# SOFTWARE ENGINEERING UNIT-1

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING.**

**Dr DEEPAK NEDUNURI**

**[ SIR C R REDDY COLLEGE OF ENGINEERING ]**

**ELURU.**

# SOFTWARE ENGINEERING
## UNIT-1

1. **The Nature of Software**
   1.1 Defining Software
   1.2 Software Application Domains
   1.3 Legacy Software
2. The Unique Nature of WebApps
3. Software Engineering
4. The Software Process
5. Software Engineering Practice
   5.1 The Essence of Practice
   5.2 General Principles
6. Software Myths
7. **A Generic Process Model**
   7.1 Defining a Framework Activity
   7.2 Identifying a Task Set
   7.3 Process Patterns
8. Process Assessment and Improvement
9. Prescriptive Process Models 9.1The Waterfall Model 9.2Incremental Process Models 9.3Evolutionary Process Models 9.4Concurrent Models
   9.5A Final Word on Evolutionary Processes
10. Specialized Process Models
    10.1 Component-Based Development
    10.2 The Formal Methods Model
    10.3 Aspect-Oriented Software Development
11. The Unified Process
    11.1 A Brief History
    11.2 Phases of the Unified Process
12. Personal and Team Process Models
    12.1 Personal Software Process (PSP)
    12.2 Team Software Process (TSP)
13. Process Technology
14. Product and Process

## 1. The Nature of Software

Today, software takes on a dual role. *It is a **product**, and at the same time, the **vehicle** for delivering a product.*

***As a product***, it delivers the computing potential embodied by computer hardware or more broadly, *by a network of computers* that are accessible by local hardware. Whether it resides within a mobile phone or operates inside a mainframe computer, ***software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources***.

***As the vehicle*** used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information.* It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context.

*Software manages business information* to enhance competitiveness;
*Software* provides a gateway to worldwide information networks (e.g., the Internet).
*Software* provides the means for acquiring information in all of its forms.
*Software* role has undergone significant change over the last half-century.
*Software* industry has become a dominant factor industrialized world.

**Engineering Discipline:**

- Engineering is a disciplined approach with some organized steps in a managed way to construction, operation, and maintenance of software.

- Engineering of a product goes through a series of stages, i.e., planning, analysis and specification, design, construction, testing, documentation, and deployment.

- The disciplined approach may lead to better results.

- The general stages for engineering the software include feasibility study and preliminary investigation, requirement analysis and specification, design, coding, testing, deployment, operation, and maintenance.

**Software Crisis:**

- Software crisis, the symptoms of the problem of engineering the software, began to enforce the practitioners to look into more disciplined software engineering approaches for software development.
- The software industry has progressed from the desktop PC to network-based computing to service-oriented computing nowadays.
- The development of programs and software has become complex with increasing requirements of users, technological advancements, and computer awareness among people.
- *Software crisis symptoms*
  - complexity,
  - hardware versus software cost,
  - Lateness and costliness,
  - poor quality,
  - unmanageable nature,
  - immaturity,
  - lack of planning and management practices,
  - Change, maintenance and migration,
  - etc.

**What is Software Engineering?**

- *The solution to these software crises is to introduce systematic software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.*
- The systematic means the methodological and pragmatic way of development, operation and maintenance of software.
- Systematic development of software helps to understand problems and satisfy the client needs.

- Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

- Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.

- Operational software must be correct, efficient, understandable, and usable for work at the client site.

- IEEE defines

    - *The systematic approach to the development, operation, maintenance, and retirement of software.*

1.1 Defining Software

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

**Software characteristics**

- Software has logical properties rather than physical.

- Software is mobile to change.

- Software is produced in an engineering manner rather than in classical sense.

- Software becomes obsolete but does not wear out or die.

- Software has a certain operating environment, end user, and customer.

- Software development is a labor-intensive task

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.***

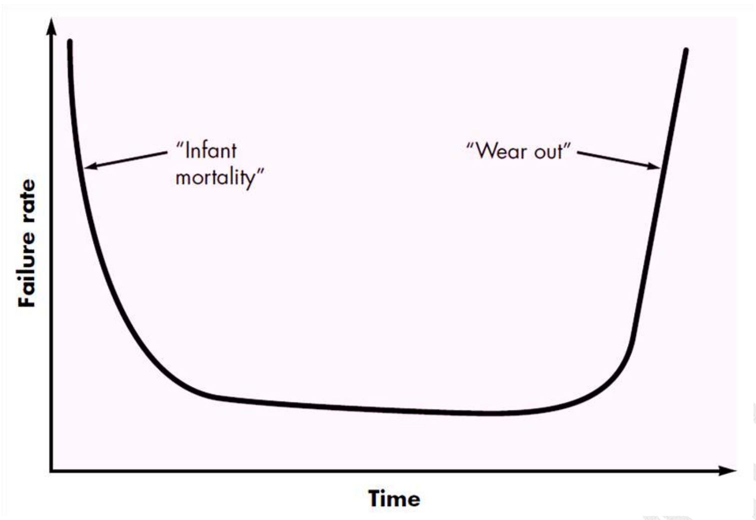
Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.

In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.

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

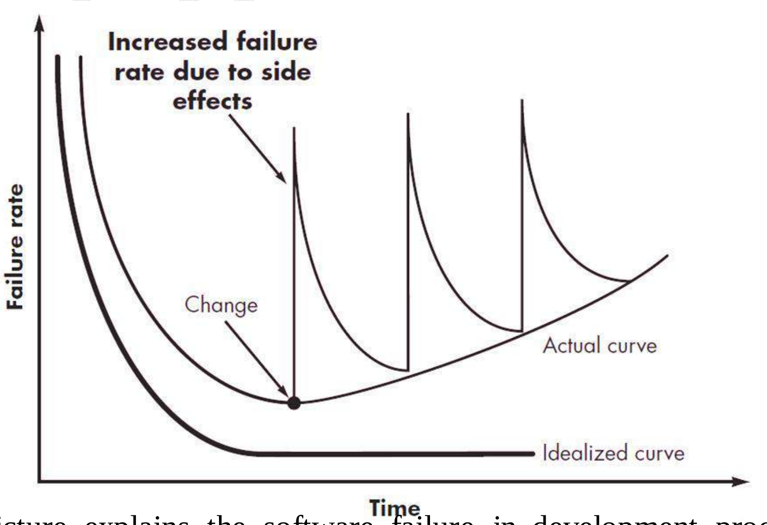
***2. Software doesn't "wear out."***



The above picture depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); ***defects are corrected*** and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.

As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

***Stated simply, the hardware begins to wear out.***
***Software doesn't "wear out." But***
***Software does deteriorate!***



The above picture explains the software failure in development process. During development process, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve". 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.***

DEPT OF CSE RGAN

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

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.

The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.

In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

## 1.2 Software Application Domains

**(Types of Software or Categories of Software)**

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

**i.** **System software**—a collection of programs written to service other programs.

***System software*** processes complex, but determinate information structures.
e.g., compilers, editors, and file management utilities

***Systems applications*** process largely indeterminate data.
(e.g., operating system components, drivers, networking software, telecommunications processors)

**ii.** **Application software**—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.
e.g., point-of-sale transaction processing, real-time manufacturing process control.

**iii.** **Engineering/scientific software**—is a special software to implement Engineering and Scientific applications. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

iv. **Embedded software**—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.

e.g., key pad control for a microwave oven.

Provide significant function and control capability

e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems.

v. **Product-line software**—designed to provide a specific capability for use by many different customers.
e.g., inventory control products, word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications.

vi. **Web applications**—called "WebApps," this network-centric software category span a wide array of applications. WebApps are linked with hypertext files. WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

vii. **Artificial intelligence software**— makes use of nonnumerical algorithms to solve complex problems of straightforward analysis.

Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

viii. **Open-world computing**—Software related to wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

ix. **Net Sourcing**—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

x. **Open source**—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development.

The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

## 1.3 <u>Legacy Software</u>

Older programs —often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

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

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' to the business." Hence, legacy software is characterized by longevity and business criticality.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.
Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

• The software must be adapted to meet the needs of new computing environments or technology.
• The software must be enhanced to implement new business requirements.
• The software must be extended to make it interoperable with other more modern systems or databases.
• The software must be re-architected to make it viable within a network environment.

When these modes of evolution occur, a legacy system must be reengineered, so that it remains useful in the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other".

## 2. <u>The Unique Nature of WebApps</u>

Web Apps means Web Applications. WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

Web engineers to provide computing capability along with informational content. *Web-based systems and applications* were born.

WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different.

The following attributes are encountered in the vast majority of WebApps.

i. **Network intensiveness.** A WebApp ***resides on a network and must serve the needs of a different types of clients***. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

ii. **Concurrency.** A large number of ***users may access the WebApp at one time***. In many cases, the patterns of usage among end users will vary greatly.

iii. **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One ***hundred users*** may show up on Monday; 10,000 may use the system on Thursday.

iv. **Performance.** WebApp should work effectively in terms of ***processing speed***. If a WebApp user must wait too long (for access, for serverside processing, for client-side formatting and display), he or she may decide to go elsewhere.

v. **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often ***demand access on a 24/7/365 basis***. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

vi. **Data driven.** The primary function of many WebApps is to use hypermedia to present ***text, graphics, audio, and video*** content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

vii. **Content sensitive.** The quality and aesthetic (beauty) ***nature of content*** remains an important determinant of the quality of a WebApp.

viii. **Continuous evolution. (Updated version) -**Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

ix. **Immediacy.** Although *immediacy*—the compelling need to get ***software to market quickly***—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.7

x. **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to ***protect sensitive content and provide secure modes of data transmission***, ***strong security measures must be implemented*** throughout the infrastructure that supports a WebApp and within the application itself.

xi. **Aesthetics.** An undeniable part of the appeal of a ***WebApp is its look and feel***. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

## 3. ***Software Engineering***

**What is Software Engineering?**
*software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.*

• The systematic means the methodological and pragmatic way of development, operation and maintenance of software.

• Systematic development of software helps to understand problems and satisfy the client needs.

• Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

• Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.

• Operational software must be correct, efficient, understandable, and usable for work at the client site.

• IEEE defines
*The systematic approach to the development, operation, maintenance, and retirement of software.*

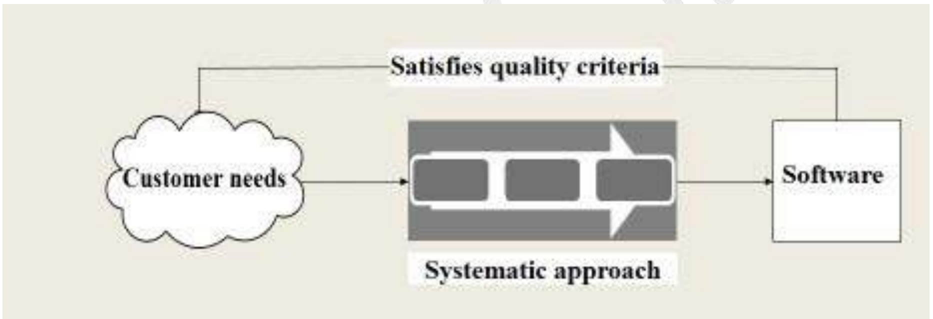


Figure 1.3: Software engineering view

Software Engineering -

• *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*

. *It follows that design becomes a unique activity.*

• *It follows that software should exhibit high quality.*

• *It follows that software should be maintainable, software in all of its forms and across all of its application domains should be engineered.*

## 4. <u>The Software Process</u>

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

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to ***deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation*** and those who will use it.

