Unit 1

Software and Software Engineering

The nature of software

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware.

Software is information transformer— producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer, The communication of information (networks), and the creation and control of other programs.

Software delivers the most important product of our time—information. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness it provides a gateway to worldwide information networks , and provides the means for acquiring information in all of its forms.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era.

Programmers are the same questions that are asked when modern computer-based systems are built:

Why does it take so long to get software finished?

- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

Defining Software

Software is: (1) instructions that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Software has characteristics that are considerably different than those of hardware

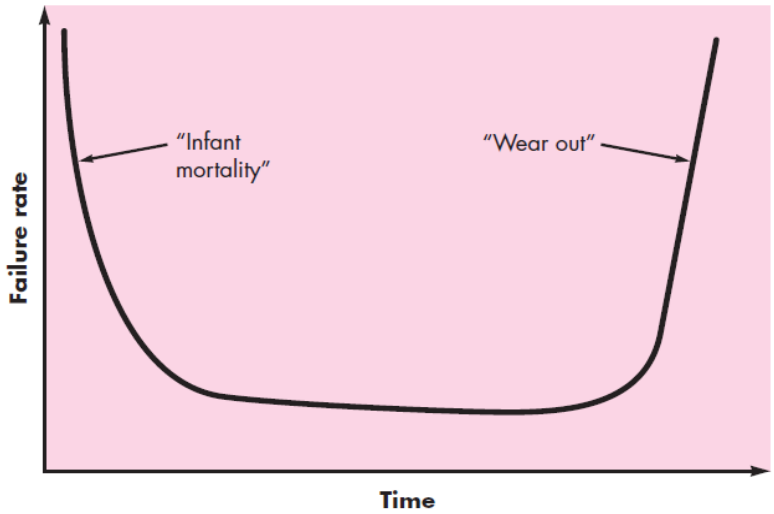
1. Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

   Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

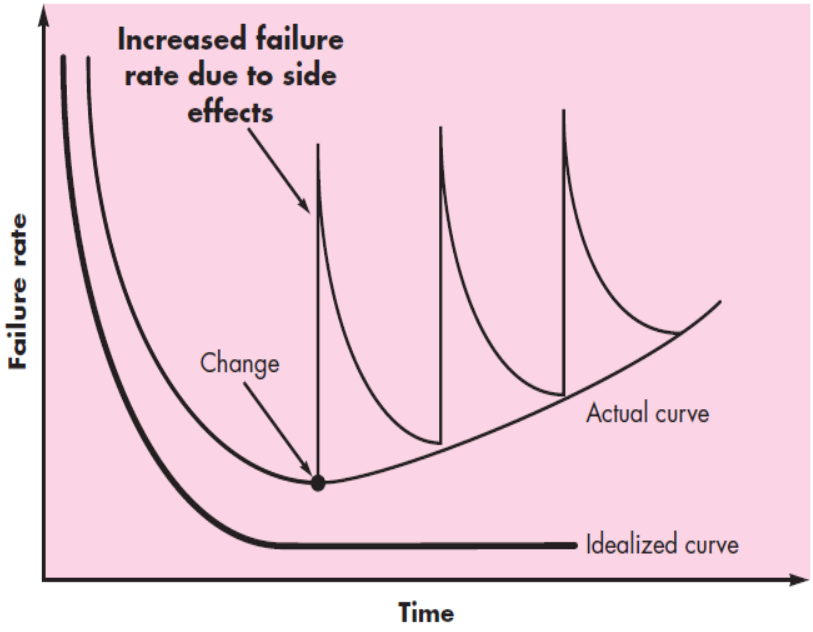
FIGURE 1.1
Failure curve for hardware



In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does Deteriorate.

During its life,2 software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

FIGURE 1.2
Failure curves for software



When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

3. Although the industry is moving toward component-based construction, most software continues to be custom built

As an engineering discipline evolves, a collection of standard design components is create. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent. Something new.In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.

Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers.

System software— a collection of programs written to service other programs. Some system software processes complex, but determinate information structures. Other systems applications process largely indeterminate data.In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time.

Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system

simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions or provide significant function and control capability.

Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets.

Web applications called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software—makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Applications within this area include robotics, expert systems, pattern recognition, artificial neural networks, theorem proving, and game playing.

Legacy Software

These older programs—often referred to as legacy software—have been the focus of continuous attention and concern since the 1960s.

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms.The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—poor quality.Legacy systems sometimes have inextensible designs, convoluted

code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.

If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons.

The software must be adapted to meet the needs of new computing environments or technology.

• The software must be enhanced to implement new business requirements.

• The software must be extended to make it interoperable with other more modern systems or databases.

• The software must be re-architected to make it viable within a network environment.

Software engineering

In order to build software that is ready to meet the challenges of the twenty-first century, must recognize a few simple realities:

Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application8 has grown dramatically. It follows that a concerted effort should be made to understand the problem before a software solution is developed.

The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. It follows that design becomes a pivotal activity.

Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. It follows that software should exhibit high quality.As the perceived value of a specific application grows, the likelihood is that its user base and longevity

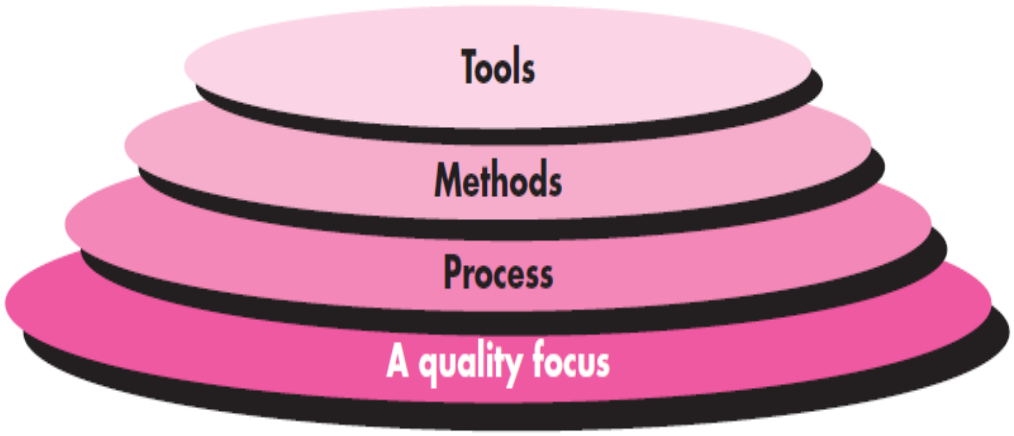
will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow. It follows that software should be maintainable.

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach must rest on an organizational commitment to quality.

FIGURE 1.3

Software engineering layers



The foundation for software engineering is the process layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

The software process

A process is a collection of activities actions, and tasks that are performed when some work product is to be created. An activity strives to achieve a broad objective and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An action (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model). A task focuses on a small, but well-defined objective that produces a tangible outcome.

In the context of software engineering, a process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks.

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities.

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks

that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements**.**

Construction. This activity combines code generation and the testing that is required to uncover errors in the code.

Deployment. The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each project iteration produces a software increment that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse and establishes mechanisms to achieve reusable components.

Software engineering practice

A generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—communication, planning, modeling, construction, and deployment—and umbrella activities establish a skeleton architecture for software engineering work.

The Essence of Practice

George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice.

1. Understand the problem (communication and analysis).
2. Plan a solution (modeling and software design).
3. Carry out the plan (code generation).
4. Examine the result for accuracy (testing and quality assurance)

Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions.

Who has a stake in the solution to the problem? That is, who are the stakeholders?

• What are the unknowns? What data, functions, and features are required to properly solve the problem?

• Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?

• Can the problem be represented graphically? Can an analysis model be created

Plan the solution. Understand the problem and can't wait to begin coding. Before you do, slow down just a bit and do a little design:

• Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?

• **Has a similar problem been solved?** If so, are elements of the solution reusable?

• **Can subproblems be defined**? If so, are solutions readily apparent for the subproblems?

• Can you represent a solution in a manner that leads to effective implementation?

Can a design model be created?

Carry out the plan. The design e created serves as a road map for the system you want to build.

Does the solution conform to the plan? Is source code traceable to the design model? Is each component part of the solution provably correct? Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm.

Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

General Principles

principles at many different levels of abstraction . Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., communication), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario).

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole

The First Principle: The Reason It All Exists

A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes

The Second Principle: KISS (Keep It Simple, Stupid!)

Software design is not a haphazard process. There are many factors to consider in any design effort. All design should be as simple as possible, but no simpler. This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity.

The Third Principle: Maintain the Vision

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two minds" about itself. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: What You Produce, Others Will Consume

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes

The Sixth Principle: Plan Ahead for Reuse

Reuse saves time and effort.15Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

The Seventh principle: Think!

Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out.

SOFTWARE MYTHS

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious.

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Myth: We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: If we get behind schedule, we can add more programmers and catch up

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software project.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations and, ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

Myth: Software requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early, the cost impact is relatively small.

Practitioner's myths: Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects**.**

Myth: The only deliverable work product for a successful project is the working program

A working program is only one part of a software configuration that includes many elements. A variety of work products provide a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Process Models:

The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved.

Software process as a framework for the activities, actions, and tasks that are required to build high-quality software. But software engineering also encompasses technologies that populate the process—technical methods and automated tools.

A generic process model

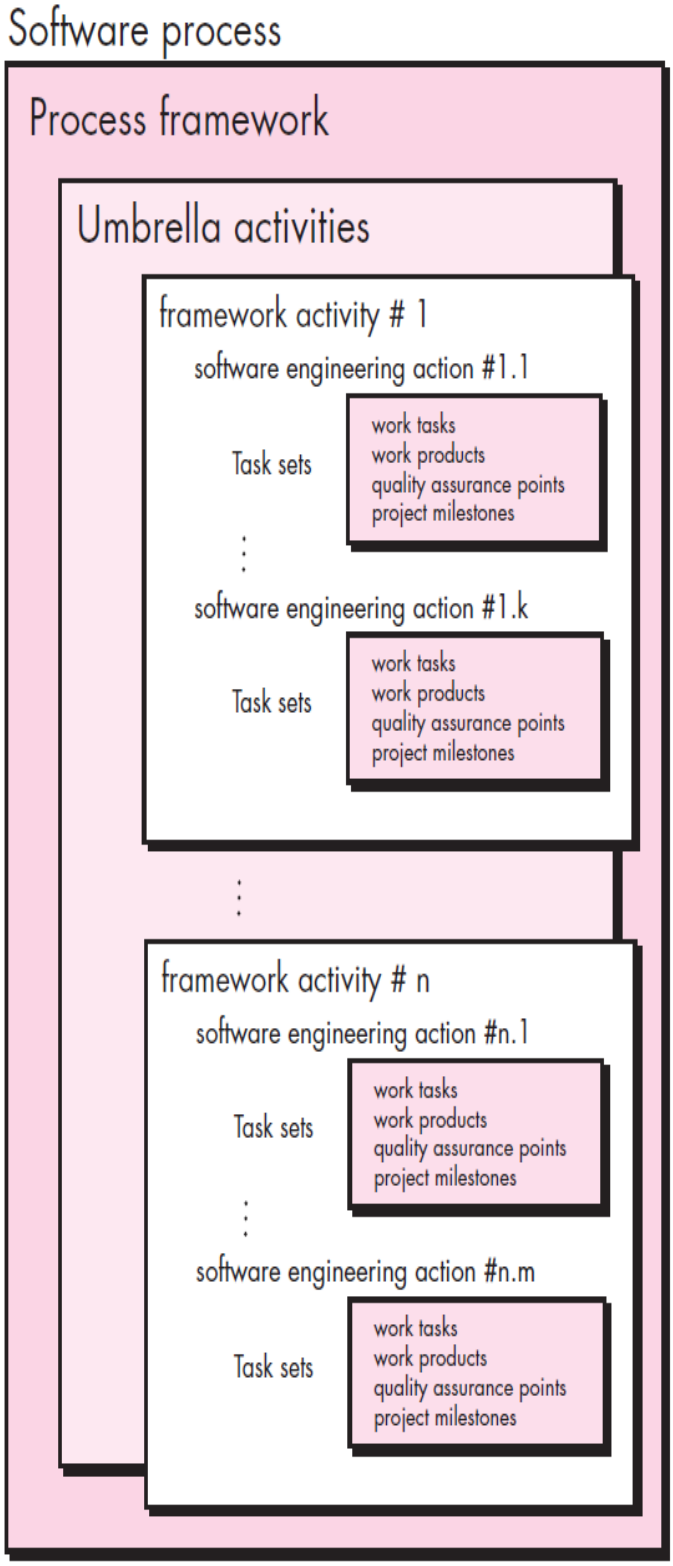
A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 2.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

Generic process framework for software engineering defines five framework activities—communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

FIGURE 2.1

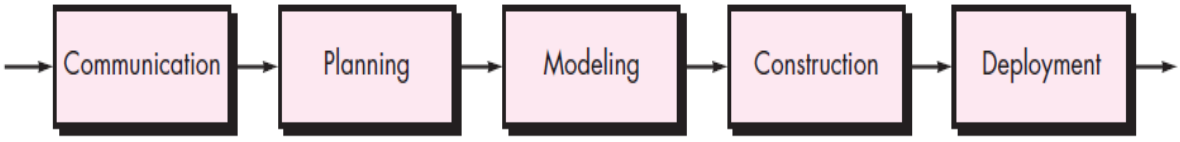
A software process framework



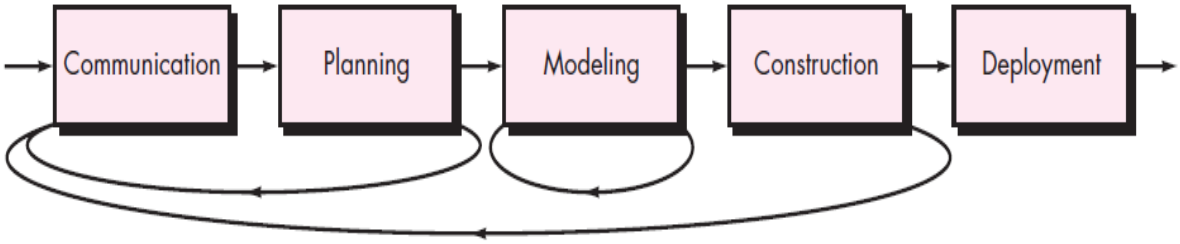
Process flow—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 2.2

A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 2.2a). An iterative process flow repeats one or more of the activities before proceeding to the next (Figure 2.2b). An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software (Figure 2.2c). A parallel process flow (Figure 2.2d) executes one or more activities in parallel with other activities.

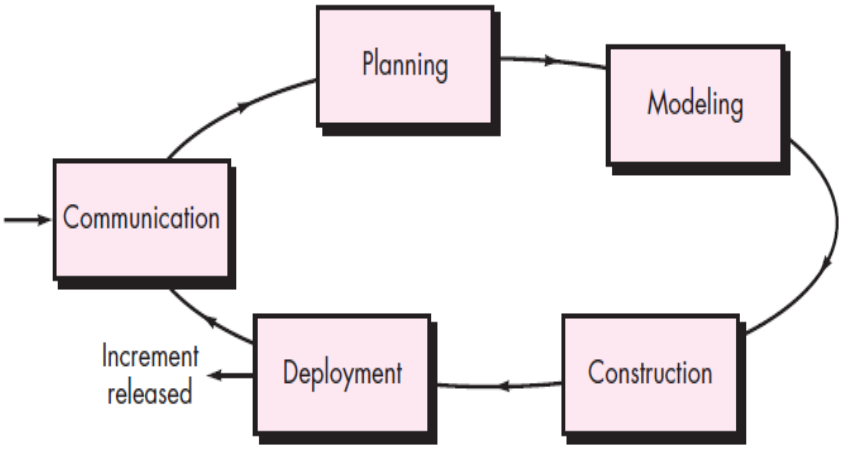
FIGURE 2.2 Process flow



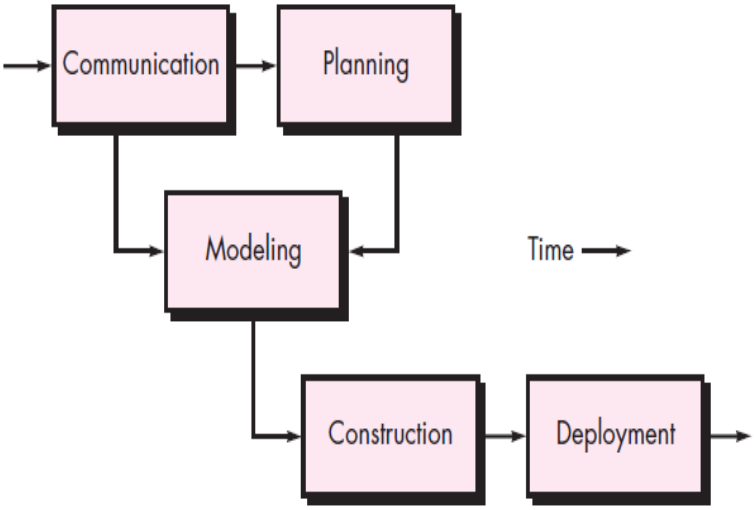
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

Identifying a Task Set

Eeach software engineering action can be represented by a number of different task sets—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

Process Patterns

A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template —a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction.2 In some cases, a pattern might be used to describe a problem associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity or an action within a framework activity.

Ambler [Amb98] has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process.

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type

1. **Stage pattern—**defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage.
2. **Task pattern**—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice.

3. **Phase pattern**—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be Spiral Model or Prototyping.

   Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern). The description is then refined into a set of stage patterns that describe framework activities and are further refined in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

PROCESS ASSESSMENT AND IMPROVEMENT

Process patterns must be coupled with solid software engineering practice. In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement (SCAMPI)—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM-Based Appraisal for Internal Process Improvement (CBAIPI)—provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

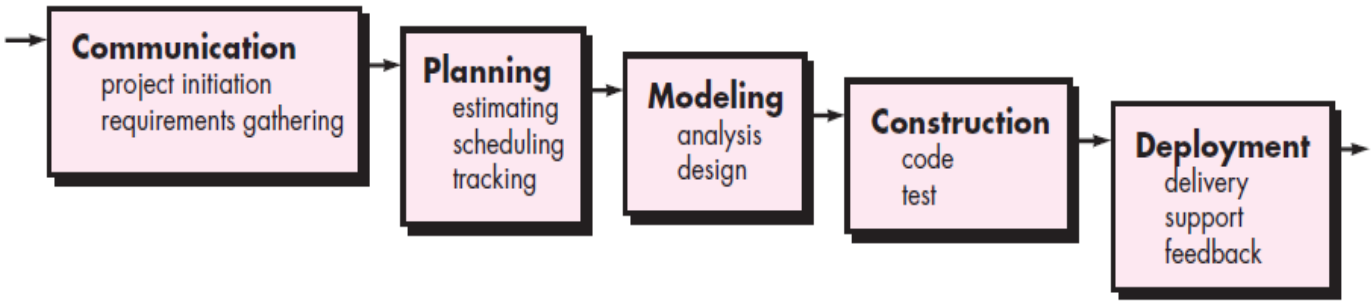
Prescriptive process models

Traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements are interrelated to one another.

The Waterfall Mode

The waterfall model, sometimes called the classic life cycle, suggests a systematic,sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

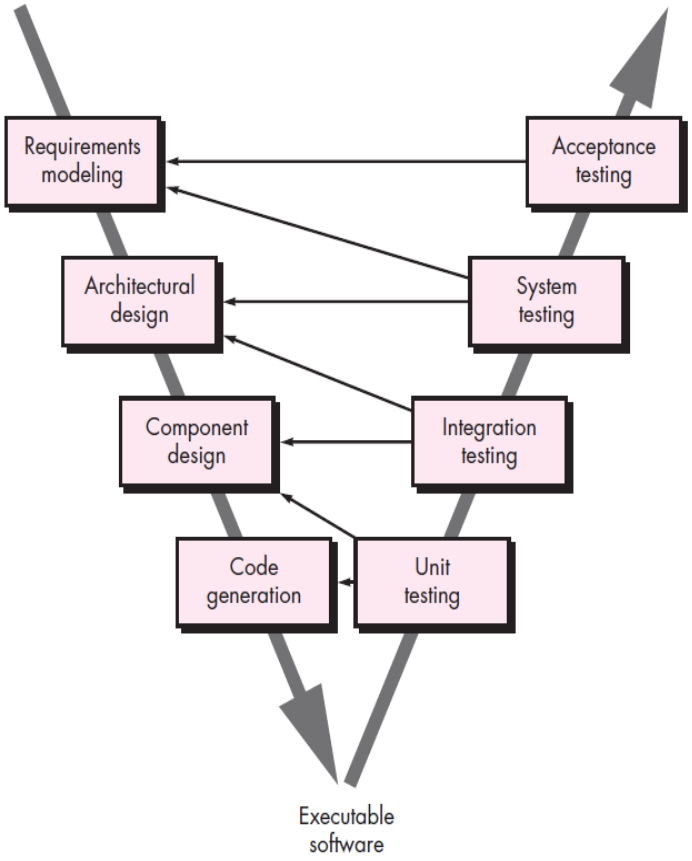
FIGURE 2.3 The waterfall model



A variation in the representation of the waterfall model is called the V-model. Represented in Figure the V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V,basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V,

essentially performing a series of tests that validate each of the models created as the team moved down the left side. In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

FIGURE 2.4
The V-model



The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even ardent supporters to question its efficacy. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span.

Incremental Process Models

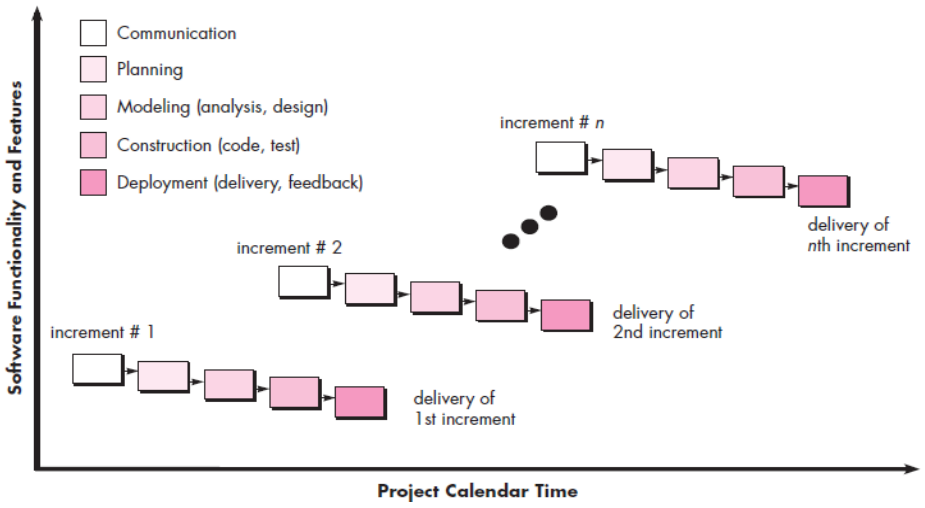
The incremental model combines elements of linear and parallel process flows Referring to figure the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

FIGURE 2.5
The incremental model



Evolutionary Process Models

A set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, need a process model that has been explicitly designed to accommodate a product that evolves over time.

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

Prototyping. Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The prototyping paradigm begins with communication. Meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users . The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.

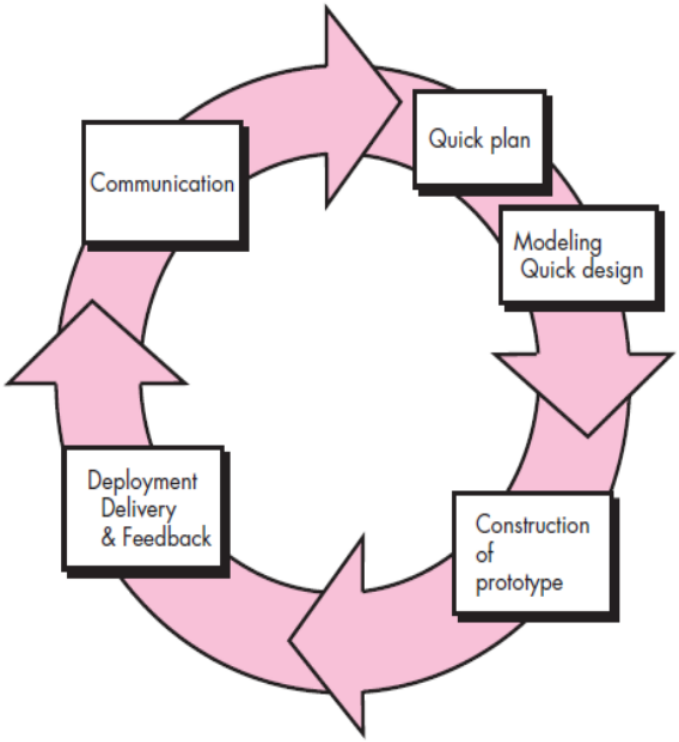
Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Prototyping can be problematic for the following reasons.

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you hasn’t considered overall software quality or long-term maintainability.

2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability.

FIGURE 2.6

The prototyping paradigm



The Spiral Model

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner**:**

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more

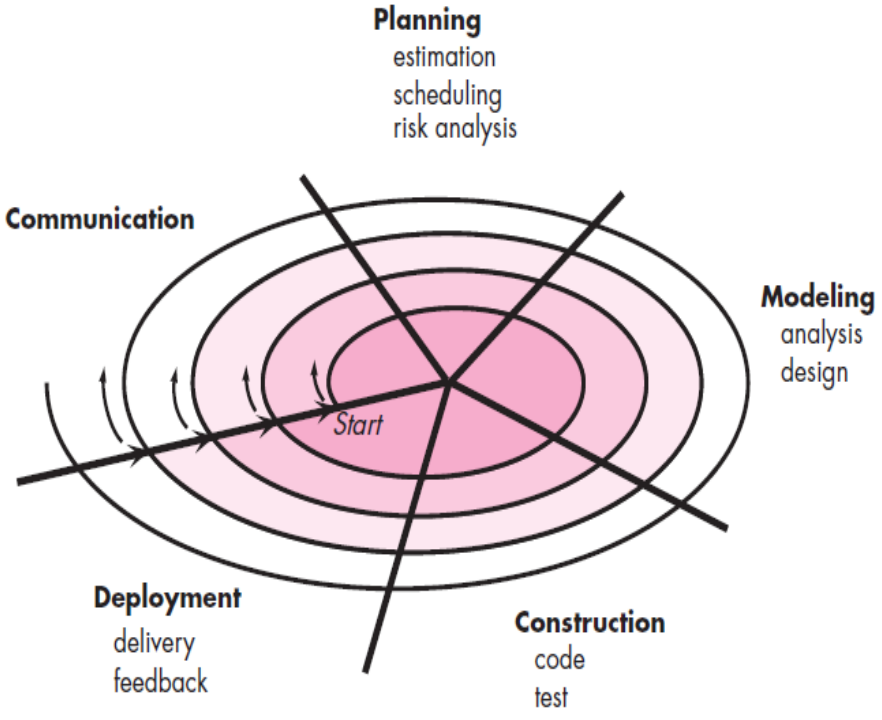
A spiral model is divided into a set of framework activities defined by the software engineering team As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete. If the concept is to be developed into an actual product,

the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point.

FIGURE 2.7
A typical spiral model



Drawbacks of spiral model.

It may be difficult to convince customers that the evolutionary approach is controllable.

It demands considerable risk assessment expertise and relies on this expertise for success.

