Unit-I

Software definition.

Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.

Software Characteristics.

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 manufacture, 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 (or easily corrected) for software. Both activities are dependent on people, but the relationship between

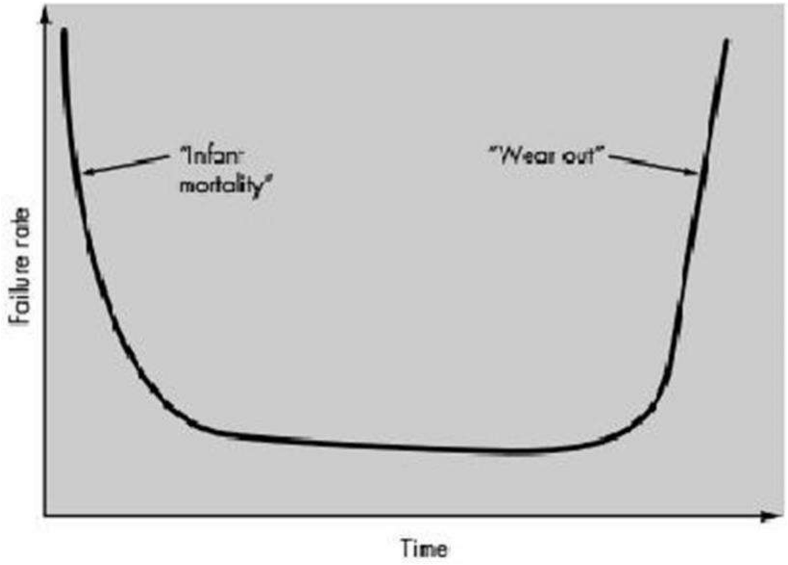


FIGURE 1.1
Failure curve for hardware

people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

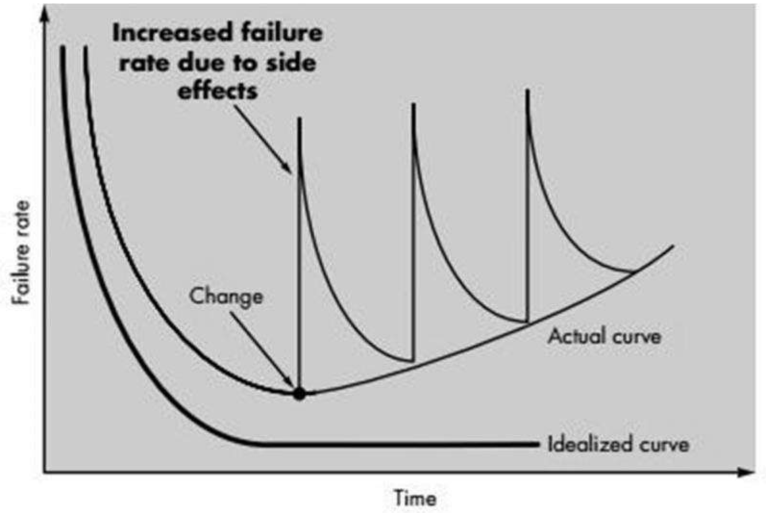
2. **Software doesn't "wear out."**

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 (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steadystate level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out. Software is not susceptible to the environmental maladies that cause hardware to wear out. 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 (ideally, without introducing other errors) and the curve flattens as shown.The idealized curve is a gross oversimplification of actual failure models (see Chapter 8 for more information) for software. However, the implication is clear—software doesn't wear out. But it does deteriorate! This is shown as the “actual curve” in Figure 1.2.

During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.2. Before the curve can return to the original steadystate 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
Idealized and actual failure curves for software



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

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an IC or a chip) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf. As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. 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.

Software myths:

Unlike ancient myths that often provide human lessons well worth heeding, software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious; for instance, they appeared to be reasonable statements of fact (sometimes containing elements of truth), they had an intuitive feel, and they were often promulgated by experienced practitioners who "knew the score."

Three types of myths.

1)**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. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

a) **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 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."

b) **Myth**: My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

c) **Myth**: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality: Software development is not a mechanistic process like manufacturing. 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.

d) **Myth**: If I decide to outsource3 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 projects.

2)**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 (by the customer) and ultimately, dissatisfaction with the developer.

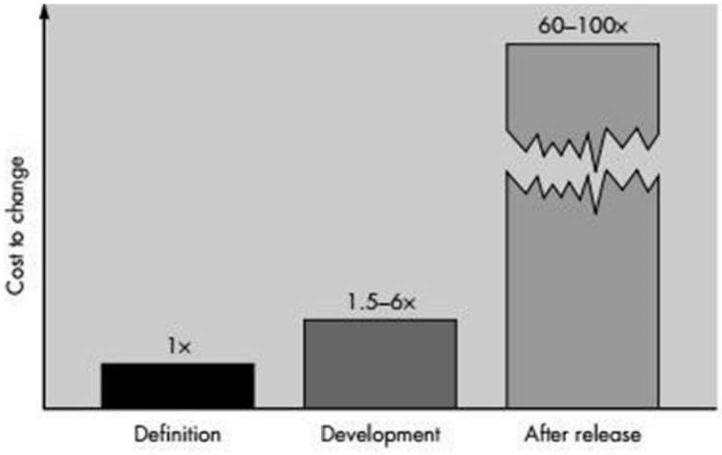
a) **Myth**: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

b) **Myth**: Project 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. Figure 1.3 illustrates the impact of change. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

FIGURE 1.3 The impact of change



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

a) **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 ([LIE80], [JON91], [PUT97]) 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.

b) **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 formal 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.

c) **Myth**: The only deliverable work product for a successful project is the working program.

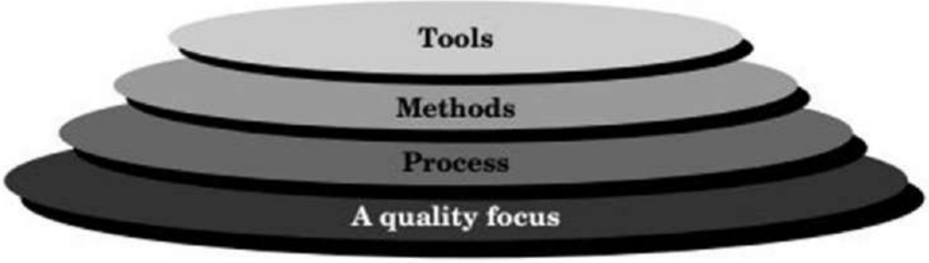
Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

d) **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 quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach.

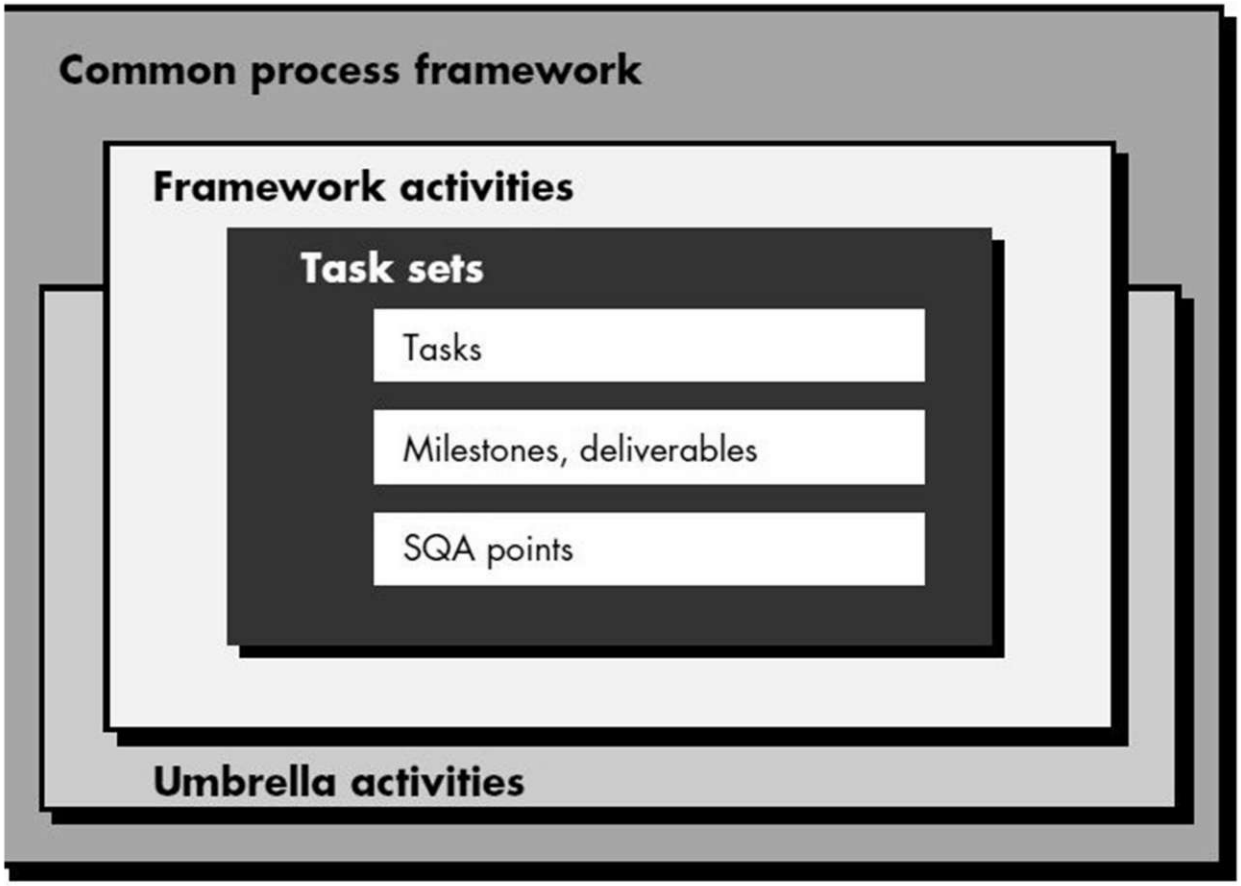
Software Process:

A software process can be characterized as shown in Figure A *common process framework* is established by defining a small number of framework activities that areapplicable to all software projects, regardless of their size or complexity.



A numberof *task sets*—each a collection of software engineering work tasks, project milestones,work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.

Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement2—overlay the process model.Umbrella activities are independent of any one framework activity and occur throughout the process.



To determine an organization’s current state of process maturity, the SEI uses an assessment that results in a five point grading scheme. The grading scheme determines compliance with a *capability maturity model* (CMM) [PAU93] that defines key activities required at different levels of process maturity.

The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1: Initial. The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined. The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

Level 4: Managed. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing. Continuous process improvement is enabled by quantitative Feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

SOFTWARE PROCESS MODELS:

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers described in and the generic phases.

This strategy is often referred to as a *process model* or a *software engineering paradigm.* A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

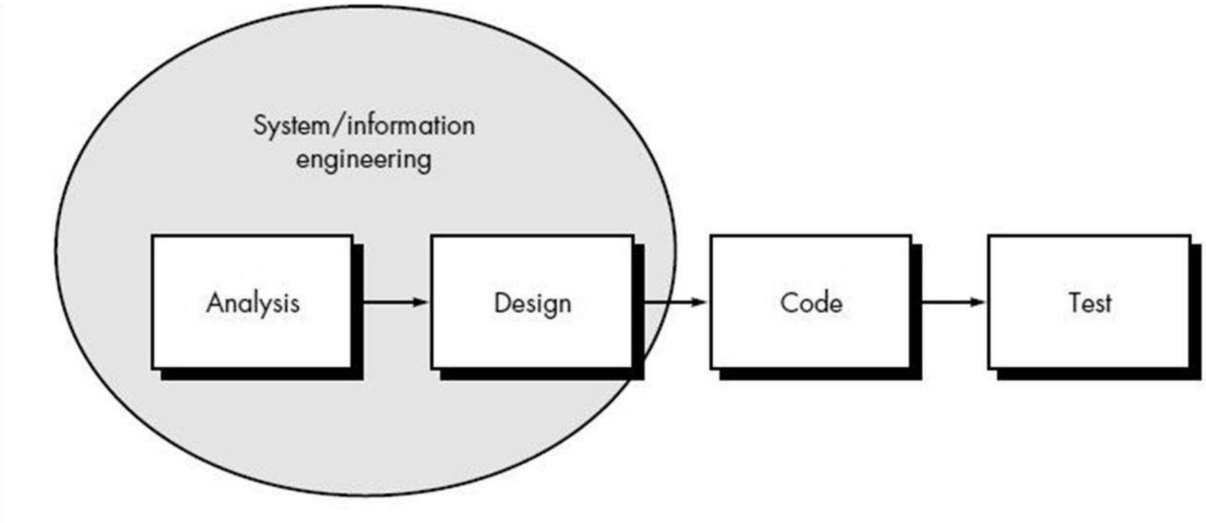
PRESCRIPTIVE AND SPECIALIZED PROCESS MODELS:

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers described in and the generic phases discussed in This strategy is often referred to as a *process model* or a *software engineering paradigm.* A process model for software engineering is chosen based on the
nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

THE LINEAR SEQUENTIAL MODEL:

Sometimes called the *classic life cycle* or the *waterfall model*, the *linear sequential model* suggests a systematic, sequential approach5 to software development that begins at the system level and progresses through analysis, design, coding, testing, and support.
Figure illustrates the linear sequential model for software engineering. Modeled after a conventional engineering cycle, the linear sequential model encompasses the following activities:

System/information engineering and modeling: Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software.This system view is essential when software must interact with other elementssuch as hardware, people, and databases. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level



design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

Software requirements analysis: The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain (described in Chapter 11) for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

Design: Software design is actually a multistep process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.

Code generation: The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing: Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results

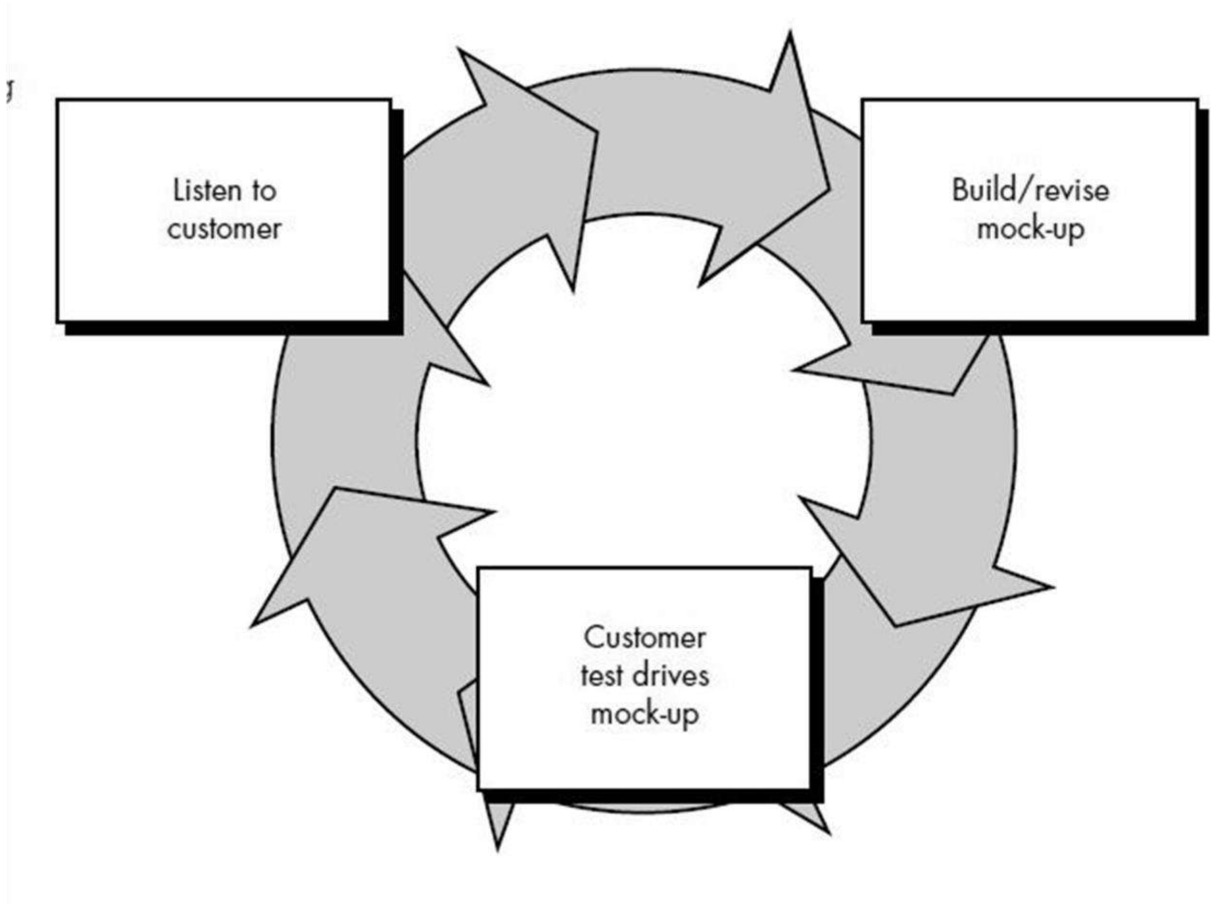
.

Support: Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.

THE PROTOTYPING MODEL:

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. 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 (Figure 2.5) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs.



The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g. Input approaches and output formats). The quick design leads to the construction of prototype.

The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

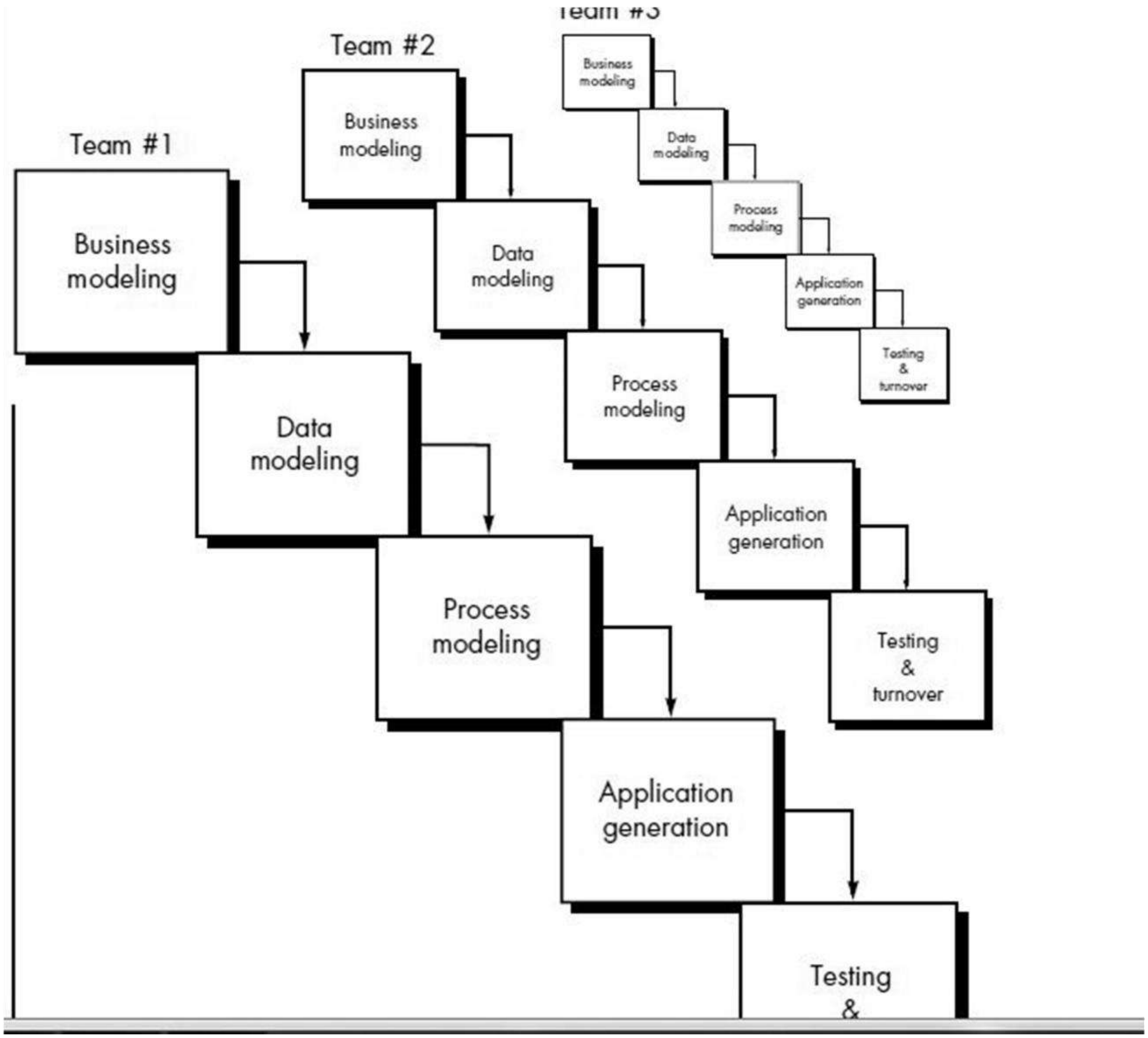
Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly.

THE RAD MODEL:

Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The RAD model is a “high-speed” adaptation of the linear sequential model in which rapid development is achieved by using component-based construction.

Business modeling: The information flow among business functions is modeled in a way that answers the following questions: What information drives the business Process? What information is generated? Who generates it? Where does the informationgo? Who processes it?

Data modeling: The information flow defined as part of the business modeling phaseis refined into a set of data objects that are needed to support the business. The char-acteristics (called *attributes*) of each object are identified and the relationships betweenthese objects defined.



Process modeling: The data objects defined in the data modeling phase are transformedto achieve the information flow necessary to implement a business function.Processing descriptions are created for adding, modifying, deleting, or retrieving adata object.

Application generation: RAD assumes the use of fourth generation techniques Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, Automated tools are used to facilitate construction of the software.

Testing and turnover: Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised

SPECIALI ZED MODELS:

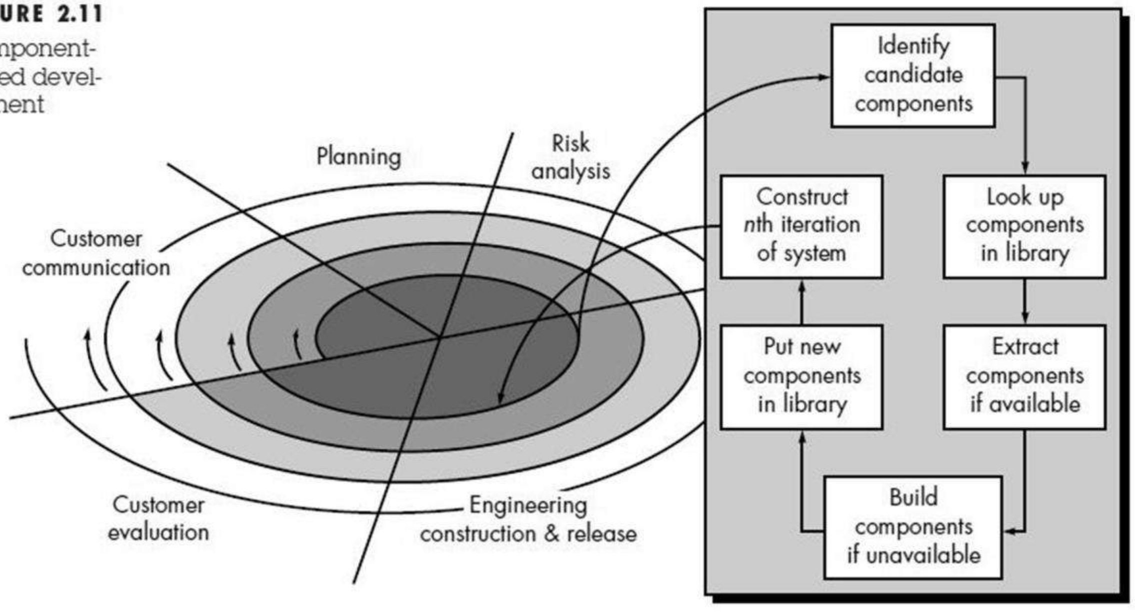
COMPONENT-BASED DEVELOPMENT:

Object-oriented technologies (Part Four of this book) provide the technical framework for a component-based process model for software engineering. The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithms used to manipulate the data. If properly designed and implemented, object-oriented classes are reusable across different applications and computer-based system architectures.

The component-based development (CBD) model (Figure 2.11) incorporates many of the characteristics of the spiral model. It is evolutionary in nature [NIE92], demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components(called *classes*).

The engineering activity begins with the identification of candidate classes. This is accomplished by examining the data to be manipulated by the application and the algorithms that will be applied to accomplish the manipulation. Corresponding data and algorithms are packaged into a class.

FIGURE 2.11 Component-based development


Planning
Risk analysis
Customer communication
Customer evaluation
Engineering construction & release
Identify candidate components
Look up components in library
Extract components if available
Build components if unavailable
Put new components in library
Construct nth iteration of system

Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods (Chapters 21–23). The first iteration of the application to be built is then composed, using classes extracted from the library and any new classes built to meet the unique needs of the application.

Process flow then returns to the spiral and will ultimately re-enter the component assembly iteration during subsequent passes through the engineering activity.

The component-based development model leads to software reuse, and reusability Provides software engineers with a number of measurable benefits. Based on studies of reusability. Although these results are a function of the robustness of the component library, there is little questioning that the component-based development model provides significant advantages for software engineers.

The *unified software development process* [JAC99] is representative of a number of component-based development models that have been proposed in the industry.
Using the *Unified Modeling Language* (UML), the unified process defines the components that will be used to build the system and the interfaces that will connect the components. Using a combination of iterative and incremental development, the unified process defines the function of the system by applying a scenario-based

approach(from the user point of view). It then couples function with an architectural framework that identifies the form the the software will take.

THE FORMAL METHODS MODEL:

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* [MIL87, DYE92], is currently applied by some software development organizations.

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 andtherefore enable the software engineer to discover and correct errors that might go undetected.

Although it is not destined to become a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, the following concerns about its applicability in a business environment have been voiced:

1. The development of formal models is currently quite time consuming and Expensive.
2. Because few software developers have the necessary background to apply formal methods, extensive training is required.
3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers.
These concerns notwithstanding, it is likely that the formal methods approach will gain adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that
would suffer severe economic hardship should software errors occur.

SOFTWARE PROJECT MANAGEMENT:

In the early days of computing, software costs constituted a small percentage of the overall computer-based system cost. An order of magnitude error in estimates of Software cost had relatively little impact. Today, software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer. Software cost and effort estimation will never be an exact science.

Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk.

As an example of LOC and FP problem-based estimation techniques, let us consider software package to be developed for a computer-aided design application for mechanical components. A review of the *System Specification* indicates that the software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high resolution color display and laser printer.

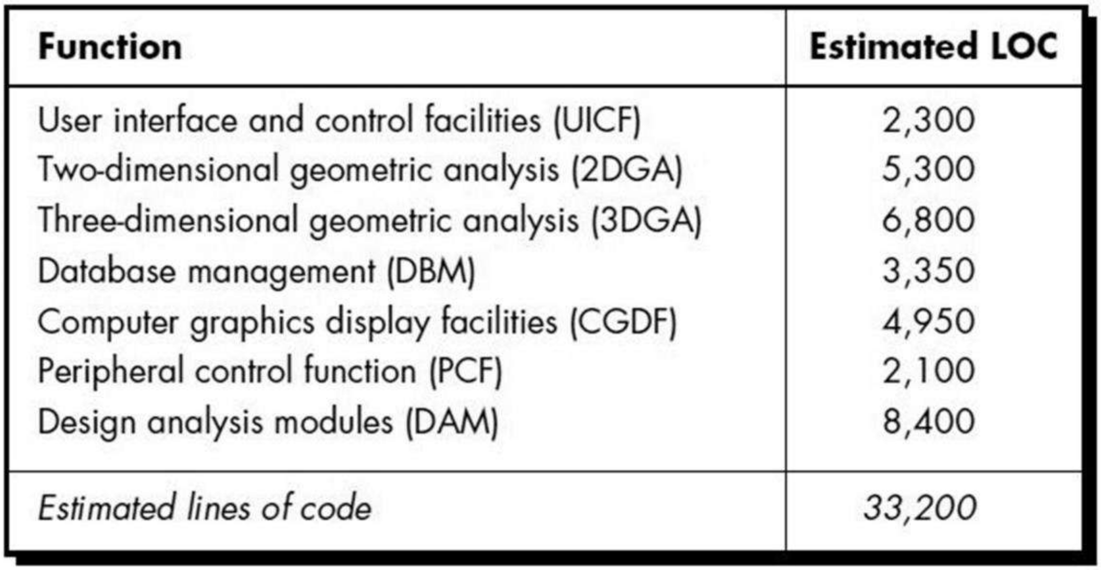
Using the *System Specification* as a guide, a preliminary statement of software scope can be developed:

The CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database.

Design analysis modules will be developed to produce the required output, which will be displayed one variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter. For our purposes, we assume that further refinement has occurred and that the following major software functions are identified:

- User interface and control facilities (UICF)
- Two-dimensional geometric analysis (2DGA)
- Three-dimensional geometric analysis (3DGA)
- Database management (DBM)
- Computer graphics display facilities (CGDF)
- Peripheral control function (PCF)
- Design analysis modules (DAM)

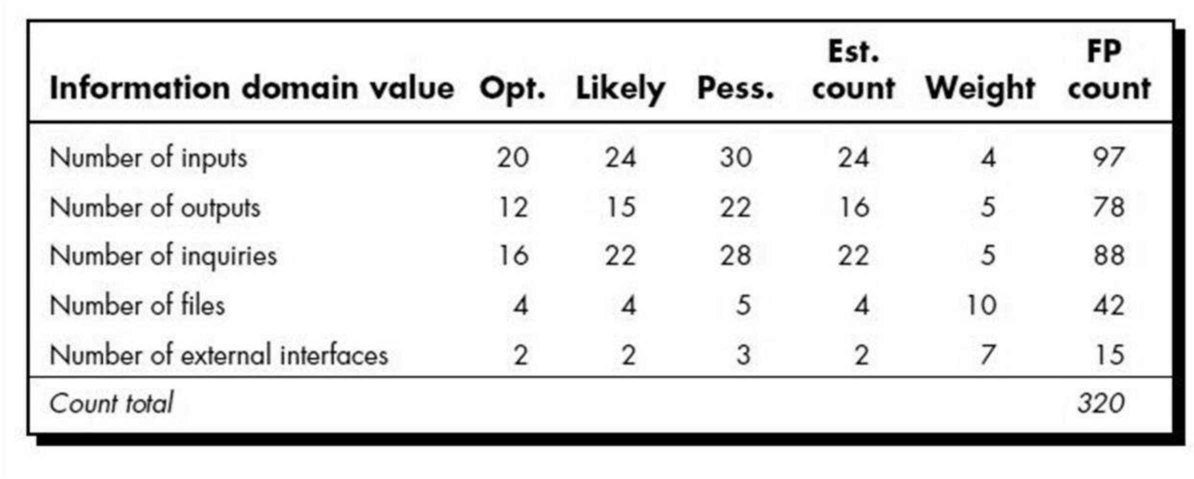
Following the decomposition technique for LOC, an estimation table, shown in Figure is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic—
4600 LOC, most likely—6900 LOC, and pessimistic—8600 LOC.



Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

FP-Based Estimation:

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the function point calculation table presented in Figure the project planner estimates inputs, outputs, inquiries, files, and external interfaces for the CAD software. For the purposes of this estimate, the complexity weighting factor is assumed to be average.



Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

Each of the complexity weighting factors is estimated and the complexity adjustment factor is computed as described:

Factor Value
Backup and recovery 4
Data communications 2
Distributed processing 0
Performance critical 4
Existing operating environment 3
On-line data entry 4
Input transaction over multiple screens 5
Master files updated on-line 3
Information domain values complex 5
Internal processing complex 5
Code designed for reuse 4
Conversion/installation in design 3
Multiple installations 5
Application designed for change 5
Complexity adjustment factor 1.17

Finally, the estimated number of FP is derived:

FP estimated = count-total x [0.65 + 0.01 x _ (*Fi*)]
FP estimated = 375

The organizational average productivity for systems of this type is 6.5 FP/pm. Basedon a burdened labor rate of $8000 per month, the cost per FP is approximately $1230.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is $461,000 and the estimated effort is 58 person-months.

COCOMO MODEL:

In this section on “software engineering economics,” Barry Boehm introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel.* The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called *COCOMO II* [BOE96, BOE00].

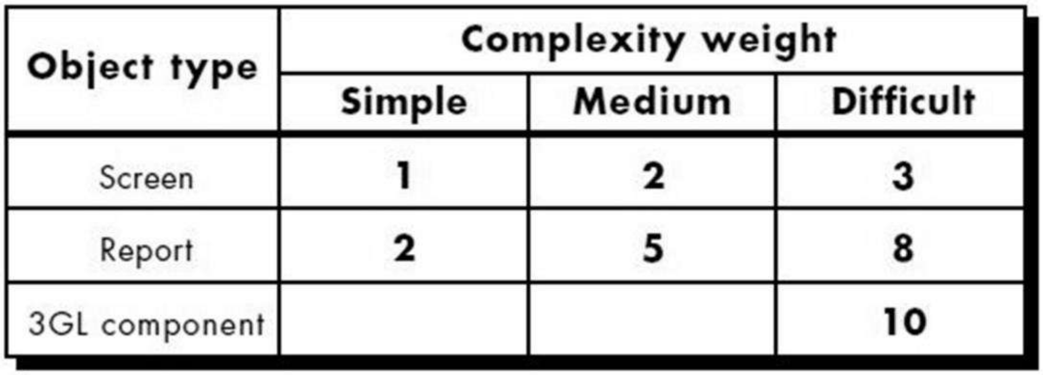
Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

Application composition model: Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

Early design stage model: Used once requirements have been stabilized and basic software architecture has been established.

Post-architecture-stage model: Used during the construction of the software.
Like all estimation models for software, the COCOMO II models require sizing information.

Three different sizing options are available as part of the model hierarchy:
object points function points lines of source code.
The COCOMO II application composition model uses object points and isillustrated in the following paragraphs. It should be noted that other, more



Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Sophisticated estimation models (using FP and KLOC) are also available as part of COCOMO II.

Like function points in the *object point* is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult) using criteria suggested by Boehm [BOE96].