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

A generic process framework for software engineering encompasses five activities:

i. **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders11 The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
ii. **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
iii. **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
iv. **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
v. **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and

provides feedback based on the evaluation. These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

For many software projects, framework activities are applied iteratively as a project progresses. That is,

**communication,**
**planning,**
**modeling,**
**construction,** and
**deployment**

are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

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

Typical umbrella activities include:

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

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

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

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

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

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

**Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

**Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists. Each of these umbrella activities is discussed in detail later in this book.

The software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

• ***Overall flow of activities***, actions, and tasks and the interdependencies
among them
• **Degree to which actions and tasks** are defined within each framework activity
• **Degree to which work products are identified and required**

• Manner in which **quality assurance activities** are applied

• Manner in which **project tracking and control activities** are applied
• **Overall degree of detail and rigor** with which the process is described
• **Degree to which the customer and other stakeholders are involved with the project**
• **Level of autonomy given to the software team**
• **Degree to which team organization and roles are prescribed**

## 5. Software Engineering Practice

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

### 5.1 The Essence of Practice

George Polya outlined the essence of problem solving, and consequently, the essence of software engineering practice:
**1.** *Understand the problem* (communication and analysis).
**2.** *Plan a solution* (modeling and software design).
**3.** *Carry out the plan* (code generation).
**4.** *Examine the result for accuracy* (testing and quality assurance).
In the context of software engineering, these commonsense steps lead to a series of essential questions

i. **Understand the problem.** It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think. Understanding isn't always that easy. It's worth spending a little time answering a few simple questions:
• ***Who has a stake in the solution to the problem?*** That is, who are the stakeholders?
• ***What are the unknowns?*** What data, functions, and features are required to properly solve the problem?
• ***Can the problem be compartmentalized?*** Is it possible to represent smaller problems that may be easier to understand?
• ***Can the problem be represented graphically?*** Can an analysis model be created?

ii. **Plan the solution.** Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

• ***Have you seen similar problems before?*** Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
• ***Has a similar problem been solved?*** If so, are elements of the solution reusable?
• ***Can subproblems be defined?*** If so, are solutions readily apparent for the subproblems?
• ***Can you represent a solution in a manner that leads to effective implementation?*** Can a design model be created?

**iii.** **Carry out the plan.** The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

• ***Does the solution conform to the plan?*** Is source code traceable to the design model?

• ***Is each component part of the solution provably correct?*** Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

**iv.** **Examine the result.** You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

• ***Is it possible to test each component part of the solution?*** Has a reasonable testing strategy been implemented?

• ***Does the solution produce results that conform to the data, functions, and features that are required?*** Has the software been validated against all stakeholder requirements? It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

## 5.2 General Principles

David Hooker has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following.

**The First Principle:** ***The Reason It All Exists***

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

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

Software design is not a random process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system.

Features should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

**The Third Principle:** ***Maintain the Vision***

*A clear vision is essential to the success of a software project*. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

**The Fourth Principle:** ***What You Produce, Others Will Consume***

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So*, always specify, design, and implement knowing someone* *else will have to understand what you are doing*. The audience for any product of software development is potentially large.

Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

**The Fifth Principle:** ***Be Open to the Future***
A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*.
Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.14 This could very possibly lead to the reuse of an entire system. This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

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

**The Seventh principle:** ***Think!***
This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer.
When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous. If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer based systems would be eliminated.

## 6. Software Myths

- Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.
- Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often announced by experienced practitioners who "know the score."
- Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike.
- However, old attitudes and habits are difficult to modify, and remnants of software myths remain.
- Software Myths are three types
  1. Managers Myths
  2. Customers Myths (and other non-technical stakeholders)
  3. Practitioners Myths

i. **Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

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

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

# SOFTWARE ENGINEERING
# UNIT-1B

**7. A Generic Process Model**

7.1 Defining a Framework Activity

7.2 Identifying a Task Set

7.3 Process Patterns

**8.** Process Assessment and Improvement

**9.** Prescriptive Process Models

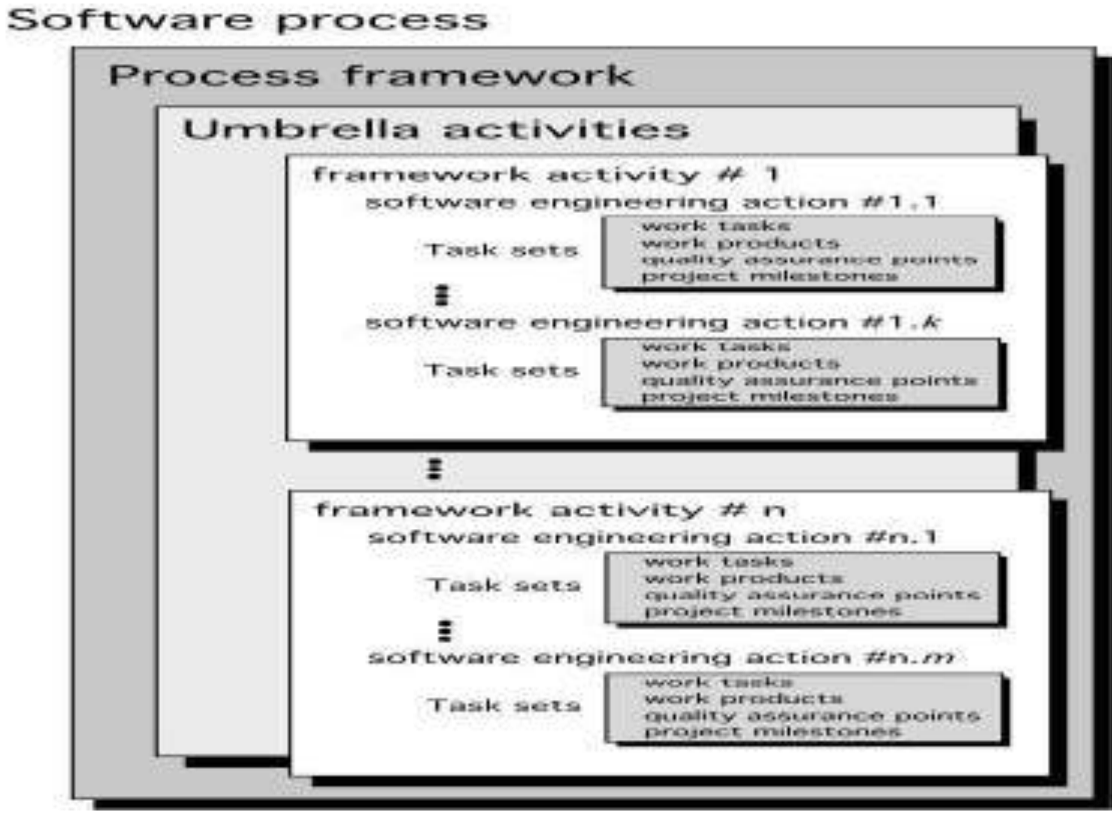
9.1 The Waterfall Model

9.2 Incremental Process Models

9.3 Evolutionary Process Models

9.4 Concurrent Models

9.5



**10.** Specialized Process Models

10.1 Component-Based Development

10.2 The Formal Methods Model

10.3 Aspect-Oriented Software Development

**11.** The Unified Process

11.1 A Brief History

11.2 Phases of the Unified Process

**12.** Personal and Team Process Models

12.1 Personal Software Process (PSP)

12.2 Team Software Process (TSP)

**13.** Process Technology

**14.** Product and Process

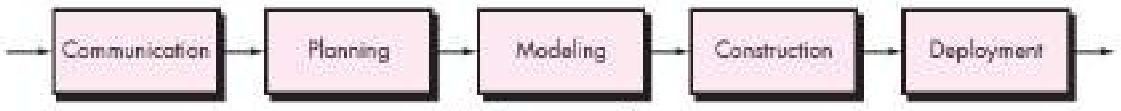
## 7. A GENERIC PROCESS MODEL

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

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

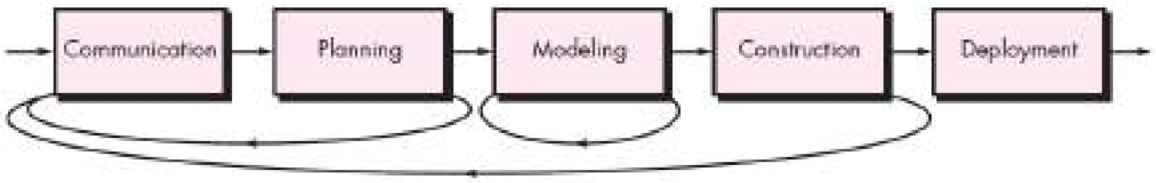
- A generic process framework for software engineering defines five framework activities
  - **communication,**
  - **planning,**
  - **modeling,**
  - **construction,** and
  - **deployment.**
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance,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

- The *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in the above Figure.
- A linear *process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment which is shown in the following Figure.



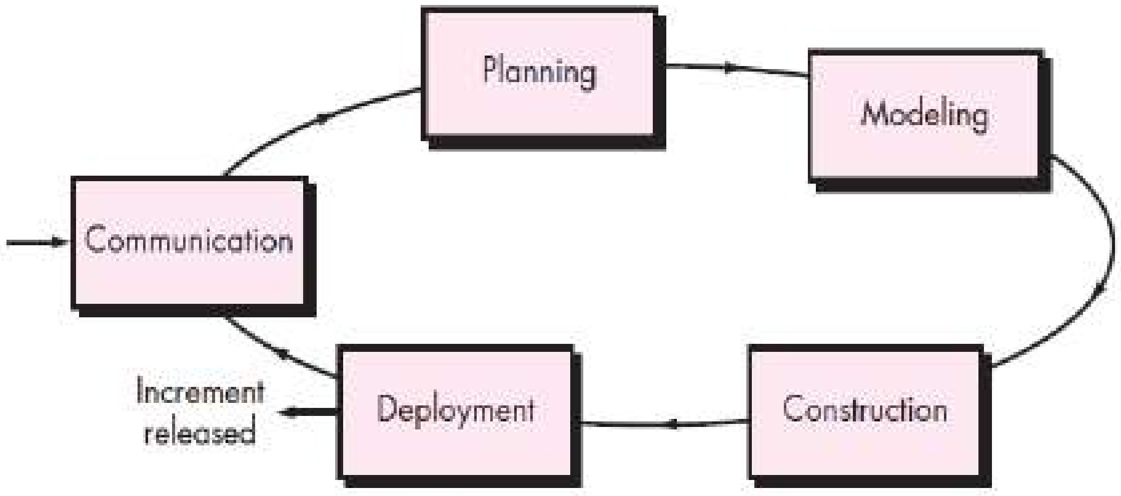
(a) Linear process flow

- An *iterative process flow* repeats one or more of the activities before proceeding to the next (shown in the following Figure).



