Packard projects [Grady, 1994]. The data from Figure 1.2 have been grouped to make them comparable to the newer data.¹

¹ Figure 1.3 reflects only the development phases. The proportion of development time devoted to the requirements and specification phases in Figure 1.2 is $(2 + 5)/33$, or 21%, as shown in Figure 1.3.

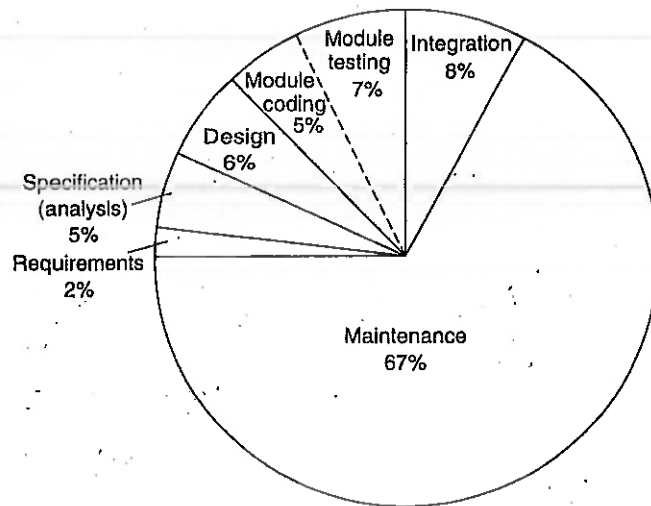


Figure 1.2 Approximate relative costs of the phases of the software life cycle.

	Various Projects between 1976 and 1981	132 More Recent Hewlett-Packard Projects
Requirements and specification (analysis) phases	21%	18%
Design phase	18	19
Implementation phase	36	34
Integration phase	24	29

Figure 1.3 Comparison of approximate average percentages of time spent on the development phases for various projects between 1976 and 1981 and for 132 more recent Hewlett-Packard projects.

As can be seen in Figure 1.2, about two-thirds of total software costs were devoted to maintenance. Newer data confirm the continuing emphasis on maintenance. For example, in 1992 between 60 and 80 percent of research and development personnel at Hewlett-Packard were involved in maintenance, and maintenance constituted between 40 and 60 percent of the total cost of software [Coleman, Ash, Lowther, and Oman, 1994]. However, many organizations devote as much as 80 percent of their time and effort to maintenance [Yourdon, 1996]. Therefore, maintenance is an extremely time consuming, expensive phase of the software life cycle.

Consider again the software organization currently using coding technique CT_{old} that learns that CT_{new} will reduce coding time by 10 percent. Even if CT_{new} has no

adverse effect on maintenance, an astute software manager will think twice before changing coding practices. The entire staff will have to be retrained, new software development tools purchased, and perhaps additional staff members hired who are experienced in the new technique. All this expense and disruption has to be endured for a possible 0.5 percent decrease in software costs because, as shown in Figure 1.2, module coding constitutes on average only 5 percent of total software costs.

Now suppose a new technique that reduces maintenance by 10 percent is developed. This probably should be introduced at once because, on average, it will reduce overall costs by 6.7 percent. The overhead involved in changing to this technique is a small price to pay for such large overall savings.

Because maintenance is so important, a major aspect of software engineering consists of those techniques, tools, and practices that lead to a reduction in maintenance costs.

1.4 SPECIFICATION AND DESIGN ASPECTS

Software professionals are human and therefore sometimes make errors while developing a product. As a result, there will be a fault in the software. If the error is made during the requirements phase, then the resulting fault probably also will appear in the specifications, the design, and the code. Clearly, the earlier we correct a fault, the better.

The relative costs of fixing a fault at various phases in the software life cycle are shown in Figure 1.4 [Boehm, 1981]. The figure reflects data from IBM [Fagan, 1974], GTE [Daly, 1977], the Safeguard project [Stephenson, 1976], and some smaller TRW projects [Boehm, 1980]. The solid line in Figure 1.4 is the best fit for the data relating to the larger projects, and the dashed line is the best fit for the smaller projects. For each of the phases of the software life cycle, the corresponding relative cost to detect and correct a fault is depicted in Figure 1.5. Each step on the solid line in Figure 1.5 is constructed by taking the corresponding point on the solid straight line of Figure 1.4 and plotting the data on a linear scale.

Suppose it costs \$40 to detect and correct a specific fault during the design phase. From the solid line in Figure 1.5 (projects between 1974 and 1980), that same fault would cost only about \$30 to fix during the specification phase. But, during the maintenance phase, that fault would cost around \$2000 to detect and correct. Newer data show that now it is even more important to detect faults early. The dashed line in Figure 1.5 shows the cost of detecting and correcting a fault during the development of system software for the IBM AS/400 [Kan et al., 1994]. On average, the same fault would have cost \$3680 to fix during the maintenance phase of the AS/400 software.

The reason that the cost of correcting a fault increases so steeply is related to what has to be done to correct a fault. Early in the development life cycle, the product essentially exists only on paper, and correcting a fault may simply mean using an eraser and pencil. The other extreme is a product already delivered to a client. At the very least, correcting a fault means editing the code, recompiling and relinking it, and then carefully testing that the problem is solved. Next, it is critical to check that making the change has not created a new problem elsewhere in the product. All

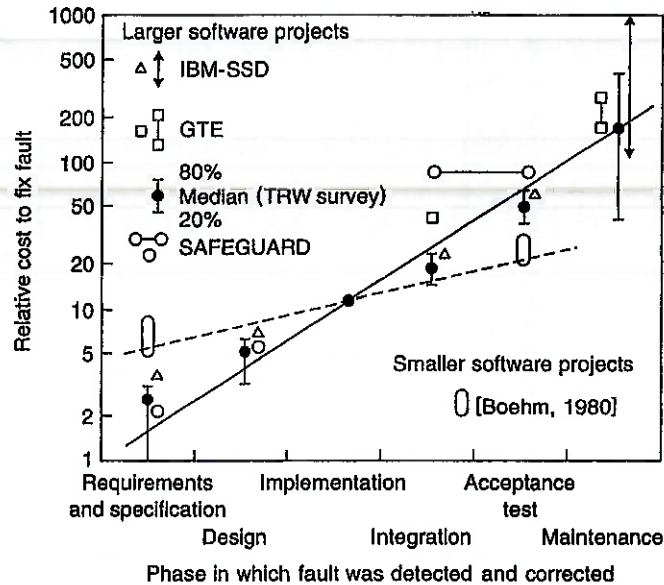


Figure 1.4 Relative cost of fixing a fault at each phase of the software life cycle. The solid line is the best fit for the data relating to the larger software projects, and the dashed line is the best fit for the smaller software projects. (Barry Boehm, *Software Engineering Economics*, ©1981, p. 40. Adapted by permission of Prentice Hall, Inc., Englewood Cliffs, NJ.)

the relevant documentation, including manuals, then needs to be updated. Finally, the corrected product must be delivered and installed. The moral of the story is this: We must find faults early or else it will cost us money. We therefore should employ techniques for detecting faults during the requirements and specification (analysis) phases.

There is a further need for such techniques. Studies have shown [Boehm, 1979] that between 60 and 70 percent of all faults detected in large-scale projects are specification or design faults. Newer results bear out this preponderance of specification and design faults. An inspection is a careful examination of a document by a team (Section 6.2.3). During 203 inspections of Jet Propulsion Laboratory software for the NASA unmanned interplanetary space program, on average, about 1.9 faults were detected per page of a specification document, 0.9 faults per page of a design, but only 0.3 faults per page of code [Kelly, Sherif, and Hops, 1992].

Therefore, it is important that we improve our specification and design techniques, not only so that faults can be found as early as possible but also because specification and design faults constitute such a large proportion of all faults. Just as the example in the previous section showed that reducing maintenance costs by 10 percent will reduce overall costs by nearly 7 percent, reducing specification and design faults by 10 percent will reduce the overall number of faults by 6 to 7 percent.

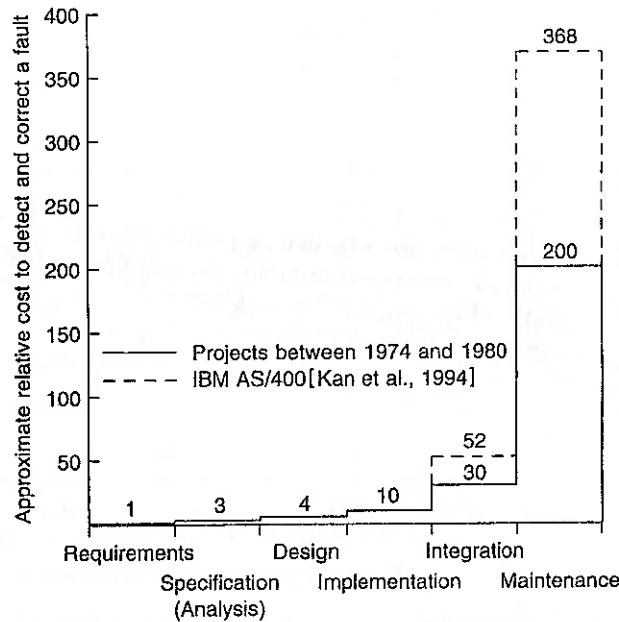


Figure 1.5 The solid line depicts the points on the solid line of Figure 1.4 plotted on a linear scale. The dashed line depicts newer data.

Newer data on a number of projects are reported in [Bhandari et al., 1994]. For example, a compiler was undergoing extensive changes. At the end of the design phase, the faults detected during the project were categorized. Only 13 percent of the faults were carry-overs from previous versions of the compiler. Of the remaining faults, 16 percent were introduced during the specification phase, and 71 percent were introduced during the design phase. That so many faults are introduced early in the software life cycle highlights another important aspect of software engineering; namely, techniques that yield better specifications and designs.

Most software is produced by a team of software engineers rather than by a single individual responsible for every phase of the development and maintenance life cycle. We now consider the implications of this.

1.5 TEAM PROGRAMMING ASPECTS

The performance-price factor of a computer may be defined as follows:

$$\text{performance-price factor} = \text{time to perform 1 million additions} \\ \times \text{cost of CPU and main memory}$$

This quantity has decreased by an order of magnitude with each succeeding generation of computers. This decrease has been a consequence of discoveries in electronics, particularly the transistor, and very large-scale integration (VLSI).

The result of these discoveries has been that organizations easily can afford hardware that can run large products, that is, products too large to be written by one person within the allowed time constraints. For example, if a product has to be delivered within 18 months but would take a single programmer 15 years to complete, then the product must be developed by a team. However, team programming leads to interface problems among code components and communication problems among team members.

For example, Joe and Freda code modules *p* and *q*, respectively, where module *p* calls module *q*. When Joe codes *p*, he writes a call to *q* with five arguments in the argument list. Freda codes *q* with five arguments but in a different order from those of Joe. Unless function prototypes are used, this will not be detected by an ANSI C compiler. A few software tools, such as the Java interpreter and loader, *lint* for C (Section 8.7.4), or an Ada linker, detect such a type violation and only if the interchanged arguments are of different types; if they are of the same type, then the problem may not be detected for a long period of time. It may be debated that this is a design problem, and if the modules had been more carefully designed, this problem would not have happened. That may be true, but in practice a design often is changed after coding commences but notification of a change may not be distributed to all members of the development team. Thus, when a design that affects two or more programmers has been changed, poor communication can lead to the interface problems Joe and Freda experienced. This sort of problem is less likely to occur when only one individual is responsible for every aspect of the product, as was the case before powerful computers that can run huge products became affordable.

But interfacing problems are merely the tip of the iceberg when it comes to problems that can arise when software is developed by teams. Unless the team is properly organized, an inordinate amount of time can be wasted in conferences between team members. Suppose that a product takes a single programmer 1 year to complete. If the same task is assigned to a team of three programmers, the time for completing the task frequently is closer to 1 year than the expected 4 months, and the quality of the resulting code may well be lower than if the entire task had been assigned to one individual. Because a considerable proportion of today's software is developed and maintained by teams, the scope of software engineering must include techniques for ensuring that teams are properly organized and managed.

As has been shown in the preceding sections, the scope of software engineering is extremely broad. It includes every phase of the software life cycle, from requirements to retirement. It also includes human aspects, such as team organization; economic aspects; and legal aspects, such as copyright law. All these aspects implicitly are incorporated in the definition of software engineering given at the beginning of this chapter; namely, that software engineering is a discipline whose aim is the production of fault-free software delivered on time, within budget, and satisfying the user's needs.

1.6 THE OBJECT-ORIENTED PARADIGM

Before 1975, most software organizations used no specific techniques; each individual worked his or her own way. Major breakthroughs were made between approximately 1975 and 1985, with the development of the so-called structured paradigm. The techniques constituting the structured paradigm include structured systems analysis (Section 11.3), data flow analysis (Section 7.1), structured programming, and structured testing (Section 14.8.2). These techniques seemed extremely promising when first used. However, as time passed, they proved to be somewhat less successful in two respects. First, the techniques sometimes were unable to cope with the increasing size of software products. That is, the structured techniques were adequate when dealing with products of (say) 5,000 or even 50,000 lines of code. Today, however, products containing 500,000 lines of code are not considered large; even products of 5 million or more lines of code are not that unusual. However, the structured techniques frequently could not scale up sufficiently to handle today's larger products.

The maintenance phase is the second area in which the structured paradigm did not live up to earlier expectations. A major driving force behind the development of the structured paradigm 30 years ago was that, on average, two-thirds of the software budget was being devoted to maintenance (see Figure 1.2). Unfortunately, the structured paradigm has not solved this problem; as pointed out in Section 1.3, many organizations still spend up to 80 percent of their time and effort on maintenance [Yourdon, 1996].

The reason for the limited success of the structured paradigm is that the structured techniques are either action oriented or data oriented but not both. The basic components of a software product are the actions² of the product and the data on which those actions operate. For example, determine average height is an action that operates on a collection of heights (data) and returns the average of those heights (data). Some structured techniques, such as data flow analysis (Section 13.3), are action oriented. That is, such techniques concentrate on the actions of the product; the data are of secondary importance. Conversely, techniques such as Jackson system development (Section 13.5) are data oriented. The emphasis here is on the data; the actions that operate on the data are less significant.

In contrast, the object-oriented paradigm considers both data and actions to be equally important. A simplistic way of looking at an object is as a unified software component that incorporates both the data and the actions that operate on the data. This definition is incomplete and will be fleshed out later in the book, once *inheritance* has been defined (Section 7.7). Nevertheless, the definition captures much of the essence of an object.

A bank account is one example of an object (see Figure 1.6). The data component of the object is the account balance. The actions that can be performed on that account balance include deposit money, withdraw money, and determine balance. From the viewpoint of the structured paradigm, a product that deals with banking

²The word *action* is used in this book rather than *process* to avoid confusion with the term *software process*.

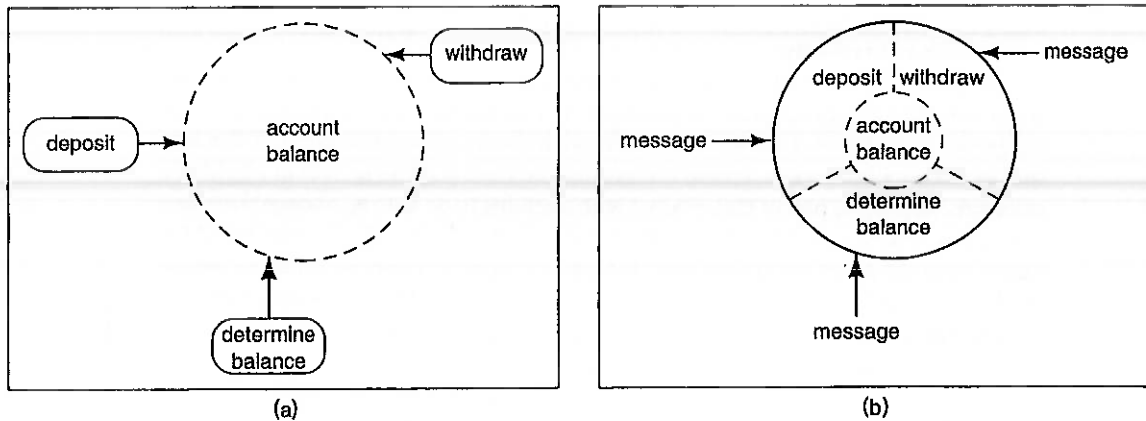


Figure 1.6 Comparison of implementations of bank account using (a) structured paradigm and (b) object-oriented paradigm. The solid black line surrounding the object denotes that details as to how account balance is implemented are not known outside the object.

would have to incorporate a data element, the account balance, and three actions, deposit, withdraw, and determine balance. From the object-oriented viewpoint, a bank account is an object. This object combines a data element together with the three actions performed on that data element in a single unit.