If a major risk is not uncovered and managed, problems will undoubtedly occur**.**

Concurrent Models

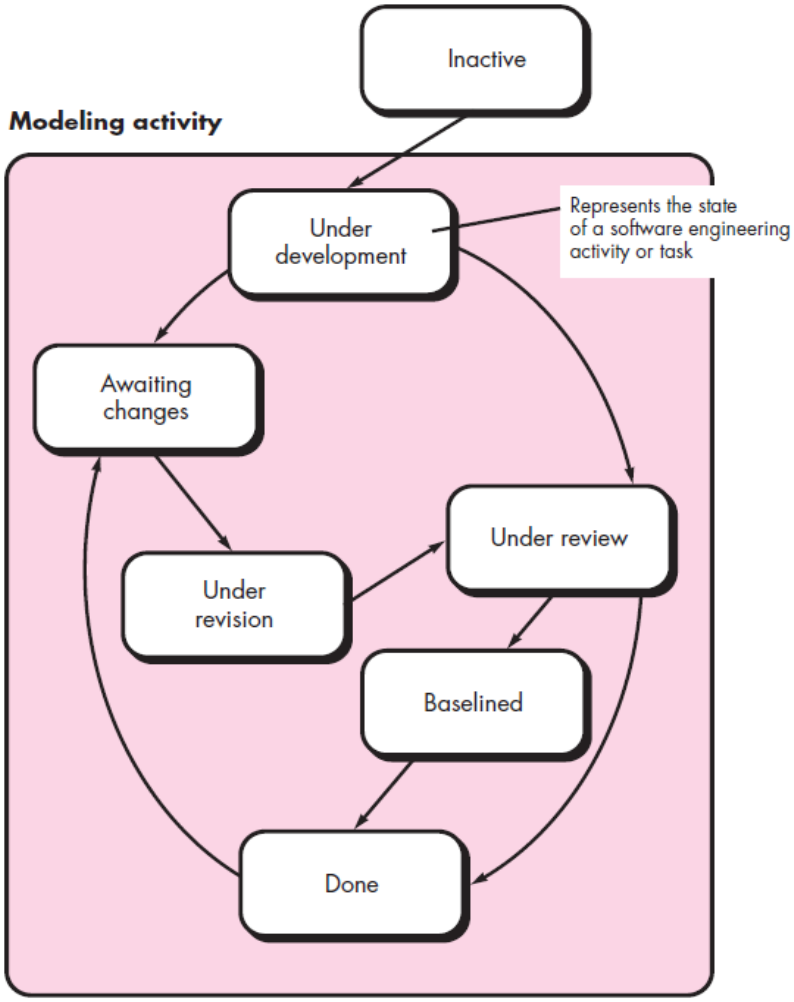
The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models.

Figure provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—modeling—may be in any one of the states12 noted at any given time. Similarly, other activities, actions, or tasks can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design, an inconsistency in the requirements model is uncovered. This generates the event analysis model correction, which will trigger the requirements analysis action from the done state into the awaiting changes state.

Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

FIGURE 2.8
One element of the concurrent process model



Specialized process models

Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software component.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented

classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps.

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The Formal Methods Model

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software.

The development of formal models is currently quite time consuming and expensive.

• Because few software developers have the necessary background to apply formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

Aspect-Oriented Software Development

As modern computer-based systems become more sophisticated, certain concerns—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as crosscutting concerns. Aspectual requirements define those crosscutting concerns

that have an impact across the software architecture. Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern"

The unified process

The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development.

The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse". It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Phases of the Unified Process

The inception phase of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires. Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them.

The elaboration phase encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline "that represents a "first cut" executable system. The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at

the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

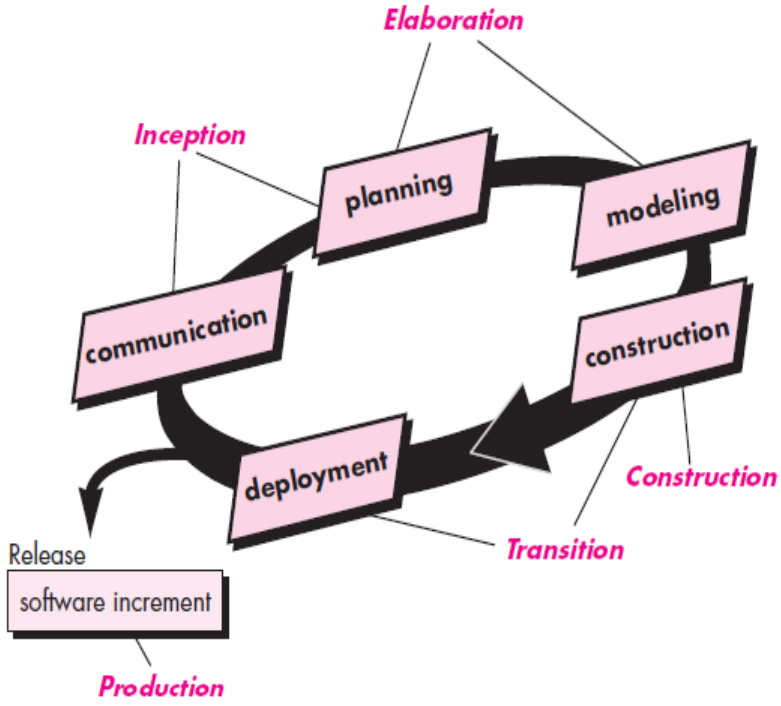
The construction phase of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code.

The transition phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment. Software is given to end users for beta testing and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release The production phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

The production phase of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

FIGURE 2.9

The Unified Process



Personal and team process models

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.

Personal Software Process (PSP)

The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities.

Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimates is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work result.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results

Postmortem. Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers, the resulting improvement in software engineering productivity and software quality are significant [Fer97]. However, PSP has not been widely adopted throughout the industry.

Team Software Process (TSP)

The goal of TSP is to build a "self-directed" project team that organizes itself to produce high-quality software. Humphrey defines the following objectives for TSP.

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.

• Show managers how to coach and motivate their teams and how to help them sustain peak performance.

• Accelerate software process improvement by making CMM Level 5 behavior normal and expected.

• Provide improvement guidance to high-maturity organizations.

• Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: project launch, high-level design, implementation, integration and test, and postmortem. Like their counterparts in PSP, these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. "Scripts" define specific process activities and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.

TSP recognizes that the best software teams are self-directed. Team members set project objectives, adapt the process to meet their needs, control the project schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineer.

Unit-II

Understanding Requirements:

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn't seem that hard.

Requirements Engineering.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. The specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resultant design.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

Inception. Most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization**.**

At project inception,3 you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.Christel and Kang identify a number of problems that are encountered as elicitation occurs.

Problems of scope. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objective

Problems of understanding. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.

Problems of volatility. The requirements change over time

Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information**.**

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs.

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification. In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

Validation. The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the

specification5 to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic (unachievable)requirements.

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents
Revision History

1. Introduction
1.1 Purpose
1.2 Document Conventions
1.3 Intended Audience and Reading Suggestions
1.4 Project Scope
1.5 References

2. Overall Description
2.1 Product Perspective
2.2 Product Features
2.3 User Classes and Characteristics
2.4 Operating Environment
2.5 Design and Implementation Constraints
2.6 User Documentation
2.7 Assumptions and Dependencies

3. System Features
3.1 System Feature 1
3.2 System Feature 2 (and so on)

4. External Interface Requirements
4.1 User Interfaces
4.2 Hardware Interfaces
4.3 Software Interfaces
4.4 Communications Interfaces

5. Other Nonfunctional Requirements
5.1 Performance Requirements
5.2 Safety Requirements
5.3 Security Requirements
5.4 Software Quality Attributes

6. Other Requirements

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

Eliciting requirements

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements.

Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines**.**

- Meetings are conducted and attended by both software engineers and other stakeholders.

• Rules for preparation and participation are established.

• An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.

• A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.

• A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. To better understand the flow of events as they occur,

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD"concentrates on maximizing customer satisfaction from the software engineering process". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.QFD identifies three types of requirements.

Normal requirements .The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements. These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the customer voice table—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements.

Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases, provide a description of how the system will be used.

Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include.

A statement of need and feasibility.

• A bounded statement of scope for the system or product.

• A list of customers, users, and other stakeholders who participated in requirements elicitation.

• A description of the system's technical environment.

• A list of requirements (preferably organized by function) and the domain constraints that apply to each.

• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.

• Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

Developing use cases

The first step in writing a use case is to define the set of "actors" that will be involved in the story. Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, where as an actor represents a class of external entities that play just one role in the context of the use case.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system. Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work. A number of questions12 that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?

• What are the actor's goals?
• What preconditions should exist before the story begins?
• What main tasks or functions are performed by the actor?
• What exceptions might be considered as the story is described?
• What variations in the actor's interaction are possible?
• What system information will the actor acquire, produce, or change?
• Will the actor have to inform the system about changes in the external environment?
• What information does the actor desire from the system?
• Does the actor wish to be informed about unexpected changes?

we define four actors: homeowner (a user), setup manager (likely the same person as homeowner, but playing a different role), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors the SafeHome home security function). For the purposes of this example, we consider only the homeowner actor. The homeowner actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC.

Use case: InitiateMonitoring
Primary actor: Homeowner.
Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions: System has been programmed for a password and to recognize various sensors.
Trigger: The homeowner decides to "set" the system, i.e., to turn on the alarm functions.

Scenario:
1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects “stay” or “away”
4. Homeowner: observes read alarm light to indicate that SafeHome has been armed
Exceptions:
1. Control panel is not ready: homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner reenters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. Stay is selected: control panel beeps twice and a stay light is lit; perimeter sensors are activated.
5. Away is selected: control panel beeps three times and an away light is lit; all sensors are activated.

FIGURE 5.1
SafeHome control panel

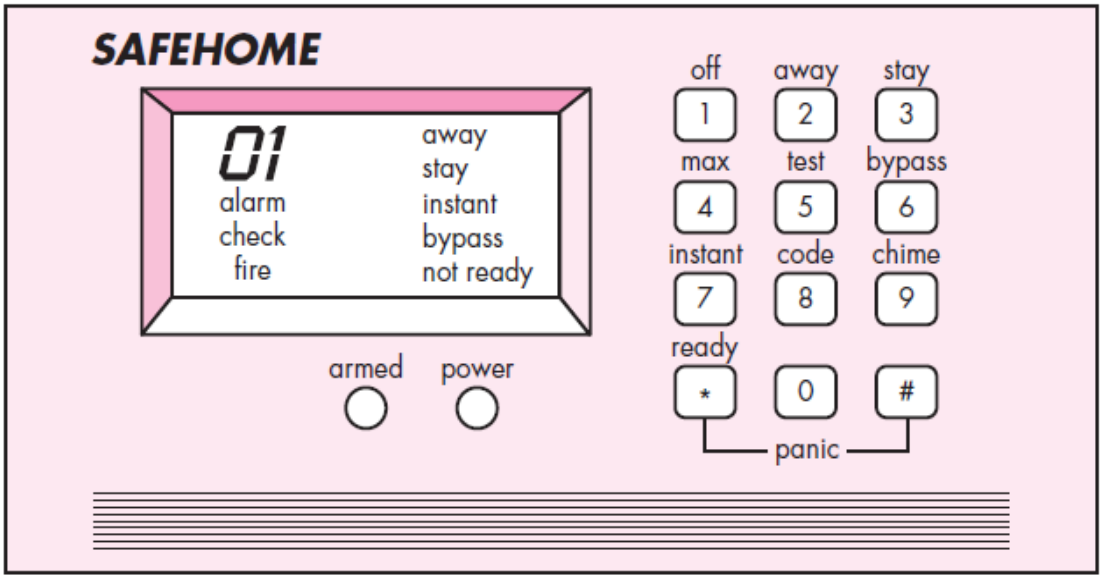
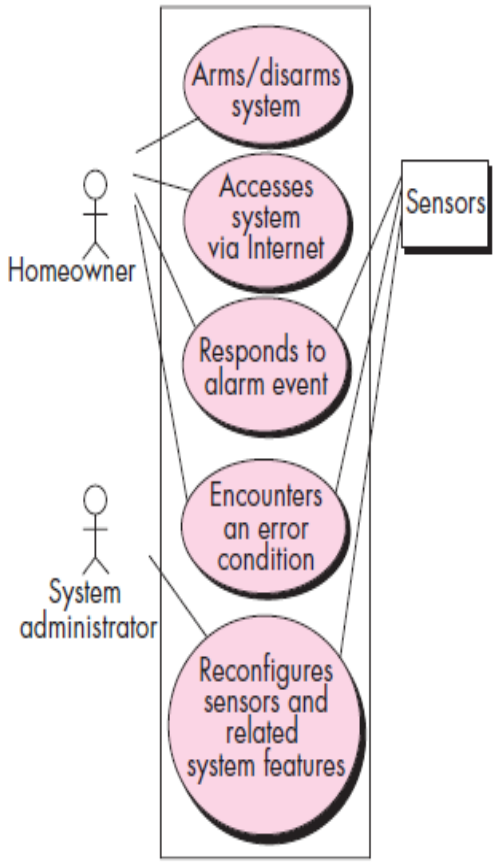


FIGURE 5.2

UML use case diagram for *SafeHome* home security function



Negotiating requirements

The best negotiations strive for a "win-win" result.20 That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined.

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team)

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

Validating requirements

The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions.

Is each requirement consistent with the overall objectives for the system/product?

• Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
• Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
• Is each requirement bounded and unambiguous?
• Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
• Do any requirements conflict with other requirements?
• Is each requirement achievable in the technical environment that will house the system or product?
• Is each requirement testable, once implemented?
• Does the requirements model properly reflect the information, function, and behavior of the system to be built.
• Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
• Have requirements patterns been used to simplify the requirements model?
Have all patterns been properly validated? Are all patterns consistent with customer requirements?

Requirements Modeling:

Requirements Analysis

software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model actually a set of models—is the first technical representation of a system.

Requirements analysis

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:
Scenario-based models of requirements from the point of view of various system "actors"
• Data models that depict the information domain for the problem
• Class-oriented models that represent object-oriented classes and the manner in which classes collaborate to achieve system requirements
• Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system
• Behavioral models that depict how the software behaves as a consequence of external "events".

These models provide a software designer with information that can be translated

to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

scenario-based modeling—a technique that is growing increasingly popular throughout the software engineering community; data modeling—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and class modeling—a representation of the object-oriented classes and the resultant collaborations that allow a system to function. Flow-oriented models, behavioral models,

Overall Objectives and Philosophy

The requirements model must achieve three primary objectives:

(1) to describe what the customer requires,

(2) to establish a basis for the creation of a software design, and

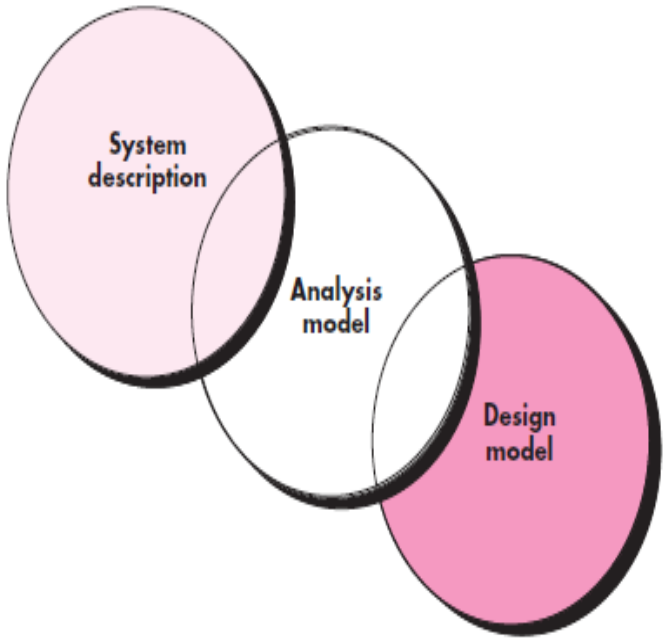
(3) to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

FIGURE 6.1

The requirements model as a bridge between the system description and the design model



Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high. "Don't get bogged down in details" that try to explain how the system will work.

• Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.

• Delay consideration of infrastructure and other nonfunctional models until design. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

• Minimize coupling throughout the system. It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

• Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; acceptance tests.

• Keep the model as simple as it can be. Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

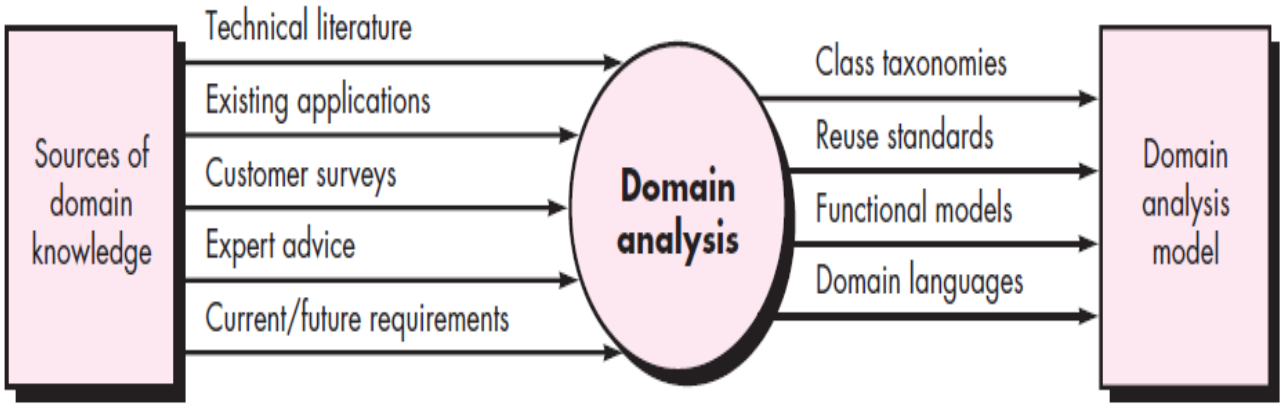
Domain Analysis

Analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

Domain analysis may be viewed as an umbrella activity for the software process. Domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master tool smith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst5 is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 6.2 illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

FIGURE 6.2 Input and output for domain analysis



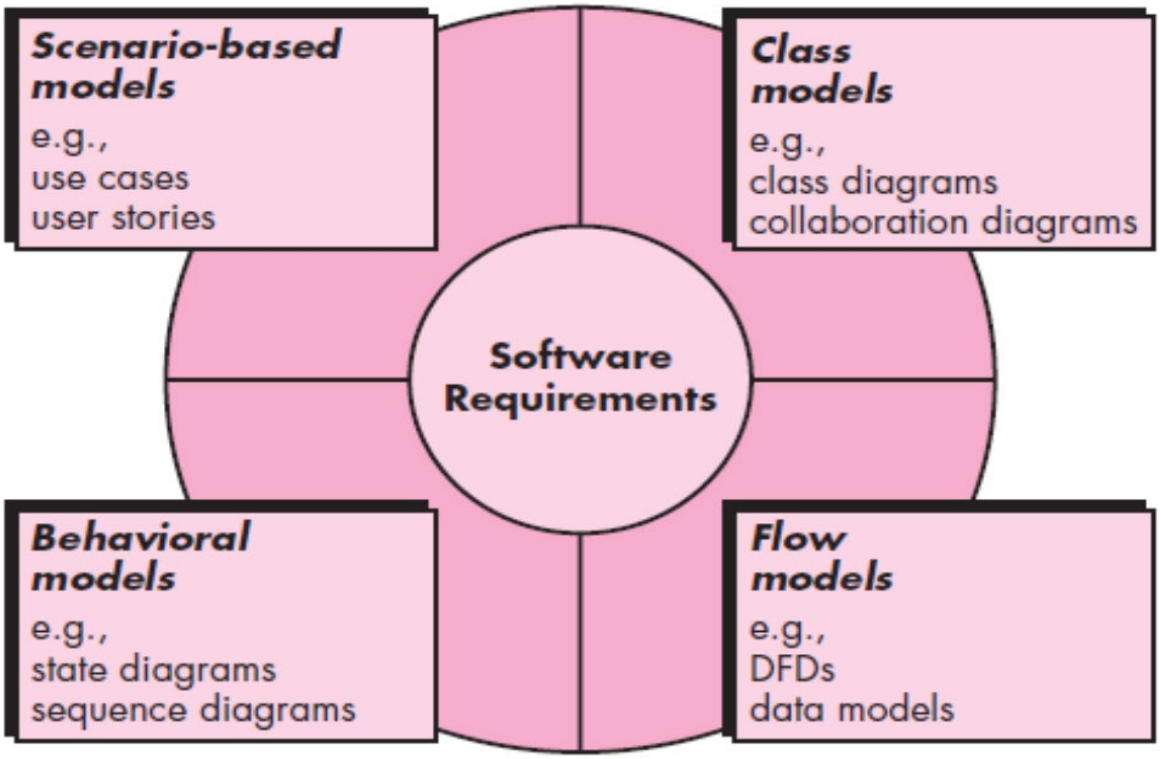
Requirements Modeling Approaches

One view of requirements modeling, called structured analysis, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called object-oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element may differ from project to project.



Scenario-based modeling

Success of a computer-based system or product is measured in many ways; user satisfaction resides at the top of the list. If you understand how end users want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

Creating a Preliminary Use Case

In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. The following are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

(1) What to write about,
(2) How much to write about it,
(3) How detailed to make your description, and
(4) How to organize the description

What to write about?

The first two requirements engineering tasks—inception and elicitation—provide you with the information need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system

mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. Obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams developed as part of requirements modeling.

The SafeHome home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the homeowner actor.

Select camera to view.

• Request thumbnails from all cameras.

• Display camera views in a PC window.

• Control pan and zoom for a specific camera.

• Selectively record camera output.

• Replay camera output.

• Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner)progress, the requirements gathering team develops use cases for each of the functions noted. A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence.

Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions

- Can the actor take some other action at this point?

• Is it possible that the actor will encounter some error condition at this point? If so, what might it be?

• Is it possible that the actor will encounter some other behavior at this point? If so, what might it be?

Is it possible that the actor will encounter some other behavior at this point If so, what might it be.

The homeowner selects “pick a camera.”

The system displays the floor plan of the house.

Can the actor take some other action at this point

Is it possible that the actor will encounter some error condition at this point?

Is it possible that the actor will encounter some other behavior at this point?

Are there cases in which some “validation function” occurs during this use case?

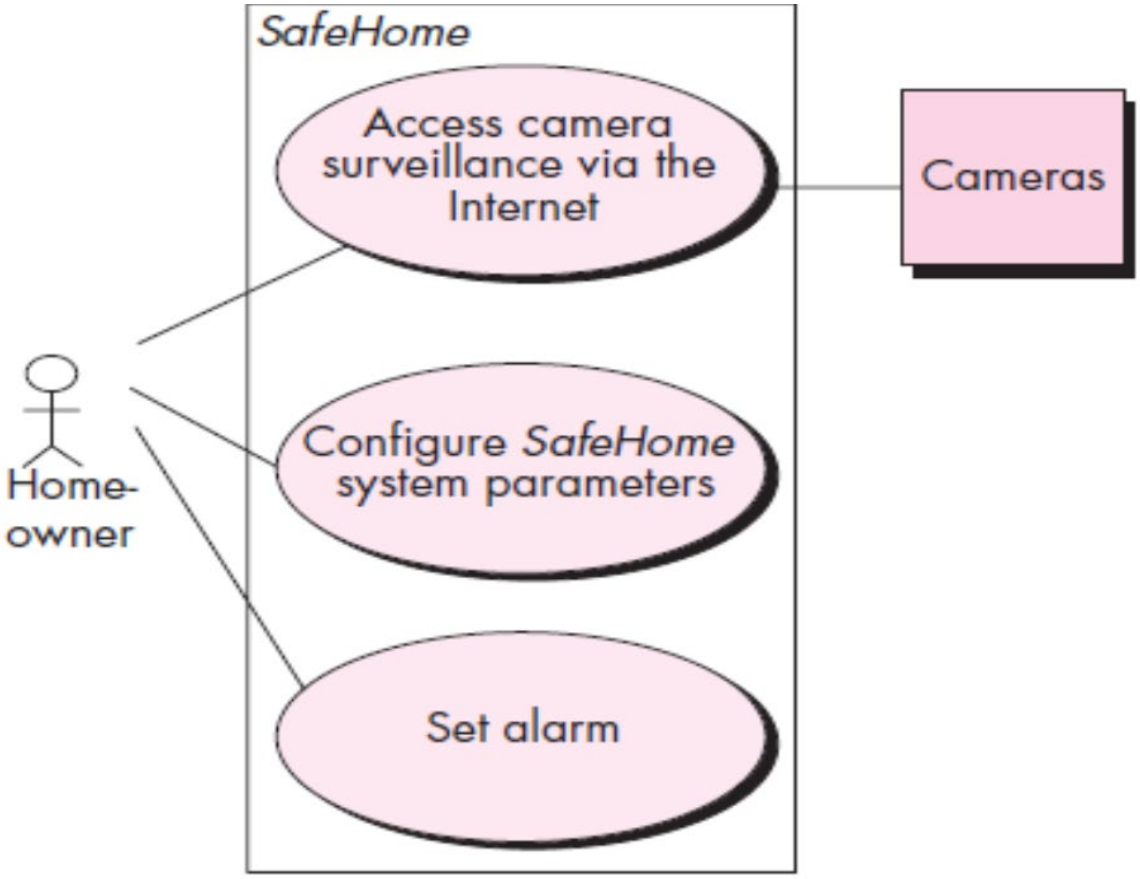
This implies that validation function is invoked and a potential error condition might occur.

Are there cases in which a supporting function (or actor) will fail to respond appropriately?
For example, a user action awaits a response but the function that is to respond times out.
Can poor system performance result in unexpected or improper user actions?

Writing a Formal Use Case

The goal in context identifies the overall scope of the use case. The precondition describes what is known to be true before the use case is initiated. The trigger identifies the event or condition that "gets the use case started". The scenario lists the specific actions that are required by the actor and the appropriate system responses. Exceptions identify the situations uncovered as the preliminary use case is refined. Additional headings may or may not be included and are reasonably self-explanatory.

Preliminary use-case diagram for the SafeHome system



UML model that supplement the Use-Case

There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner.

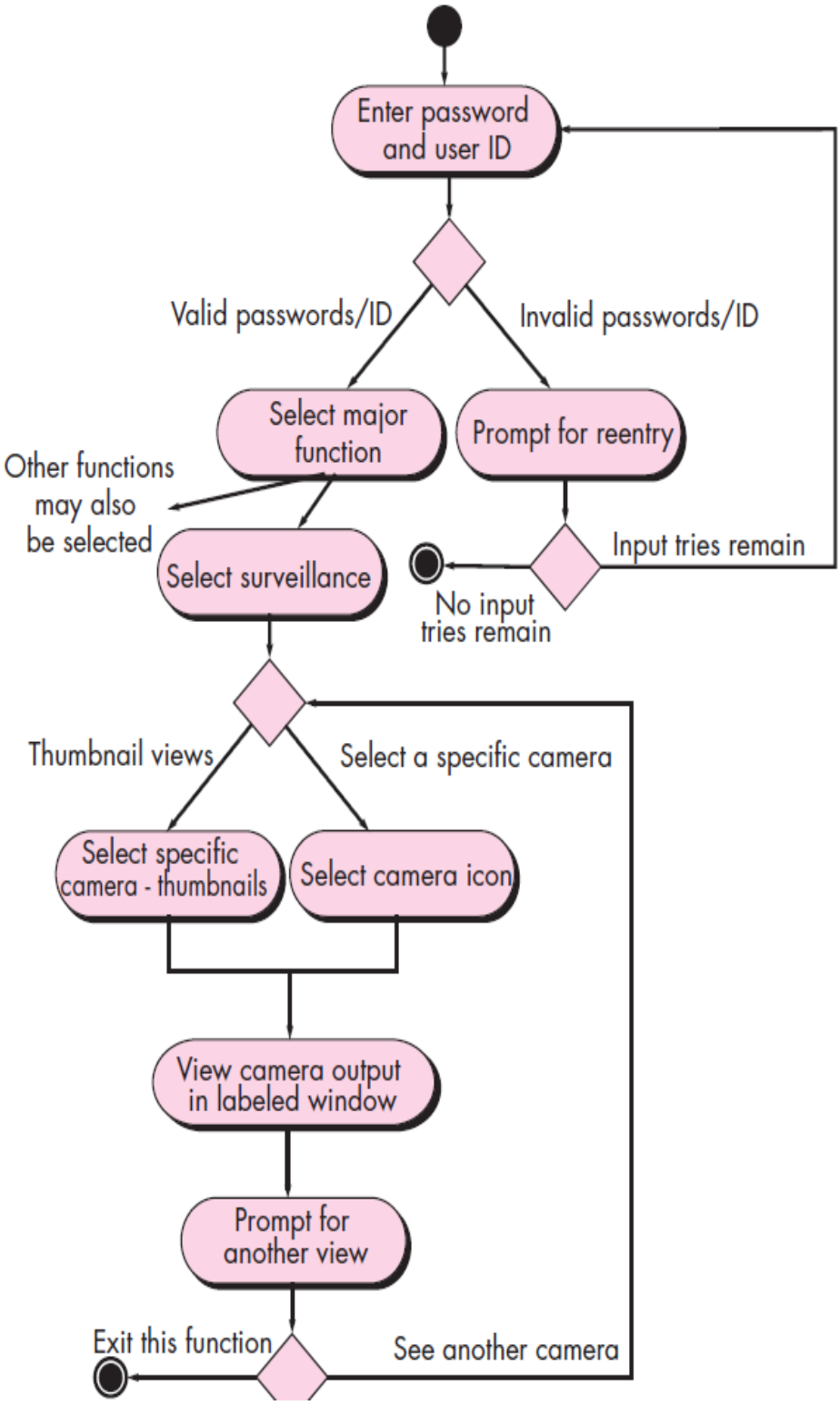
Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function,arrows to represent flow

through the system, decision diamonds to depict a branching decision and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the ACS-DCV use case is shown in Figure 6.5. It should be noted that the activity diagram adds additional detail not directly mentioned by the use case.