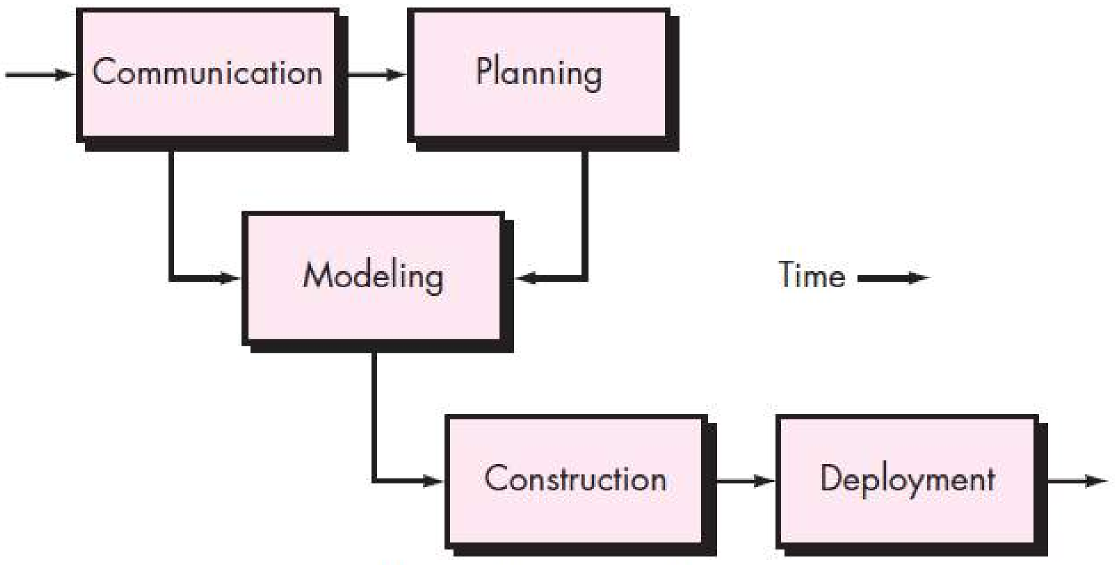
(b) Iterative process flow

- An *evolutionary process flow* executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software (shown in the following Figure).



(c) Evolutionary process flow

- A *parallel process flow* executes one or more activities in parallel with other activities
  e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software. (shown in the following figure)



(d) Parallel process flow

### 7.1.1 Defining a Framework Activity

There are five framework activities, they are

- **communication,**
- **planning,**
- **modeling,**
- **construction,** and
- **deployment.**

These five framework activities provide a basic definition of Software Process. These Framework activities provides basic information like *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

**1.** Make contact with stakeholder via telephone.
**2.** Discuss requirements and take notes.
**3.** Organize notes into a brief written statement of requirements.
**4.** E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of requirements, the communication activity might have six distinct actions: *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

### 7.1.2 Identifying a Task Set

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

### 7.1.3 Process Patterns

- Every software team encounters problems as it moves through the software process.
- It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.
- A *process pattern*1 describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a template a consistent method for describing problem solutions within the context of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Patterns can be defined at any level of abstraction.
- In some cases, a pattern might be used to describe a problem and solution associated with a complete process model (e.g., prototyping).

- In other situations, patterns can be used to describe a problem and solution associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).
- Ambler has proposed a template for describing a process pattern:

**Pattern Name.** The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

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

**Type.** The pattern type is specified. Ambler suggests three types:

**1.** ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity).

An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.

**2.** ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

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

**Initial context.** Describes the conditions under which the pattern applies. Prior to the initiation of the pattern:

(1) What organizational or team-related activities have already occurred?
(2) What is the entry state for the process?
(3) What software engineering information or project information already exists?

For example, the **Planning** pattern (a stage pattern) requires that

(1) customers and software engineers have established a collaborative communication;
(2) successful completion of a number of task patterns [specified] for the **Communication** pattern has occurred; and
(3) the project scope, basic business requirements, and project constraints are known.

**Problem.** The specific problem to be solved by the pattern.

**Solution.** Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern.

**Resulting Context.** Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?
(2) What is the exit state for the process?

(3) What software engineering information or project information has been developed?

**Related Patterns.** Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns:
**Project Team, Collaborative Guidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription,** and **ScenarioCreation.**

**Known Uses and Examples.** Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Conclusion on Process Patterns

- Process patterns provide an effective mechanism for addressing problems associated with any software process.
- The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).
- The description is then refined into a set of stage patterns that describe framework activities
- Once process patterns have been developed, they can be reused for the definition of process variants..

## 8. PROCESS ASSESSMENT AND IMPROVEMENT

- The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs.
- Process patterns must be coupled with solid software engineering practice
- In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

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

1. **Standard CMMI Assessment Method for Process Improvement (SCAMPI)**—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.
2. **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.
3. **SPICE (ISO/IEC15504)**—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.
4. **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

## 9. PRESCRIPTIVE PROCESS MODELS

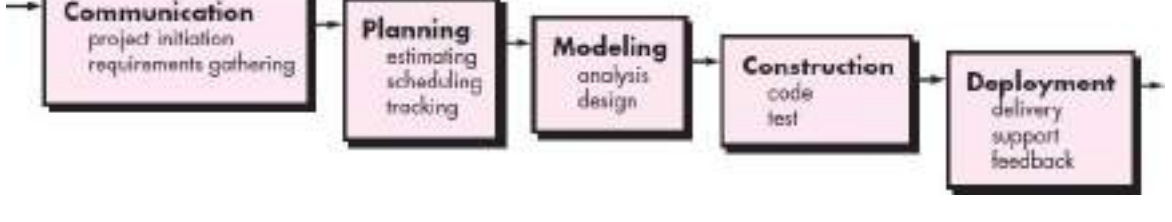
- Prescriptive process models were originally proposed to bring order to the chaos (disorder) of software development. History
- these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.
- The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise".
- The edge of chaos can be visualized as an unstable, partially structured state.
- It is unstable because it is constantly attracted to chaos or to absolute order.
- The prescriptive process approach in which order and project consistency are dominant issues.
- "prescriptive" means prescribe a set of process elements
  - framework activities,
  - software engineering actions,
  - tasks,
  - work products,
  - quality assurance, and
  - change control mechanisms for each project.
- Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.
- All software process models can accommodate the generic framework activities, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity in a different manner.

### 9.1.1 The Waterfall Model

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

**Context:** Used when requirements are reasonably well understood.

**Advantage:** It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



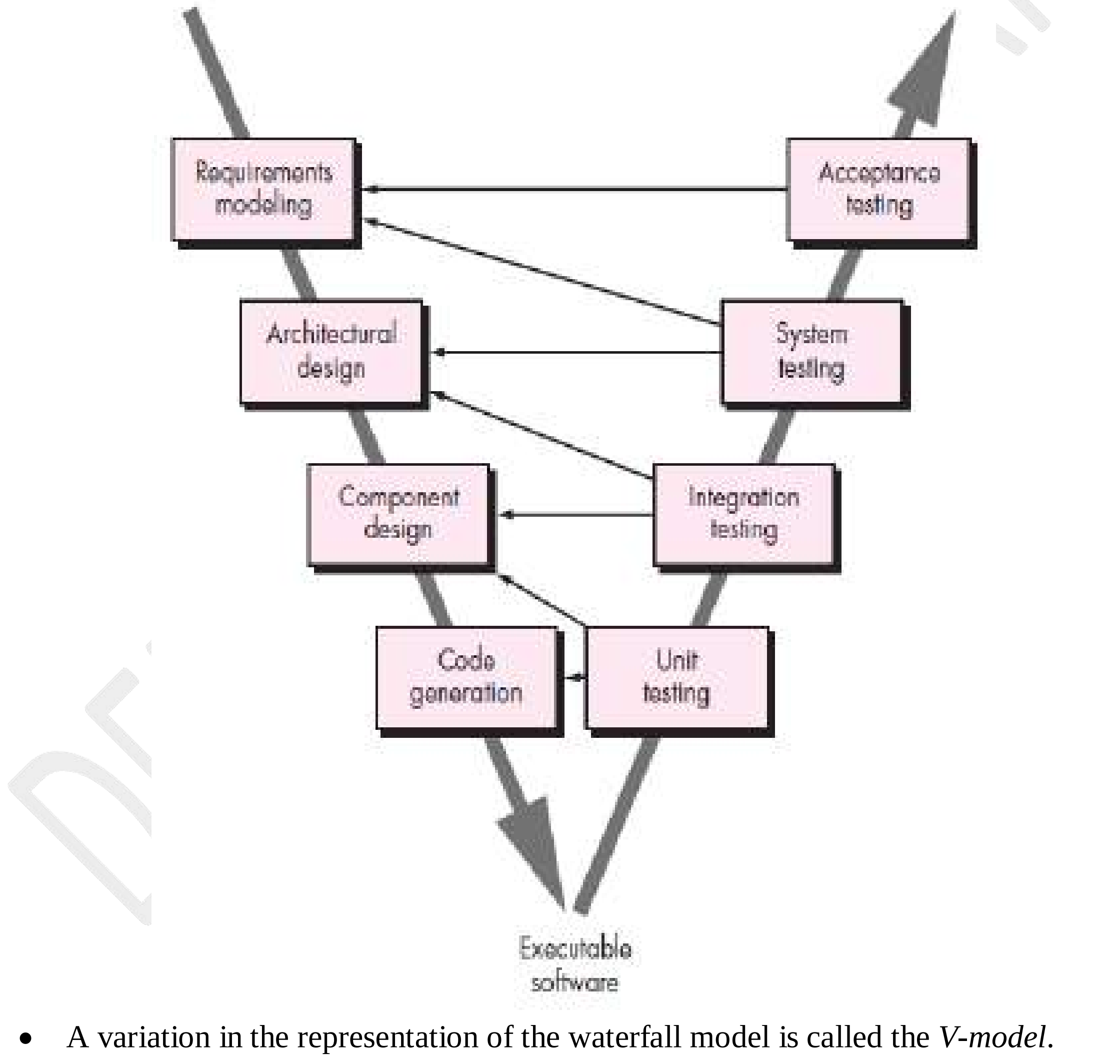
The **problems** that are sometimes encountered when the ***waterfall model*** is applied are:

i. 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.

ii. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.

