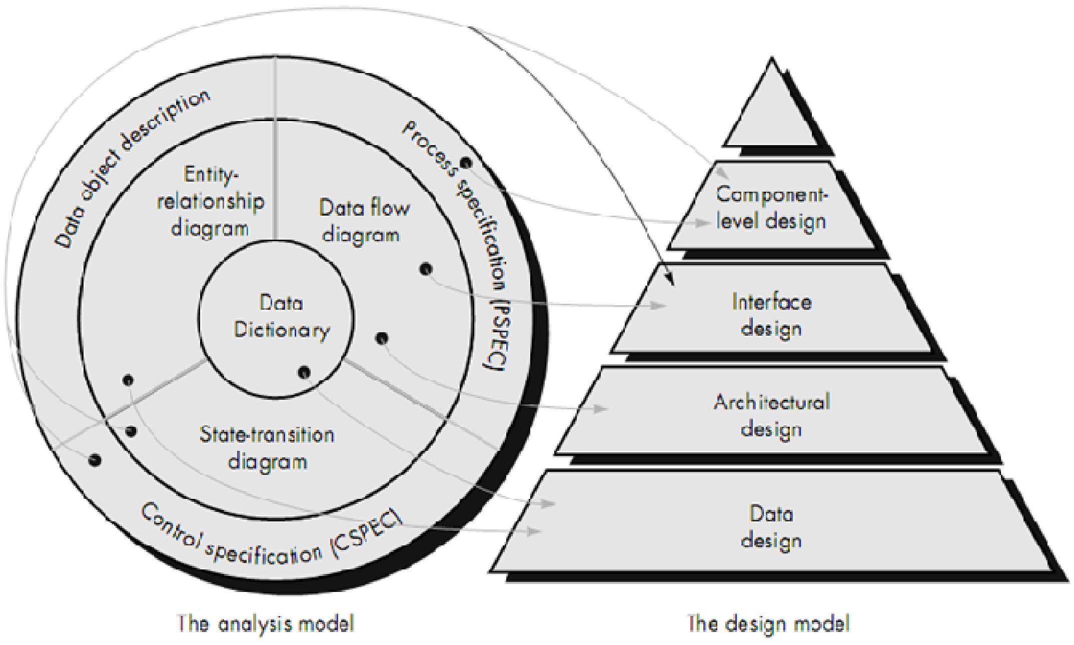
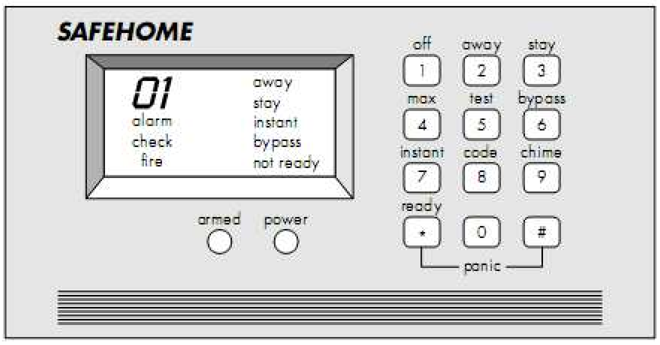
Special Edition for CSEDU Students

TOUCH-N-PASS EXAM CRAM GUIDE SERIES

SOFTWARE ENGINEERING





Prepared By

Sharafat Ibn Mollah Mosharraf

CSE, DU
12th Batch (2005-2006)

Includes Solutions to DU Software Engineering Final Exam Questions of 5 Years (2003-2007)

TABLE OF CONTENTS

CHAPTER 1
THE PRODUCT

Theories

1.1 Software Engineering

Software engineering is:

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

1.2 Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and bread-boarded prototype[1] evolve into a physical product (chips, circuit boards, power supplies, etc.).

Software is a *logical* rather than a *physical* system element. Therefore, 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 people applied and work accomplished is entirely different.[2]

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 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 steady-state 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.

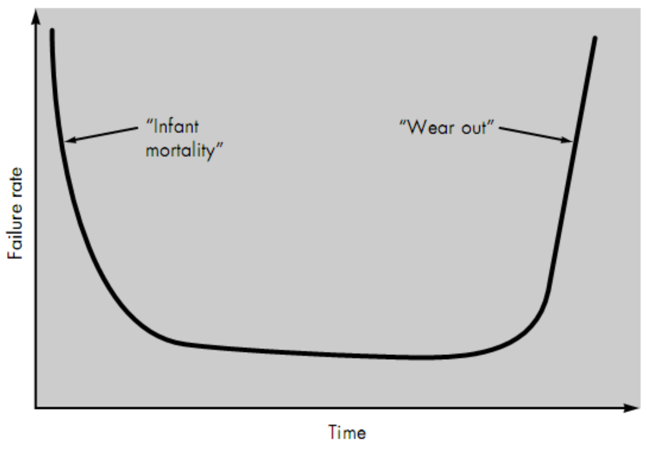


Figure 1.1: Failure curve for hardware.

[1] **Prototype:** Full-scale working model of something built for study or testing or display.

[2] **A brief explanation on the point:** Suppose, in hardware manufacturing, if 10 people can produce 100 items in 30 days, then 100 people will produce 100 items in 3 days. So, increasing people would decrease time of manufacturing. Therefore, if we get behind schedule, we can add more people to catch up. However, in case of software development, this idea is not true. In the words of Brooks: "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.

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 implication is clear — software doesn't wear out. But it does deteriorate[3]!

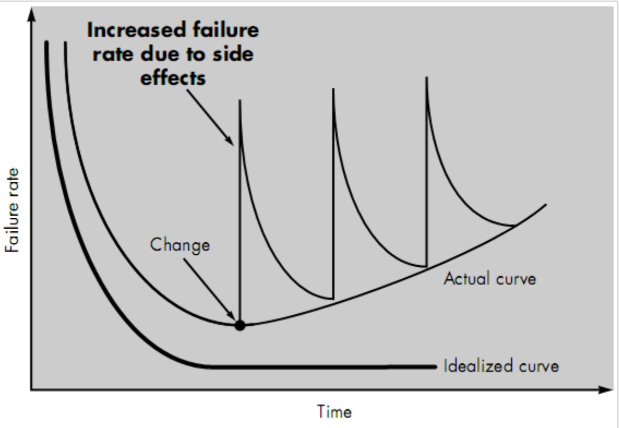


Figure 1.2: Idealized and actual failure curves for software.

This seeming contradiction can best be explained by considering the *actual curve* shown 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 steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise — the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. Most software is custom-built rather than being assembled from existing components.

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.

4. Software is reusable.

A software component should be designed and implemented so that it can be reused in many different programs. Today, the view of reuse is extended to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

Questions

1.1 **What characteristics of software make it different from others? [2004, Marks: 4]**

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. For example, 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.

[3] **Deteriorate:** Become worse or disintegrate.

2. ***Software doesn’t ‘wear out’***.

   Software is not susceptible to the environmental maladies that cause hardware to wear out. However, due to changes made to software, it deteriorates.

   …Include the figures 1.1 and 1.2 here…

3. ***Most software is custom-built rather than being assembled from existing components.***

   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.

CHAPTER 2
THE PROCESS

Theories

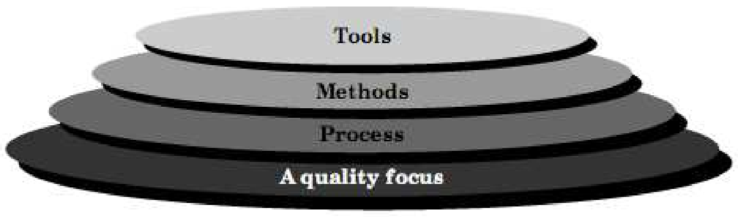
2.1 Software Process

When a product or system is built, it's important to go through a series of predictable steps — a road map that helps to create a timely, high-quality result. The road map followed is called a *software process*.

From a technical point of view, a software process is defined as a framework[4] for the tasks that are required to build high-quality software.[5]

2.2 Software Engineering Layers

FIGURE 2.1 Software engineering layers



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

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

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established. CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.

2.3 A Generic View of Software Engineering

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity. Each phase addresses one or more of the questions noted previously.

The *definition phase* focuses on *what*. That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending on the software engineering paradigm (or combination of paradigms) that is applied, three

[4] **Framework:** A structure supporting or containing something.

[5] Is *software process* synonymous with *software engineering*? The answer is *yes* and *no*. A software process defines the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process — management and technical methods and automated tools.

major tasks will occur in some form: system or information engineering, software project planning, and requirements analysis.

The *development phase* focuses on *how*. That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed. The methods applied during the development phase will vary, but three specific technical tasks should always occur: software design, code generation, and software testing.

The *support phase* focuses on *change* associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements. The support phase reapplies the steps of the definition and development phases but does so in the context of existing software. Four types of change are encountered during the support phase: correction, adaption, enhancement and prevention.

2.4 The Software Process Model

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.

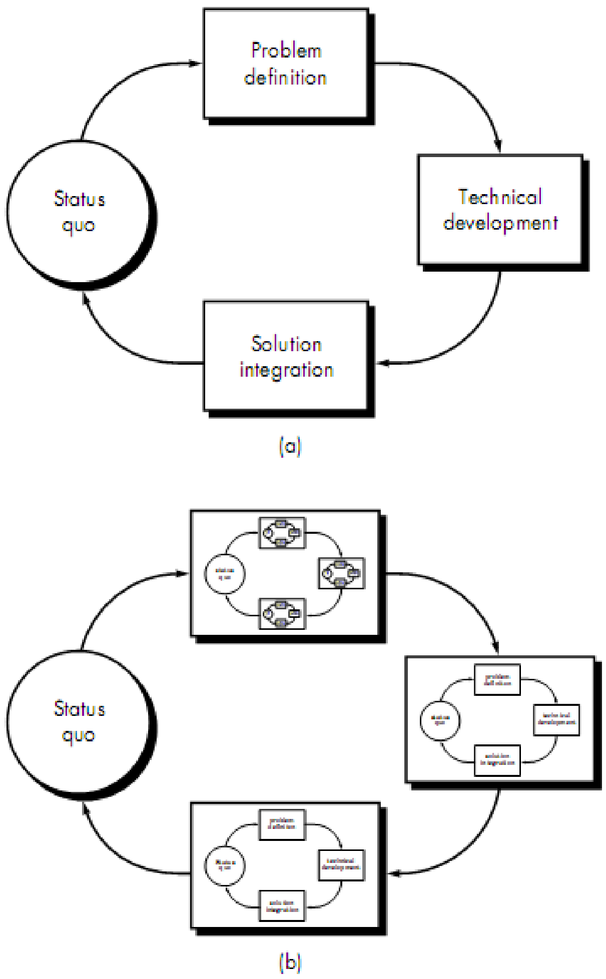
All software development can be characterized as a problem solving loop (depicted in the figure beside) in which four distinct stages are encountered: status quo, problem definition, technical development, and solution integration.