FIGURE 6.5

Activity diagram for Access camera surveillance via the Internet—display camera views function.



Swimlane Diagrams

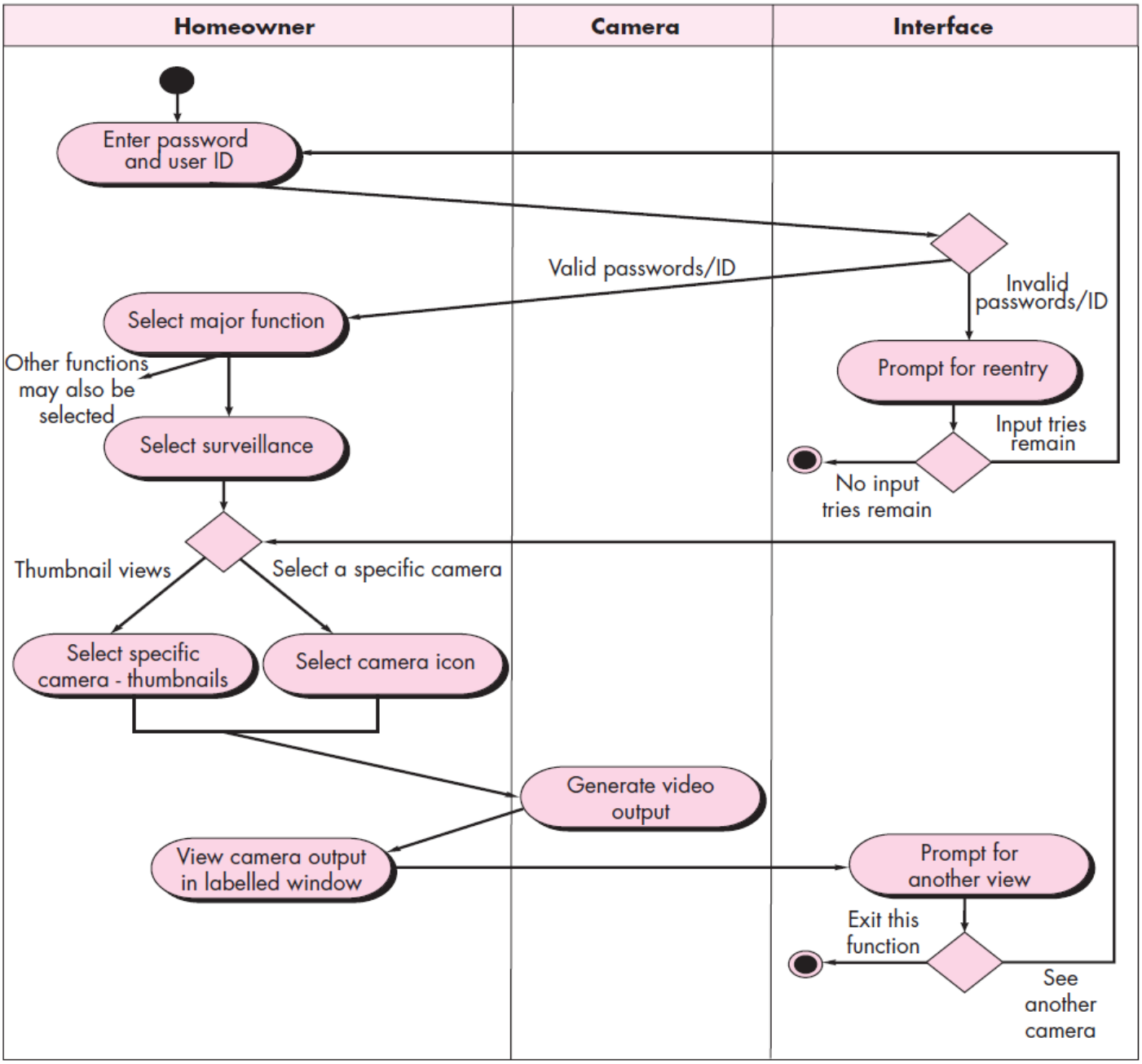
The UML swimlane diagram is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class (discussed later in this chapter) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—Homeowner, Camera, and Interface—have direct or indirect responsibilities in the context of the activity diagram represented in Figure.

The activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the Interface swimlane.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system.

FIGURE 6.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function



Data modeling concepts

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The entity-relationship

diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed and produced within an application.

Data Objects

A data object is a representation of composite information that must be understood by software. By composite information, Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g.,accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) Name an instance of the data object,

(2) Describe the instance, or

(3) Make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context.

FIGURE 6.7

Tabular representation of data objects

Instance

Naming attributes

Ties one data object to another, in this case, owner

Identifier

Descriptive attributes

Referential attributes

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Relationship

Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation illustrated in Figure. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. Establish a set of object /relationship pairs that define the relevant relationships. For example

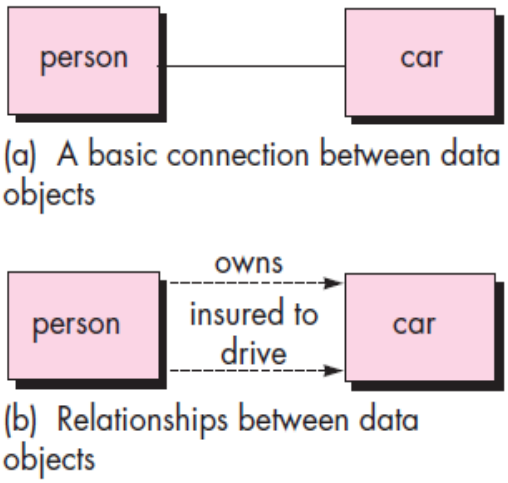
• A person owns a car.

• A person is insured to drive a car.

The relationships own and insured to drive define the relevant connections between person and car. Figure illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretation.

FIGURE 6.8

Relationships between data objects



(a) A basic connection between data objects

(b) Relationships between data objects

Class-Responsibility-Collaborator (CRC) Modeling

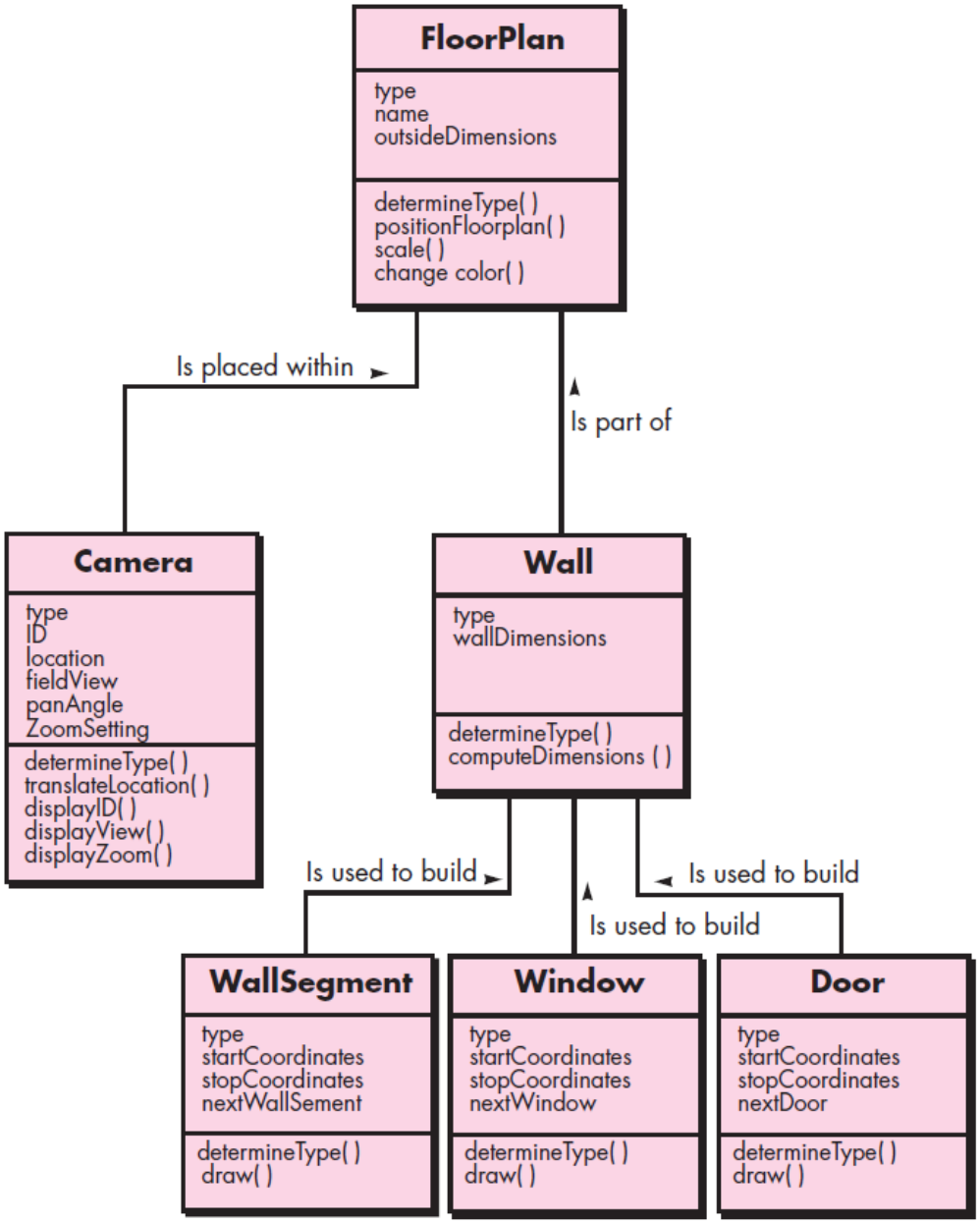
Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does". Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a collaboration implies either a request for information or a request for some action.

FIGURE 6.10

Class diagram for FloorPlan (see sidebar discussion)



A simple CRC index card for the FloorPlan class is illustrated in Figure.

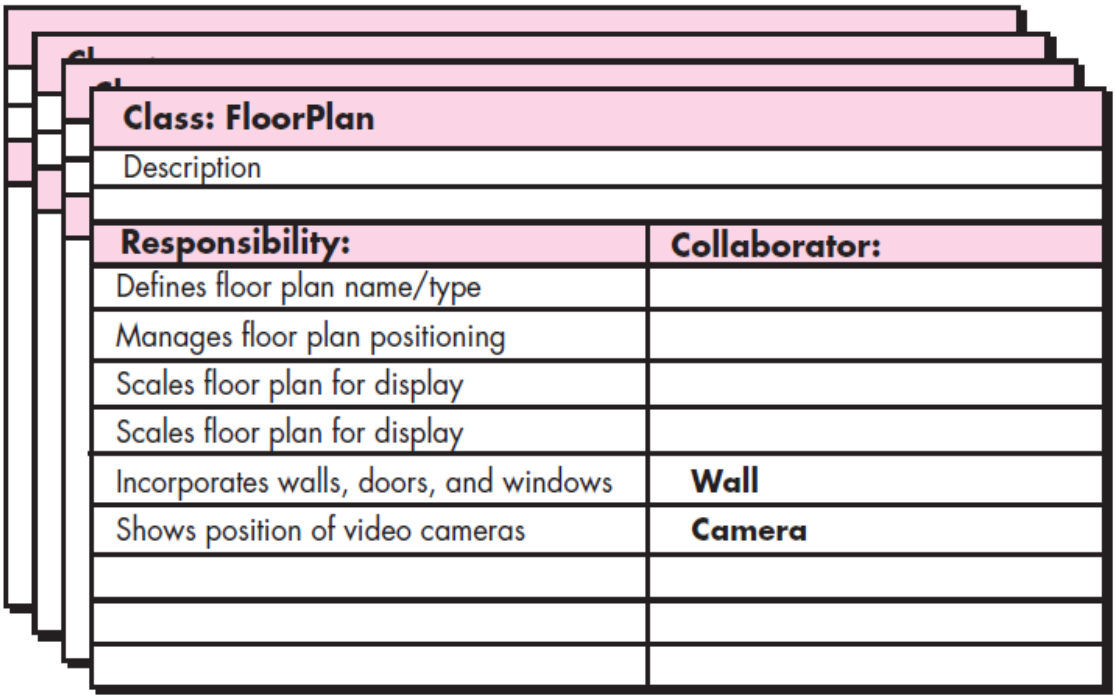
The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification.

Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types extended by considering the following categories.

Entity classes, also called model or business classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor). These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.

Boundary classes are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. ntity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

FIGURE 6.11
CRC model dex card



Controller classes manage a "unit of work" from start to finish. That is, controller classes can be designed to manage
(1) the creation or update of entity objects,
(2) the instantiation of boundary objects as they obtain information from entity objects,
(3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application.
In general, controller classes are not considered until the design activity has begun.

Responsibilities.

Basic guidelines for identifying responsibilities (attributes and operations). five guidelines for allocating responsibilities to classes.

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways.

   To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence. In addition, the responsibilities for each class should exhibit the same level of abstraction.
2. **Each responsibility should be stated as generally as possible**. This guideline implies that general responsibilities should reside high in the class hierarchy.

3. **Information and the behavior related to it should reside within the same class**.This achieves the object-oriented principle called encapsulation. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes**. A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes**, when appropriate. There are many cases in which a variety of related objects must all exhibit the same behavior at the same time

Collaborations

Classes fulfill their responsibilities in one of two ways:

(1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or

(2) a class can collaborate with other classes.

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A Collaboration is the embodiment of the contract between the client and the server. We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

To help in the identification of collaborators, examine three different generic relationships between classes:

(1) the is-part-of relationship,

(2) the has-knowledge-of relationship, and

(3) the depends-upon relationship.

Each of the three generic relationships is considered briefly in the paragraphs that follow.

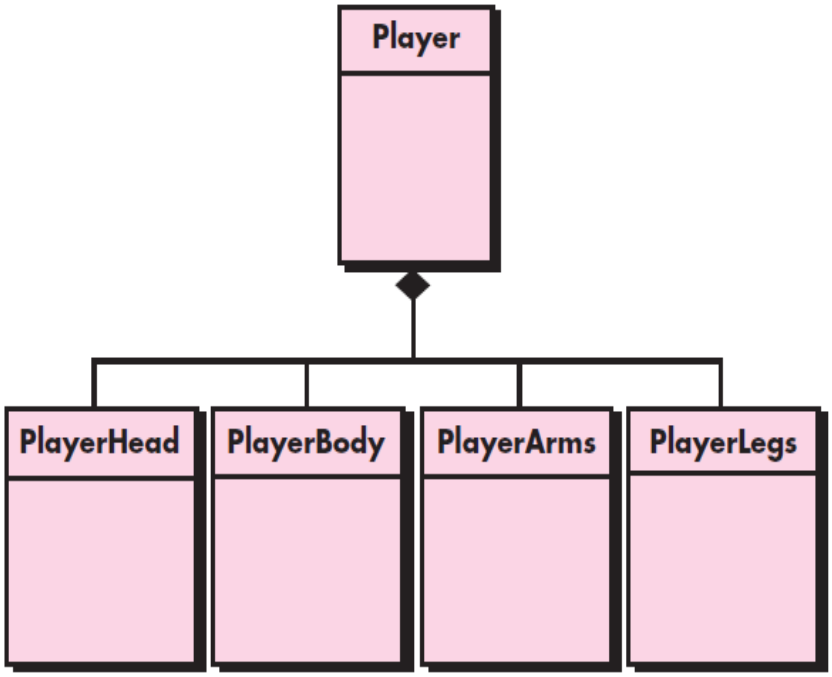
All classes that are part of an aggregate class are connected to the aggregate class via an is-part-of relationship. When one class must acquire information from another class, the has-knowledgeof relationship is established. The determine-sensor-status() responsibility noted earlier is an example of a has-knowledge-of relationship.

The depends-upon relationship implies that two classes have a dependency that is not achieved by has-knowledge-of or is-part-of.

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of

responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled.

FIGURE 6.12
A composite aggregate class



When a complete CRC model has been developed, stakeholders can review the model using the following approach.

1. All participants in the review are given a subset of the CRC model index cards. Cards that collaborate should be separated.
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
4. When the token is passed, the holder of the Sensor card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes or the specification of new or revised responsibilities or collaborations on existing cards.

Associations and Dependencies

In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another. In UML these relationships are called associations.

In some cases, an association may be further defined by indicating multiplicity. Referring to Figure. Where “one or more” is represented using 1. .*, and “0 or more” by 0 . .*. In UML, the asterisk indicates an unlimited upper bound on the range.1.

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a dependency relationship is established. Dependencies are defined by a stereotype. A stereotype is an “extensibility mechanism” within UML that allows defining a special modeling element whose semantics are custom defined. In UML stereotypes are represented in double angle brackets (e.g., <<stereotype>>).

FIGURE 6.13

Multiplicity

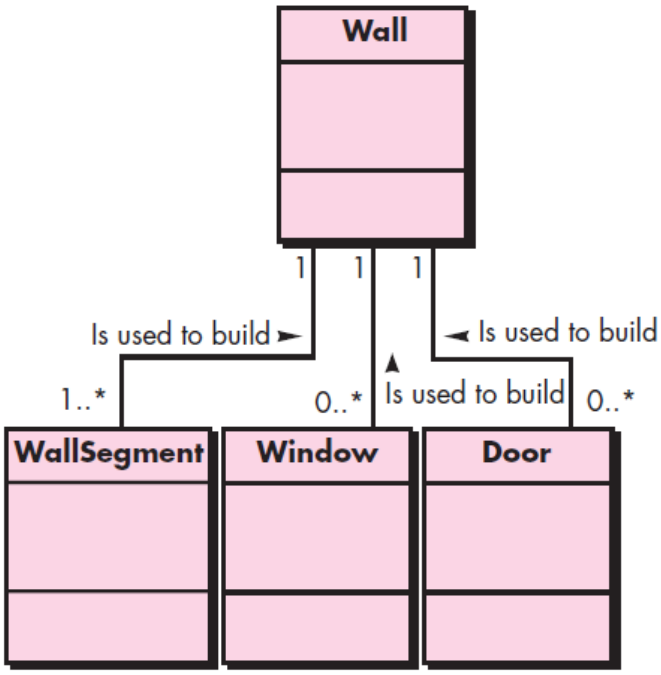
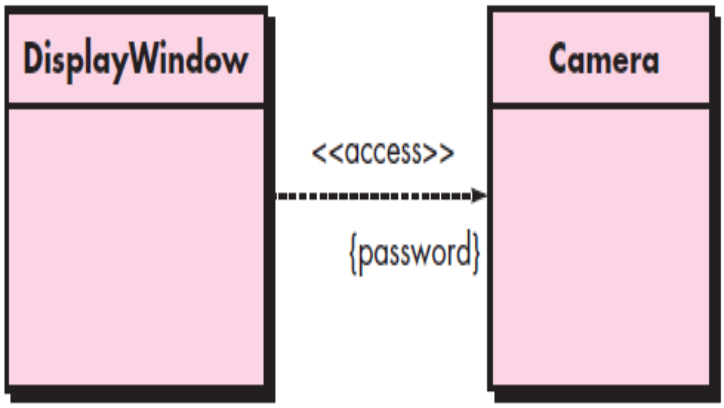


FIGURE 6.14

Dependencies



Analysis Packages

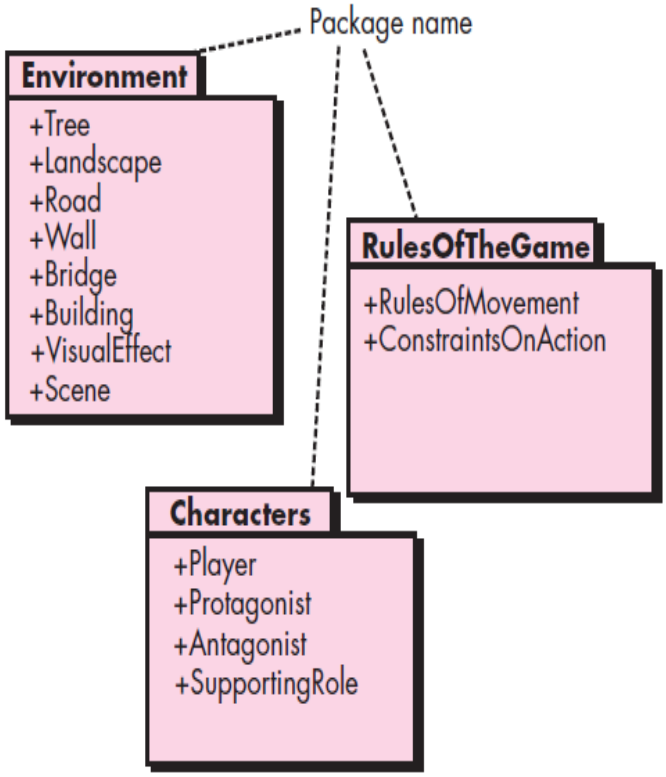
An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an analysis package—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that as the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as Tree, Landscape, Road, Wall, Bridge, Building, and VisualEffect might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints.

The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages. Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

FIGURE 6.15

Packages



Unit-V

Software project management

Effective project management is crucial to the success of any software development project. In the past, several projects have failed not for want of competent technical professionals neither for lack of resources, but due to the use of faulty project management practices. Therefore, it is important to carefully learn the latest software project management techniques.

The main goal of software project management is to enable a group of developers to work effectively towards the successful completion of a project.

A project manager is usually an experienced member of the team who essentially works as the administrative leader of the team. For small software development projects, a single member of the team assumes the responsibilities for both project management and technical management. For large projects, a different member of the team (other than the project manager) assumes the responsibility of technical leadership. The responsibilities of the technical leader includes addressing issues such as which tools and techniques to use in the project, high-level solution to the problem, specific algorithms to use,etc

Once a project has been found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed before any development activity starts.

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. For this reason, project planning is undertaken by the project managers with utmost care and attention.

During project planning, the project manager performs the following activities. Note that we have given only a very brief description of the activities.

Estimation: The following project attributes are estimated.

- **Cost:** How much is it going to cost to develop the software product?
- **Duration:** How long is it going to take to develop the product?
- **Effort:** How much effort would be necessary to develop the product?

The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

Scheduling: After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.

Staffing: Staff organisation and staffing plans are made.

Risk management : This includes risk identification, analysis, and abatement planning.

Miscellaneous plans: This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

Miscellaneous plans: This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

Figure shows the order in which the planning activities are undertaken. Observe that size estimation is the first activity that a project manager undertakes during project planning.

Size is the most fundamental parameter based on which all other estimations and project plans are made.

As can be seen from Figure based on the size estimation, the effort required to complete a project and the duration over which the development is to be carried out are estimated. Based on the effort estimation, the cost of the project is computed. The estimated cost forms the basis on which price negotiations with the customer is carried out. Other planning activities such as staffing, scheduling etc. are undertaken based on the effort and duration estimates made.

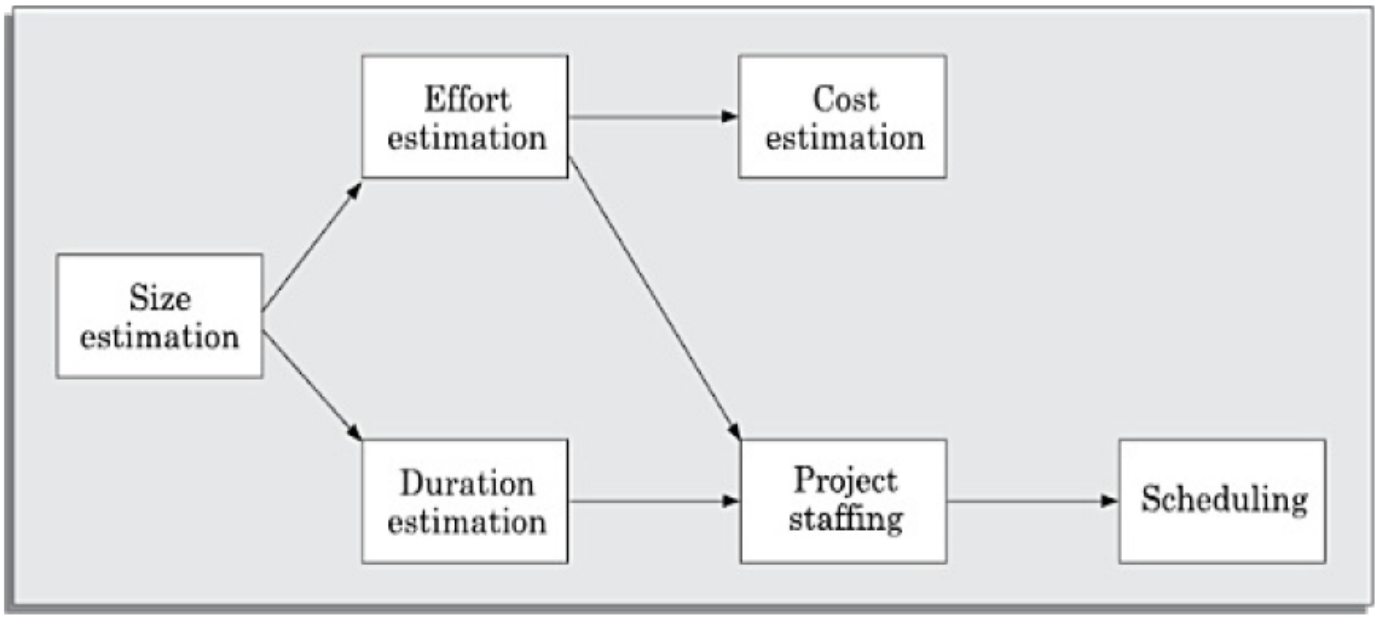


Figure 3.1: Precedence ordering among planning activities.

Sliding Window Planning

Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as sliding window planning. In the sliding window planning technique, starting with an initial plan, the project is planned more accurately over a number of stages.

At the start of a project, the project manager has incomplete knowledge about the nitty-gritty of the project. His information base gradually improves as the project progresses through different development phases. The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear. The project parameters are re-estimated periodically as understanding grows and also aperiodically as project parameters change. By taking these developments into account, the project manager can plan the subsequent activities more accurately and with increasing levels of confidence.

The SPMP Document of Project Planning

Once project planning is complete, project managers document their plans in a software project management plan (SPMP) document. Listed below are the different items that the SPMP document should discuss. This list can be used as a possible organization of the SPMP document.

Organisation of the software project management plan (SPMP) document

1. **Introduction**

(a) Objectives

(b) Major Functions

(c) Performance Issues

(d) Management and Technical Constraints

2**. Project estimates**

(a) Historical Data Used

(b) Estimation Techniques Used

(c) Effort, Resource, Cost, and Project Duration Estimates

3**. Schedule**

(a) Work Breakdown Structure

(b) Task Network Representation

(c) Gantt Chart Representation

(d) PERT Chart Representation

4. **Project resources**

(a) People

(b) Hardware and Software

(c) Special Resources

5. **Staff organisation**

(a) Team Structure

(b) Management Reporting

6**. Risk management plan**

(a) Risk Analysis

(b) Risk Identification

(c) Risk Estimation

(d) Risk Abatement Procedures

7. **Project tracking and control plan**

(a) Metrics to be tracked

(b) Tracking plan

(c) Control plan

8. **Miscellaneous plans**

(a) Process Tailoring

(b) Quality Assurance Plan

(c) Configuration Management Plan

(d) Validation and Verification

(e) System Testing Plan

(f) Delivery, Installation, and Maintenance Plan

Metrics for project size estimation

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product. Currently, two metrics are popularly being used to measure **size—lines of code (LOC**) and **function point (FP)**. Each of these metrics has its own advantages and disadvantages.