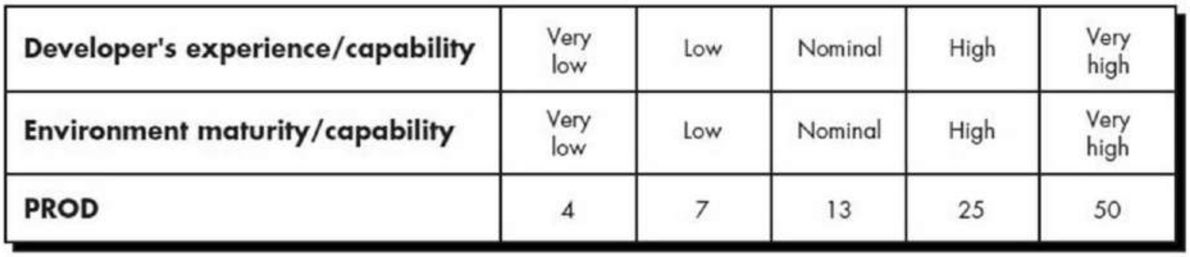
In essence, complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

Once complexity is determined, the number of screens, reports, and components are weighted according to Table 5.1. The object point count is then determined by multiplying the original number of object instances by the weighting factor in Table and summing to obtain a total object point count. When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

NOP = (object points) x [(100 _ %reuse)/100] ,where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value,"productivity rate" must be derived. Table 5.2 presents the productivity rate

PROD = NOP/person-month



Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

The software equation [PUT92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software.

PROJECT SCHEDULING:

You've selected an appropriate process model, you've identified the software Engineering tasks that have to be performed, you estimated the amount of work and the number of people, you know the deadline, you've even considered the risks. Now it's time to connect the dots.

That is, you have to create a network of software engineering tasks that will enable you to get the job done on time. Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality. In a nutshell, that's softwareproject scheduling and tracking.

Basic Principles:

The reality of a technical project (whether it involves building a hydroelectric plant or developing an operating system) is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the "critical" path.4 If these "critical" tasks fall behind schedule, the completiondate of the entire project is put into jeopardy.

Compartmentalization: The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

Interdependency: The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produceds by another is available. Other activities can occur independently.

Time allocation: Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation: Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned staff members

Defined responsibilities: Every task that is scheduled should be assigned to a specific team member.

Defined outcomes: Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a module) or a part of a work product. Work products are often combined in deliverables.

Defined milestones: Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved. Each of these principles is applied as the project schedule evolves.

EARNED VALUE ANALYSIS:

we discussed a number of qualitative approaches to project tracking. Each provides the project manager with an indication of progress, but an assessment of the information provided is somewhat subjective. It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule. In fact, a technique for performing quantitative analysis of progress does exist. It is called *earned value analysis* (EVA).

Humphrey [HUM95] discusses earned value in the following manner:
The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total. Stated even more simply, earned value is a measure of progress.
It enables us to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling. In fact, Fleming and Koppleman [FLE98] argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project. "To determine the earned value, the following steps are performed:Wilkens [WIL99] notes that "the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed andthe latter represents the budget of the activities that actually were completed." Givenvalues for BCWS, BAC, and BCWP, important progress indicators can be computed:
Schedule performance index, SPI = BCWP/BCWS
Schedule variance, SV = BCWP – BCWS
SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule.
SV is simply an absolute indication of variance from the planned schedule. Percent scheduled for completion = BCWS/BAC
provides an indication of the percentage of work that should have been completed by time *t.*
Percent complete = BCWP/BAC
provides a quantitative indication of the percent of completeness of the project at a given point in time, *t.*
It is also possible to compute the *actual cost of work performed,* ACWP. The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute
Cost performance index, CPI = BCWP/ACWP
Cost variance, CV = BCWP – ACWP

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables the software project manager to take corrective action before a project crisis develops.

RISK MANAGEMENT:

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

REACTIVE VS. PROACTIVE RISK STRATEGIES:

Reactive risk strategies have been laughingly called the "Indiana Jones school of risk management" [THO92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, "Don't worry, I'll think of something!" Never worrying about problems until they happened, Indy would reacting some heroic way.

Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks.

Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire fighting mode*. When this fails, "crisis management" [CHA92] takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance.

Then, the software team establishes a plan for managing risk. The primaryobjective is to avoid risk, but because not all risks can be avoided, the team worksto develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, we discuss a proactive strategy for risk management.

SOFTWARE RISKS:

Although there has been considerable debate about the proper definition for softwarerisk, there is general agreement that risk always involves two characteristics.

Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks.1

- *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur.

RISK IDENTIFICATION:

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks,the project manager takes a first step toward avoiding them when possible and controllingthem when necessary. *Product size*—risks associated with the overall size of the software to be builtor modified.

- *Business impact*—risks associated with constraints imposed by management or the marketplace.
- *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.

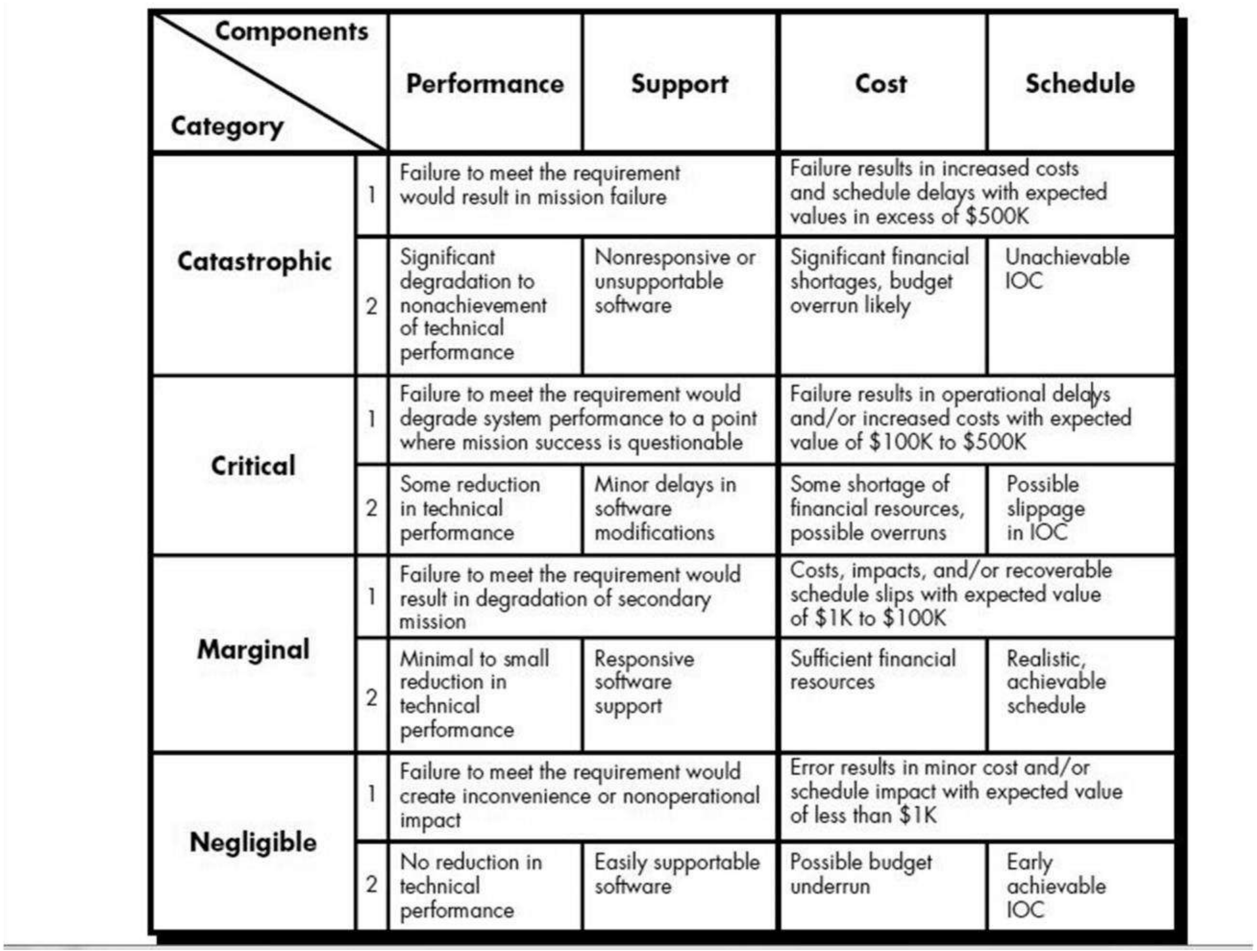
- *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system. • *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

Assessing Overall Project Risk:

The following questions have derived from risk data obtained by surveying experienced software project managers in different part of the world [KEI98]. The questions are ordered by their relative importance to the success of a project.

Risk Components and Drivers he U.S. Air Force [AFC88] has written a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk componentsperformance, cost, support, and schedule. In the context of this discussion, the riskcomponents are defined in the following manner:

- *Performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- *Cost risk*—the degree of uncertainty that the project budget will be maintained.
- *Support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *Schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.



Category \ Components		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

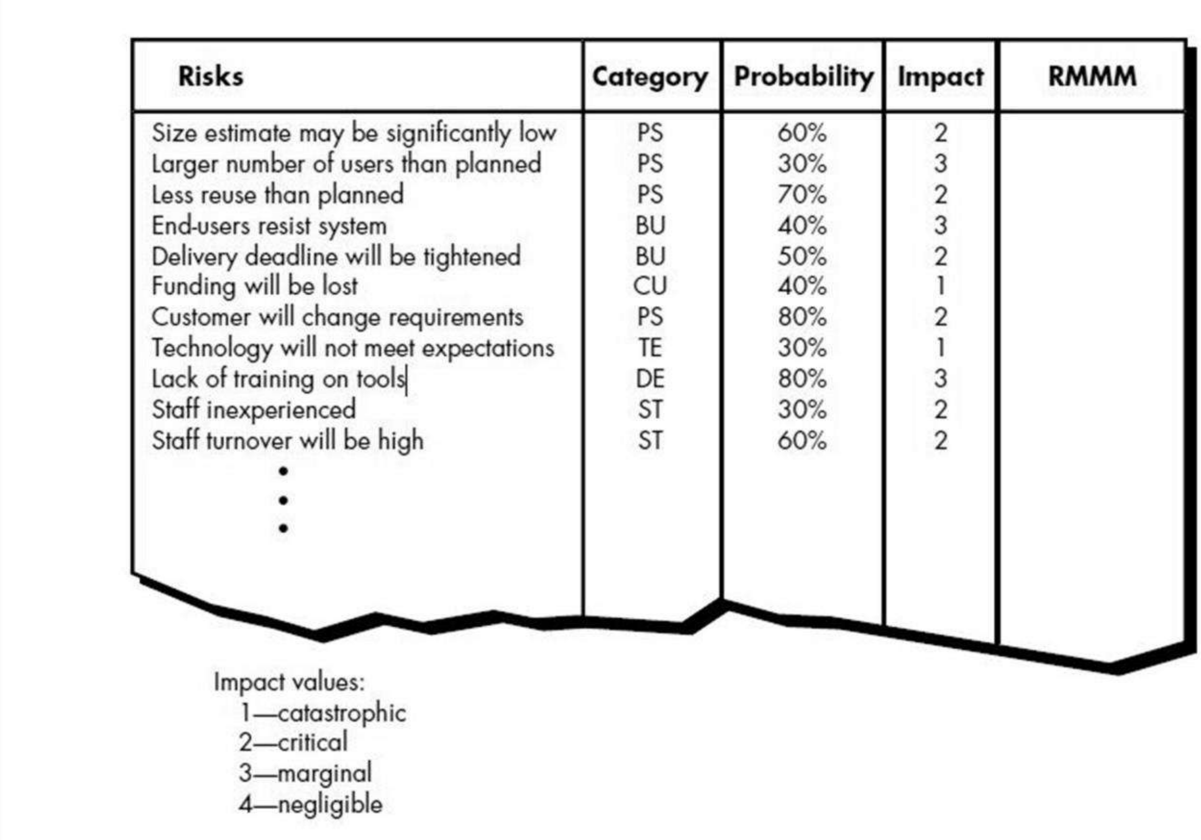
a characterization of the potential consequences of errors (rows labeled 1) or a failureto achieve a desired outcome (rows labeled 2) are described. The impact category ischosen based on the characterization that best fits the description in the table.

RISK PROJECTION:

Risk projection, also called *risk estimation,* attempts to rate each risk in two ways—the
likelihood or probability that the risk is real and the consequences of the problems associatedwith the risk, should it occur.

Developing a Risk Table:

A risk table provides a project manager with a simple technique for risk projection



Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
• • •				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], the following steps are recommended to determine the overall consequences of a risk:

1. Determine the average probability of occurrence value for each risk component.

2. Using Figure 6.1, determine the impact for each component based on the criteria shown.

3. Complete the risk table and analyze the results as described in the preceding sections.

The overall *risk exposure*, RE, is determined using the following relationship [HAL98]:

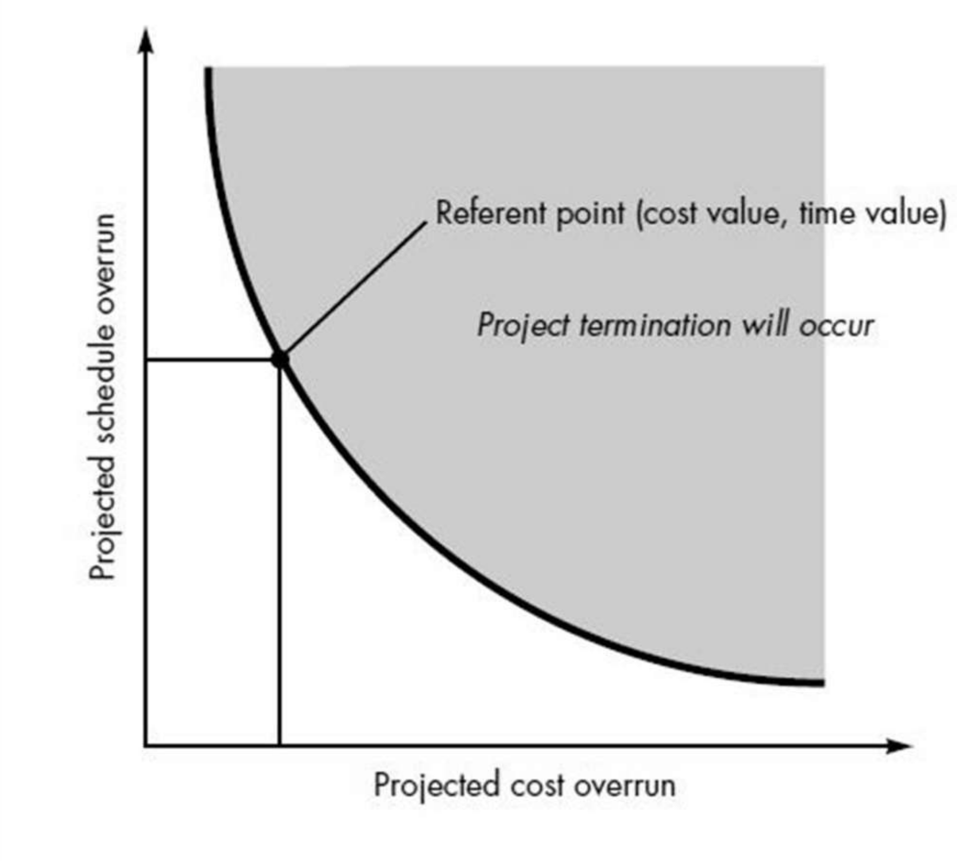
RE = P x C

Risk Assessment:

At this point in the risk management process, we have established a set of triplets of the form [CHA89]:

[*ri*, *li*, *xi*]

Where *ri* is risk, *li* is the likelihood (probability) of the risk, and *xi* is the impact of the risk. During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to rank the risks that have been uncovered, and begin thinking about ways to control and/or avert risks that are likely to occur.



RISK REFINEMENT:

This general condition can be refined in the following manner:
Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards.
Subcondition 2: The design standard for component interfaces has not been solidifiedand may not conform to certain existing reusable components.

Subcondition 3: Certain reusable components have been implemented in a language thatis not supported on the target environment.

RISK MITIGATION, MONITORING, AND MANAGEMENT:

All of the risk analysis activities presented to this point have a single goal—to assistthe project team in developing a strategy for dealing with risk. An effective strategymust consider three issues:

- risk avoidance
- risk monitoring
- risk management and contingency planning

DATA DICTIONARY:

A **data dictionary** is a collection of descriptions of the **data** objects or items in a **data**model for the benefit of programmers and others who need to refer to them. A first step in analyzing a system of objects with which users interact is to identify each object and its relationship to other objects.

When developing programs that use the data model, a data dictionary can be consulted to understand where a data item fits in the structure, what values it may contain, and basically what the data item means in real-world terms. For example, a bank or group of banks could model the data objects involved in consumer banking. They could then provide a data dictionary for a bank's programmers. The data dictionary would describe each of the data items in its data model for consumer banking (for example, "Account holder" and ""Available credit").

Unit –II

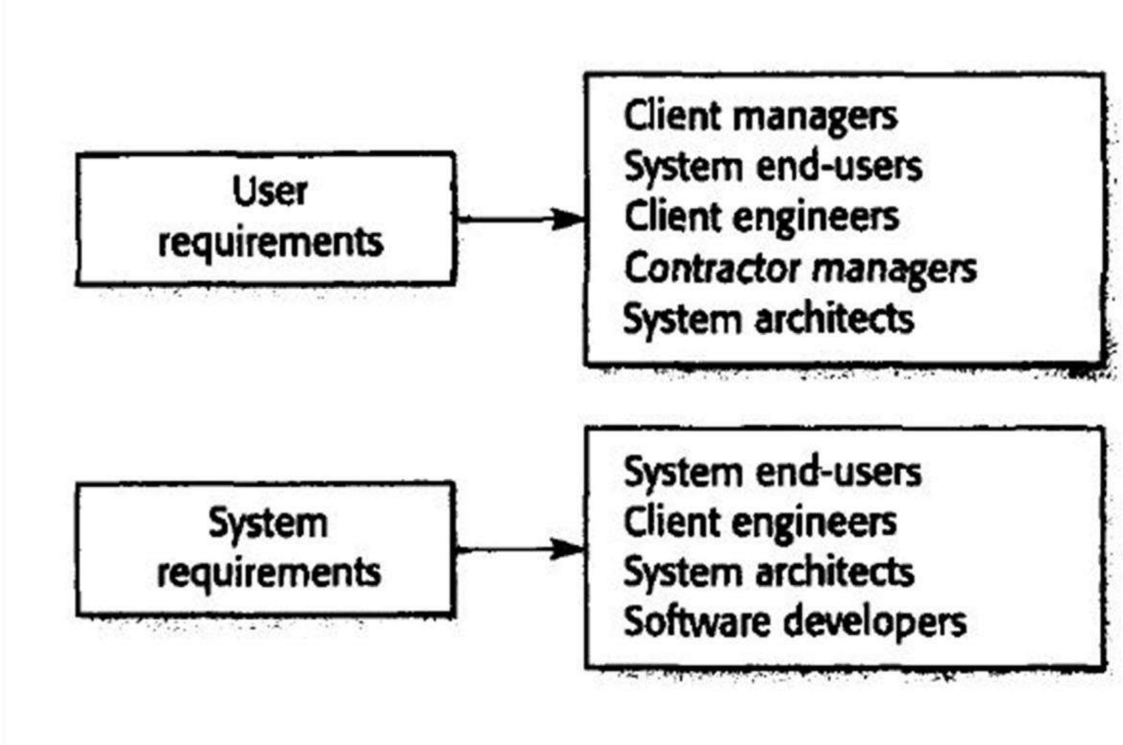
REQUIREMENTS ANALYSIS AND SPECIFICATION

FUMCTIONAL AND NON-FUNCTIONAL REQUIREMENTS:

Software system requirements are often classified as functional requirements, nonfunctional requirements or domain requirements:

I. Functional requirements*:* These are statements of services the system should provide, how the system should! React to particular inputs and how the system should behave in particular simulations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. Non-functional requirements*:* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole.



FUNCTIONAL REQUIREMENTS:

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software and the general approach taken by the organization when writing requirements. When expressed as user requirements, the requirements are usually described in a fairly abstract way. However functional system requirements describe the system function in detail, its inputs and outputs, exceptions, and so on. Functional requirements for a software system may be expressed in a number of ways.

For example, here are examples of functional requirements for a university library system called L:"BSYS, used by students and faculty to order books and documents from other libraries.

I. The user shall be able to search either all of the initial set of databases or select a subset from it.
2. The system shall provide appropriate viewers for the user to read documents in the document store.
3. Every order shall be allocated a unique identifier (ORDER_ill), which the user shall be able to copy to the account's permanent storage area.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document, and they illustrate that functional requirements may be written at different levels of detail (contrast requirements I and 3).

The UBSYS system is a single interface to a range of article databases. It allows users to download copies of published articles in magazines, newspapers and scientific journals.

Imprecision in the requirements specification is the cause of many software engineering problems" It is natural for a system developer to interpret an ambiguous requirement to simplify its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system
Of course, this delays system delivery and increases costs. Consider the second example requirement for the library system that refers to apocopate viewers provided by the system. The library system can deliver documents in a range of formats; the intention of this requirement is that viewers for all of these formats should be available.

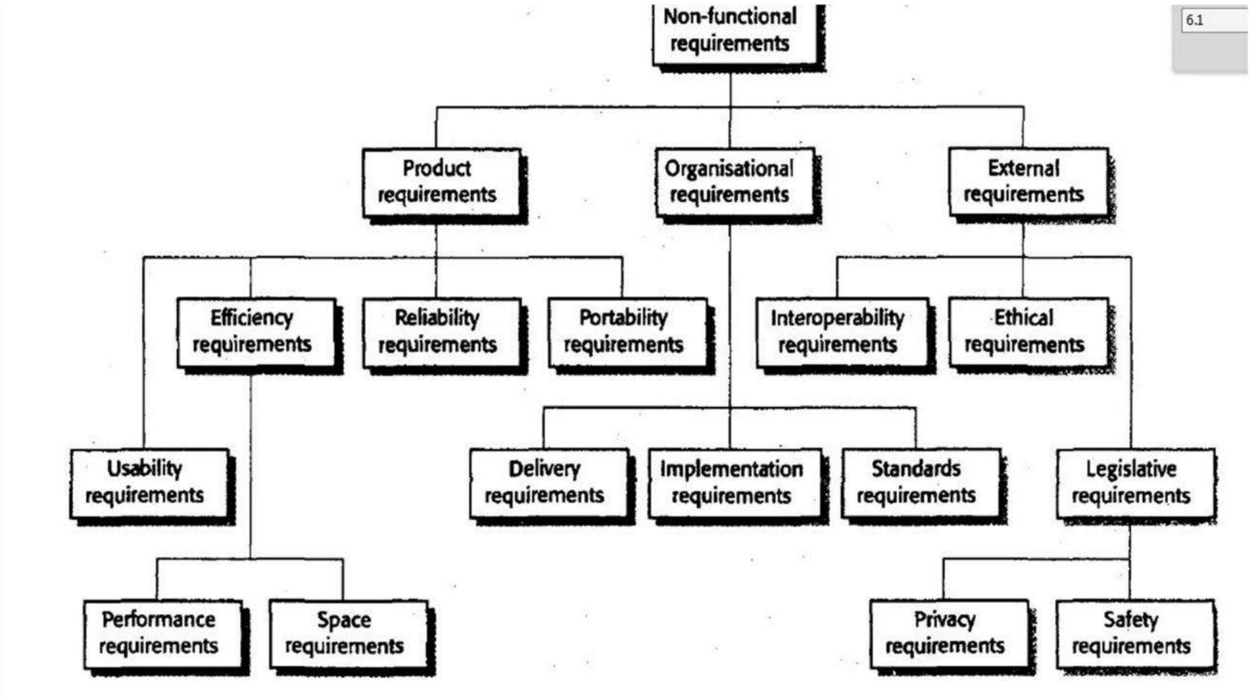
However, the requirement is worded ambiguously; it does not make clear that viewers for each document format should be provided. A developer under schedule pressure simply provide a text viewer and claim that requirement had been met. In principle, the functional requirements specification of a system should be both complete and consistent. *Completeness* means that all services required by the user should be defined. *Consistency* means that requirements should not have contradictory definitions. In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness.

One reason for this is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that different system stakeholders have different-and often inconsistent-needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification.

.

NON FUNCTIONAL REQUIREMENTS:

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may defll1e constraints on the system such as the capabilities of *VO* devices and the data representations used in system interfaces.



The types of non-functional requirements are:

1. **Product requirements:** These requirements specify product behavior. Examples include performance requirements on how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; portability requirements; and usability requirements.
2. **Organizational requirements:** These requirements are derived from policies and procedures in the customer s and developer s organization. Examples include process standards that must be used; implementation requirements such as the programming language or design method used; and delivery requirements that speedy when the product and :Its documentation are to be delivered.
3. **External requirements:** This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include interoperability requirements that define how the system interacts withsystems in other organizations: legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements. Ethical requirements are requirements placed on a system.

USER REQUIREMENTS:

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users without detailed technical knowledge. They should only specify the external behavior of the system and should avoid, as far as possible, system design characteristics. Consequently, if you are writing user requirements, you should not use software jargon, structured notations or formal notations, or describe the requirement by describing thesystem implementation. You should write user requirements in simple language, with simple tables and forms and intuitive diagrams.

1. Lack of clarity: It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.

2. Requirements confusion: Functional requirements, non-functional requirements, system goals and design intonation may not be clearly distinguished.

3. Requirements amalgamation: Several different requirements may be expressed together as a single requirement.

SYSTEM REQUIREMENTS:

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system The first sentence mixes up three kinds of requirements.

1. A conceptual, functional requirement states that the editing system should provide a grid. It presents a rationale for this.
2. A non-functional requirement giving detailed information about the grid units (centimetres or inches).
3. A non-functional user interface requirement that defines how the grid is switched on and off by the user.

The requirement in Figure 6.9 also gives some but not all initialization information. It defines that the grid is initially off. However, it does not define its units when turned on. It provides some detailed information-namely, that the user may toggle between units-but not the spacing between grid lines.

User requirements that include too much information constrain the freedom of the system developer to provide innovative solutions to user problems and are difficult to understand. The user requirement should simply focus on the key facilities to be provided. I have rewritten the editor grid requirement to focus only on the essential system features.

Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and is particularly useful when requirements are changed.

For example, the rationale in Figure recognizes that an active grid where positioned objects automatically 'snap' to a grid line can be useful. However, this has been deliberately rejected in favor of manual positioning. If a change to this is proposed at some later stage, it will be clear that the use of a passive grid was deliberate rather than an implementation decision.

To minimize misunderstandings when writing user requirements, I recommend that you follow some simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. The format I use shows the initial requirement in boldface, including a statement of rationale with each user requirement and reference to the more detailed system requirement specification.

2. Use language consistently. You should always distinguish between mandatory and desirable requirements. *Mandatory* requirements are requirements that the system must support and are usually written using 'shall'. *Desirable requirements* are not essential and are written using 'should'.
3. Use text highlighting (bold, italic or color) to pick out key parts of the requirement.
4. Avoid, as far as possible, the use of computer jargon. Inevitably, however, detailed technical tens will creep into the user requirements.

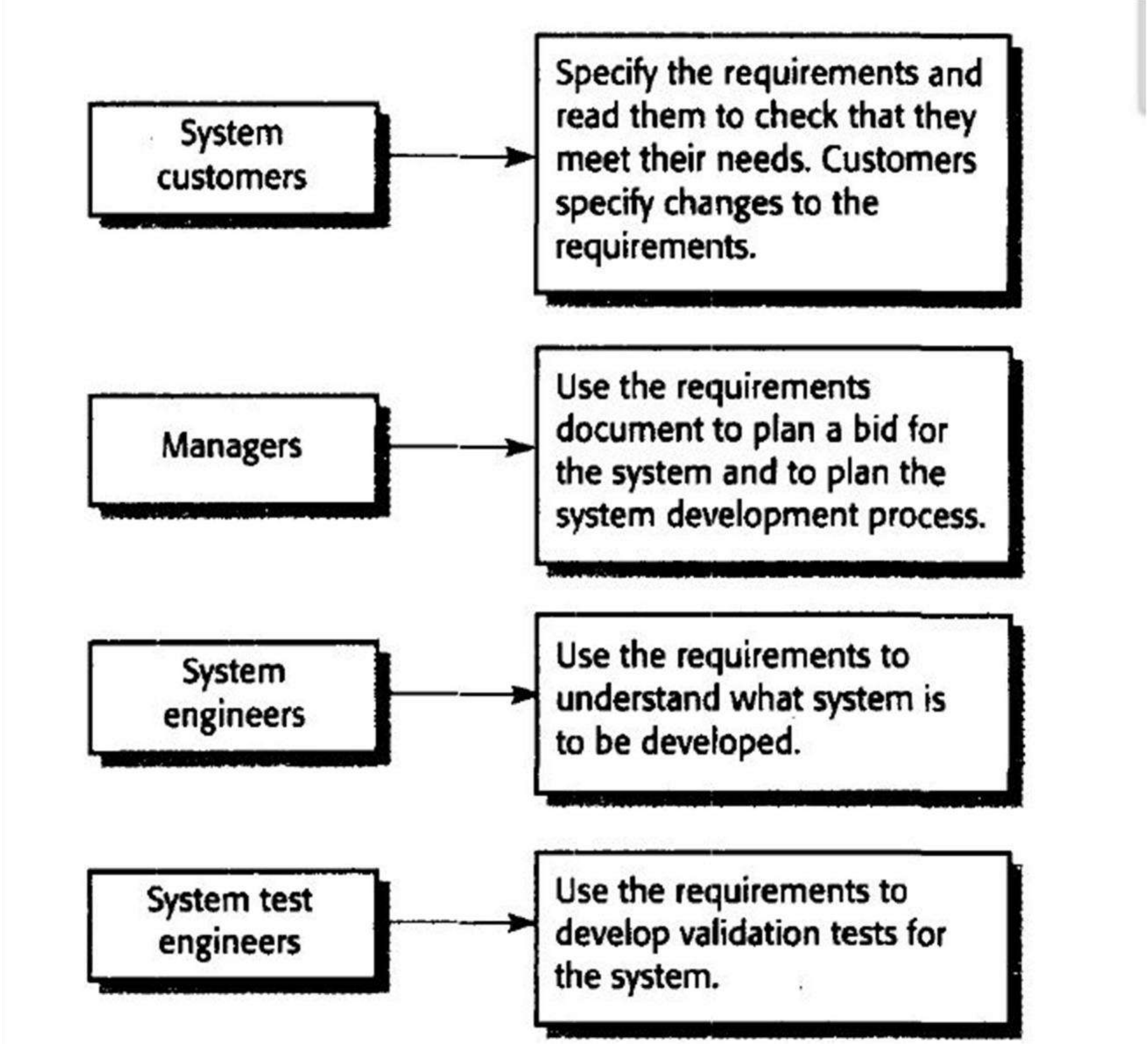
THE SOFTWARE REQUIREMENTS DOCUMENT:

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. In some cases, the user and system requirements may be integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification.

If there are a large number of requirements, the detailed system requirements may be presented in a separate document. The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software.

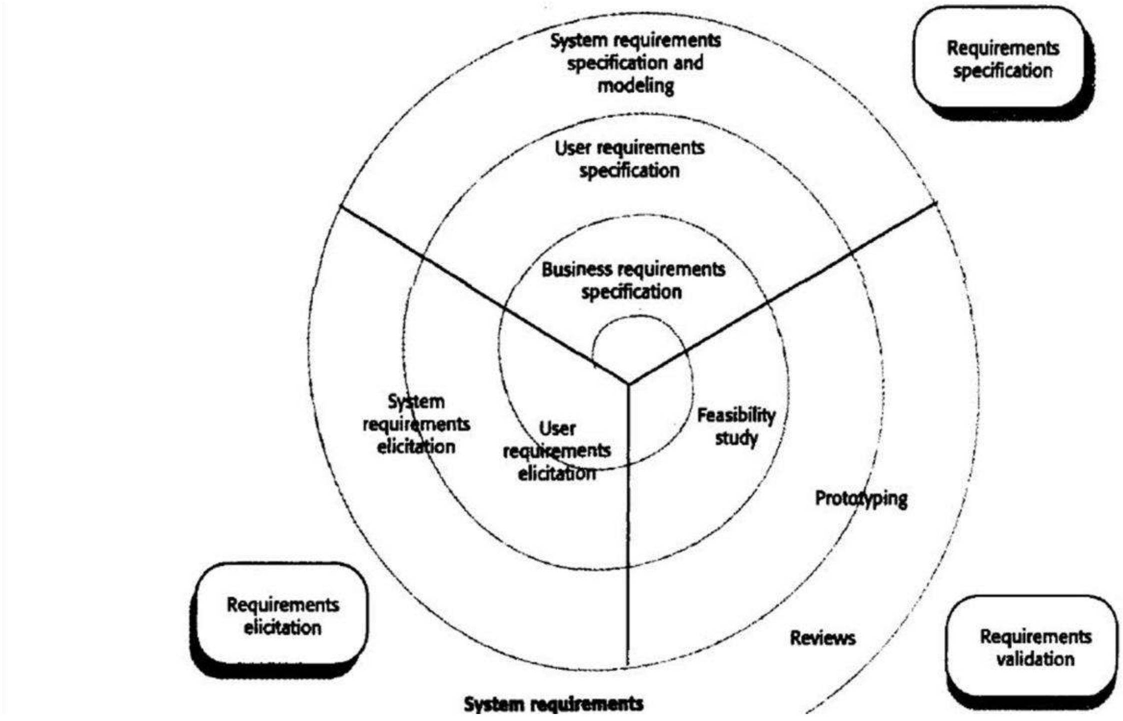
The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements.