Status quo "represents the current state of affairs"; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results (e.g., documents, programs, data, new business function, new product) to those who requested the solution in the first place.

FIGURE 2.3

(a) The phases of a problem solving loop [RAC95]

(b) The phases within phases of the problem solving loop [RAC95]



2.5 The Linear Sequential / Waterfall / Classic Life Cycle Model

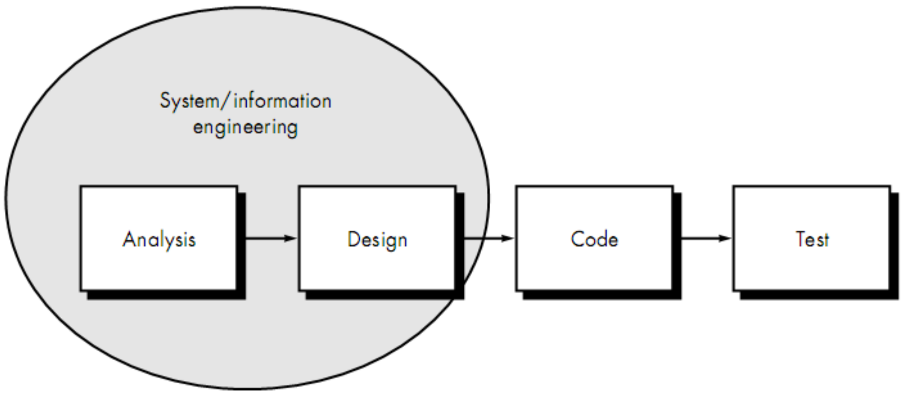


Figure 2.5: The Linear Sequential Model.

System / Information engineering and modeling:

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 requirement analysis:

- Information domain (data and events) for the software.
- Required function, behavior, performance, and interface.

Design:

- Data structure
- Software architecture
- Interface representations
- Procedural (algorithmic) detail.

Code generation:

The design is translated into a machine-readable form.

Testing:

- Logical internals of the software – ensuring that all statements have been tested
- Functional externals – 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. Changes will occur because:

- Errors have been encountered
- 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)
- The customer requires functional or performance enhancements.

Limitations / Problems of Linear Sequential Model:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

2.6 The Prototyping Model

Why needed?

- Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements.
- 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.

Description

- Developer and customer meet and define the overall objectives for the software.
- A quick design occurs focusing on 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 a prototype.
- The prototype is evaluated by the customer / user and used to refine requirements for the software to be developed.

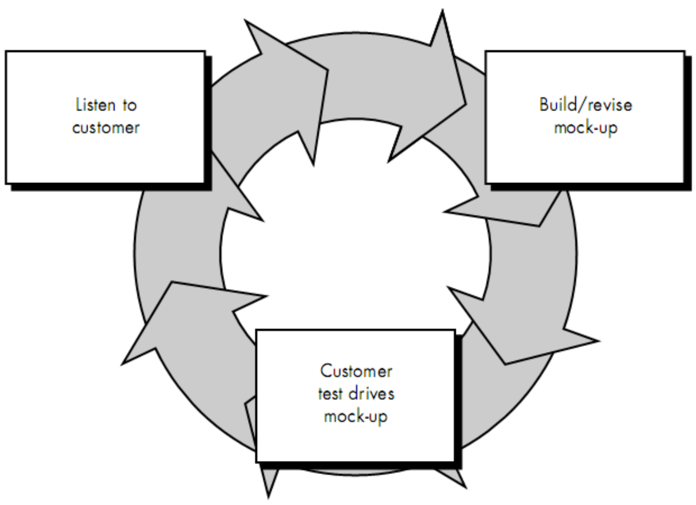


Figure 2.6: The Prototyping Model.

- 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.

Limitations / Problems of Prototyping Model:

1. Customer is unaware that nobody considered quality of the software. When he is informed that the software needs to be rebuilt, he demands that a few fixes should be applied to make the prototype a working product rather than rebuilding.
2. The developer forgets that some inappropriate algorithms were implemented.

2.7 The Spiral Model

Why needed?

Because, software is evolutionary. They evolve over a period time.

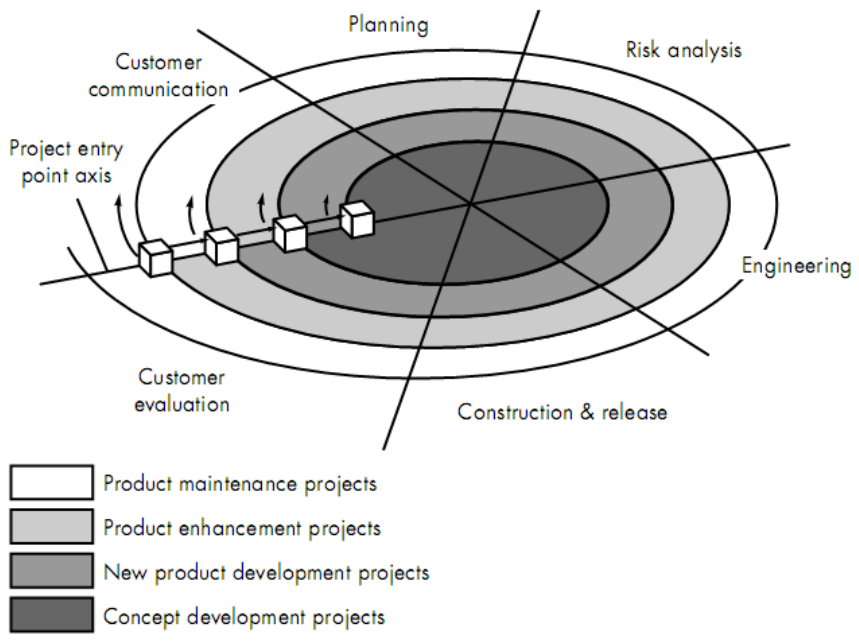


Figure 2.7: The Spiral Model.

Description

- The model is divided into a number of *task regions* (i.e., framework activities):
  - **Customer communication** — tasks required to establish effective communication between developer and customer.
  - **Planning** — tasks required to define resources, timelines, and other project-related information.
  - **Risk analysis** — tasks required to assess both technical and management risks.
  - **Engineering** — tasks required to build one or more representations of the application.
  - **Construction and release** — tasks required to construct, test, install, and provide user support (e.g., documentation and training).

  Each of the regions is populated by a set of work tasks, called a task set, that are adapted to the characteristics of the project to be undertaken.
- As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center.
- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation.

Questions

2.1 **Give a generic overview of software engineering. What is software process? Describe the software process model. [2003, Marks: 5]**

See Theories 2.3, 2.1 and 2.4.

2.2 **What is the difference between a software process model and a software process? Suggest two ways in which a software process model might be helpful in identifying possible process improvements. [2005, Marks: 5]**

2.3 **Briefly describe the linear sequential software engineering model. Why does the linear model sometimes fail? [2004, Marks: 6]**

The linear sequential software engineering model:

This model encompasses the following activities:

<u>System / information engineering and modeling:</u>

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.

<u>Software requirement analysis:</u>

To understand the nature of the program(s) to be built, the software engineer (*analyst*) must understand the information domain for the software, as well as required function, behavior, performance, and interface.

<u>Design:</u>

The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Software design focuses on data structure, software architecture, interface representations and algorithmic detail.

Code generation:

The design is translated into a machine-readable form.

Testing:

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 support / maintenance reapplies each of the preceding phases to an existing program.

…Include the figure of this model here…

Reasons for failure of this model:

1. As real projects are evolutionary, they rarely follow the sequential flow that the model proposes.
2. Often, the customer cannot state all his requirements. So, after all the processing steps are done, he demands some changes which causes much difficulty if this model is followed.
3. The customer has to wait long for a working version of the program to be available. A major blunder, if undetected until the working program is reviewed, can be disastrous.

2.4 **Describe the waterfall model of software process. What are the difficulties to follow waterfall model in developing real life systems? How can those be accomplished in the spiral process model? [2006, 2007. Marks: 6]**

OR, Provide a comparative analysis of the waterfall and spiral models. [In-course 1, 2008. Marks: 5]

See Question 2.3 for the waterfall model and its limitations.

How can the difficulties of the waterfall model be accomplished in the spiral process model:

In the spiral process model, a number of framework activities (customer communication, planning, risk analysis, engineering, construction and release) are performed linearly, but by circuiting around a spiral where each pass results in adjustments to the project plan according to customer feedback.

Hence, in spiral model, the customer can get a working version of the program quickly and provide suggestions and comments. Thus, program can be changed quickly and efficiently, and the evolutionary nature of the program can also be accommodated easily.

2.5 **Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model. [2003, 2005. Marks: 5]**

In the spiral process model, a number of framework activities (known as task regions) are performed:

- **Customer communication** — tasks required to establish effective communication between developer and customer.
- **Planning** — tasks required to define resources, timelines, and other project-related information.
- **Risk analysis** — tasks required to assess both technical and management risks.
- **Engineering** — tasks required to build one or more representations of the application.
- **Construction and release** — tasks required to construct, test, install, and provide user support.

These activities are performed sequentially just like in the waterfall model (requirement analysis, design, coding, testing, support).

However, in the spiral model, the software engineering team circuits around the spiral in a clockwise direction. The first circuit might result in the development of product specification; subsequent passes might be used to develop a prototype, and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. This is

	similar to the prototyping model where customer feedback is used to refine requirements for the software. Thus, both the waterfall model and the prototyping model can be accommodated in the spiral process model. *…Include the figure of this model here…*
2.6	**Describe the prototyping model of software engineering with some example software systems where it can be used most successfully. [2004, Marks 5; 2006, 2007, Marks: 4]** ***The prototyping model of software engineering:*** *Write the 'description' part of Theory 2.6 and provide the figure therein. Include the limitations of this model if you have enough time.* ***Example software systems where prototyping model can be used most successfully:*** It has been found that prototyping is very effective in the analysis and design of on-line systems[6], especially for transaction processing, where the use of screen dialogs is much more in evidence. The greater the interaction between the computer and the user, the greater the benefit is that can be obtained from building a quick system and letting the user play with it. Prototyping is especially good for designing good human-computer interfaces.

[6] **On-line System:** Connected to a computer network or accessible by computer. For example: on-line database etc.

CHAPTER 3
PROJECT MANAGEMENT CONCEPTS

Theories

3.1 Project Management

Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to an operational implementation.

Effective software project management focuses on the four P's: *people*, *product*, *process*, and *project*. The order is not arbitrary.