iii. The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

## 9.1.2 V-odel



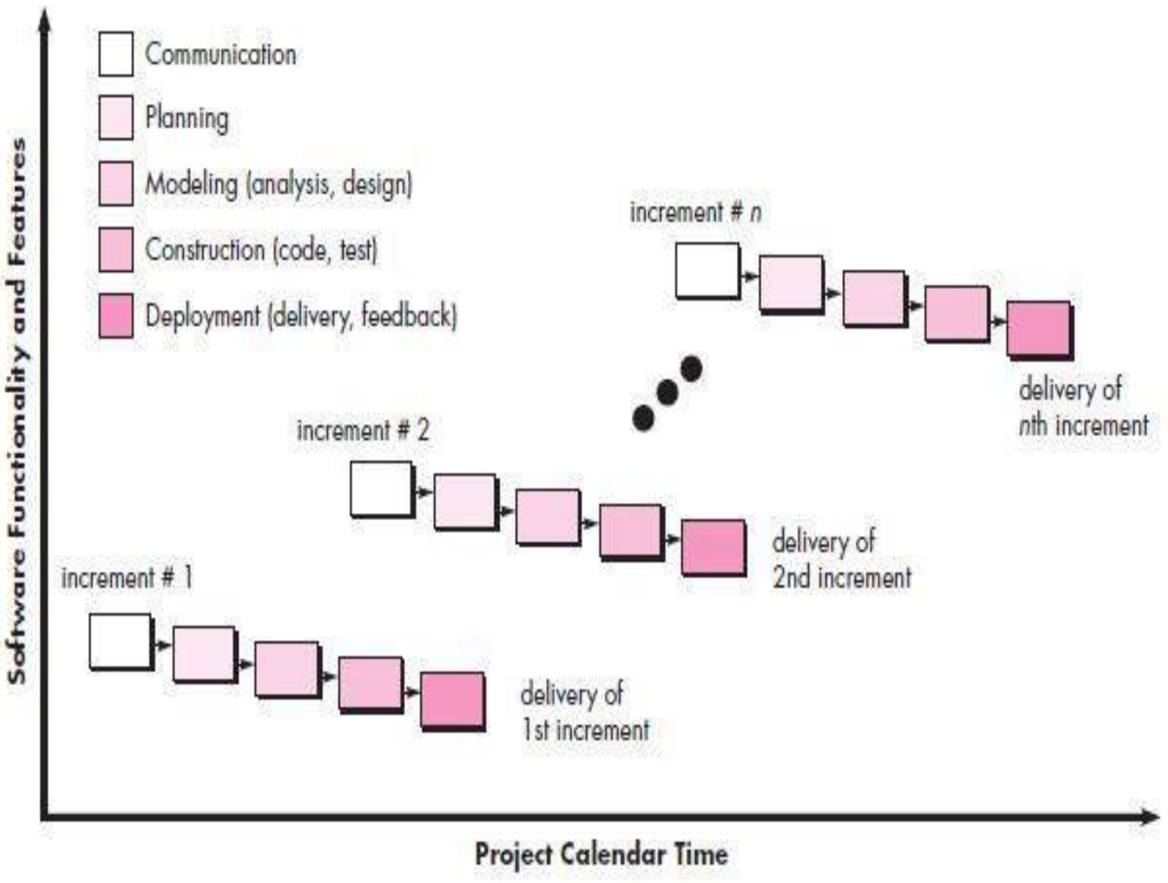
- A variation in the representation of the waterfall model is called the *V-model*.

Represented in the above Figure.

- The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.
- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

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

## 9.2 Incremental Process Models



Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
Software Functionality and Features
Project Calendar Time

- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort difficulty to implement linear process.
- Need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.
- In such cases, you can choose a process model that is designed to produce the software in increments.
- The *incremental* model combines elements of linear and parallel process flows. The above Figure shows the incremental model which applies linear sequences
- Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.
- For example, MS-Word software developed using the incremental paradigm might deliver
    - basic file management, editing, and document production functions in the first increment;

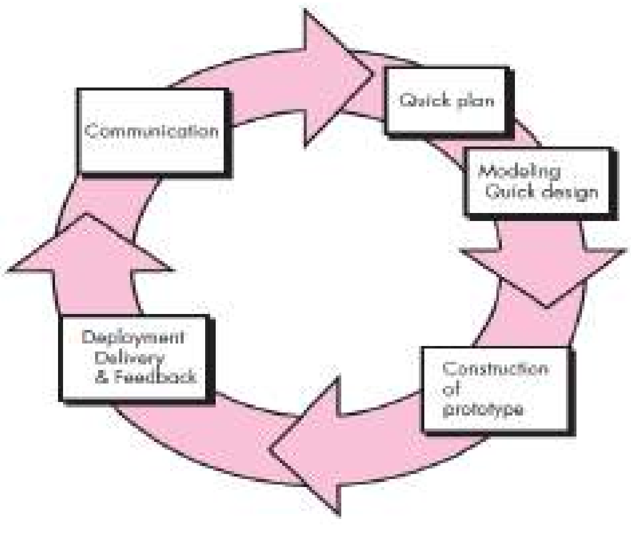
    - more sophisticated editing and document production capabilities in the second increment;
    - spelling and grammar checking in the third increment; and
    - Advanced page layout capability in the fourth increment.
    - It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model focuses on the delivery of an operational product with each increment.
- Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.
- Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

## **9.3 Evolutionary Process Models**

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

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

### **9.3.1 Prototyping.**



-  Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.
-  The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
-  Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.
-  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.

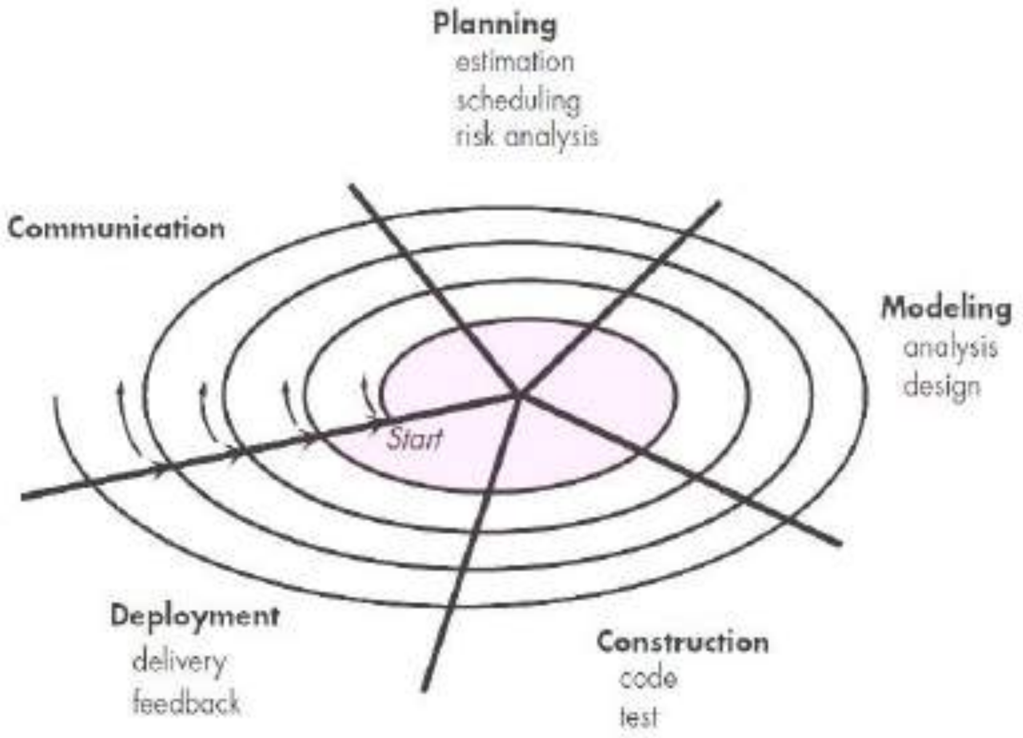
**Example:**

-  If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.
-  If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

**Advantages**:

-  The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.
-  The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.
-  Prototyping can be **problematic** for the following reasons:
-  The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability.
-  When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development relents.
-  The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

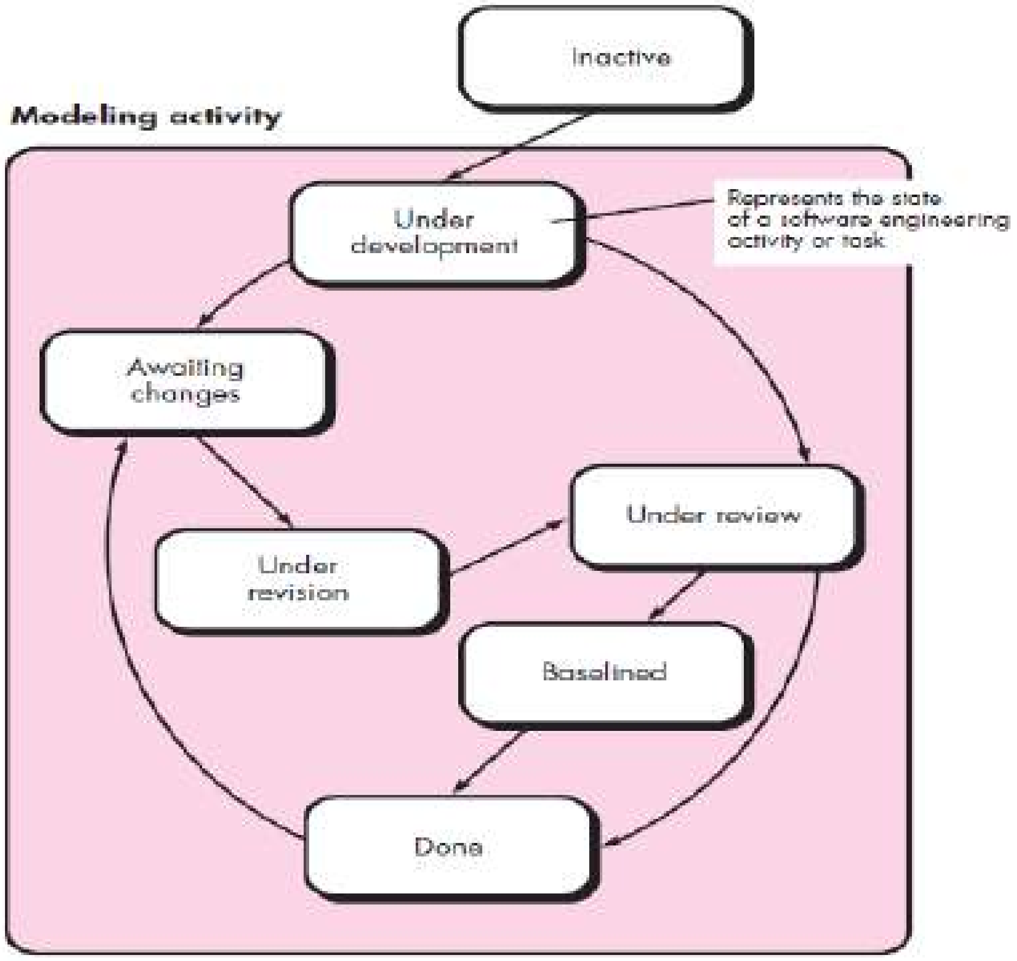
## 9.3.2 The Spiral Model.



-  The Spiral model is proposed by Barry Boehm.
-  The *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
-  It provides the potential for rapid development of increasingly more complete versions of the software.
-  Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are
-  **Anchor point milestones**- a combination of work products and conditions that are attained along the path of the spiral- are noted for each evolutionary pass.
-  The first circuit around the spiral might result in the development of product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
-  Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.
-  It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
-  The first circuit around the spiral might represent a “**concept development project**” which starts at the core of the spiral and continues for multiple iterations until concept development is complete.

 If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "**new product development project**" commences.
 Later, a circuit around the spiral might be used to represent a "**product enhancement project**." In essence, the spiral, when characterized in this way, remains operative until the software is retired.

## 9.4 Concurrent Models



- The *concurrent development model*, sometimes called *concurrent engineering*, allows a software team to represent iterative and concurrent elements of any of the process models.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.
- The above Figure provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach.
- The activity—**modeling**—may be in any one of the states - noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner.
- All software engineering activities exist concurrently but reside in different states.

- For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state.
- The modeling activity (which existed in the **inactive** state while initial communication was completed, now makes a transition into the **under development** state.
- If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.
- Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.
- For example, during early stages of design an inconsistency in the requirements model is uncovered. This generates the event *analysis model correction,* which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.
- Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project.
- Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

## 10. SPECIALIZED PROCESS MODELS

- Specialized process models take on many of the characteristics of one or more of the traditional models.