Up to now, there seems to be little difference between the two approaches. However, a key point is the way in which an object is implemented. Specifically, details as to how the data element of an object is stored are not known from outside the object. This is an instance of “information hiding” and is discussed in more detail in Section 7.6. In the case of the bank account object shown in Figure 1.6(b), the rest of the software product is aware that there is such a thing as a balance within a bank account object, but it has no idea as to the format of account balance. That is, there is no knowledge outside the object as to whether the account balance is implemented as an integer or a floating-point number or whether it is a field (component) of some larger structure. This information barrier surrounding the object is denoted by the solid black line in Figure 1.6(b), which depicts an implementation using the object-oriented paradigm. In contrast, a dashed line surrounds account balance in Figure 1.6(a), because all the details of account balance are known to the modules in the implementation using the structured paradigm, and the value of account balance therefore can be changed by any of them.

Returning to Figure 1.6(b), the object-oriented implementation, if a customer deposits \$10 in an account, then a *message* is sent to the deposit action (*method*) of the relevant object telling it to increment the account balance data element (*attribute*) by \$10. The deposit method is within the bank account object and knows how the account balance is implemented; this is denoted by the dashed line inside the object. But no entity external to the object needs to have this knowledge. That the three methods in

Figure 1.6(b) shield account balance from the rest of the product symbolizes this localization of knowledge.

At first sight, the fact that implementation details are local to an object may not seem to be terribly useful. The payoff comes during maintenance. First, suppose that the banking product has been constructed using the structured paradigm. If the way that an account balance is represented is changed from (say) an integer to a field of a structure, then every part of that product with anything to do with an account balance has to be changed and these changes have to be made consistently. In contrast, if the object-oriented paradigm is used, then the only changes that need be made are within the bank account object itself. No other part of the product has knowledge of how an account balance is implemented, so no other part can have access to an account balance. Consequently, no other part of the banking product needs to be changed. Thus, the object-oriented paradigm makes maintenance quicker and easier, and the chance of introducing a regression fault (that is, a fault inadvertently introduced into one part of a product as a consequence of making an apparently unrelated change to another part of the product) is greatly reduced.

In addition to maintenance benefits, the object-oriented paradigm makes development easier. In many instances, an object has a physical counterpart. For example, the object bank account in the bank product corresponds to an actual bank account in the bank for which this product is being written. As will be shown in Chapter 12, modeling plays a major role in the object-oriented paradigm. The close correspondence between the objects in a product and their counterparts in the real world promotes better software development.

There is yet another way of looking at the benefits of the object-oriented paradigm. Well-designed objects are independent units. As has been explained, an object consists of both data and the actions performed on the data. If all the actions performed on the data of an object are included in that object, then the object can be considered a conceptually independent entity. Everything in the product that relates to the portion of the real world modeled by that object can be found in the object itself. This conceptual independence sometimes is termed *encapsulation* (Section 7.4). But there is an additional form of independence, physical independence. In a well-designed object, information hiding ensures that implementation details are hidden from everything outside that object. The only allowable form of communication is the sending of a message to the object to carry out a specific action. The way that the action is carried out is entirely the responsibility of the object itself. For this reason, object-oriented design sometimes is referred to as *responsibility-driven design* [Wirfs-Brock, Wilkerson, and Wiener, 1990] or *design by contract* [Meyer 1992a]. (For another view of responsibility-driven design, see the Just in Case You Wanted to Know box on page 20.)

A product built using the structured paradigm essentially is a single unit. This is one reason why the structured paradigm has been less successful when applied to larger products. In contrast, when the object-oriented paradigm is used correctly, the resulting product consists of a number of smaller, largely independent units. The object-oriented paradigm reduces the level of complexity of a software product and hence simplifies both development and maintenance.

JUST IN CASE YOU WANTED TO KNOW

Suppose that you live in New Orleans, and you want to have a floral arrangement delivered to your aunt in Iowa City on her birthday [Budd, 1991]. One way would be to try to obtain a list of all the florists in Iowa City, then determine which one is located closest to your aunt's home. An easier way is to call 1-800-FLOWERS and leave the entire responsibility for delivering the floral arrangement to that organization. You need not know

the identity of the Iowa City florist who will deliver the flowers.

In exactly the same way, when a message is sent to an object, not only is it entirely irrelevant how the request is carried out, but the unit that sends the message is not even allowed to know the internal structure of the object. The object itself is entirely responsible for every detail of carrying out the message.

Another positive feature of the object-oriented paradigm is that it promotes reuse: Because objects are independent entities, they can be utilized in future products. This reuse of objects reduces the time and cost of both development and maintenance, as explained in Chapter 8.

When the object-oriented paradigm is utilized, the software life cycle (Figure 1.1) has to be modified somewhat. Figure 1.7 shows the software life cycles of both structured and object-oriented paradigms. To appreciate the difference, first consider the design phase of the structured paradigm. As stated in Section 1.3, this phase is divided into two subphases: architectural design followed by detailed design. In the architectural design subphase, the product is decomposed into components, called *modules*. Then, during the detailed design subphase, the data structures and algorithms of each module are designed in turn. Finally, during the implementation phase, these modules are implemented.

If the object-oriented paradigm is used instead, one of the steps during the object-oriented analysis phase is to determine the objects. Because an object is a kind of module, architectural design therefore is performed during the object-oriented analysis

Structured Paradigm	Object-Oriented Paradigm
1. Requirements phase	1. Requirements phase
2. Specification (analysis) phase	2'. Object-oriented analysis phase
3. Design phase	3'. Object-oriented design phase
4. Implementation phase	4'. Object-oriented programming phase
5. Integration phase	5. Integration phase
6. Maintenance phase	6. Maintenance phase
7. Retirement	7. Retirement

Figure 1.7 Comparison of life cycles of the structured paradigm and the object-oriented paradigm.

Structured Paradigm	Object-Oriented Paradigm
2. Specification (analysis) phase <ul style="list-style-type: none"> • Determine what the product is to do 	2'. Object-oriented analysis phase <ul style="list-style-type: none"> • Determine what the product is to do • Extract the objects
3. Design phase <ul style="list-style-type: none"> • Architectural design (extract the modules) • Detailed design 	3'. Object-oriented design phase <ul style="list-style-type: none"> • Detailed design
4. Implementation phase <ul style="list-style-type: none"> • Implement in appropriate programming language 	4'. Object-oriented programming phase <ul style="list-style-type: none"> • Implement in appropriate object-oriented programming language

Figure 1.8 Differences between the structured paradigm and the object-oriented paradigm.

phase. Thus, object-oriented analysis goes further than the corresponding specification (analysis) phase of the structured paradigm. This is shown in Figure 1.8.

This difference between the two paradigms has major consequences. When the structured paradigm is used, there almost always is a sharp transition between the analysis (specification) phase and the design phase. After all, the aim of the specification phase is to determine *what* the product is to do, whereas the purpose of the design phase is to decide *how* to do it. In contrast, when object-oriented analysis is used, objects enter the life cycle from the very beginning. The objects are extracted in the analysis phase, designed in the design phase, and coded in the implementation phase. Thus, the object-oriented paradigm is an integrated approach; the transition from phase to phase is far smoother than with the structured paradigm, thereby reducing the number of faults during development.

As already mentioned, it is inadequate to define an object merely as a software component that encapsulates both data and actions and implements the principle of information hiding. A more complete definition is given in Chapter 7, where objects are examined in depth. But first, the terminology used in this book must be considered in greater detail.

1.7 TERMINOLOGY

A word used on almost every page of this book is *software*. Software consists of not just code in machine-readable form but also all the documentation that is an intrinsic component of every project. Software includes the specification document, the design document, legal and accounting documents of all kinds, the software project management plan, and other management documents as well as all types of manuals.

Since the 1970s, the difference between a *program* and a *system* has become blurred. In the “good old days,” the distinction was clear. A program was an autonomous piece of code, generally in the form of a deck of punched cards, that could be executed. A system was a related collection of programs. Thus, a system might consist of programs P, Q, R, and S. Magnetic tape T₁ was mounted, then program P was run. It caused a deck of data cards to be read in and produced as output tapes T₂ and T₃. Tape T₂ then was rewound, and program Q was run, producing tape T₄ as output. Program R now merged tapes T₃ and T₄ into tape T₅; T₅ served as input for program S, which printed a series of reports.

Compare that situation with a product, running on a machine with a front-end communications processor and a back-end database manager, that performs real-time control of a steel mill. The single piece of software controlling the steel mill does far more than the old-fashioned system, but in terms of the classic definitions of program and system, this software undoubtedly is a program. To add to the confusion, the term *system* now also is used to denote the hardware–software combination. For example, the flight control system in an aircraft consists of both the in-flight computers and the software running on them. Depending on who is using the term, the flight control system also may include the controls, such as the joystick, that send commands to the computer and the parts of the aircraft, such as the wing flaps, controlled by the computer.

To minimize confusion, this book uses the term *product* to denote a nontrivial piece of software. There are two reasons for this convention. The first is simply to obviate the program versus system confusion by using a third term. The second reason is more important. This book deals with the *process* of software production, and the end result of a process is termed a *product*. *Software production* consists of two activities: *software development* followed by *maintenance*. Finally, the term *system* is used in its modern sense, that is, the combined hardware and software, or as part of universally accepted phrases, such as *operating system* and *management information system*.

Two words widely used within the context of software engineering are *methodology* and *paradigm*. Both are used in the same sense, a collection of techniques for carrying out the complete life cycle. This usage offends language purists; after all, *methodology* means the science of methods and a *paradigm* is a model or a pattern. Notwithstanding the best efforts of the author and others to encourage software engineers to use the words correctly, the practice is so widespread that, in the interests of clarity, both words are used in this book in the sense of a *collection of techniques*. Erudite readers offended by this corruption of the English language are warmly invited to take up the cudgels of linguistic accuracy on the author’s behalf; he is tired of tilting at windmills.

One term that is avoided as far as possible is *bug* (the history of this word is in the Just in Case You Wanted to Know box on page 23). The term *bug* nowadays is simply a euphemism for *error*. Although there generally is no real harm in using euphemisms, the word *bug* has overtones that are not conducive to good software production. Specifically, instead of saying, “I made an error,” a programmer will say, “A bug crept into the code” (not *my* code but *the* code), thereby transferring responsibility for the error from the programmer to the bug. No one blames a programmer for coming

JUST IN CASE YOU WANTED TO KNOW

The first use of the word *bug* to denote a fault is attributed to the late Rear Admiral Grace Murray Hopper, one of the designers of COBOL. On September 9, 1945, a moth flew into the Mark II computer that Hopper and her colleagues used at Harvard and lodged between the contact plates of a relay. Thus, there was actually a bug in the system. Hopper taped the bug to the log book and wrote, "First actual case of bug being found." The log book, with moth still attached, is in the Naval Museum at the Naval Surface Weapons Center, in Dahlgren, Virginia.

Although this may have been the first use of *bug* in a computer context, the word was used in engineering

slang in the 19th century [Shapiro, 1994]. For example, Thomas Alva Edison wrote on November 18, 1878, "This thing gives out and then that—'Bugs'—as such little faults and difficulties are called ..." [Josephson, 1992]. One of the definitions of *bug* in the 1934 edition of *Webster's New English Dictionary* is "A defect in apparatus or its operation." It is clear from Hopper's remark that she, too, was familiar with the use of the word in that context; otherwise, she would have explained what she meant.

down with a case of influenza, because the flu is caused by the flu bug. Referring to an error as a *bug* is a way of casting off responsibility. In contrast, the programmer who says, "I made an error," is a computer professional who takes responsibility for his or her actions.

There is considerable confusion regarding object-oriented terminology. For example, in addition to the term *attribute* for a data component of an object, the term *state variable* sometimes is used in the object-oriented literature. In Java, the term is *instance variable*; in C++, the term *field* is used. With regard to the actions of an object, the term *method* usually is used; in C++, however, the term is *member function*. In C++, a *member* of an object refers to either an attribute ("field") or a method. In Java, the term *field* is used to denote either an attribute ("instance variable") or a method. To avoid confusion, wherever possible, the generic terms *attribute* and *method* are used in this book.

Fortunately, some terminology is widely accepted. For example, when a method within an object is invoked, this almost universally is termed *sending a message* to the object.

In this section we defined the various terms used in this book. One of those terms, *process*, is the subject of the next chapter.

CHAPTER REVIEW

Software engineering is defined (Section 1.1) as a discipline whose aim is the production of fault-free software that satisfies the user's needs and is delivered on time and within budget. To achieve this goal, appropriate techniques have to be used in

all phases of software production, including specification (analysis) and design (Section 1.4) and maintenance (Section 1.3). Software engineering addresses all phases of the software life cycle and incorporates aspects of many different areas of human knowledge, including economics (Section 1.2) and the social sciences (Section 1.5). In Section 1.6, objects are introduced, and a brief comparison between the structured and object-oriented paradigms is made. In the final section (Section 1.7), the terminology used in this book is explained.

FOR FURTHER READING

A classic source of information on the scope of software engineering is [Boehm, 1976]. [DeMarco and Lister, 1989] is a report on the extent to which software engineering techniques actually are used. For an analysis of the extent to which software engineering can be considered to be a true engineering discipline, see [Wasserman, 1996] and [Ebert, Matsubara, Pezzé, and Bertelsen, 1997]. The future of software engineering is discussed in [Lewis 1996a, 1996b; Leveson, 1997; Brereton et al., 1999; Kroeker et al., 1999; and Finkelstein, 2000]. Critical factors in software development are discussed in the May/June 1999 issue of *IEEE Software*, especially [Reel, 1999].

The current practice of software engineering is described in [Yourdon, 1996]. For a view on the importance of maintenance in software engineering and how to plan for it, see [Parnas, 1994]. The unreliability of software and the resulting risks (especially in safety-critical systems) are discussed in [Littlewood and Strigini, 1992; Mellor, 1994; and Neumann, 1995]. Modern views of the software crisis appear in [Gibbs, 1994] and [Glass, 1998]. [Zvegintzov, 1998] explains how little accurate data on software engineering practice actually is available.

The fact that mathematics underpins software engineering is stressed in [Parnas, 1990]. The importance of economics in software engineering is discussed in [Boehm, 1981] and [Baetjer, 1996].

Two classic books on the social sciences and software engineering are [Weinberg, 1971] and [Shneiderman, 1980]. Neither book requires prior knowledge of psychology or the behavioral sciences in general. A newer book on the topic is [DeMarco and Lister, 1987].

Brooks's timeless work, *The Mythical Man-Month* [Brooks, 1975], is a highly recommended introduction to the realities of software engineering. The book includes sections on all the topics mentioned in this chapter.

Excellent introductions to the object-oriented paradigm are [Budd, 1991] and [Meyer, 1997]. A balanced perspective of the paradigm is given in [Radin, 1996]. [Khan, Al-A'ali, and Girgis, 1995] explains the differences between the classical and object-oriented paradigms. Three successful projects carried using the object-oriented paradigm are described in [Capper, Colgate, Hunter, and James, 1994]. A survey of the attitudes of 150 experienced software developers toward the object-oriented paradigm is reported in [Johnson, 2000]. Lessons learned from developing large-scale object-oriented products are presented in [Maring, 1996] and [Fichman and Kemerer, 1997].