3.2 Description of the four P's

The People

- The people are the most important contributors to a successful software project.
- The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team / culture development.
- Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The Product

- Before a project can be planned, *product objectives and scope* should be established, alternative solutions should be considered, and technical and management constraints should be identified.
- Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.
- The software developer and customer must meet to define product objectives and scope.
- Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved.
- Scope identifies the primary data, functions and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.
- Once the product objectives and scope are understood, alternative solutions are considered.

The Process

A software process provides the framework from which a comprehensive plan for software development can be established.

The Project

In order to avoid project failure, a software project manager and the software engineers who build the product must:

- Avoid a set of common warning signs.
- Understand the critical success factors that lead to good project management.
- Develop a commonsense approach for planning, monitoring and controlling the project.

3.3 Detailed study of *people*

In this section, we examine the players who participate in the software process and the manner in which they are organized to perform effective software engineering.

The Players

1. **Senior managers** who define the business issues that often have significant influence on the project.
2. **Project (technical) managers** who must plan, motivate, organize, and control the practitioners who do software work.
3. **Practitioners** who deliver the technical skills that are necessary to engineer a product or application.
4. **Customers** who specify the requirements for the software to be engineered and other *stakeholders* who have a peripheral interest in the outcome.
5. **End-users** who interact with the software once it is released for production use.

Team Leaders

Characteristics a team leader should have

There are two views regarding the characteristics that a team leader should have:

1. The MOI model of leadership (*suggested by Weinberg*)

Motivation. The ability to encourage (by *push or pull*) technical people to produce to their best ability.

Organization. The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

2. Another View

Problem solving. An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity. A good project manager must take charge of the project. He must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement. To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to *read* people; he must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

The Software Team

The *best* team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei suggests three generic team organizations:

Democratic decentralized (DD). This software engineering team has no permanent leader. Rather, task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks. Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

Controlled decentralized (CD). This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

Controlled Centralized (CC). Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical.

Mantei describes seven project factors that should be considered when planning the structure of software engineering teams:

• The difficulty of the problem to be solved.
• The size of the resultant program(s) in lines of code or function points.
• The time that the team will stay together (team lifetime).
• The degree to which the problem can be modularized.
• The required quality and reliability of the system to be built.
• The rigidity of the delivery date.
• The degree of sociability (communication) required for the project.

The impact of project characteristics on team structure:

	DD	CD	CC	Explanation
Difficulty				Because a centralized structure completes tasks faster, it is the most adept at handling simple problems. Decentralized teams generate more and better solutions than individuals. Therefore such teams have a greater probability of success when working on difficult problems.
High	√			
Low		√	√	
Size				Because the performance of a team is inversely proportional to the amount of communication that must be conducted, very large projects are best addressed by teams with a CC or CD structures when subgrouping can be easily accommodated.
Large		√	√	
Small	√			
Team Lifetime				The length of time that the team will "live together" affects team morale. It has been found that DD team structures result in high morale and job satisfaction and are therefore good for teams that will be together for a long time.
Long	√			
Short		√	√	
Modularity				The DD team structure is best applied to problems with relatively low modularity, because of the higher volume of communication needed. When high modularity is possible (and people can do their own thing), the CC or CD structure will work well.
High		√	√	
Low	√			
Reliability				CC and CD teams have been found to produce fewer defects than DD teams, but these data have much to do with the specific quality assurance activities that are applied by the team.
High	√	√		
Low			√	
Delivery Date				Decentralized teams generally require more time to complete a project than a centralized structure and at the same time are best when high sociability is required.
Strict			√	
Lax	√	√		
Sociability				
High	√			
Low		√	√	

Questions

3.1	**Describe the importance of a team leader in a software project. What are the major characteristics that a team leader should have? [2006, 2007, Marks: 5]** ***Importance of a team leader in a software project*:** Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers." Therefore, to be effective, a project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader. ***Major characteristics of a team leader*:** *See the MOI model of leadership* (*in Theory 3.3*).
3.2	**Describe briefly the different team organizations along with impact of project characteristics on team structure. [In-course 1, 2008. Marks: 5]** *See Theory 3.3 – The Software Team.*

CHAPTER 4
SOFTWARE PROCESS AND PROJECT METRICS

Theories

4.1 Software Process and Project Metrics

Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy[7] of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

4.2 Measures, Metrics and Indicators

Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The IEEE Standard Glossary of Software Engineering Terms defines metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each). Software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per person-hour expended on reviews).

A software engineer collects measures and develops metrics so that indicators will be obtained. An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to:

1. Assess the status of an ongoing project.
2. Track potential risks.
3. Uncover problem areas before they go *critical*.
4. Adjust work flow or tasks.
5. Evaluate the project team's ability to control quality of software work products.

For example, four software teams are working on a large software project. Each team must conduct design reviews but is allowed to select the type of review that it will use. Upon examination of the metric, errors found per person-hour expended, the project manager notices that the two teams using more formal review methods exhibit an errors found per person-hour expended that is 40 percent higher than the other teams. Assuming all other parameters equal, this provides the project manager with an indicator that formal review methods may provide a higher return on time investment than another less formal review approach. He may decide to suggest that all teams use the more formal approach. The metric provides the manager with insight. And insight leads to informed decision making.

4.3 Software Measurement

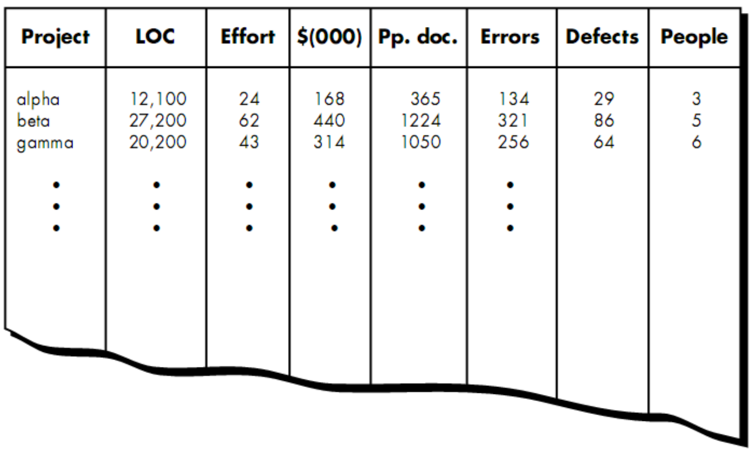
Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the *quality* of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. *Indirect measures* of the product include functionality, quality, complexity, efficiency, reliability, maintainability etc.

[7] **Efficacy:** Capacity or power to produce a desired effect.

4.4 Size-oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in *Figure 4.4*, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.



Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

Figure 4.4: Size-oriented Metrics.

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose *lines of code* as our normalization[8] value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects[9] per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Advantages of size-oriented metrics (i.e., the LOC measure)

1. LOC is an artifact[10] of all software development projects that can be easily counted.
2. Many existing software estimation models use LOC or KLOC as a key input.
3. A large body of literature and data predicated on LOC already exists.

Disadvantages of size-oriented metrics

1. LOC measures are programming language dependent.
2. They penalize well-designed but shorter programs.
3. They cannot easily accommodate nonprocedural languages.
4. Their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

4.5 Function-oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since functionality cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics are based on a measure called the *function point*. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Function points are computed by completing the table shown in *Figure 4.5*. Information domain values are defined in the following manner:

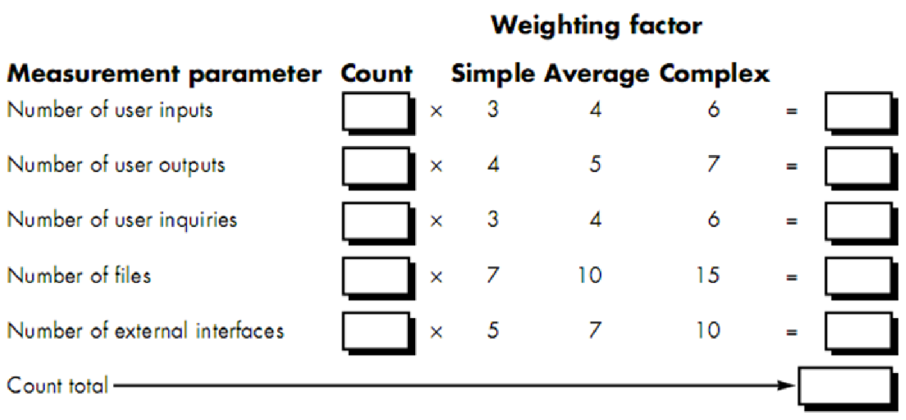
Number of user inputs. Each user input that provides distinct application-oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

[8] **Normalize:** Make normal / Cause to conform to a standard.

[9] A defect occurs when quality assurance activities (e.g., formal technical reviews) fail to uncover an error in a work product produced during the software process.

[10] **Artifact:** A man-made object taken as a whole.

Number of user outputs. Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.



		Weighting factor				
Measurement parameter	Count	Simple	Average	Complex		
Number of user inputs	☐	× 3	4	6	=	☐
Number of user outputs	☐	× 4	5	7	=	☐
Number of user inquiries	☐	× 3	4	6	=	☐
Number of files	☐	× 7	10	15	=	☐
Number of external interfaces	☐	× 5	7	10	=	☐
Count total					→	☐

Figure 4.5: Computing Function Points.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective[11].

To compute function points (FP), the following relationship is used:

$$\textbf{FP = count total} \times \mathbf{[0.65 + 0.01} \times \sum_{i=1}^{14} F_i \mathbf{]}$$

where count total is the sum of all FP entries obtained from *Figure 4.5*.

The F_i (i = 1 to 14) are *complexity adjustment values* based on responses to the following questions:

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

[11] **Subjective:** Taking place within the mind which is modified by individual bias. [In easy words – according to one's own will / choice / emotion]. The antonym of subjective is ***objective***, which means "undistorted by emotion or personal bias; based on observable phenomena" [in easy words – *logically*, not *emotionally*].

Each of these questions is answered using the scale below:

0 – Not important or applicable / No influence
1 – Incidental / Rarely needed
2 – Moderate
3 – Average
4 – Significant
5 – Essential

The constant values in the equation and the weighting factors that are applied to information domain counts are determined empirically[12].

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

4.6 Extended Function Point Metrics

Feature Points

The feature point measure accommodates applications in which algorithmic complexity is high. Real-time, process control and embedded software applications tend to have high algorithmic complexity and are therefore amenable to the feature point.

To compute the feature point, information domain values are again counted and weighted as described in function point metrics. In addition, the feature point metric counts new software characteristic— *algorithms*. An algorithm is defined as "a bounded computational problem that is included within a specific computer program”. Inverting a matrix, decoding a bit string, or handling an interrupt are all examples of algorithms.