### 10.1 Component-Based Development

- Commercial off-the-shelf (COTS) **software components**, developed by vendors who offer them as **products**, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The *component-based development model* incorporates many of the characteristics of the **spiral model**.
- It is **evolutionary in nature**, demanding an iterative approach to the creation of software.
- However, the component-based development model **constructs applications from prepackaged software components.**
- Modeling and construction activities begin with the identification of **candidate components**.
- These **candidate components** can be designed as either conventional software modules or object-oriented classes or packages of classes.
- Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps:
  1. Available component-based products are researched and evaluated for the application domain in question.
  2. Component integration issues are considered.
  3. A software architecture is designed to accommodate the components.
  4. Components are integrated into the architecture.
  5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

## 10.2 The Formal Methods Model

- The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software.
- Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.
- A variation on this approach, called *clean room software engineering*, is currently applied by some software development organizations.
- When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms.
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through the review, but through the application of mathematical analysis.
- When formal methods are used during design, they serve as a basis for program verification and "discover and correct errors" that might otherwise go undetected.
- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

## 10.3 Aspect-Oriented Software Development

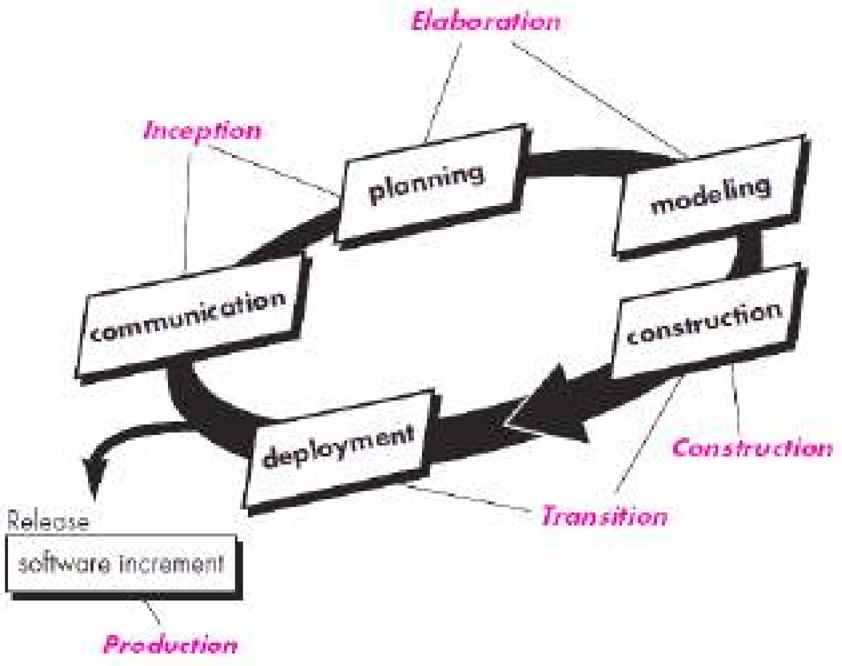
- Localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture.
- As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture.
- Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).
- When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns.*
- ***Aspectual requirements*** define those crosscutting concerns that have an impact across the software architecture.
- ***Aspect-oriented software development*** **(AOSD)**, often referred to as ***aspect-oriented programming*** **(AOP),** is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".
- ***aspect-oriented component engineering*** **(AOCE):** AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects".
- Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects);
  - event generation,
  - transport and receiving (distribution aspects);
  - data store/retrieve and indexing (persistency aspects);
  - authentication,
  - encoding and access rights (security aspects);

    - transaction atomicity,
    - concurrency control and logging strategy (transaction aspects); and so on.
- Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.
- The evolutionary model is appropriate as aspects are identified and then constructed.
- The parallel nature of concurrent development is essential because aspects are engineered independently.

## 11. THE UNIFIED PROCESS

- The unified process related to "use case driven, architecture-centric, iterative and incremental" software process.
- The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models.
- The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system.
- It emphasizes the important role of software architecture and "helps the architect focus on the right goals.
- It suggests a process flow that is iterative and incremental.
- During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a "unified method".
- The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems.
- By 1997, UML became a de facto industry standard for object-oriented software development.
- UML is used to represent both requirements and design models.
- UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework.
- Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process,* a framework for object-oriented software engineering using UML.
- Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds.
- The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

## 11.1 Phases of the Unified Process



- The above figure depicts the different phases in Unified Process.
- The ***inception phase*** of the UP encompasses both customer communication and planning activities.
  - By collaborating with stakeholders, business requirements for the software are identified;
  - a rough architecture for the system is proposed; and
  - a plan for the iterative, incremental nature of the ensuing project is developed.
  - Fundamental business requirements are described.
  - The architecture will be refined.
  - Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases.
- The **elaboration phase** encompasses the communication and modeling activities of the generic process model.
  - Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model.
  - In some cases, elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.
  - The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system.
  - In addition, the plan is carefully reviewed.
  - Modifications to the plan are often made at this time.
- The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.
  - The elaboration phase reflect the final version of the software increment.

    - All necessary and required features and functions for the software increment are then implemented in source code.
    - As components are being implemented, unit tests are designed and executed for each.
    - In addition, integration activities are conducted.
    - Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.
- The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.
    - Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
    - In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.
    - At the conclusion of the transition phase, the software increment becomes a usable software release.
- The ***production phase*** of the UP coincides with the deployment activity of the generic process.
    - During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.
    - It is likely that at the same time the construction, transition, and production phases are being conducted.
    - Work may have already begun on the next software increment.
    - This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

- A software engineering workflow is distributed across all UP phases.
- In the context of UP, a *workflow* is a task set
- That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.
- It should be noted that not every task identified for a UP workflow is conducted for every software project.
- The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

## 12. PERSONAL AND TEAM PROCESS MODELS

- Software process model has been developed at a corporate or organizational level.
- It can be effective only if it is helpful to significant adaptation to meet the needs of the project team that is actually doing software engineering work.
- In an ideal setting, it create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization.
- Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.
- It is possible to create a “personal software process” and/or a “team software process.”
- Both require hard work, training, and coordination, but both are achievable.

## 12.1 Personal Software Process (PSP)

- Every developer uses some process to build computer software.
- The process may be temporary; may change on a daily basis; may not be efficient, effective, or even successful; but a “process” does exist.
- Personal process, an individual must move through four phases, each requiring training and careful instrumentation.
- The *Personal Software Process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.
- In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.
- The PSP model defines five framework activities:

  **1. Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

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

  **3. High-level design review.** Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

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

  **5. Postmortem.** Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.
- PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.
- PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [Hum96], the resulting improvement in software engineering productivity and software quality are significant [Fer97].
- However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach.
- PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high.
- The required level of measurement is culturally difficult for many software people. Can PSP be used as an effective software process at a personal level? The answer is an unequivocal “yes.” But even if PSP is not adopted in its entirely, many of the personal process improvement concepts that it introduces are well worth learning.

### 2.6.2 Team Software Process (TSP)

- *Team Software Process* (TSP) build a “self-directed” project team that organizes itself to produce high-quality software.
- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.
- A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality);
- TSP identifies a team process that is appropriate for the project and a strategy for implementing the process;
- TSP defines local standards that are applicable to the team’s software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.
- TSP defines the following framework activities:
  - **project launch,**
  - **high-level design,**
  - **implementation,**
  - **integration and test,** and
  - **postmortem.**
- These activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product.
- The postmortem sets the stage for process improvements.
- TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.
- TSP recognizes that the best software teams are self-directed.
- Team members set
  - project objectives,
  - adapt the process to meet their needs,
  - control the project schedule, and
  - analysis of the metrics collected,
  - work continually to improve the team’s approach to software engineering.
- Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality.
- The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

## 13. PROCESS TECHNOLOGY

- ***Process technology tools*** have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality.
- ***Process technology tools*** allow a software organization to build an automated model of the process framework, task sets, and umbrella activities.
- The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.
- Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model.
- Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted.
- The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

## 2.8 PRODUCT AND PROCESS

- If the process is weak, the end product will undoubtedly suffer.
- But an obsessive overreliance on process is also dangerous.
- Product is final developed software
- Process is set of activities, actions and tasks to develop product.
- Structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute’s Software Development Capability Maturity Model (process).
- These swings are harmful in and of themselves because they confuse the average software practitioner.
- So software analyst must concentrate on Product by streamline the Process.
- All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result (Product) in a representation.
- Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity.
- The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

# SOFTWARE ENGINEERING UNIT-2

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING.

Dr DEEPAK NEDUNURI

[ SIR C R REDDY COLLEGE OF ENGINEERING ]

ELURU.

# SOFTWARE ENGINEERING
# UNIT-2

1. Requirements Analysis And Specification:
2. Requirements Gathering and Analysis.
3. Software Requirement Specification (SRS).
4. Formal System Specification.
5. Software Design: Overview of the Design Process.
6. How to Characterize a Design?
7. Cohesion and Coupling.
8. Layered Arrangement of Modules.
9. Approaches to Software Design.

# 1. REQUIREMENTS ANALYSIS AND SPECIFICATION

- The requirements analysis and specification to be a very important phase of software development life cycle and undertake it with utmost care.
- Before starting to develop a software, the exact requirements of the customer must be understood and documented.
- In the past, many projects have suffered because the developers started to implement something without determining whether they were building what the customers exactly wanted.
- Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the life cycle phases.
- Experienced developers take considerable time to understand the exact requirements of the customer.
- Without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution.
- For any type of software development project, availability of a good quality requirements document has been acknowledged to be a key factor in the successful completion of the project.
- A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases.
- When software is developed in a contract mode, the crucial role played by documentation of the requirements.

## An overview of requirements analysis and specification phase

- The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.
- The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed.
- The requirements specification document is usually called as the *software requirements specification* (SRS) document.
- The goal of the requirements analysis and specification phase is
  - to clearly understand the customer requirements and
  - to systematically organize the requirements into a document called the Software Requirements Specification (SRS) document
    .

## Who carries out requirements analysis and specification?

- Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site.
- The engineers who gather and analyse customer requirements and then write the requirements specification document are known as *system analysts* in the software industry parlance.
- System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done.
- After understanding the precise user requirements, the analysts analyse the requirements to remoe inconsistencies, anomalies and incompleteness.

- They then proceed to write the *software requirements specification* (SRS) document. The SRS document is the final outcome of the requirements analysis and specification phase.

**How is the SRS document validated?**

- Once the SRS document is ready, it is first reviewed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete.
- The SRS document is then given to the customer for review.
- After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organization.

**What are the main activities carried out during requirements analysis and specification phase?**

- Requirements analysis and specification phase mainly involves carrying out the following two important activities:
    - Requirements gathering and analysis
    - Requirements specification

## 2. REQUIREMENTS GATHERING AND ANALYSIS

- The complete set of requirements are almost never available in the form of a single document from the customer.
- The complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be gathered by the analyst from several sources in bits and pieces.
- These gathered requirements need to be analyzed to remove several types of problems.
- Requirements gathering and analysis activity divided into two separate tasks:
    - Requirements gathering
    - Requirements analysis

### 2.1 Requirements Gathering

- Requirements gathering is also popularly known as *requirements elicitation*.
- The primary objective of the requirements gathering task is to collect the requirements from the *stakeholders*.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.
- Requirements gathering may sound like a simple task.
- Gathering requirements very difficult when no working model of the software is available .
- Availability of a working model is usually of great help in requirements gathering.
- Typically even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed.
- During visit to the customer site, the analysts normally interview the end-users and customer representatives, carry out requirements gathering activities such as
    - questionnaire surveys,
    - task analysis,
    - scenario analysis, and
    - Form analysis.

- Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete.
- In the following, the important ways in which an experienced analyst gathers requirements:

**1. Studying existing documentation:**

- The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site.
- Customers usually provide statement of purpose (SoP) document to the developers.
- Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.

**2. Interview:**

- Typically, there are many different categories of users of a software.
- Each category of users typically requires a different set of features from the software.
- Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each.
- For example, the different categories of users of a library automation software could be
  - the library members,
  - the librarians, and
  - the accountants.
- **The library members** would like to use the software to query availability of books and issue and return books.
- **The librarians** might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc.
- **The accounts** personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.
- To systematise this method of requirements gathering, the Delphi technique can be followed.
- In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users.
- Based on their feedback, he refines his document.
- This procedure is repeated till the different users agree on the set of requirements.

**3. Task analysis:**

- The users usually have a black-box view of software and consider the software as something that provides a set of services.
- A service supported by software is also called a *task.*
- The software performs various tasks of the users.
- The analyst tries to identify and understand the different tasks to be performed by the software.
- For each task, the analyst tries to formulate the different steps necessary to realize the required functionality in consultation with the users.
- For example, for the issue book service, the steps may be—
  - authenticate user,
  - check the number of books issued to the customer and
  - determine if the maximum number of books that this member can borrow has been reached,

    - check whether the book has been reserved,
    - post the book issue details in the member's record, and
    - Finally print out a book issue slip that can be presented by the member at the security counter to take the book out of the library premises.
- Task analysis helps the analyst to understand the various user tasks and to represent each task as a hierarchy of subtasks.
- **Scenario analysis:** A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations.
- **Form analysis:** Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.

## 2.2 Requirements Analysis

- After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to find out any problems in the gathered requirements.
- It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness.
- Therefore, it is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.
- The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements.
- The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:
    - What is the problem?
    - Why is it important to solve the problem?
    - What exactly are the data input to the system and what exactly are the data output by the system?
    - What are the possible procedures that need to be followed to solve the problem?
    - What are the likely complexities that might arise while solving the problem?
    - If there are external software or hardware with which the developed software has to interface, then what should be the data interchange formats with the external systems?
- After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.
- During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:
  - Anomaly
  - Inconsistency
  - Incompleteness

Let us examine these different types of requirements problems in detail.

- **Anomaly:** It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible.
    - Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.

- **Incompleteness:** An incomplete set of requirements is one in which some requirements have been overlooked.
    - The lack of these features would be felt by the customer much later, possibly while using the software.
    - Often, incompleteness is caused by the inability of the customer to visualize the system that is to be developed and to anticipate all the features that would be required.
    - An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

## 3. SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

- After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organize the requirements in the form of an SRS document.
- The SRS document usually contains all the user requirements in a structured though an informal form.
- Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write.
- In the following, the different categories of users of an SRS document and their needs from it.

### 3.1 Users of SRS Document

- Some of the important categories of users of the SRS document and their needs for use are as follows:
- **Users, customers, and marketing personnel:** These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.
- **Software developers:** The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.
- **Test engineers:** The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.
- **User documentation writers:** The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.
- **Project managers:** The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.
- **Maintenance engineers:** The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system.
    - A clear knowledge of the functionalities can help them to understand the design and code.
    - Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.
    - Many software engineers in a project consider the SRS document to be a reference document.

## <u>3.2</u> <u>Uses of a well-formulated SRS document</u>

- **Forms an agreement between the customers and the developers:** A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.
- **Reduces future reworks:** The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.
- **Provides a basis for estimating costs and schedules:** Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The project manager also uses the SRS document for work scheduling.
- **Provides a baseline for validation and verification:** The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.
- **Facilitates future extensions:** The SRS document usually serves as a basis for planning future enhancements. Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document.

## <u>3.3</u> <u>Characteristics of a Good SRS Document</u>

- The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects.
- However, the analyst should be aware of the desirable qualities that every good SRS document should possess.
- IEEE Recommended Practice for Software Requirements Specifications describes the content and qualities of a good software requirements specification (SRS).

Some of the identified desirable qualities of an SRS document are the following:

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete.
- **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
    - It should only specify what the system should do and avoid that how to do these.
    - This means that the SRS document should specify the externally visible behavior of the system and not discuss the implementation issues.
    - This view with which a requirements specification is written, has been shown in Figure 4.1.
    - **Figure 4.1:** The black-box view of a system as performing a set of functions.

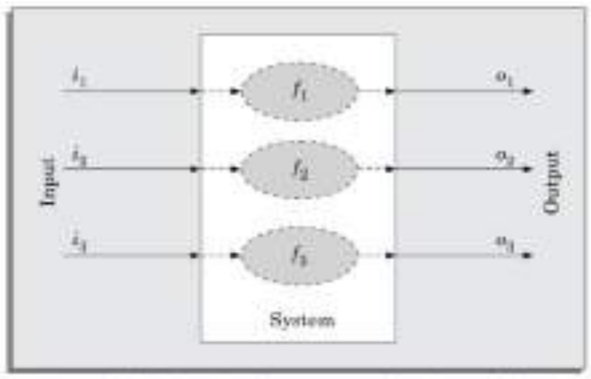


Figure 4.1: The black-box view of a system as performing a set of functions.

- **Traceable:** It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*.
    - Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*.
    - Traceability is also important to verify the results of a phase.
- **Modifiable:** Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development.
- **Identification of response to undesired events:** The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.
- **Verifiable:** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation.

## 3.4 Bad SRS Documents

- SRS documents written by novices frequently suffer from a variety of problems.

Some of the important categories of problems that many SRS documents suffer from are as follows:

- **Over-specification:** It occurs when the analyst tries to address the "how to" aspects in the SRS document.
- **Forward references:** One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.
- **Wishful thinking:** This type of problems concern description of aspects which would be difficult to implement.
- **Noise:** The term noise refers to presence of material not directly relevant to the software development process.

## 3.5 Categories of Customer Requirements

- A good SRS document, should properly categorize and organize the requirements into different sections.
- As per the IEEE 830 guidelines, the important categories of user requirements are the following. An SRS document should clearly document the following aspects of a software:
    - Functional requirements
    - Non-functional requirements—
        - Design and implementation constraints
        - External interfaces required
        - Other non-functional requirements

DEPT OF CSE RGAN

- Constraints
- Goals of implementation.

In the following, the different categories of requirements.

- **Functional requirements**
  - The functional requirements capture the functionalities required by the users from the system.
  - It is useful to consider a software as offering a set of functions {f*i*} to the user.
  - These functions can be considered similar to a mathematical function f : I → O, meaning that a function transforms an element (i*i*) in the input domain (I) to a value (o*i*) in the output (O).
  - This functional view of a system is shown schematically in Figure 4.1.

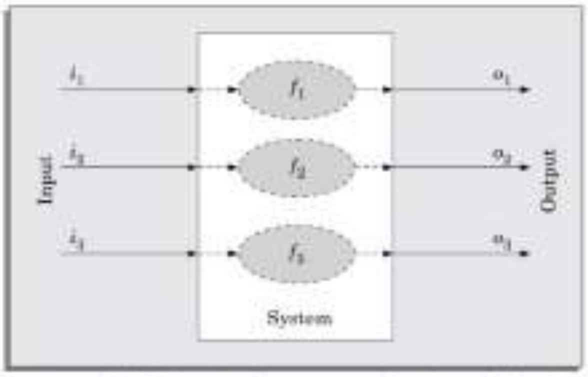


Figure 4.1: The black-box view of a system as performing a set of functions.

  - Each function f*i* of the system can be considered as reading certain data i*i*, and then transforming a set of input data (i*i*) to the corresponding set of output data (o*i*).
  - The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set.
  - Considering that the functional requirements are a crucial part of the SRS document,
- **Non-functional requirements**
  - The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data).
  - Non-functional requirements usually address aspects concerning
    - external interfaces,
    - user interfaces,
    - maintainability,
    - portability,
    - usability,
    - maximum number of concurrent users,
    - timing, and
    - Throughput (transactions per second, etc.).
  - The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum level in these requirements.
  - The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections.

DEPT OF CSE

- **Constraints:** Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers.
    - Some of the example constraints can be—
        - corporate or regulatory policies that needs to be honored;
        - hardware limitations;
        - interfaces with other applications;
        - specific technologies,
        - tools, and databases to be used;
        - specific communications protocols to be used;
        - security considerations;
        - design conventions or
        - Programming standards to be followed, etc.
        - Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organization.
- **External interfaces required:** Examples of external interfaces are—
    - hardware, software and communication interfaces,
    - user interfaces,
    - Report formats, etc.
    - To specify the user interfaces, each interface between the software and the users must be described.
    - The description may include sample screen images,
    - any GUI standards or style guides that are to be followed,
    - screen layout constraints,
    - standard buttons and functions (e.g., help) that will appear on every screen,
    - keyboard shortcuts,
    - Error message display standards, and so on.
    - One example of a user interface requirement of software can be that it should be usable by factory shop floor workers who may not even have a high school degree.
    - The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.
- **Other non-functional requirements:** This section contains a description of non-functional requirements that are neither design constraints and nor are external interface requirements.
    - An important example is a performance requirement such as the number of transactions completed per unit time.
    - The other non-functional requirements may include
        - reliability issues,
        - accuracy of results, and
        - Security issues.
- **Goals of implementation:** The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed.
    - These are not binding on the developers, and they may take these suggestions into account if possible.
    - For example, the developers may use these suggestions while choosing among different design solutions.

- A goal, in contrast to the functional and non-functional requirements, is not checked by the customer for conformance at the time of acceptance testing.
- The goals of implementation section might document issues such as
    - Easier revisions to the system functionalities.
    - easier support for new devices to be supported,
    - Reusability issues, etc.
- These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.
- It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and *vice versa*.

**Example -3.1 (Withdraw cash from ATM):**

- An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (SoP).
- An SoP serves as a starting point for the analyst and he proceeds with the requirements gathering activity after a basic understanding of the SoP.
- However, the functionalities of withdraw cash from ATM is intuitively obvious to anyone who has used a bank ATM.
- So, we are not including an informal description of withdraw cash functionality here and in the following, we documents this functional requirement.

***R.1: Withdraw cash (R.1 Means Requirement. one)***

*Description:* The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

***R.1.1 : Select withdraw amount optio*n**

*Input:* "Withdraw amount" option selected *Output:* User prompted to enter the account type

***R.1.2 : Select account type***

*I n p u t :* User selects option from any one of the followings— savings/checking/deposit.
*Output:* Prompt to enter amount

***R.1.3 : Get required amount***

*Input:* Amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.
***Output:*** The requested cash and printed transaction statement.
***Processing:*** The amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

## Example 3.2 (Search book availability in library):

- An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (SoP) based on which an later requirements gathering, the analyst understand the functionality.
- However, the functionalities of *search book availability* is intuitively obvious to any one who has used a library. So, we are not including an informal description of *search book availability* functionality here and in the following, we documents this functional requirement.