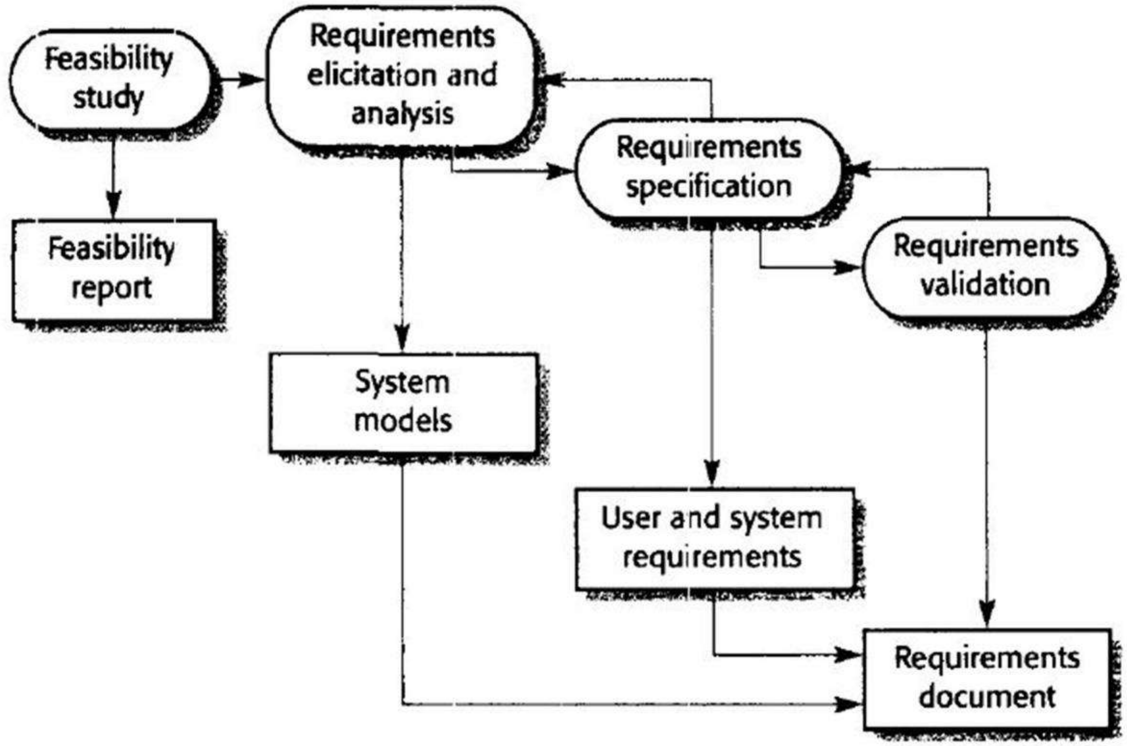
The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. When the system will be developed by an external contractor, critical system specifications need to be precise and very detailed. When there is more flexibility in the requirements and where an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities resolved during development of the system.


System customers
Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers
Use the requirements document to plan a bid for the system and to plan the system development process.
System engineers
Use the requirements to understand what system is to be developed.
System test engineers
Use the requirements to develop validation tests for the system.

REQUIREENTS ENGINEERING PROCESS:

The goal of the requirements engineering process is to create and maintain a system requirements document. The overall process includes four high-level requirements engineering sub-processes. These are concerned with assessing whether the system is useful to the business (feasibility study); discovering requirements (elicitation and analysis); converting these requirements into some standard form (specification);and checking that the requirements actually define the system that the customer wants (validation)


System requirements specification and modeling
User requirements specification
Business requirements specification
Requirements specification
System requirements elicitation
User requirements elicitation
Feasibility study
Prototyping
Reviews
Requirements elicitation
Requirements validation
System requirements



FEASIBILITY STUDY:

For all new systems, the requirements engineering process should start with a feasibility study. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support. business processes. The results of the feasibility study should be a report that recommends whether or not it is worth carrying on with the requirements engineering and system development process.

A feasibility study is a short, focused study that aims to answer a number of question:

I. Does the system contribute to the overall objectives of the organisation?

2. Can the system be implemented using (current technology and within given cost and schedule constraints?

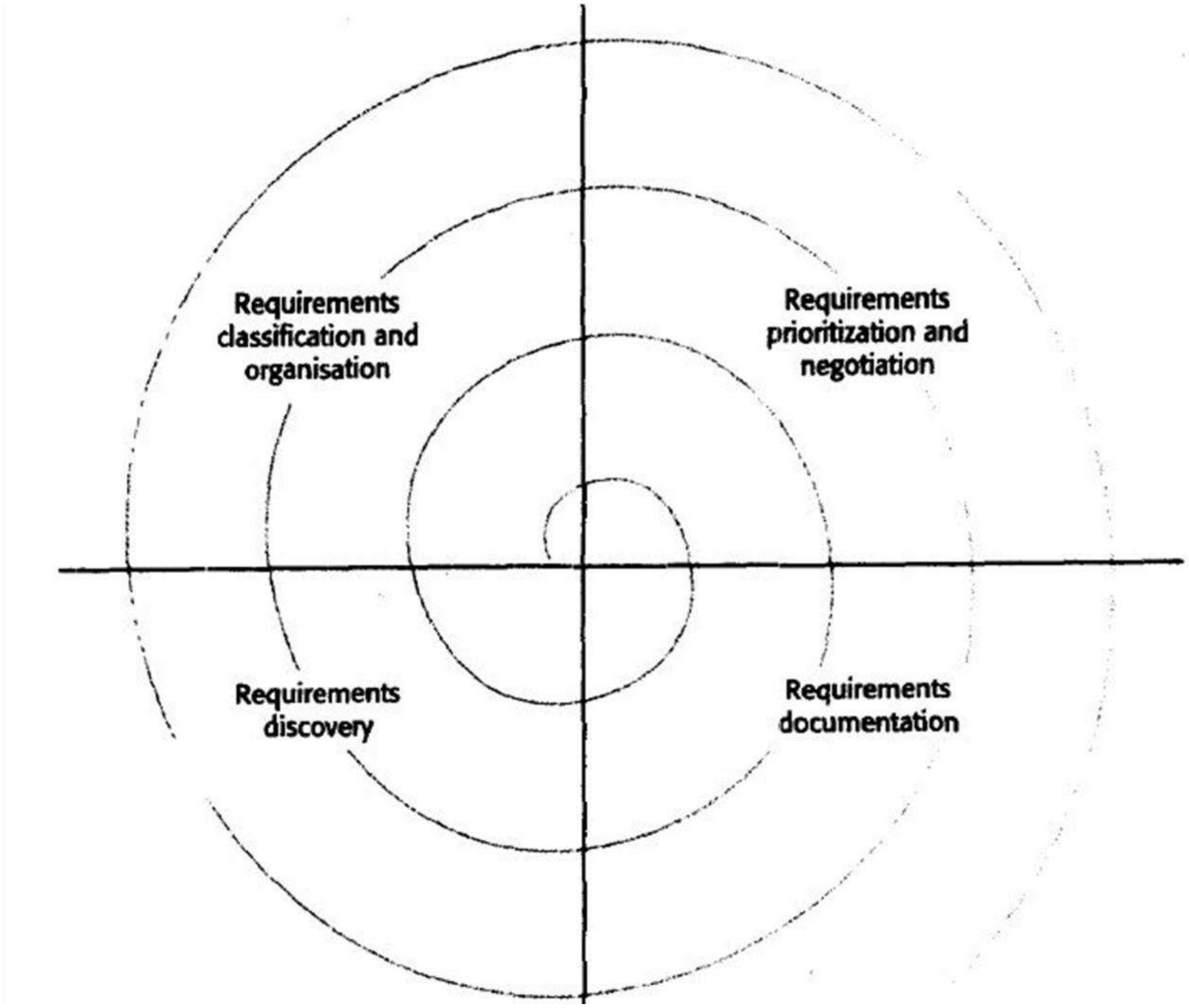
1. Can the system be integrated with other systems which are already in place?

REQUIREMENTS ELICITATION AND ANALYSIS:

Requirements elicitation and analysis may involve a variety of people in an organization.

The term *stakeholder* is used to refer to any person or group who will be affected by the system, directly or indirectly. Stakeholders include end-users who interact withthe system and everyone else in an organization that may be affected by its installation.

Other system stakeholders may be engineers who are developing or maintaining related systems, business managers, domain experts and trade union representatives.


Requirements classification and organisation
Requirements prioritization and negotiation
Requirements discovery
Requirements documentation

1. **Requirements discovery** *:*This is the process of interacting with stakeholders in the system to collect their requirements. Domain requin:ments from stakeholders and documentation are also discovered during this activity.

2. **Requirements classijication and organisation:** This activity takes the unstructured collection of requirements, groups related requirements and organises them into coherent clusters.

3. **Requirements prioritisation and negotiation:** Inevitably, where multiple stakeholders

are involved, requirements will conflict. This a,ctivity is concerned with pnonusmg requirements, and finding and resolving requirements conflicts through negotiation.

4. **Requirements documentation** *:*The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

Requirements discovery:

Requirements discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.Sources of information during the requirements discovery phase include documentation,system stakeholders and specifications of similar systems. You interactwith stakeholders through interviews and observation, and may use scenarios andprototypes to help with the requirements discovery.

VIEW POINTS:

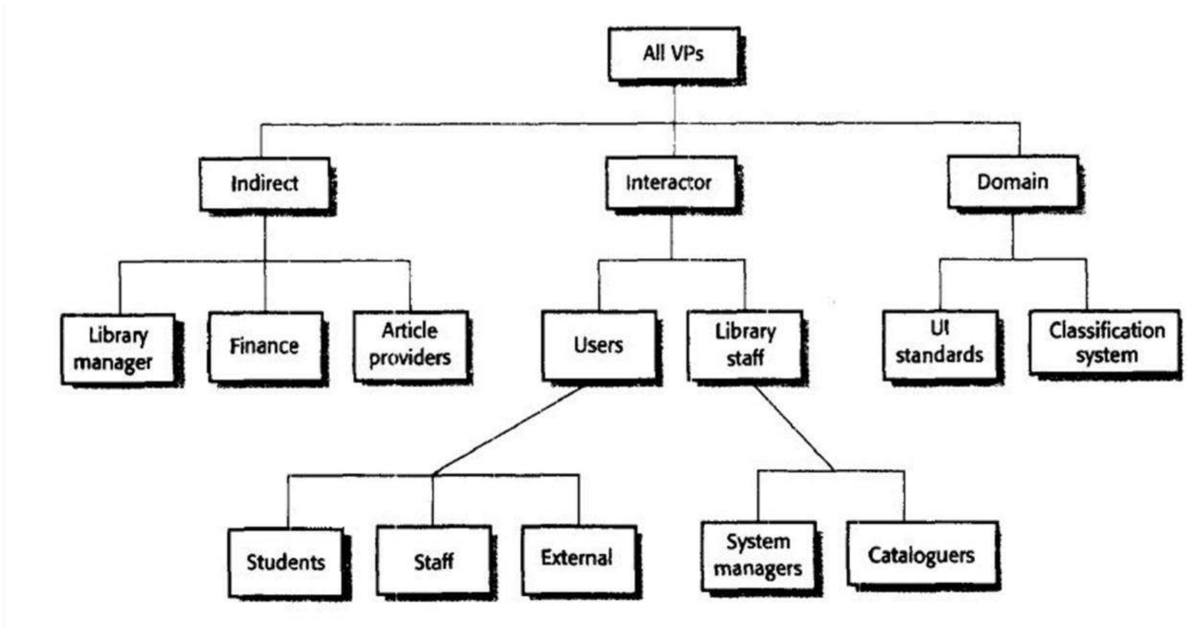
Akey strength of viewpoint-oriented analysis is that it recognises multiple perspectives and provides a framework for discovering confliicts in the requirements proposed by different stakeholders.

1. ***Interactor viewpoints :*** represent people or other systems that interact directly with the system. In the bank ATM system, examples of interactor viewpoints are the bank's customers and the bank's account database.

2. ***Indirect viewpoints:*** represent stakeholders who do not use the system themselves but who influence the requirements in some way. In the bank ATM system, examples of indirect viewpoints are the management of the bank and the bank security staff.

3. ***Domain viewpoints:*** represent domain characteristics and constraints that influence

the system requirements. In the bank ATM system, an example of a domain viewpoint would be the standards that have been developed for interbank communications.



INTERVIEWING:

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they use and the system to be develped.

Requirements are derived from the answers to these questions. Interviews may be of two types:

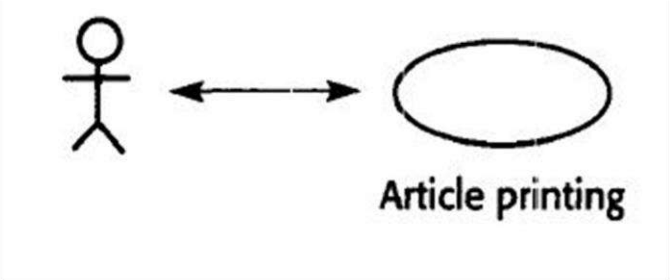
1. Closed interviews where the stakeholder answers a predefined set of questions. 2. Open interviews where there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

SCENARIOS:

Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions. Each scenario covers one or more possible interactions. Several fonns of scenarios have been developed,each of which provides different types of information at different levels of detail about the system.

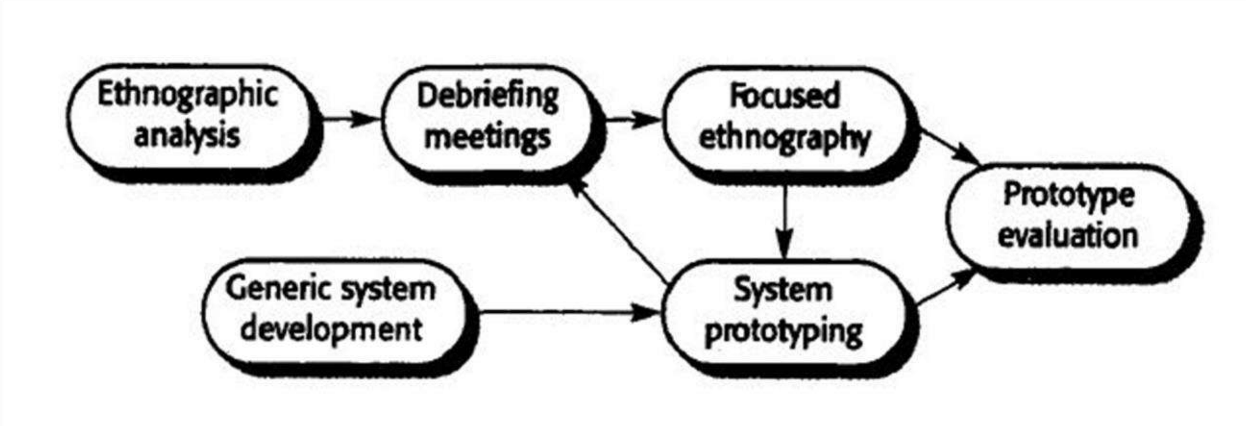
USECASES:

Use-cases are a scenario-based technique for requirements elicitation which were first introduced in the Objector method (Jacobsen, et al., 1993). They have now become a fundamental feature of the UML notation for describing object-oriented system models. In their simplest form, a use-case identifies the type of interaction and the actors involved.



ETHNOGRAPHY:

Ethnography is an observational technique that can be used to understand social and organizational requirements. An analyst immerse s him or herself in the working environment where the system will be used. He: or she observes the day - to-day work and notes made of the actual tasks in which participan ts are involved. The value of ethnography is that it helps analysts discover im plicit system requirements that reflect the actual rather than the fin al processes ill which people are involved.



REQUIREMENTS VALIDATION:

Requirements validation is concerned with showing that the requirements actually define the system that the customer wants. Requirements validation overlaps analysis in that it is concerned with finding problems with the requirements. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when they are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed and then the system must be tested again.

1. *Validity checks:* A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have: diverse stakeholders with distinct needs, and any set of requirements is inevitably a compromise across the stakeholder community.
2. *Consistency checks:* Requirements in the document should not conflict. That is, there should be no contradictory constraints or descriptions of the same system function.
3. *Completeness checks:* The requirements document should include requirements, which define all functions, and constraints intended by the system user.
4. *Realism checks* Using knowledge: of existing technology, the requirements shouldbe checked to ensure that they could actually be implemented. These checks should also take account of the budget and schedule for the system development' *5. Verifiability :*To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

REQUIREMENTS MANAGEMENT:

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems
Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing. These requirements must then evolve to reflect this changed problem view.

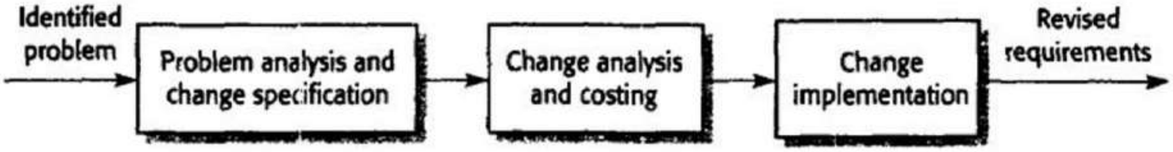
ENDURING AND VOLATILE REQUIREMENTS:

Requirements evolution during the RE process and after a system has gone into services inevitable. Developing software requirements focuses attention on software capabilities, business objectives and other business systems. As the requirements definition is developed, you normally develop a better understanding of users needs.

I. Enduring requirements: These are relatively stable requirements that derive from the core activity of the organization and which relate directly to the domain of the system.
2. Volatile requirements: These are requirements that are likely to change during the system development process or after the system has been become operational. Planning is an essential first stage in the requirements management process.

Requirements management is very expensive. For each project, the planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

1. *Requirements identification:* Each requirement must be uniquely identified so that it can be cross-referenced by other requirements and so that it may be used in traceability assessments.
2. A *change management process:* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies:* These policies define the relationships between requirements, and between the requirements and the system design that should be recorded and how these records should be maintained.
4. *CASE tool support :*Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.


Identified problem
Problem analysis and change specification
Change analysis and costing
Change implementation
Revised requirements

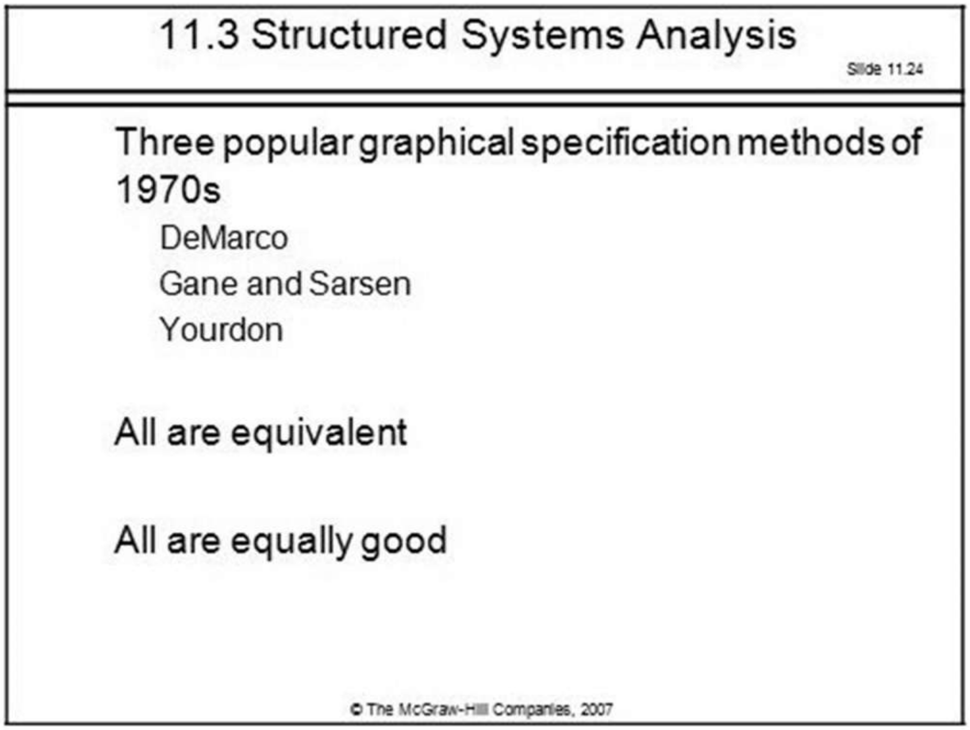
REQUIREMENTS CHANGE MANAGEMENT:

Requirements change management should be applied to all proposed changes to the requirements. The advantage of using a formal process for change management is that all change proposals are treated consistently and that changes to the requirements document are made in a controlled way. There are three principal stages to a change management process:

1. Problem analysis and change specification: The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. The results of the analysis are fed back to the change requestor, and sometimes a more specific requirements change proposal is then made.
*2. Change analysis and costing :*The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. *3. Change implementation :*The requirements document and, where necessary, the system design and implementation are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization.

CLASSICAL ANALYSIS:

STRUCTURED SYSTEM ANALYSIS:



11.3 Structured Systems Analysis

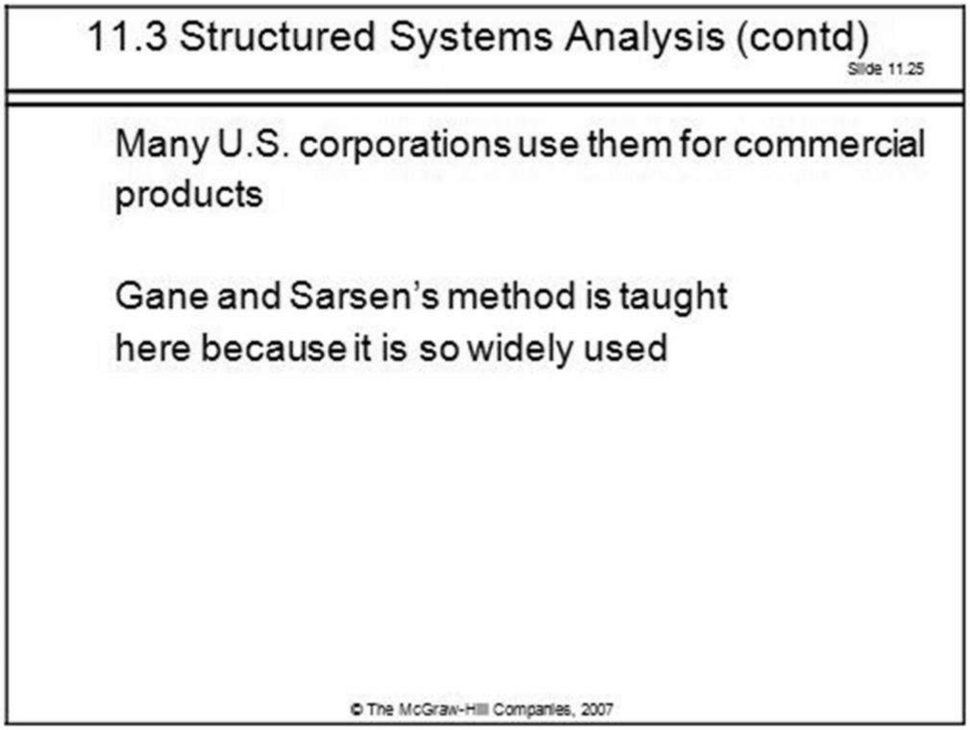
Slide 11.24

Three popular graphical specification methods of 1970s

- DeMarco
- Gane and Sarsen
- Yourdon

All are equivalent

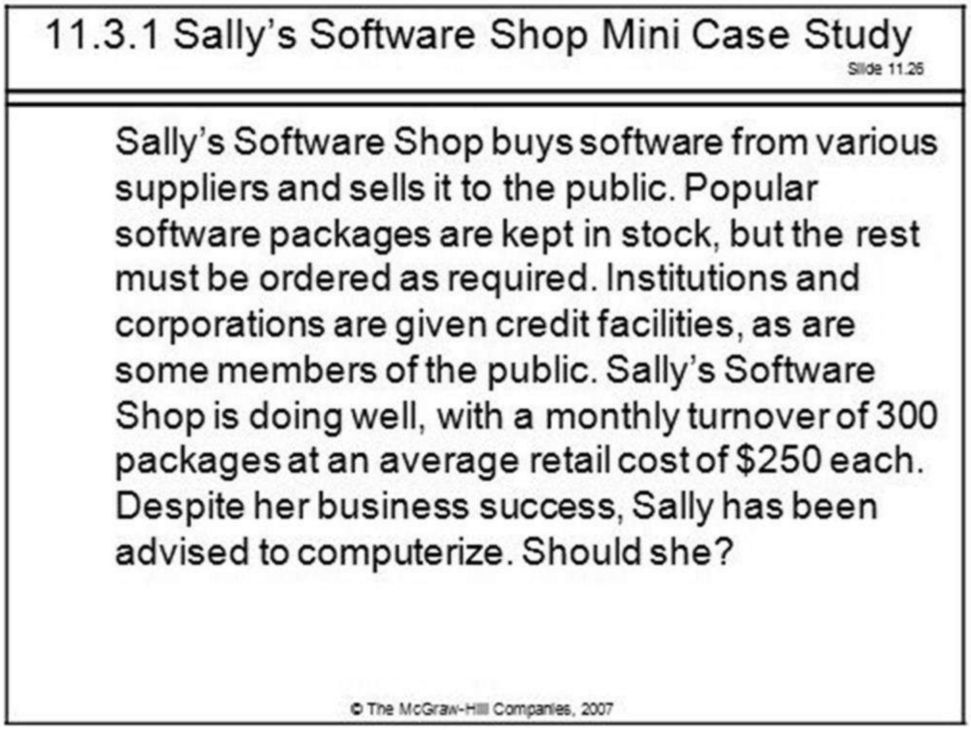
All are equally good



11.3 Structured Systems Analysis (contd)

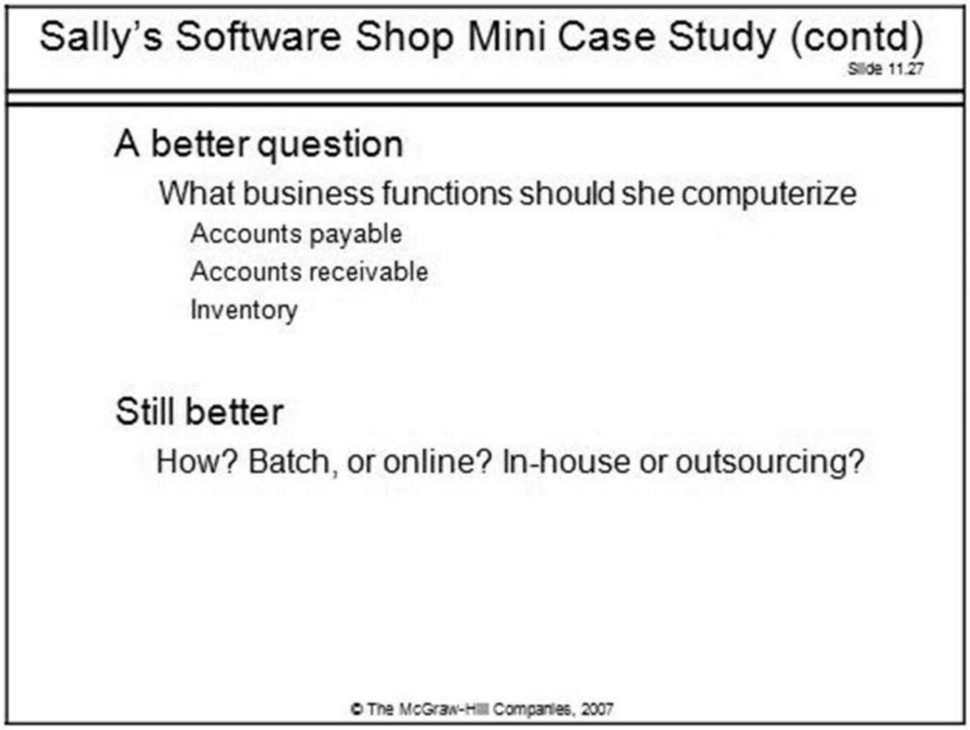
Many U.S. corporations use them for commercial products

Gane and Sarsen’s method is taught here because it is so widely used



11.3.1 Sally's Software Shop Mini Case Study

Sally's Software Shop buys software from various suppliers and sells it to the public. Popular software packages are kept in stock, but the rest must be ordered as required. Institutions and corporations are given credit facilities, as are some members of the public. Sally's Software Shop is doing well, with a monthly turnover of 300 packages at an average retail cost of $250 each. Despite her business success, Sally has been advised to computerize. Should she?



Sally's Software Shop Mini Case Study (contd)

Slide 11.27

A better question

What business functions should she computerize

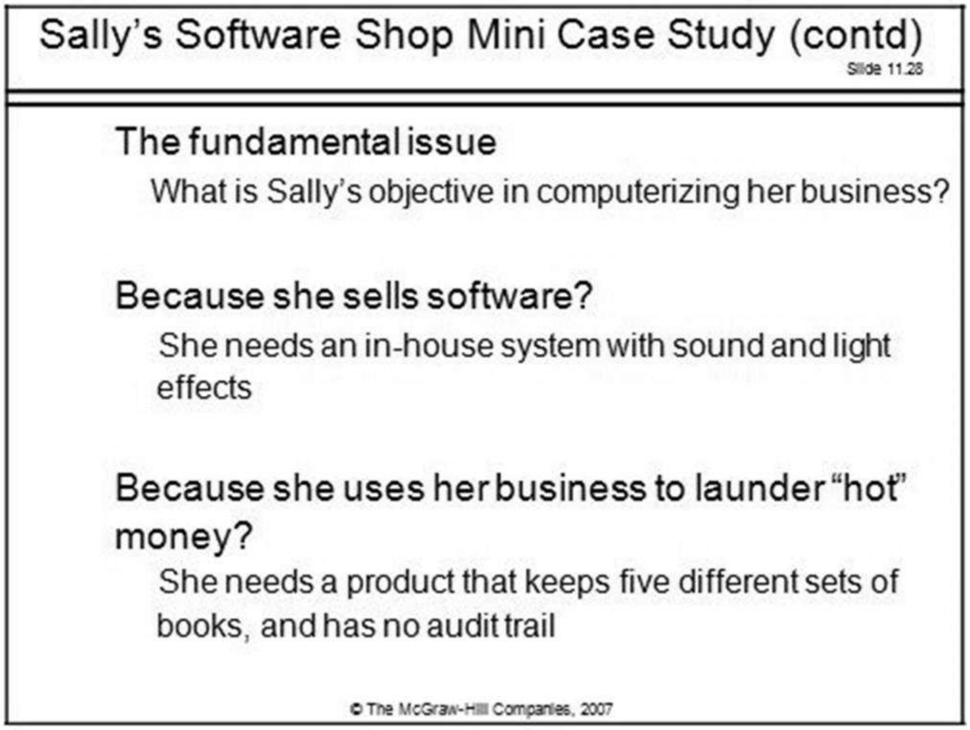
Accounts payable

Accounts receivable

Inventory

Still better

How? Batch, or online? In-house or outsourcing?



Sally's Software Shop Mini Case Study (contd)

Slide 11.28

The fundamental issue

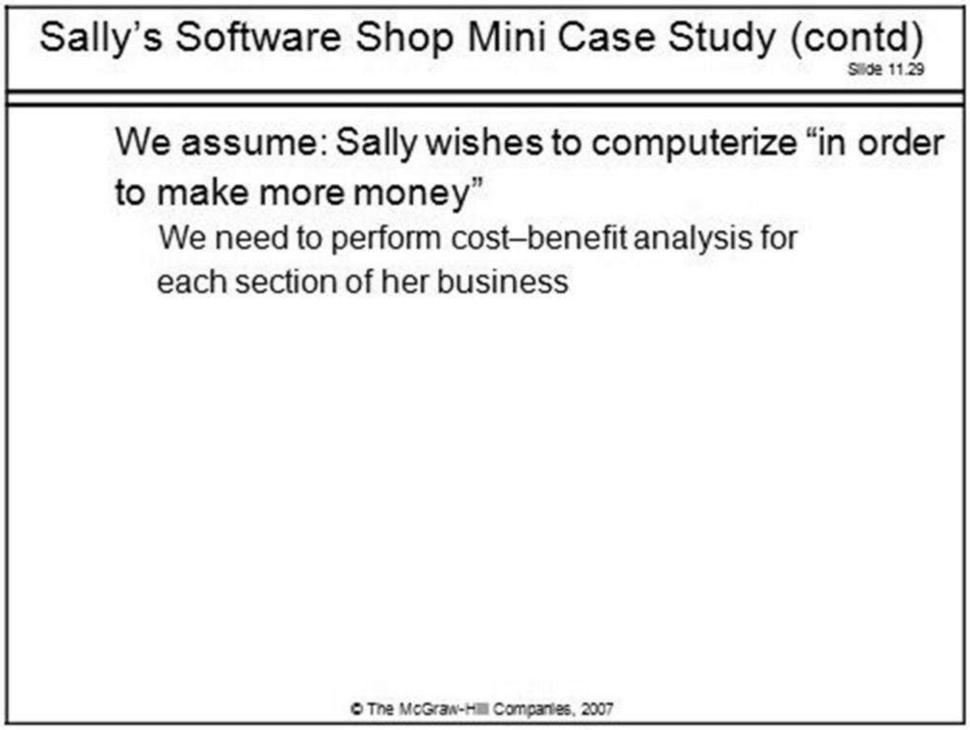
What is Sally's objective in computerizing her business?

Because she sells software?

She needs an in-house system with sound and light effects

Because she uses her business to launder "hot" money?