3D Function Point

To compute 3D function points, the following relationship is used:

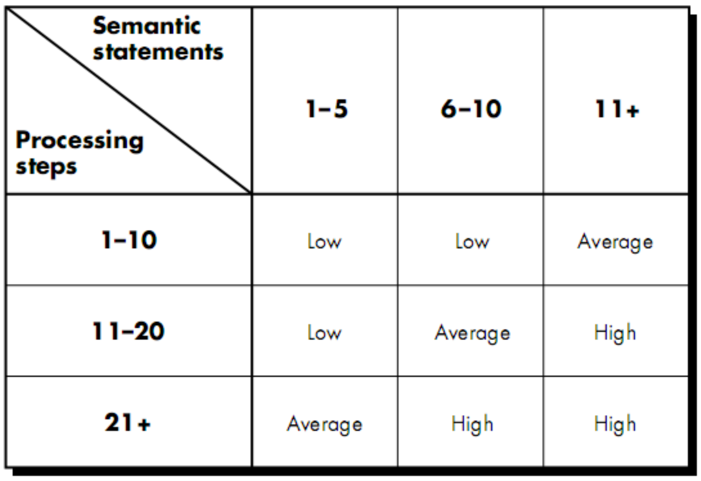
$$\text{index} = I + O + Q + F + E + T + R$$

where I, O, Q, F, E, T, and R represent complexity weighted values for the elements discussed already: inputs, outputs, inquiries, internal data structures, external files, transformation, and transitions, respectively. Each complexity weighted value is computed using the following relationship:

$$\text{complexity weighted value} = N_{il}W_{il} + N_{ia}W_{ia} + N_{ih}W_{ih}$$

where N_{il}, N_{ia}, and N_{ih} represent the number of occurrences of element i (e.g., outputs) for each level of complexity (low, medium, high); and W_{il}, W_{ia}, and W_{ih} are the corresponding weights. The overall complexity of a transformation for 3D function points is shown in the figure beside.

FIGURE 4.6 Determining the complexity of a transformation for 3D function points [WHI95].



Semantic statements / Processing steps	1–5	6–10	11+
1–10	Low	Low	Average
11–20	Low	Average	High
21+	Average	High	High

[12] **Empirical:** Derived from experiment and observation rather than theory.

4.7 Metrics for Software Quality

Measuring Quality

1. **Correctness**

   Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.

2. **Maintainability**

   Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements.

3. **Integrity**

   This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documents.

   To measure integrity, two additional attributes must be defined: *threat* and *security*. Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

   $$\textbf{integrity} = \sum [(\mathbf{1} - \boldsymbol{threat}) \times (\mathbf{1} - \boldsymbol{security})]$$

   where *threat* and *security* are summed over each type of attack.

4. **Usability**

   Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

   a. The physical and or intellectual skill required to learn the system.
   b. The time required to become moderately efficient in the use of the system.
   c. The net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient.
   d. A subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system.

Defect Removal Efficiency (DRE)

DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$\mathbf{DRE = E / (E + D)}$$

where **E** = The number of errors found before delivery of the software to the end-user
D = The number of defects found after delivery

The ideal value for DRE is 1. That is, no defects are found in the software.

Questions

4.1 Define and describe the measures, metrics and indicators in relation to their importance to software engineering. [2005, 2006, 2007. Marks: 4]

***Measures*:**

Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. For example, lines of code (LOC), efficiency (amount of person-month) etc.

***Metrics*:**

Metric can be defined as a quantitative measure of the degree to which a system, component, or process possesses a given attribute. Software metric relates the individual measures in some way (e.g., the average number of errors found per review etc.).

***Indicators*:**

An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to adjust the process or the project to make things better.

Importance of measures, metrics and indicators to software engineering:

If software is *not* measured, judgement can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

4.2 **Discuss the size-oriented metrics and function-point metrics. Compute the function-point value for the project with the following information domain characteristics:**

Number of inputs:	**24**
Number of outputs:	**45**
Number of files:	**40**
Number of inquiries:	**90**
Number of external interfaces:	**4**

Assume all complexity adjustment values are average. [2005, Marks: 6]

Size-oriented metrics:

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures can be created, where each software development project that has been completed over the past few years and corresponding measures for that project are listed.

In order to develop metrics that can be assimilated with similar metrics from other projects, *lines of code* is chosen as normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project, for example errors per KLOC, defects per KLOC etc.

Function-oriented metrics:

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Function-oriented metrics are based on a measure called the *function point*. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Here, the project planner estimates inputs, outputs, inquiries, files, external interfaces and each of the complexity weighting factors; and computes the complexity adjustment factor. Finally, the estimated number of FP is derived using the following formula:

$$\text{FP} = \text{count total} \times [0.65 + 0.01 \times \textstyle\sum_{i=1}^{14} F_i\ ]$$

Solution to mathematical Problem:

FP = (24 × 4 + 45 × 5 + 40 × 10 + 90 × 4 + 4 × 7) × [0.65 + 0.01 × (3 × 14)] = 1186.63

4.3 **What is a function point? What will be the function-point value for a project with the following information domain characteristics:**

Number of inputs:	**30**
Number of outputs:	**55**

Number of files:	**24**
Number of inquiries:	**8**
Number of external interfaces:	**2**

Assume all complexity adjustment values are average. [2004, Marks: 6]

Function Point:

A function point is a unit of measurement to express the amount of business functionality an information system provides to a user.

Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Solution to mathematical Problem:

FP = (30 × 4 + 55 × 5 + 24 × 10 + 8 × 4 + 2 × 7) × [0.65 + 0.01 × (3 × 14)] = 728.67

4.4 **Compute the function-point value for the project with the following information domain characteristics:**

Number of inputs:	**32**
Number of outputs:	**60**
Number of files:	**8**
Number of inquiries:	**24**
Number of external interfaces:	**2**

Assume that all complexity adjustment values are average. Assume that 14 algorithms have been counted. Compute feature point value under the same condition. [2007, Marks: 4]

FP = (32 × 4 + 60 × 5 + 8 × 10 + 24 × 4 + 2 × 7) × [0.65 + 0.01 × (3 × 14)] = 661.26

Feature point value

= (32 × 4 + 60 × 5 + 8 × 10 + 24 × 4 + 2 × 7 + 14) × [0.65 + 0.01 × (3 × 14)] = 676.24

4.5 **What do you mean by software quality? Discuss factors that affect software quality. [2007, Marks: 1 + 2]**

Software Quality:

Software quality means the collective quality of the requirement analysis that describe the problem, the design that models the solution, the code that leads to an executable program, and of the tests that exercise the software to uncover errors.

Factors affecting software quality:

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

Functionality. The degree to which the software satisfies stated needs as indicated by the following sub-attributes: suitability, accuracy, interoperability, compliance, and security.

Reliability. The amount of time that the software is available for use as indicated by the following sub-attributes: maturity, fault tolerance, recoverability.

Usability. The degree to which the software is easy to use as indicated by the following sub-attributes: understandability, learnability, operability.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following sub-attributes: time behavior, resource behavior.

Maintainability. The ease with which repair may be made to the software as indicated by the following sub-attributes: analyzability, changeability, stability, testability.

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following sub-attributes: adaptability, installability, conformance, replaceability.

CHAPTER 5
SOFTWARE PROJECT PLANNING: ESTIMATION

Theories

5.1 Software Project Planning

Software project planning actually encompasses project estimation, risk analysis and management, project scheduling and tracking, quality assurance, and configuration management. However, in the context of this chapter, planning involves *estimation* — the attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product.

5.2 Activities of Software Project Planning

1. **Determination of software scope**

   Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability.

2. **Estimation of the resources required to accomplish the software development effort**

   Figure 5.2 illustrates development resources as a pyramid.

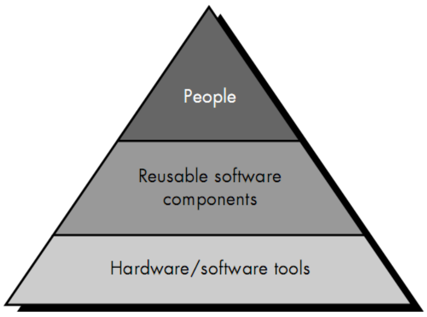


Figure 5.2: Software Project Resources.

   a. **Human Resources**

   b. **Reusable Software Resources**

   Bennatan suggests four software resource categories that should be considered as planning proceeds:

   i. **Off-the-shelf components.** Existing software that can be acquired from a third party or that has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

   ii. **Full-experience components.** Existing specifications, designs, built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, medications required for full-experience components will be relatively low-risk.

   iii. **Partial-experience components.** Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

   iv. **New components.** Software components that must be built by the software team specifically for the needs of the current project.

   The following guidelines should be considered by the software planner when reusable components are specified as a resource:

   1. If off-the-shelf components meet project requirements, acquire them. The cost for acquisition and integration of off-the-shelf components will almost always be less than the cost to develop equivalent software. In addition, risk is relatively low.
   2. If full-experience components are available, the risks associated with modification and integration is generally acceptable. The project plan should reflect the use of these components.
   3. If partial-experience components are available, their use for the current project must be

analyzed. If extensive modification is required before the components can be properly integrated with other elements of the software, proceed carefully — risk is high. The cost to modify partial-experience components can sometimes be greater than the cost to develop new components.

c. Environmental Resources

The environment that supports the software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.

5.3 Software Project Estimation Techniques ♣

There are several techniques for software project estimation. For example:

1. Decomposition techniques
   a. Software sizing
   b. Problem-based estimation
      i. LOC-based estimation
      ii. FP-based estimation
2. Empirical estimation models
   a. LOC-oriented empirical estimation models
      i. Walston-Felix model
      ii. Bailey-Basil model
      iii. Boehm simple model
      iv. Doty model for KLOC > 9
   b. FP- oriented empirical estimation models
      i. Albrecht and Gaffney model
      ii. Kemerer model
      iii. Matson, Barnett, and Mellichamp model
   c. COCOMO model
   d. Software equation

5.4 Empirical Estimation Models ♣

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP. Values for LOC or FP are estimated using the approach described in Sections 5.6.2 and 5.6.3. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The structure of estimation models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form:

$$\mathbf{E = A + B \times (\boldsymbol{ev})^C}$$

where *A*, *B*, and *C* are empirically derived constants, *E* is effort in person-months, and *ev* is the estimation variable (either LOC or FP).

In addition to the relationship noted in the above equation, the majority of estimation models have some form of project adjustment component that enables *E* to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). Among the many LOC-oriented estimation models proposed in the literature are:

$E = 5.2 \times (\text{KLOC})^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Bailey-Basili model
$E = 3.2 \times (\text{KLOC})^{1.05}$	Boehm simple model

♣ Not needed for the in-course or final exams. Skip if you wish. These two topics are provided only to help you *better* understand the COCOMO model.