[Scholtz et al., 1993] is a report on a workshop held in April 1993 on the state of the art and the practice of object-oriented programming. A variety of short articles on recent trends in the object-oriented paradigm can be found in [El-Rewini et al., 1995]. Important articles on the object-oriented paradigm are found in the October 1992 issue of *IEEE Computer*, the January 1993 issue of *IEEE Software*, and the January 1993 and November 1993 issues of the *Journal of Systems and Software*. Potential pitfalls of the object-oriented paradigm are described in [Webster, 1995].

PROBLEMS

- 1.1 You are in charge of developing a raw materials control system for a major manufacturer of digital telephones. Your development budget is \$430,000. Approximately how much money should you devote to each phase of the development life cycle? How much do you expect future maintenance will cost?
- 1.2 You are a software-engineering consultant. The executive vice-president of a publisher of paperback books wants you to develop a product that will carry out all the accounting functions of the company and provide online information to the head office staff regarding orders and inventory in the various company warehouses. Terminals are required for 15 accounting clerks, 32 order clerks, and 42 warehouse clerks. In addition, 18 managers need access to the data. The president is willing to pay \$30,000 for the hardware and the software together and wants the complete product in 4 weeks. What do you tell him? Bear in mind that, as a consultant, you want his business, no matter how unreasonable his request.
- 1.3 You are a vice-admiral of the Navy of the Republic of Claremont. It has been decided to call in a software development organization to develop the control software for a new generation of ship-to-ship missiles. You are in charge of supervising the project. To protect the government of Claremont, what clauses do you include in the contract with the software developers?
- 1.4 You are a software engineer and your job is to supervise the development of the software in Problem 1.3. List ways your company can fail to satisfy the contract with the Navy. What are the probable causes of such failures?
- 1.5 Fifteen months after delivery, a fault is detected in a mechanical engineering product that determines the optimal viscosity of oil in internal combustion engines. The cost of fixing the fault is \$18,730. The cause of the fault is an ambiguous sentence in the specification document. Approximately how much would it have cost to have corrected the fault during the specification phase?
- 1.6 Suppose that the fault in Problem 1.5 had been detected during the implementation phase. Approximately how much would it have cost to have fixed it then?
- 1.7 You are the president of an organization that builds large-scale software. You show Figure 1.5 to your employees, urging them to find faults early in the software life cycle. Someone responds that it is unreasonable to expect anyone to remove faults before having entered the product. For example, how can anyone remove a fault during the design phase if the fault in question is a coding fault? What do you reply?

- 1.8** Look up the word *system* in a dictionary. How many different definitions are there? Write down those definitions that are applicable within the context of software engineering.
- 1.9** It is your first day at your first job. Your manager hands you a listing and says, "See if you can find the bug." What do you reply?
- 1.10** You are in charge of developing the product in Problem 1.1. Will you use the object-oriented paradigm or the structured paradigm? Give reasons for your answer.
- 1.11** (Term Project) Suppose that the Broadlands Area Children's Hospital (BACH) product of Appendix A has been implemented exactly as described. Now the Joining Children with their Families (JCF) program is to be extended to all patients, not just those living within a 500-mile radius of the city of Broadlands. In what ways will the existing product have to be changed? Would it be better to discard everything and start again from scratch?
- 1.12** (Readings in Software Engineering) Your instructor will distribute copies of [Reel, 1999]. How would you manage a product to be able to detect any signs of project failure as early as possible?

REFERENCES

- [Baetjer, 1996] H. BAETJER, *Software as Capital: An Economic Perspective on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1996.
- [Bhandari et al., 1994] I. BHANDARI, M. J. HALLIDAY, J. CHAAR, R. CHILLAREGE, K. JONES, J. S. ATKINSON, C. LEPORI-COSTELLO, P. Y. JASPER, E. D. TARVER, C. C. LEWIS, AND M. YONEZAWA, "In-Process Improvement through Defect Data Interpretation," *IBM Systems Journal* 33 (No. 1, 1994), pp. 182–214.
- [Boehm, 1976] B. W. BOEHM, "Software Engineering," *IEEE Transactions on Computers* C-25 (December 1976), pp. 1226–41.
- [Boehm, 1979] B. W. BOEHM, "Software Engineering, R & D Trends and Defense Needs," in *Research Directions in Software Technology*, P. Wegner (Editor), The MIT Press, Cambridge, MA, 1979.
- [Boehm, 1980] B. W. BOEHM, "Developing Small-Scale Application Software Products: Some Experimental Results," *Proceedings of the Eighth IFIP World Computer Congress*, October 1980, pp. 321–26.
- [Boehm, 1981] B. W. BOEHM, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.
- [Brereton et al., 1999] P. BRERETON, D. BUDGEN, K. BENNETT, M. MUNRO, P. LAZZELL, L. MACAULAY, D. GRIFFITHS, AND C. STANNETT, "The Future of Software," *Communications of the ACM* 42 (December 1999), pp. 78–84.
- [Brooks, 1975] F. P. BROOKS, JR., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, MA, 1975. Twentieth Anniversary Edition, Addison-Wesley, Reading, MA, 1995.
- [Budd, 1991] T. A. BUDD, *An Introduction to Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1991.

- [Capper, Colgate, Hunter, and James, 1994] N. P. CAPPER, R. J. COLGATE, J. C. HUNTER, AND M. F. JAMES, "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories," *IBM Systems Journal* 33 (No. 1, 1994), pp. 131-57.
- [Coleman, Ash, Lowther, and Oman, 1994] D. COLEMAN, D. ASH, B. LOWTHER, AND P. OMAN, "Using Metrics to Evaluate Software System Maintainability," *IEEE Computer* 27 (August 1994), pp. 44-49.
- [Daly, 1977] E. B. DALY, "Management of Software Development," *IEEE Transactions on Software Engineering* SE-3 (May 1977), pp. 229-42.
- [DeMarco and Lister, 1987] T. DEMARCO AND T. LISTER, *Peopleware: Productive Projects and Teams*, Dorset House, New York, 1987.
- [DeMarco and Lister, 1989] T. DEMARCO AND T. LISTER, "Software Development: The State of the Art vs. State of the Practice," *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 271-75.
- [Ebert, Matsubara, Pezzé, and Bertelsen, 1997] C. EBERT, T. MATSUBARA, M. PEZZÉ, AND O. W. BERTELSEN, "The Road to Maturity: Navigating between Craft and Science," *IEEE Software* 14 (November/December 1997), pp. 77-88.
- [El-Rewini et al., 1995] H. EL-REWINI, S. HAMILTON, Y.-P. SHAN, R. EARLE, S. MCGAUGHEY, A. HELAL, R. BADRACHALAM, A. CHIEN, A. GRIMSHAW, B. LEE, A. WADE, D. MORSE, A. ELMAGRAMID, E. PITOURA, R. BINDER, AND P. WEGNER, "Object Technology," *IEEE Computer* 28 (October 1995), pp. 58-72.
- [Elshoff, 1976] J. L. ELSHOFF, "An Analysis of Some Commercial PL/I Programs," *IEEE Transactions on Software Engineering* SE-2 (June 1976), 113-20.
- [Fagan, 1974] M. E. FAGAN, "Design and Code Inspections and Process Control in the Development of Programs," Technical Report IBM-SSD TR 21.572, IBM Corporation, December 1974.
- [Fichman and Kemerer, 1997] R. G. FICHMAN AND C. F. KEMERER, "Object Technology and Reuse: Lessons from Early Adopters," *IEEE Computer* 30 (July 1997), pp. 47-57.
- [Finkelstein, 2000] A. FINKELSTEIN (EDITOR), *The Future of Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 2000.
- [Gibbs, 1994] W. W. GIBBS, "Software's Chronic Crisis," *Scientific American* 271 (September 1994), pp. 86-95.
- [Glass, 1998] R. L. GLASS, "Is There Really a Software Crisis?" *IEEE Software* 15 (January/February 1998), pp. 104-5.
- [Grady, 1994] R. B. GRADY, "Successfully Applying Software Metrics," *IEEE Computer* 27 (September 1994), pp. 18-25.
- [IEEE 610.12, 1990] "A Glossary of Software Engineering Terminology," IEEE 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., New York, 1990.
- [ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.
- [Johnson, 2000] R. A. JOHNSON, "The Ups and Downs of Object-Oriented System Development," *Communications of the ACM* 43 (October 2000), pp. 69-73.
- [Josephson, 1992] M. JOSEPHSON, *Edison: A Biography*, John Wiley and Sons, New York, 1992.
- [Kan et al., 1994] S. H. KAN, S. D. DULL, D. N. AMUNDSON, R. J. LINDNER, AND R. J. HEDGER, "AS/400 Software Quality Management," *IBM Systems Journal* 33 (No. 1, 1994), pp. 62-88.
- [Kelly, Sherif, and Hops, 1992] J. C. KELLY, J. S. SHERIF, AND J. HOPS, "An Analysis of Defect Densities Found during Software Inspections," *Journal of Systems and Software* 17 (January 1992), pp. 111-17.

- [Khan, Al-A'ali, and Girgis, 1995] E. H. KHAN, M. AL-A'ALI, AND M. R. GIRGIS, "Object-Oriented Programming for Structured Procedural Programming," *IEEE Computer* 28 (October 1995), pp. 48-57.
- [Kroeker et al., 1999] K. K. KROEKER, L. WALL, D. A. TAYLOR, C. HORN, P. BASSETT, J. K. OUSTERHOUT, M. L. GRISS, R. M. SOLEY, J. WALDO, AND C. SIMONYI, "Software [R]evolution: A Roundtable," *IEEE Computer* 32 (May 1999), pp. 48-57.
- [Leveson, 1997] N. G. LEVESON, "Software Engineering: Stretching the Limits of Complexity," *Communications of the ACM* 40 (February 1997), pp. 129-31.
- [Leveson and Turner, 1993] N. G. LEVESON AND C. S. TURNER, "An Investigation of the Therac-25 Accidents," *IEEE Computer* 26 (July 1993), pp. 18-41.
- [Lewis, 1996a] T. LEWIS, "The Next 10,000₂ Years: Part I," *IEEE Computer* 29 (April 1996), pp. 64-70.
- [Lewis, 1996b] T. LEWIS, "The Next 10,000₂ Years: Part II," *IEEE Computer* 29 (May 1996), pp. 78-86.
- [Lientz, Swanson, and Tompkins, 1978] B. P. LIENTZ, E. B. SWANSON, AND G. E. TOMPKINS, "Characteristics of Application Software Maintenance," *Communications of the ACM* 21 (June 1978), pp. 466-71.
- [Littlewood and Strigini, 1992] B. LITTLEWOOD AND L. STRIGINI, "The Risks of Software," *Scientific American* 267 (November 1992), pp. 62-75.
- [Maring, 1996] B. MARING, "Object-Oriented Development of Large Applications," *IEEE Software* 13 (May 1996), pp. 33-40.
- [Mellor, 1994] P. MELLOR, "CAD: Computer-Aided Disaster," Technical Report, Centre for Software Reliability, City University, London, U.K., July 1994.
- [Meyer, 1992a] B. MEYER, "Applying 'Design by Contract'," *IEEE Computer* 25 (October 1992), pp. 40-51.
- [Meyer, 1997] B. MEYER, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 1997.
- [Naur, Randell, and Buxton, 1976] P. NAUR, B. RANDELL, AND J. N. BUXTON (Editors), *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*, Petrocelli-Charter, New York, 1976.
- [Neumann, 1980] P. G. NEUMANN, Letter from the Editor, *ACM SIGSOFT Software Engineering Notes* 5 (July 1980), p. 2.
- [Neumann, 1995] P. G. NEUMANN, *Computer-Related Risks*, Addison-Wesley, Reading, MA, 1995.
- [Parnas, 1990] D. L. PARNAS, "Education for Computing Professionals," *IEEE Computer* 23 (January 1990), pp. 17-22.
- [Parnas, 1994] D. L. PARNAS, "Software Aging," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 279-87.
- [Radin, 1996] G. RADIN, "Object Technology in Perspective," *IBM Systems Journal* 35 (No. 2, 1996), pp. 124-126.
- [Reel, 1999] J. S. REEL, "Critical Success Factors in Software Projects," *IEEE Software* 16 (May/June 1999), pp. 18-23.
- [Scholtz et al., 1993] J. SCHOLTZ, S. CHIDAMBER, R. GLASS, A. GOERNER, M. B. ROSSON., M. STARK, AND I. VESSEY, "Object-Oriented Programming: The Promise and the Reality," *Journal of Systems and Software* 23 (November 1993), pp. 199-204.
- [Shapiro, 1994] F. R. SHAPIRO, "The First Bug," *Byte* 19 (April 1994), p. 308.
- [Shneiderman, 1980] B. SHNEIDERMAN, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, Cambridge, MA, 1980.
- [Stephenson, 1976] W. E. STEPHENSON, "An Analysis of the Resources Used in Safeguard System Software Development," Bell Laboratories, Draft Paper, August 1976.

- [Wasserman, 1996] A. I. WASSERMAN, "Toward a Discipline of Software Engineering," *IEEE Software* 13 (November/December 1996), pp. 23-31.
- [Webster, 1995] B. F. WEBSTER, *Pitfalls of Object-Oriented Development*, M&T Books, New York, 1995.
- [Weinberg, 1971] G. M. WEINBERG, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.
- [Wirfs-Brock, Wilkerson, and Wiener, 1990] R. WIRFS-BROCK, B. WILKERSON, AND L. WIENER, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [Yourdon, 1996] E. YOURDON, *Rise and Resurrection of the American Programmer*, Yourdon Press, Upper Saddle River, NJ, 1996.
- [Zelkowitz, Shaw, and Gannon, 1979] M. V. ZELKOWITZ, A. C. SHAW, AND J. D. GANNON, *Principles of Software Engineering and Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.
- [Zvegintzov, 1998] N. ZVEGINTZOV, "Frequently Begged Questions and How to Answer Them," *IEEE Software* 15 (January/February 1998), pp. 93-96.

chapter

3

SOFTWARE LIFE-CYCLE MODELS

A software product usually begins as a vague concept, such as “Wouldn’t it be nice if the computer could plot our graphs of radioactivity levels,” or “If this corporation doesn’t have an exact picture of our cash flow on a daily basis, we will be insolvent in six months,” or even “If we develop and market this new type of spreadsheet, we’ll make a million dollars!” Once the need for a product has been established, the product goes through a series of development phases. Typically, the product is specified, designed, and implemented. If the client is satisfied, the product is installed and, while operational, maintained. When the product finally comes to the end of its useful life, it is decommissioned. The series of steps through which the product progresses is called the *life-cycle model*.

The life-cycle history of each product is different. Some products spend years in the conceptual stage, perhaps because current hardware just is not fast enough for the product to be viable or because fundamental research has to be done before an efficient algorithm can be developed. Other products are quickly designed and implemented, then spend years in the maintenance phase being modified to meet the users’ changing needs. Yet other products are designed, implemented, and maintained; then after many years of radical maintenance, it will be cheaper to develop a completely new product rather than attempt to patch the current version yet again.

In this chapter, a number of different life-cycle models are described. The two most widely used models are the waterfall model and the rapid prototyping model. In addition, the spiral model now is receiving considerable attention. To help shed light on the strengths and weaknesses of these three models, other life-cycle models also are examined, including incremental models, the synchronize-and-stabilize model, and the highly unsatisfactory build-and-fix model.