She needs a product that keeps five different sets of books, and has no audit trail

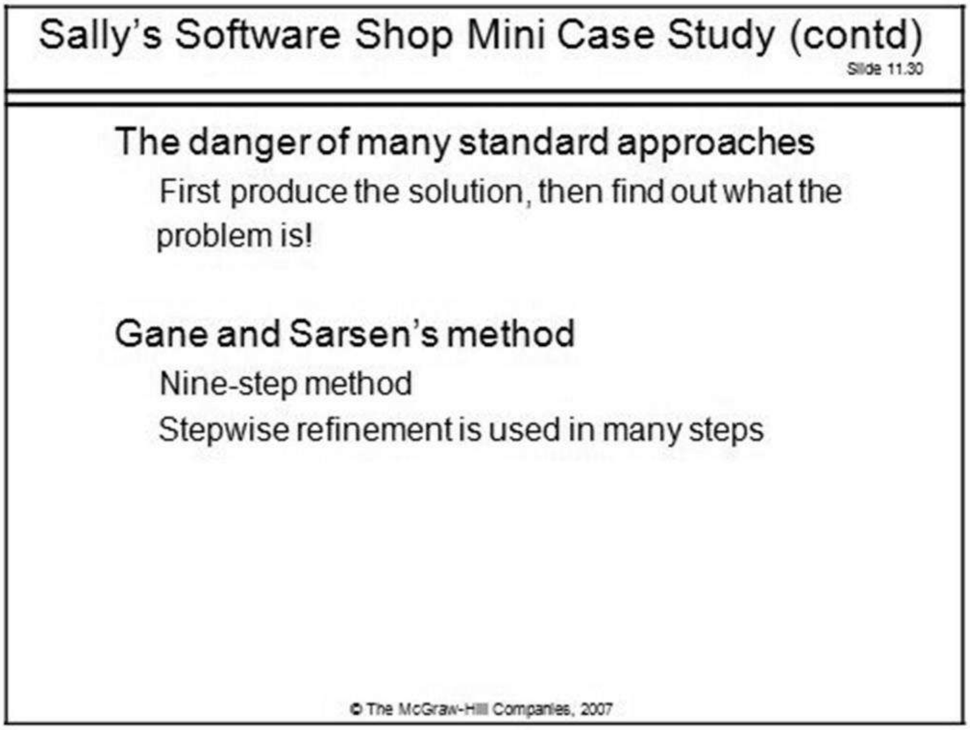


Sally’s Software Shop Mini Case Study (contd)

Slide 11.29

We assume: Sally wishes to computerize “in order to make more money”

- We need to perform cost–benefit analysis for each section of her business



Sally's Software Shop Mini Case Study (contd)

Slide 11.30

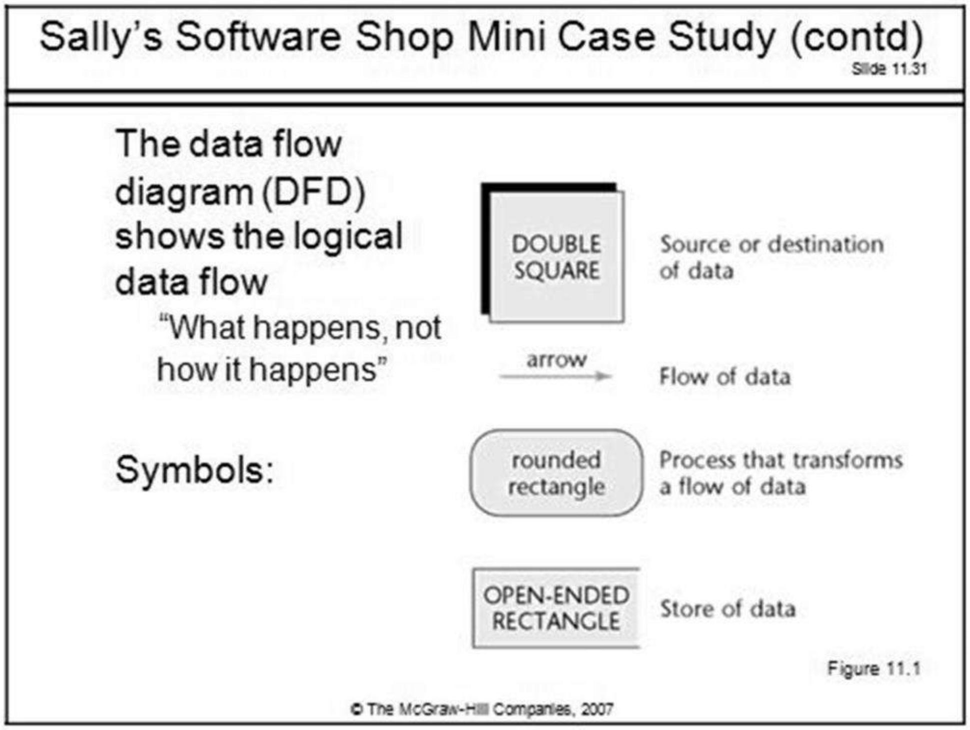
The danger of many standard approaches

First produce the solution, then find out what the problem is!

Gane and Sarsen's method

Nine-step method

Stepwise refinement is used in many steps



Sally's Software Shop Mini Case Study (contd)

Slide 11.31

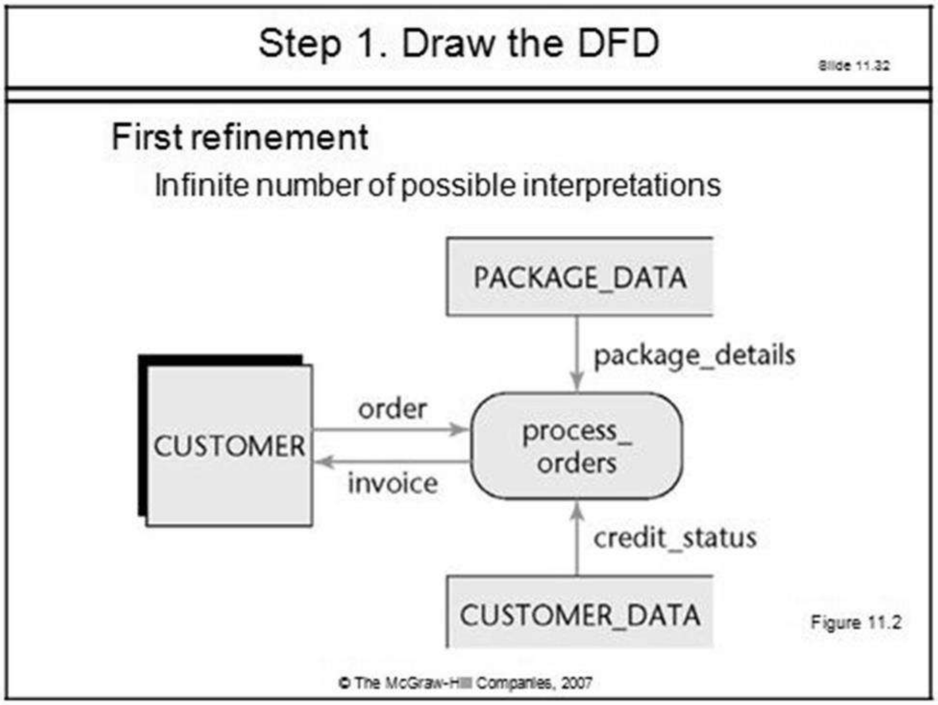
The data flow diagram (DFD) shows the logical data flow

"What happens, not how it happens"

Symbols:

DOUBLE SQUARE	Source or destination of data
arrow	Flow of data
rounded rectangle	Process that transforms a flow of data
OPEN-ENDED RECTANGLE	Store of data

Figure 11.1



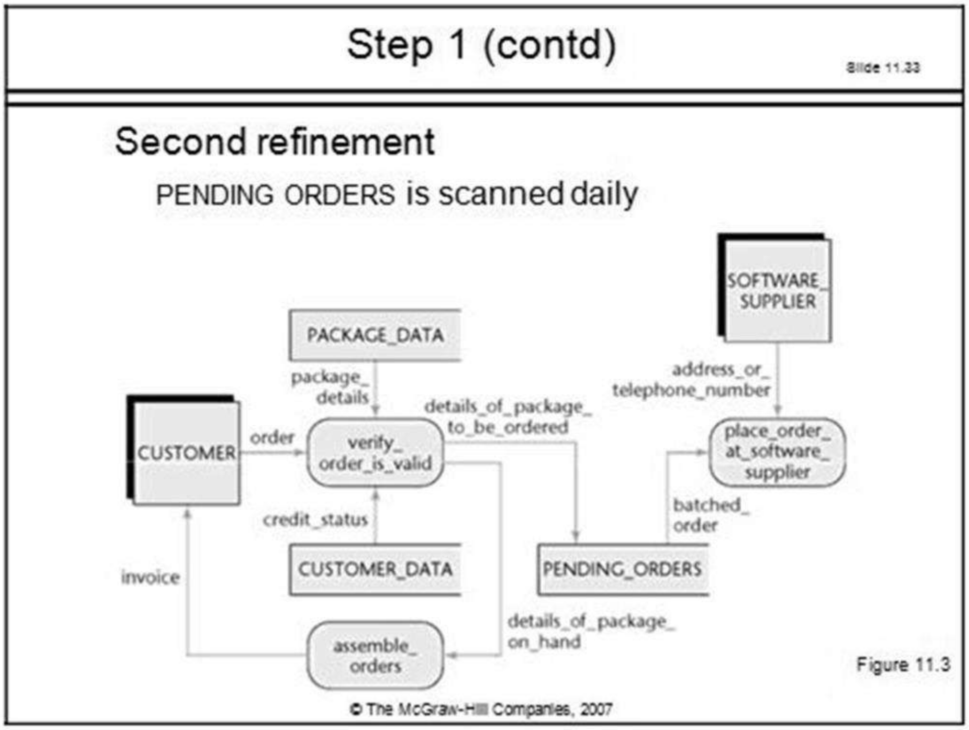
Step 1. Draw the DFD

Slide 11.32

First refinement

Infinite number of possible interpretations

Figure 11.2



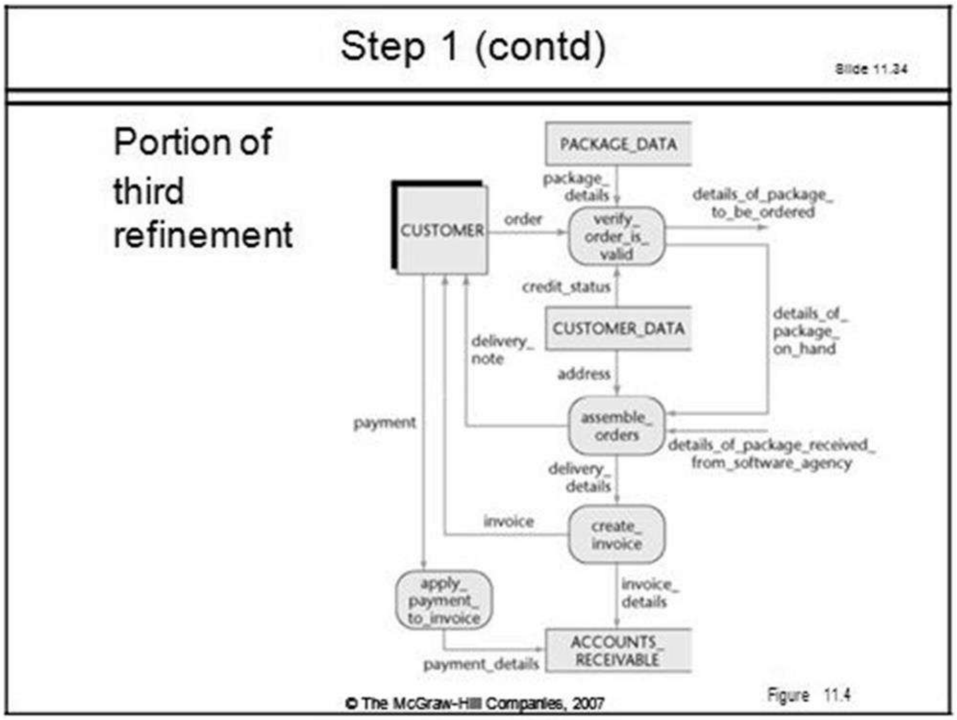
Step 1 (contd)

Slide 11.33

Second refinement

PENDING ORDERS is scanned daily

Figure 11.3

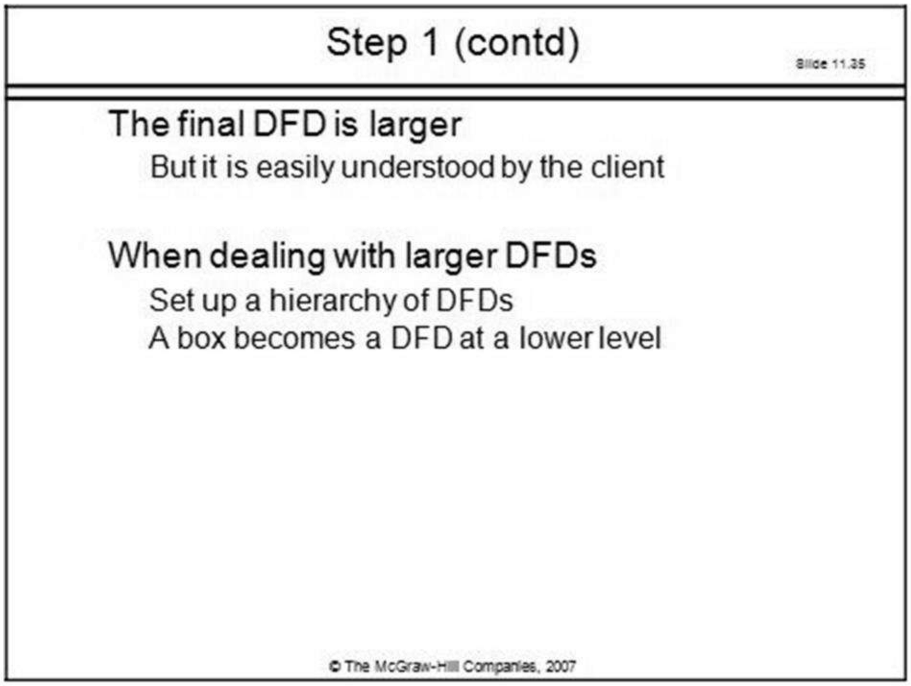


Step 1 (contd)

Slide 11.34

Portion of third refinement

Figure 11.4



Step 1 (contd)

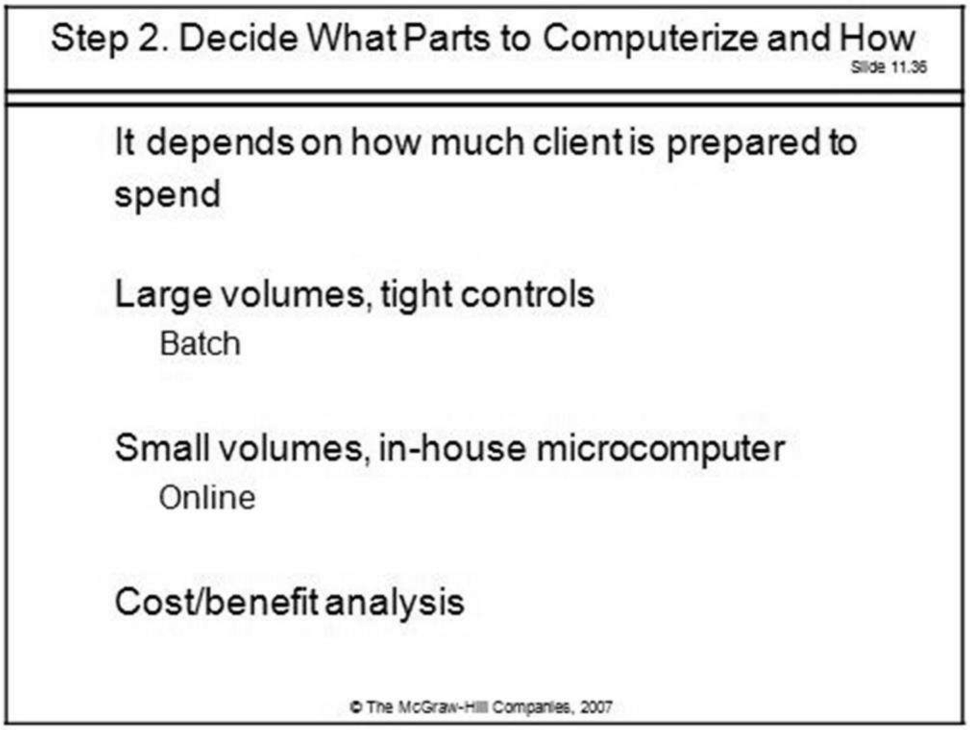
Slide 11.35

The final DFD is larger

- But it is easily understood by the client

When dealing with larger DFDs

- Set up a hierarchy of DFDs
- A box becomes a DFD at a lower level



Step 2. Decide What Parts to Computerize and How

Slide 11.36

It depends on how much client is prepared to spend

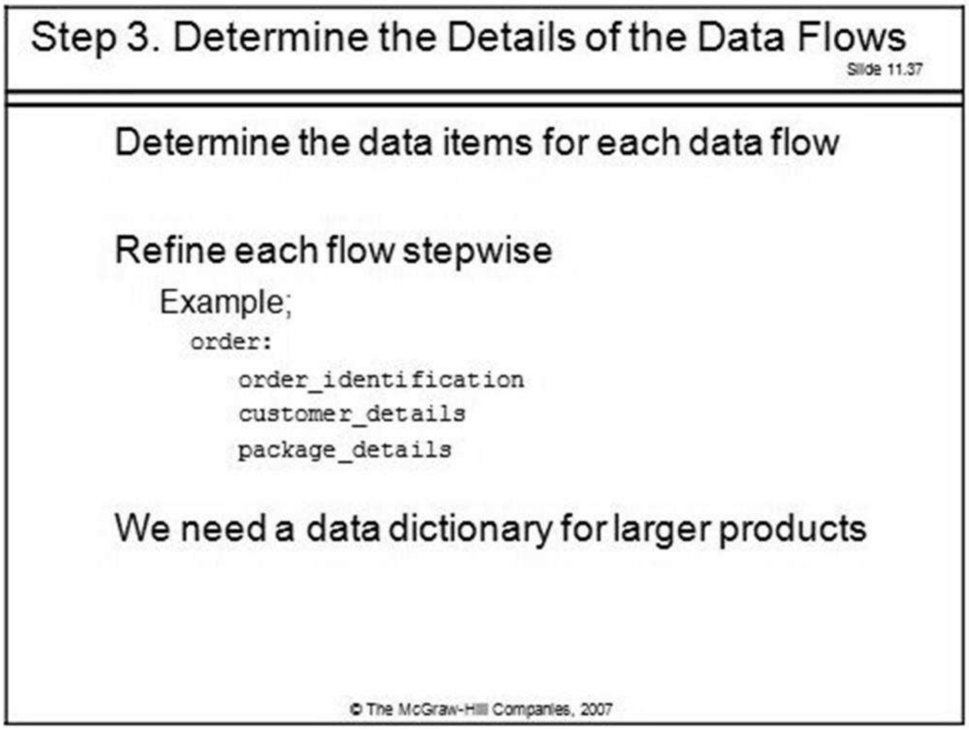
Large volumes, tight controls

- Batch

Small volumes, in-house microcomputer

- Online

Cost/benefit analysis



Step 3. Determine the Details of the Data Flows

Slide 11.37

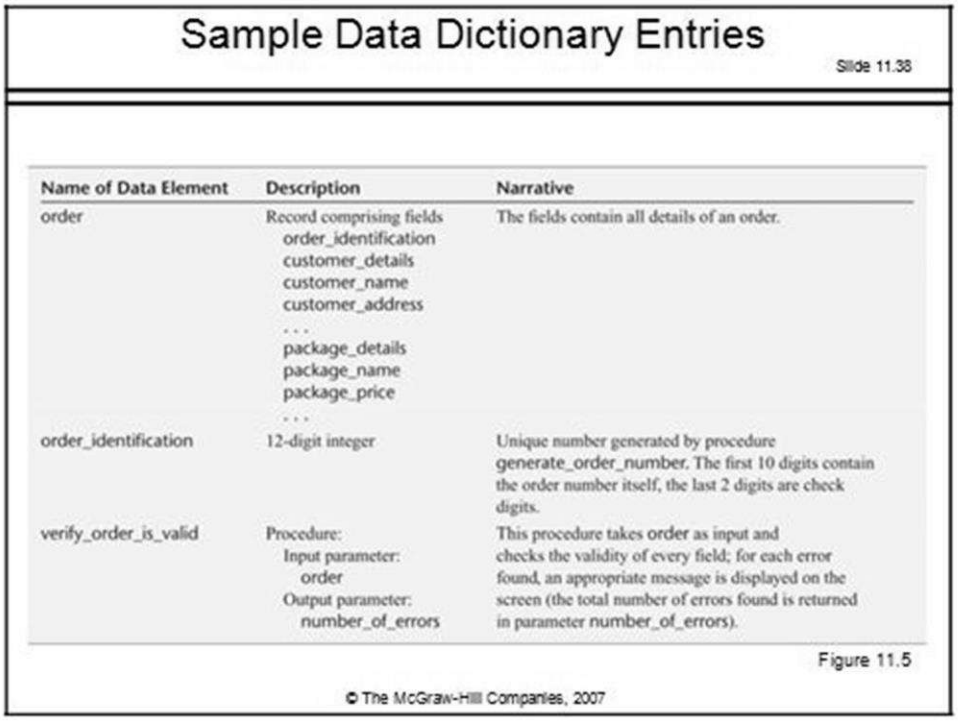
Determine the data items for each data flow

Refine each flow stepwise

Example;

```
order:
    order_identification
    customer_details
    package_details
```

We need a data dictionary for larger products

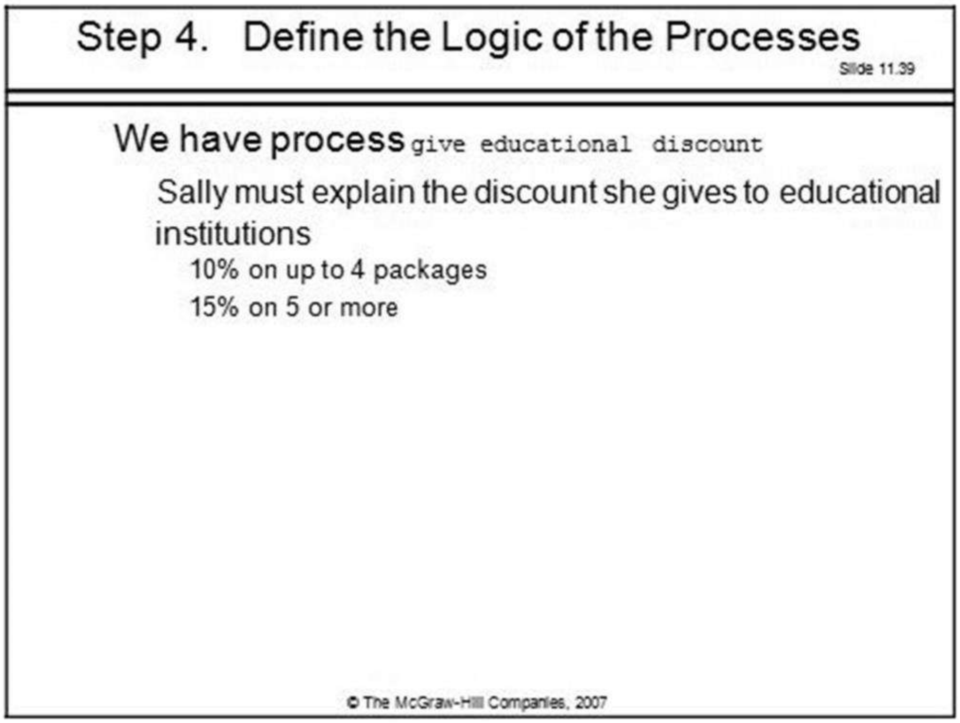


Sample Data Dictionary Entries

Slide 11.38

Name of Data Element	Description	Narrative
order	Record comprising fields order_identification customer_details customer_name customer_address . . . package_details package_name package_price . . .	The fields contain all details of an order.
order_identification	12-digit integer	Unique number generated by procedure generate_order_number. The first 10 digits contain the order number itself, the last 2 digits are check digits.
verify_order_is_valid	Procedure: Input parameter: order Output parameter: number_of_errors	This procedure takes order as input and checks the validity of every field; for each error found, an appropriate message is displayed on the screen (the total number of errors found is returned in parameter number_of_errors).

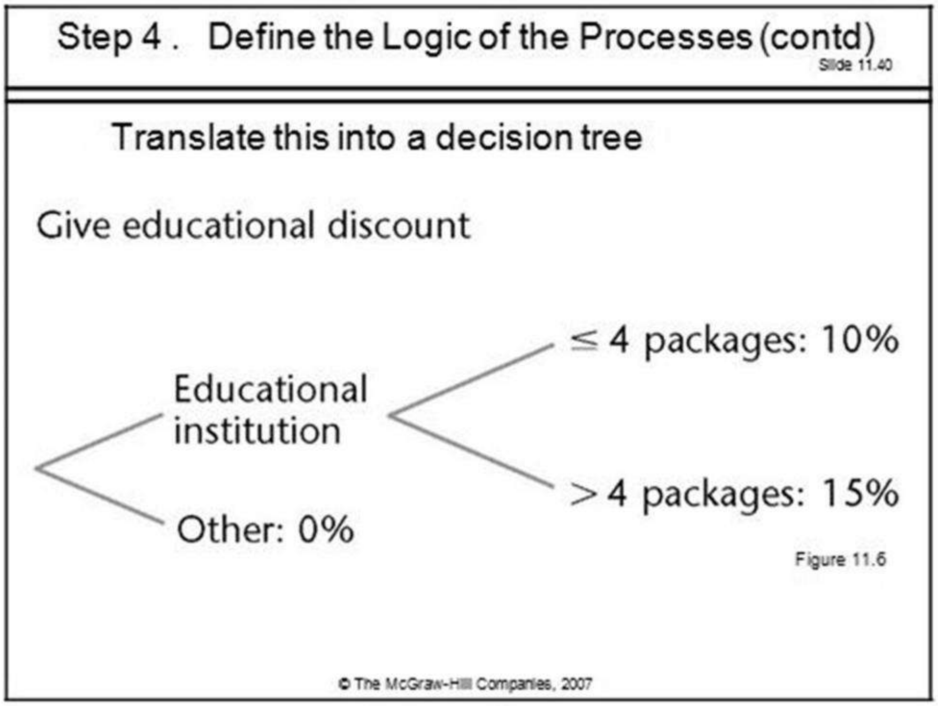
Figure 11.5



Step 4. Define the Logic of the Processes

Slide 11.39

- We have process `give educational discount`
  - Sally must explain the discount she gives to educational institutions
    - 10% on up to 4 packages
    - 15% on 5 or more



Step 4 . Define the Logic of the Processes (contd)

Slide 11.40

Translate this into a decision tree

Give educational discount

Educational institution

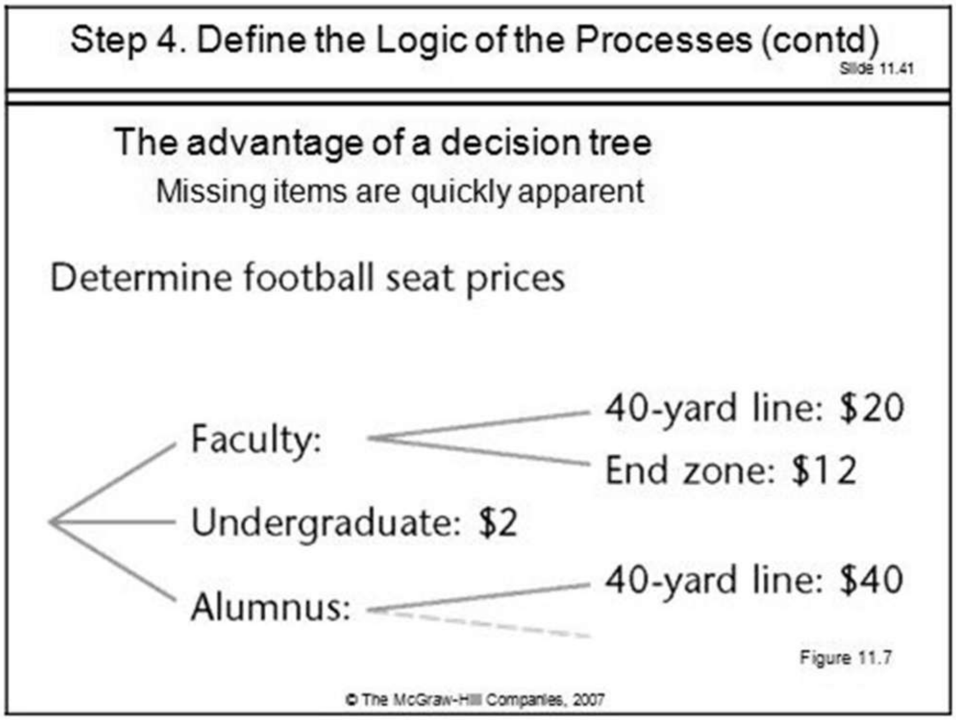
≤ 4 packages: 10%

> 4 packages: 15%

Other: 0%

Figure 11.6

© The McGraw-Hill Companies, 2007



Step 4. Define the Logic of the Processes (contd)

Slide 11.41

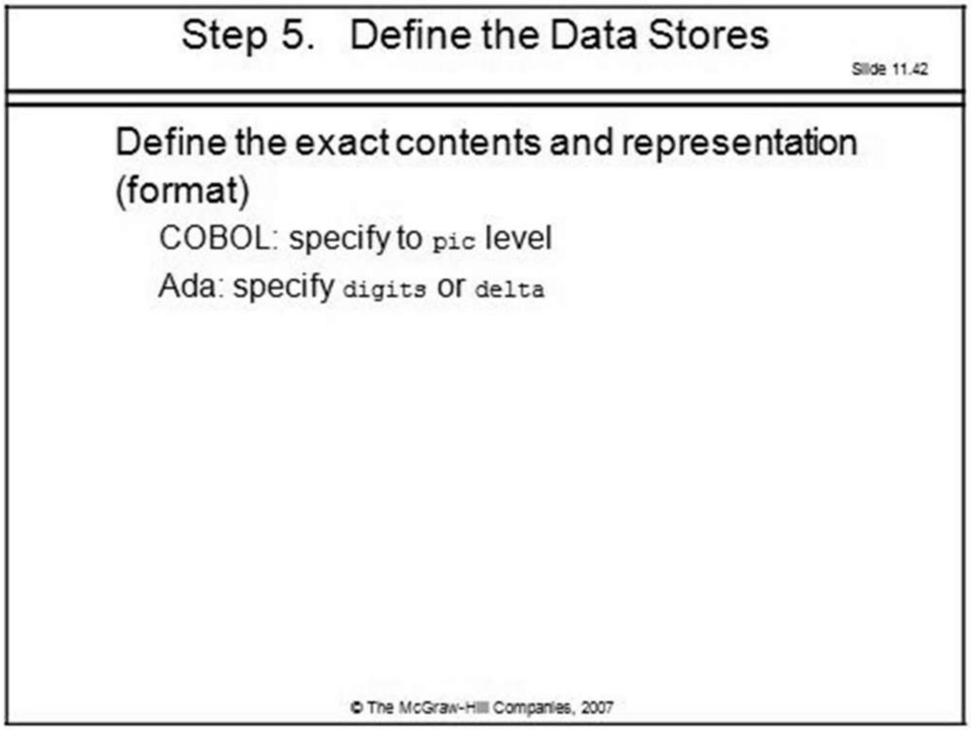
The advantage of a decision tree

- Missing items are quickly apparent

Determine football seat prices

Faculty:
40-yard line: $20
End zone: $12
Undergraduate: $2
Alumnus:
40-yard line: $40

Figure 11.7

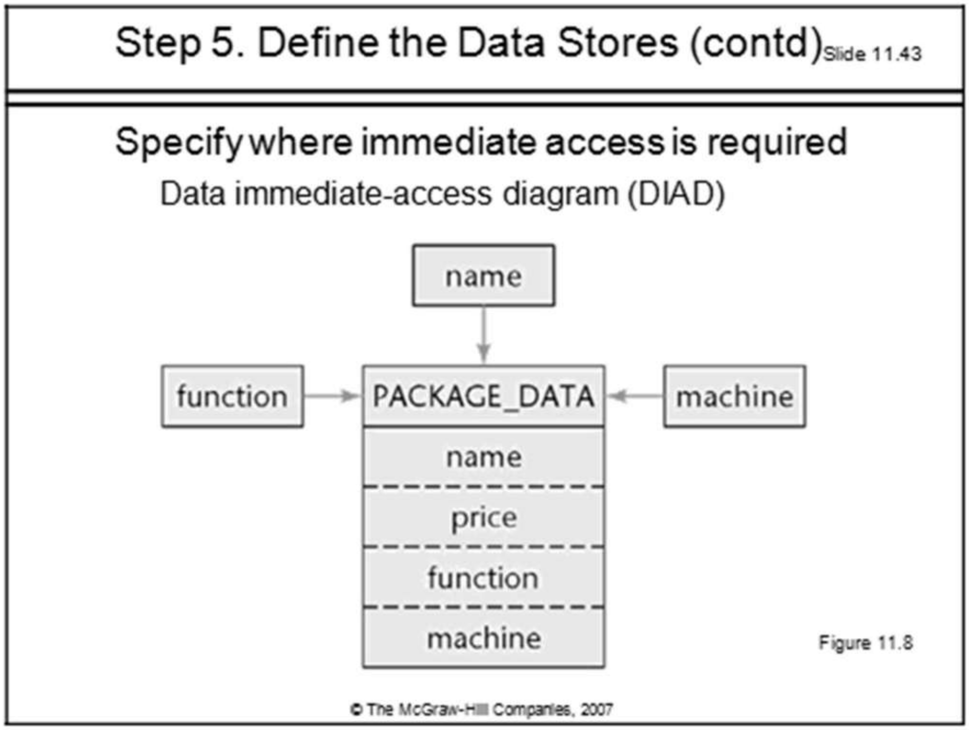


Step 5. Define the Data Stores

Slide 11.42

Define the exact contents and representation (format)

- COBOL: specify to `pic` level
- Ada: specify `digits` or `delta`



Step 5. Define the Data Stores (contd)