$E = 5.288 \times (KLOC)^{1.047}$	Doty model for KLOC > 9

FP-oriented models have also been proposed. These include:

$E = 13.39 + 0.0545\ FP$	Albrecht and Gaffney model
$E = 60.62 \times 7.728 \times 10^{-8}\ FP^3$	Kemerer model
$E = 585.7 + 15.12\ FP$	Matson, Barnett, and Mellichamp model

5.5 The COCOMO (Constructive Cost Model) Model

In his classic book on "software engineering economics," Barry Boehm introduced a hierarchy of software estimation models bearing the name COCOMO. Boehm's hierarchy of models takes the following form:

Model 1: The *Basic* COCOMO Model – Computes software development effort (and cost) as a function of program size expressed in estimated lines of code (LOC).

Model 2: The *Intermediate* COCOMO Model – Computes software development effort (and cost) as a function of program size and a set of cost drivers that include subjective[13] assessments of product, hardware, personnel, and project attributes.

Model 3: The *Advanced* COCOMO Model – Incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on *each* step (analysis, design etc.) of the software engineering process.

The COCOMO models are defined for three classes of software projects. Using Boehm's terminology, these are:

1. **Organic mode –** Relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements (e.g., a thermal analysis program developed for a heat transfer group).
2. **Semi-detached mode –** An intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements (e.g., a transaction processing system with fixed requirements for terminal hardware and database software).
3. **Embedded mode –** A software project that must be developed within a set of tight hardware, software and operational constraints (e.g., flight control (auto-pilot) software for aircraft).

5.6 The Basic COCOMO Model Equation

The basic COCOMO model equation takes the form:

$$\mathbf{E} = a_b\ \mathbf{KLOC}^{b_b}$$
$$\mathbf{D} = c_b\ \mathbf{E}^{d_b}$$
$$\mathbf{N} = \mathbf{E} / \mathbf{D}$$

Here,		
	E	= effort applied (in person-months)
	D	= development time (in months)
	N	= number of people employed
	KLOC	= estimated number of lines of code for the project (expressed in thousands)
	a_b, b_b, c_b, d_b	= constants whose values can be found from the following table:

Project Class	a_b	b_b	c_c	d_d
Organic	2.4	1.05	2.5	0.38
Semi-organic	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

[13] **Subjective:** *See footnote 11 for the meaning.*

Questions

5.1	**Describe the function-point based estimation. [2003, Marks: 1 (*or 2*)]** In FP-based estimation, the project planner estimates inputs, outputs, inquiries, files, external interfaces and each of the complexity weighting factors; and computes the complexity adjustment factor. Finally, the estimated number of FP is derived using the following formula: $\text{FP} = \text{count total} \times [0.65 + 0.01 \times \sum_{i=1}^{14} F_i\ ]$
5.2	**Briefly describe the COCOMO model of software estimation. [2003, 2004, 2005, Marks: 4; 2007, Marks: 3]** *See Theory 5.5.*
5.3	**What are the project resources? Describe COCOMO model II. Use COCOMO model II to estimate the effort required to build software for a simple ATM that produces 12 screens, 20 reports and will require approximately 88 software components. Assume average complexity and average development environment maturity. Use application composition model with object points. [2005, Marks: 6]** The project resources are: 1. Human Resources. 2. Reusable Software Resources. 3. Environmental Resources. [*COCOMO Model II is out of scope of our syllabus. However, the solution to the mathematical problem is provided below:*] New Object Point, NOP = 12 × 2 + 20 × 5 + 88 = 212 Productivity Rate, PROD = 13 ∴ Estimated Effort = NOP / PROD = 212 / 13 = 16.31

CHAPTER 6
RISK ANALYSIS & MANAGEMENT

Theories

6.1 Conceptual Definition of Risk

First, risk concerns *future happenings*.

Second, risk involves *change*, such as in changes of mind, opinion, actions, or places.

Third, risk involves *choice*, and the *uncertainty* that choice itself entails.

Risk Analysis and 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 a 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.

6.2 Risk (Management) Strategies

4. **Reactive Risk Strategy**

   According to this strategy, the action that should be taken is decided only when a problem *actually* occurs.

5. **Proactive Risk Strategy**

   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 primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

6.3 Characteristics of Risk

1. **Uncertainty** — the risk may or may not happen; that is, there are no *100% probable* risks.[14]
2. **Loss** — if the risk becomes a reality, unwanted consequences or losses will occur.

6.4 Categories of Risk

Categories of Risks according to the level of uncertainty and the degree of loss associated with them

1. **Project Risks**

   Threatens: The project plan.

   Loss associated: It is likely that project schedule will slip and that costs will increase.

   Identifies: Potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty are also defined as project risk factors.

2. **Technical Risks**

   Threatens: The quality and timeliness of the software to be produced.

   Loss associated: Project implementation may become difficult or impossible.

[14] A risk that is 100% probable is a *constraint* on the software project.

Identifies: Potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "*leading-edge*" technology are also risk factors.

Reason of occurrence: Technical risks occur because the problem is harder to solve than we thought it would be.

3. Business Risks

Threatens: The viability[15] of the software to be built.

Loss associated: Often jeopardize[16] the project or the product.

Candidates:

(1) **Market Risk –** building an excellent product or system that no one really wants.
(2) **Strategic Risk –** building a product that no longer fits into the overall business strategy.
(3) Building a product that the sales force doesn't understand how to sell.
(4) **Management Risk –** losing the support of senior management due to a change in focus or a change in people.
(5) **Budget Risks –** losing budgetary or personnel commitment.

General categorization of risks

1. Known Risks

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

2. Predictable Risks

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

6.5 Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan.

By identifying known and predictable risks, the project manager takes a first step towards avoiding them when possible and controlling them when necessary.

Identifying Risks / Risk Item Checklist

A risk item checklist can be created which can be used for risk identification.

- **Product size** — risks associated with the overall size of the software to be built or 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.

[15] **Viability:** Practicality; Usefulness.

[16] **Jeopardize:** Put at risk; Pose a threat to; Present a danger to.

Questions

6.1 **Why is risk important in Software Engineering? What is the difference between *reactive* and *proactive* risk strategies? [2006, Marks: 4]**

Software is a difficult undertaking[17]. Lots of things can go wrong, and in fact, many often do. It's for this reason that being prepared — understanding the risks and taking proactive measures to avoid or manage them — is a key element of good software project management.

Differences between reactive and proactive risk strategies:

Reactive Strategy	Proactive Strategy
1. Action that should be taken is decided only when a problem *actually* occurs.	1. 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.
2. When reactive strategy fails, "crisis management" takes over and the project goes in real jeopardy.	2. Plan is established before a risk becomes a reality. So, measures are taken beforehand so that risk management does not fail.

6.2 **What type of risks are we likely to encounter as the software is built? [2004. Marks: 5]**

See Theory 6.4.

6.3 **What is risk projection? Explain the steps involved in risk projection. [2006. Marks: 4]**

Out of our syllabus…

6.4 **Write down the characteristics of risks. [2007. Marks: 2]**

See Theory 6.3.

6.5 **What is risk identification? Explain the method to create a risk item checklist for risk identification. [2007. Marks: 1 + 2]**

See Theory 6.5.

6.6 **Describe the difference between risk components and risk drivers. [2007. Marks: 2]**

The difference between risk components and risk drivers is that risk drivers *affect* risk components. There are four risk components – performance, cost, support and schedule risks. The impact of each risk driver on the risk component is divided into one of four impact categories – negligible, marginal, critical or catastrophic.

[17] **Undertaking:** Any piece of work that is undertaken or attempted.

CHAPTER 12
ANALYSIS MODELING

Theories

12.1 Analysis Modeling

Analysis modeling uses a combination of text and diagrammatic forms to depict requirements for data, function, and behavior in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

12.2 Objectives of the Analysis Model

The analysis model must achieve three primary objectives:

1. To describe what the customer requires,
2. To establish a basis for the creation of a software design, and
3. To define a set of requirements that can be validated once the software is built.

12.3 The Structure / Elements of the Analysis Model

To accomplish the objectives of the analysis model, the analysis model derived during structured analysis takes the form illustrated in *figure* 12.3.

At the core of the model lies the *data dictionary* — a repository that contains descriptions of all data objects consumed or produced by the software.

Three different diagrams surround the core.

The *entity relation diagram* (ERD) depicts relationships between data objects. The attributes of each data object noted in the ERD can be described using a *data object description*.

The *data flow diagram* (DFD) serves two purposes:

1. To provide an indication of how data are transformed as they move through the system.

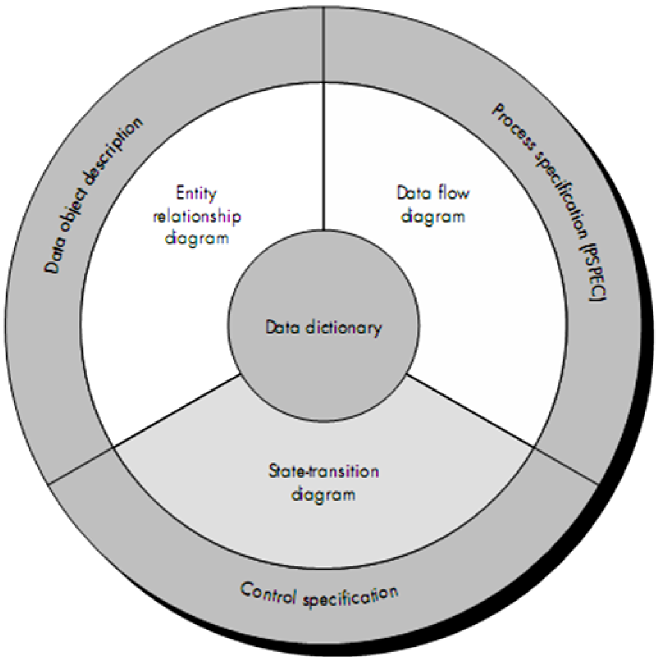


Figure 12.3: The Structure of the analysis model.

2. To depict the functions (and subfunctions) that transform the data flow.

A description of each function presented in the DFD is contained in a *process specification* (PSPEC).

The *state transition diagram* (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. Additional information about the control aspects of the software is contained in the *control specification* (CSPEC).