3.1 BUILD-AND-FIX MODEL

It is unfortunate that many products are developed using what might be termed the *build-and-fix* model. The product is constructed with no specifications or attempt at design. Instead, the developers simply build a product that is reworked as many times

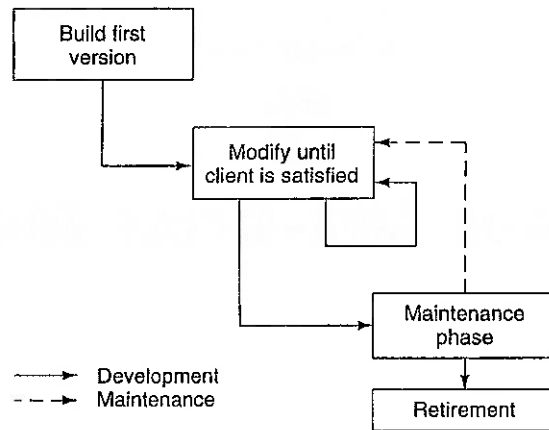


Figure 3.1 Build-and-fix model.

as necessary to satisfy the client. This approach is shown in Figure 3.1. Although this approach may work well on short programming exercises 100 or 200 lines long, the build-and-fix model is totally unsatisfactory for products of any reasonable size. Figure 1.5 shows that the cost of changing a software product is relatively small if the change is made during the requirements, specification, or design phases but grows unacceptably large if changes are made after the product has been coded or, worse, if it is already in the maintenance phase. Thus, the cost of the build-and-fix approach actually is far greater than the cost of a properly specified, carefully designed product. In addition, the maintenance of a product can be extremely difficult with no specification or design documents and the chance of a regression fault occurring is considerably greater.

Instead of the build-and-fix approach, it is essential that, before development of a product begins, an overall game plan or *life-cycle model* be chosen. The life-cycle model (sometimes abbreviated to just *model*) specifies the various phases of the software process, such as the requirements, specification, design, implementation, integration, and maintenance phases, and the order in which they are to be carried out. Once the life-cycle model has been agreed to by all parties, development of the product can begin.

Until the early 1980s, the waterfall model was the only widely accepted life-cycle model. This model is now examined in some detail.

3.2 WATERFALL MODEL

The waterfall model was first put forward by Royce [Royce, 1970]. A version of the model appears as Figure 3.2. First, requirements are determined and checked by

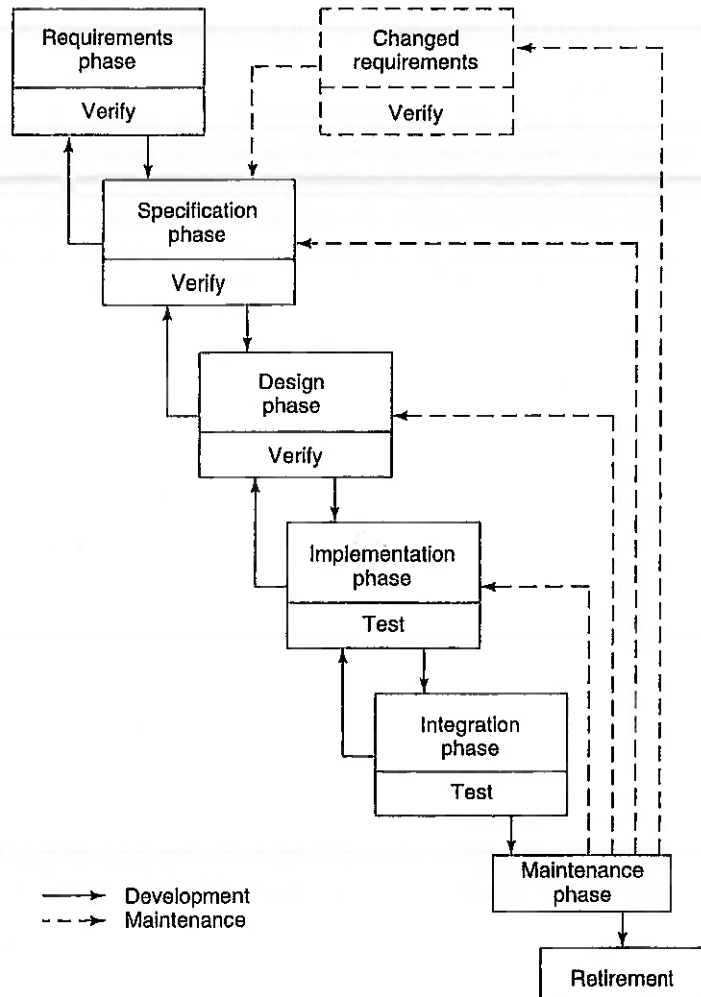


Figure 3.2 Waterfall model.

the client and members of the software quality assurance (SQA) group. Then the specifications for the product are drawn up; that is, a document is produced stating what the product is to do. The specifications are checked by the SQA group and shown to the client. Once the client has signed off the specification document, the next step is to draw up the software project management plan, a detailed timetable for developing the software. This plan also is checked by the SQA group. When the client has approved the developers' duration and cost estimates for the product, the design phase begins. In contrast to the specification document that describes *what* the product is to do, the design documents describe *how* the product is to do it.

During the design phase, sometimes a fault in the specification document becomes apparent. The specifications may be incomplete (some features of the product have been omitted), contradictory (two or more statements in the specification document define the product in incompatible ways), or ambiguous (the specification document has more than one possible interpretation). The presence of incompleteness, contradictions, or ambiguities necessitates a revision of the specification document before the software development process can continue. Referring again to Figure 3.2, the arrow from the left side of the design phase box back to the specification phase box constitutes a feedback loop. The software production process follows this loop if the developers have to revise the specification document during the design phase. With the client's permission, the necessary changes are made to the specification document, and the planning and design documents are adjusted to incorporate these changes. When the developers finally are satisfied, the design documents are handed to the programmers for implementation.

Flaws in the design may appear during implementation. For example, the design of a real-time system may prove to be too slow when implemented. An example of such a design flaw results from the fact that most compilers generate code to store the elements of an array b in row-major order, that is, in the order $b(1, 1)$, $b(1, 2)$, $b(1, 3)$, ..., $b(1, n)$, $b(2, 1)$, $b(2, 2)$, $b(2, 3)$, ..., $b(2, n)$, and so on. Suppose a 200×200 array b is stored on disk with one row to a block, that is, a 200-word row is read into a buffer in main memory each time a *read* statement is executed. The complete array is to be read from disk into main memory. If the array is read row by row, then exactly 200 blocks have to be transferred from the disk to main memory to read all 40,000 elements. The first *read* statement causes the first row to be put in the buffer, and the first 200 *reads* use the contents of the buffer. Only when the 201st element is required does a second block need to be transferred from disk to main memory. But, if the product reads the array column by column, then a fresh block has to be transferred for every *read*, because consecutive *reads* access different rows and, hence, different blocks. Thus, 40,000 block transfers would be required, instead of 200 when the array is read in row-major order, and the input-output time for that part of the product would be 200 times longer. Design faults of this type must be corrected before the team can continue software development.

During the implementation phase, the waterfall model with its feedback loops permits modifications to be made to the design documents, the specification document, and even the requirements, if necessary. Modules are implemented, documented, and integrated to form a complete product. (In practice, the implementation and integration phases usually are carried out in parallel. As described in Chapter 15, each module is integrated as soon as it has been implemented and tested.) During integration it may be necessary to backtrack and make modifications to the code, preceded perhaps by modifications to the specification and design documents.

A critical point regarding the waterfall model is that no phase is complete until the documentation for that phase has been completed and the products of that phase have been approved by the SQA group. This carries over into modifications; if the products of an earlier phase have to be changed as a consequence of following a feedback loop, that earlier phase is deemed to be complete only when the documentation for the phase has been modified and the modifications have been checked by the SQA group.

nelle
 Design
 proprietà e
 dei problemi
 nelle specifiche
 Ci si può
 sfuggire che
 le specifiche
 → incomplete
 - contraddittorie,
 definiti non le
 prodotti in
 nomele compatibilità
 0
 → ambiguo

When the developers feel that the product has been successfully completed, it is given to the client for acceptance testing. Deliverables at this stage include the user manual and other documentation listed in the contract. When the client agrees that the product indeed satisfies its specification document, the product is handed over to the client and installed on the client's computer.

Once the client has accepted the product, any changes, whether to remove residual faults or to extend the product in any way, constitute maintenance. As can be seen in Figure 3.2, maintenance may require not just implementation changes but also design and specification changes. In addition, enhancement is triggered by a change in requirements. This, in turn, is implemented via changes in the specification document, design documents, and code. The waterfall model is a dynamic model, and the feedback loops play an important role in this dynamism. Again, it is vital that the documentation be maintained as meticulously as the code itself and the products of each phase be checked carefully before the next phase commences.

The waterfall model has been used with great success on a wide variety of products. However, there have also been failures. To decide whether or not to use the waterfall model for a project, it is necessary to understand both its strengths and weaknesses.

3.2.1 ANALYSIS OF THE WATERFALL MODEL

The waterfall model has many advantages, including the enforced disciplined approach—the stipulation that documentation be provided at each phase and the requirement that all the products of each phase (including the documentation) be checked carefully by SQA. An essential aspect of the milestone terminating each phase is approval by the SQA group of all the products of that phase, including all the documentation stipulated for that phase.

Inherent in every phase of the waterfall model is testing. Testing is not a separate phase to be performed only after the product has been constructed, it is not to be performed only at the end of each phase. Instead, as stated in Chapter 2, testing should proceed continuously throughout the software process. Specifically, while the requirements are being drawn up they must be verified, as must the specification document and the software project management plan as well as the design documents. The code must be tested in a variety of ways. During maintenance it is necessary to ensure not only that the modified version of the product still does what the previous version did—and still does it correctly (regression testing)—but that it totally satisfies any new requirements imposed by the client.

The specification document, design documents, code documentation, and other documentation such as the database manual, user manual, and operations manual are essential tools for maintaining the product. As stated in Chapter 1, on average 67 percent of a software budget is devoted to maintenance, and adherence to the waterfall model with its documentation stipulations make this maintenance easier. As mentioned in the previous section, the same methodical approach to software production continues during maintenance. Every change must be reflected in the

relevant documentation. Many of the successes of the waterfall model have been due to its essence as a documentation-driven model.

However, the fact that the waterfall model is documentation driven also can be a disadvantage. To see this, consider the following two somewhat bizarre scenarios. First, Joe and Jane Johnson decide to build a house. They consult with an architect. Instead of showing them sketches, plans, and perhaps a model, the architect gives them a 20-page single-spaced typed document describing the house in highly technical terms. Even though neither Joe nor Jane has any previous architectural experience and hardly understands the document, they enthusiastically sign it and say, "Go right ahead, build the house."

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

The preceding two scenarios are highly unlikely. Nevertheless, they typify precisely the way software often is constructed using the waterfall model. The process begins with the specifications. In general, specification documents are long, detailed, and quite frankly, boring to read. The client usually is inexperienced in reading software specifications, and this difficulty is compounded because the specification documents usually are written in a style with which the client is unfamiliar. The difficulty is even worse when the specifications are written in a formal specification language like Z [Spivey, 1992] (Section 11.8). Nevertheless, the client proceeds to sign off the specification document, whether properly understood or not. In many ways, there is little difference between Joe and Jane Johnson contracting to have a house built from a written description they only partially comprehend and clients approving a software product described in terms of a specification document that they only partially understand.

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

What has gone wrong? There is a considerable difference between the way a client understands a product as described by the specification document and the actual product. The specifications exist only on paper; the client therefore cannot really understand what the product itself will be like. The waterfall model, depending as it so crucially does on written specifications, can lead to the construction of products that simply do not meet the clients' real needs.

In fairness, it should be pointed out that, just as an architect can help a client to understand what is to be built by providing models, sketches, and plans, so the software engineer can use graphical techniques, such as data flow diagrams (Section 11.3.1), to communicate with the client. The problem is that these graphical aids do not describe how the finished product will work. For example, there is a considerable difference between a flowchart (a diagrammatic description of a product) and the working product itself.

The strength of the next life-cycle model to be examined, the rapid prototyping model, is that it can help ensure that the client's real needs are met.

3.3 RAPID PROTOTYPING MODEL

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

The first step in the rapid prototyping life-cycle model depicted in Figure 3.3 is to build a rapid prototype and let the client and future users interact and experiment with the rapid prototype. Once the client is satisfied that the rapid prototype indeed does most of what is required, the developers can draw up the specification document with some assurance that the product meets the client's real needs.

Having constructed the rapid prototype, the software process continues as shown in Figure 3.3. A major strength of the rapid prototyping model is that the development of the product essentially is linear, proceeding from the rapid prototype to the delivered product; the feedback loops of the waterfall model (see Figure 3.2) are less likely to be needed in the rapid prototyping model. There are a number of reasons for this. First, the members of the development team use the rapid prototype to construct the specification document. Because the working rapid prototype has been validated through interaction with the client, it is reasonable to expect that the resulting specification document will be correct. Second, consider the design phase. Even though the rapid prototype (quite rightly) has been hurriedly assembled, the design team can gain insights from it—at worst, they will be of the “how not to do it” variety. Again, the feedback loops of the waterfall model are less likely to be needed here.

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

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

An essential aspect of a rapid prototype is embodied in the word *rapid*. The developers should endeavor to construct the prototype as rapidly as possible to speed up the software development process. After all, the sole use of the rapid prototype is to determine what the client's real needs are; once this has been determined, the rapid prototype implementation is discarded but the lessons learned are retained and used in subsequent development phases. For this reason, the internal structure of the rapid

Si uso il
rapid prototyping
science da
specifica

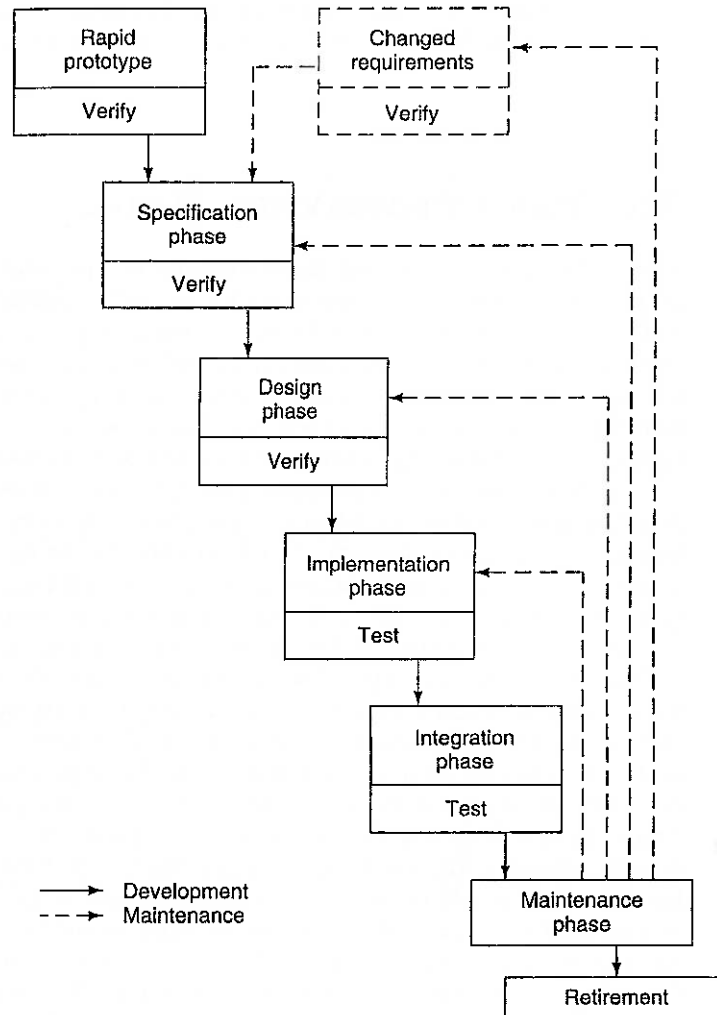


Figure 3.3 Rapid prototyping model.

prototype is not relevant. What is important is that the prototype be built rapidly and modified rapidly to reflect the client's needs. Therefore, speed is of the essence.

3.3.1 INTEGRATING THE WATERFALL AND RAPID PROTOTYPING MODELS

Despite the many successes of the waterfall model, it has a major drawback in that what is delivered to the client may not be what the client really needs. The rapid prototyping model has also had many successes. Nevertheless, as described in Chapter 10, it has not yet been proven beyond all doubt, and the model may have weaknesses of its own.

One solution is to combine the two approaches: This can be seen by comparing the phases of Figure 3.2 (waterfall model) with those of Figure 3.3 (rapid prototyping model). Rapid prototyping can be used as a requirements analysis technique; in other words, the first step is to build a rapid prototype to determine the client's real needs, then to use that rapid prototype as the input to the waterfall model.