Slide 11.43

Specify where immediate access is required

Data immediate-access diagram (DIAD)

name

function

PACKAGE_DATA

machine

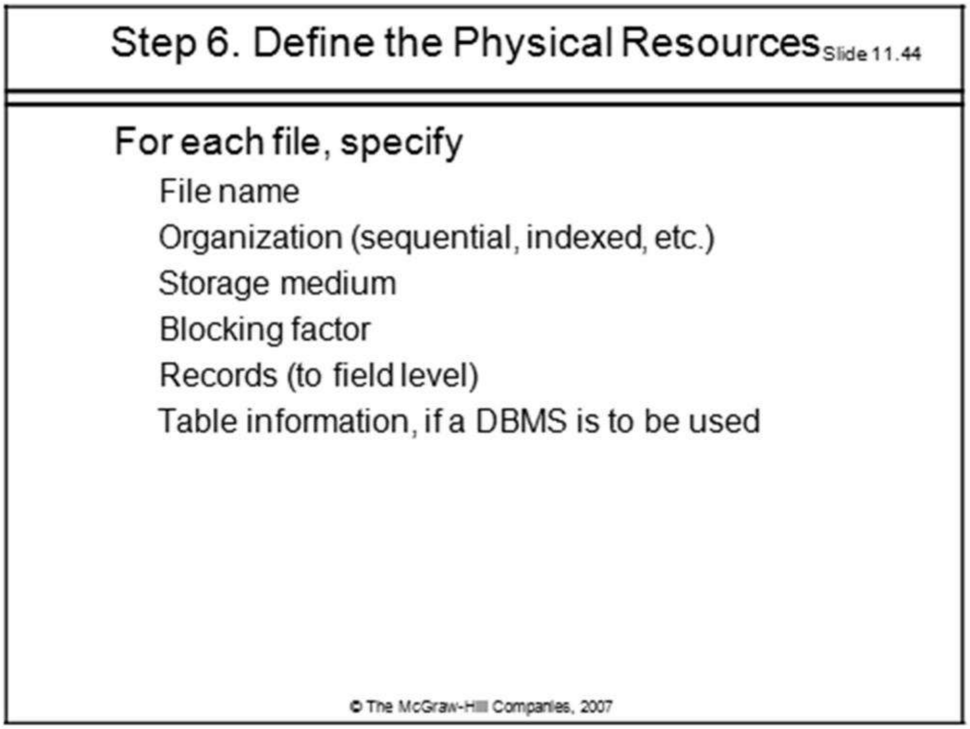
name

price

function

machine

Figure 11.8

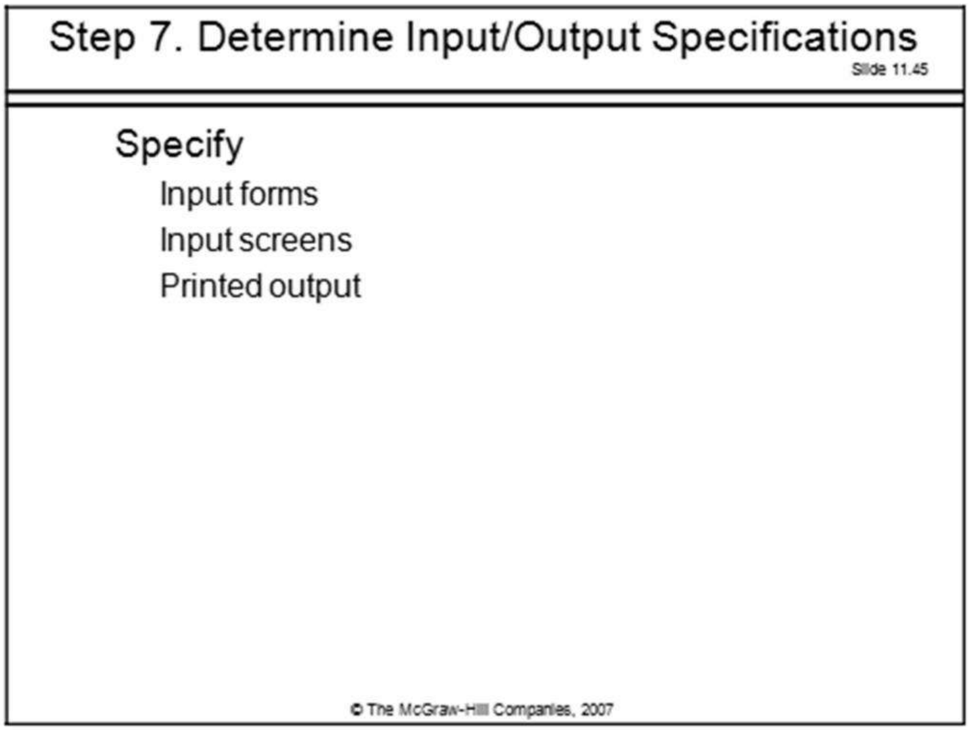


Step 6. Define the Physical Resources

Slide 11.44

For each file, specify

- File name
- Organization (sequential, indexed, etc.)
- Storage medium
- Blocking factor
- Records (to field level)
- Table information, if a DBMS is to be used

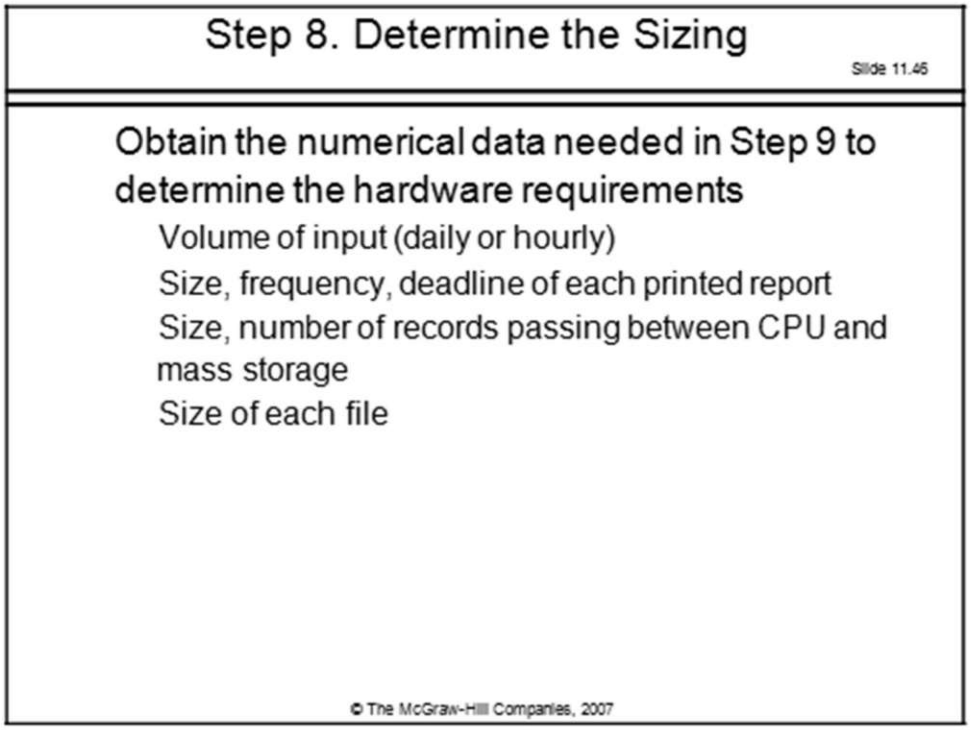


Step 7. Determine Input/Output Specifications

Slide 11.45

Specify

- Input forms
- Input screens
- Printed output

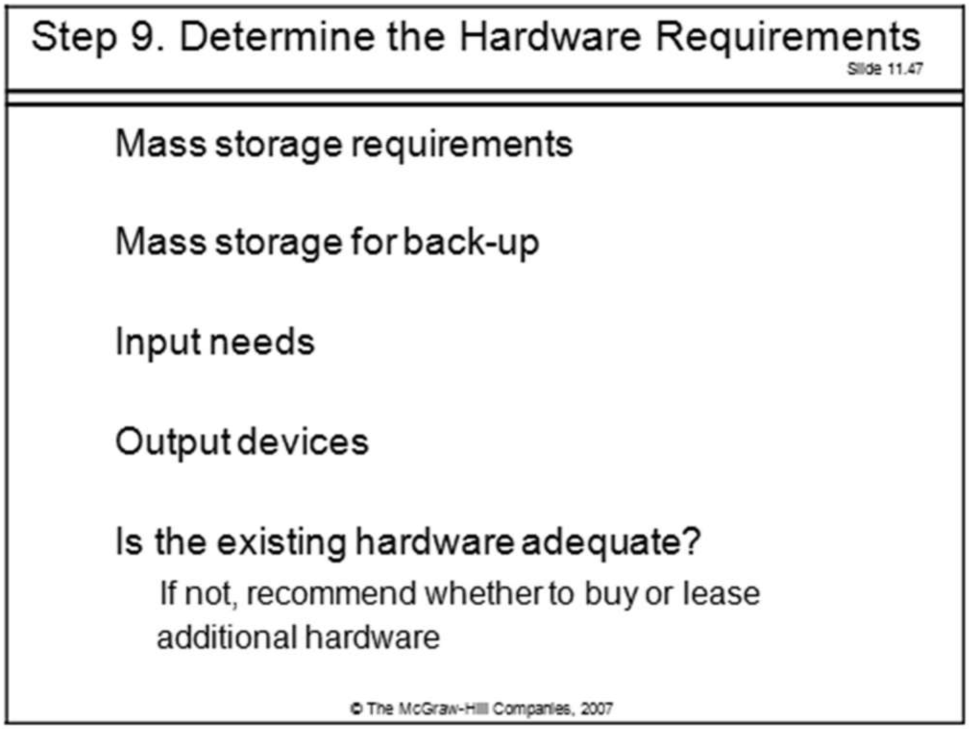


Step 8. Determine the Sizing

Slide 11.46

Obtain the numerical data needed in Step 9 to determine the hardware requirements

- Volume of input (daily or hourly)
- Size, frequency, deadline of each printed report
- Size, number of records passing between CPU and mass storage
- Size of each file

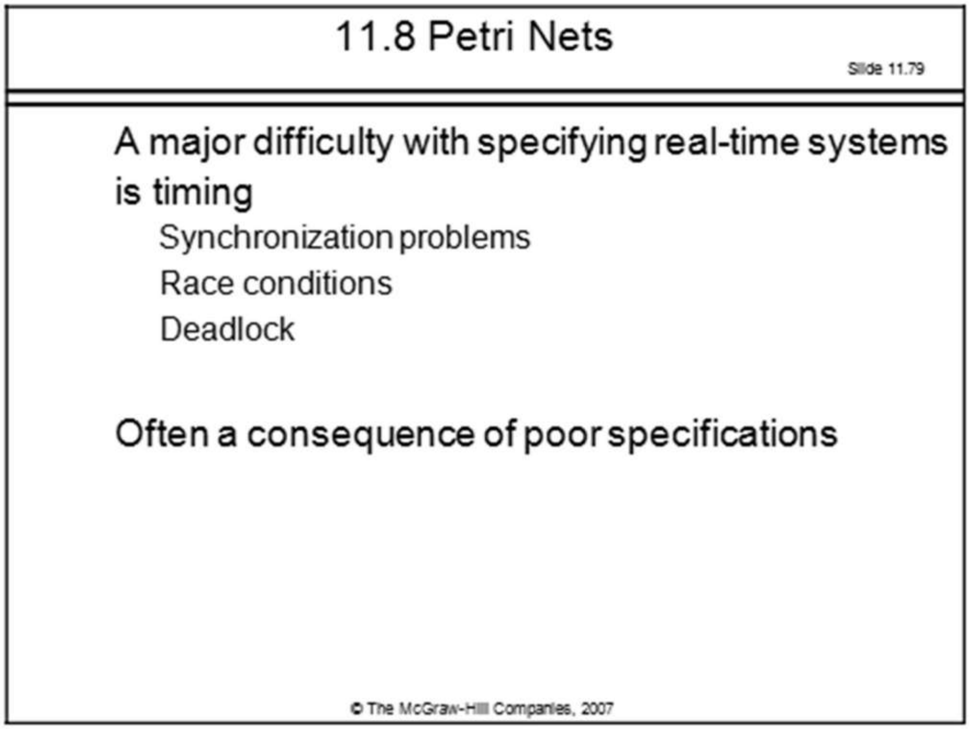


Step 9. Determine the Hardware Requirements

Slide 11.47

- Mass storage requirements
- Mass storage for back-up
- Input needs
- Output devices
- Is the existing hardware adequate?
  - If not, recommend whether to buy or lease additional hardware

PETRI NETS:



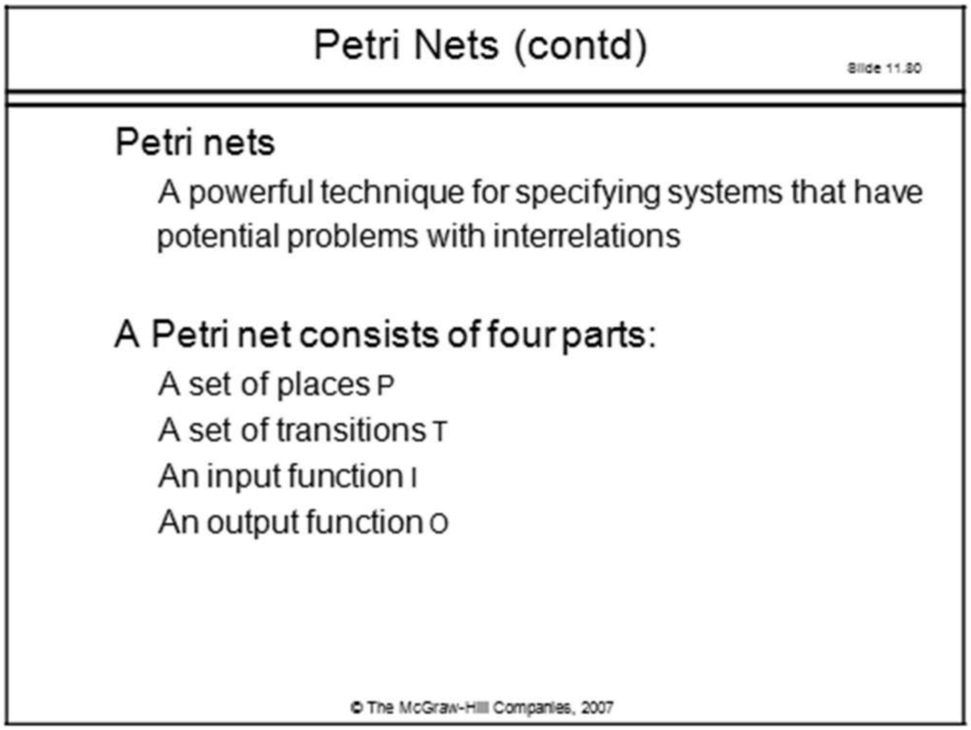
11.8 Petri Nets

Slide 11.79

A major difficulty with specifying real-time systems is timing

- Synchronization problems
- Race conditions
- Deadlock

Often a consequence of poor specifications



Petri Nets (contd)

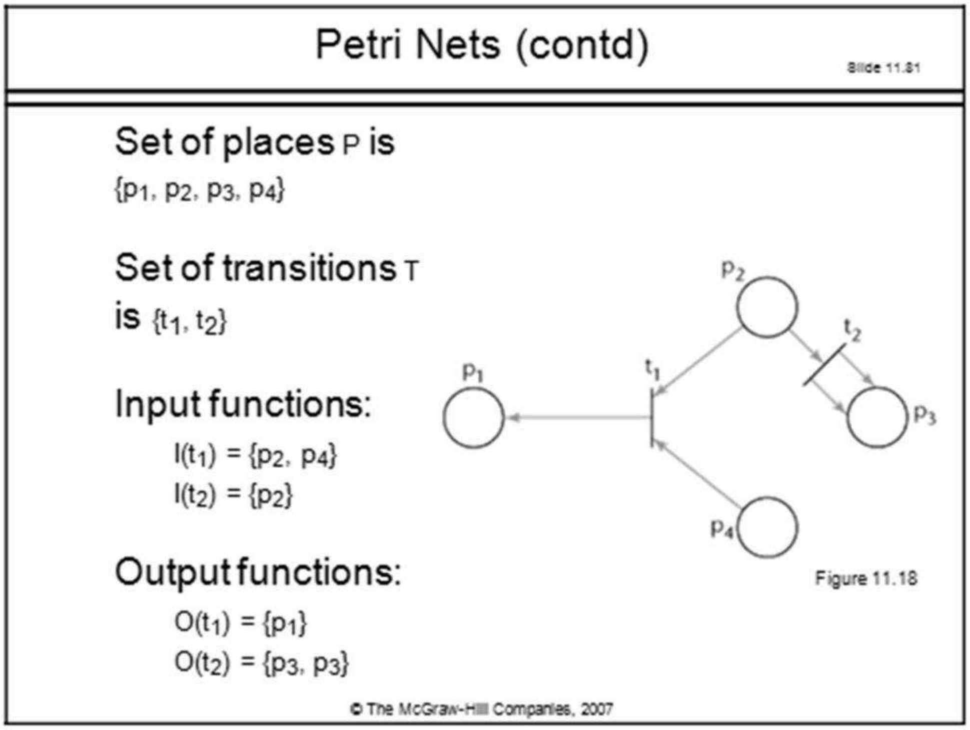
Slide 11.80

Petri nets

- A powerful technique for specifying systems that have potential problems with interrelations

A Petri net consists of four parts:

- A set of places P
- A set of transitions T
- An input function I
- An output function O



Petri Nets (contd)

Slide 11.81

Set of places P is
{p_1, p_2, p_3, p_4}

Set of transitions T
is {t_1, t_2}

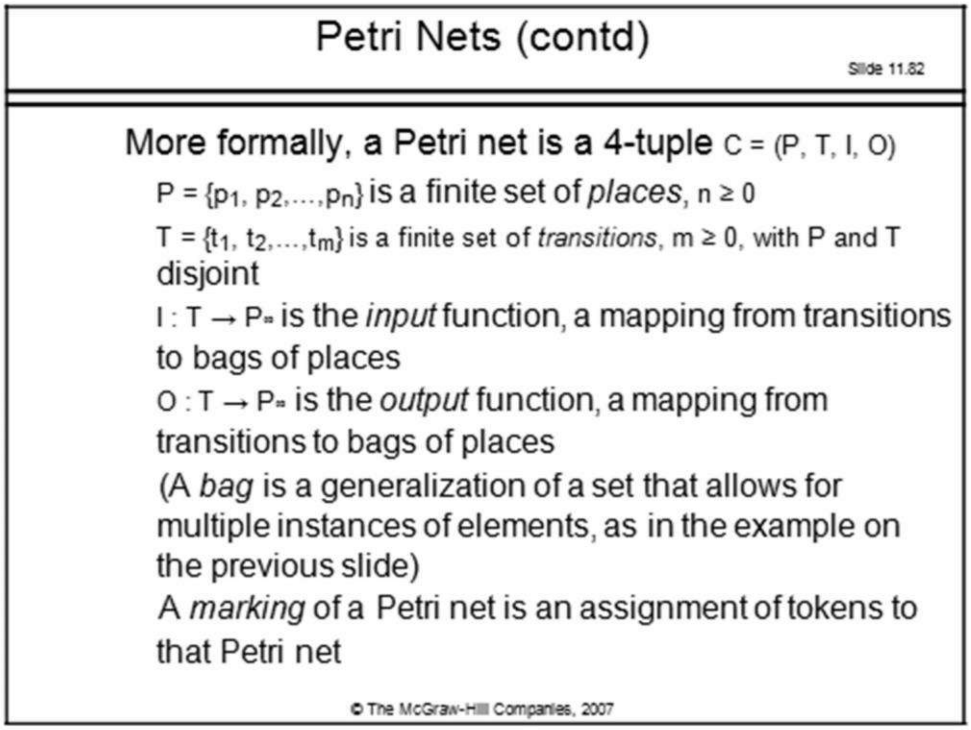
Input functions:

$I(t_1) = \{p_2, p_4\}$
$I(t_2) = \{p_2\}$

Output functions:

$O(t_1) = \{p_1\}$
$O(t_2) = \{p_3, p_3\}$

Figure 11.18



Petri Nets (contd)

Slide 11.82

More formally, a Petri net is a 4-tuple $C = (P, T, I, O)$

$P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of *places*, $n \geq 0$

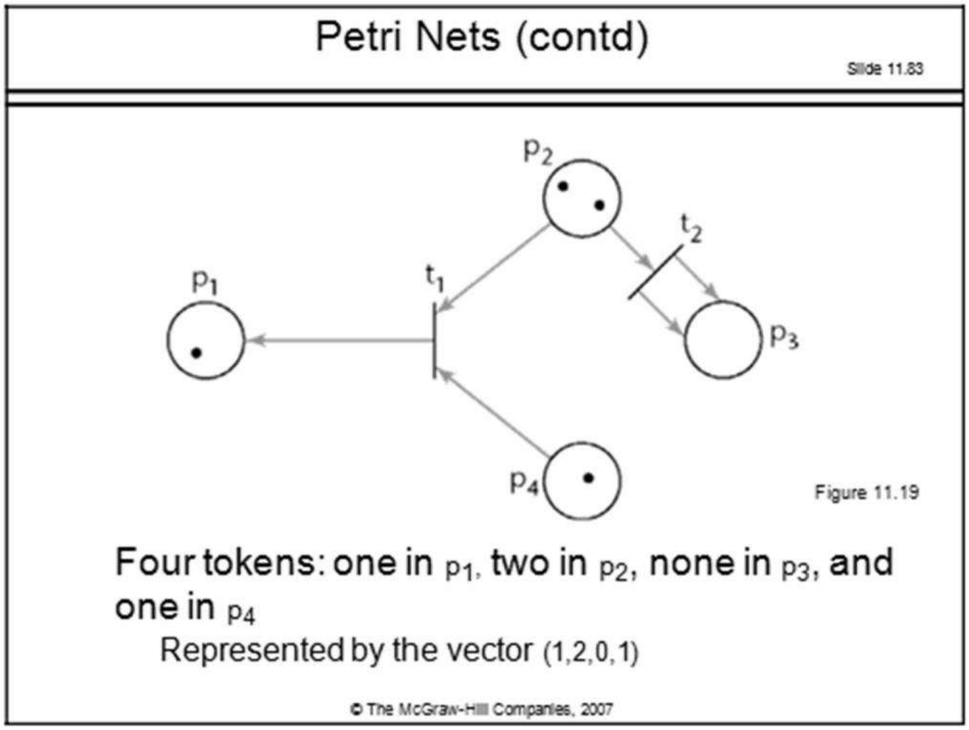
$T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of *transitions*, $m \geq 0$, with P and T disjoint

$I : T \rightarrow P^{\infty}$ is the *input* function, a mapping from transitions to bags of places

$O : T \rightarrow P^{\infty}$ is the *output* function, a mapping from transitions to bags of places

(A *bag* is a generalization of a set that allows for multiple instances of elements, as in the example on the previous slide)

A *marking* of a Petri net is an assignment of tokens to that Petri net



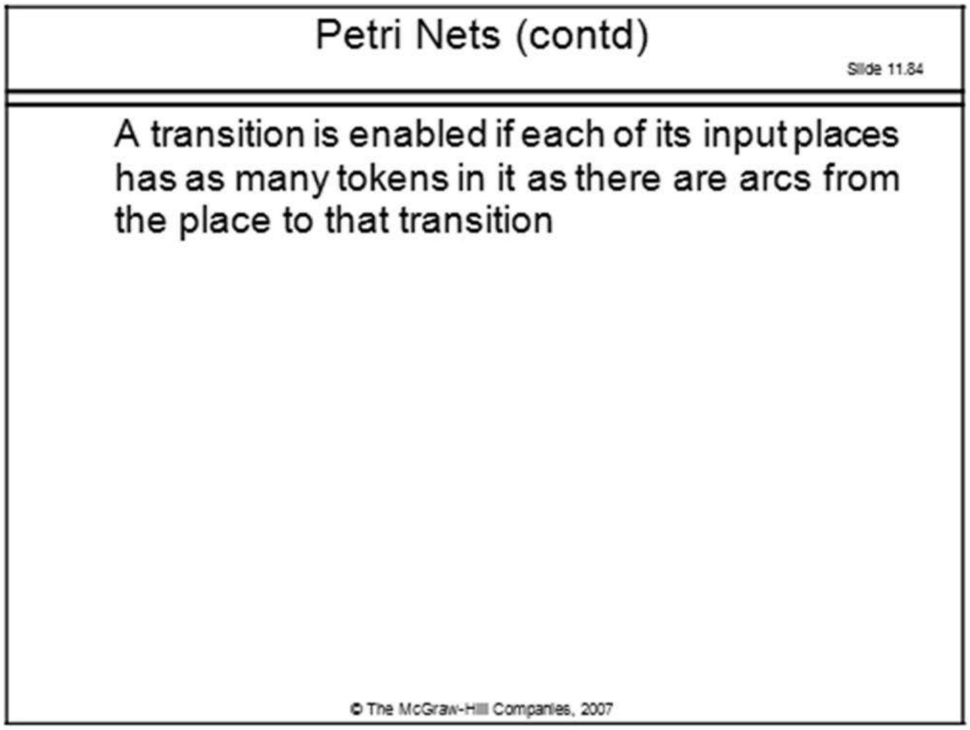
Petri Nets (contd)

Slide 11.83

Figure 11.19

Four tokens: one in p_1, two in p_2, none in p_3, and one in p_4

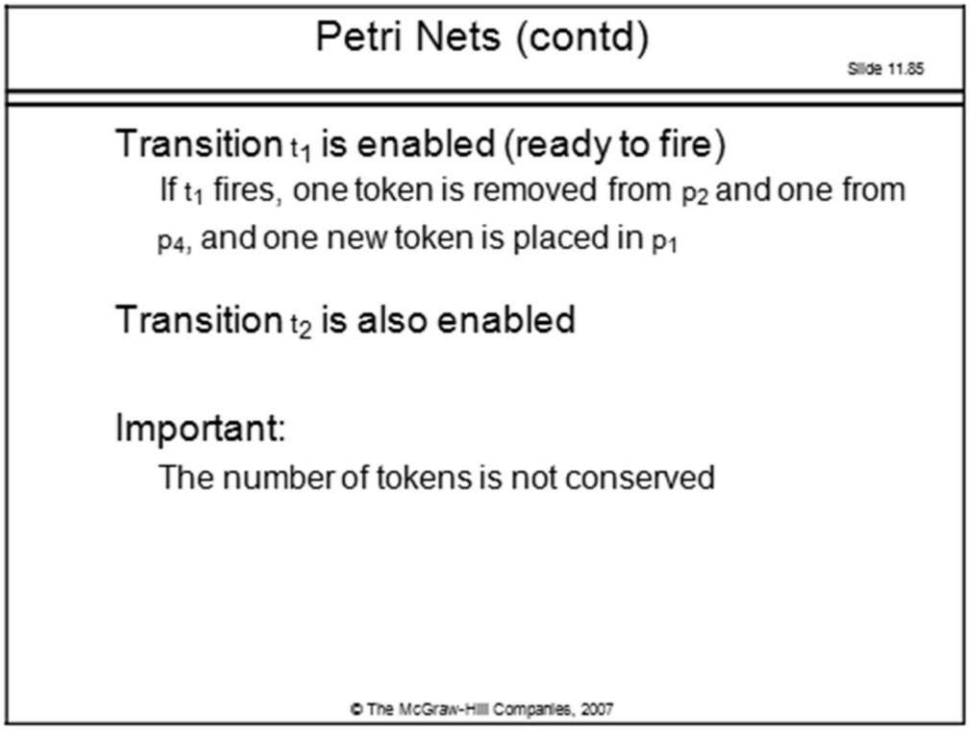
Represented by the vector (1,2,0,1)



Petri Nets (contd)

Slide 11.84

A transition is enabled if each of its input places has as many tokens in it as there are arcs from the place to that transition



Petri Nets (contd)

Slide 11.85

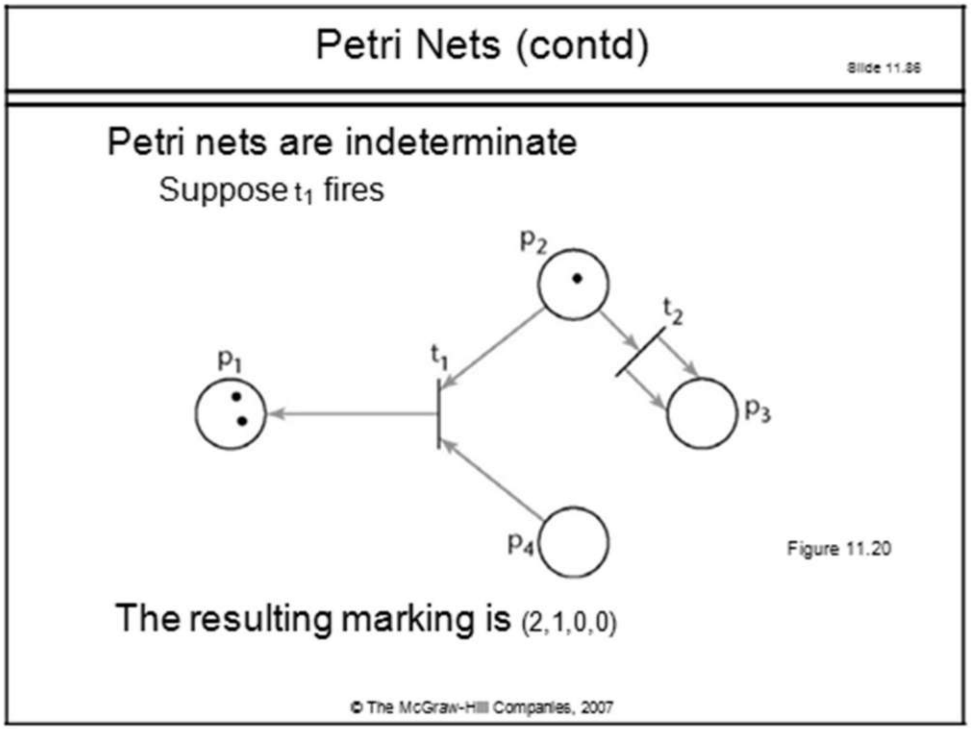
Transition t_1 is enabled (ready to fire)

If t_1 fires, one token is removed from p_2 and one from p_4, and one new token is placed in p_1

Transition t_2 is also enabled

Important:

The number of tokens is not conserved



Petri Nets (contd)

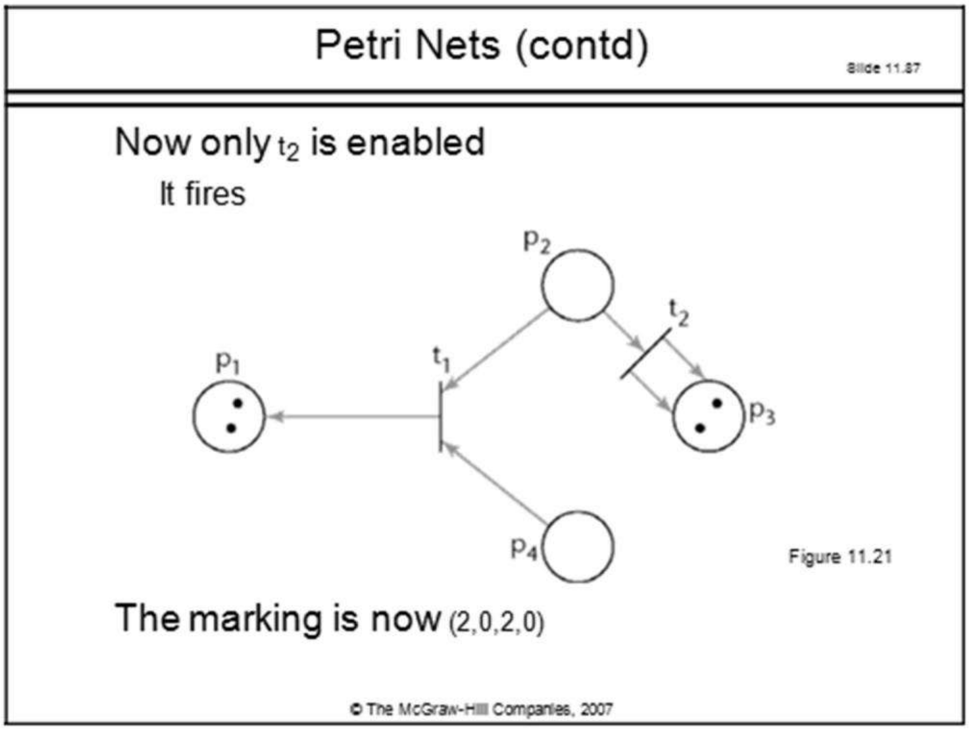
Slide 11.86

Petri nets are indeterminate

Suppose t_1 fires

Figure 11.20

The resulting marking is (2,1,0,0)



Petri Nets (contd)

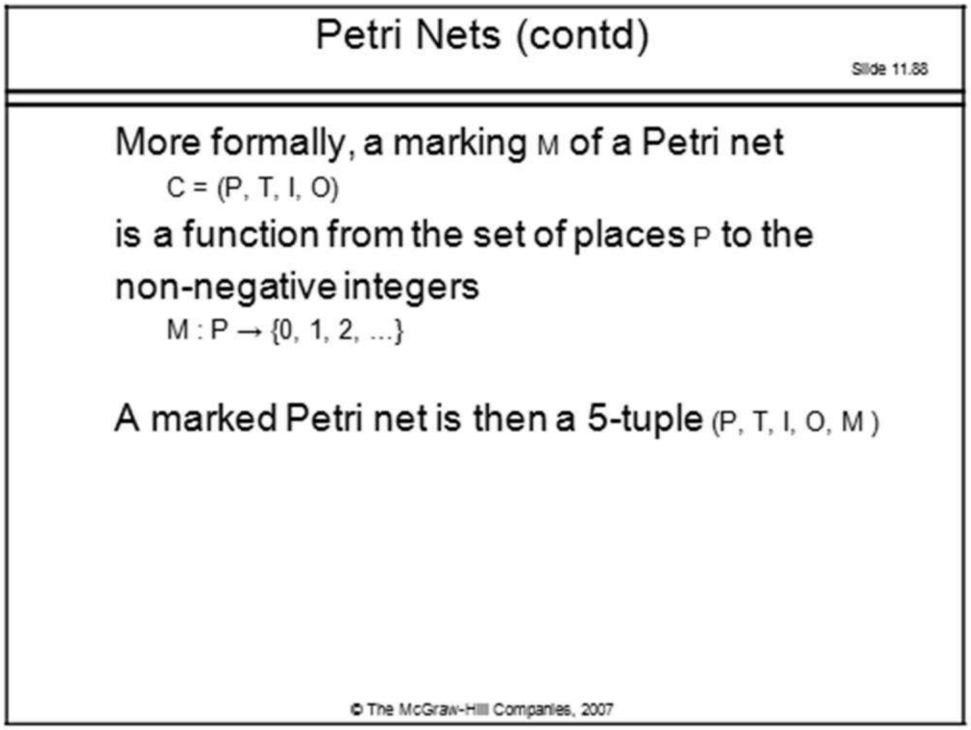
Slide 11.87

Now only t_2 is enabled

It fires

Figure 11.21

The marking is now (2,0,2,0)



Petri Nets (contd)

Slide 11.88

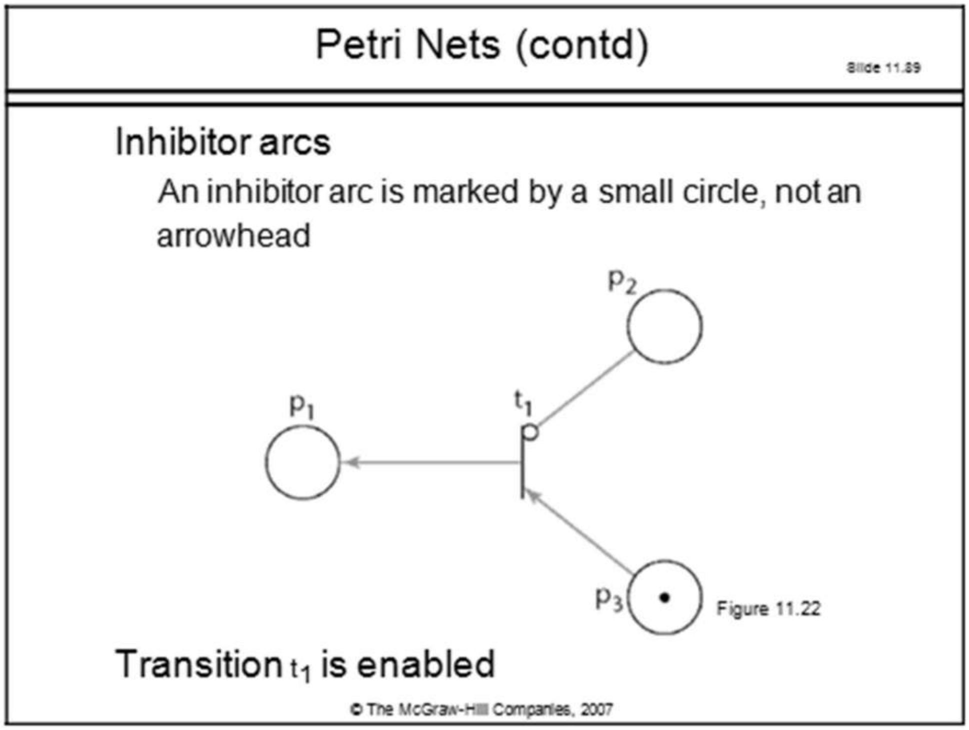
More formally, a marking M of a Petri net

C = (P, T, I, O)

is a function from the set of places P to the non-negative integers

$M : P \rightarrow \{0, 1, 2, \ldots\}$

A marked Petri net is then a 5-tuple (P, T, I, O, M)



Petri Nets (contd)

Slide 11.89

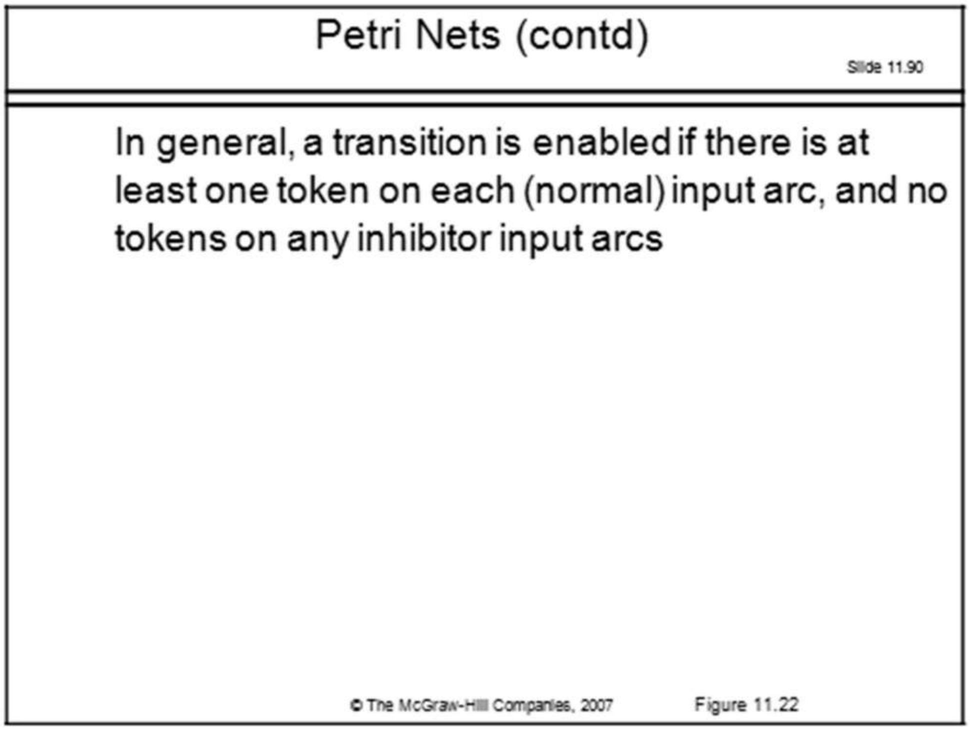
Inhibitor arcs

An inhibitor arc is marked by a small circle, not an arrowhead

Figure 11.22

Transition t_1 is enabled

© The McGraw-Hill Companies, 2007



Petri Nets (contd)

Slide 11.90

In general, a transition is enabled if there is at least one token on each (normal) input arc, and no tokens on any inhibitor input arcs

© The McGraw-Hill Companies, 2007 Figure 11.22

Unit-III DESIGN CONCEPTS

Software design is the process by which an agent creates a specification of a **software** artifact, intended to accomplish goals, using a set of primitive components and subject to constraints.