Lines of Code (LOC)

LOC is possibly the simplest among all metrics available to measure project size. Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.

One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and so on, until the LOC of the leaf-level modules are small enough to be predicted. To be able to predict the LOC count for the various leaf-level modules sufficiently accurately, past experience in developing similar modules is very helpful. By adding the estimates for all leaf level modules together, project managers arrive at the total size estimation.

LOC is a measure of coding activity alone.

A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e. specification, design, code, test, etc.) and not just the coding effort. LOC, however, focuses on the coding activity alone—it merely computes the number of source lines in the final program.

LOC count depends on the choice of specific instructions:

LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers. By coding style, we mean the choice of code layout, the choice of the instructions in writing the program, and the specific algorithms used. Different programmers may lay out their code in very different ways.

LOC measure correlates poorly with the quality and efficiency of the code:

Larger code size does not necessarily imply better quality of code or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms.

LOC metric penalises use of higher-level programming languages and code reuse:

A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. Modern programming methods such as object-oriented programming and reuse of components makes the relationships between LOC and other project attributes even less precise.

LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities:

Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic. To realise why this is so, imagine the effort that would be required to develop a program having multiple nested loops and decision constructs and compare that with another program having only sequential control flow.

It is very difficult to accurately estimate LOC of the final program from problem specification: As already discussed, at the project initiation time, it is a very difficult task to accurately estimate the number of lines of code (LOC) that the program would have after development. The LOC count can accurately be computed only after the code has fully been developed. Since project planning is carried out even before any development activity starts, the LOC metric is of little use to the project managers during project planning.

Function Point (FP) Metric

One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself. Using the LOC metric, on the other hand, the size can accurately be determined only after the product has fully been developed. The size of a software product is directly dependent on the number of different high-level functions or features it supports. This assumption is reasonable, since each feature would take additional effort to implement.

For example, the query book feature of a Library Automation Software takes the name of the book as input and displays its location in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data. It can therefore be argued that the computation of the number of input and output data items would give a more accurate indication of the code size compared to simply counting the number of high-level functions supported by the system.

Function point (FP) metric computation

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

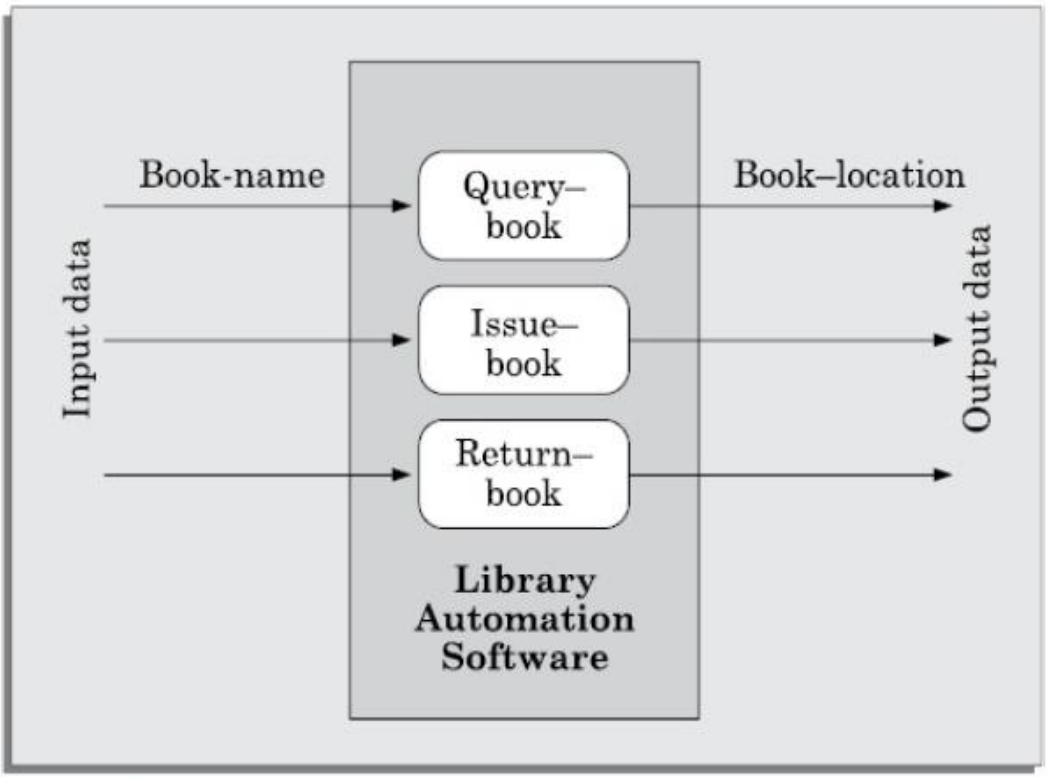


Figure 3.2: System function as a mapping of input data to output data.

Step 1: Compute the unadjusted function point (UFP) using a heuristic expression.

Step 2: Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.

Step 3: Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort**.**

Step 1: UFP computation

The unadjusted function points (UFP) is computed as the weighted sum of five characteristics of a product as shown in the following expression.

UFP = (Number of inputs)*4 + (Number of outputs)*5 +(Number of inquiries)*4 + (Number files)*10 + (Number of interfaces)*10

1. **Number of inputs**: Each data item input by the user is counted. However, it should be noted that data inputs are considered different from user inquiries. Inquiries are user commands such as printaccount-balance that require no data values to be input by the user. Inquiries are counted separately. It needs to be further noted that individual data items input by the user

are not simply added up to compute the number of inputs, but related inputs are grouped and considered as a single input.

2. **Number of outputs**: The outputs considered include reports printed, screen outputs, error messages produced, etc. While computing the number of outputs, the individual data items within a report are not considered; but a set of related data items is counted as just a single output.
3. **Number of inquiries**: An inquiry is a user command (without any data input) and only requires some actions to be performed by the system. Thus, the total number of inquiries is essentially the number of distinct interactive queries (without data input) which can be made by the users. Examples of such inquiries are print account balance; print all student grades, display rank holders' names, etc.
4. **Number of files**: The files referred to here are logical files. A logical file represents a group of logically related data. Logical files include data structures as well as physical files.
5. **Number of interfaces:** Here the interfaces denote the different mechanisms that are used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.

Step 2: Refine parameters

UFP computed at the end of step 1 is a gross indicator of the problem size. This UFP needs to be refined. This is possible, since each parameter (input,output,etc.) has been implicitly assumed to be of average complexity. The complexity of each parameter is graded into three broad categories—simple, average, or complex. The weights for the different parameters are determined based on the numerical values shown in Table 3.1. Based on these weights of the parameters, the parameter values in the UFP are refined. For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs.

Table 3.1: Refinement of Function Point Entities

Type	Simple	Average	Complex
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Step 3: Refine UFP based on complexity of the overall project

In the final step, several factors that can impact the overall project size are considered to refine the UFP computed in step 2. Examples of such project parameters that can influence the project sizes include high transaction rates,response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort. The list of these parameters have been shown in Table 3.2. Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence).The resulting numbers are summed, yielding the total degree of influence (DI). A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of the corresponding project parameters on the development effort. TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.49. Finally, FP is given as the product of UFP and TCF. That is,FP=UFP*TCF.

Table 3.2: Function Point Relative Complexity Adjustment Factors
Requirement for reliable backup and recovery
Requirement for data communication
Extent of distributed processing
Performance requirements
Expected operational environment
Extent of online data entries
Extent of multi-screen or multi-operation online data input
Extent of online updating of master files
Extent of complex inputs, outputs, online queries and files
Extent of complex data processing
Extent that currently developed code can be designed for reuse
Extent of conversion and installation included in the design
Extent of multiple installations in an organisation and variety of customer organisations
Extent of change and focus on ease of use

Project estimation techniques

Estimation of various project parameters is an important project planning activity. The different parameters of a project that need to be estimated include—project size, effort required to complete the project,project duration, and cost. Accurate estimation of these parameters is important, since these not only help in quoting an appropriate project cost to the customer, but also form the basis for resource planning and scheduling. A large number of estimation techniques have been proposed by researchers**.**

These can broadly be classified into three main categories:

Empirical estimation techniques

Heuristic techniques

Analytical estimation techniques

Empirical Estimation Techniques

Empirical estimation techniques are essentially based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalized to a large extent.

Heuristic Techniques

Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression. Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

Single variable estimation models

Assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:

Estimated Parameter =c1 * e^{d1}

In the above expression, e represents a characteristic of the software that has already been estimated (independent variable). Estimated P arameter is the dependent parameter (to be estimated). The dependent parameter to be estimated could be effort, project duration, staff size, etc., c1 and d1 are constants. The values of the constants c1 and d1 are usually determined using data collected from past projects (historical data).

A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:

Estimated Resource =c1* $p1^{d1}$+c2*$p2^{d2}$+…….

where, p1, p2, ... are the basic (independent) characteristics of the software already estimated, and c1, c2, d1, d2, ... are constants.

Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the different sets of constants c1 d1 , c2 , d2 , ... Values of these constants are usually determined from an analysis of historical data.

Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project. Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis**.**

Empirical estimation techniques

These techniques are easy to use and give reasonably accurate estimates. Two popular empirical estimation techniques are—Expert judgement and Delphi estimation techniques.

Expert Judgement

Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analysing the problem thoroughly**.**

Usually, the expert estimates the cost of the different components (i.e.modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate.However, this technique suffers from several shortcomings. The

outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant experience and knowledge of all aspects of a project.

A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimised when the estimation is done by a group of experts. However, the estimate made by a group of experts may still exhibit bias. Another important shortcoming of the expert judgment technique is that the decision made by a group may be dominated by overly assertive members.

Delphi Cost Estimation

Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator.In this approach, the co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the co-ordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process. The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the co-ordinator takes the responsibility of compiling the results and preparing the final estimate. The Delphi estimation, though consumes more time and effort,

Staffing level estimation

Once the effort required to complete a software project has been estimated, the staffing requirement for the project can be determined.

Norden's Work

Norden studied the staffing patterns of several R&D projects. He found that the staffing pattern of R&D type of projects is very different from that of manufacturing or sales. In a sales outlet, the number of sales staff does not usually vary with time. As the project progresses, the manpower requirement increases, until it reaches a peak.Thereafter, the manpower requirement gradually diminishes.

Norden represented the Rayleigh curve by the following equation:

$$E = \frac{K}{{t_d}^2} * t * e^{\frac{-t^2}{2{t_d}^2}}$$

where, E is the effort required at time t. E is an indication of the number of developers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value.

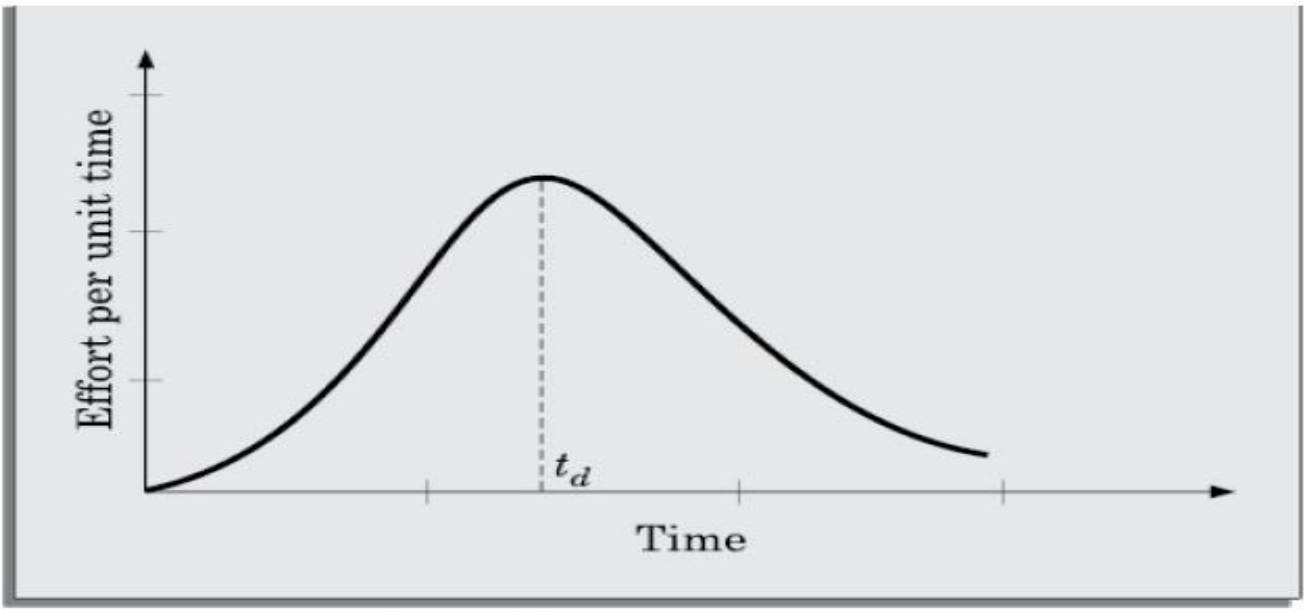


Figure 3.6: Rayleigh curve.

Putnam's Work

Only a small number of developers are needed at the beginning of a project to carry out the planning and specification tasks. As the project progresses and more detailed work is performed, the number of developers increases and reaches a peak during product testing. After implementation and unit testing, the number of project staff falls.

Putnam found that the Rayleigh-Norden curve can be adapted to relate the number of delivered lines of code to the effort and the time required to develop the product. By analyzing a large number of defence projects, Putnam derived the following expression.

$$L = C_k K^{1/3} t_d^{4/3}$$

where the different terms are as follows:

K is the total effort expended (in PM) in the product development and L is

the product size in KLOC.

• t d corresponds to the time of system and integration and testing. Therefore, td can be approximately considered as the time required developing the software.

Ck is the state of technology constant and reflects constraints that impede the progress of the programmer.

Typical values of C k =2 for poor development environment (no methodology, poor documentation, and review, etc.), C k =8 for good software development environment (software engineering principles are adhered to), Ck =11 for an excellent environment (in addition to following software engineering principles,automated tools and techniques are used). The exact value of Ck for a specific project can be computed from historical data of the organization developing it.

Jensen's Model

Jensen model [Jensen 84] is very similar to Putnam model. However, it attempts to soften the effect of schedule compression on effort to make it applicable to smaller and medium sized projects. Jensen proposed the equation:

$$L = C_{te} t_d K^{1/2}$$

Or

$$\frac{K_1}{K_2} = \frac{t_{d_2}^2}{t_{d_1}^2}$$

where, Cte is the effective technology constant, td is the time to develop the software, and K is the effort needed to develop the software.

Organization and team structures

Usually every software development organisation handles several projects at any time. Software organisations assign different teams of developers to handle different software projects. With regard to staff organisation, there are two important issues—how is the organization as a whole structured? And, how are the individual project teams structured? There are a few standard ways in which software organizations and teams can be structured.

Organisation Structure

Essentially there are three broad ways in which a software development organisation can be structured—functional format, project format, and matrix format.

Functional format

In the functional format, the development staff are divided based on the specific functional group to which they belong to. This format has schematically been shown in Figure.

The different projects borrow developers from various functional groups for specific phases of the project and return them to the functional group upon the completion of the phase. As a result, different teams of programmers from different functional groups perform different phases of a project. The partially completed product passes from one team to another as the product evolves. Therefore, the functional format requires considerable communication among the different teams and development of good quality documents because the work of one team must be clearly understood by the subsequent teams working on the project. The functional organisation therefore mandates good quality documentation to be produced after every activity.

Project format

In the project format, the development staff are divided based on the project for which they work (See Figure 3.13(b)). A set of developers is assigned to every project at the start of the project, and remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. An advantage of the project format is that it provides job rotation.That is, every developer undertakes different life cycle activities in a project. However, it results in poor manpower utilisation, since the full project team is formed since the start of the project, and there is very little work for the team during the initial phases of the life cycle.

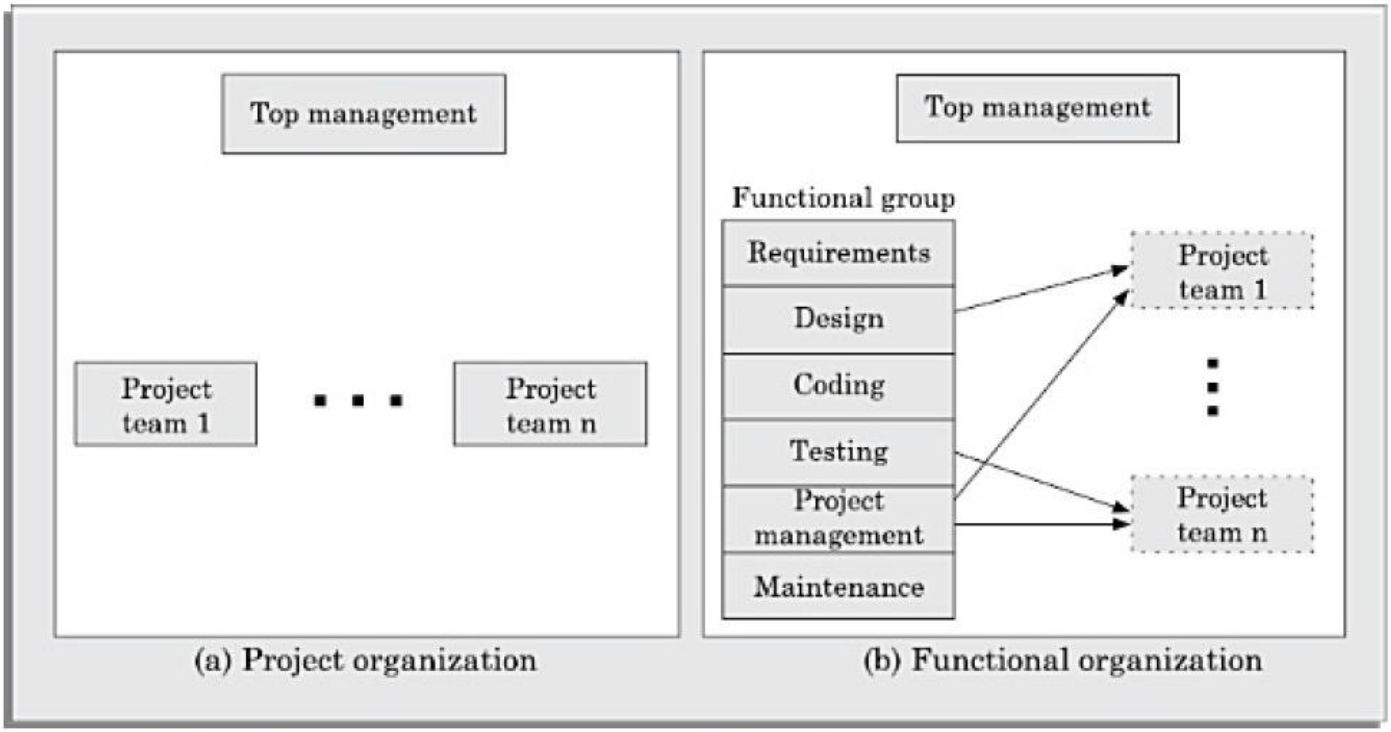


Figure 3.13: Schematic representation of the functional and project organisation.

Functional versus project formats

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional organisation are.

Ease of staffing

Production of good quality documents

Job specialization

Efficient handling of the problems associated with manpower turnover

The functional organisation allows the developers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases.

A further advantage of the functional organisation is that it is more effective in handling the problem of manpower turnover. This is because developers can be brought in from the functional pool when needed. Also, this organization mandates production of good quality documents, so new developers can quickly get used to the work already done.

Disadvantages.

The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, co der, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organisations to fill slots for some roles such as the maintenance, testing, and coding groups.

Another problem with the functional organisation is that if an organisation handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects.

For obvious reasons the functional format is not suitable for small organisations handling just one or two projects.

Matrix format

A matrix organization is intended to provide the advantages of both functional and project structures. In a matrix organization, the pool of functional specialists are assigned to different projects as needed. Thus, the deployment of the different functional specialists in different projects can be represented in a matrix. Figure 3.14 observe that different members of a functional specialization are assigned to different projects. Therefore in a matrix organization, the project manager needs to share responsibilities for the project with a number of individual functional managers.

Matrix organisations can be characterised as weak or strong, depending upon the relative authority of the functional managers and the project managers. In a strong functional matrix, the functional managers have authority to assign workers to projects and project managers have to accept the assigned personnel. In a weak matrix, the project manager controls the project budget, can reject workers from functional groups, or even decide to hire outside workers.

	Project			
Functional group	#1	#2	#3	
#1	2	0	3	Functional manager 1
#2	0	5	3	Functional manager 2
#3	0	4	2	Functional manager 3
#4	1	4	0	Functional manager 4
#5	0	4	6	Functional manager 5
	Project manager 1	Project manager 2	Project manager 3	

Figure 3.14: Matrix organisation.

Team Structure

Team structure addresses organisation of the individual project teams. Let us examine the possible ways in which the individual project teams are organised**.** Consider only three formal team structures—democratic, chief programmer, and the mixed control team organisations, although several other variations to these structures are possible. Projects of specific complexities and sizes often require specific team structures for efficient working.

Chief programmer team

In this team organisation, a senior engineer provides the technical leadership and is designated the chief programmer. The chief programmer partitions the task into many smaller tasks and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in Figure 3.15. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer. That is, a project might suffer severely, if the chief programmer either leaves the organisation or becomes unavailable for some other reasons.

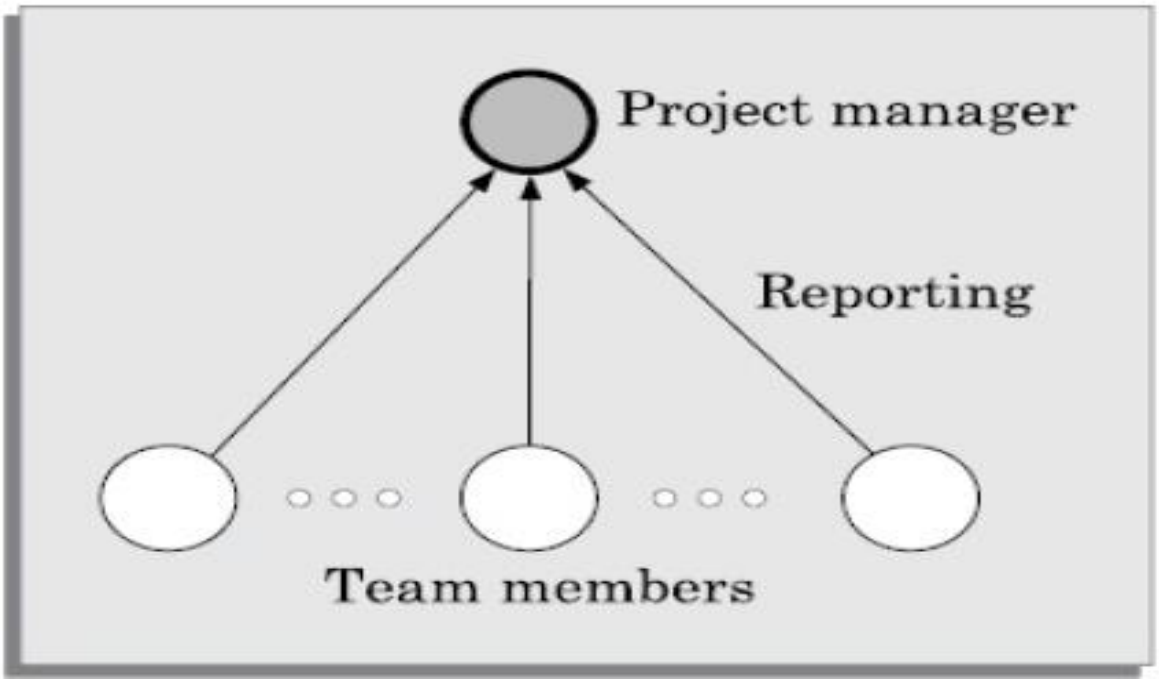


Figure 3.15: Chief programmer team structure.

The chief programmer team structure works well when the task is within the intellectual grasp of a single individual. However, even for simple and well understood problems, an organisation must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early completion outweighs other factors such as team morale, personal developments, etc.

Democratic team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy. Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

In a democratic organisation, the team members have higher morale and job satisfaction. Consequently, it suffers from less manpower turnover. Though the democratic teams are less productive compared to the chief programmer team, the democratic team structure is appropriate for less understood problems, since a group of developers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for research-oriented projects requiring less than five or six developers.

Mixed control team organization

The mixed control team organisation, as the name implies, draws upon the ideas from both the democratic organisation and the chief-programmer organisation. The mixed control team organisation is shown pictorially in Figure. This team organisation incorporates both hierarchical reporting and democratic set up. In Figure, the communication paths are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organisation is suitable for large team sizes. The democratic arrangement at the senior developers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organisation is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.

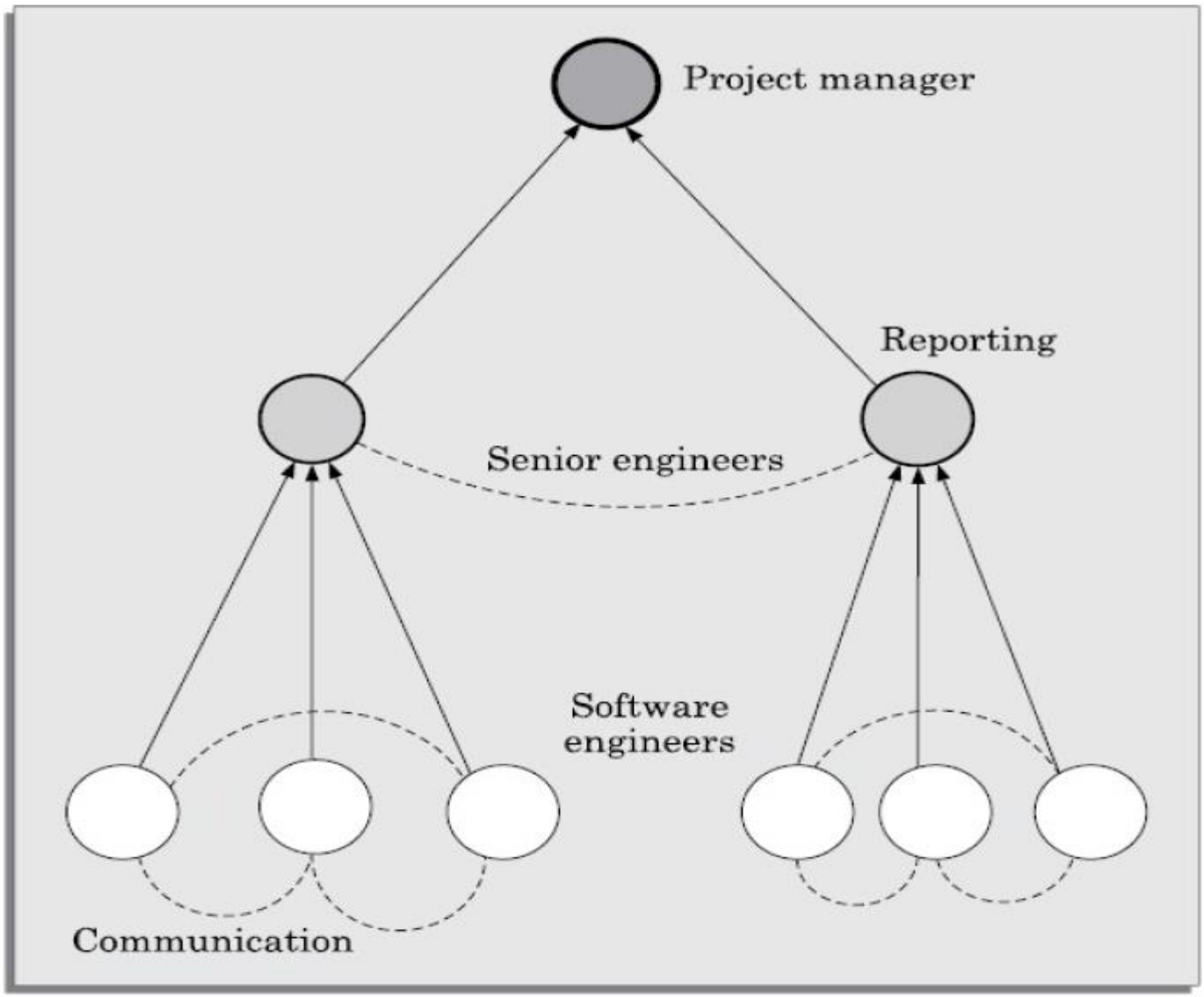


Figure 3.17: Mixed team structure.