This approach has a useful side effect. Some organizations are reluctant to use the rapid prototyping approach because of the risks involved in using any new technology. Introducing rapid prototyping into the organization as a front end to the waterfall model gives management the opportunity to assess the technique while minimizing the associated risk.

The rapid prototyping model is analyzed in greater detail in Chapter 10, where the requirements phase is described. A different class of life-cycle model is now examined.

3.4 INCREMENTAL MODEL

Software is built, not written. That is, software is constructed step by step, in the same way that a building is constructed. While a software product is in the process of being developed, each step adds to what has gone before. One day the design is extended; the next day another module is coded. The construction of the complete product proceeds incrementally in this way until completion.

Of course, it is not quite true that progress is made every day. Just as a contractor occasionally has to tear down an incorrectly positioned wall or replace a pane of glass that a careless painter has cracked, it sometimes is necessary to respecify, redesign, recode, or at worst, throw away what already has been completed and start again. But the fact that the product sometimes advances in fits and starts does not negate the basic reality that a software product is built piece by piece.

The realization that software is engineered incrementally has led to the development of a model that exploits this aspect of software development, the so-called *incremental model* shown in Figure 3.4. The product is designed, implemented, integrated, and tested as a series of incremental *builds*, where a build consists of code pieces from various modules interacting to provide a specific functional capability.

For example, if the product is to control a nuclear submarine, then the navigation system could constitute a build, as could the weapons control system. In an operating system, the scheduler could be a build and so could the file management system. At each stage of the incremental model, a new build is coded then integrated into the structure, which is tested as a whole. The process stops when the product achieves the target functionality, that is, when the product satisfies its specifications. The developer is free to break up the target product into builds as he or she sees fit, subject only to the constraint that, as each build is integrated into the existing software, the resulting product must be testable. If the product is broken into too few builds, then the incremental model degenerates into the build-and-fix approach (Section 3.1). Con-

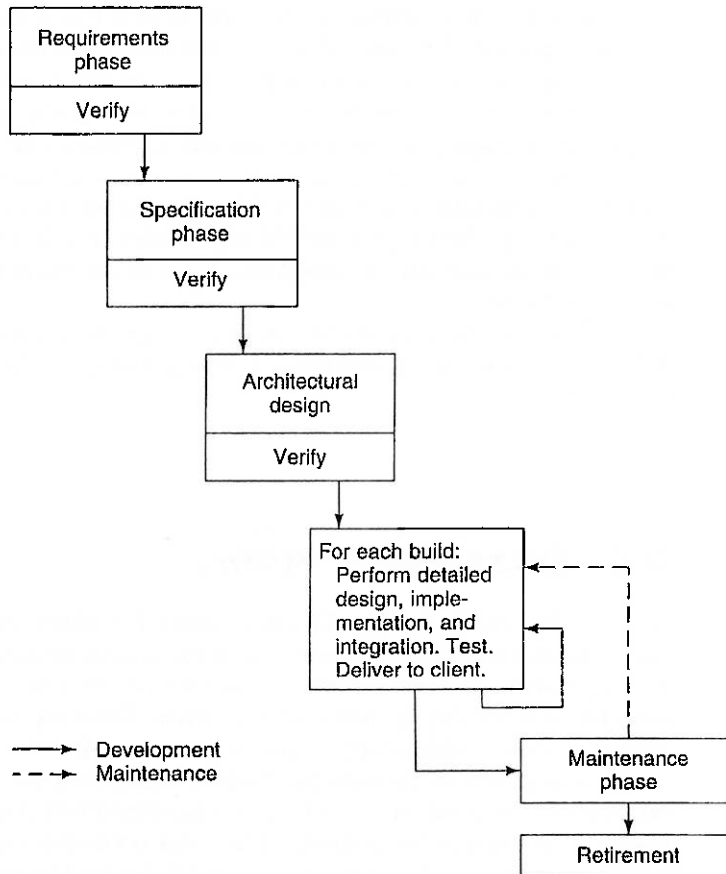


Figure 3.4 Incremental model.

versely, if the product consists of too many builds, then at each stage, considerable time is spent in the integration testing of only a small amount of additional functionality. What constitutes an optimal decomposition into builds varies from product to product and from developer to developer.

The strengths and weaknesses of the incremental model are now presented.

3.4.1 ANALYSIS OF THE INCREMENTAL MODEL

The aim of both the waterfall and rapid prototyping models is delivery to the client of a complete, operational-quality product. That is, the client is presented with a product that satisfies *all* requirements and is ready for use. Correct use of either the waterfall or the rapid prototyping model results in a product that will have been thoroughly tested, and the client should be confident that the product can be utilized for the

purpose for which it was developed. Furthermore, the product comes with adequate documentation so that, not only can it be used by the client's organization, but all three types of maintenance—adaptive, perfective, and corrective (Section 1.3)—can be performed as necessary. With both models there is a projected delivery date, and the intention is to deliver the complete product in full working order on or before that date.

In contrast, the incremental model delivers an operational-quality product at each stage, but one that satisfies only a subset of the client's requirements. The complete product is divided into builds, and the developer delivers the product build by build. A typical product usually consists of 5 to 25 builds. At each stage, the client has an operational-quality product that does a portion of what is required; from delivery of the first build, the client is able to do useful work. With the incremental model, portions of the total product might be available within weeks, whereas the client generally waits months or years to receive a product built using the waterfall or rapid prototyping model.

Another advantage of the incremental model is that it reduces the traumatic effect of imposing a completely new product on the client organization. The gradual introduction of the product via the incremental model provides time for the client to adjust to the new product. Change is an integral part of every growing organization; because a software product is a model of reality, the need for change is an integral part of delivered software. Change and adaptation are natural to the incremental model, whereas change can be a threat when products are developed and introduced in one large step.

From the client's financial viewpoint, phased delivery requires no large capital outlay. Instead, there is an excellent cash flow, particularly if the earliest builds are chosen on the basis of delivering a high return on investment. A related advantage of the incremental model is that it is not necessary to complete the product to get a return on investment. Instead, the client can stop development of the product at any time.

A difficulty with the incremental model is that each additional build somehow has to be incorporated into the existing structure without destroying what has been built to date. Furthermore, the existing structure must lend itself to extension in this way, and the addition of each succeeding build must be simple and straightforward. Although this need for an open architecture certainly is a short-term difficulty, in the long term it can be a real strength. Every product undergoes development, followed by maintenance. During development it is indeed important to have clear specifications and a coherent and cohesive design. But once a product enters the maintenance phase, the requirements for that product change, and radical enhancement easily can destroy a coherent and cohesive design to the extent that further maintenance becomes impossible. In such a case, the product must be rebuilt virtually from scratch. That the design must be open-ended is not merely a prerequisite for development using the incremental model but is essential for maintenance, irrespective of the model selected for the development phase. Thus, although the incremental model may require more careful design than the holistic waterfall and rapid prototyping models, the payoff comes in the maintenance phase. If a design is flexible enough to support the incremental model, then it certainly will allow virtually any sort of maintenance without

falling apart. In fact, the incremental model does not distinguish between developing a product and enhancing (maintaining) it; each enhancement is merely an additional build.

All too frequently, the requirements change while development is in progress; this problem is discussed in greater detail in Section 16.4.4. The flexibility of the incremental model gives it a major advantage over the waterfall and rapid prototyping models in this regard. On the negative side, the incremental model too easily can degenerate into the build-and-fix approach. Control of the process as a whole can be lost, and the resulting product, instead of being open-ended, becomes a maintainer's nightmare. In a sense, the incremental model is a contradiction in terms, requiring the developer to view the product as a whole in order to begin with a design that will support the entire product, including future enhancements, and simultaneously to view that product as a sequence of builds, each essentially independent of the next. Unless the developer is skilled enough to be able to handle this apparent contradiction, the incremental model may lead to an unsatisfactory product.

In the incremental model of Figure 3.4, the requirements, specifications, and architectural design all must be completed before implementation of the various builds commences. A more risky concurrent version of the incremental model is depicted in Figure 3.5. Once the client's requirements have been elicited, the specifications of the first build are drawn up. When this has been completed, the specification team turns to the specifications of the second build while the design team designs the first build. Thus, the various builds are constructed in parallel, with each team using information gained in all the previous builds.

This approach incurs the real risk that the resulting builds will not fit together. With the incremental model of Figure 3.4, the need for the specification and architectural design to be completed before starting the first build means an overall design at the start. With the concurrent incremental model of Figure 3.5, unless the process is monitored carefully, the entire project risks falling apart. Nevertheless, this concurrent model has had some successes. For example, it was used by Fujitsu to develop a large-scale communication system [Aoyama, 1993].

3.5 EXTREME PROGRAMMING

Extreme programming [Beck, 1999] is a somewhat controversial new approach to software development based on the incremental model. The first step is that the software development team determines the various features (*stories*) that the client would like the product to support. For each such feature, the team informs the client how long it will take to implement that feature and how much it will cost. This first step corresponds to the requirements and specification phases of the incremental model (see Figure 3.4)

The client selects the features to be included in the each successive build using cost-benefit analysis (Section 5.2), that is, on the basis of the time and the cost estimates provided by the development team as well as the potential benefits of the

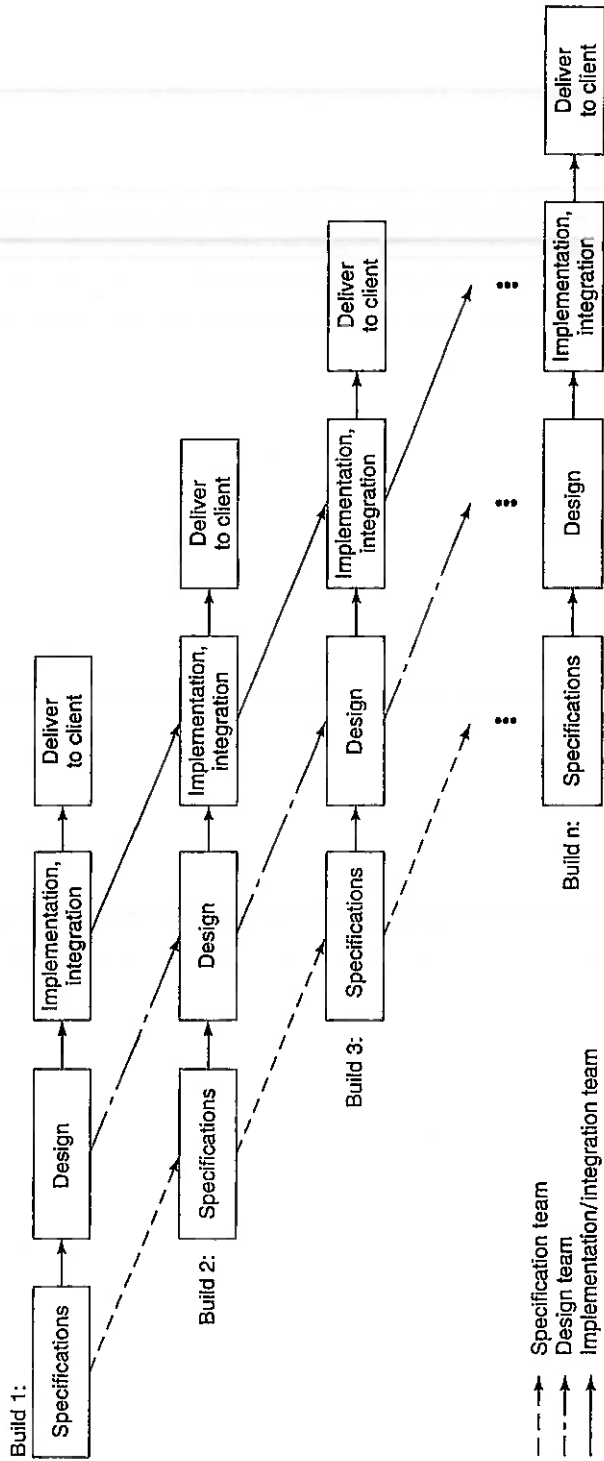


Figure 3.5 More risky concurrent incremental model.

feature to his or her business. The proposed build is broken down into smaller pieces, termed *tasks*. A programmer first draws up test cases for a task. Then, working with a partner on one screen (*pair programming*) [Williams, Kessler, Cunningham, and Jeffries, 2000], the programmer implements the task, ensuring that all the test cases work correctly. The task then is integrated into the current version of the product. Ideally, implementing and integrating a task should take no more than a few hours. In general, a number of pairs implement tasks in parallel, so integration can take place continuously. The test cases used for the task are retained and utilized in all further integration testing.

A number of features of extreme programming (XP) are somewhat unusual:

1. The computers of the XP team are set up in the center of a large room lined with small cubicles.
2. A client representative works with the XP team at all times.
3. No individual can work overtime for two successive weeks.
4. There is no specialization. Instead, all members of the XP team work on specifications, design, code, and testing.
5. As in the more risky incremental model depicted in Figure 3.5, there is no overall design phase before the various builds are constructed. Instead, the design is modified while the product is being built. This procedure is termed *refactoring*. Whenever a test case will not run, the code is reorganized until the team is satisfied that the design is simple, straightforward, and runs all the test cases satisfactorily.

XP has been used successfully on a number of small- and medium-size projects. These range from a 9 person-month shipping tariff calculation system to a 100 person-year cost analysis system [Beck, 1999]. The strength of XP is that it is useful when the client's requirements are vague or changing. However, XP has not yet been used widely enough to determine whether this version of the incremental model will fulfill its early promise.

3.6 SYNCHRONIZE-AND-STABILIZE MODEL

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

The requirements analysis phase is conducted by interviewing numerous potential customers for the package and extracting a list of features with priorities set by the customers. A specification document is drawn up. Next, the work is divided into three or four builds. The first build consists of the most critical features, the second build consists of the next most critical features, and so on. Each build is carried out by a number of small teams working in parallel. At the end of each day all the teams

synchronize, that is, they put together the partially completed components and test and debug the resulting product. *Stabilization* is performed at the end of each build. Any remaining faults that have been detected are fixed and the build is *frozen*; that is, no further changes will be made to the specifications.

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

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

3.7 SPIRAL MODEL

Almost always, an element of risk is involved in the development of software. For example, key personnel can resign before the product has been adequately documented. The manufacturer of hardware on which the product is critically dependent can go bankrupt. Too much, or too little, can be invested in testing and quality assurance. After spending hundreds of thousands of dollars on developing a major software product, technological breakthroughs can render the entire product worthless. An organization may research and develop a database management system, but before the product can be marketed, a lower-priced, functionally equivalent package is announced by a competitor. The components of a product built using the incremental model of Figure 3.5 may not fit together. For obvious reasons, software developers try to minimize such risks wherever possible.

One way of minimizing certain types of risk is to construct a prototype. As described in Section 3.3, an excellent way of reducing the risk that the delivered product will not satisfy the client's real needs is to construct a rapid prototype during the requirements phase. During subsequent phases, other sorts of prototypes may be appropriate. For example, a telephone company may devise a new, apparently highly effective algorithm for routing calls through a long-distance network. If the product is implemented but does not work as expected, the telephone company will have wasted the cost of developing the product. In addition, angry or inconvenienced customers may take their business elsewhere. This scenario can be avoided by constructing a prototype to handle only the routing of calls and testing it on a simulator. In this way, the actual system is not disturbed, and for the cost of implementing just the routing algorithm, the telephone company can determine whether it is worthwhile to develop an entire network controller incorporating the new algorithm.

The idea of minimizing risk via the use of prototypes and other means is the concept underlying the *spiral* model [Boehm, 1988]. A simplistic way of looking at this life-cycle model is as a waterfall model with each phase preceded by risk analysis, as shown in Figure 3.6. (A portion of that figure is redrawn as Figure 3.7, to reflect

the term *spiral model*.) Before commencing each phase, an attempt is made to control (or resolve) the risks. If it is impossible to resolve all the significant risks at that stage, then the project is immediately terminated.