Design is a meaningful engineering representation of something that is to be built. Itcan be traced to a customer's requirements and at the same time assessed for quality against asset of predefined criteria for "good" design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components. The concepts and principles discussed in this chapter apply to all four

DESIGN CONCEPTS:

ABSTRACTION:

When we consider a modular solution to any problem, many *levels of abstraction* can be posed. 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 procedural orientation is taken. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution

A *procedural abstraction:* is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g, walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A *data abstraction :* is a named collection of data that describes a data object . In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

Control abstraction: is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the *synchronization semaphore* [KAI83] used to coordinate activities in an operating system

Software Architecture :

Software architecture alludes to "the overall structure of the software and the ways inwhich that structure provides conceptual integrity for a system" [SHA95a]. In its simplestform, architecture is the hierarchical structure of program components (modules),the manner in which these components interact and the structure of data thatare used by the components. In a broader sense, however, *components* can be generalized

Structural properties:

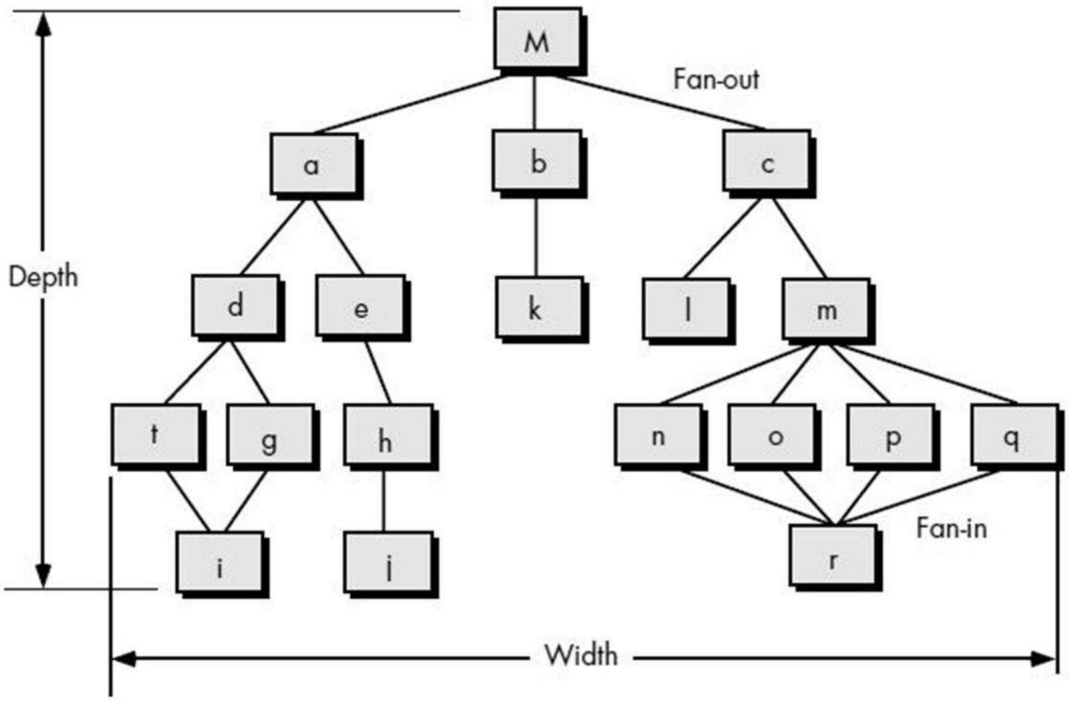
This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via thevinvocation 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.



PATTERNS:

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Objectoriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.[4]

There are many types of design patterns: Structural patterns address concerns related to the high level structure of an application being developed. Computational patterns address concerns related to the identification of key computations. Algorithm strategy patterns address concerns related to high level strategies that describe how to exploit application characteristic on a computation platform. Implementation strategy patterns address concerns related to the realization of the source code to support how the program itself is organized and the common data structures specific to parallel programming. Execution patterns address concerns related to the support of the execution of an application, including the strategies in executing streams of tasks and building blocks to support the synchronization between tasks.

MODULARITY:

The concept of modularity in computer software has been espoused for almost five decades. Software architecture (described in Section 13.4.4) embodies modularity; that is, software is divided into separately named and addressable components, often called *modules,* that are integrated to satisfy problem requirements.

Let C(x) be a function that defines the perceived complexity of a problem x, and E(x) be a function that defines the effort (in time) required to solve a problem x. For two problems, p1 and p2, if $C(p1) > C(p2)$ (13-1a) it follows that

$E(p1) > E(p2)$ (13-1b)

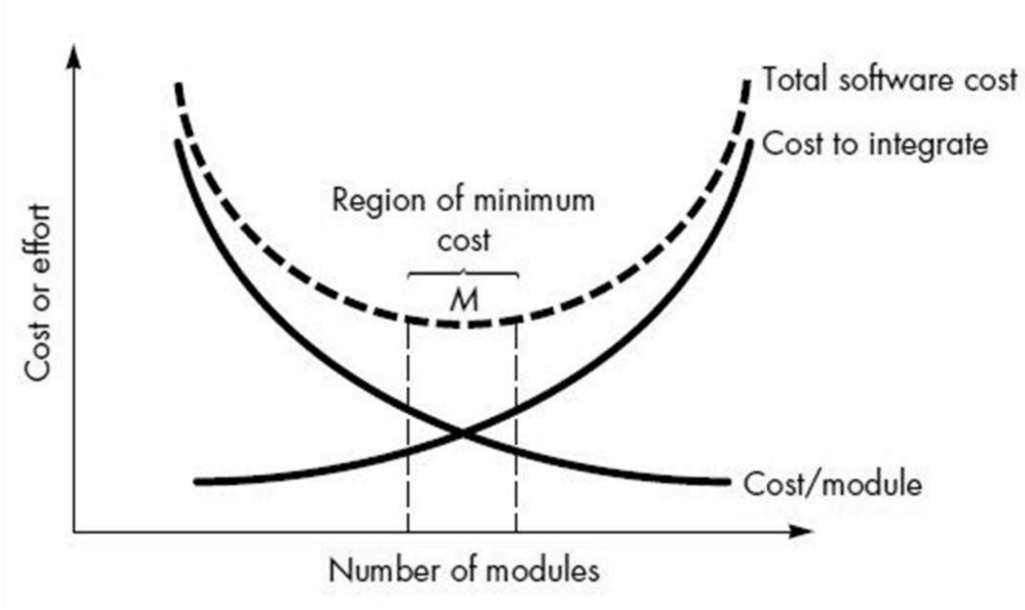
As a general case, this result is intuitively obvious. It does take more time to solve adifficult problem.

Another interesting characteristic has been uncovered through experimentation in human problem solving. That is, $C(p1 + p2) > C(p1) + C(p2)$ (13-2)

Expression (13-2) implies that the perceived complexity of a problem that combines *p1* and *p2* is greater than the perceived complexity when each problem is considered separately. Considering Expression (13-2) and the condition implied by Expressions (13-1), it follows that

$E(p1 + p2) > E(p1) + E(p2)$ (13-3)

This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in Expression has important implications with regard to modularity and software. It is, in fact, an argument for modularity. It is possible to conclude from Expression



Modular Decomposability: If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular compensability: If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability: If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

Modular continuity: If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.

Modular protection:If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

1. Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Functional independence:

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In landmark papers on software design Parnas [PAR72] and Wirth [WIR71] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [STE74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific sub functionof requirements and has a simple interface when viewed from other parts of the program structure.

It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). 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.

1. Refinement - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.

REFACTORING:

Refactoring is changing a software system by improving its internal structure without changing its external behavior, i.e. it is a technique to restructure the code in a disciplined way.

- It makes the software easier to understand and cheaper to modify.
- To find the bugs

  It helps in finding the Bugs present in the program.
- To program faster

  It helps us to do the coding/programming faster as we have better understanding of the situation.
- <u>When you add a function</u>

  Helps you to understand the code you are modifying.

  Sometimes the existing design does not allow you to easily add the feature.
- <u>When you need to fix a bug</u>

  If you get a bug report its a sign the code needs refactoring because the code was not clear enough for you to see the bug in the first place.
- <u>When you do a Code Review</u>
  - Code reviews help spread knowledge through the development team.
  - Works best with small review groups

Properties of Refactoring

- Preserve Correctness • One step at a time
- Frequent Testing.

DESIGN CLASSES:

A **design class** is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics. **1. User 1.interface classes**

- These classes are designed for Human Computer Interaction(HCI).

These interface classes define all abstraction which is required for Human Computer Interaction(HCI).

- **2. Business domain classes**
- These classes are commonly refinements of the analysis classes.

- These classes are recognized as attributes and methods which are required to implement the elements of the business domain.

3.Process classes:

It implement the lower level business abstraction which is needed to completely manage the business domain class.

4. Persistence classes:

It shows data stores that will persist behind the execution of the software.

5. System Classes:

System classes implement software management and control functions that allow to operate and communicate in computing environment and outside world.

DESIGN HEURISTIC:

1. The main goal of heuristic evaluations is to identify any problems associated with the design of user interfaces. Usability consultant Jakob Nielsen developed this method on the basis of several years of experience in teaching and consulting about usability engineering.

2. Heuristic evaluations are one of the most informal methods of usability inspection in the field of human-computer interaction. There are many sets of usability design heuristics; they are not mutually exclusive and cover many of the same aspects of user interface design.

3. Quite often, usability problems that are discovered are categorized— often on a numeric scale—according to their estimated impact on user performance or acceptance. Often the heuristic evaluation is conducted in the context of use cases (typical user tasks), to provide feedback to the developers on the extent to which the interface is likely to be compatible with the intended users' needs and preferences.

2.

3. The simplicity of heuristic evaluation is beneficial at the early stages of design. This usability inspection method does not require user testing which can be

burdensome due to the need for users, a place to test them and a payment for their time. Heuristic evaluation requires only one expert, reducing the complexity and expended time for evaluation. Most heuristic evaluations can be accomplished in a matter of days. The time required varies with the size of the artifact, its complexity, the purpose of the review, the nature of the usability issues that arise in the review, and the competence of the reviewers. Using heuristic evaluation prior to user testing will reduce the number and severity of design errors discovered by users. Although heuristic evaluation can uncover many major usability issues in a short period of time, a criticism that is often leveled is that results are highly influenced by the knowledge of the expert reviewer(s). This "one-sided" review repeatedly has different results than software performance testing, each type of testing uncovering a different set of problems.

4.

ARCHITECHTURAL DESIGN:

Introduction:

The software needs the architectural design to represents the design of software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles. Each style will describe a system category that consists of :

- A set of components(eg: a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that how components can be integrated to form the system.
- Semantic models that help the designer to understand the overall properties of the system.

The use of architectural styles is to establish a structure for all the components of the system.

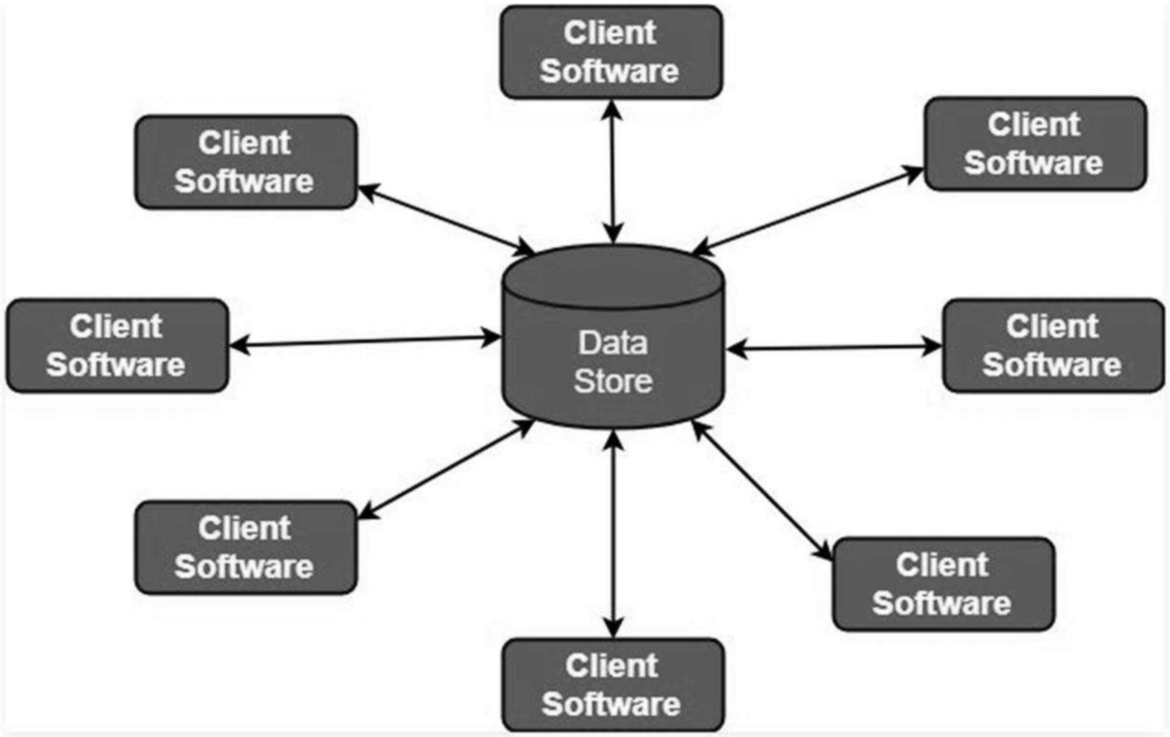
Taxonomy of Architectural styles:

1. There’s a pattern or type of architecture at the back of each artist.!
2. Differentiate a house from other styles!
3. Software also exhibits some styles!
4. Each style describes a system category that encompasses: !

(1) set of components (e.g., a database, computational modules) that perform a function required by a system, !
(2) 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. !

1. **DATA CENTRED ARCHITECTURES:**

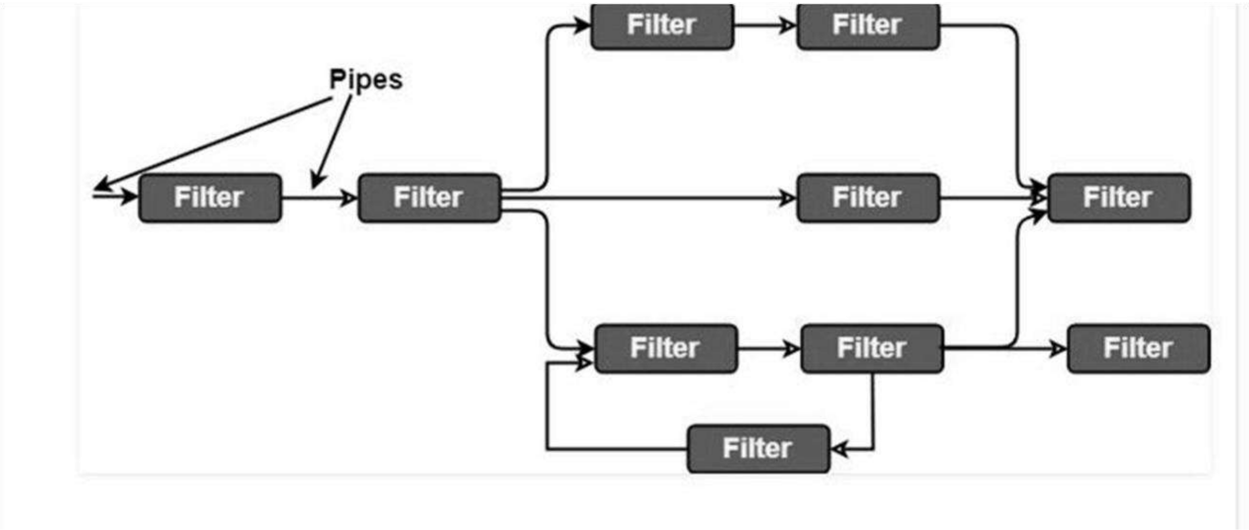
- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
- The figure illustrates a typical data centered style. The client software access a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using blackboard mechanism.



2.

DATA FLOW ARCHITECTURES:

- This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
- Pipes are used to transmit data from one component to the next.



- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

CALL AND RETURN ARCHITECTURES:

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

REMOTE PROCEDURE CALL ARCHITECTURE:
This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.

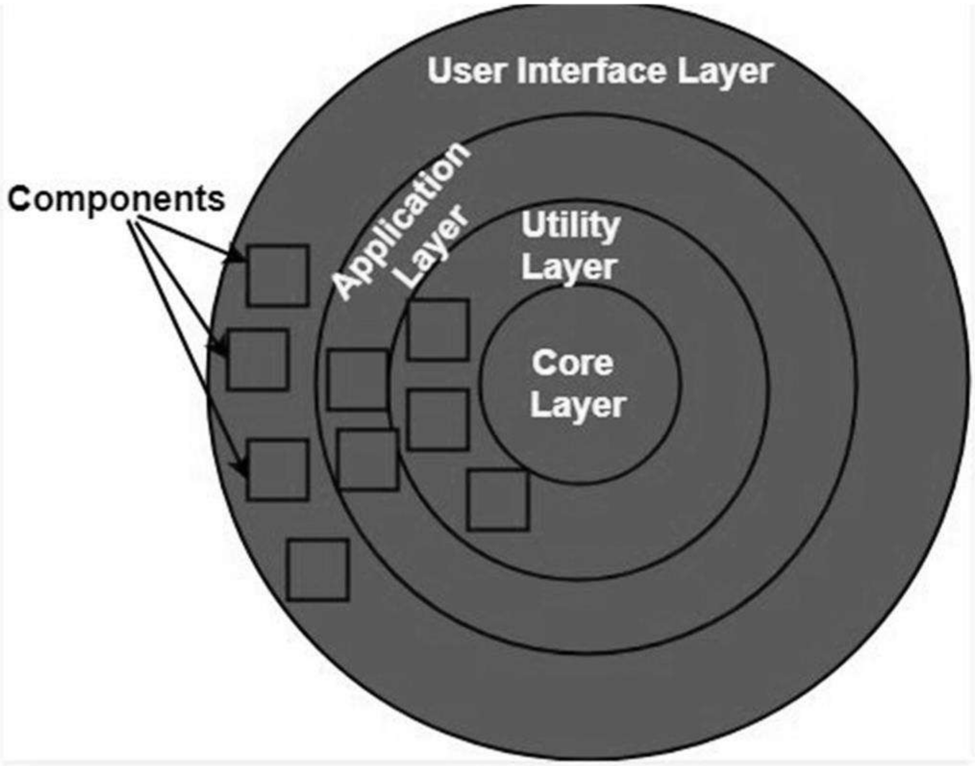
MAIN PROGRAM OR SUBPROGRAM ARCHITECTURES: The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

OBJECT ORIENTED ARCHITECTURE: The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

LAYERED ARCHITECTURE:

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)

- Intermediate layers to utility services and application software functions.



ARCHITECHTURAL DESIGN:

1.REPRESENTING THE SYSTEM IN CONTEXT:

A sytem context diagram accomplishea this requirement by representing the flow of information in to and out of thesystem ,the user interface, and relavent support processing **superordinate systems:** those systems that use the target sytems as a part of some higher level processing scheme **subordinate systems:**
those systems that are used by the target systems and processing that are necessary to complete target system functionality **peer-level system:**

Those systems that interact on a peer-to-peer basis information is either produced or consumed by the peers and the target system.

ACTORS:

Those entities that interact with the target system by producing or consuming information that is necessary for requisite processing.

DEFINING ARCH TYPES:

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architechture for the target system.

Node: represents a cohesive collection of input and output elements of the home security function.

Detector:

An abstraction that encompasses all sensing equipment that feeds information in to target system.

Indicator:

An abstraction that represents all machanisms 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.

REFINING THE ARCHITECHTURE INTO COMPONETNS:

As the software architechture is refined into components the structure of the system begins to emerge

1. External communication management
2. Control panel processing
3. Detector management
4. Alarm processing

DESCRIBING INSTANTIATIONS OF THE SYSTEM:

The architechtural design that has been modeled to this point is still relatively h igh leve the context of the system has that indicate the been represented a archetypes that indicate the important abstractions within the problem domain have been defined.

ARCHITECTURAL MAPPING USING DATA FLOW | TRANSFORM MAPPING:

Architectural Mapping Using Data Flow

A mapping technique, called structured design, is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture.

The transition from information flow to program structure is accomplished as part of a six step process:

(1) The type of information flow is established,

(2) Flow boundaries are indicated,

(3) The DFD is mapped into the program structure,

(4) Control hierarchy is defined,

(5) The resultant structure is refined using design measures.

(6) The architectural description is refined and elaborated.

Example of data flow mapping, a step-by-step “transform” mapping for a small part of the SafeHome security function.

In order to perform the mapping,

The type of information flow must be determined. It is called***transform flow and exhibits a linear quality.***

Data flows into the system along an incoming flow path where it is transformed from an external world representation into internalized form. Once it has been internalized, **it is processed at a transform center.**

Finally, it flows out of the system along an outgoing flow path that transforms the data into external world form.

Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

To illustrate this approach, we again consider the SafeHome security function.

To map these data flow diagrams into a software architecture, you would initiate the following design steps:

Step 1. Review the fundamental system model.
Step 2. Review and refine data flow diagrams for the software.
Step 3. Determine whether the DFD has transform or transaction flow characteristics.
Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.
Step 5. Perform “first-level factoring.”
Step 6. Perform “second-level factoring.”
Step 7. Refine the first-iteration architecture using design heuristic for improved software quality.

USER INTERFACE DESIGN:

User interface design (UI design) refers to the design of various types of software and hardware interfaces through which users interact with computers and other technologies. In today's diverse tech world, UI design involves a wide spectrum of engineering practices applied to different kinds of products and devices.

1. INERFACE ANALYSIS AND DESIGN:

Four different models establishes the profile of end users of the system.

1.novices
2.knowlwdgeable intermittent users
3.knowledgeable frequent users
4.the users mental model

THE PROCESS:

The analysis and design process for the user interface is iterative and can be represented using a spiral model
1.user,task,and environment analysis and modeling

2.interface design 3.interface
construction
4.interface validation:

INTERFACE ANALYSIS:

1.user analysis User
revies
Marketing input
Support input

TASK ANALYSIS AND MODELING:
Usecases
Task elaboration
Object elaboration
Work flow analysis
Hierrarchial representation

ANALYSIS AND DISPLAY CONTENT:
For modern applications dislay content can range from character based reports ,graphical displays a histogram.these data objectcan be generayed by components.

ANALYSIS OF THE WORK ENVIRONMENT:
In some applications user interface for a computer based system is placed in a user friendly location but in others lighting may be suboptimal, noise may be factor The interface designer may be constrained by factors that mitigate against ease of use.

Component Level Design:

Designing Class Based Components:
A software component is a modular building block for the computer software. Component is defined as a modular, deployable and replaceable part of the system which encloses the implementation and exposes a set of interfaces.

COMPONENTS VIEW:

The components has different views as follows:

1. An object-oriented view

- An object-oriented view is a set of collaborating classes.
- The class inside a component is completely elaborated and it consists of all the attributes and operations which are applicable to its implementation.
- To achieve object-oriented design it elaborates analysis classes and the infrastructure classes.

□

2. The traditional view

- A traditional component is known as module.
- It resides in the software and serves three important roles which are control component, a problem domain component and an infrastructure component.
- A control component coordinate is an invocation of all other problem domain components.
- A problem domain component implements a complete function which is needed by the customer.
- An infrastructure component is responsible for function which support the processing needed in the problem domain.

□

3. The Process related view

- This view highlights the building system out of existing components.
- The design patterns are selected from a catalog and used to populate the architecture.

Class-based design components

The principles for class-based design component are as follows:

Open Closed Principle (OCP):

Any module in OCP should be available for extension and modification.

The Liskov Substitution Principle (LSP):

- The subclass must be substitutable for their base class.
- This principle was suggested by Liskov.
-

Dependency Inversion Principle (DIP)

- It depends on the abstraction and not on concretion.
- Abstraction is the place where the design is extended without difficulty.
-

The Interface Segregation Principle (ISP)

Many client specific interfaces is better than the general purpose interface.

The Release Reuse Equivalency Principle (REP)

- A fragment of reuse is the fragment of release.
- The class components are designed for reuse which is an indirect contract between the developer and the user.
-

The common closure principle (CCP)

The classes change and belong together i.e the classes are packaged as part of design which should have the same address and functional area.

The Common Reuse Principle (CRP)

The classes that are not reused together should not be grouped together.

USER INTERFACE DESIGN:

- User interface design helps in succession most of the software.
- It is part of the user and computer.

- Good interface design is user friendly.

Types of user interface:

1. Command Interpreter :

Commands help the user to communicate with the computer system.

2. Graphical User Interfaces (GUI):

- It is another approach to communicate with system.
- It allows a mouse-based, window-menu-based systems as an interface.

The Golden Rules:

The golden rules are known as interface design principles.

The golden rule are as follows:

1. Place the user in control

- The interaction should be defined in such a way that the user is not forced to implement unnecessary actions.
- The technical internal details must be hidden from the casual user.
- Design for the direct interaction with objects that appear on the screen.

2. Reduce the user's memory load

- The user interface must be designed in such a way that it reduces the demands on the user's short term memory.
- Create the meaningful defaults value as an advantage for the average users in the start of application.
- There must be a reset option for obtaining the default values.

- The shortcut should be easily remembered by the users.
- The interface screen should be friendly to users.

3. Make the interface consistent

- The system must allow the user to put task into meaningful context.
- Consistency should be maintained for all the interaction.
- Do not change the past system that is created by the user expectation unless there is a good reason to do that.

TRADITIONAL COMPONENT DESIGN:

Introduction :

Traditional components are designed based on different constructs like
Sequence, Condition,
Repetition.

Sequence :implements processing steps that are essential in the specification of any algorithm.

Condition: provides the facility for selected processing based on some logical occurrence.

Repetition :allows for looping.

These three constructs are fundamental to structured programming— an important component-level design technique.

The use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability.

The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call **chunking**.

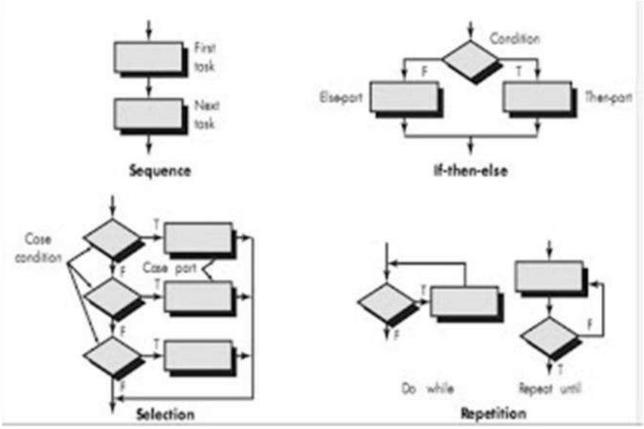
Graphical Design Notation :

A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words.

There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail. The activity diagram allows you to represent sequence, condition, and repetition— all elements of structured programming.

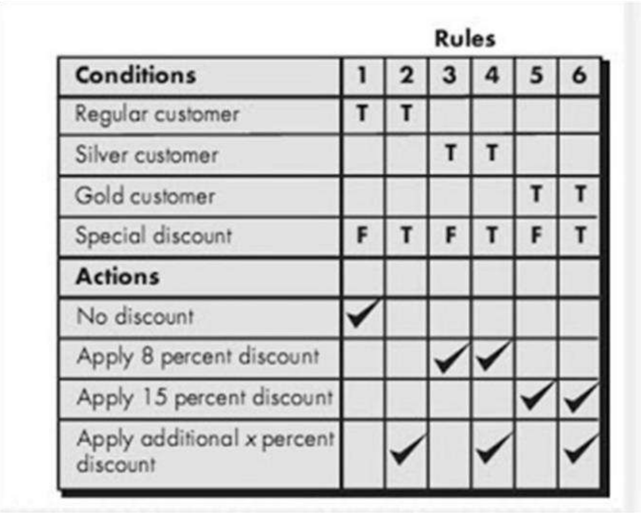
It is a descendent of an earlier pictorial design representation (still used widely) called a flowchart.

A flowchart, like an activity diagram, is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control



In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions into a tabular form.

The table is difficult to misinterpret and may even be used as a machine -readable input to a table-driven algorithm.



	Rules					
Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

Tabular Design Notation

Program Design Language

Program design language (PDL), also called structured English or pseudocode, It incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).

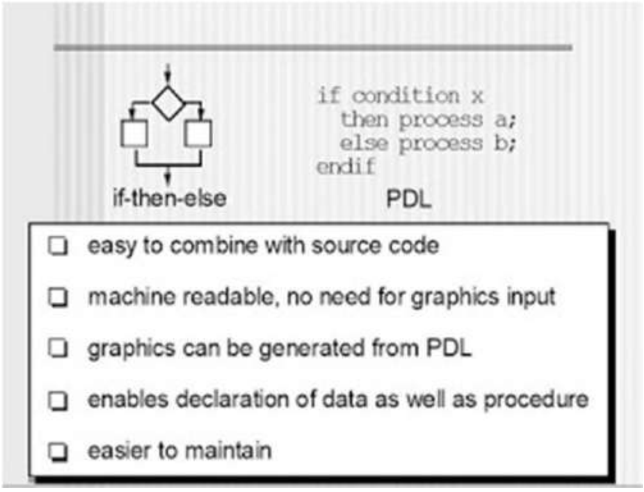
Narrative text (e.g., English) is embedded within a programming language-like syntax.