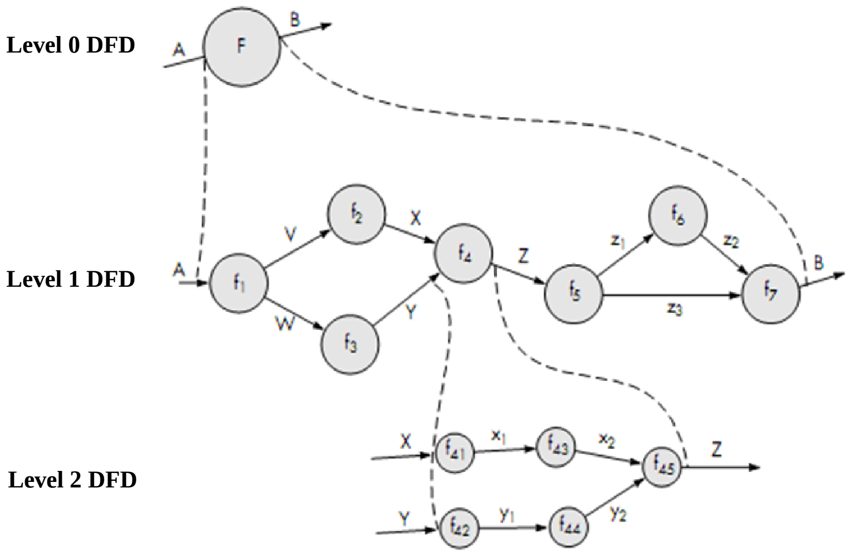
12.4 Data Flow Diagram (DFD) / Data Flow Graph / Bubble Chart

A *data flow diagram* is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output.

Levels of Abstraction of DFD

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail.

A ***level 0 DFD***, also called a *fundamental system model* or a *context model*, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively.



Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a ***level 1 DFD*** might contain five or six bubbles with inter-connecting arrows. Each of the processes represented at level 1 is a subfunction of the overall system depicted in the context model.

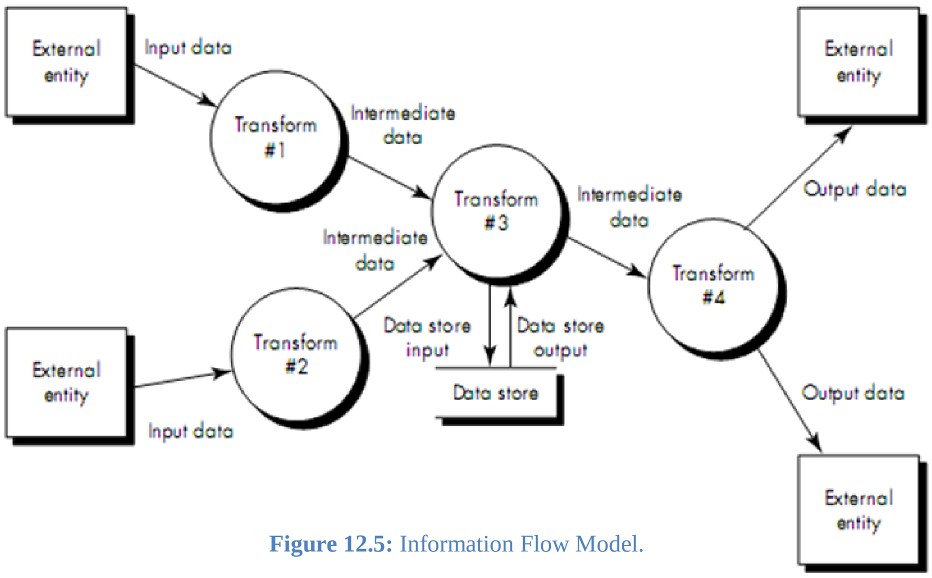
Figure 12.4: Levels of DFD.

As was noted earlier, each of the bubbles may be refined or layered to depict more detail. *Figure* 12.4 illustrates this concept.

12.5 Representation of a DFD

A computer-based system is represented as an information transform as shown in *figure* 12.5.

A *rectangle* is used to represent an external entity; that is, a system element (e.g., hardware, a person, another program etc.) or another system that produces information for transformation by the software or receives information produced by the software.



A *circle* (sometimes called a *bubble*) represents a process or transform that is applied to data (or control) and changes it in some way.

Figure 12.5: Information Flow Model.

An *arrow* represents one or more data items (data objects). All arrows on a data flow diagram should be labeled.

The *double line* represents a data store — stored information that is used by the software.

12.6 Creating a Data Flow Model

A few simple guidelines can aid immeasurably during derivation of a data flow diagram:

1. The level 0 data flow diagram should depict the software/system as a single bubble.
2. Primary input and output should be carefully noted.
3. Refinement should begin by isolating candidate processes, data objects, and stores to be represented at the next level.
4. All arrows and bubbles should be labeled with meaningful names.

5. Information flow continuity must be maintained from level to level.
6. One bubble at a time should be refined.

There is a natural tendency to overcomplicate the data flow diagram. This occurs when the analyst attempts to show too much detail too early or represents procedural aspects of the software in lieu of information flow.

12.7 Example of Level 0, Level 1 and Level 2 DFDs

Software Product Description:

SafeHome software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the SafeHome control panel shown in *figure* 12.7.

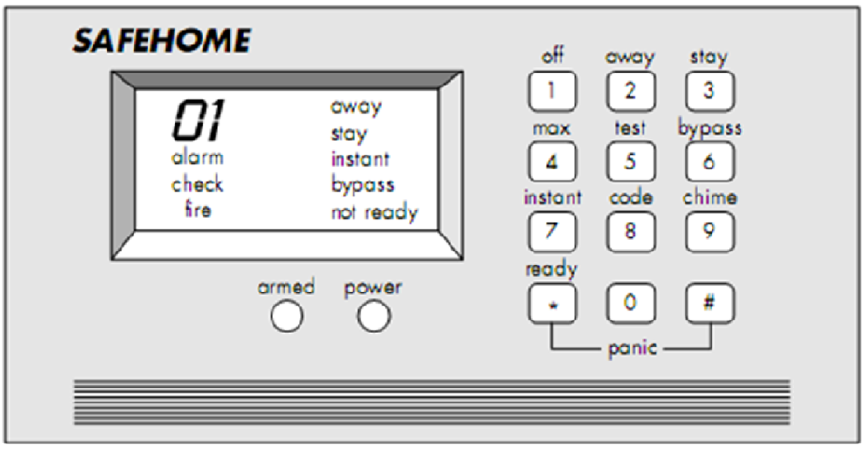


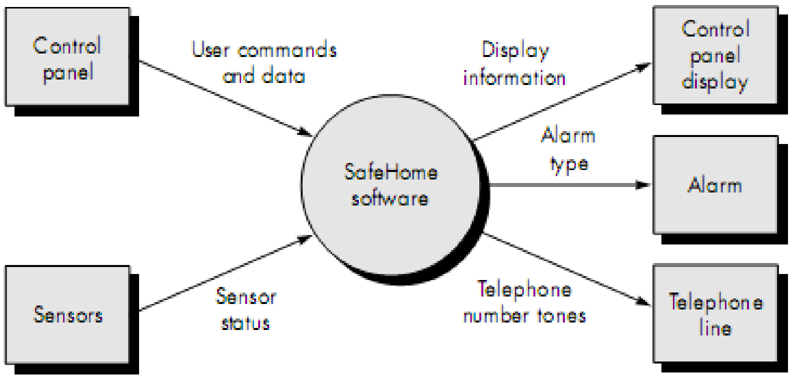
Figure 12.7: SafeHome Control Panel.

During installation, the SafeHome control panel is used to *program* and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

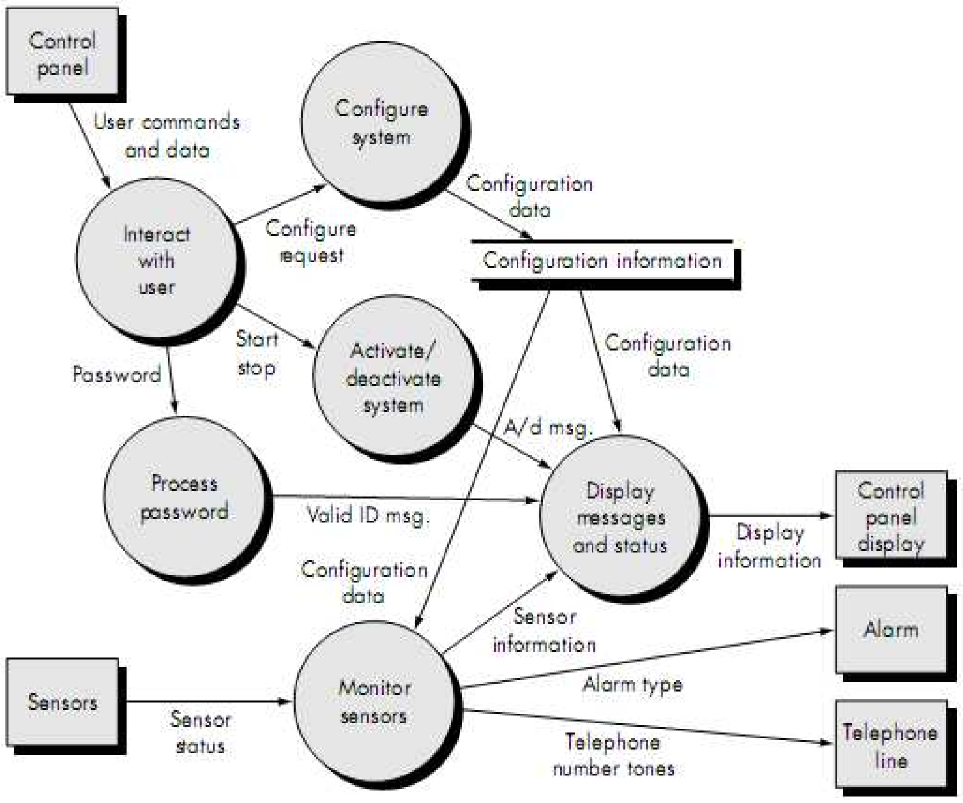
When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until telephone connection is obtained.

All interaction with SafeHome is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, and displays system status information on the LCD display. Keyboard interaction takes the following form…

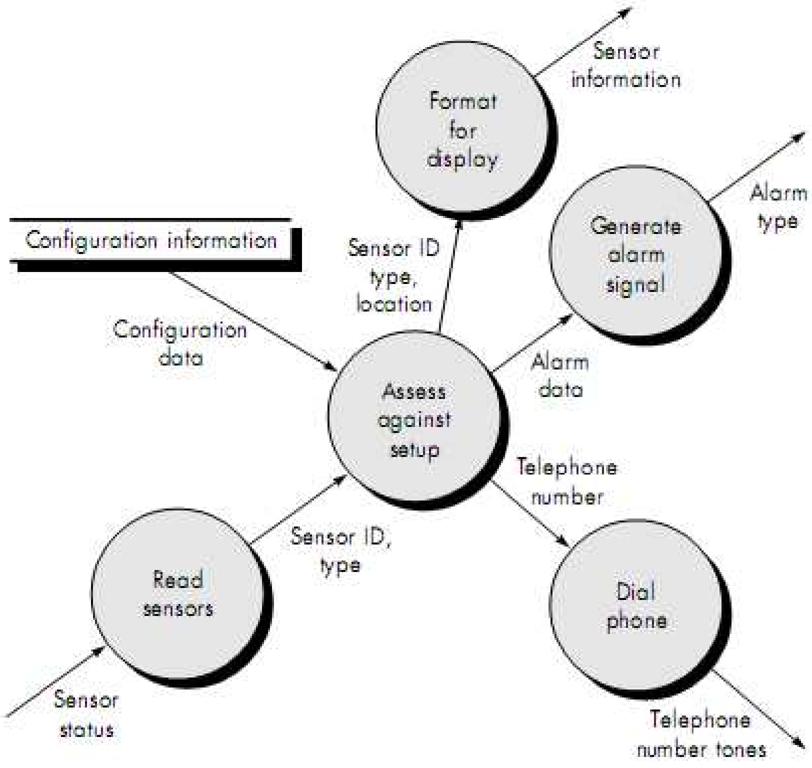
***Level 0 DFD for SafeHome*:**



***Level 1 DFD*:**



***Level 2 DFD that refines the monitor sensors process*:**



12.8 Data Dictionary

The data dictionary is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations.