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

Other areas of risk are less amenable to prototyping. For example, there often is a risk that the software personnel necessary to build the product cannot be hired or that key personnel may resign before the project is complete. Another potential risk is that a

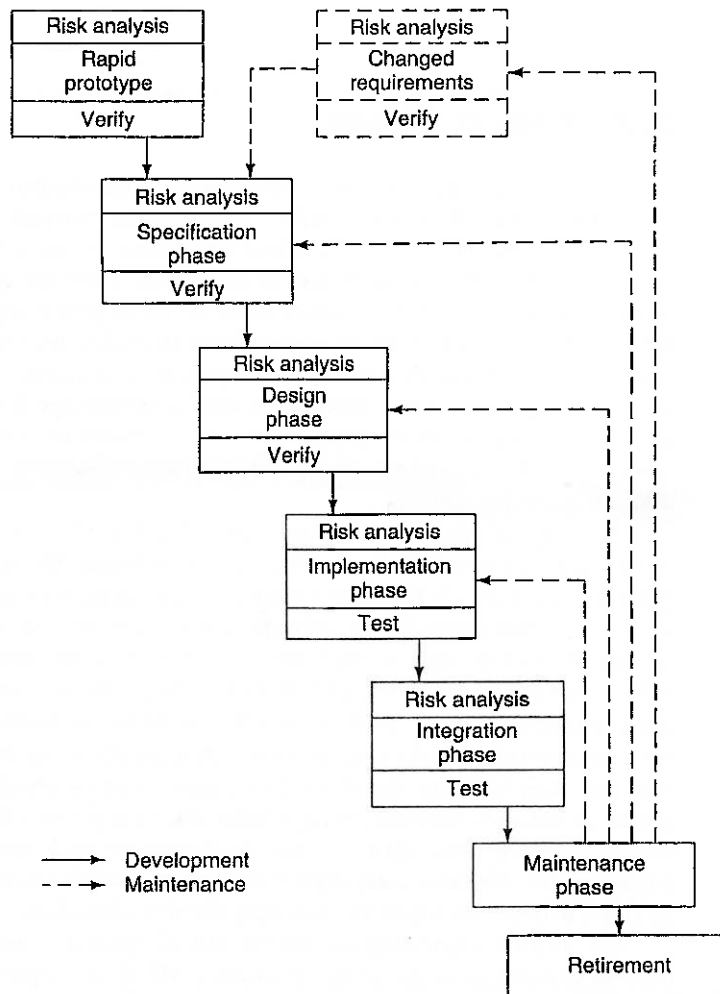


Figure 3.6 Simplistic version of spiral model.

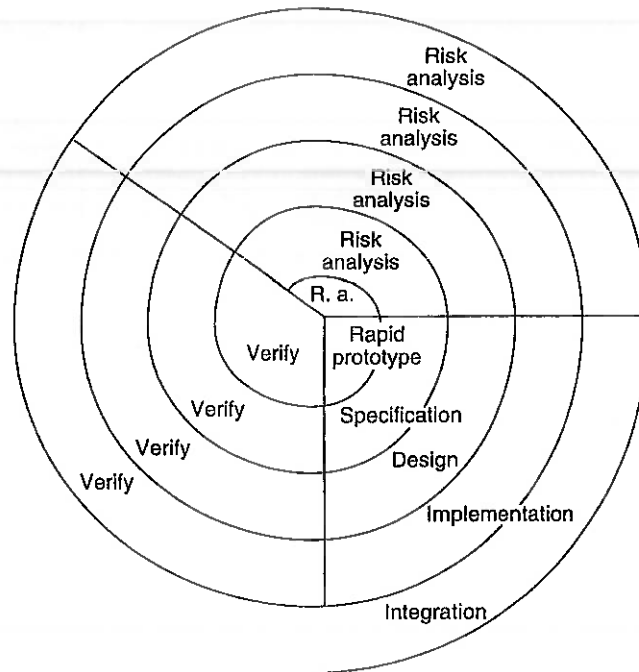


Figure 3.7 Portion of Figure 3.6 redrawn as a spiral.

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

The full spiral model is shown in Figure 3.8. The radial dimension represents cumulative cost to date, the angular dimension represents progress through the spiral. Each cycle of the spiral corresponds to a phase. A phase begins (in the top left quadrant) by determining objectives of that phase, alternatives for achieving those

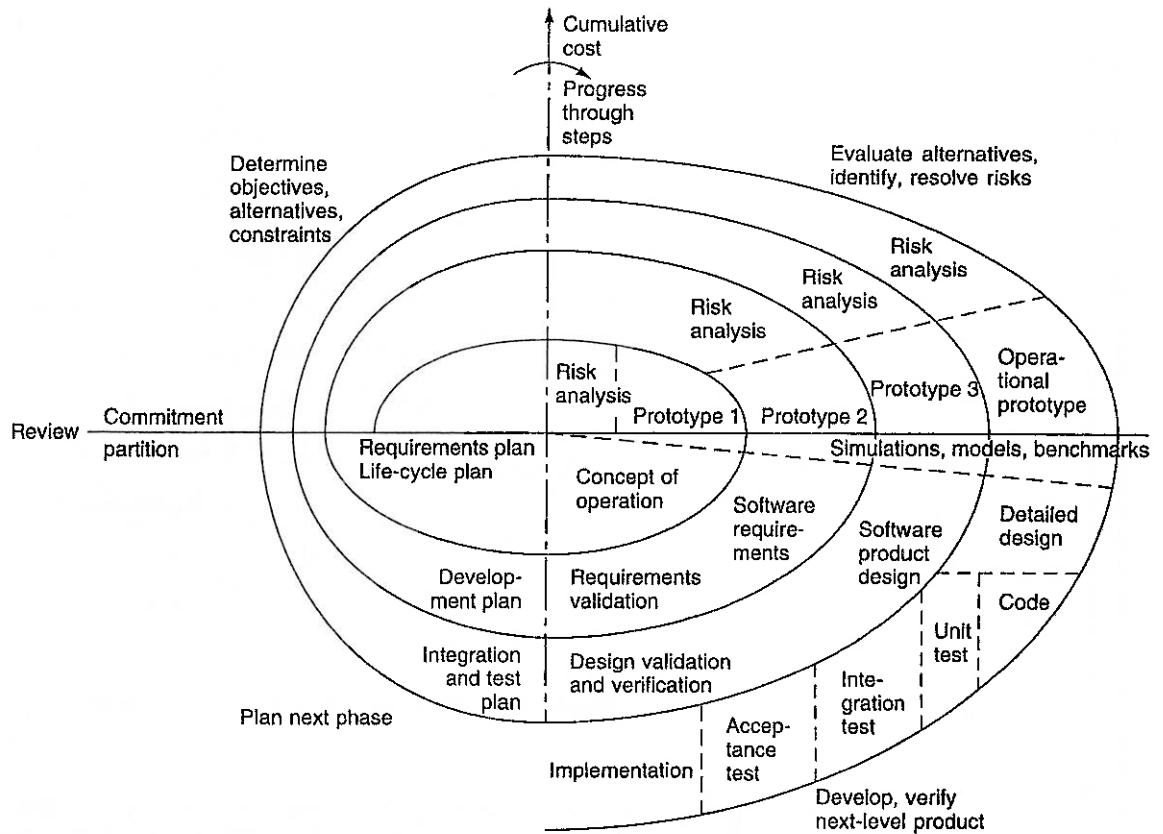


Figure 3.8 Full spiral model [Boehm, 1988]. (©1988, IEEE.)

objectives, and constraints imposed on those alternatives. This process results in a strategy for achieving those objectives. Next, that strategy is analyzed from the viewpoint of risk. Attempts are made to resolve every potential risk, in some cases by building a prototype. If certain risks cannot be resolved, the project may be terminated immediately; under some circumstances, however, a decision could be made to continue the project but on a significantly smaller scale. If all risks are successfully resolved, the next development step is started (bottom right quadrant). This quadrant of the spiral model corresponds to the pure waterfall model. Finally, the results of that phase are evaluated and the next phase is planned.

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

3.7.1 ANALYSIS OF THE SPIRAL MODEL

The spiral model has a number of strengths. The emphasis on alternatives and constraints supports the reuse of existing software (Section 8.1) and the incorporation of software quality as a specific objective. In addition, a common problem in software development is determining when the products of a specific phase have been adequately tested. Spending too much time on testing is a waste of money, and delivery of the product may be unduly delayed. Conversely, if too little testing is performed, then the delivered software may contain residual faults, resulting in unpleasant consequences for the developers. The spiral model answers this question in terms of the risks that would be incurred by not doing enough testing or by doing too much testing. Perhaps most important, within the structure of the spiral model, maintenance is simply another cycle of the spiral; essentially, no distinction is made between maintenance and development. Thus, the problem that maintenance sometimes is maligned by ignorant software professionals does not arise, because maintenance is treated the same way as development.

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

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

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

3.8 OBJECT-ORIENTED LIFE-CYCLE MODELS

Experience with the object-oriented paradigm has shown that the need for iteration between phases or portions of phases of the process appears to be more common with the object-oriented paradigm than with the structured paradigm. Object-oriented life-

cycle models have been proposed that explicitly reflect the need for iteration. One such model is the fountain model [Henderson-Sellers and Edwards, 1990] shown in Figure 3.9. The circles representing the various phases overlap, explicitly reflecting an overlap between activities. The arrows within a phase represent iteration within that phase. The maintenance circle is smaller, to symbolize reduced maintenance effort when the object-oriented paradigm is used.

In addition to the fountain model, other object-oriented life-cycle models have been put forward, including recursive/parallel life cycle [Berard, 1993], round-trip gestalt design [Booch, 1994], and the model underlying the unified software development process [Jacobson, Booch, and Rumbaugh, 1999]. All these models are iterative, incorporate some form of parallelism (overlap of activities), and support incremental development (Section 3.4). The danger of such life-cycle models is that they may be misinterpreted as simply attempts to make a virtue out of necessity, and thereby lead to a totally undisciplined form of software development in which team members move almost randomly between phases, first designing one piece of the product, next analyzing another piece, and then implementing a third piece that has been neither analyzed nor designed; the Just in Case You Wanted to Know box on page 84 gives more on this undesirable approach. A better way to proceed is to have as an overall objective a linear process (such as the rapid prototyping model of Section 3.3 or the central vertical line in Figure 3.9) but appreciate that the realities of the object-oriented paradigm are such that frequent iterations and refinements certainly are needed.

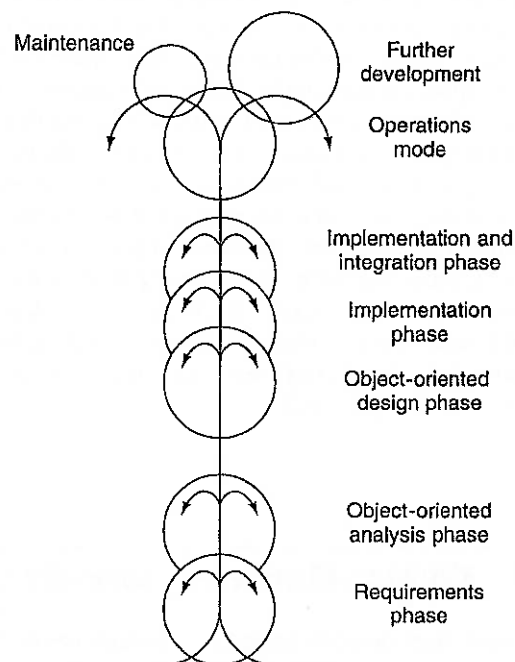


Figure 3.9 Fountain model.

JUST IN CASE YOU WANTED TO KNOW

When the members of the development team move in essentially haphazard fashion from one task to another, this is sometimes referred to as CABTAB (code a bit, test a bit). The acronym initially was used in a positive sense to refer to successful iterative models, such as those that have been used in conjunction with the

object-oriented paradigm. However, just as the word *hacker* now has a pejorative context in addition to its original meaning, so CABTAB now is also used in a derogatory sense to refer to this undisciplined approach to software development.

It might be suggested that this problem is simply a consequence of the relative newness of the object-oriented paradigm. As software professionals acquire more experience with object-oriented analysis and object-oriented design, the argument goes, and as the whole discipline matures, the need for repeated review and revision will decrease. To see that this argument is fallacious, consider the various life-cycle models previously described in this chapter. First came the waterfall model (Section 3.2) with its explicit feedback loops. The next major development was the rapid prototyping model (Section 3.3); one of its major aims was to reduce the need for iteration. However, this was followed by the spiral model (Section 3.7), which explicitly reflects an iterative approach to software development and maintenance. In addition, it has been shown [Honiden, Kotaka, and Kishimoto, 1993] that backtracking is an intrinsic aspect of the Coad-Yourdon technique for object-oriented analysis [Coad and Yourdon, 1991a], and it is likely that similar results hold for the newer object-oriented analysis techniques as well. In other words, it appears that iteration is an intrinsic property of software production in general and the object-oriented paradigm in particular.

3.9 COMPARISON OF LIFE-CYCLE MODELS

Six different classes of software life-cycle models have been examined with special attention paid to some of their strengths and weaknesses. The build-and-fix model (Section 3.1) should be avoided. The waterfall model (Section 3.2) is a known quantity. Its strengths are understood and so are its weaknesses. The rapid prototyping model (Section 3.3) was developed as a reaction to a specific perceived weakness in the waterfall model, that the delivered product may not be what the client really needs. Less is known about the newer rapid prototyping model than the familiar waterfall model, and the rapid prototyping model may have some problems of its own, as described in Chapter 10. One alternative is to combine the strengths of both models, as suggested in Section 3.3.1. Another is to use a different model, the incremental model (Section 3.4). This model, notwithstanding its successes, also has some drawbacks. Extreme programming (Section 3.5) is a controversial new approach. The

synchronize-and-stabilize model (Section 3.6) has been used with great success by Microsoft, but as yet there is no evidence of comparable successes in other corporate cultures. Yet another alternative is to use the spiral model (Section 3.7) but only if the developers are adequately trained in risk analysis and risk resolution. A further factor that needs to be considered is that, when the object-oriented paradigm is used, the life-cycle model needs to be iterative; that is, it must support feedback (Section 3.8). The strengths and weaknesses of the various life-cycle models of this chapter are summarized in Figure 3.10.

Each software development organization should decide on a life-cycle model appropriate for that organization, its management, its employees, and its software process and vary the model depending on the features of the specific product currently under development. Such a model will incorporate appropriate aspects of the various life-cycle models, utilizing their strengths and minimizing their weaknesses.

Life-Cycle Model	Strengths	Weaknesses
Build-and-fix model (Section 3.1)	Fine for short programs that will not require any maintenance	Totally unsatisfactory for nontrivial programs
Waterfall model (Section 3.2)	Disciplined approach Document-driven	Delivered product may not meet client's needs
Rapid prototyping model (Section 3.3)	Ensures that delivered product meets client's needs	Not yet proven beyond all doubt
Incremental model (Section 3.4)	Maximizes early return on investment Promotes maintainability	Requires open architecture May degenerate into build-and-fix
Extreme programming (Section 3.5)	Maximizes early return on investment Works well when client's requirements are vague	Has not yet been widely used
Synchronize-and-stabilize model (Section 3.6)	Future users' needs are met Ensures components can be successfully integrated	Has not been widely used other than at Microsoft
Spiral model (Section 3.7)	Incorporates features of all the above models	Can be used only for large-scale, in-house products Developers have to be competent in risk analysis and risk resolution
Object-oriented models (Section 3.8)	Support iteration within phases, parallelism between phases	May degenerate into CABTAB

Figure 3.10 Comparison of life-cycle models described in this chapter and the section in which each is defined.

CHAPTER REVIEW

A number of different life-cycle models are described, including the build-and-fix model (Section 3.1), waterfall model (Section 3.2), rapid prototyping model (Section 3.3), incremental model (Section 3.4), extreme programming (Section 3.5), synchronize-and-stabilize model (Section 3.6), spiral model (Section 3.7), and object-oriented life-cycle models (Section 3.8). In Section 3.9, these life-cycle models are compared and contrasted, and suggestions are made regarding choice of life-cycle model for a specific project.