Risk management

Every project is susceptible to a large number of risks. Without effective management of the risks, even the most meticulously planned project may go hay ware. A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway.

If a risk becomes real, the anticipated problem becomes a reality and is no more a risk. If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project. Therefore, it is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that contingency plans can be prepared beforehand to contain each risk. In this context, risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real.

Risk management consists of three essential activities—

Risk identification,

Risk assessment, and

Risk mitigation

Risk Identification

The project manager needs to anticipate the risks in a project as early as possible. As soon as a risk is identified, effective risk management plans are made, so that the possible impacts of the risks is minimised. So, early risk identification is important. Whether they would turn in poor quality work, whether some of your key personnel might leave the organisation, etc. All such risks that are likely to affect a project must be identified and listed.

A project can be subject to a large variety of risks. In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorise risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project:

Project risks,

Technical risks, and

Business risks.

Project risks: Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software

is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen.

Technical risks: Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due the development team's insufficient knowledge about the product.

Business risks: This type of risks includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

Risk Assessment

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

The likelihood of a risk becoming real (r).

The consequence of the problems associated with that risk (s). Based on these two factors, the priority of each risk can be computed as follows:

$$p = r * s$$

where, p is the priority with which the risk must be handled, r is the probability of the risk becoming real, and s is the severity of damage caused due to the risk becoming real. If all identified risks are prioritized, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for those risks.

Risk Mitigation

After all the identified risks of a project have been assessed, plans are made to contain the most damaging and the most likely risks first. Different types of risks require different containment procedures.

There are three main strategies for risk containment:

Avoid the risk: Risks can be avoided in several ways. Risks often arise due to project constraints and can be avoided by suitably modifying the constraints. The different categories of constraints that usually give rise to risks are

Process-related risk: These risks arise due to aggressive work schedule, budget, and resource utilisation.

Product-related risks: These risks arise due to commitment to challenging product features, quality, reliability etc.

Technology-related risks: These risks arise due to commitment to use certain technology

Transfer the risk: This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.

Risk reduction: This involves planning ways to contain the damage due to a risk.

The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.

To choose the most appropriate strategy for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this we may compute the risk leverage of the different risks. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

$$\text{risk leverage} = \frac{\text{risk exposure before reduction} - \text{risk exposure after reduction}}{\text{cost of reduction}}$$

Software configuration management

The results of a large software development effort typically consist of a large number of objects, e.g., source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software developers throughout the life cycle of the software. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

The configuration of the software is the state of all project deliverables at any point of time; and software configuration management deals with effectively tracking and controlling the configuration of a software during its life cycle. As a software is changed, new revisions and versions get created.

Software revision versus version

A new version of a software is created when there is significant change in functionality, technology, or the hardware it runs on, etc. On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc. Even the initial delivery might consist of several versions and more versions might be added later on.

As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace an old one. Often systems are described as version m, release n; or simply mn. Formally, a history relation is version of can be defined between objects. This relation can be split into two sub relations is revision of and is variant of.

Necessity of Software Configuration Management

There are several reasons for putting an object under configuration management. The following are some of the important problems that can crop up, if configuration management is not used: every software developer has a personal copy of an object (e.g. source code). When a developer makes changes to his local copy, he is expected to intimate the changes that he has made to other developers, so that the necessary changes in interfaces could be uniformly carried out across all modules**.**

Problems associated with concurrent access: Possibly the most important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems can appear. Assume that only a single copy of a program module is maintained, and several developer are working on it. Two developers may simultaneously carry out changes to different functions of the same module, and while saving overwrite each other. Similar problems can occur for any other deliverable object.

Providing a stable development environment: When a project work is underway, the team members need a stable environment to make progress.

A baseline is the status of all the objects under configuration control. When any of the objects under configuration control is changed, a new baseline gets formed**.**

When any team member needs to change any of the objects under configuration control, he is provided with a copy of the baseline item. The requester makes changes to his private copy. Only

after the requester is through with all modifications to his private copy, the configuration is updated and a new baseline gets formed instantly. This establishes a baseline for others to use and depend on.

System accounting and maintaining status information: System accounting denotes keeping track of who made a particular change to an object and when the change was made.

Handling variants: Existence of variants of a software product causes some peculiar problems. Suppose you have several variants of the same module, and find that a bug exists in one of them. Then, it has to be fixed in all versions and revisions.

Configuration Management Activities

Configuration management is carried out through two principal activities:

Configuration identification: It involves deciding which parts of the system should be kept track of.

Configuration control: It ensures that changes to a system happen smoothly. Normally, a project manager performs the configuration management activity by using a configuration management tool. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the developer to change various components in a controlled manner.

Configuration identification

Project managers normally classify the objects associated with a software development into three main categories—controlled, precontrolled, and uncontrolled.

Controlled objects are those that are already under configuration control. The team members must follow some formal procedures to change them.

Precontrolled objects are not yet under configuration control, but will eventually be under configuration control.

Uncontrolled objects are not subject to configuration control. Controllable objects include both controlled and precontrolled objects.

Typical controllable objects include:

Requirements specification document

Design documents

Tools used to build the system, such as compilers, linkers, lexical analysers, parsers, etc.

Source code for each module

Test cases

Problem reports

Configuration control

Configuration control is the process of managing changes to controlled objects. The configuration control part of a configuration management system that most directly affects the day-to-day operations of developers. Configuration control allows only authorised changes to the controlled objects to occur and prevents unauthorized changes.

In order to change a controlled object such as a module, a developer can get a private copy of the module by a reserve operation. Configuration management tools allow only one person to reserve a module at any time. Once an object is reserved, it does not allow anyone else to reserve this module until the reserved module is restored.

The developer needing to change a module first makes a reserve request. After the reserve command successfully executes, a private copy of the module is created in his local directory. Then, he carries out all necessary changes on his private copy. Once has satisfactorily completes all necessary changes, the changes need to be restored in configuration management repository. However, restoring the changed module to the system configuration requires the permission of a change control board (CCB). The CCB is usually constituted from among the development team members. For every change that needs to be carried out, the CCB reviews the changes made to the controlled object and certifies several things about the change:

1. Change is well-motivated.
2. Developer has considered and documented the effects of the change.
3. Changes interact well with the changes made by other developers.
4. Appropriate people (CCB) have validated the change, e.g., someone has tested the changed code, and has verified that the change is consistent with the requirement.

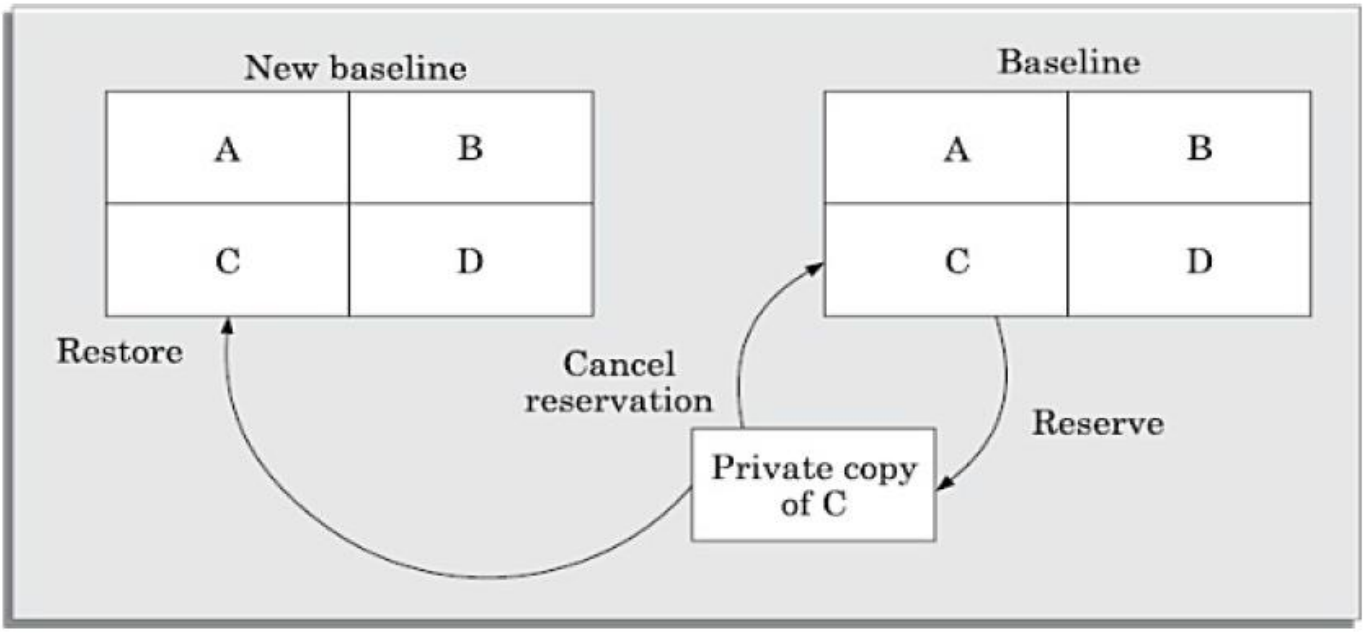


Figure 3.18: Reserve and restore operation in configuration control.

Software reliability and quality management

Reliability of a software product is an important concern for most users. Users not only want the products they purchase to be highly reliable, but for certain categories of products they may even require a quantitative guarantee on the reliability of the product before making their buying decision.

Software reliability

The reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, the reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

The most used 10 per cent instructions are often called the core1 of a program. The rest 90 per cent of the program statements are called non-core and are on the average executed only for 10 per cent of the total execution time. It therefore may not be very surprising to note that removing 60 per cent product defects from the least used parts of a system would typically result in only 3 per cent improvement to the product reliability.

Apart from this, reliability also depends upon how the product is used, or on its execution profile. If the users execute only those features of a program that are "correctly" implemented, none of the errors will be exposed and the perceived reliability of the product will be high. On

the other hand, if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low.

main reasons that make software reliability more difficult to measure than hardware reliability:

The reliability improvement due to fixing a single bug depends on where the bug is located in the code.

The perceived reliability of a software product is observer-dependent.

The reliability of a product keeps changing as errors are detected and fixed.

Hardware versus Software Reliability

An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.

Comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product are sketched in Figure .Observe that the plot of change of reliability with time for a hardware component appears like a "bath tub". For a software component the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced. The system then enters its useful life, where the rate of failure is almost constant. In contrast to the hardware products, the software product show the highest failure rate just after purchase and installation in Figure. As the system is used, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no more error correction occurs and the failure rate remains unchanged.

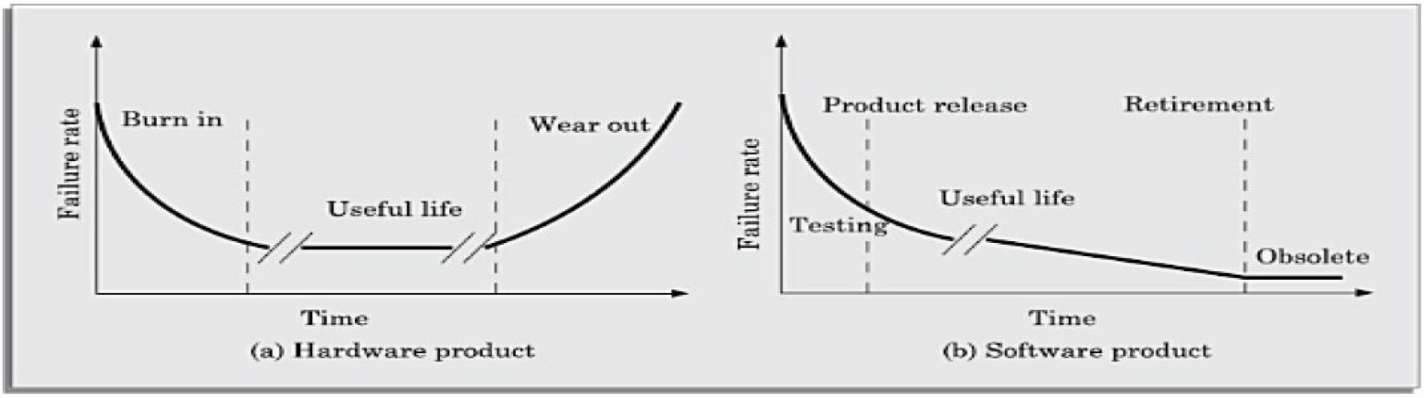


Figure 11.1: Change in failure rate of a product.

Reliability Metrics of Software Products

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the software requirements specification (SRS) document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. six metrics that correlate with reliability as follows:

Rate of occurrence of failure (ROCOF): ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.

Mean time to failure (MTTF): MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t1, t2, ..., tn. Then, MTTF can be calculated as

$$\sum_{i=1}^{n} \frac{t_{i+1} - t_i}{(n-1)}.$$

Mean time to repair (MTTR): Once failure occurs, sometime is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean time between failure (MTBF): The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours**.**

Probability of failure on demand (POFOD): Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

Unit-IV

User Interface Design

If a product is to be successful, it must exhibit good usability—a qualitative measure of the ease and efficiency with which a human can employ the functions and features offered by the high-technology product.

As technologists studied human interaction, two dominant issues arose. First, setoff golden rules were identified. These applied to all human interaction with technology products. Second, a set of interaction mechanisms were defined to enable software designers to build systems that properly implemented the golden rules. These interaction mechanisms, collectively called the graphical user interface (GUI), have eliminated some of the most egregious problems associated with human interfaces.

The golden rules

Theo Mandel [Man97] coins three golden rules:

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

Place the User in Control

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction.

Define interaction modes in a way that does not force a user into unnecessary or undesired actions

Provide for flexible interaction.

Because different users have different interaction preferences, choices should be provided But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. **Even when involved**

In a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action

Streamline interaction as skill levels advance and allow the interaction to be customized.

Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user.

The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

Reduce the User's Memory Load

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory.

Reduce demand on short-term memory.

When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults.

The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive.

When mnemonics are used to accomplish a system function, the mnemonic should be tied to the action in a way that is easy to remember,

The visual layout of the interface should be based on a real-world metaphor.

Disclose information in a progressive fashion

The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays,

(2) Input mechanisms are constrained to a limited set that is used consistently throughout the application, and

(3) Mechanisms for navigating from task to task are consistently defined and implemented. defines a set of design principles that help make the interface consistent.

Allow the user to put the current task into a meaningful context.

Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications.

A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

User interface analysis and design

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). Begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a user model, the software engineer

creates a design model, the end user develops a mental image that is often called the user's mental model or the system perception, and the implementers of the system create an implementation model.

The user model establishes the profile of end users of the system. In his introductory column on "user-centric design,"

users can be categorized as

Novices. No syntactic knowledge1 of the system and little semantic knowledge of the application or computer usage in general.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, frequent users. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's mental model (system perception) is the image of the system that end users carry in their heads. The accuracy of the description will depend upon the user's profile (e.g.,novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

The implementation model combines the outward manifestation of the computerbased system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure 11.1, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities.

(1) Interface analysis and modeling,

(2) interface design,

(3) Interface construction, and

(4) Interface validation.

The spiral shown in Figure implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral). Finally, analysis of the user environment focuses on the physical work environment.

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences. The goal of interface design is to define a set of interface objects and actions that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

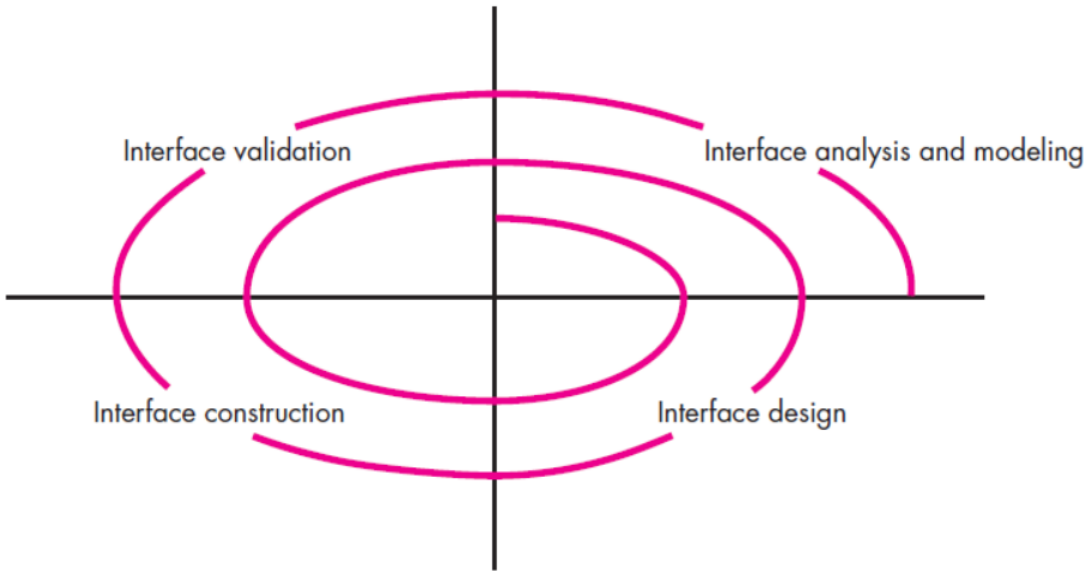
Interface validation focuses on

(1) The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

(2) The degree to which the interface is easy to use and easy to learn, and

(3) The users' acceptance of the interface as a useful tool in their work.

FIGURE 11.1
The user interface design process



Interface design steps

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.

Although many different user interface design models have been proposed, all suggest some combination of the following steps.

1. Using information developed during interface analysis define interface objects and actions.
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.

4. Indicate how the user interprets the state of the system from information provided through the interface.

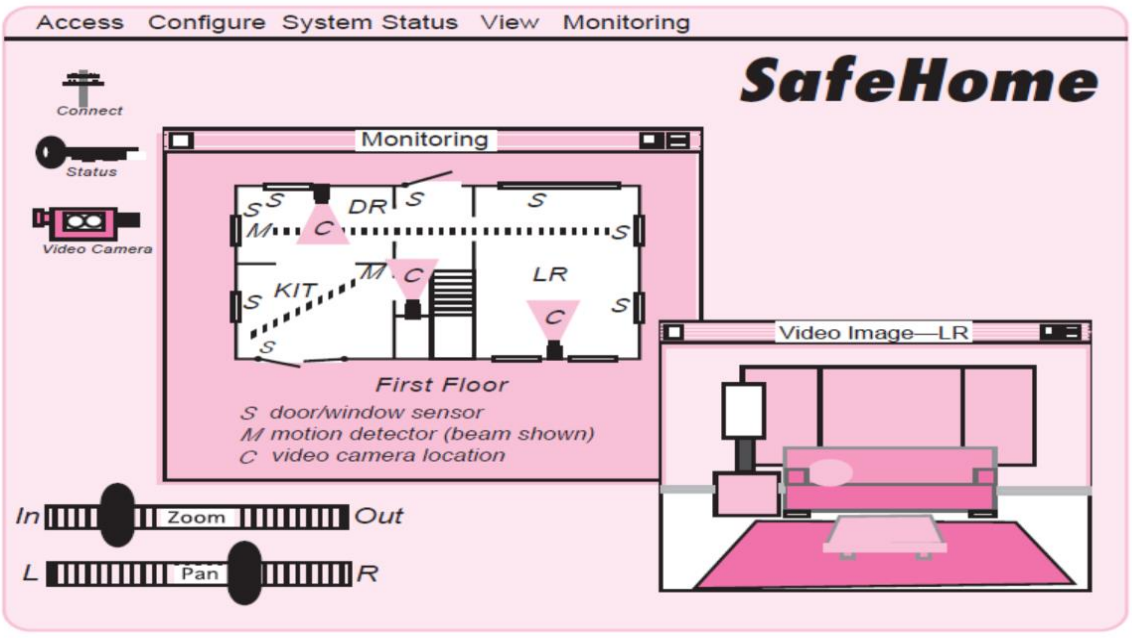
Applying Interface Design Steps

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon). The implication of this action is to create a hard-copy report. An application object represents application-specific data that are not directly manipulated as part of screen interaction.

A preliminary sketch of the screen layout for video monitoring is created.6 To invoke the video image, a video camera location icon, C, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

FIGURE 11.3 Preliminary screen layout



User Interface Design Patterns

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged a design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

Design Issues

The design of a user interface evolves, four common design issues almost always surface system response time, user help facilities, error information handling, and command labeling. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long.

Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of "user manuals" may be the only option.

A number of design issues [Rub88] must be addressed when a help facility is considered:

Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.

How will the user request help? Options include a help menu, a special function key, or a HELP command.

• How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.

How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.

How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

Error handling.

Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. There are few computer users who have not encountered an error of the form.

In general, every error message or warning produced by an interactive system should have the following characteristics:

The message should describe the problem in jargon that the user can understand.

The message should provide constructive advice for recovering from the error

The message should indicate any negative consequences of the error (e.g.,potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."

The message should be "nonjudgmental." That is, the wording should never place blame on the user.

Menu and command labeling.

The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-andpick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

Will every menu option have a corresponding command?

• What form will commands take? Options include a control sequence (e.g.,alt-P), function keys, or a typed word.

• How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?

Can commands be customized or abbreviated by the user?

• Are menu labels self-explanatory within the context of the interface?

• Are submenus consistent with the function implied by a master menu item?

Application accessibility.

As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility.

Internationalization.

Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. Localization features enable the interface to be customized for a specific market.

A variety of internationalization guidelines are available to software engineers.These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

Testing

The aim of program testing is to help realiseidentify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Consider a function taking a floating point number as argument.

Basic Concepts and Terminologies

How to test a program?

Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction. A highly simplified view of program testing is schematically shown in Figure . The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs. Unless the conditions under which software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers. For examples, software might fail for a test case only when a network connection is enabled.

Terminologies

A mistake is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity. For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation. Both these mistakes can lead to an incorrect result.

An error is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. One example of an error is a call made to a wrong function. Though the terms error, fault, bug, and defect are all used interchangeably by the program testing community.

A failure of a program essentially denotes an incorrect behavior exhibited by the program during its execution. An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program. Every failure is caused by some bugs present in the program. In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:

– The result computed by a program is 0, when the correct result is 10.

– A program crashes on an input.

– A robot fails to avoid an obstacle and collides with it.

A test case is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its execution mode. As an example, consider the different execution modes of certain text editor software. The text editor can at any time during its execution assume any of the following execution modes—edit, view, create, and display. In simple words, we can say that a test case is a set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.

An example of a test case is—[input: “abc”, state: edit, result: abc is displayed], which essentially means that the input abc needs to be applied in the edit mode, and the expected result is that the string abc would be displayed.

A test scenario is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed. An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implements the test scenarios as test cases.

A test script is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases. A test case is said to be a positive test case if it is designed to test whether the software correctly performs a required functionality. A test case is

said to be negative test case, if it is designed to test whether the software carries out something, that is not required of the system. A positive test case can be designed to check if a login system validates a user with the correct user name and password. A negative test case in this case can be a test case that checks whether the the login functionality validates and admits a user with wrong or bogus login user name or password.

A test suite is the set of all test that have been designed by a tester to test a given program.

Testability of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

A failure mode of software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.

Equivalent faults denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.

Verification versus validation

The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in software. In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different.

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase whereas validation is the process of determining whether a fully developed software conforms to its requirements specification. Thus, the objective of verification is to check if the work products produced after a phase conform to that which was input to the phase. On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer’s requirements.

The primary techniques used for verification include review, simulation, formal verification, and testing. Review, simulation, and testing are usually considered as informal verification techniques. Formal verification usually involves use of theorem proving techniques or use of automated tools such as a model checker. On the other hand, validation techniques are primarily based on product testing. Note that we have categorised testing both under program verification and validation. The reason being that unit and integration testing can be considered as verification steps where it is verified whether the code is a s per the module and module interface specifications. On the other hand, system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.

Verification does not require execution of the software, whereas validation requires execution of the software.

Verification is carried out during the development process to check if the development activities are proceeding alright, whereas validation is carried out to check if the right as required by the customer has been developed.

Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and is an important software engineering principle. The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors. Phase containment of errors can reduce the effort required for correcting bugs.

We can consider the verification and validation techniques to be different types of bug filters. To achieve high product reliability in a cost-effective manner, a development team needs to perform both verification and validation activities. The activities involved in these two types of bug detection techniques together are called the “V and V” activities.

Testing Activities

Testing involves performing the following main activities

Test suite design: The set of test cases using which a program is to be tested is designed possibly using several test case design techniques.

Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and

expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

Locate error: In this activity, the failure symptoms are analyzed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

Error correction: After the error is located during debugging, the code is appropriately changed to correct the error.

The testing activities have been shown schematically in Figure. As can be seen, the test cases are first designed; the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected. Of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

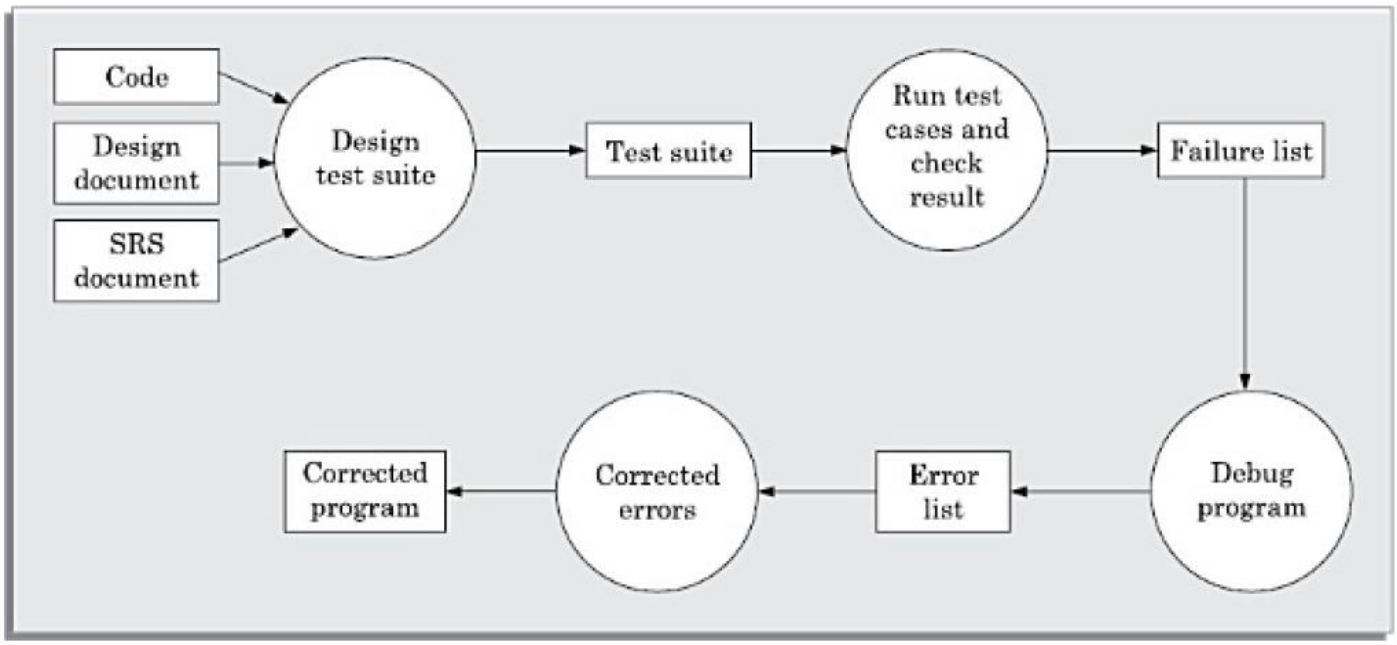


Figure 10.2: Testing process.

Testing in the Large versus Testing in the Small

A software product is normally tested in three levels or stages:

Unit testing

Integration testing

System testing

During unit testing, the individual functions (or units) of a program are tested.

Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.

After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing). Finally, the fully integrated system is tested (system testing). Integration and system testing are known as testing in the large.