Although the format of dictionaries varies from tool to tool, most contain the following information:

- **Name** — the primary name of the data or control item, the data store or an external entity.
- **Alias** — other names used for the first entry.
- **Where used / how used** — a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.
- **Content description** — a notation for representing content.
- **Supplementary information** — other information about data types, preset values (if known), restrictions or limitations, and so forth.

Questions

12.1 **What are the objectives of analysis model? Briefly describe the elements of analysis model. [In-course 2, 2008. Marks: 5]**

See Theories 12.2 and 12.3.

12.2 **You have been asked to build a network-based course registration system for your university. Develop an entity-relationship diagram that describes data objects, relationships and attributes. Also design the data flow diagram of the system. [2003. Marks: 7]**

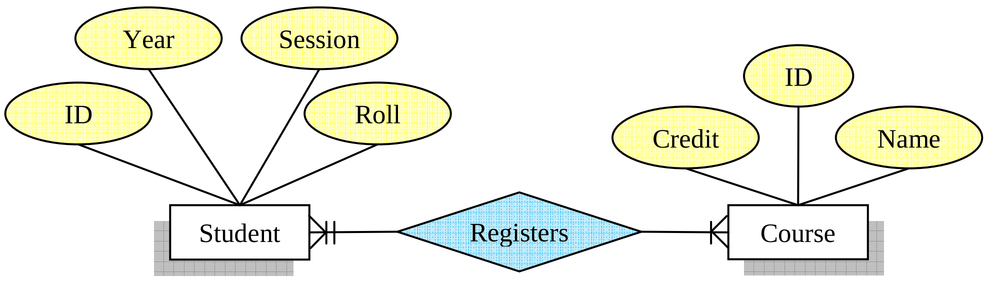


Figure: Entity-Relationship Diagram for Course Registration System.

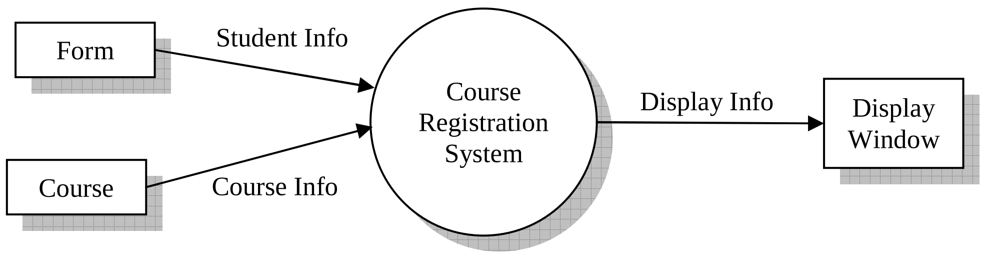
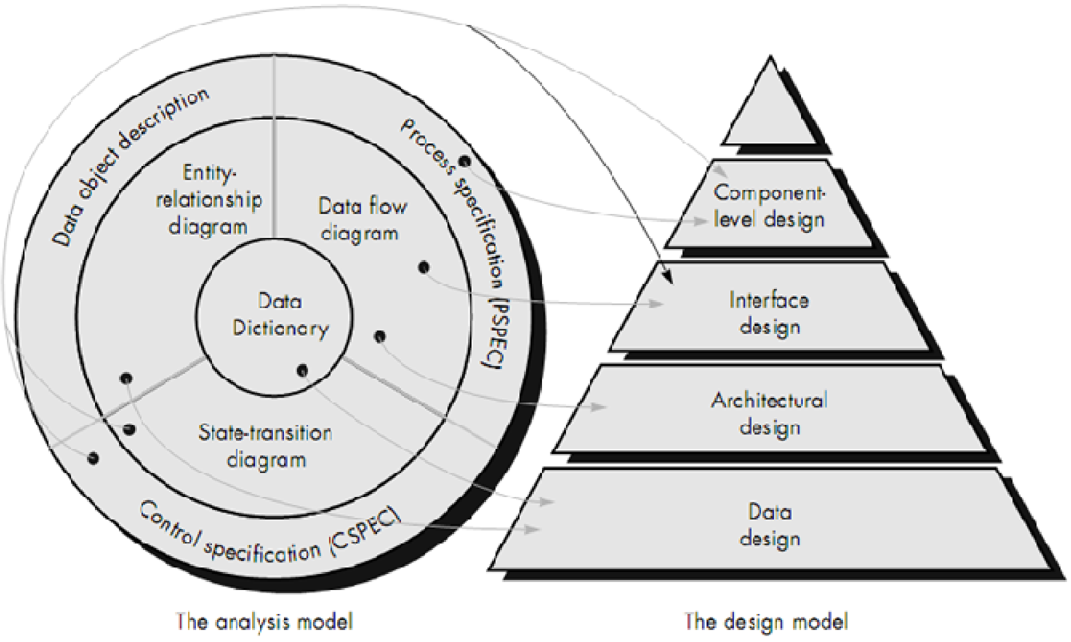


Figure: Level 0 DFD for Course Registration System.

CHAPTER 13
DESIGN CONCEPTS & PRINCIPLES

Theories

13.1	**Design** Design is a meaningful engineering representation of something that is to be built. In the software engineering context, *design* focuses on *four* major areas of concern: data, architecture, interfaces and components.
13.2	**Translating the Analysis Model into a Design Model**  Data object description Process specification (PSPEC) Entity-relationship diagram Data flow diagram Data Dictionary State-transition diagram Control specification (CSPEC) Component-level design Interface design Architectural design Data design The analysis model The design model **Figure 13.2:** Translating the Analysis Model into a Design Model. Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in *figure* 13.2. The *data design* transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. The *architectural design* defines the relationship between major structural elements of the software, the "*design patterns*" that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied. The architectural design representation — the framework of a computer-based system — can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model. The *interface design* describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design. The *component-level design* transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.

13.3 Characteristics of a Good Design / Goal of the Design Process

1. The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
2. The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
3. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

13.4 (Technical) Criteria for Good Design / How the Goals of the Design Process can be Achieved

1. A design should exhibit an architectural structure that
   (1) has been created using recognizable design patterns,
   (2) is composed of components that exhibit good design characteristics, and
   (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be *modular*; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.
3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).
4. A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

13.5 Basic Design Principles

- **The design process should not suffer from "*tunnel vision*[18]".** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.
- **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
- **The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.** That is, the structure of the software design should (whenever possible) mimic[19] the structure of the problem domain.
- **The design should exhibit uniformity and integration.** A design is uniform if it appears that *one* person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

[18] **Tunnel Vision:** Visual defect/disorder.

[19] **Mimic:** Imitate; Appear alike, as in behavior or appearance.

- **The design should be structured to accommodate *change*.**
- **The design should be structured to degrade gently, even when abnormal data, events, or operating conditions are encountered.** Well-designed software should never “*bomb*”. It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful[20] manner.
- **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
- **The design should be assessed for quality as it is being created, not after the fact.**
- **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minor issues when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency etc.) have been addressed before worrying about the syntax of the design model.

13.6 Effective Modular Design: Functional Independence

Functional independence is achieved by developing modules with "*single-minded*"[21] function and an "*aversion*"[22] to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure.

13.7 Cohesion and Coupling

Functional independence is measured using two qualitative criteria: *cohesion* and *coupling*.

A *cohesive* module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just *one* thing.

Cohesion may be represented as a "*spectrum*"[23]. We always strive for *high* cohesion, although the mid-range of the spectrum is often acceptable.

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for *lowest* possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "*ripple effect*" caused when errors occur at one location and propagate through a system.

Questions

13.1 Briefly explain how software requirement analysis is linked with software engineering and software design. [2006. Marks: 4]

See Theory 13.2.

13.2 Describe the principles of software design. [In-course 2, 2008. Marks: 5]

See Theory 13.5.

---

[20] **Graceful:** Characterized by beauty of movement, style, form, or execution.
[21] A module is “*single-minded*” if it can be described with a simple sentence — subject, predicate, object.
[22] **Aversion:** A feeling of extreme dislike.
[23] **Spectrum:** A broad range of related objects or values or qualities or ideas or activities.

CHAPTER 16
SOFTWARE TESTING TECHNIQUES

Theories

16.1 Introduction

Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors — but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic of software components, and (2) exercise the input and output domains of the program to uncover errors in program function, behavior and performance.

16.2 Testing Objectives

In an excellent book on software testing, Glen Myers states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

In a word, our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

16.3 Testing Benefits

1. If testing is conducted successfully, it will uncover errors in the software.
2. Testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.
3. Data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.

16.4 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set of testing principles that have been adapted for use in this book:

- **All tests should be traceable to customer requirements.** As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- **Tests should be planned long before testing begins.** Test planning (Chapter 18) can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- **The Pareto principle applies to software testing.** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
- **Testing should begin "in the small" and progress toward testing "in the large."** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system (Chapter 18).
- **Exhaustive testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every

combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

- **To be most effective, testing should be conducted by an independent third party.** By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing). For reasons that have been introduced earlier in this chapter and are considered in more detail in Chapter 18, the software engineer who created the system is not the best person to conduct all tests for the software.

16.5 Testability

Software testability is simply how easily [a computer program] can be tested. The checklist that follows provides a set of characteristics that lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."

- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development and testing).

Observability. "What you see is what you test."

- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

Controllability. "The better we can control the software, the more the testing can be automated and optimized."

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

- The software system is built from independent modules.
- Software modules can be tested independently.

Simplicity. "The less there is to test, the more quickly we can test it."

- Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements).
- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing."

- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.

Understandability. "The more information we have, the smarter we will test."

- The design is well understood.
- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated.

- Technical documentation is instantly accessible.
- Technical documentation is well organized.
- Technical documentation is specific and detailed.
- Technical documentation is accurate.

16.6 White-Box / Glass-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.

16.7 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.

Error Categories That Can be Found Using Black-Box Testing

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 database access,

(4) behavior or performance errors, and

(5) initialization and termination errors.