***R.1: Search book***
*Description* Once the user selects the search option, he would be asked to enter the keywords. The system would search the book in the book list based on the key words entered. After making the search, the system should output the details of all books whose title or author name match any of the key words entered. The book details to be displayed include: title, author name, publisher name, year of publication, ISBN number, catalog number, and the Location in the library.
***R.1.1 : Select search option***
*Input:* "Search" option
*Output:* User prompted to enter the key words
***R.1.2 : Search and display***
*Input:* Key words
*Output:* Details of all books whose title or author name matches any of the key words entered by the user. The book details displayed would include— title of the book, author name, ISBN number, catalog number, year of publication, number of copies available, and the location in the library.
*Processing:* Search the book list based on the key words:
***R.2: Renew book***
*Description:* When the "renew" option is selected, the user is asked to enter his membership number and password. After password validation, the list of the books borrowed by him is displayed. The user can renew any of his borrowed books by indicating them. A requested book cannot be renewed if it is reserved by another user. In this case, an error message would be displayed.
***R.2.1 : Select renew option***
*State:* The user has logged in and the main menu has been displayed.
*Input:* "Renew" option selection.
*Output:* Prompt message to the user to enter his membership number and password.
***R.2.2 : Login***
*State:* The renew option has been selected.
*Input:* Membership number and password.
*Output:* Li st of the books borrowed by the user is displayed, and user is prompted to select the books to be renewed, if the password is valid. If the password is invalid, the user is asked to re-enter the password.
*Processing:* Password validation, search the books issued to the user from the borrower's list and display.
*Next function:* R.2.3 if password is valid and R.2.2 if password is invalid.
***R.2.3 : Renew selected books***
*Input:* User choice for books to be renewed out of the books borrowed by him.
*Output:* Confirmation of the books successfully renewed and apology message for the books that could not be renewed.
*Processing:* Check if anyone has reserved any of the requested books. Renew the books selected by the user in the borrower's list, if no one has reserved those books. In order to properly identify the high-level requirements, a lot of common sense and the ability to visualize various scenarios that might arise in the operation of a function are required. Please note that when any of the aspects of a requirement, such as the state, processing description, next function to be executed, etc. are obvious, we have omitted it. We have to make a trade-off between cluttering the document with trivial details versus missing out some important descriptions.

## 3.6 Organization of the SRS Document

- **Introduction :** Introduction about the project
- **Purpose:** This section should describe where the software would be deployed and and how the software would be used.
- **Project scope:** This section should briefly describe the overall context within which the software is being developed.
- **Environmental characteristics:** This section should briefly outline the environment (hardware and other software) with which the software will interact.

### Overall description of organization of SRS document

- **Product perspective:** This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing systems, or it is a new software.
- **Product features:** This section should summarize the major ways in which the software would be used.
- **User classes:** Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities
- **Operating environment:** This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.
- **Design and implementation constraints:** In this section, the different constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.
- **User documentation:** This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.

### Functional requirements for organization of SRS document

- This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.
  1. User class 1
  (a) Functional requirement 1.1
  (b) Functional requirement 1.2
  2. User class 2
  (a) Functional requirement 2.1
  (b) Functional requirement 2.2

### External interface requirements

- **User interfaces:** This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides etc.
- **Hardware interfaces:** This section should describe the interface between the software and the hardware components of the system.
- **Software interfaces:** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries, and integrated commercial components, etc.

- **Communications interfaces:** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc.

### Other non-functional requirements for organization of SRS document

- **Performance requirements:** Some performance requirements may be specific to individual functional requirements or features.
- **Safety requirements:** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here.
- **Security requirements:** This section should specify any requirements regarding security or privacy requirements on data used or created by the software.

### Functional requirements

1. Operation mode 1
(a) Functional requirement 1.1
(b) Functional requirement 1.2
2. Operation mode 2
(a) Functional requirement 2.1
(b) Functional requirement 2.2

## 3.7 Representing Complex Logic

- A good SRS document should properly characterize the conditions.
- Sometimes the conditions can be complex and numerous and several alternative interaction and processing sequences may exist depending on the outcome of the corresponding condition checking.
- A simple text description in such cases can be difficult to analyze.
- In such situations, a decision tree or a decision table can be used to represent the logic and the processing involved.
- Also, when the decision making in a functional requirement has been represented as a decision table, it becomes easy to automatically or at least manually design test cases for it.
- There are two main techniques available to analyze and represent complex processing logic—decision trees and decision tables.
- Once the decision making logic is captured in the form of trees or tables, the test cases to validate these logic can be automatically obtained.

### Decision tree

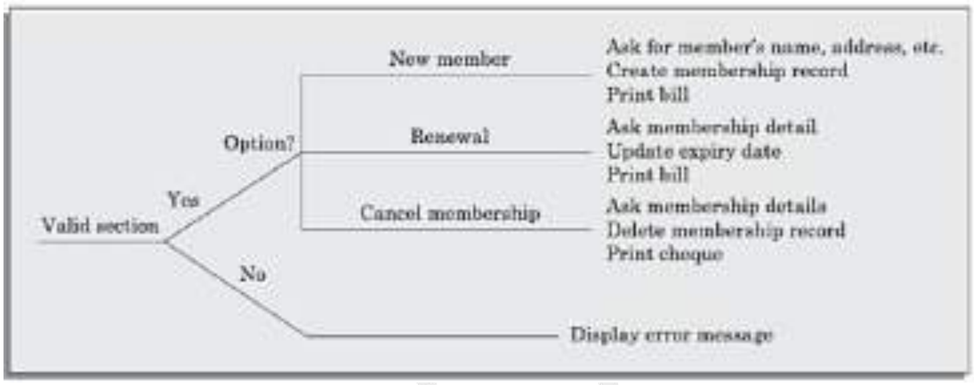
- A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken.
- The **edges** of a decision tree **represent conditions** and the **leaf nodes represent the actions** to be performed depending on the outcome of testing the conditions.

**Example: 3.3 A library membership management software (LMS)** should support the following three options—

new member,
renewal, and
cancel membership.

- When the *new member* option is selected, the software should ask the member’s name, address, and phone number.

- If proper information is entered, the software should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable.
- If the *renewal* option is chosen, the LMS should ask the member's name and his membership number and check whether he is a valid member.
- If the member details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership charge payable by the member should be printed.
- If the membership details entered are invalid, an error message should be displayed.
- If the *cancel membership* option is selected and the name of a valid member is entered, then the membership is cancelled, a choke for the balance amount due to the member is printed and his membership record is deleted.
- The decision tree representation for this problem is shown in the following Figure.

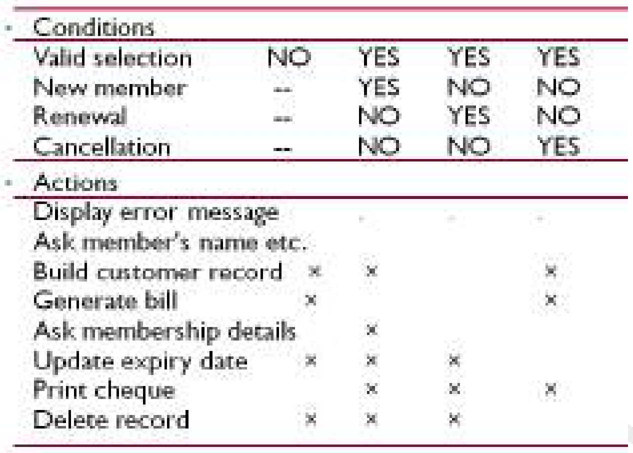


- Observe from above Figure that the internal nodes represent conditions, the edges of the tree correspond to the outcome of the corresponding conditions.
- The leaf nodes represent the actions to be performed by the system. In the decision tree of Figure, first the user selection is checked. Based on whether the selection is valid, either further condition checking is undertaken or an error message is displayed. Observe that the order of condition checking is explicitly represented.

## Decision table

- A decision table shows the decision making logic and the corresponding actions taken in a tabular or a matrix form.
- The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an evaluation test is satisfied.
- A column in the table is called a *rule*. A rule implies that if a certain condition combination is true, then the corresponding action is executed.

- The decision table for the LMS problem of Example- 3.3 is as shown in Table.



| Conditions | | | | |
|---|---|---|---|---|
| Valid selection | NO | YES | YES | YES |
| New member | -- | YES | NO | NO |
| Renewal | -- | NO | YES | NO |
| Cancellation | -- | NO | NO | YES |
| **Actions** | | | | |
| Display error message | | | | |
| Ask member's name etc. | | | | |
| Build customer record | × | × | | × |
| Generate bill | × | | | × |
| Ask membership details | | × | | |
| Update expiry date | × | × | × | |
| Print cheque | | × | × | × |
| Delete record | × | × | × | |

**Decision table *versus* decision tree**
Even though both decision tables and decision trees can be used to represent complex program logic, they can be distinguishable on the following three considerations:
**Readability:** Decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit.
**Explicit representation of the order of decision making:** In contrast to the decision trees, the order of decision making is abstracted out in decision tables. A situation where decision tree is more useful is when multilevel decision making is required. Decision trees can more intuitively represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.
**Representing complex decision logic:** Decision trees become very complex to understand when the number of conditions and actions increase. It may even be to draw the tree on a single page. When very large number of decisions are involved, the decision table representation may be preferred.

## 4. FORMAL SYSTEM SPECIFICATION

### 4.1 What is a Formal Technique?

- A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc.
- The mathematical basis of a formal method is provided by its specification language.
- In general, formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.
- **Syntactic domains:** The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.
- **Semantic domains:** Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronization trees, partial orders, state machines, etc.

DEPT OF CSE RGAN

- **Satisfaction relation:** Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as *semantic abstraction function*. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, t wo broad classes of semantic abstraction functions are defined— those that *preserve* a system's behaviour and those that *preserve* a system's structure.
- **Operational Semantics:** Informally, the *operational semantics* of a formal method is the way computations are represented. There are different types of operational semantics.
    - **Linear semantics:** In this approach, a *run* of a system is described by a sequence (possibly infinite) of events or states.
    - **Branching semantics:** In this approach, the behaviour of a system is represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state.
    - **Maximally parallel semantics:** In this approach, all the concurrent actions enabled at any state are assumed to be taken together.
    - **Partial order semantics:** Under this view, the semantics ascribed to a system is a *structure of states* satisfying a partial order relation among the states (events).
- **Algebraic specification:** In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages. Essentially, algebraic specifications define a system as a *heterogeneous algebra*. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations defined in this set; e.g. { I, +, -, *, / }.

## 5. OVERVIEW OF THE DESIGN PROCESS

- During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document.
- Main objectives of the design phase is transform the SRS document into the design document.
- This view of a design process has been shown schematically in Figure 5.1.

SRS document
Design process
Design document

**Figure 5.1:** The design process.

- As shown in Figure 5.1, the design process starts using the SRS document and completes with the production of the design document.
- The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.
- The design process essentially transforms the SRS document into a design document.

In the following sections, few important issues associated with the design process.

## 5.1 Outcome of the Design Process

The following items are designed and documented during the design phase.

- **Different modules required:**
  - The different modules in the solution should be clearly identified.
  - Each module is a collection of functions and the data shared by the functions of the module.
  - Each module should accomplish some well-defined task out of the overall responsibility of the software.
  - Each module should be named according to the task it performs.
- **Control relationships among modules:**
  - A control relationship between two modules essentially arises due to function calls across the two modules.
  - The control relationships existing among various modules should be identified in the design document.
- **Interfaces among different modules:**
  - The interface between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.
- **Data structures of the individual modules:**
  - Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module.
  - Suitable data structures for storing and managing the data of a module need to be properly designed and documented.
- **Algorithms required to implement the individual modules:**
  - Each function in a module usually performs some processing activity.
  - The algorithms required to accomplish the processing activities of various modules.
  - Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps.
  - The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

## 5.2 Classification of Design Activities

- A good software design is seldom realized by using a single step procedure, rather it requires iterating over a series of steps called the design activities.
- Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.
  - Preliminary (or high-level) design, and
  - Detailed design.
- The meaning and scope of these two stages can vary considerably from one design methodology to another.
- However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level design as follows:

- Through high-level design, a problem is decomposed into a set of modules.
- The control relationships among the modules are identified, and also the interfaces among various modules are identified.
- The outcome of high-level design is called the program structure or the software architecture.
- High-level design is a crucial step in the overall design of software.
- When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among them, and are arranged in a hierarchy.
- A notation that is widely being used for procedural development is a tree-like diagram called the structure chart.
- Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.
- Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design.