Automated tools can be used to enhance the application of PDL.

A basic PDL syntax should include constructs for

1. Component definition,
2. Interface description,
3. Data declaration,
4. Block structuring,
5. Condition constructs,
6. Repetition constructs,
7. Input-output (I/O) constructs.

It should be noted that PDL can be extended to include keywords.



```
if condition x
  then process a;
  else process b;
endif
```

- □ easy to combine with source code
- □ machine readable, no need for graphics input
- □ graphics can be generated from PDL
- □ enables declaration of data as well as procedure
- □ easier to maintain

UNIT-IV SOFTWARE TESTING FUNDAMENTALS

Internal and external views of Testing:

Inferences are said to possess internal validity if a causal relation between two variables is properly demonstrated. A causal inference may be based on a relation when three criteria are satisfied:

INTERNAL AND EXTERNAL VIEWS OF TESTING:

Inferences are said to possess internal validity if a causal relation between two variables is properly demonstrated. A causal inference may be based on a relation when three criteria are satisfied:

1. the "cause" precedes the "effect" in time (temporal precedence),

2. the "cause" and the "effect" are related (covariation), and
3. there are no plausible alternative explanations for the observed covariation (nonspuriousness)

In scientific experimental settings, researchers often manipulate a variable (the independent variable) to see what effect it has on a second variable (the dependent variable)] For example, a researcher might, for different experimental groups, manipulate the dosage of a particular drug between groups to see what effect it has on health.

In this example, the researcher wants to make a causal inference, namely, that different doses of the drug may be *held responsible* for observed changes or differences. When the researcher may confidently attribute the observed changes or differences in the dependent variable to the independent variable, and when he can rule out other explanations (or *rival hypotheses*), then his causal inference is said to be internally valid

In many cases, however, the magnitude of effects found in the dependent variable may not just depend on variations in the independent variable, the power of the instruments and statistical procedures used to measure and detect the effects, and the choice of statistical methods (see: Statistical conclusion validity).

Rather, a number of variables or circumstances uncontrolled for (or uncontrollable) may lead to additional or alternative explanations (a) for the effects found and/or (b) for the magnitude of the effects found. Internal validity, therefore, is more a matter of degree than of either-or, and that is exactly why research designs other than true experiments may also yield results with a high degree of internal validity.

In order to allow for inferences with a high degree of internal validity, precautions may be taken during the design of the scientific study. As a rule of thumb, conclusions based on correlations or associations may only allow for lesser degrees of internal validity than conclusions drawn on the basis of direct manipulation of the independent variable.

And, when viewed only from the perspective of Internal Validity, highly controlled true experimental designs (i.e. with random selection, random assignment to either the control or experimental groups, reliable instruments, reliable manipulation processes, and safeguards against confounding factors) may be the "gold standard" of scientific research. By contrast, however, the very strategies employed to control these factors may also limit the generalizability or External Validity of the findings.

External validity is the validity of generalized (causal) inferences in scientific research, usually based one xperiments as experimental validity. In other words, it is the extent to which the results of a study can be generalized to other situations and to other people For example, inferences based on comparative psychotherapy studies often employ specific samples (e.g. volunteers, highly depressed, no comorbidity). If psychotherapy is found effective for these sample patients, will it also be effective for non-volunteers or the mildly depressed or patients with concurrent other disorders?

Situation: All situational specifics (e.g. treatment conditions, time, location, lighting, noise, treatment administration, investigator, timing, scope and extent of measurement, etc. etc.) of a study potentially limit generalizability.

Pre-test effects: If cause-effect relationships can only be found when pre-tests are carried out, then this also limits the generality of the findings.
Post-test effects: If cause-effect relationships can only be found when post-tests are carried out, then this also limits the generality of the findings.

Reactivity (placebo, novelty, and Hawthorne effects): If cause-effect relationships are found they might not be generalizable to other settings or situations if the effects found only occurred as an effect of studying the situation.

Rosenthal effects: Inferences about cause-consequence relationships may not be generalizable to other investigators or researchers.

WHITE-BOX TESTING:

White-box testing, sometimes called *glass-box testing,* is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once,(2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

A reasonable question might be posed at this juncture: "Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort It is not possible to exhaustively test every program path because the number of paths is simply too large. White-box tests can be designed only after a component-level design(or source code)exists. The logical details of the program must be available.
ensuring that program requirements have been met?" Stated another way, why don'twe spend all of our energy on black-box tests? The answer lies in the nature of softwaredefects

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement function, conditions, or control that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control

and data may lead us to make design errors that are uncovered only once path testing commences.

- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

Each of these reasons provides an argument for conducting white-box tests. Blackbox testing, no matter how thorough, may miss the kinds of errors noted here. Whitebox testing is far more likely to uncover them.

BASIS PATH TESTING:

Basis path testing is a white-box testing technique first proposed by Tom McCabe [MCC76]. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

17.4.1 Flow Graph Notation:

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced.3The flow graph depicts logical control flow using the notation illustrated in Figure Each structured construct (Chapter 16) has a corresponding flow graph symbol.

To illustrate the use of a flow graph, we consider the procedural design representation in Figure. Here, a flowchart is used to depict program control structure.

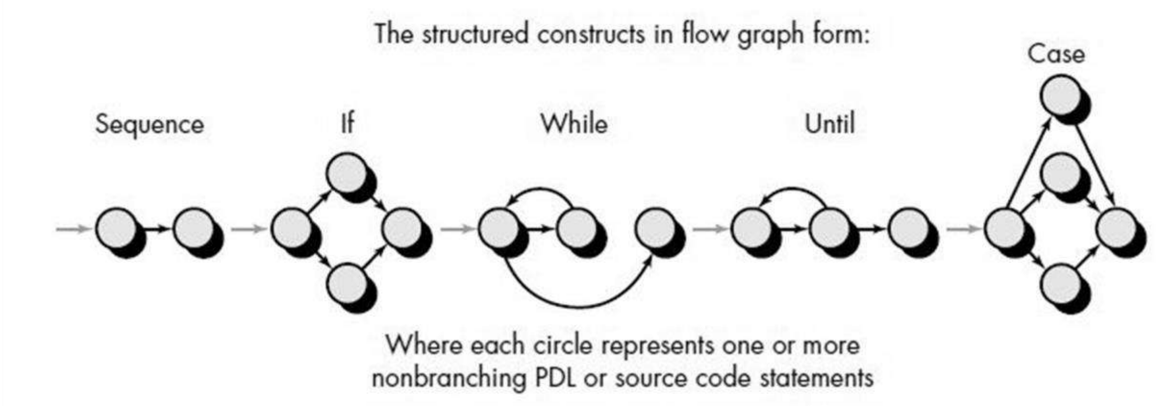
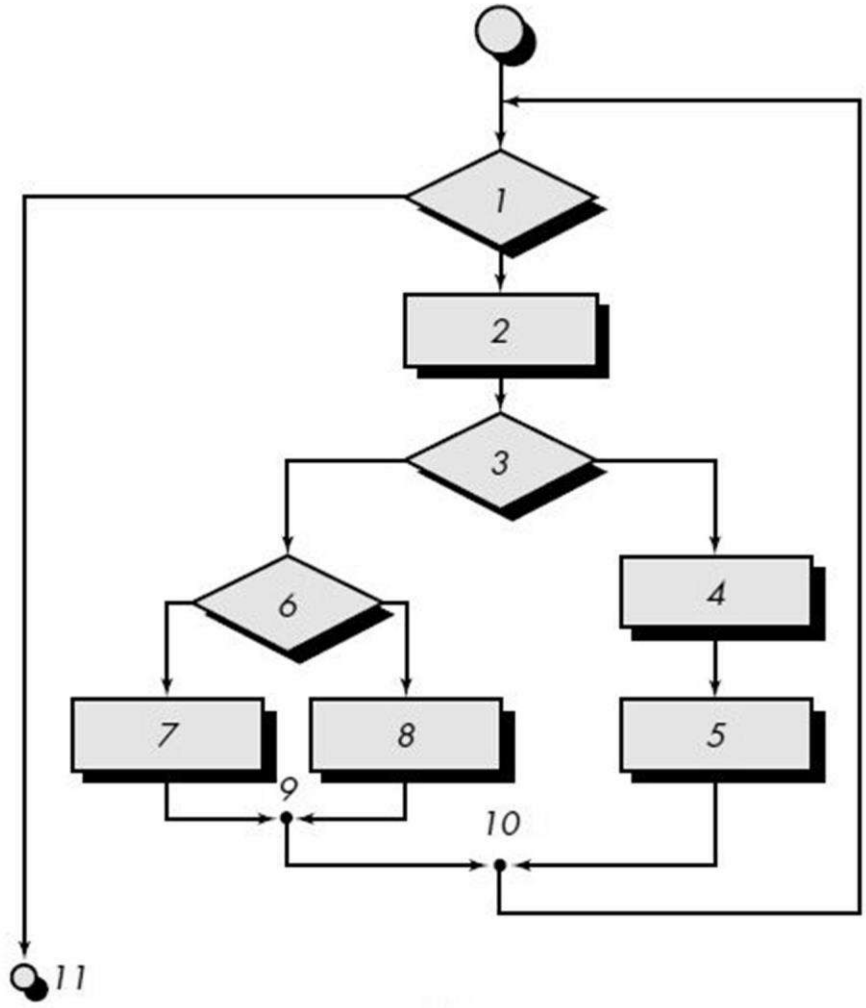

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

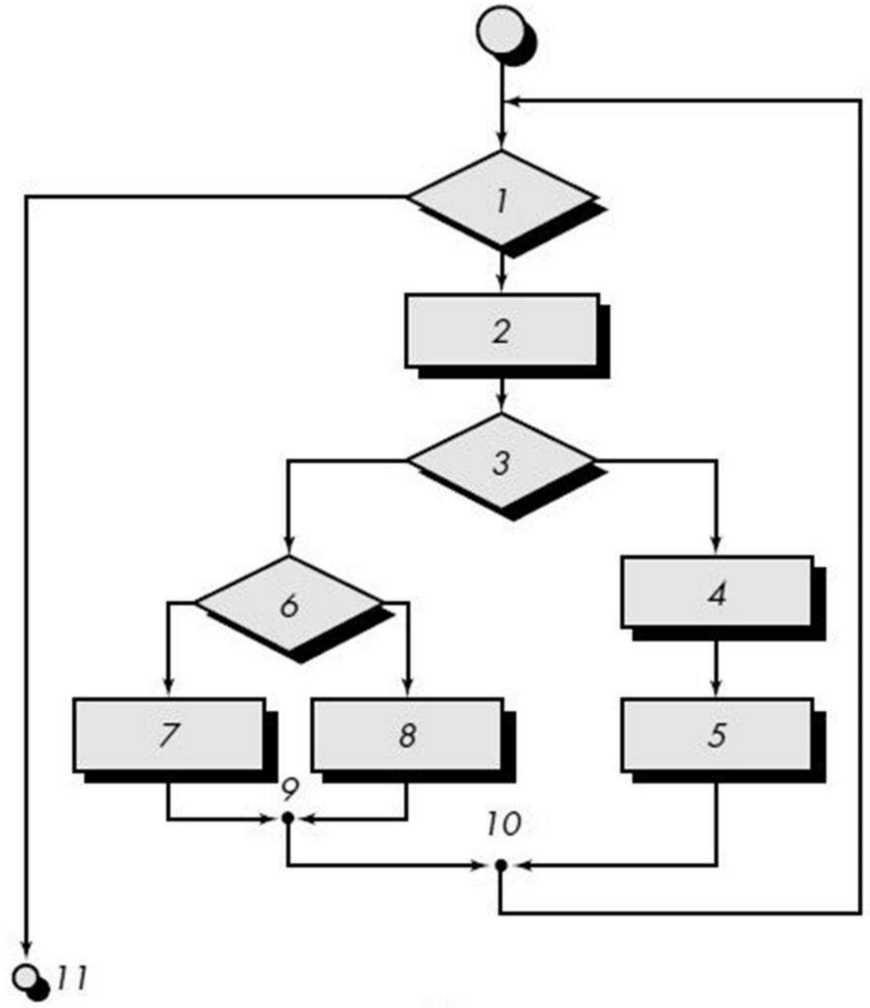
Figure 17.2B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 17.2B, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links*, represent low of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., seethe symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.4.

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 17.3, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicat enode* and is characterized by two or more edges emanating from it.
*Cyclomatic complexity :*is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow

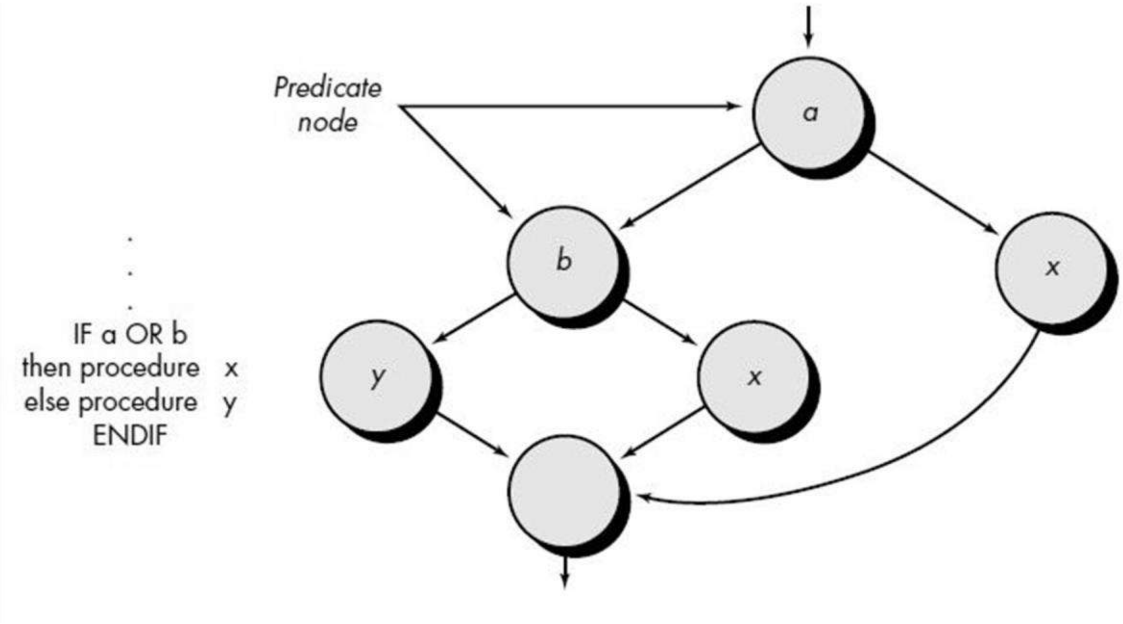


(A)



(A)

path 1: 1



graph, an independent path must move along at least one edge that has not been traversedbefore the path is defined. For example, a set of independent paths for the flow graph .illustrated in Figure is

path 2: 1-2-3-4-5-10-1-11
path 3: 1-2-3-6-8-9-10-1-11
path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11is not considered to be an independent path because it is simpl y a combination of already specified paths and does not traverse any new edges. Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in Figure 17.2B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph correspond to the cyclomatic complexity.
2. Cyclomatic complexity, *V*(*G*), for a flow graph, *G*, is defined as
V(*G*) = *E* _ *N* + 2
Predicatenode
...
IF a OR b then
procedure x
else procedure y
ENDIF
y b a x
x

How is cyclomatic complexity computed?

where *E* is the number of flow graph edges, *N* is the number of flow graph nodes.
3. Cyclomatic complexity, *V*(*G*), for a flow graph, *G*, is also defined as
V(*G*) = *P* + 1 where *P* is the number of predicate nodes contained in the
flow graph G.

Referring once more to the flow graph in Figure 17.2B, the cyclomatic complexity can be computed using each of the algorithms just noted:
1. The flow graph has four regions.
2. *V*(*G*) = 11 edges _ 9 nodes + 2 = 4.
3. *V*(*G*) = 3 predicate nodes + 1 = 4.
Therefore, the cyclomatic complexity of the flow graph in Figure 17.2B is 4.

More important, the value for *V*(*G*) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average,* depicted in PDL in Figure 17.4, will be used as an example to illustrate each step in the test case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps canbe applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules presentedin Section 16.4.1. Referring to the PDL for average in Figure , a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 17.5. **2. Determine the cyclomatic**

complexity of the resultant flow graph. The cyclomatic complexity, *V*(*G*), is determined by applying the algorithms described in Section 17.5.2. It should be noted that *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average*, compound conditions count as two) and

adding 1. Referring to Figure 17.5,

V(*G*) = 6 regions

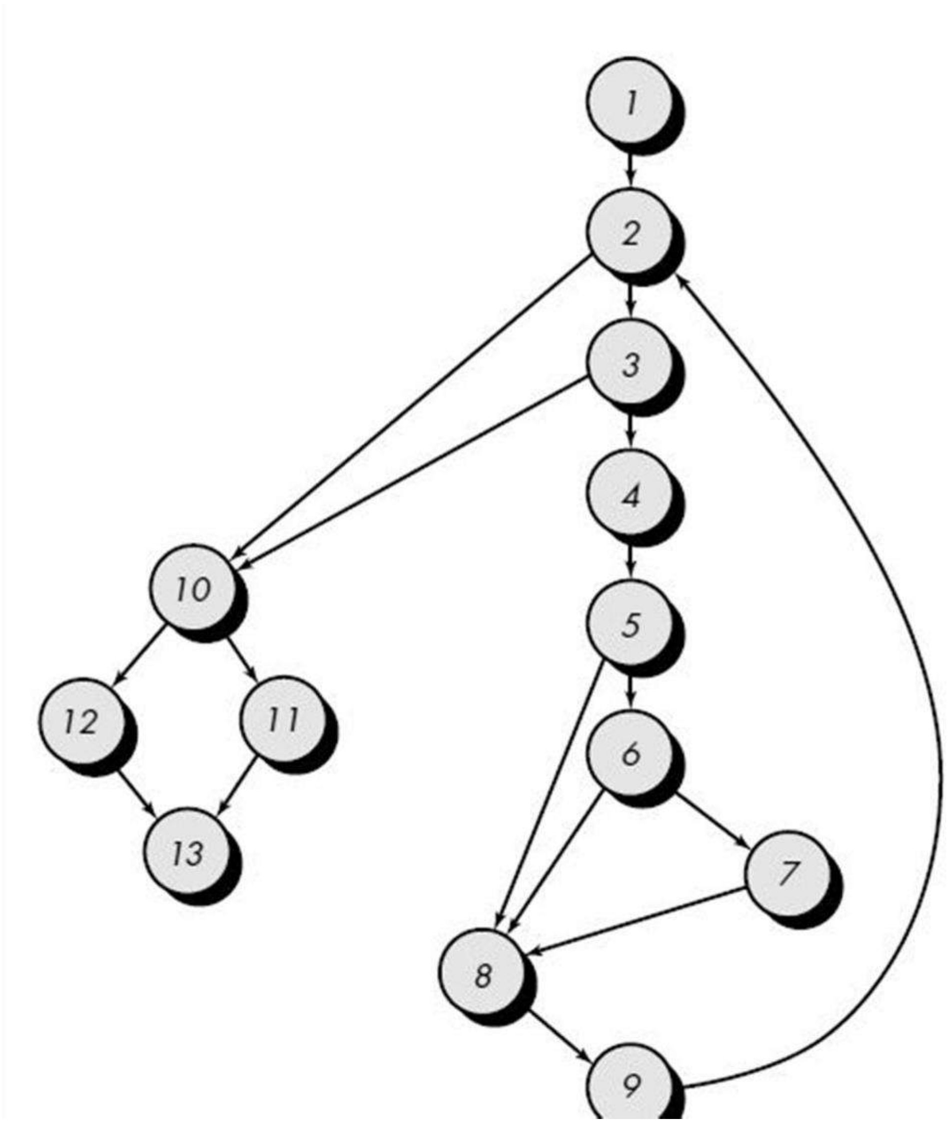
V(*G*) = 17 edges _ 13 nodes + 2 = 6

V(*G*) = 5 predicate nodes + 1 = 6

3. Determine a basis set of linearly independent paths. The value of *V*(*G*) provides the number of linearly independent paths through the program control structure. In the case of procedure *average*, we expect to specify six paths: path 1: 1-2-10-11-13 path 2: 1-2-10-12-13 path 3: 1-2-3-10-11-13 path 4: 1-2-3-4-5-8-9-2-. . . path 5: 1-2-3-4-5-6-8-9-2-. . . path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set



Path 1 test case:
value(*k*) = valid input, where *k* < *i* for 2 ≤ *i* ≤ 100
value(*i*) = _999 where 2 ≤ *i* ≤ 100
Expected results: Correct average based on *k* values and proper totals.
Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests. **Path 2 test case:** value(1) = _999
Expected results: Average = _999; other totals at initial values.

Path 3 test case:
Attempt to process 101 or more values. First 100 values should be valid.
Expected results: Same as test case 1.

Path 4 test case: value(*i*) = valid input where i < 100 value(*k*) < minimum where *k* < *i*
Expected results: Correct average based on *k* values and proper totals.

Path 5 test case: value(*i*) = valid input where *i* < 100 value(*k*) > maximum where *k* <= *i*

Expected results: Correct average based on *n* values and proper totals.

Path 6 test case: value(*i*) = valid input where *i* < 100

Expected results: Correct average based on *n* values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once. It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

Graph Matrices

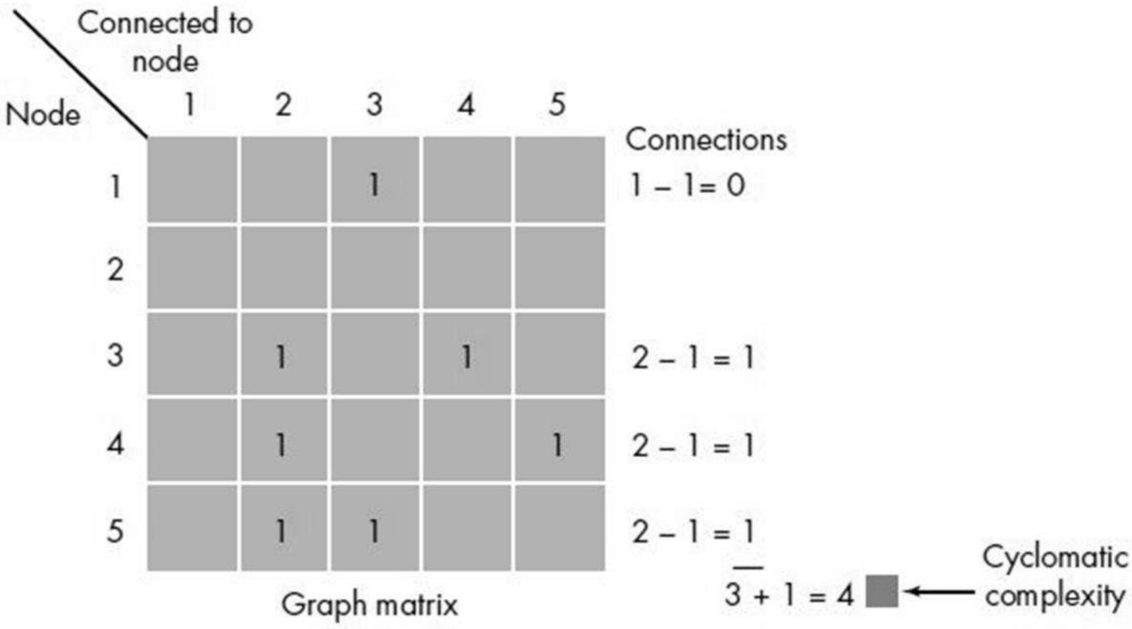
The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a *graph matrix,* can be quite useful. A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [BEI90] is shown in Figure 17.6.

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b.*

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a *link weight* to each matrix entry, the graph matri can become a powerful tool for evaluating program control structure during testing.

The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

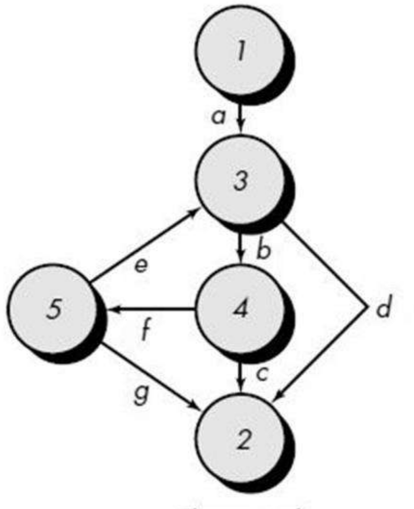
- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.



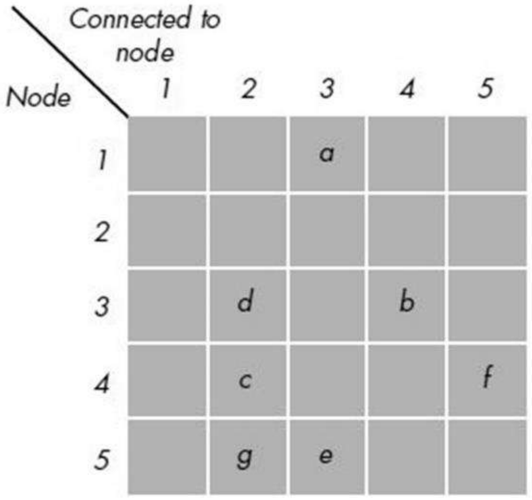
Node \ Connected to node	1	2	3	4	5	Connections
1			1			1 – 1= 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1

$\overline{3} + 1 = 4$ ■ ← Cyclomatic complexity

Graph matrix



Flow graph



Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in Figure is redrawn as shown in Figure Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity).

Represented in this form, the graph matrix is called a *connection matrix*. Referring to Figure , each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity

Beizer [BEI90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

CONTROL STRUCTURE TESTING:

The basis path testing technique described in Section 17.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form *E1* <relational-operator> *E2* where *E1* and *E2* are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT
(¬). A condition without relational expressions is referred to as a *Boolean expression.* Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies (discussed later in this section) generally have two advantages.

First, measurement of test coverage of a condition is simple. Second, the test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program *P* is effective for detecting errors in the conditions contained in *P*, it is likely that this test set is also effective for detecting other errors in *P*. In addition, if a testing strategy is effective for detecting errors in a condition, then it is likely that this strategy will also be effective for detecting errors in a program.

A number of condition testing strategies have been proposed. *Branch testing* is probably the simplest condition testing strategy. For a compound condition *C*, the true and false branches of *C* and every simple condition in *C* need to be executed at least once [MYE79].

Domain testing [WHI80] requires three or four tests to be derived for a relational expression. For a relational expression of the form *E1* <relational-operator> *E2* three tests are required to make the value of *E1* greater than, equal to, or less than that of *E2* [HOW82]. If <relational-operator> is incorrect and *E1* and *E2* are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in *E1* and *E2,* a test that makes the value of *E1* greater or less than that of *E2* should make the difference between these two values as small as possible.

For a Boolean expression with *n* variables, all of 2*n* possible tests are required (*n* > 0).

This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if *n* is small.

Error-sensitive tests for Boolean expressions can also be derived [FOS84, TAI87]. For a singular Boolean expression (a Boolean expression in which each Boolean variable occurs only once) with *n* Boolean variables (*n* > 0), we can easily generate a test set with less than 2*n* tests such that this test set guarantees the detection of multiple Boolean operator errors and is also effective for detecting other errors.

Tai [TAI89] suggests a condition testing strategy that builds on the techniques just outlined. Called *BRO* (branch and relational operator) testing, the technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables.

The BRO strategy uses condition constraints for a condition *C*. A condition constraint for *C* with *n* simple conditions is defined as (*D1, D2, . . ., Dn*), where *Di* (0 < *i* ≤ *n*) is a symbol specifying a constraint on the outcome of the *i*th simple condition in condition *C*. A condition constraint *D* for condition *C* is said to be covered by an execution of *C* if, during this execution of *C,* the outcome of each simple condition in *C* satisfies the corresponding constraint in *D.*

For a Boolean variable, *B,* we specify a constraint on the outcome of *B* that states that *B* must be either true (t) or false (f). Similarly, for a relational expression, the symbols

>, =, < are used to specify constraints on the outcome of the expression

As an example, consider the condition
C1: *B1* & *B2*
where *B1* and *B2* are Boolean variables. The condition constraint for *C1* is of the form
(*D1*, *D2*), where each of *D1* and *D2* is t or f. The value (t, f) is a condition constraint for *C1* and is covered by the test that makes the value of *B1* to be true and the value of *B2* to be false. The BRO testing strategy requires that the constraint set {(t, t), (f, t),
(t, f)} be covered by the executions of *C1*. If *C1* is incorrect due to one or more Boolean operator errors, at least one of the constraint set will force *C1* to fail.
As a second example, a condition of the form *C2*: *B1* & (*E3* = *E4*) where *B1* is a Boolean expression and *E3* and *E4* are arithmetic expressions. A condition constraint for *C2* is of the form (*D1*, *D2*), where each of *D1* is t or f and *D2* is
>, =, <. Since *C2* is the same as *C1* except that the second simple condition in *C2* is a
relational expression, we can construct a constraint set for *C2* by modifying the constraint set {(t, t), (f, t), (t, f)} defined for *C1*. Note that t for (*E3* = *E4*) implies = and that
f for (*E3* = *E4*) implies either < or >. By replacing (t, t) and (f, t) with (t, =) and (f, =), respectively, and by replacing (t, f) with (t, <) and (t, >), the resulting constraint set for *C2* is {(t, =), (f, =), (t, <), (t, >)}. Coverage of the preceding constraint set will guarantee detection of Boolean and relational operator errors in *C2*.
As a third example, we consider a condition of the form
C3: (*E1* > *E2*) & (*E3* = *E4*) where *E1*, *E2*, *E3*, and *E4* are arithmetic expressions. A condition constraint for *C3* is of the form (*D1*, *D2*), where each of *D1* and *D2* is >, =, <. Since *C3* is the same as *C2* except that the first simple condition in *C3* is a relational expression, we can construct a constraint set for *C3* by modifying the constraint set for *C2*, obtaining
{(>, =), (=, =), (<, =), (>, >), (>, <)}
Coverage of this constraint set will guarantee detection of relational operator errors in *C3*.

BLACK-BOX TESTING:

Black-box testing, also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing (see Chapter 18). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation? By applying black-box techniques, we derive a set of test cases that satisfy the following criteria [MYE79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing and (2) test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing (see Chapter 18). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

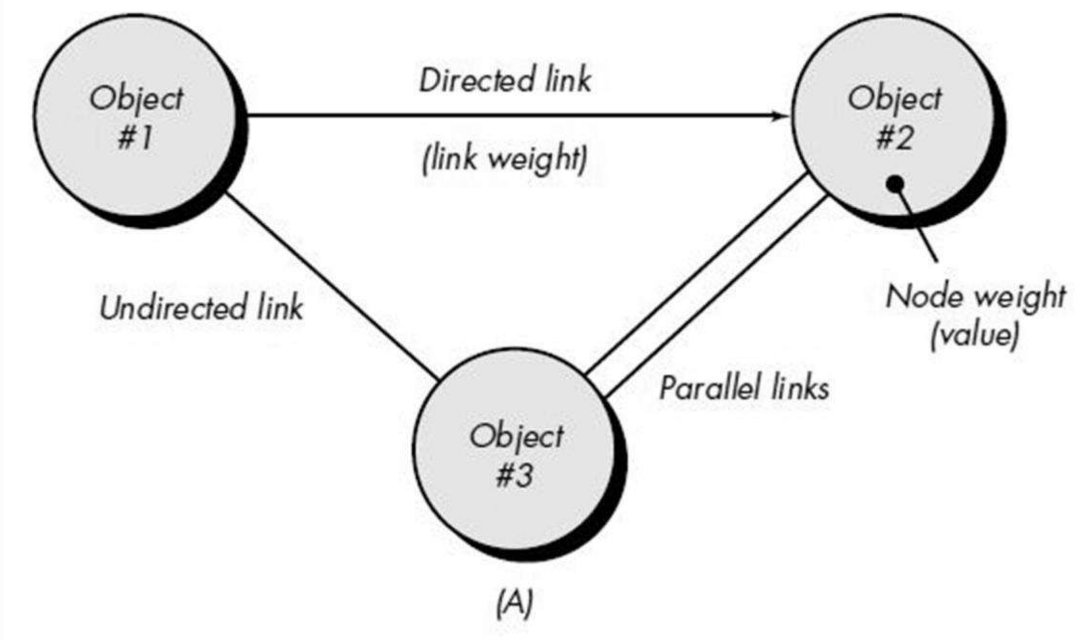
- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation? By applying black-box techniques, we derive a set of test cases that satisfy the

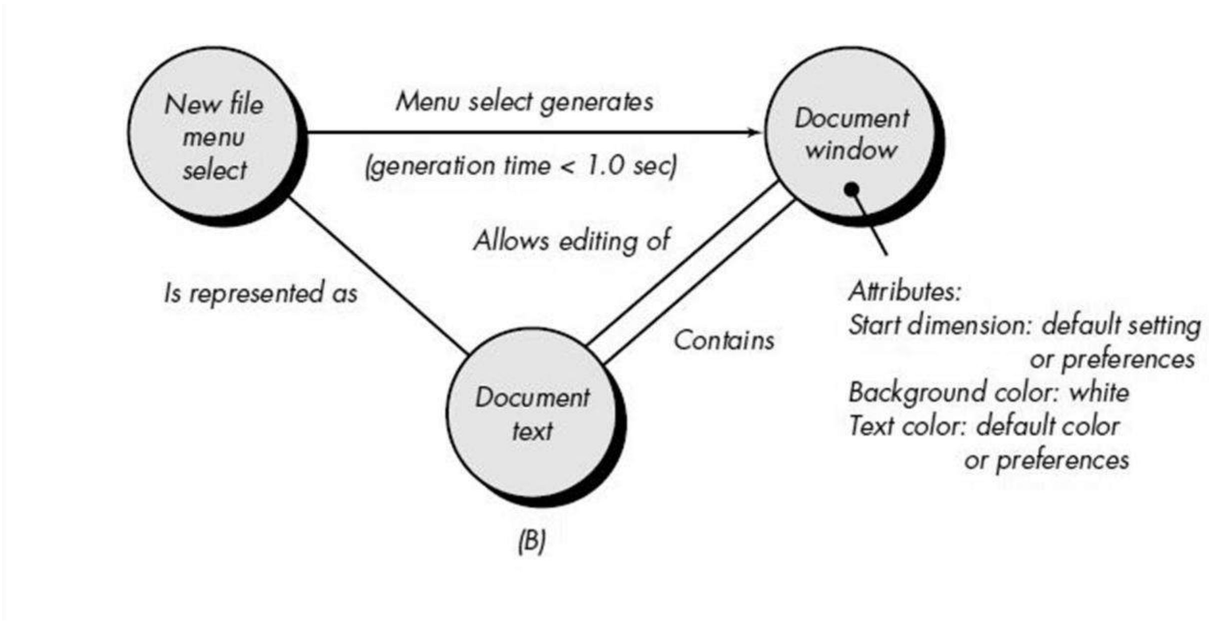
following criteria [MYE79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing and (2) test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Testing Methods

The first step in black-box testing is to understand the objects6 that are modeled in software and the relationships that connect these objects. Once this has been accomplished,the next step is to define a series of tests that verify "all objects have the expected relationship to one another [BEI95]." Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, the software engineer begins by creating a *graph*—a collection of *nodes* that represent objects; *links* that represent the relationships between objects; *node weights* that describe the properties of a node (e.g., a specific data value or state behavior); and *link weights* that describe some characteristic of a link.7





he symbolic representation of a graph is shown in Figure 17.9A. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction.