FOR FURTHER READING

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

For an introduction to rapid prototyping, two suggested books are [Connell and Shafer, 1989] and [Gane, 1989]. The role of computer-aided prototyping is assessed in [Luqi and Royce, 1992]. The February 1995 issue of *IEEE Computer* contains several articles on rapid prototyping.

A description of the evolutionary delivery model, one version of the incremental model, can be found in [Gilb, 1988]. The concurrent incremental model is described in [Aoyama, 1993]. The synchronize-and-stabilize model is outlined in [Cusumano and Selby, 1997] and described in detail in [Cusumano and Selby, 1995]. Insights into the synchronize-and-stabilize model can be obtained from [McConnell, 1996]. The spiral model is explained in [Boehm, 1988], and its application to the TRW Software Productivity System appears in [Boehm et al., 1984]. Extreme programming is described in [Beck, 1999] and [Beck, 2000]; refactoring is the subject of [Fowler et al., 1999].

Risk analysis is described in [Boehm, 1991; Jones, 1994c; Karolak, 1996; and Keil, Cule, Lyytinen, and Schmidt, 1998]. The May/June 1997 issue of *IEEE Software* contains 10 articles on risk management.

Object-oriented life-cycle models are described in [Henderson-Sellers and Edwards, 1990; Rajlich, 1994; and Jacobson, Booch, and Rumbaugh, 1999].

Many other life-cycle models have been put forward. For example, a life-cycle model that emphasizes human factors is presented in [Mantei and Teorey, 1988], and [Rajlich and Bennett, 2000] describes a maintenance-oriented life-cycle model. A life-cycle model recommended by the Software Engineering Laboratory is described in [Landis et al., 1992]. The July/August 2000 issue of *IEEE Software* has a variety of papers on software life-cycle models, including [Williams, Kessler, Cunningham, and Jeffries, 2000], which describes an experiment on pair programming, one component of extreme programming. The proceedings of the International Software Process Workshops are a useful source of information on life-cycle models. [ISO/IEC 12207, 1995] is a standard for software life-cycle processes.

PROBLEMS

- 3.1 Suppose that you have to build a product to determine the inverse of 3.748571 to four decimal places. Once the product has been implemented and tested, it will be thrown away. Which life-cycle model would you use? Give reasons for your answer.
- 3.2 You are a software engineering consultant and have been called in by the vice-president for finance of Deplorably Decadent Desserts, a corporation that manufactures and sells a variety of desserts to restaurants. She wants your organization to build a product that will monitor the company's product, starting with the purchasing of the various ingredients and keeping track of the desserts as they are manufactured and distributed to the various restaurants. What criteria would you use in selecting a life-cycle model for the project?
- 3.3 List the risks involved in developing the software of Problem 3.2. How would you attempt to resolve each risk?
- 3.4 Your development of the stock control product for Deplorably Decadent Desserts is highly successful. As a result, Deplorably Decadent Desserts wants the product to be rewritten as a COTS package to be sold to a variety of different organizations that prepare and sell food to restaurants as well as to retail organizations. The new product therefore must be portable and easily adapted to new hardware and operating systems. How do the criteria you would use in selecting a life-cycle model for this project differ from those in your answer to Problem 3.2?
- 3.5 Describe the sort of product that would be an ideal application for the incremental model.
- 3.6 Now describe the type of situation where the incremental model might lead to difficulties.
- 3.7 Describe the sort of product that would be an ideal application for the spiral model.
- 3.8 Now describe the type of situation where the spiral model is inappropriate.
- 3.9 What do waterfalls and fountains have in common? What do the waterfall model and fountain model have in common? How do they differ?
- 3.10 (Term Project) Which software life-cycle model would you use for the Broadlands Area Children's Hospital project described in Appendix A? Give reasons for your answer.
- 3.11 (Readings in Software Engineering) Your instructor will distribute copies of [Beck, 1999]. Would you like to work in an organization that uses extreme programming?

REFERENCES

- [Aoyama, 1993] M. AOYAMA, "Concurrent-Development Process Model," *IEEE Computer* 10 (July 1993), pp. 46-55.
- [Beck, 1999] K. BECK, "Embracing Change with Extreme Programming," *IEEE Computer* 32 (October 1999), pp. 70-77.

- [Beck, 2000] K. BECK, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Longman, Reading, MA, 2000.
- [Berard, 1993] E. V. BERARD, *Essays on Object-Oriented Software Engineering*, Volume 1, Prentice Hall, Englewood Cliffs, NJ, 1993.
- [Boehm, 1988] B. W. BOEHM, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* 21 (May 1988), pp. 61–72.
- [Boehm, 1991] B. W. BOEHM, "Software Risk Management: Principles and Practices," *IEEE Software* 8 (January 1991), pp. 32–41.
- [Boehm et al., 1984] B. W. BOEHM, M. H. PENEDO, E. D. STUCKLE, R. D. WILLIAMS, AND A. B. PYSTER, "A Software Development Environment for Improving Productivity," *IEEE Computer* 17 (June 1984), pp. 30–44.
- [Booch, 1994] G. BOOCH, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.
- [Coad and Yourdon, 1991a] P. COAD AND E. YOURDON, *Object-Oriented Analysis*, 2nd ed., Yourdon Press, Englewood Cliffs, NJ, 1991.
- [Connell and Shafer, 1989] J. L. CONNELL AND L. SHAFER, *Structured Rapid Prototyping: An Evolutionary Approach to Software Development*, Yourdon Press, Englewood Cliffs, NJ, 1989.
- [Cusumano and Selby, 1995] M. A. CUSUMANO AND R. W. SELBY, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press/Simon and Schuster, New York, 1995.
- [Cusumano and Selby, 1997] M. A. CUSUMANO AND R. W. SELBY, "How Microsoft Builds Software," *Communications of the ACM* 40 (June 1997), pp. 53–61.
- [Fowler et al., 1999] M. FOWLER WITH K. BECK, J. BRANT, W. OPDYKE, AND D. ROBERTS, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading, MA, 1999.
- [Gane, 1989] C. GANE, *Rapid System Development: Using Structured Techniques and Relational Technology*, Prentice Hall, Englewood Cliffs, NJ, 1989.
- [Gilb, 1988] T. GILB, *Principles of Software Engineering Management*, Addison-Wesley, Wokingham, U.K., 1988.
- [Henderson-Sellers and Edwards, 1990] B. HENDERSON-SELLERS AND J. M. EDWARDS, "The Object-Oriented Systems Life Cycle," *Communications of the ACM* 33 (September 1990), pp. 142–59.
- [Honiden, Kotaka, and Kishimoto, 1993] S. HONIDEN, N. KOTAKA, AND Y. KISHIMOTO, "Formalizing Specification Modeling in OOA," *IEEE Software* 10 (January 1993), pp. 54–66.
- [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, 1994c] C. JONES, *Assessment and Control of Computer Risks*, Prentice Hall, Englewood Cliffs, NJ, 1994.
- [Karolak, 1996] D. W. KAROLAK, *Software Engineering Risk Management*, IEEE Computer Society, Los Alamitos, CA, 1996.
- [Keil, Cule, Lyytinen, and Schmidt, 1998] M. KEIL, P. E. CULE, K. LYYTINEN, AND R. C. SCHMIDT, "A Framework for Identifying Software Project Risks," *Communications of the ACM* 41 (November 1998), pp. 76–83.
- [Landis et al., 1992] L. LANDis, S. WALIGARA, F. MCGARRY, ET AL., "Recommended Approach to Software Development: Revision 3," Technical Report SEL-81-305, Software Engineering Laboratory, Greenbelt, MD, June 1992.

- [Luqi and Royce, 1992] LUQI AND W. ROYCE, "Status Report: Computer-Aided Prototyping," *IEEE Software* 9 (November 1992), pp. 77-81.
- [Mantei and Teorey, 1988] M. M. MANTEI AND T. J. TEOREY, "Cost/Benefit Analysis for Incorporating Human Factors in the Software Development Lifecycle," *Communications of the ACM* 31 (April 1988), pp. 428-39.
- [McConnell, 1996] S. McCONNELL, "Daily Build and Smoke Test," *IEEE Computer* 13 (July 1996), pp. 144, 143.
- [Rajlich, 1994] V. RAJLICH, "Decomposition/Generalization Methodology for Object-Oriented Programming," *Journal of Systems and Software* 24 (February 1994), pp. 181-86.
- [Rajlich and Bennett, 2000] V. RAJLICH AND K. H. BENNETT, "A Staged Model for the Software Life Cycle," *IEEE Computer* 33 (July 2000), pp. 66-71.
- [Royce, 1970] W. W. ROYCE, "Managing the Development of Large Software Systems: Concepts and Techniques," *1970 WESCON Technical Papers, Western Electronic Show and Convention*, Los Angeles, August 1970, pp. A/1-1-A/1-9. Reprinted in *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 328-38.
- [Royce, 1998] W. ROYCE, *Software Project Management: A Unified Framework*, Addison-Wesley, Reading, MA, 1998.
- [Spivey, 1992] J. M. SPIVEY, *The Z Notation: A Reference Manual*, Prentice Hall, New York, 1992.
- [Williams, Kessler, Cunningham, and Jeffries, 2000] L. WILLIAMS, R. R. KESSLER, W. CUNNINGHAM, AND R. JEFFRIES, "Strengthening the Case for Pair Programming," *IEEE Software* 17 (July/August 2000), pp. 19-25.

chapter

10

REQUIREMENTS PHASE

The chances of a product being developed on time and within budget are somewhat slim unless the members of the software development team agree on what the software product will do. The first step in achieving this unanimity is to analyze the client's current situation as precisely as possible. For example, it is inadequate to say, "They need a computer-aided design system because they claim their manual design system is lousy." Unless the development team knows exactly what is wrong with the current manual system, there is a high probability that aspects of the new computerized system will be equally "lousy." Similarly, if a personal computer manufacturer is contemplating development of a new operating system, the first step is to evaluate the firm's current operating system and analyze carefully exactly why it is unsatisfactory. To take an extreme example, it is vital to know whether the problem exists only in the mind of the sales manager, who blames the operating system for poor sales, or whether users of the operating system are thoroughly disenchanted with its functionality and reliability. Only after a clear picture of the present situation has been gained can the team attempt to answer the critical question, What must the new product be able to do? The process of answering this question is carried out during the requirements phase.

A commonly held misconception is that, during the requirements phase, the developers must determine what software the client *wants*. On the contrary, the real objective of the requirements phase is to determine what software the client *needs*. The problem is that many clients do not know what they need. Furthermore, even a client who has a good idea of what is needed may have difficulty in accurately conveying these ideas to the developers, because most clients are less computer literate than the members of the development team.

In 1967, U. S. presidential candidate George Romney put his foot into his mouth one time too many. Calling a press conference, he proceeded to announce: "I know you believe you understood what you think I said, but I am not sure you realize that what you heard is not what I meant." This excuse applies equally well to the issue of requirements analysis. The developers hear their client's requests, but what they hear is not what the client should be saying.

Chapter 3 pointed out that one way of solving this communication-based problem is to build a rapid prototype. In this chapter, the requirements phase is described in greater detail, and the strengths and weaknesses of various requirements analysis techniques are described.

We begin by looking at requirements elicitation.

10.1 REQUIREMENTS ELICITATION

The process of discovering the client's requirements is termed *requirements elicitation* (or *requirements capture*). Once an initial set of requirements has been determined, they are refined and extended; this process is termed *requirements analysis*. The requirements phase usually begins with one or more members of the requirements team meeting with one or more members of the client organization to determine what is needed in the target product.

To elicit the client's needs, the members of the requirements team must be familiar with the application domain, that is, the general area in which the proposed software product is to be used. For example, it is not easy to ask meaningful questions of a banker or a nurse without first acquiring some familiarity with banking or nursing. Therefore, one of the initial tasks of each member of the requirements analysis team is to acquire familiarity with the application domain unless he or she already has experience in that general area. It is particularly important to use correct terminology when communicating with the client and potential users of the target software. After all, it is hard to be taken seriously by a person working in a specific domain unless the interviewer uses the nomenclature appropriate for that domain. More important, use of an inappropriate word may lead to a misunderstanding, eventually resulting in a faulty product being delivered. The same problem can arise if the members of the requirements team do not understand the subtleties of the terminology of the domain. For example, to a layperson words like *brace*, *beam*, *girder*, and *strut* may appear synonyms, but to a civil engineer they are distinct terms. If a developer does not appreciate that a civil engineer is using these four terms in a precise way and if the civil engineer assumes that the developer is familiar with the distinctions between the terms, the developer may treat the four terms as equivalent; the resulting computer-aided bridge design software may contain faults that result in a bridge collapsing. Computer professionals hope that the output of every program will be scrutinized carefully by a human being before decisions are made based on that program, but the growing popular faith in computers means that it is distinctly unwise to rely on the likelihood of such a check being made. So, it is no means far-fetched that a misunderstanding in terminology could lead to the software developers being sued for negligence.

One way to solve the problem with terminology is to build a glossary. The initial entries are inserted while the team learns the application domain. Then the glossary is updated whenever the members of the requirements team encounter new terminology. Not only does such a glossary reduce confusion between client and developers, it also is useful in lessening misunderstandings between members of the development team.

Once the requirements team have acquired familiarity with the domain, the next step is for them to start to determine the client's needs, that is, requirements elicitation. The primary elicitation technique is interviewing.

10.1.1 INTERVIEWS

The members of the requirements team meet with members of the client organization until they are convinced that they have elicited all relevant information from the client

and future users of the product. There are two basic types of interview, structured and unstructured. In a structured interview, specific, preplanned, close-ended questions are posed. For example, the client might be asked how many salespeople the company employs or how fast a response time is required. In an unstructured interview, open-ended questions are asked, to encourage the person being interviewed to speak out. For instance, asking the client, "Why is the current product unsatisfactory?" may explain many aspects of the client's approach to business. Some of these facts might not have come to light had the interview been more structured. At the same time, it is not a good idea if the interview is too unstructured. Saying to the client, "Tell me about the current product" is unlikely to yield much pertinent information. Therefore, questions should be posed in such a way as to encourage the person being interviewed to give wide-ranging answers but within the context of the information needed by the interviewer.

Conducting a good interview is not always easy. First, the interviewer must be familiar with the application domain. Second, there is no point in interviewing a member of the client organization if the interviewer already has made up his or her mind regarding the client's needs. No matter what he or she previously has been told or learned by other means, the interviewer must approach every interview with the intention of listening carefully to what the person being interviewed has to say while firmly suppressing any preconceived notions regarding the client company or the needs of the clients and potential uses of the software product to be built.

After the interview is concluded, the interviewer must prepare a written report outlining the results of the interview. It is strongly advised to give a copy of the report to the person interviewed; he or she may want to clarify statements or add overlooked items.

10.1.2 SCENARIOS

Scenarios are another technique for requirements analysis. A scenario is a way a user might utilize the target product to accomplish some objective. For example, suppose the target product is a weight-loss planner. One possible scenario describes what happens when the dietitian enters the age, gender, weight, height, and other personal data of a patient. The product then prints out sample menus for that patient. When this scenario is shown to a future user of the target product, the dietitian quickly points out that the menus would be unsuitable for a patient with special food requirements, such as a diabetic, a vegetarian, or someone who is lactose intolerant. The developers modify the scenario so that the user is asked about special dietary needs before any menus are printed. The use of scenarios enables users to communicate their needs to the requirements analysts.

Taking the example of the weight-loss planner further, suppose that the program solicits the height of the patient in inches but the dietitian enters the height in centimeters. This is an example of an *exception*. A scenario should include not just the expected sequence of events but also all exceptions.

A scenario can be depicted in a number of ways. One technique is simply to list the actions comprising the scenario; this is done in Chapter 11. Another technique is to set

up a storyboard, a series of diagrams depicting the sequence of events. A storyboard can be considered a paper prototype [Rettig, 1994], that is, a series of sheets of paper each depicting the relevant screens and the user's response. But, whatever method is chosen, the scenario should depict the starting state, the expected sequence of events, and the finishing state, together with the exceptions to the expected sequence.

Scenarios are useful in a number of different ways.