Unit testing

Unit testing is undertaken after a module has been coded and reviewed. This activity is typically undertaken by the coder of the module himself in the coding phase. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed.

Driver and stub modules

In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module:

The procedures belonging to other modules that the module under test calls.

Non-local data structures that the module accesses.

A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

Stub: The role of stub and driver modules is pictorially shown in Figure .A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behavior.

Driver: A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

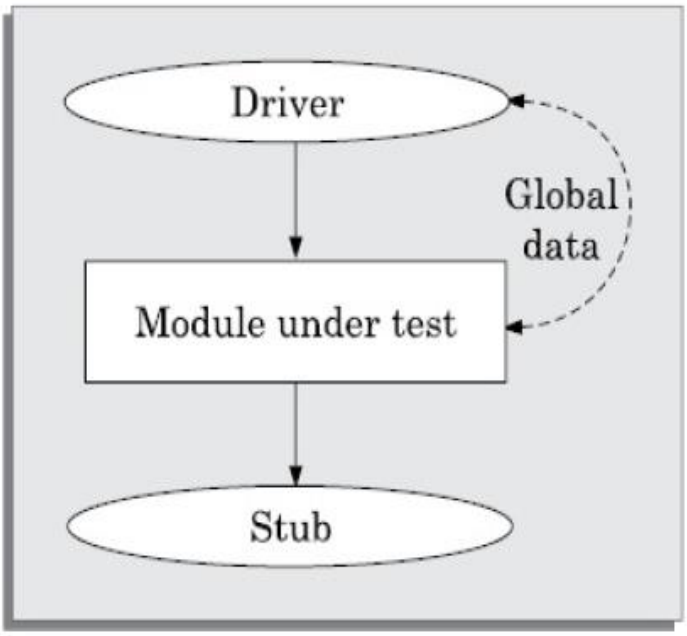


Figure 10.3: Unit testing with the help of driver and stub modules.

Black-box testing

In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black box test cases: Equivalence class partitioning Boundary value analysis

Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes.

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are

{A,B,C},then the invalid equivalence class is -{A,B,C}, where universe of possible input values.

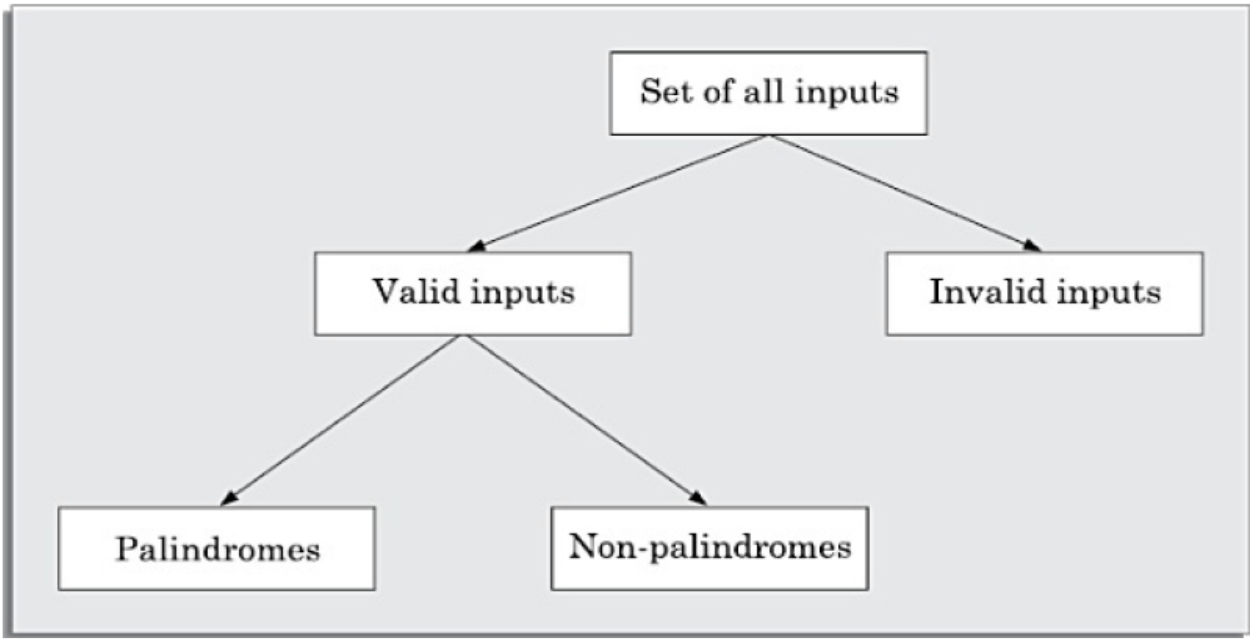
In the following, we illustrate equivalence class partitioning-based test case generation through examples.

Example. For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and the black box test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {–5,500,6000}.

Example. Design equivalence class partitioning test suite for a function z that reads a character string of size less than five characters and displays whether it is a palindrome.

Answer: The equivalence classes are the leaf level classes shown in Figure. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc,aba,abcdef}.



Boundary Value Analysis

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values no boundary value test cases can be defined. For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.

Example . For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1,5000,5001}.

Summary of the Black-box Test Suite Design Approach

Examine the input and output values of the program.

Identify the equivalence classes.

Design equivalence class test cases by picking one representative value from each equivalence class.

Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

White-box testing

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

Basic Concepts

A white-box testing strategy can either be coverage-based or faultbased

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitute the fault model of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage. We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

Stronger versus weaker testing

We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.

A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

When none of two testing strategies fully covers the program element exercised by the other, then the two are called complementary testing strategies. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.

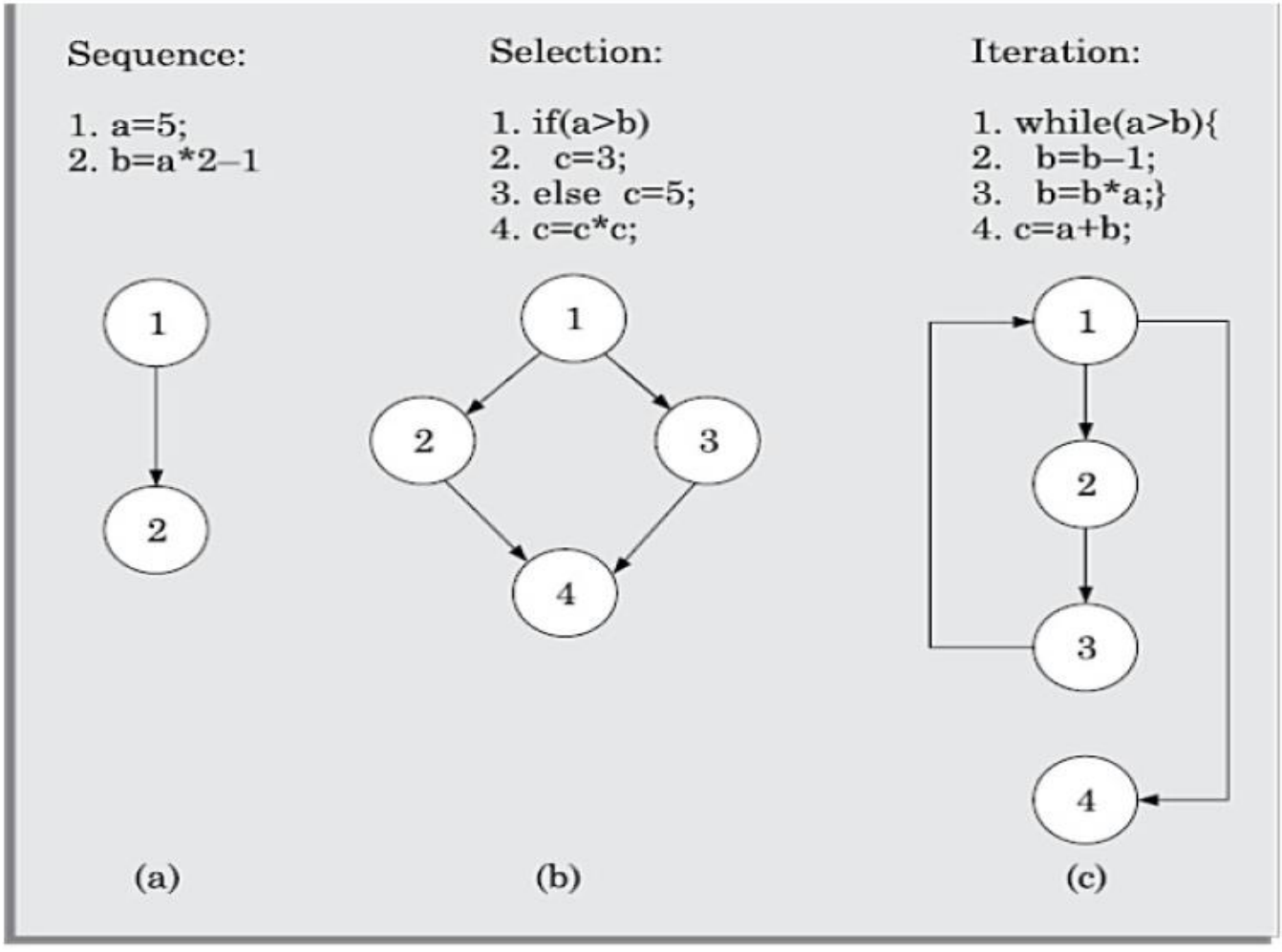


Figure 10.6: Illustration of stronger, weaker, and complementary testing strategies.

Statement Coverage

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement- coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate a test suite that achieves statement coverage.

Example . Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
int x,y;
{
1.while (x!=y)
2.if (x>y) then
3.x=x-y;
4.else y=y-x;
5.}
6.return x;
}
```

Answer: To design the test cases for the statement coverage, the conditional expression of the while statement needs to be made true and the conditional expression of the if statement needs to be made both true and false. By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y =4)}, all statements of the program would be executed at least once.

Branch Coverage

A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.

Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

Example : For the program of Example 10.11, determine a test suite to achieve branch coverage.

Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x =3, y = 4)} achieves branch coverage.

It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

Multiple Condition Coverage

In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3). A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing.

Example :Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.

Answer: Consider the following C program segment:

```
if(temperature>150 || temperature>50)
setWarningLightOn();
```

The program segment has a bug in the second component condition, it should have been temperature<50. The test suite {temperature=160, temperature=40} achieves branch coverage. But, it is not able to check that setWarningLightOn(); should not be called for temperature values within 150 and 50.

Path Coverage

A test suite achieves path coverage if it executes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow

graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control flow graph (CFG)

A control flow graph describes how the control flows through the program. A control flow graph describes the sequence in which the different instructions of a program get executed.

In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph. There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.

More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n*N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.

The CFG representation of the sequence and decision types of statements is straight forward. Please note carefully how the CFG for the loop. (iteration) construct can be drawn for iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop. That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop.Using these basic ideas, the CFG of the program.

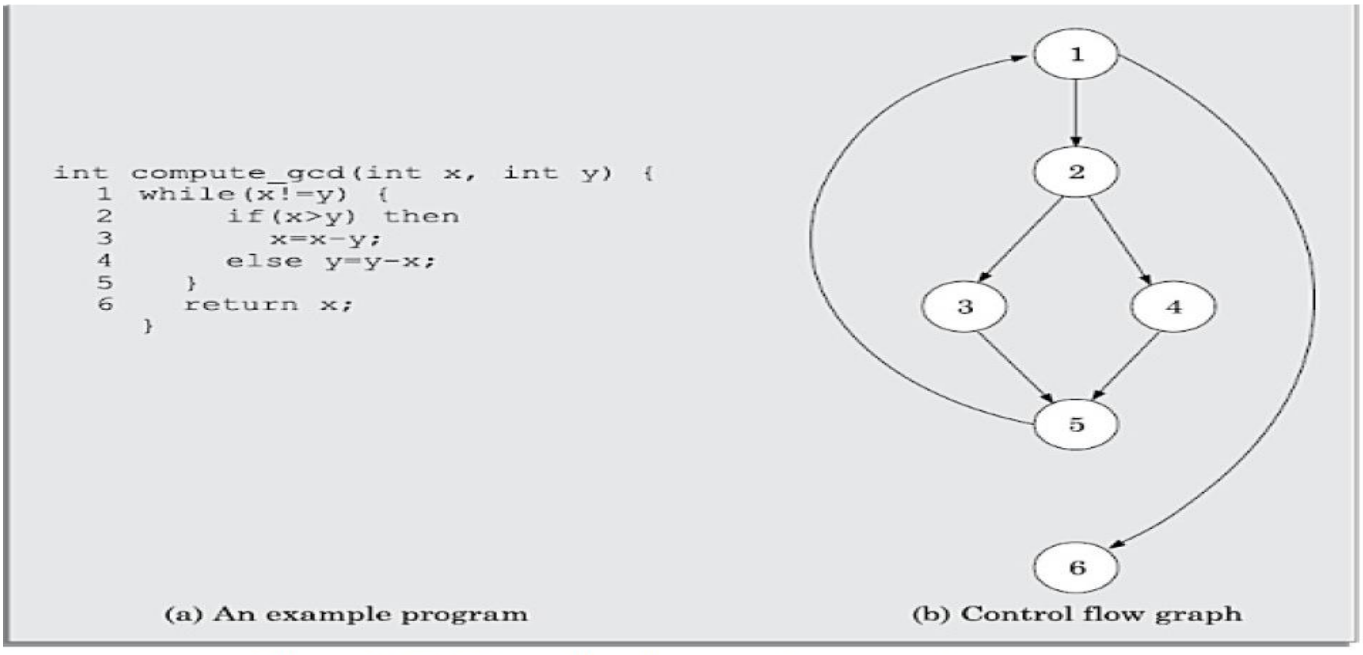


Figure 10.7: Control flow diagram of an example program.

Path

A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Please note that a program can have more than one terminal nodes when it contains multiple exits or return type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops.

If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths. This is because; any path having a new node would automatically have a new edge.

If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.

In fact, any arbitrary path of a program can be synthesized by carrying out linear operations on the basis paths. Possibly, the name basis set comes from the observation that the paths in the basis set form the “basis” for all the paths of a program.

In this context, McCabe’s cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program. McCabe’s cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe’s cyclomatic complexity is very simple to compute.

McCabe’s Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph ofa program. McCabe’s cyclomatic complexity defines an upper bound on the number of independent paths in a program.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$\mathbf{V(G) = E – N + 2}$$

where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph. For the CFG of example shown in Figure, E = 7 and N = 6.

Therefore, the value of the Cyclomatic complexity = 7 – 6 + 2 = 3.

Method 2: An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows —In this method, the cyclomatic complexity V (G) for a graph G is given by the following expression:

V(G) = Total number of non-overlapping bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area.

The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability of a program. Consider the CFG example shown in Figure. From a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computed with this method is also 2+1=3.

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program,then the McCabe's metric is equal to N + 1.

Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program.

1. Draw control flow graph for the program.
2. Determine the McCabe's metric V(G).
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. repeat

Uses of McCabe's cyclomatic complexity metric

Beside its use in path testing, cyclomatic complexity of programs has many other interesting applications such as the following:

Estimation of structural complexity of code

Complexity is a measure of the structural complexity of a program. The reason for this is that it is computed based on the code structure (number of decision and iteration constructs used). Intuitively, the McCabe's complexity metric correlates with the difficulty level of understanding a program, since one understands a program by understanding the computations carried out along all independent paths of the program.

Cyclomatic complexity of a program is a measure of the psychological complexity or the level of difficulty in understanding the program.

Estimation of testing effort:

Cyclomatic complexity is a measure of the maximum number of basis paths. Thus, it indicates the minimum number of test cases required to achieve path coverage. Therefore, the testing effort and the time required to test a piece of code satisfactorily is proportional to the cyclomatic complexity of the code. To reduce testing effort, it is necessary to restrict the cyclomatic complexity of every function to seven.

Estimation of program reliability:

Experimental studies indicate there exists a clear relationship between the McCabe's metric and the number of errors latent in the code after testing. This relationship exists possibly due to the correlation of cyclomatic complexity with the structural complexity of code. Usually the larger is the structural complexity, the more difficult it is to test and debug the code.

Data Flow-based Testing

Data flow based testing method selects test paths of a program

According to the definitions and uses of different variables in a program.Consider a program P . For a statement numbered S of P , let

DEF(S) = {X /statement S contains a definition of X } and

USES(S)= {X /statement S contains a use of X }

For the statement S: a=b+c;, DEF(S)={a}, USES(S)={b, c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X .

All definitions criterion is a test coverage criterion that requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the tes **All use criterion** requires that all uses of a definition should be covered. Clearly, all-uses criterion is stronger than all-definitions criterion.

An even stronger criterion is all definition-use-paths criterion, which requires the coverage of all possible definition-use paths that either are cycle-free or have only simple cycles. A simple cycle is a path in which only the end node and the start node are the same.

Mutation Testing

All white-box testing strategies that we have discussed so far, are coverage-based testing techniques. In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies. After the initial testing is complete, mutation testing can be taken up.

The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated program and the change effected is called a mutant. An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of simple errors. A mutation operator makes specific changes to a program.

A mutated program is tested against the original test suite of the program. If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant.

An important advantage of mutation testing is that it can be automated to a great extent. The process of generation of mutants can be automated by predefining a set of primitive changes that can be applied to the program. Mutation testing involves generating a large number of

mutants. Also each mutant needs to be tested with the full test suite. Obviously therefore; mutation testing is not suitable for manual testing.

Debugging

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed.

Debugging Approaches

Brute force method

This is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

Backtracking

This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method

In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis**.**

Program slicing

This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements. However, the search space is reduced by defining slices. A slice of a program for a particular variable and at a particular statement is the set of source

lines preceding this statement that can influence the value of that variable. Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity and experience. The following are some general guidelines for effective debugging:

Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.

Debugging may sometimes even require full redesign of the system. Insuch cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.

One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

Program analysis tools

A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, adequacy of testing, etc. We can classify various program analysis tools into the following two broad categories:

Static analysis tools

Dynamic analysis tools

Static Analysis Tools

Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyse the source code to compute certain metrics characterising the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions. These also check the conformance of the code with the prescribed coding standards. In this context, it displays the following analysis results:

To what extent the coding standards have been adhered to?

Whether certain programming errors such as uninitialized variables, mismatch between actual and formal parameters, variables that are declared but never used, etc., exist? A list of all such errors is displayed.

A major practical limitation of the static analysis tools lies in their inability to analyse run-time information such as dynamic memory references using pointer variables and pointer arithmetic, etc.

Static analysis tools often summarise the results of analysis of every function in a polar chart known as Kiviat Chart. A Kiviat Chart typically shows the analysed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead's metrics, etc.

Dynamic Analysis Tools

Dynamic program analysis tools can be used to evaluate several program characteristics based on an analysis of the run time behaviour of a program.These tools usually record and analyse the actual behaviour of a program while it is being executed. A dynamic program analysis tool (also called a dynamic analyser) usually collects execution trace information by instrumenting the code. Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program. The instrumented code when executed, records the behaviour of the software for different test cases.

After software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the coverage that has been achieved by the complete test suite for the program.

Normally the dynamic analysis results are reported in the form of a histogram or pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily to provide evidence that thorough testing has been carried out.

Integration testing

Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). The objective of integration testing is to check whether the different modules of a program interface with each other properly.

During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realise the full system. After each integration step, the partially integrated system is tested.

A structure chart (or module dependency graph) specifies the order in which different modules call each other. Thus, by examining the structure chart, the integration plan can be developed. Any one (or a mixture) of the following approaches can be used to develop the test plan:

1.Big-bang approach to integration testing

2.Top-down approach to integration testing

3.Bottom-up approach to integration testing

4.Mixed (also called sandwiched) approach to integration testing

Big-bang approach to integration testing

Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words, all the unit tested modules of the system are simply linked together and tested. However, this technique can meaningfully be used only for very small systems. The main problem with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules. Therefore, debugging errors reported during big-bang integration testing are very expensive to fix.As a result, big-bang integration testing is almost never used for large programs.

Bottom-up approach to integration testing

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.

The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily. The test cases must be carefully chosen to exercise the interfaces in all possible manners

In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems. The principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level.

Top-down approach to integration testing

Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing

The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In topdown approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. Therefore, this is one of the most commonly used integration testing approaches. In this approach, both stubs and drivers are required to be designed.

Phased versus Incremental Integration Testing

Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules

can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

In incremental integration testing, only one new module is added to the partially integrated system each time.

In phased integration, a group of related modules are added to the partial system each time.

Obviously, phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system while using the incremental testing approach since the errors can easily be traced to the interface of the recently integrated module.

System testing

After all the units of a program have been integrated together and tested,system testing is taken up. System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation is immaterial.

There are essentially three main kinds of system testing depending on who carries out testing:

1**. Alpha Testing**: Alpha testing refers to the system testing carried out by the test team within the developing organisation.

2. **Beta Testing**: Beta testing is the system testing performed by a select group of friendly customers.

3. **Acceptance Testing**: Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

The system test cases can be classified into functionality and performance test cases.

Before a fully integrated system is accepted for system testing, smoke testing is performed. Smoke testing is done to check whether at least the main functionalities of the software are working properly.

Unless the software is stable and at least the main functionalities are working satisfactorily, system testing is not undertaken.

The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests, on the other hand, test the conformance of the system with the non-functional requirements of the system.

Smoke Testing

Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail. The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing. For smoke testing, a few test cases are designed to check whether the basic functionalities are working. For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned.

Performance Testing

Performance testing is an important type of system testing. Performance testing is carried out to check whether the system meets the nonfunctional requirements identified in the SRS document.

There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document.All performance tests can be considered as black-box tests.

Stress testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the

Capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc., are tested beyond the designed capacity. For example, suppose an operating system is supposed to support fifteen concurrent transactions, then the system is stressed by attempting to initiate fifteen or more transactions simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that under normal circumstances operate below their maximum capacity but may be severely stressed at some peak demand hours.

Volume testing

Volume testing checks whether the data structures (buffers, arrays,queues, stacks, etc.) have been designed to successfully handle extraordinary situations. For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.

Configuration testing

Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements.Sometimes systems are built to work in different configurations for different users. For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing. The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.

Compatibility testing

This type of testing is required when the system interfaces with external systems (e.g., databases, servers, etc.). Compatibility aims to check whether the interfaces with the external systems are performing as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression testing

This type of testing is required when software is maintained to fix some bugs or enhance functionality, performance, etc.

Recovery testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance testing

This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation testing

It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance of this requirement.

Usability testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested. A GUI being just being functionally correct is not enough.

Security testing

Security testing is essential for software that handle or process confidential data that is to be gurarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers. Over the last few years, a large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports, etc.

Error Seeding

Sometimes customers specify the maximum number of residual errors that can be present in the delivered software. These requirements are often expressed in terms of maximum number of allowable errors per line of source code. The error seeding technique can be used to estimate the number of residual errors in a software.

Error seeding, as the name implies, it involves seeding the code with some known errors. In other words, some artificial errors are introduced (seeded) into the program. The number of these seeded errors that are detected in the course of standard testing is determined. These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program**.**

The number of errors remaining in the product. The effectiveness of the testing strategy.

Let N be the total number of defects in the system, and let n of these defects be found by testing.Let S be the total number of seeded defects, and let s of these defects be found during testing. Therefore, we get:

$$\frac{n}{N} = \frac{s}{S}$$

Or

$$N = S \times \frac{n}{s}$$

Defects still remaining in the program after testing can be given by:

$$N - n = n \times \frac{(S-1)}{s}$$

Error seeding works satisfactorily only if the kind seeded errors and their frequency of occurrence matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in software. To some extent, the different categories of errors those are latent and their frequency of occurrence can be estimated by analyzing historical data collected from similar projects.

Unit-III

Design Concepts

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product .Design principles establish an overriding philosophy that guides you in the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design within the context of software engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction.

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure .The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

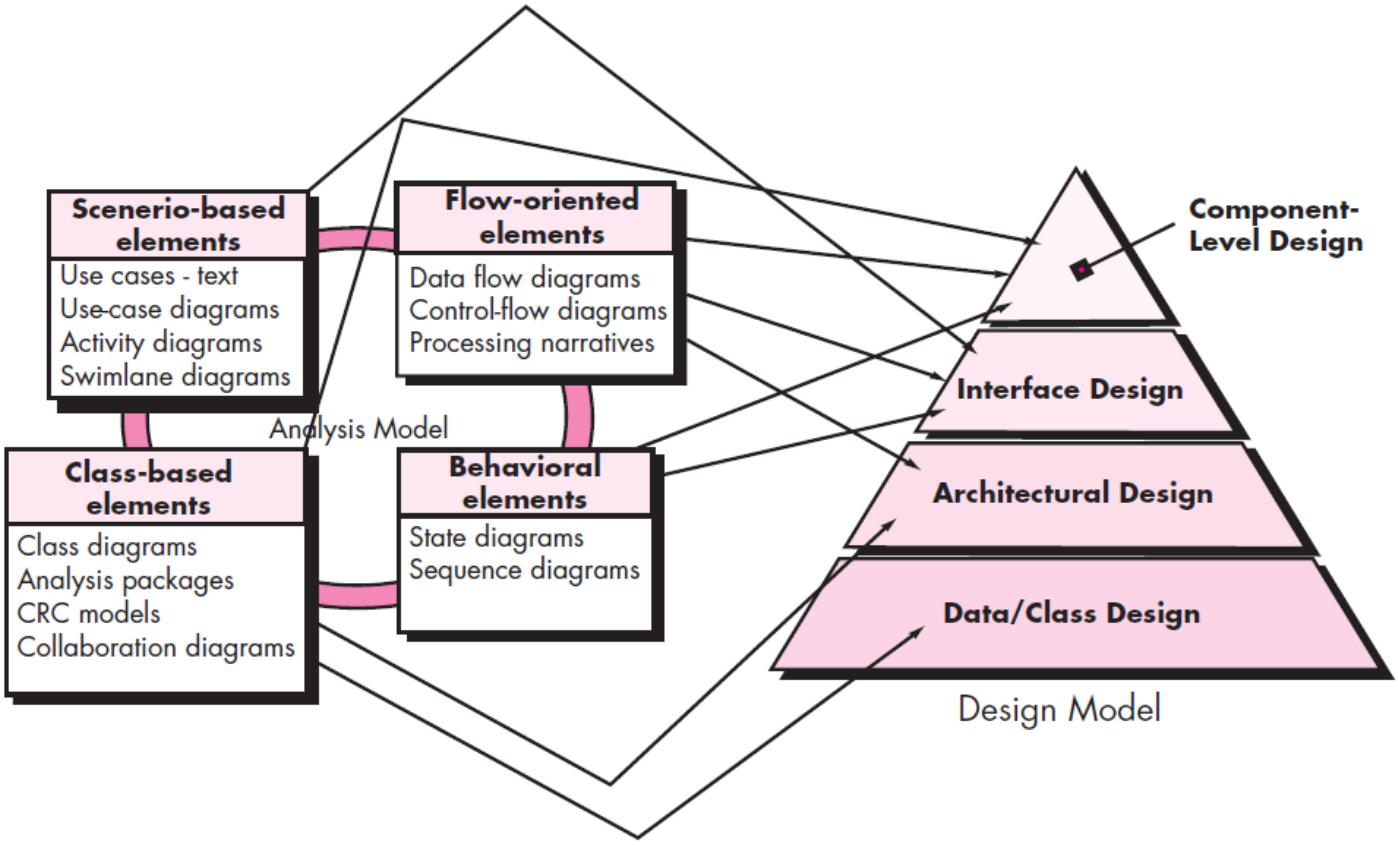
The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design. The importance of software design can be stated with a single word—quality. Design is the place

where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

FIGURE 8.1 Translating the requirements model into the design model



The design process

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed. McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design.

The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.

The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines. In order to evaluate the quality of a design representation, and other members of the software team must establish technical criteria for good design.

1. A design should exhibit an architecture that

(1) Has been created using recognizable architectural styles or patterns,

(2) Is composed of components that exhibit good design characteristics and

(3) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture,interfaces, and components

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes. Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation.

Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

Performance is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

Supportability combines the ability to extend the program (extensibility),adaptability, serviceability—these three attributes represent a more common term, maintainability—and in

addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design

Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a topdown manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods,[Jac04]), model-driven development and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

Methods have a number of common characteristics:

(1) A mechanism for the translation of the requirements model into a design representation,

(2) A notation for representing functional components and their interfaces,

(3) Heuristics for refinement and partitioning, and

(4) Guidelines for quality assessment.

Design concepts

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps answer the following questions:

What criteria can be used to partition software into individual components?

• How is function or data structure detail separated from a conceptual representation of the software?

• What uniform criteria define the technical quality of a software design?

Abstraction

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

A data abstraction is a named collection of data that describes a data object.

Architecture

In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to solve common design problems.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design.

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Structural models

Represent architecture as an organized collection of program components.

Framework models

Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models

Address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models