A *bidirectional link,* also called a *symmetric link*, implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 17.9B) where

Object #1 = **new file menu select**

Object #2 = **document window**

Object #3 = **document text**

Referring to the figure, a menu select on **new file** generates a **document window.** The node weight of **document window** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second Contains

(A) Graph
notation (B)
Simple
example

PART THREE CONVENTIONAL METHODS FOR SOFTWARE ENGINEERING

symmetric relationship between the **new file menu select** and **document text,** and parallel links indicate relationships between **document window** and **document text.** In reality, a far more detailed graph would have to be generated as a precursor to test case design. The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer [BEI95] describes a number of behavioral testing methods that can make use of graphs

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., **flight,information,input** is followed by *validation/availability.processing*). The data flow diagram (Chapter 12) can be used to assist in creating graphs of this type.

Finite state modeling. The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **order-information** is verified during *inventory-availability look-up* and is followed by **customer-billing-information input**). The state transition diagram can be used to assist in creating graphs of this type.

Data flow modeling. The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node **FICA.tax.withheld (FTW)** is computed from **gross.wages (GW)** using the relationship, FTW = 0.62 _ GW.

Timing modeling. The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. The interested reader should see [BEI95] for a comprehensive discussion.

It is worthwhile, however, to provide a generic outline of the graph-based testing approach.

Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model (Chapter 12) can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.

Once nodes have been identified, links and link weights should be established. In general, links should be named, although links that represent control flow between program objects need not be named.

SOFTWARE TESTING TECHNIQUES

In many cases, the graph model may have loops (i.e., a path through the graph in which one or more nodes is encountered more than one time). Loop testing can also be applied at the behavioral (black-box) level. The graph will assist in identifying those loops that need to be tested.

Each relationship is studied separately so that test cases can be derived. The *transitivity* of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, **X, Y,** and **Z.** Consider the following relationships:

X is required to compute **Y**

Y is required to compute **Z**

Therefore, a transitive relationship has been established between **X** and **Z**:

X is required to compute **Z**

Based on this transitive relationship, tests to find errors in the calculation of **Z** must consider a variety of values for both **X** and **Y.**

The *symmetry* of a relationship (graph link) is also an important guide to the design of test cases. If a link is indeed bidirectional (symmetric), it is important to test this feature. The UNDO feature [BEI95] in many personal computer applications implements limited symmetry. That is, UNDO allows an action to be negated after it has been completed. This should be thoroughly tested and all exceptions (i.e., places where UNDO cannot be used) should be noted. Finally, every node in the graph should have a relationship that leads back to itself; in essence, a “no action” or “null action” loop. These *reflexive* relationships should also be tested.

As test case design begins, the first objective is to achieve *node coverage*. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, *link coverage* is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present.

A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked Equivalence Partitioning.

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [BEI95]. An *equivalence class* represents a set of valid or invalid states for input conditions.

Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined.

4. If an input condition is *Boolean,* one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application.

The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking applicationaccepts data in the form area code—blank or three-digit number prefix—three-digit number not beginning with 0 or 1 suffix—four-digit number password—six digit alphanumeric string commands—check, deposit, bill pay, and the like The input conditions associated with each data element for the banking application can be specified as area code: Input condition, *Boolean*—the area code may or may not be present. Input condition, *range*—values defined between 200 and 999, with specific exceptions. prefix: Input condition, *range*—specified value >200 Input condition, value—four-digit length password: Input condition, *Boolean*—a password may or may not be present. Input condition, *value*—sixcharacter string. command: Input condition, *set*—containing commands noted previously. Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected that the largest number of attributes of an equivalence class are exercised at once.

Boundary Value Analysis

For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [MYE79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b.*

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

Comparison Testing

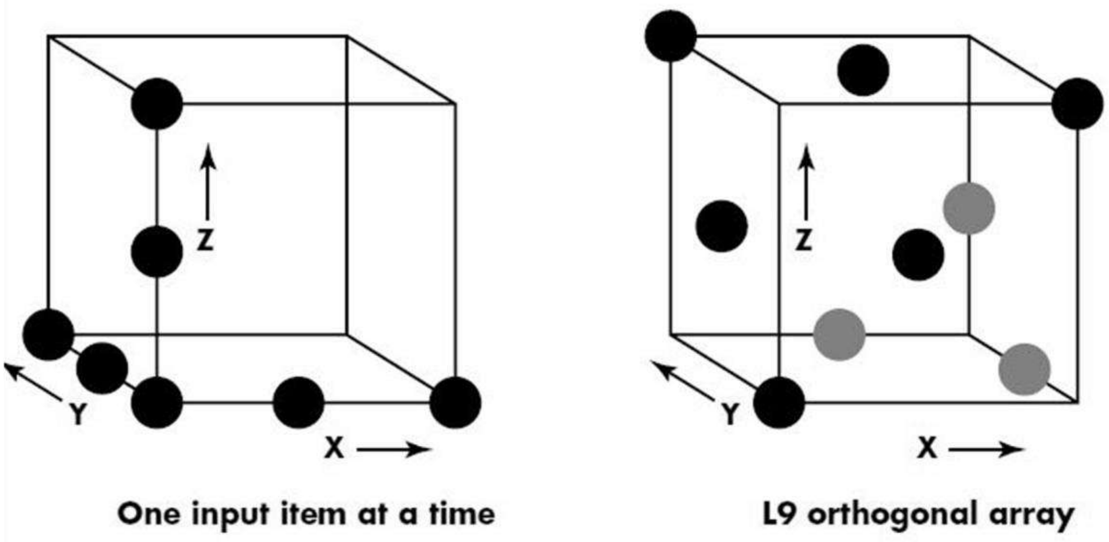
There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical. In such applications redundant hardware and software are often used to minimize the possibility of error. When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification. In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

Using lessons learned from redundant systems, researchers (e.g., [BRI87]) have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system. These independent versions form the basis of a black-box testing technique called *comparison testing* or *back-to-back testing* [KNI89].

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software. If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference. In most cases, the comparison of outputs can be performed by an automated tool.

Comparison testing is not foolproof. If the specification from which all versions have been developed is in error, all versions will likely reflect the error. In addition, if each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.

Orthogonal Array Testing There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three inpu parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test processing of the input domain. However, as thenumber of input values grows and the number of discrete values for each data item increases, exhaustive testing become impractical or impossible. *Orthogonal array testing* can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding errors associated with *region faults*—an error category associated with faulty logic within a software component.To illustrate the difference between orthogonal array testing and more conventional “one input item at a time” approaches, consider a system that has three input items, *X*, *Y*, and *Z*. Each of these input items has three discrete values associated withit. There are 33 = 27 possible test cases. Phadke [PHA97] suggests a geometric view of the possible test cases associated with *X, Y,* and *Z* illustrated in Figure. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure



domain," as illustrated in the right-hand cube in Figure 17.10. Test coverage across the input domain is more complete.

To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now

P1 = 2, send it one hour later

P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other *send* functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1,1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke [PHA97] assesses these test cases in the following manner:

Such test cases are useful only when one is certain that these test parameters do not interact.

They can detect logic faults where a single parameter value makes the software malfunction.

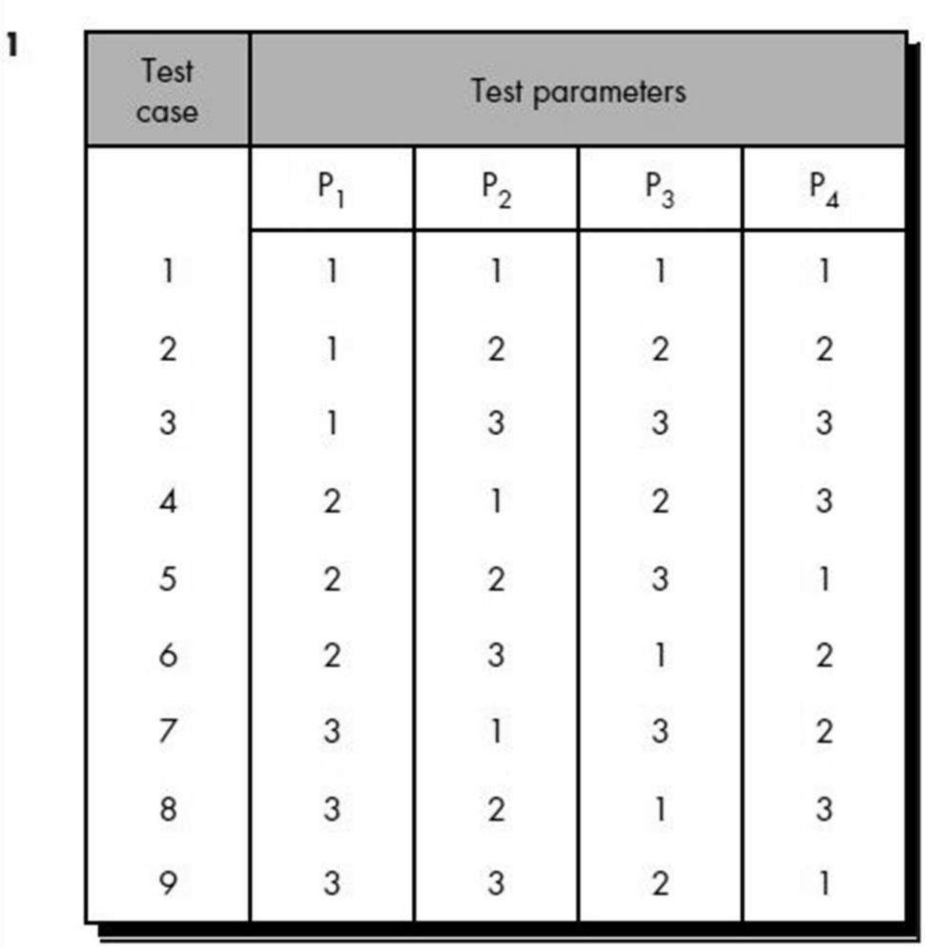
These faults are called *single mode faults*. This method cannot detect logic faults that cause malfunction when two or more parameters simultaneously take certain values;

that is, it cannot detect any interactions. Thus its ability to detect faults is limited. Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is 34 = 81, large, but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high. Phadke [PHA97] assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault.* Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.



1

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred

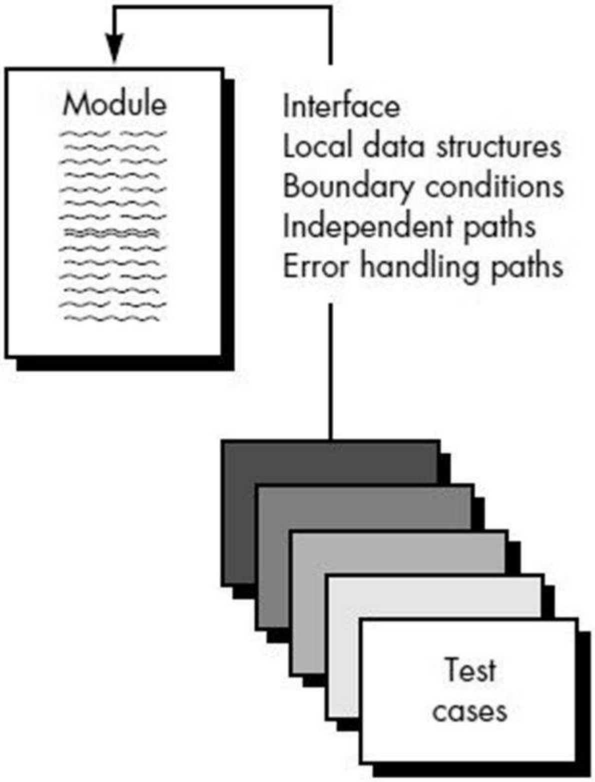
UNIT TESTING:

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

Unit Test Considerations:

The tests that occur as part of unit tests are illustrated schematically in Figure 18.4. The module interface is tested to ensure that information properly flows into and outof the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.

Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.



Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Among the more common errors in computation are (1) misunderstood or incorrect arithmetic precedence, (2) mixed mode operations, (3) incorrect initialization, (4) precision inaccuracy, (5) incorrect symbolic representation of an expression. Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as (1) comparison of different data types, (2) incorrect logical operators or precedence, (3) expectation of equality when precision error makes equality unlikely, (4) incorrect comparison of variables, (5) improper or nonexistent loop termination, (6) failure to exit when divergent iteration is encountered, and (7) improperly modified loop variables.

Good design dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does occur. Yourdon [YOU75] calls this approach *antibugging*. Unfortunately, there is a tendency to incorporate error handling into software and then never test it. A true story may serve to illustrate:

A major interactive design system was developed under contract. In one transaction processing module, a practical joker placed the following error handling message after a series of conditional tests that invoked various control flow branches: ERROR! THERE IS NO WAY YOU CAN GET HERE. This

"error message" was uncovered by a customer during user training! Among the potential errors that should be tested when error handling is evaluated are **1.** Error description is unintelligible.

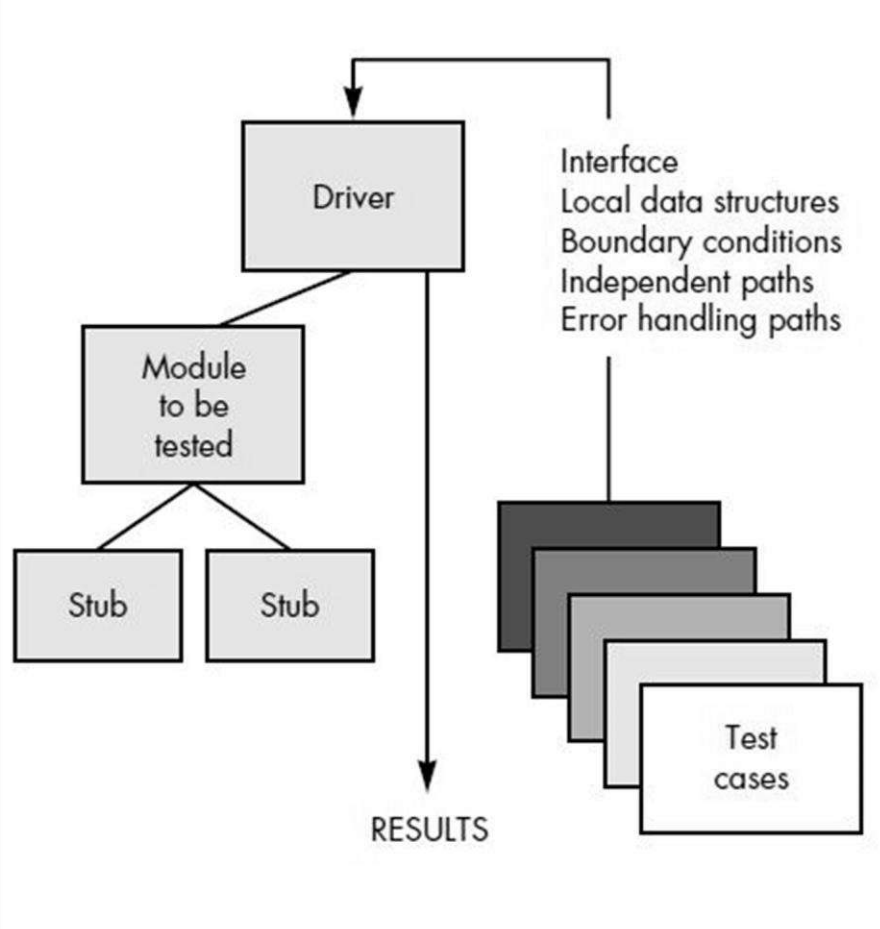
2. Error noted does not correspond to error encountered.

3. Error condition causes system intervention prior to error handling.

4. Exception-condition processing is incorrect.

5. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the last (and probably most important) task of the unit test step. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with



i passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure

In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used). Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

INTEGRATION TESTING:

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; sub functions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on .Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.

The objective is to take unit tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop. Incremental integration is the antithesis of the big bang approach. The program

is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the sections that follow, a number of different incremental integration strategies are discussed.

Top-down Integration:

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate(and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to Figure 18.6, *depth-first integration* would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.

For example, selecting the lefth and path, components M1, M2 , M5 would be integrated first. Next, M8 or (if neces-sary for proper functioning of M2) M6 would be integrated. Then, the central and right hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

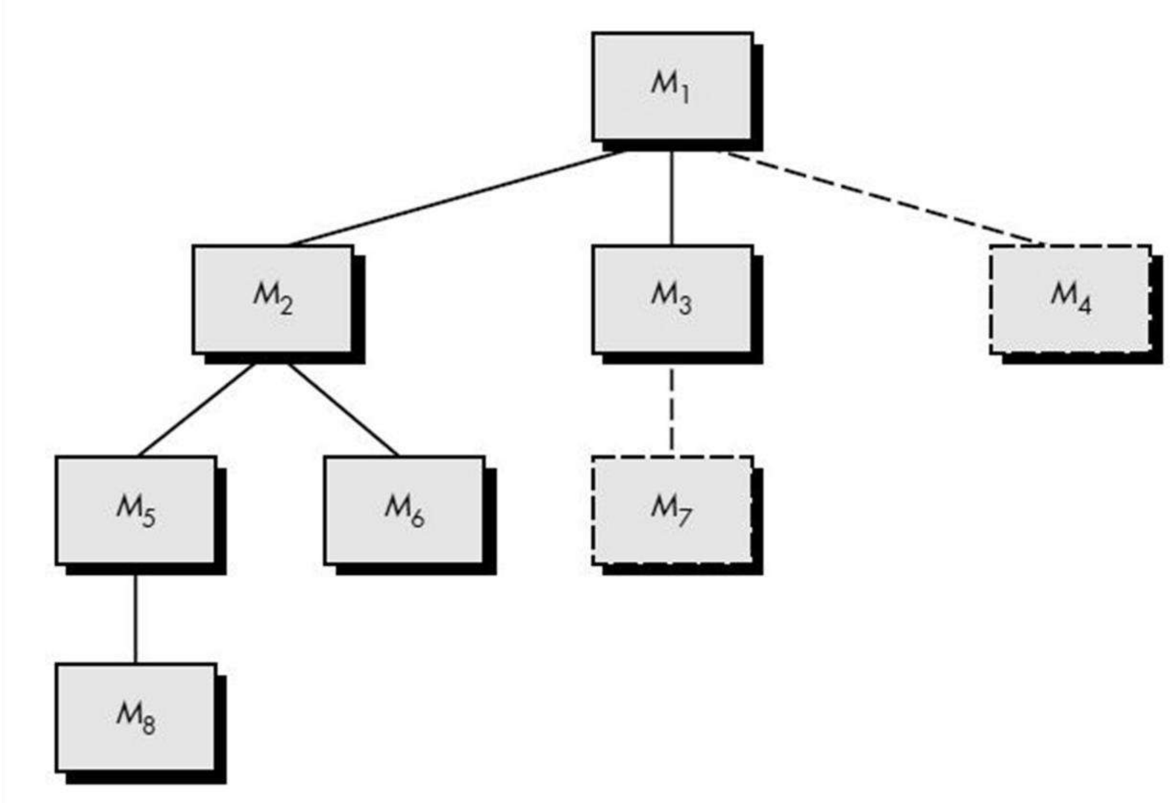
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (Section 18.4.3) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built. The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure (Chapter 14) in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path.



The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing (Section 18.4.3) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure (Chapter 14) in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. The tester is left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) causes us to loose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex.

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

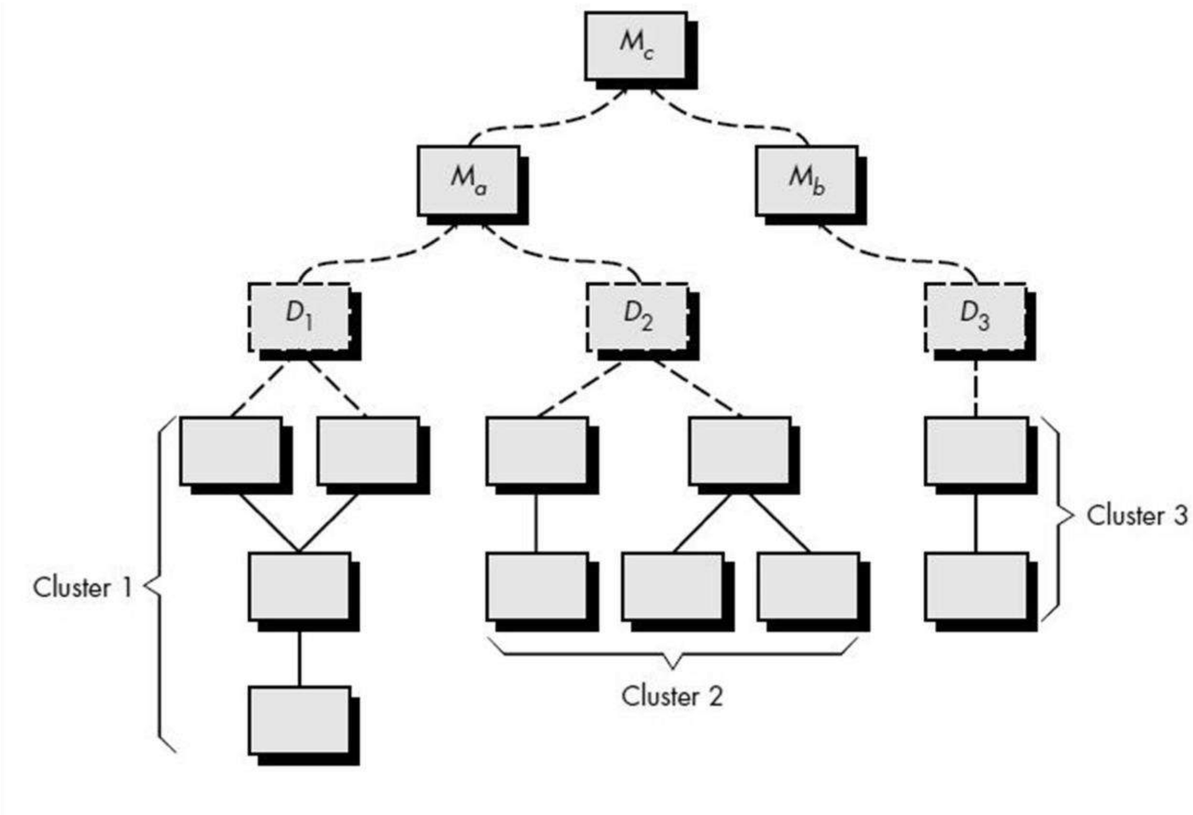
A bottom-up integration strategy may be implemented with the following steps: **1.** Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in Figure . Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both



the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.* Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Smoke Testing

Smoke testing is an integration testing approach that is commonly used when "shrink wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a requent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up. The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [MCO96] describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly. Smoke testing might be characterized as a rolling integration strategy. The software is rebuilt (with new components added) and exercised every day.

SOFTWARE TESTING STRATEGIES

Smoke testing provides a number of benefits when it is applied on complex, timecritical software engineering projects:

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered. • *The quality of the end-product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.
- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

VALIDATION TESTING

At the culmination of integration testing, software is completely assembled as a package,

interfacing errors have been uncovered and corrected, and a final series of software tests—*validation testing*—may begin. Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the *Software Requirements Specification*— a document (Chapter 11) that describes all user-visible attributes of the software. The specification contains a section called *Validation Criteria*. Information contained in that section forms the basis for a validation testing approach.

Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability). After each validation test case has been conducted, one of two possible conditions exist: (1) The function or performance characteristics conform to specification and are accepted or (2) a deviation from specification is uncovered and a *deficiency list* is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

An important element of the validation process is a *configuration review*. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle.

ALPHA AND BETA TESTING

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field. When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements. Conducted by the enduser rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time. If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builderuse a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

The *alpha test* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

SYSTEM TESTING

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger-pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests [BEI84] that are worthwhile for software-based systems.

Recovery Testing

Many computer based systems must recover from faults and resume processing with in a prespecified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performedby the system itself), reinitialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptablelimits.

Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain. *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

To quote Beizer [BEI84]: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

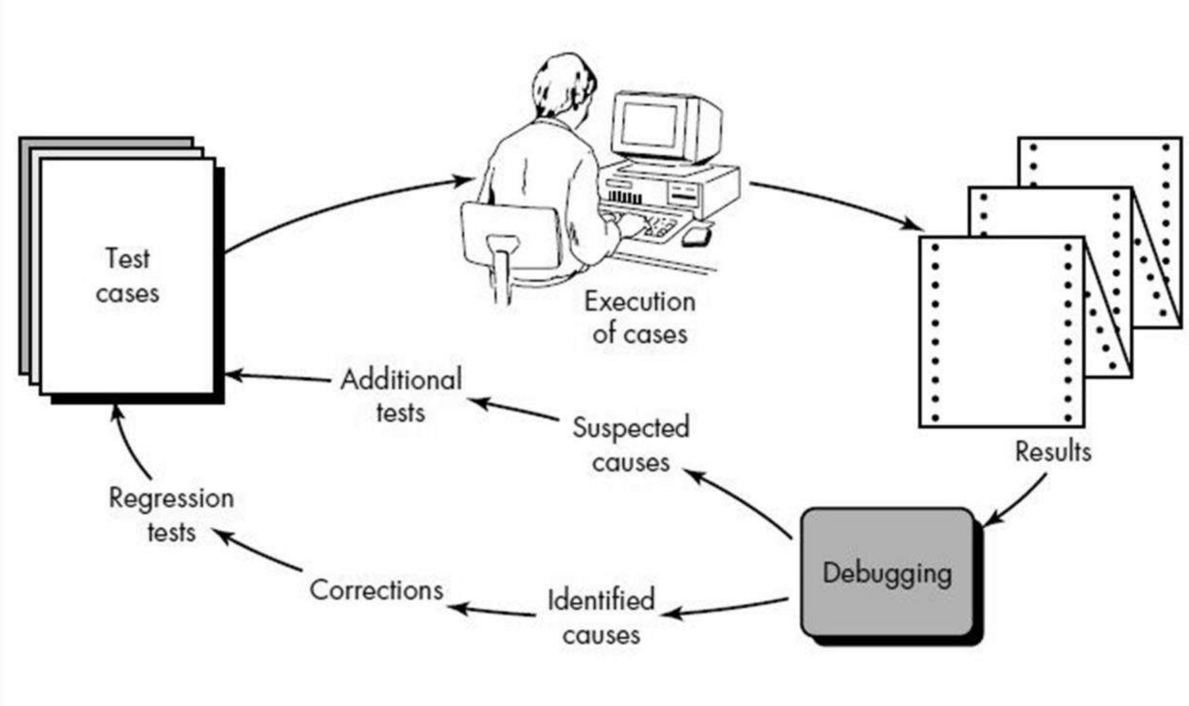
During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?" *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

Performance Testing:



For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. *Performance testing* is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

THE ART OF DEBUGGING:

Software testing is a process that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

Although debugging can and should be an orderly process, it is still very much an art.
A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

The Debugging Process:

Debugging is not testing but always occurs as a consequence of testing.4 Referring to Figure , the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.

500 PART THREE CONVENTIONAL METHODS FOR SOFTWARE ENGINEERING
of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.
The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.
Why is debugging so difficult? In all likelihood, human psychology (see the next section) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures (Chapter 13) exacerbate this situation.

2. The symptom may disappear (temporarily) when another error is corrected. **3.** The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors [CHE90].

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [VAN89] suggests three simple questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

1. Is the cause of the bug reproduced in another part of the program? In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.

2. What "next bug" might be introduced by the fix I'm about to make? Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.

3. What could we have done to prevent this bug in the first place? This question is the first step toward establishing a statistical software quality assurance approach (Chapter 8). If we correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs

CODING AND REFACTORING:

Code refactoring is one of the key terms in software development and today I would like to talk about code refactoring techniques that might increase your efficiency!

But first, let's agree on what is code refactoring! Basically, code refactoring is the process of changing a program's source code without modifying its external functional behavior, in order to improve some of the nonfunctional attributes of the software. In other words, code refactoring is the process of clarifying and simplifying the design of existing code, without changing its behavior. Nowadays, agile software development is literally a must and agile teams are maintaining and extending their code a lot from iteration to iteration, and without continuous refactoring, this is hard to do.

This is because un-refactored code tends to rot: unhealthy dependencies between classes or packages, bad allocation of class responsibilities, way too many responsibilities per method or class, duplicated code, and many other varieties of confusion and clutter. So, the advantages include improved code readability and reduced complexity; these can improve source-code maintainability and create a more expressive internal architecture.

Another reason why you should read it is that it is written by legends of our time, by people who actually tried it first and developed the concept! There are other interesting books about this topic and you can find them here, but this one is a high priority one.

Some tips for doing code refactoring techniques right

- Code refactoring should be done as a series of small changes, each of which makes the existing code slightly better while still leaving the program in working order. Don’t mix a whole bunch of refactorings into one big change.

- When you do refactoring, you should definitely do it using TDD and CI1. Without being able to run those tests after each little step in a refactoring, you create a risk of introducing bugs.
- The code should become cleaner.

- New functionality should not be created during refactoring. Do not mix refactoring and direct development of new features. Try to separate these processes at least within the confines of individual commits.

Benefits of code refactoring:

See the whole picture: If you have one main method that handles all of the functionality, it’s most likely way too long and incredibly complex. But if it’s broken down into parts, it’s easy to see what is really being done.

Make it readable for your team

Make it easy to understand for your peers, don’t write it for yourself, think on the long-term.

Maintainability

Integration of updates and upgrades is a continuous process that is unavoidable and should be welcomed. When the codebase is unorganized and built on weak foundation, developers are often hesitant to make changes. But with code refactoring, organized code, the product will be built on a clean foundation and will be ready for future updates.

Efficiency

Code refactoring may be considered as investment, but it gets good results. You reduce the effort required for future changes to the code, either by you or other developers, thus improving efficiency.

Reduce complexity

Make it easier for you and your team to work on the project.

List of main code refactoring techniques

There are many code refactoring techniques and I do not want to cover them all, as this post would end up becoming a book in itself. So, I decided to pick the ones we feel are the most common and useful.

Red-green refactoring:

Lets start by briefly talking about the very popular red-green code refactoring technique. Red Green Refactor is the Agile engineering pattern which underpins Test Driven Development. Characterized by a “test-first” approach to design and implementation. This lays the foundation for all forms of refactoring. You incorporate refactoring into the test driven development cycle by starting with a failing “red” test, writing the simplest code possible to get the test to pass “green” and finally work on improving and enhancing your code while keeping the test “green”. This approach is about how one can seamlessly integrate refactoring into your overall development process and work towards keeping code clean. There are two distinct parts to this: writing code that adds a new function to your system, and improving the code that does this function. The important thing is to remember to not do both at the same time during the workflow.

Preparatory refactoring :

As a developer, there are things you can do to your codebase to make the building of your next feature a little more painless. Martin Fowler calls this preparatory refactoring. This again can be executed using the red-green technique described above. Preparatory refactoring can also involve paying down technical debt that was accumulated during the earlier phases of feature development. Even though the end-users may not see eye to eye with the engineering team on such efforts, the developers almost always appreciate the value of a good refactoring exercise.

Branching by abstraction refactoring:

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa. For example: Pull up field, pull up method, pull up constructor body, push down field, push down method, extract subclass, extract superclass, extract interface, collapse hierarchy, form template method, replace inheritance with delegation, replace delegation with Inheritance, etc.

There are two types of refactoring efforts that is classified based on scope and complexity. Branching by abstraction is a technique that some of the teams use to take on large scale refactoring. The basic idea is to build an abstraction layer that wraps the part of the system that is to be refactored and the counterpart that is eventually going to replace it. For example: encapsulate field – force code to access the field with getter and setter methods, generalize type – create more general types to allow for more code sharing, replace type-checking code with state, replace conditional with polymorphism, etc.

Composing methods refactoring:

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand and even harder to change. The code refactoring techniques in this group streamline methods, remove code duplication. Examples can be: extract method, inline method, extract variable, inline Temp, replace Temp with Query, split temporary variable, remove assignments to parameters, etc.

Moving features between objects refactoring:

These code refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access. For example: move method, move field, extract class, inline class, hide delegate, remove middle man, introduce foreign method, introduce local extension, etc.

Simplifying conditional expressions refactoring :

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well. For example: consolidate conditional expression, consolidate duplicate conditional fragments, decompose conditional, replace conditional with polymorphism, remove control flag, replace nested conditional with guard clauses,etc.

Simplifying method calls refactoring:

These techniques make method calls simpler and easier to understand. This simplifies the interfaces for interaction between classes. For example: add parameter, remove parameter, rename method, separate query from modifier, parameterize Method, introduce parameter object, preserve whole object, remove setting method, replace parameter with explicit methods, replace parameter with method call, etc.

Breaking code apart into more logical pieces refactoring:

Componentization breaks code down into reusable semantic units that present clear, well-defined, simple-to-use interfaces. For example: extract class moves part of the code from an existing class into a new class, extract method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.

User Interface Refactoring:

A simple change to the UI retains its semantics, for example: align entry field, apply common button size, apply font, indicate format, reword in active voice and increase color contrast, etc.