Test Case Design Criteria for Black-Box Testing

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

(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.

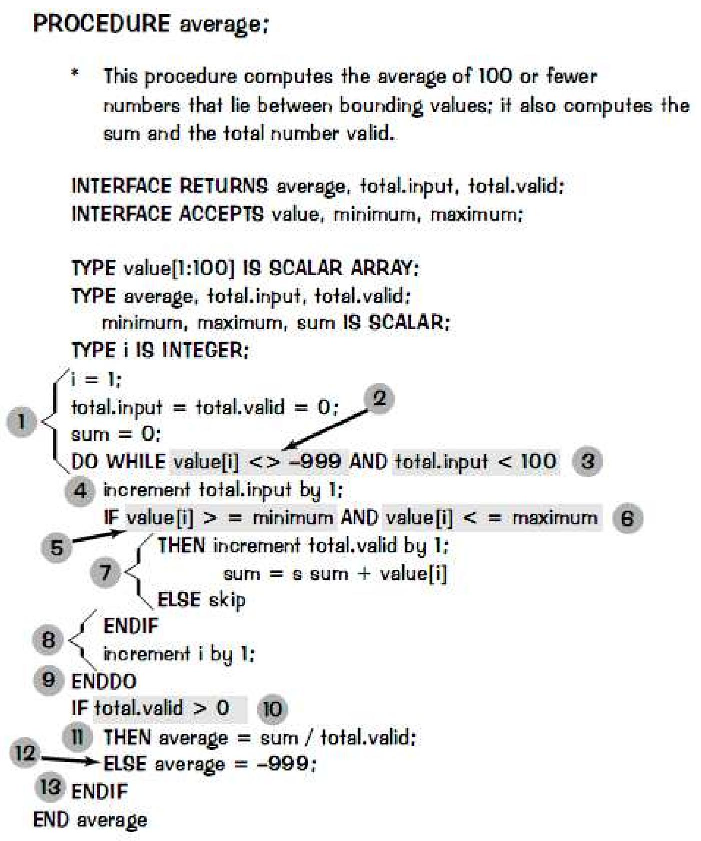
16.8 Cyclomatic Complexity

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.

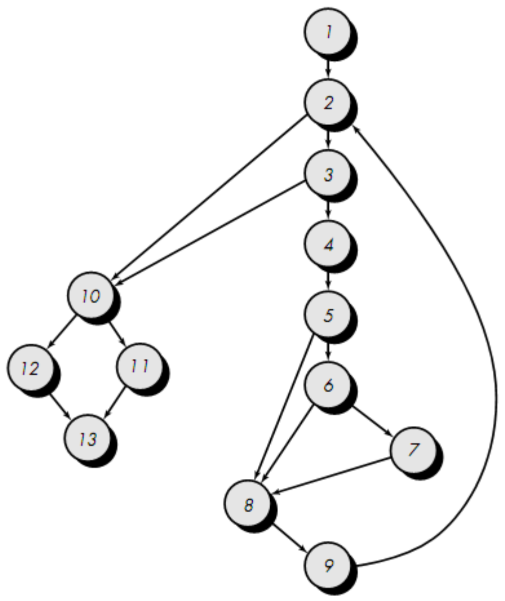
Independent Path

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 graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

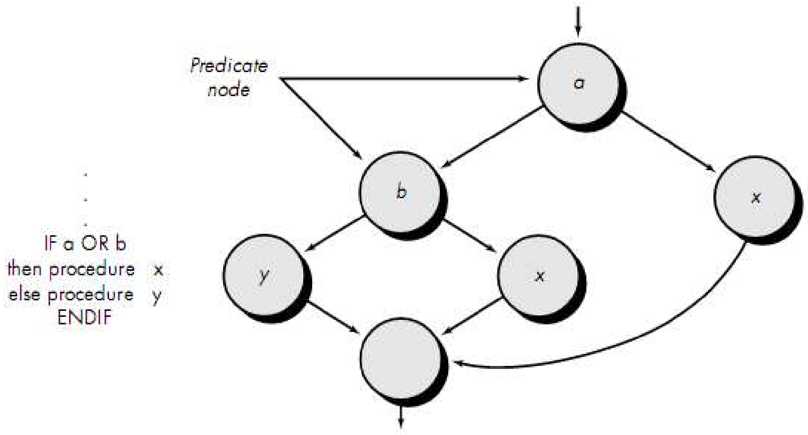
16.9 **Converting a Code Segment into a Flow Graph**



```
PROCEDURE average;

    *   This procedure computes the average of 100 or fewer
        numbers that lie between bounding values; it also computes the
        sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100
        increment total.input by 1;
        IF value[i] > = minimum AND value[i] < = maximum
            THEN increment total.valid by 1;
                sum = s sum + value[i]
            ELSE skip
        ENDIF
        increment i by 1;
    ENDDO
    IF total.valid > 0
        THEN average = sum / total.valid;
        ELSE average = -999;
    ENDIF
END average
```



Case where there is an *OR* in code segment rather than an *AND*



Note: Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

Questions

16.1 **What is the principle of software testing? Describe testability. [2003, 2004. Marks: 5]**

The principle of software testing is to use techniques that provide systematic guidance for designing tests that (1) exercise the internal logic of software components, and (2) exercise the input and output domains of the program to uncover errors in program function, behavior and performance.

See Theory 16.5 for Testability.

16.2 **Discuss the differences between verification and validation. [2005, 2006. Marks: 2]**
Explain why validation is a particularly difficult process. [2005. Marks: 3]

[Out of our syllabus. But the answer to the first question is provided below.]

Differences between verification and validation:

Verification refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

16.3 **What is White box testing? Differentiate between white box testing and black box testing. [2004, 2005, 2006, 2007. Marks: 5]**

White-box testing:

White-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.

Difference between white-box and black-box testing:

White-Box Testing	Black-Box Testing
White-box testing is a test case design method that uses the control structure of the procedural design to derive test cases.	*Black-box 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.
White-box testing is performed early in the testing process.	Black-Box testing tends to be applied during later stages of testing.
White-box testing of software is predicated on close examination of procedural detail.	Black-Box testing alludes to tests that are conducted at the software interface.

16.4 **Describe basis path testing method. [2003. Marks: 5]**

The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

[Now describe each point… How? Well, try yourself…!]

16.5 **For the following code segment,**

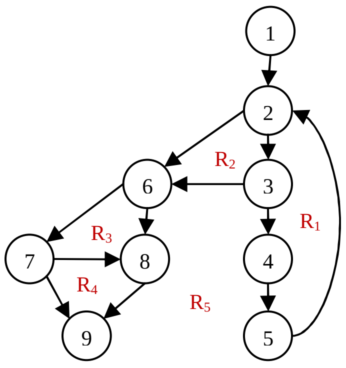
i) Draw flow graph.
ii) Calculate cyclomatic complexity of the flow graph.
iii) Calculate cyclomatic complexity using graph matrix. [2004, 2005, 2006, 2007. Marks: 6]

```
for (i = 0; i < length – 1 and ch != EOF; i++) {
   str[i] = c;
   c = getch();
}
if (c == '\n' || c == '\r')
   str[i] = c;
```

i) এটা আমরা দুভাবে করতে পারি – ১. কোডটাকে while–এ কনভার্ট করে Theory 16.9-এর মত করে সলভ করা। ২. এ অবস্থায় রেখেই সলভ করা। তবে সব ক্ষেত্রেই cyclomatic complexity একই আসবে।

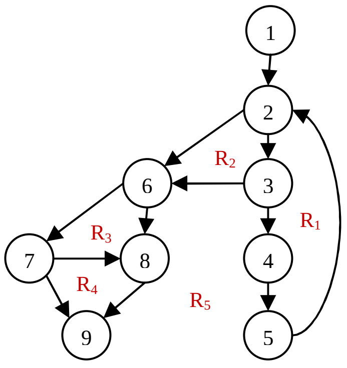
Way 1

```
                 2                      3
1  i = 0;
   DO WHILE i < length - 1 AND ch != EOF
          / i++;
      4  <  str[i] = c;
          \ c = getch();
5  ENDDO
          6              7
   IF c == '\n' OR c == '\r'
          8 THEN str[i] = c;
9  ENDIF
```



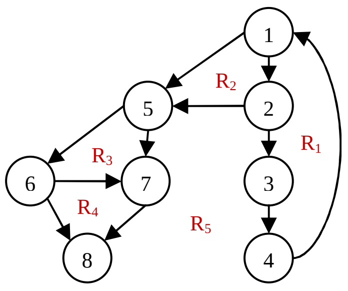
Way 2

```
           1          2                   3
   for (i = 0; i < length - 1 and ch != EOF; i++) {
       4  / str[i] = c;
          \ c = getch();
5  }
           6            7
   if (c == '\n' || c == '\r'){
         8 str[i] = c;
9  }
```



Way 3

```
                      1                   2
   for (i = 0; i < length - 1 and ch != EOF; i++) {
       3  / str[i] = c;
          \ c = getch();
4  }
            5            6
   if (c == '\n' || c == '\r'){
         7 str[i] = c;
8  }
```



আমরা এখানে Way 2 (ইচ্ছামত) বেছে নিলাম বাকি প্রশ্নগুলোর উত্তর দেওয়ার জন্য।

ii) Cyclomatic Complexity:

1. Using number of regions: 5.
2. Using $V(G) = E - N + 2 = 12 - 9 + 2 = 5$.
3. Using $V(G) = P + 1 = 4 + 1 = 5$. [The predicate nodes are 2, 3, 6 and 7]

iii) Cyclomatic Complexity using graph matrix:

Connected to Node

Node	1	2	3	4	5	6	7	8	9	Connections
1		1								1 – 1 = 0
2			1			1				2 – 1 = 1
3				1		1				2 – 1 = 1
4					1					1 – 1 = 0
5		1								1 – 1 = 0
6							1	1		2 – 1 = 1
7								1	1	2 – 1 = 1
8									1	1 – 1 = 0
9										

4 + 1 = 5

CHAPTER
MISCELLANEOUS QUESTIONS

Questions

1	**You are a software engineer of a big software development company that has developers who work remotely around the world. You have to analyze and develop a software system for the coordination of distributed development. Show database structures, different modules, flow chart and team structure to develop the software. [2004, 2006, 2007. Marks: 10]**
2	**You are a systems analyst of a multinational company that sells IT products and services with warranty of different time durations on different product and services. You have to research and develop a CRM (Customer Relationship Management) software for the company so that the company can properly track and give immediate warranty services to the clients. Design the database structure, different modules, flowcharts and a team structure to implement the system. Explain every decision you have made during this process. What software engineering model will you choose and why? [2003, 2005. Marks: 10]**
14.1	**What is transform flow? What are the required steps to map transform flow for the software design? [2003, 2004. Marks: 5]** *See chapter 14…*