1. They can demonstrate the behavior of the product in a way that is comprehensible to the user. This can result in additional requirements coming to light, as in the weight-loss planner example.
2. Because scenarios can be understood by users, the utilization of scenarios can ensure that the client and users play an active role throughout the requirements analysis process. After all, the aim of the requirements analysis phase is to elicit the real needs of the client, and the only source of this information is the client and the users.
3. Scenarios (or more precisely, use cases) play an important role in object-oriented analysis. This is discussed in detail in Section 12.3.

We now consider other techniques for eliciting requirements.

10.1.3 OTHER REQUIREMENTS ELICITATION TECHNIQUES

Yet another way of eliciting needs is to send a questionnaire to the relevant members of the client organization. This technique is useful when the opinions of, say, hundreds of individuals need to be determined. Furthermore, a carefully thought-out written answer may be more accurate than an immediate verbal response to a question posed by an interviewer. However, an unstructured interview conducted by a methodical interviewer who listens carefully and poses questions that expand on initial responses usually yields far better information than a thoughtfully worded questionnaire. Because questionnaires are preplanned, there is no way that a question can be posed in response to an answer.

A different way of eliciting requirements, particularly in a business environment, is to examine the various forms used by the client. For example, a form in a print shop might reflect press number, paper roll size, humidity, ink temperature, paper tension, and so on. The various fields in this form shed light on the flow of print jobs and the relative importance of the steps in the printing process. Other documents, such as operating procedures and job descriptions, also can be powerful tools for finding out exactly what is done and how. Such comprehensive information regarding how the client currently does business can be extraordinarily helpful in determining the client's needs. Therefore, careful perusal of client documentation should never be overlooked as a source of information that can lead to an accurate assessment of the client's needs.

A newer way of obtaining such information is to set up videotape cameras within the workplace to record (with the prior written permission of those being observed) exactly what is being done. One difficulty of this technique is that it can take a long time to analyze the tapes. In general, one or more members of the requirements analysis

team has to spend an hour playing back the tape for every hour that the cameras record. This time is in addition to what is needed to assess what was observed. More seriously, this technique has been known to backfire badly because employees may view the cameras as an unwarranted invasion of privacy. It is important that the requirements analysis team have the full cooperation of all employees; it can be extremely difficult to obtain the necessary information if people feel threatened or harassed. The possible risks should be considered carefully before introducing cameras or, for that matter, taking any other action that has the potential to anger employees.

Once an initial set of requirements has been elicited, the next step is to refine them, a process called *requirements analysis*.

10.2 REQUIREMENTS ANALYSIS

At this stage in the process, the requirements team has a preliminary set of requirements. These requirements are of two types, functional and nonfunctional. *Functional* requirements relate to the functionality of the target software; for example, "Royalties for each artist shall be computed from the playlist data using the May 1998 CMS formula." *Nonfunctional* specifications specify properties of the target software, such as reliability and maintainability, or relate to the environment in which the software must run; for example, "All bar codes shall be read using the Mach/Zor ASRCA input device."

It is essential that the software be *traceable*; that is, it must be possible to trace each statement in the requirements document through the specifications, design, and code. In this way, the SQA group can check that every statement in the requirements has been implemented and that this has been done correctly. To achieve traceability, each statement in the requirements document needs to be numbered.

All the items in the preliminary requirements document are given to the client to get their priorities. The client (or a client team) ranks each preliminary requirement using categories such as essential, highly desirable, desirable, and so on. During the course of this process, it may become apparent that certain requirements are incorrect or irrelevant. Any such requirements are corrected or deleted.

The next step is to further refine the preliminary requirements document. First, the members of the requirements team discuss the list of requirements with the various individuals interviewed to determine if anything has been omitted. Then, because the most accurate and powerful requirements analysis technique is rapid prototyping, a rapid prototype is built. This is described in the next section.

10.3 RAPID PROTOTYPING

A rapid prototype is hastily built software that exhibits the key functionality of the target product. For example, a product that helps manage an apartment complex must

incorporate an input screen that allows the user to enter details of a new tenant and print an occupancy report for each month. These aspects are incorporated into the rapid prototype. However, error-checking capabilities, file-updating routines, and complex tax computations probably are not included. The key point is that a rapid prototype reflects the functionality that the client sees, such as input screens and reports, but omits "hidden" aspects such as file updating. (For a different way of looking at rapid prototypes, see the Just in Case You Wanted to Know box below.)

The client and intended users of the product now experiment with the rapid prototype, while members of the development team watch and take notes. Based on their hands-on experience, users tell the developers how the rapid prototype satisfies their needs and, more important, identify the areas that need improvement. The developers change the rapid prototype until both sides are convinced that the needs of the client are accurately encapsulated in the rapid prototype. The rapid prototype then is used as the basis for drawing up the specifications.

An important aspect of the rapid prototyping model is embodied in the word *rapid*. The whole idea is to build the prototype as quickly as possible. After all, the purpose of the rapid prototype is to provide the client with an understanding of the product, and the sooner, the better. It does not matter if the rapid prototype hardly works, if it crashes every few minutes, or if the screen layouts are less than perfect. The purpose of the rapid prototype is to enable the client and the developers to agree as quickly as possible on what the product is to do. Therefore, any imperfections in the rapid prototype may be ignored, provided they do not seriously impair the functionality of the rapid prototype and thereby give a misleading impression of how the product will behave.

JUST IN CASE YOU WANTED TO KNOW

The idea of constructing models to show key aspects of a product goes back a long time. For example, a 1618 painting by Domenico Cresti (known as "Il Passignano" because he was born in the town of Passignano in the Chianti region of Italy) shows Michelangelo presenting a wooden model of his design for St. Peter's (in Rome) to Pope Paul IV. Such architectural models could be huge; a model of an earlier design proposal for St. Peter's by the architect Bramante is more than 20 feet long on each side.

Architectural models were used for a number of different purposes. First, as depicted in the Cresti painting (now hanging in Casa Buonarroti in Florence), models were used to try to interest a client in funding a project. This is analogous to the use of a rapid

prototype to determine the client's real needs. Second, in an age before architectural drawings, the model showed the builder the structure of the building and indicated to the stone masons how the building was to be decorated. This is similar to the way we now build a rapid prototype of the user interface, as described in Section 10.4.

It is not a good idea, however, to draw too close a parallel between such architectural models and software rapid prototypes. Rapid prototypes are used during the requirements phase to elicit the client's needs. Unlike architectural models, they are not used to represent either the architectural design or the detailed design; the design is produced two phases later, that is, during the design phase.

A second major aspect of the rapid prototyping model is that the rapid prototype must be built for change. If the first version of the rapid prototype is not what the client needs, then the prototype must be transformed rapidly into a second version that, it is hoped, better satisfies the client's requirements. To achieve rapid development throughout the rapid prototyping process, fourth-generation languages (4GLs) and interpreted languages, such as Smalltalk, Prolog, and Lisp, have been used. Popular rapid prototyping languages of today include HTML and Perl, as well as visual C++ and J++. Concerns have been expressed about the maintainability of certain interpreted languages, but from the viewpoint of classic rapid prototyping, this is irrelevant. All that counts is, Can a given language be used to produce a rapid prototype? And can the rapid prototype be changed quickly? If the answer to both questions is yes, then that language probably is a good candidate for rapid prototyping.

Turning now to the use of rapid prototyping in conjunction with the object-oriented paradigm, three very different object-oriented projects carried out by IBM showed significant improvements compared to projects using the structured paradigm [Capper, Colgate, Hunter, and James, 1994]. One of the recommendations that resulted from these projects is that it is important to build a rapid prototype as early as possible in the object-oriented life cycle.

Rapid prototyping also is particularly effective when developing the user interface to a product. This use is discussed in the next section.

10.4 HUMAN FACTORS

It is important that both the client and the future users of the product interact with the rapid prototype of the user interface. Encouraging users to experiment with the human-computer interface (HCI) greatly reduces the risk that the finished product will have to be altered. In particular, this experimentation helps achieve user-friendliness, a vital objective for all software products.

The term *user-friendliness* refers to the ease with which human beings can communicate with the software product. If users have difficulty learning how to use a product or find the screens confusing or irritating, then they will either not use the product or use it incorrectly. To try to eliminate this problem, menu-driven products were introduced. Instead of having to enter a command such as Perform computation or Print service rate report, the user merely has to select from a set of possible responses, such as

1. Perform computation
2. Print service rate report
3. Select view to be graphed

In this example, the user enters 1, 2, or 3 to invoke the corresponding command.

Nowadays, instead of simply displaying lines of text, HCIs employ graphics. Windows, icons, and pull-down menus are components of a graphical user interface

(GUI). Because of the plethora of windowing systems, standards such as X Window have evolved. Also, "point and click" selection is becoming the norm. The user moves a mouse (that is, a handheld pointing device) to move the screen cursor to the desired response ("point") and pushes a mouse button ("click") to select that response.

However, even when the target product employs modern technology, the designers must never forget that the product is to be used by human beings. In other words, the HCI designers must consider *human factors* such as size of letters, capitalization, color, line length, and the number of lines on the screen.

Another example of human factors applies to the preceding menu. If the user chooses option 3. Select view to be graphed, then another menu appears with another list of choices. Unless a menu-driven system is thoughtfully designed, there is the danger that the users will encounter a lengthy sequence of menus to achieve even a relatively simple operation. This delay can anger users, sometimes causing them to make inappropriate menu selections. Also, the HCI must allow the user to change a previous selection without having to return to the top-level menu and start again. This problem can exist even when a GUI is used because many graphical user interfaces essentially are a series of menus displayed in an attractive screen format.

Sometimes, a single user interface cannot cater to all users. For example, if a product is to be used by both computer professionals and high-school dropouts with no previous computer experience, then it is preferable that two different sets of HCIs be designed, each carefully tailored to the skill level and psychological profile of its intended users. This technique can be extended by incorporating sets of user interfaces requiring varied levels of sophistication. If the product deduces that the user would be more comfortable with a less-sophisticated user interface, perhaps because the user is making frequent mistakes or is continually invoking help facilities, then the user automatically is shown screens more appropriate to his or her current skill level. But, as the user becomes more familiar with the product, streamlined screens that provide less information are displayed, leading to speedier completion. This automated approach reduces user frustration and leads to increased productivity [Schach and Wood, 1986].

Many benefits can accrue when human factors are taken into account during the design of an HCI, including reduced learning time and lower error rates. Although help facilities always must be provided, they are utilized less with a carefully designed HCI. This, too, increases productivity. Uniformity of HCI appearance across a product or group of products can result in users intuitively knowing how to use a screen they have never seen before because it is similar to other screens with which they are familiar. Designers of Macintosh software have taken this principle into account; this is one of the many reasons that software for the Macintosh generally is so user-friendly.

It has been suggested that simple common sense is all that is needed to design a user-friendly HCI. Whether or not this charge is true, it is essential that a rapid prototype of the HCI of every product be constructed. Intended users of the product can experiment with the rapid prototype of the HCI and inform the designers whether the target product indeed will be user-friendly, that is, whether the designers have taken the necessary human factors into account.

In the next two sections, superficially attractive but dangerous variants of the rapid prototyping model are discussed.

10.5 RAPID PROTOTYPING AS A SPECIFICATION TECHNIQUE

The conventional form of the rapid prototyping model, as described in Section 3.3 and depicted in Figure 3.3, is reproduced here as Figure 10.1. (Again, the implementation and integration steps generally are performed in parallel.) The rapid prototype is used

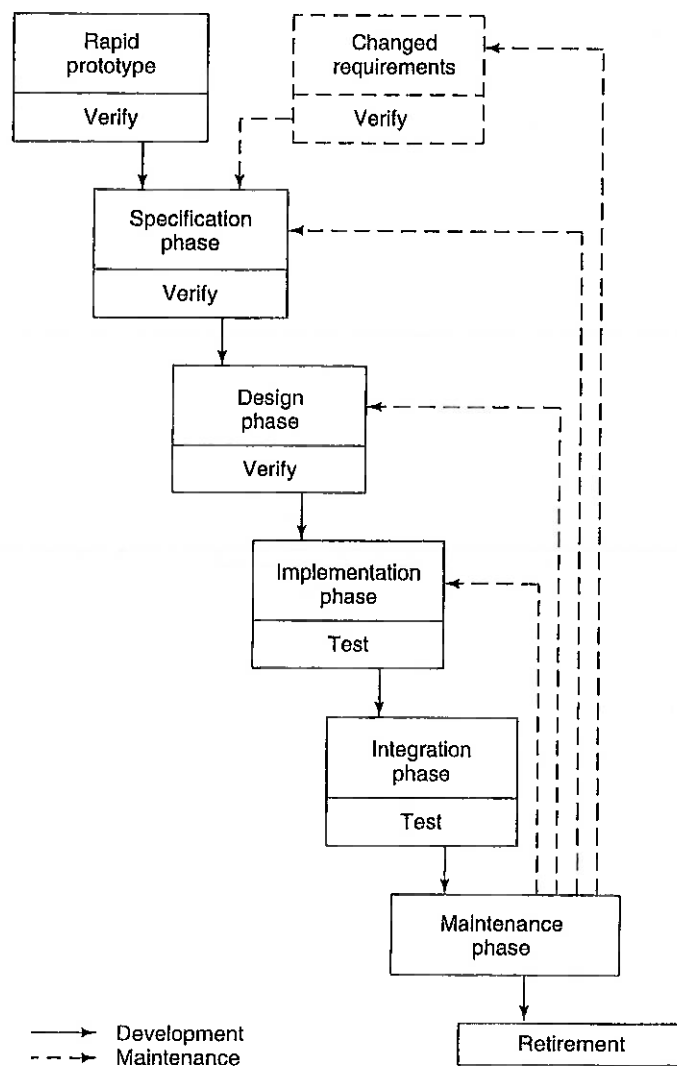


Figure 10.1 Rapid prototyping model.

This version of the rapid prototyping model can have a major drawback. If there is a disagreement as to whether the developers have satisfactorily discharged their obligations, it is unlikely that a rapid prototype will stand as a legal statement of a contract between developer and client. For this reason, the rapid prototype should never be used as the sole specification, not even if the software is developed internally (that is, when the client and developers are members of the same organization). Although it is unlikely that the head of the investment management division of a bank will take the data processing division to court, disagreements between client and developers nevertheless can arise just as easily within an organization. Therefore, to protect themselves, software developers should not use the rapid prototype as the specifications even when software is developed internally.

A second reason why the rapid prototype should not take the place of written specifications is potential problems with maintenance. As described in Chapter 16, maintenance is challenging, even when all the documentation is available and up to date. If there are no specifications, maintenance rapidly can become a nightmare. The problem is particularly acute in the case of enhancement, where changes in requirements have to be implemented. It can be exceedingly difficult to change the design documents to reflect the new specifications because, in the absence of written specifications, the maintenance team have no clear statement of the current specifications.

For both these reasons, the rapid prototype should be used simply as a requirements analysis technique, that is, a means of ensuring that the client's real needs have been elicited correctly. Thereafter, written specification documents should be produced using the rapid prototype as a basis.

10.6 REUSING THE RAPID PROTOTYPE

In both versions of the rapid prototyping model discussed previously, the rapid prototype is discarded early in the software process. An alternate, but generally unwise, way of proceeding is to develop and refine the rapid prototype until it becomes the product. This is shown in Figure 10.3. In theory, this approach should lead to fast software development; after all, instead of throwing away the code constituting the rapid prototype, along with the knowledge built into it, the rapid prototype is converted into the final product. However, in practice the process is very similar to the build-and-fix approach of Figure 3.1. As with the build-and-fix model, the first problem with this form of the rapid prototyping model is that, in the course of refining the rapid prototype, changes have to be made to a working product. This is an expensive way to proceed, as shown in Figure 1.5. A second problem is that a primary objective when constructing a rapid prototype is speed of building. A rapid prototype (correctly) is put together hurriedly, rather than carefully specified, designed, and implemented. In the absence of specification and design documents, the resulting code is difficult and expensive to maintain. It might seem wasteful to construct a rapid prototype then throw it away and design the product from scratch, but it is far cheaper in both the