Focus on the design of the business or technical process that the system must accommodate. Finally,

Functional models

Can be used to represent the functional hierarchy of a system

Patterns

A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work,
(2) Whether the pattern can be reused (hence, saving design time), and
(3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller and therefore more manageable pieces, a problem takes less effort and time to solve.

For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements**.**

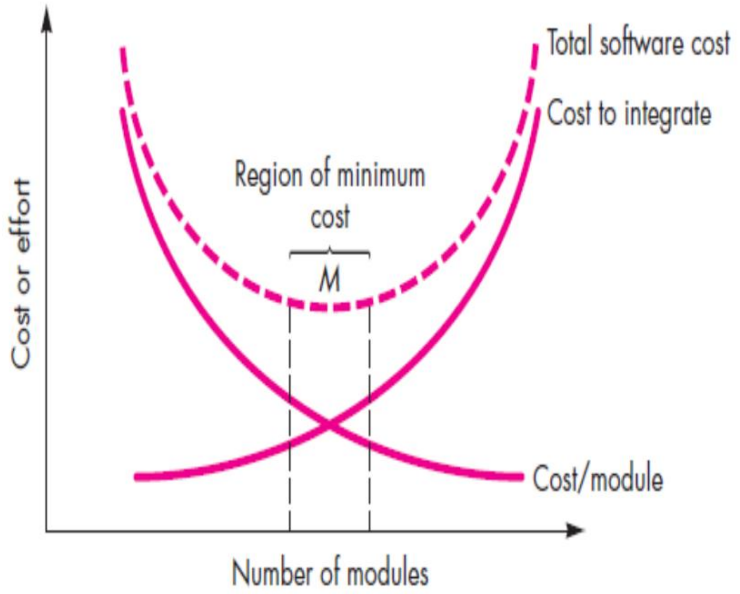
It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Referring to Figure the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grow. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

A design so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

FIGURE 8.2

Modularity and software cost



Information Hiding

Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification

are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept described in. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Refinement

A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. Begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

Aspects

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants**.**

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account. An aspect is a representation of a crosscutting concern.

It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components [Ban06]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

Refactoring

An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion.

Object-Oriented Design Concepts

Design Classes

The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Five different types of design classes, each representing a different layer of the design architecture, can be developed.

User interface classes define all abstractions that are necessary for humancomputer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.

Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

Process classes implement lower-level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.

System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class.

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the

class name) to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

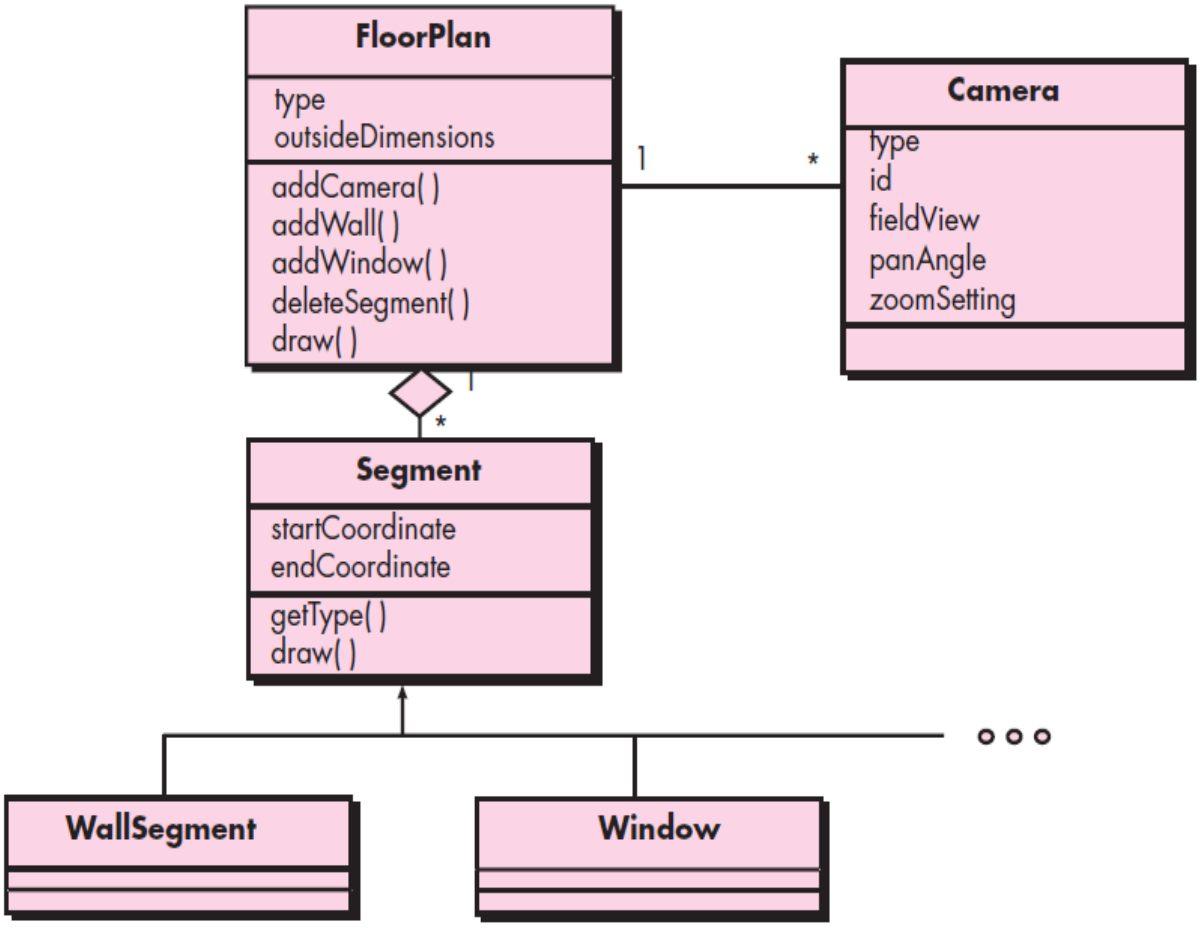
Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class VideoClip might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes,

FIGURE 8.3

Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



Architectural Design

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements.

Software architecture

What Is Architecture

The architecture is not the operational software. Rather, it is a representation that enables to

(1) Analyze the effectiveness of the design in meeting its stated requirements,

(2) Consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) Reduce the risks associated with the construction of the software

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Why Is Architecture Important?

Three key reasons that software architecture is important:

Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together.

Architectural Descriptions

The implication is that different stakeholders will see architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems,, with the following objectives:

(1) to establish a conceptual framework and vocabulary for use during the design of software architecture,

(2) to provide detailed guidelines for representing an architectural description, and

(3) to encourage sound architectural design practices.

The IEEE standard defines an architectural description (AD) as "a collection of products to document architecture." The description itself is represented using multiple views, where each view is "a representation of a whole system from the perspective of a related set of concerns." A view is created according to rules and conventions defined in a viewpoint—"a specification of the conventions for constructing and using a view".

Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

Architectural genres

The architectural genre will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, genre implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories.

Software Architecture Grady Booch suggests the following architectural genres for software-based systems.

Artificial intelligence—Systems that simulate or augment human cognition, locomotion, or other organic processes.

• **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.

• **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.

• **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts

Devices—Systems that interact with the physical world to provide some point service for an individual.

• **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.

• **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.

• **Games**—Systems that provide an entertainment experience for individuals or groups.

• **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.

• **Industria**l—Systems that simulate or control physical processes.

• **Legal**—Systems that support the legal industry.

• **Medical**—Systems that diagnose or heal or that contribute to medical research.

Operating systems—Systems that sit just above hardware to provide basic software services.

• **Platforms**—Systems that sit just above operating systems to provide advanced services.
• **Scientific**—Systems that are used for scientific research and applications.
• **Tools**—Systems that are used to develop other systems.
• **Transportation**—Systems that control water, ground, air, or space vehicles.
• **Utilities**—Systems that interact with other software to provide some point service.

ARCHITECTURAL STYLES

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

(1) A set of components (e.g., a database, computational modules) that perform a function required by a system;

(2) A set of connectors that enable "communication, coordination and cooperation" among components;

(3) Constraints that define how components can be integrated to form the system; and

(4) Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.

An architectural pattern, like an architectural style, imposes a transformation on the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;

(2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

(3) Architectural patterns tend to address specific behavioral issues within the context of the architecture Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

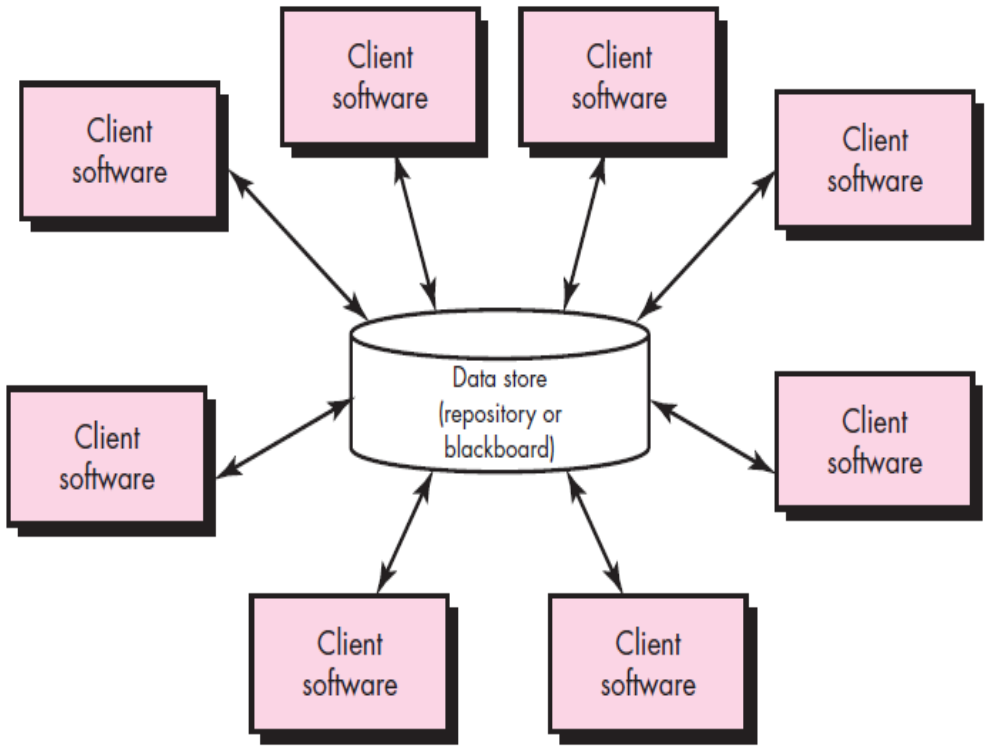
A Brief Taxonomy of Architectural Styles

Data-centered architectures.

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote integrability.That is, existing components can be changed and new client components added to the architecture without concern about other clients.

FIGURE 9.1
Data-centered architecture



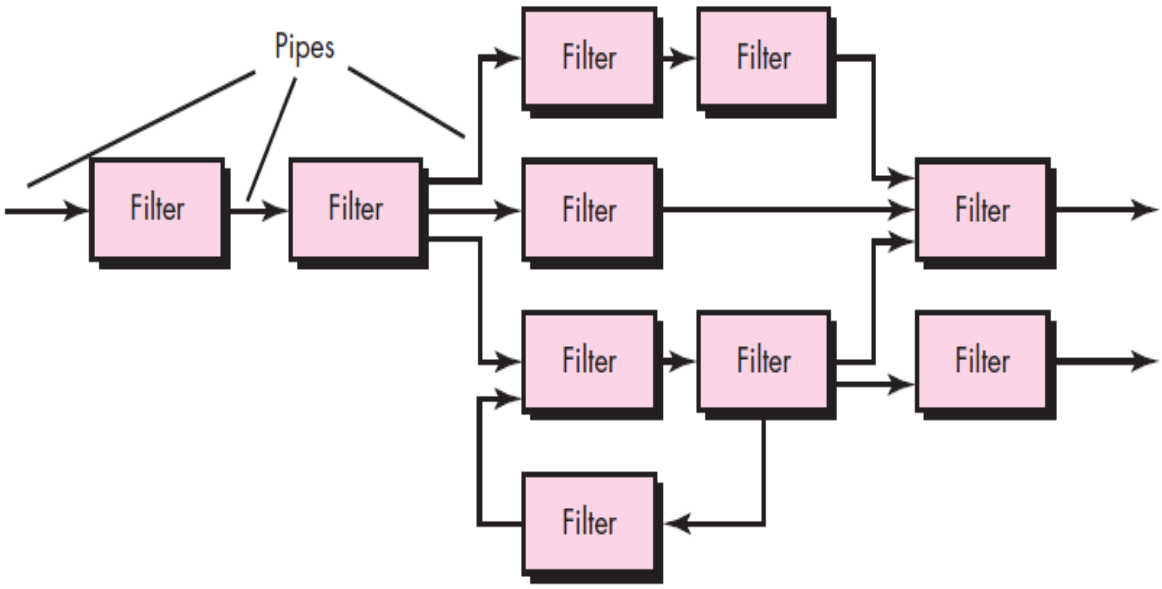
Data-flow architectures.

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern Figure has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform.

FIGURE 9.2

)ata-flow
ırchitecture



Pipes and filters

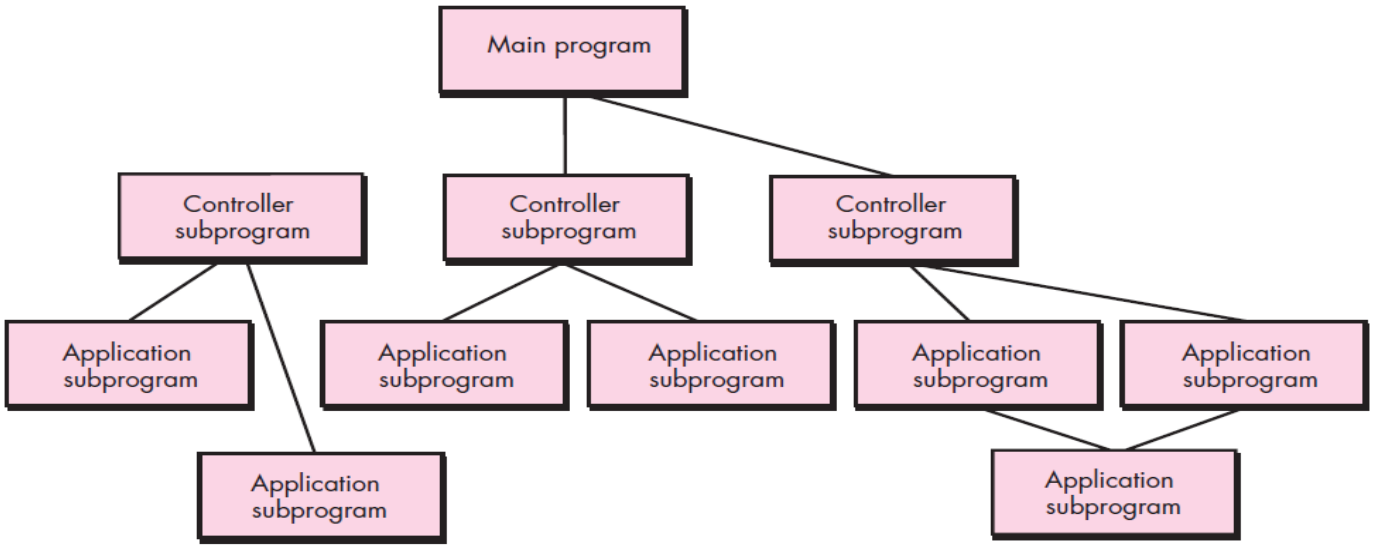
Call and return architectures.

This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category.

Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure illustrates architecture of this type.

Remote procedure call architectures. The components of main program/subprogram architecture are distributed across multiple computers on a network.

FIGURE 9.3 Main program/subprogram architecture



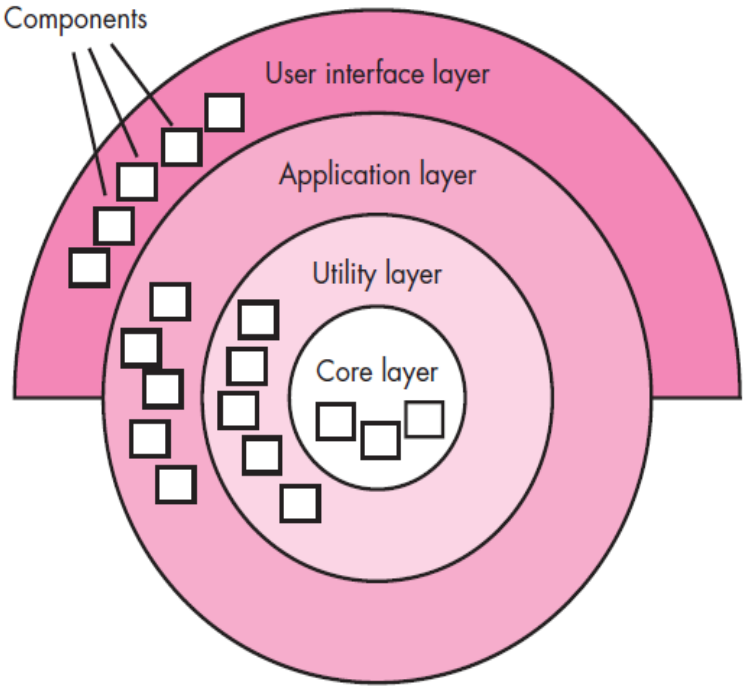
Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered architectures.

The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style can be combined with a data-centered architecture in many database applications.

FIGURE 9.4
Layered architecture



Architectural Patterns

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Organization and Refinement

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy.

How do components transfer control within the system?

How is control shared among components? What is the control topology?

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically?

What is the mode of data transfer?

Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?

How do functional components interact with data components?

Are data components passive or active?

How do data and control interact within the system?

Architectural design

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An archetype is an abstraction that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure.

Referring to the figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.

Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Peer-level systems—those systems that interact on a peer-to-peer basis.Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

FIGURE 9.5

Architectural context diagram

Source: Adapted from [Bos00].

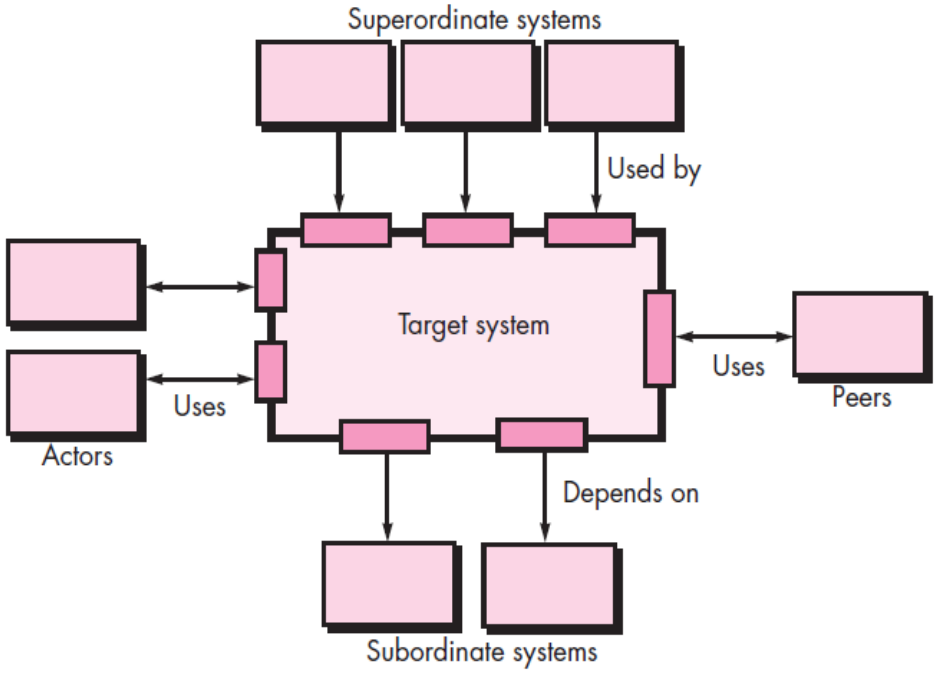
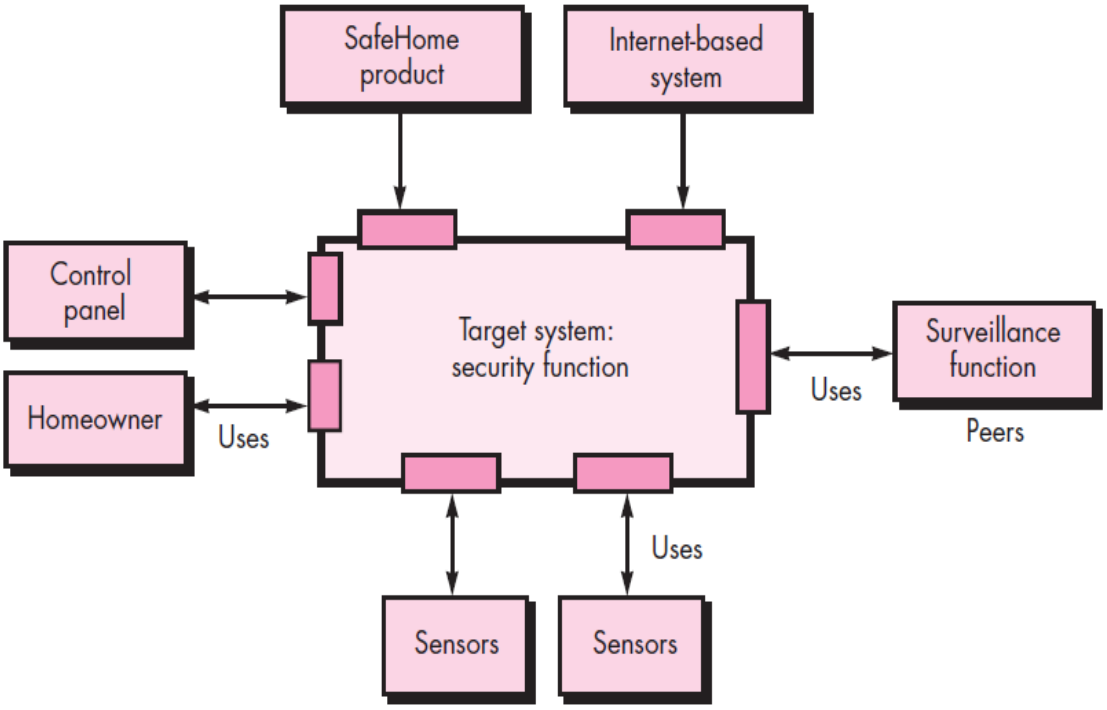


FIGURE 9.6

Architectural context diagram for the *SafeHome* security function



Defining Archetypes

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the SafeHome home security function, might define the following archetypes.

Node. Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of

(1) various sensors and

(2) a variety of alarm (output) indicators.

Detector. An abstraction that encompasses all sensing equipment that feeds information into the target system,

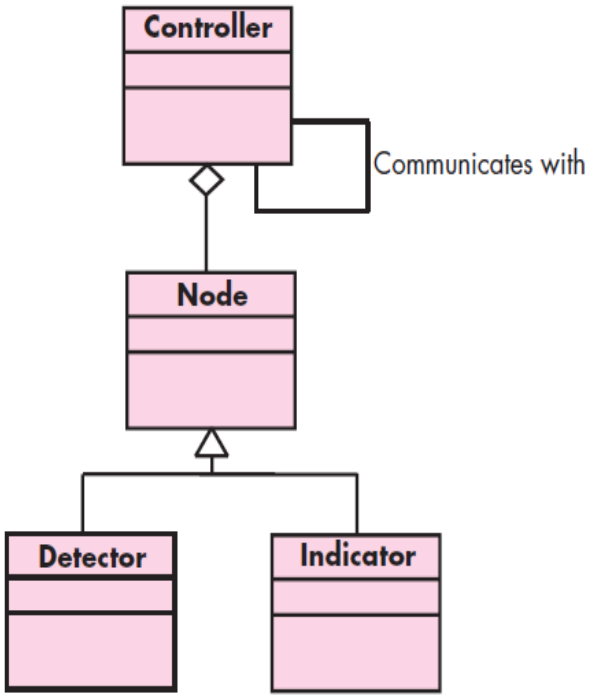
Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

Controller. An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

FIGURE 9.7

UML relationships for *SafeHome* security function archetypes

Source: Adapted from [Bos00].



Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model. These analysis classes represent entities within the application domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.

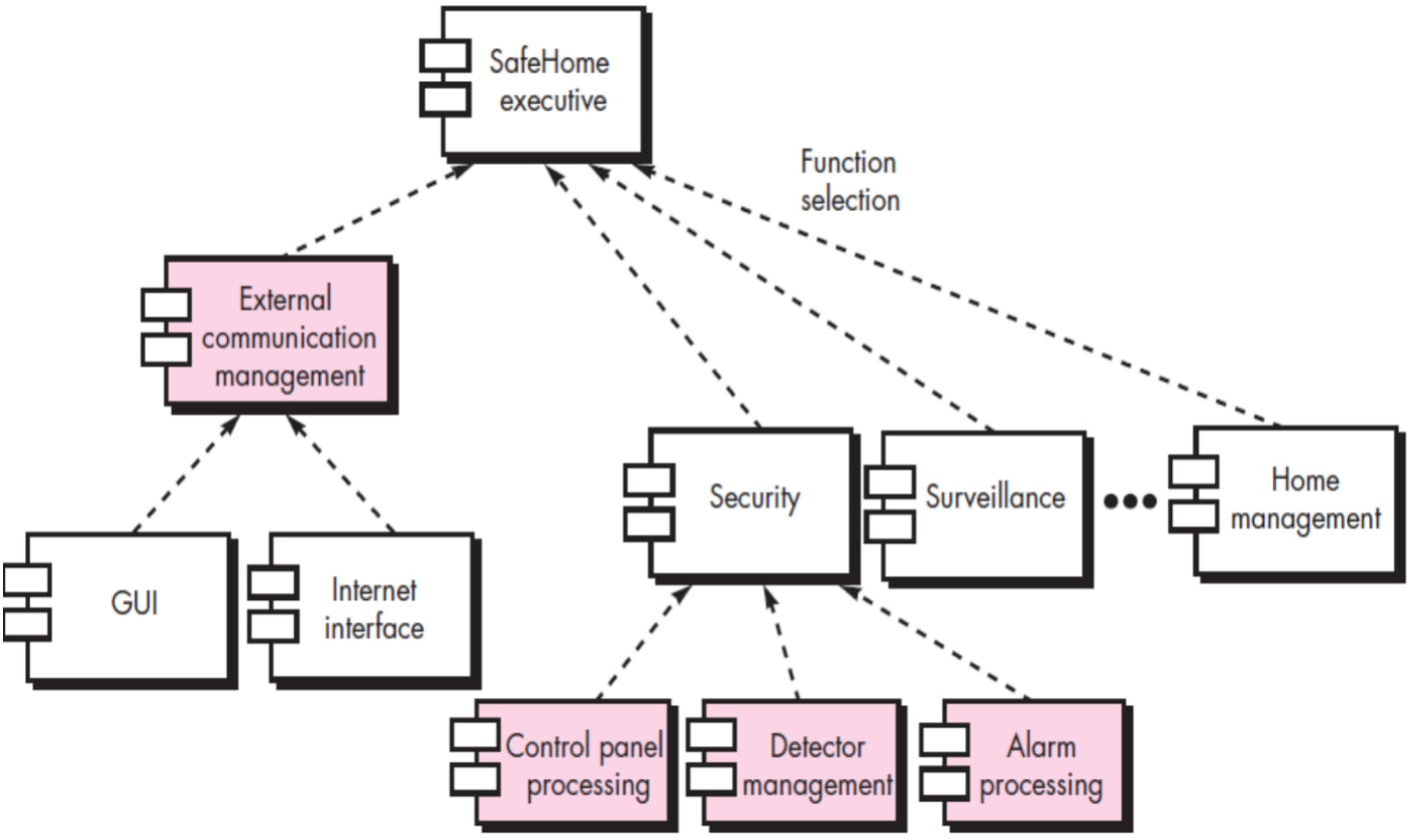
Define the set of top-level components that address the following functionality:

External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

- **Control panel processing**—manages all control panel functionality.
- **Detector management**—coordinates access to all detectors attached to the system.
- **Alarm processing**—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall SafeHome architecture. Design classes would be defined for each.

FIGURE 9.8 Overall architectural structure for *SafeHome* with top-level components



Component-level design

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low.

It is possible to represent the component-level design using a programming language. In essence, the program is created using the architectural design model as a guide. An alternative approach is to represent the component-level design using some intermediate representation that can be translated easily into source code. Regardless of the mechanism that is used to represent the component-level design, the data structures, interfaces, and algorithms defined should conform to a variety of well-established design guidelines that help you to avoid errors as the procedural design evolves.

What i s a component?

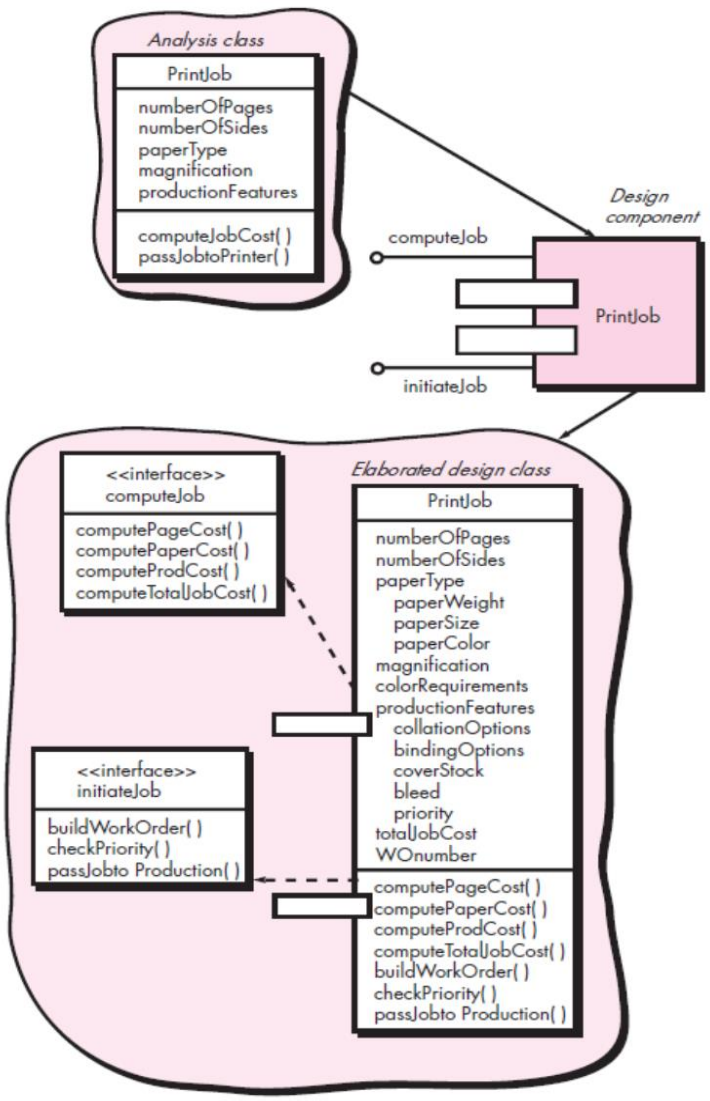
A component is a modular building block for computer software. More formally, the OMG Unified Modeling Language Specification [OMG03a] defines a component as ". . . a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined.

To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called PrintJob was derived. The attributes and operations defined during analysis are noted at the top of Figure. During architectural design, PrintJob is defined as a component within the software architecture and is represented using the shorthand UML notation shown in the middle right of the figure. Note that PrintJob has two interfaces, computeJob, which provides job costing capability, and initiateJob, which passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

FIGURE 10.1
Elaboration of a design component



In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

A traditional component, also called a module, resides within the software architecture and serves one of three important roles:

(1) A control component that coordinates the invocation of all other problem domain components,

(2) A problem domain component that implements a complete or partial function that is required by the customer, or

(3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, traditional software components are derived from the analysis model. In this case, however, the data flow-oriented element of the analysis model

serves as the basis for the derivation. Each transform (bubble) represented at the lowest levels of the data flow diagram is mapped into a module hierarchy.

A set of data flow diagram would be derived during requirements modeling. Assume that these are mapped into an architecture shown in Figure . Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the PrintJob class. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

FIGURE 10.2
Structure chart for a traditional system

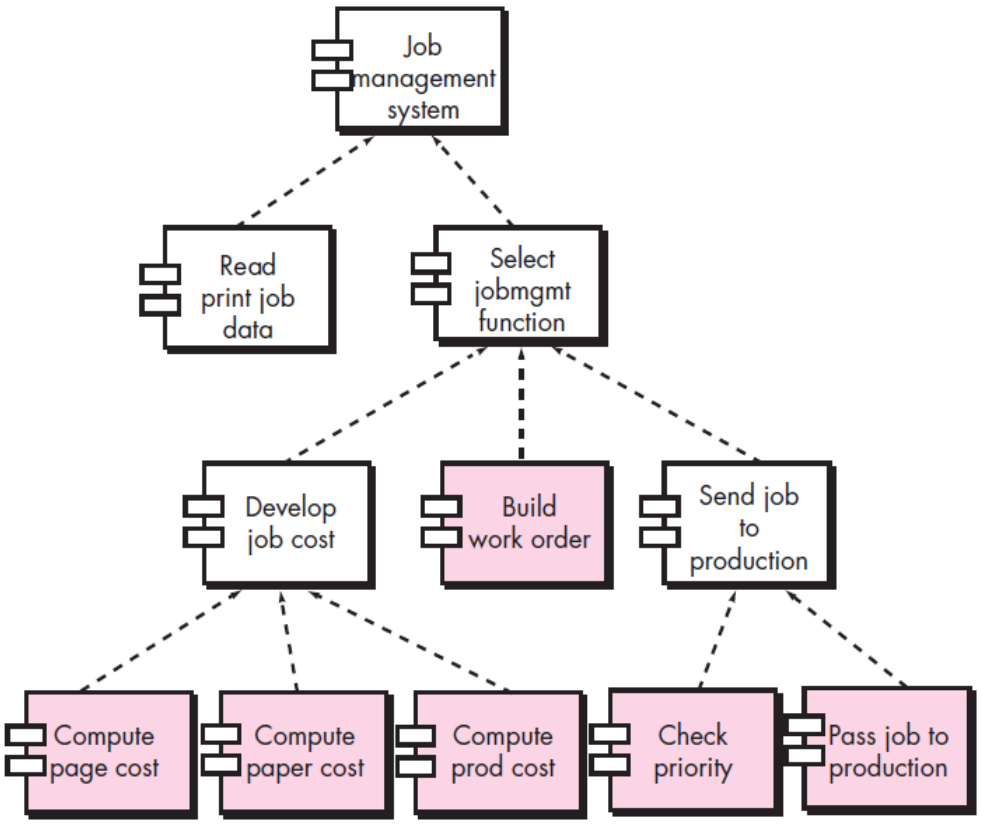
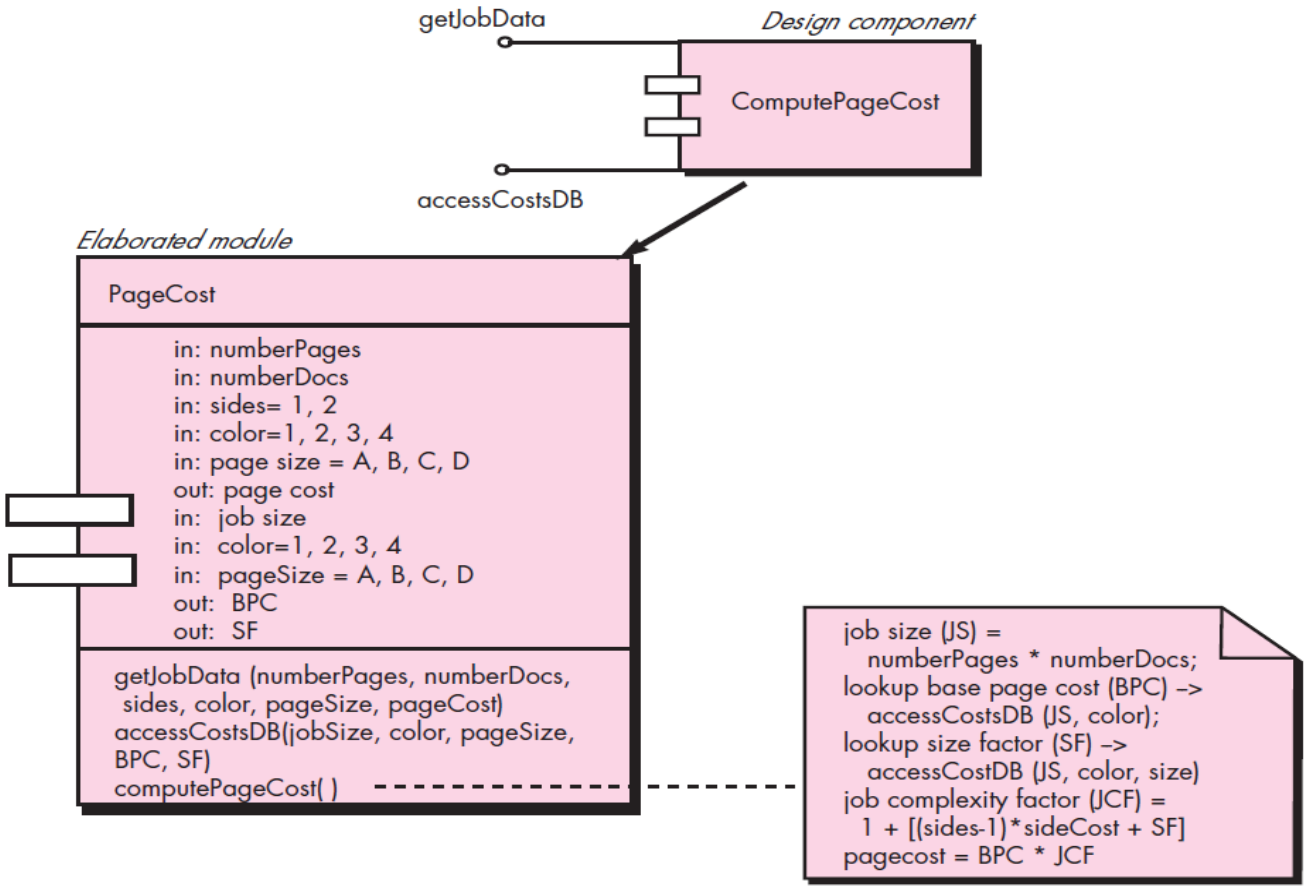


Figure 10.3 represents the component-level design using a modified UML notation. The ComputePageCost module accesses data by invoking the module getJobData, which allows all relevant data to be passed to the component, and a database interface, accessCostsDB, which enables the module to access a database that contains all printing costs. As design continues, the ComputePageCost module is elaborated to provide algorithm detail and interface detail. Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component.

FIGURE 10.3 Component-level design for *ComputePageCost*



A Process-Related View

Create a new component based on specifications derived from the requirements model. As the software architecture is developed, choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available.

Designing class -based components

When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model.

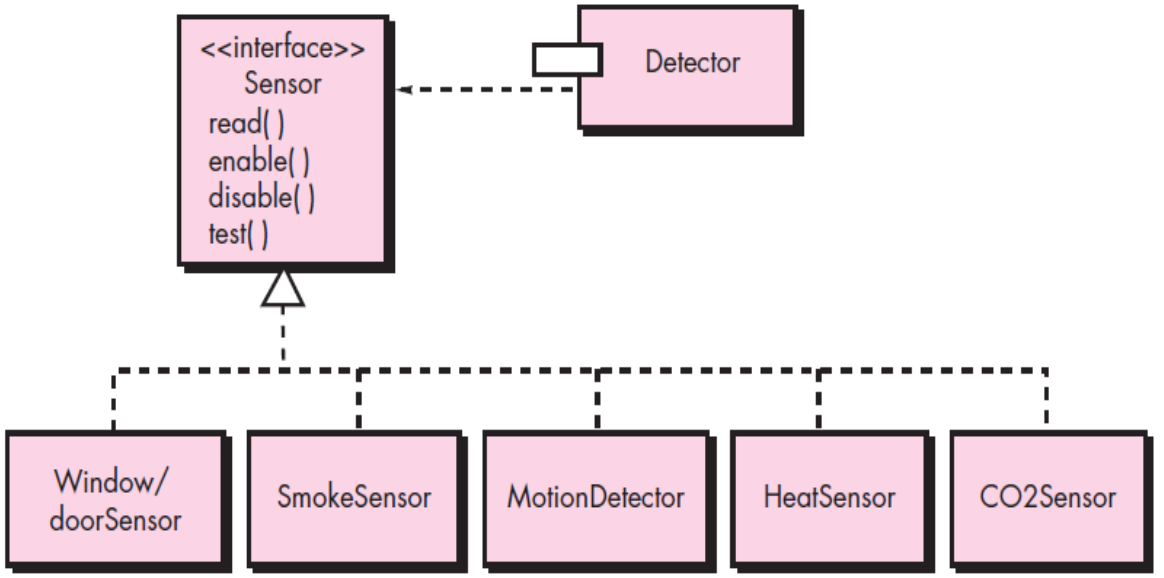
Basic Design Principles

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.

The Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification" . This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal modifications to the component itself. The

sensor interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the Detector class. The OCP is preserved

FIGURE 10.4
Following the OCP



The Liskov Substitution Principle (LSP). "Subclasses should be substitutable for their base classes" [Mar00]. This design principle, originally proposed by Barbara Liskov suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a precondition that must be true before the component uses a base class and a postcondition that should be true after the component uses a base class.When you create derived classes, be sure they conform to the pre-and postconditions.

Dependency Inversion Principle (DIP). "Depend on abstractions. Do not depend on concretions" . As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP). "Many client-specific interfaces are better than one general purpose interface" . There are many instances in which multiple client components use the operations provided by a server class. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

The Release Reuse Equivalency Principle (REP). "The granule of reuse is the granule of release". When classes or components are designed for reuse, there is an implicit contract that is

established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

The Common Closure Principle (CCP). "Classes that change together belong together." Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). "Classes that aren't reused together should not be grouped together" .When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

Component-Level Design Guidelines

These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design. Ambler suggests the following guidelines.

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

Choose to use stereotypes to help identify the nature of components at the detailed design level. For example, <<infrastructure>> might be used to identify an infrastructure component, <<database>> could be used to identify a database that services one or more design classes or the entire system; <<table>> can be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP). However, unfettered representation of interfaces tends to complicate component diagrams.

(1) Lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex;

(2) For consistency, interfaces should flow from the left-hand side of the component box;

(3) Only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.

Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom to top (base classes). In addition, component interdependencies should be represented via interfaces, rather than by representation of a component-to component dependency.

Cohesion. Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

Different types of cohesion

Functional. Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer,but lower layers do not access higher layers.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing.

Coupling .Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.

Define the following coupling categories:

Content coupling.Occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding—a basic design concept.

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary, common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

Control coupling. Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.

Stamp coupling.Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

Data coupling.Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

Routine call coupling.Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it doesincrease the connectedness of a system.

Type use coupling. Occurs when component A uses a data type defined in component BIf the type definition changes, every component that uses the definition must also change.

Inclusion or import coupling. Occurs when component A imports or includes a package or the content of component B.

External coupling. Occurs when a component communicates or collaborates with infrastructure components. Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system. Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Conducting component-level design

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain.

Using the requirements and architectural model, each analysis class and architectural component is elaborated.

Step 2. Identify all design classes that correspond to the infrastructure domain.

These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. As we have noted earlier, classes and components in this category include GUI components operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components.

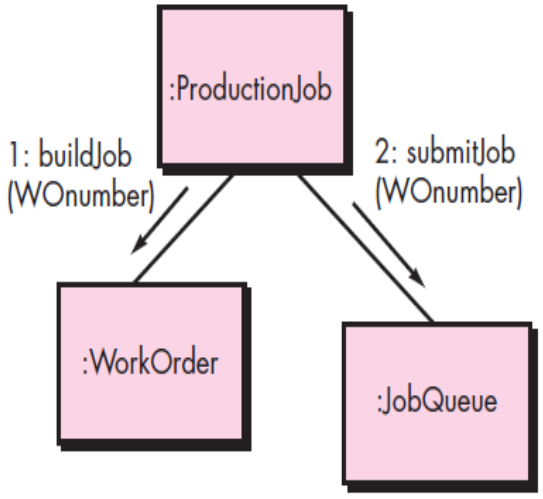
Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate.

The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

FIGURE 10.6

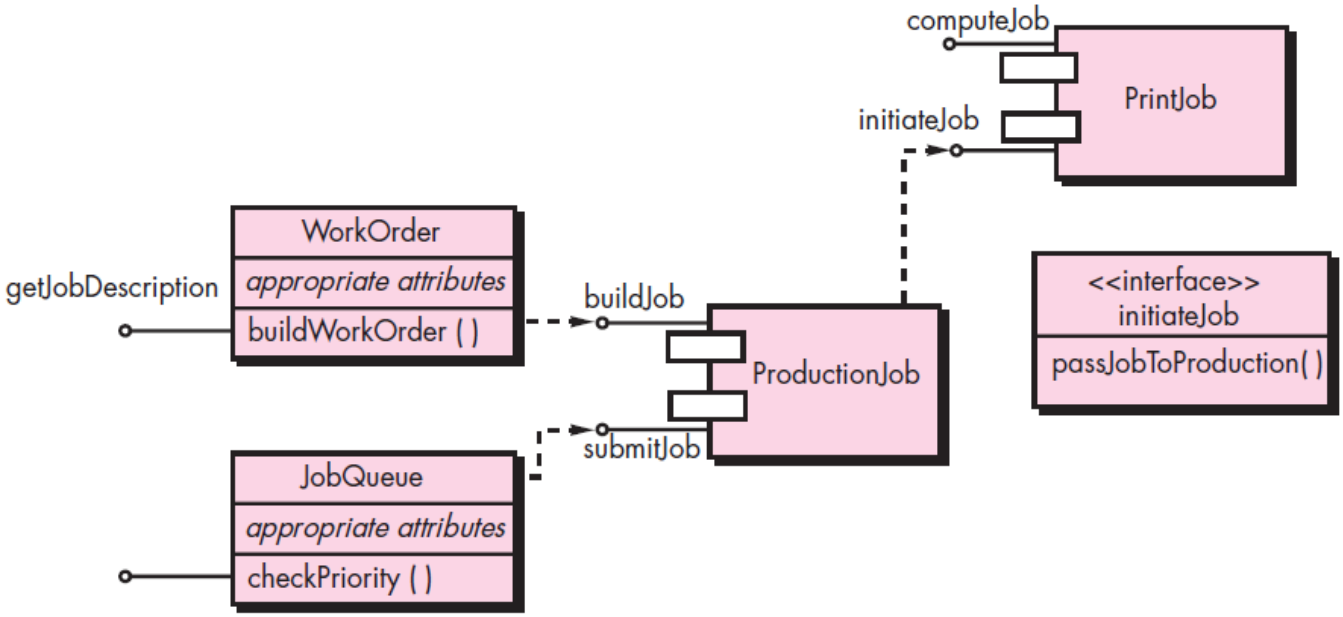
Collaboration diagram with messaging



Step 3b. Identify appropriate interfaces for each component

Within the context of component-level design, a UML interface is "a group of externally visible operations. The interface contains no internal structure, it has no attributes, no associations ." Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. In essence, operations defined for the design class are categorized into one or more abstract classes. Every operation within the abstract class (the interface) should be cohesive; that is,it should exhibit processing that focuses on one limited function or subfunction.

FIGURE 10.7 Refactoring interfaces and class definitions for PrintJob



Step 3c. Elaborate attributes and define data types and data structures required to implement them

In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following.

syntax:

name : type-expression initial-value {property string}

which defines paperType-weight as a string variable initialized to the value A that can take on one of four values from the set {A,B,C, D}.

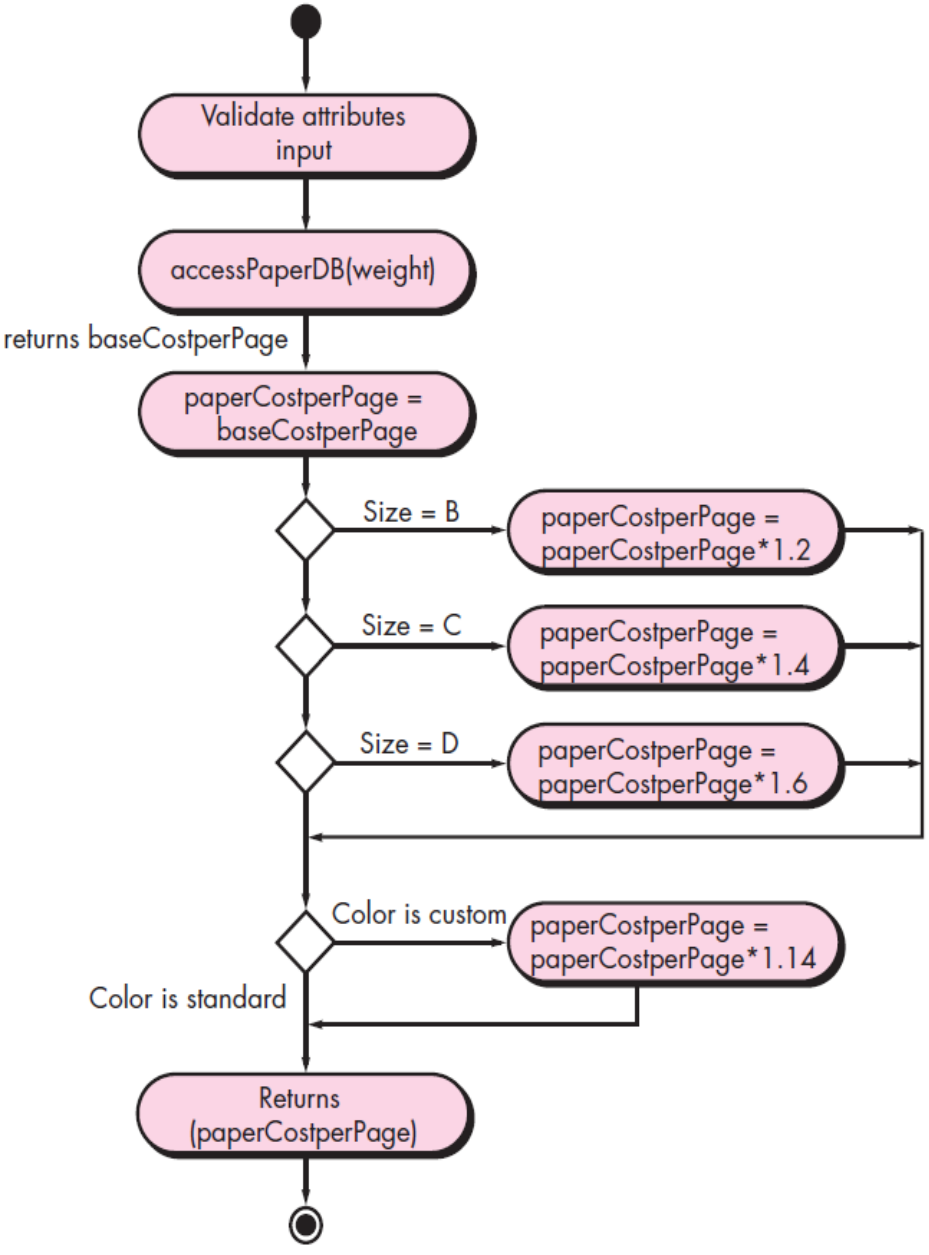
Step 3d. Describe processing flow within each operation in detail.

This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept.

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name.

If the algorithm required to implement computePaperCost() is simple and widely understood, no further design elaboration may be necessary.The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 10.8 depicts a UML activity diagram for computePaperCost(). When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code.

FIGURE 10.8
UML activity diagram for *compute-PaperCost()*



Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data source.

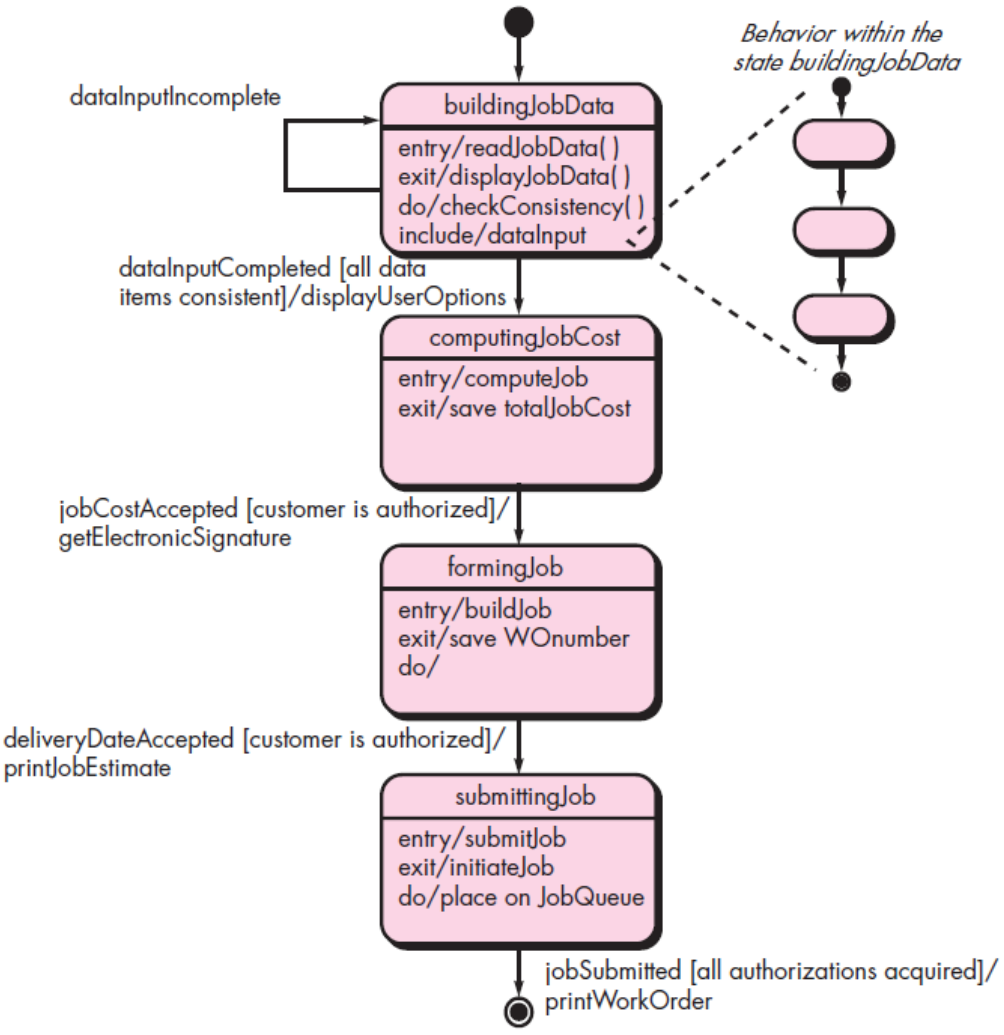
Step 5. Develop and elaborate behavioral representations for a class or component.

UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design.

The dynamic behavior of an object is affected by events that are external to it and the current state of the object. To understand the dynamic behavior of an object, should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states are represented using a UML statechart as illustrated in Figure 10.9.

FIGURE 10.9

Statechart fragment for PrintJob class



Step 6. Elaborate deployment diagrams to provide additional implementation detail

Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions are represented within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. The reason for this is to avoid diagrammatic complexity. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Refactor every component-level design representation and always consider alternatives. Throughout this book, I have emphasized that design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the nth iteration you apply to the model. It is essential to refactor as design work is conducted.