## 5.3 Classification of Design Methodologies

- The design activities vary considerably based on the specific design methodology being used.
- A large number of software design methodologies are available.
- These methodologies classified into procedural and object-oriented approaches.
- These two approaches are two fundamentally different design paradigms.
- **Do design techniques result in unique solutions?**
    - Even while using the same design methodology, different designers usually arrive at very different design solutions.
    - The reason is that a design technique often requires the designer to make many subjective decisions and work out compromises to contradictory objectives.
    - As a result, it is possible that even the same designer can work out many different solutions to the same problem.
    - Therefore, obtaining a good design would involve trying out several alternatives (or candidate solutions) and picking out the best one.
    - However, a fundamental question that arises at this point is—how to distinguish superior design solution from an inferior one?
    - Unless we know what a good software design is and how to distinguish a superior design solution from an inferior one, we cannot possibly design one. We investigate this issue in the next section.
- **Analysis versus design :**
    - Analysis and design activities differ in goal and scope.
    - The goal of any **analysis** technique is to elaborate the customer requirements through careful thinking and at the same time consciously avoiding making any decisions regarding the exact way the system is to be implemented.
    - The analysis results are generic and does not consider implementation or the issues associated with specific platforms.
    - The analysis model is usually documented using some graphical formalism.
    - In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart.

- On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modelling language (UML).
- The analysis model would normally be very difficult to implement using a programming language.
- **The design** model is obtained from the analysis model through transformations over a series of steps. In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented.
- The design model should be detailed enough to be easily implementable using a programming language.

## 6. HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

- Coming up with an accurate characterization of a good software design that would hold across diverse problem domains is certainly not easy.
- In fact, the definition of a "good" software design can vary depending on the exact application being designed.

These characteristics are listed below:

- **Correctness:** A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
- **Understandability:** A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
- **Efficiency:** A good design solution should adequately address resource, time, and cost optimization issues.
- **Maintainability:** A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

## 7. COHESION AND COUPLING

- We have so far discussed that effective problem decomposition is an important characteristic of a good design.
- Good module decomposition is indicated through **high cohesion** of the individual modules and **low coupling** of the modules with each other.

Let us now define what is meant by cohesion and coupling.

- **Cohesion** is a measure of the functional strength of a module.
  - Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution.
  - When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion.
  - If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion.
- **Coupling** is a measure of the degree of interaction (or interdependence) between the two modules.
  - Two modules are said to be highly coupled, if either of the following two situations arise:
    - If the function calls between two modules involve passing large volumes of shared data, the modules are **tightly coupled**.

- If the interactions occur through some shared data, then also we say that they are **highly coupled.**
- If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have **low coupling.**

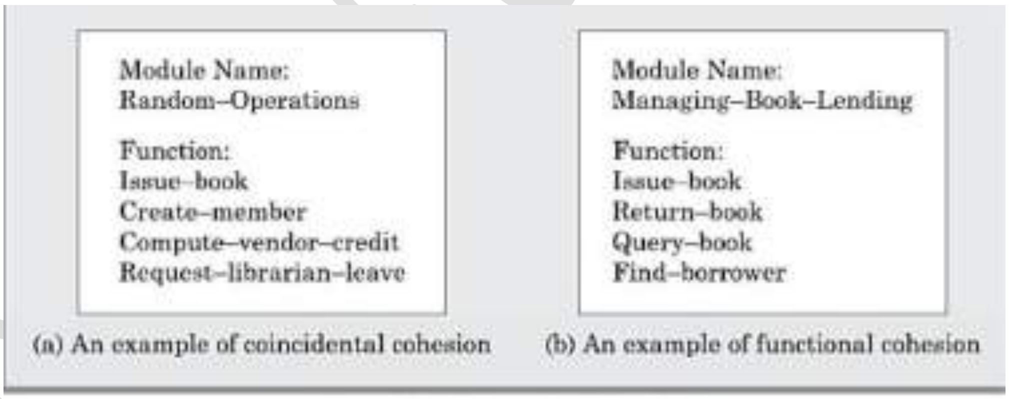
## 7.1 Classification of Cohesiveness

- Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective.
- The different modules of a design can possess different degrees of freedom.
- Different classes of cohesion that modules can possess are depicted in the following diagram.

| Coincidental | Logical | Temporal | Procedural | Communicational | Sequential | Functional |
|---|---|---|---|---|---|---|
| Low ---------------------------------------------------------------------------------------------→High | | | | | | |

- The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

- **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that **relate to each other very loosely**, if at all. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design.
  - An example of a module with coincidental cohesion has been shown in the following Figure.



Module Name:
Random–Operations

Function:
Issue–book
Create–member
Compute–vendor–credit
Request–librarian–leave

(a) An example of coincidental cohesion

Module Name:
Managing–Book–Lending

Function:
Issue–book
Return–book
Query–book
Find–borrower

(b) An example of functional cohesion

**Figure 5.4:** Examples of cohesion.

  - Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

- **Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform **similar operations**, such as error handling, data input, data output, etc.
  - As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

- **Temporal cohesion:** When a module contains functions that are related by the fact that these functions are executed in the **same time span**, then the module is said to possess temporal cohesion.
  - As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.
- **Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the **module are executed one after the other**, though these functions may work towards entirely different purposes and operate on very different data.
  - Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), printbill(), place-order-on-vendor(), update-inventory(), and l ogout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.
- **Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or **update the same data structure**.
  - As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.
- **Sequential cohesion:** A module is said to possess sequential cohesion, if the different functions of **the module execute in a sequence**, and the output from one function is input to the next in the sequence.
  - As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create-order(), check-item-availability(), placeorder- on-vendor() are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create-order() creates an order that is processed by the function check-item-availability() (whether the items are available in the required quantities in the inventory) is input to place-order-on-vendor().
- **Functional cohesion:** A module is said to possess functional cohesion, if different functions of **the module co-operate to complete a single task**.
  - For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion.
  - In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the payslips of the employees.

## 7.2 Classification of Coupling

- The coupling between two modules indicates **the degree of interdependence** between them.
- Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled.
- Let us now classify the different types of coupling that can exist between two modules.
- Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities have also been shown in Figure 5.5.

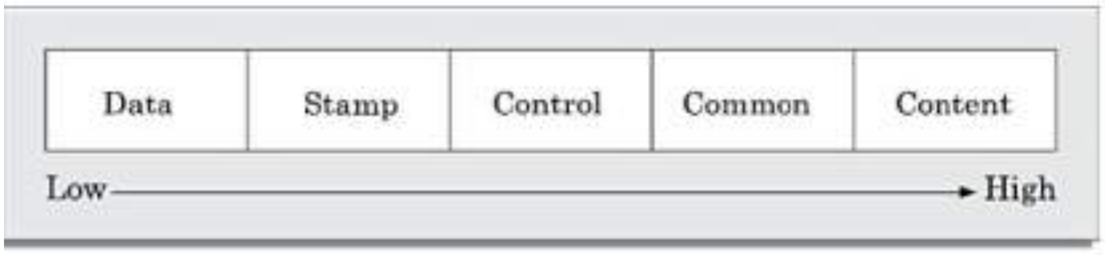


**Figure 5.5:** Classification of coupling.

- **Data coupling:** Two modules are data coupled, if they communicate using an **elementary data item that is passed as a parameter between the two**,
  - e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

- **Stamp coupling:** Two modules are stamp coupled, if they communicate using a **composite data** item such as a record in PASCAL or a structure in C.

- **Control coupling:** Control coupling exists between two modules, if data from one module is used to direct the order of **instruction execution** in another.
  - An example of control coupling is a flag set in one module and tested in another module.

- **Common coupling:** Two modules are common coupled, if they share some **global data** items.

- **Content coupling:** Content coupling exists between two modules, if they share **code**. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

- The different types of coupling are shown schematically in Figure 5.5. The degree of coupling increases from data coupling to content coupling. **High coupling among modules not only makes a design solution difficult to understand and maintain**, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

## 8. LAYERED ARRANGEMENT OF MODULES

- The control hierarchy represents the organization of program components in terms of their call relationships.
- Thus we can say that the control hierarchy of a design is determined by the order in which different modules call each other.
- Many different types of notations have been used to represent the control hierarchy.
- The most common notation is a tree-like diagram known as a structure chart
- However, other notations such as Warnier-Orr [1977, 1981] or Jackson diagrams [1975] may also be used.
- Since, Warnier-Orr and Jackson's notations are not widely used nowadays.
- In a layered design solution, the modules are arranged into several layers based on their call relationships.
- A module is allowed to call only the modules that are at a lower layer.

- That is, a module should not call a module that is either at a higher layer or even in the same layer.
- Figure 5.6(a) shows a layered design, whereas Figure 5.6(b) shows a design that is not layered.

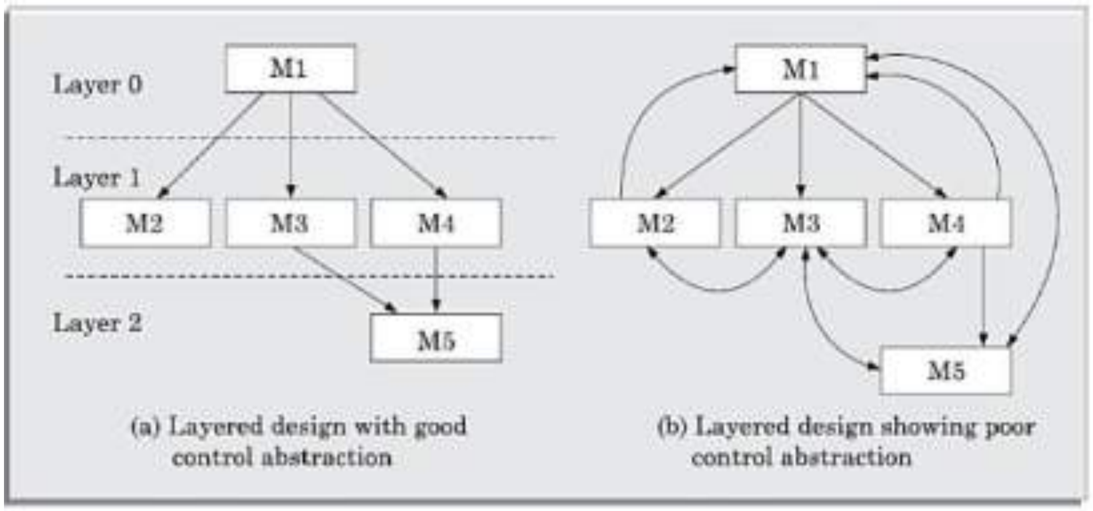

Layer 0
M1
Layer 1
M2
M3
M4
Layer 2
M5
(a) Layered design with good control abstraction
M1
M2
M3
M4
M5
(b) Layered design showing poor control abstraction

**Figure 5.6:** Examples of good and poor control abstraction.

- Observe that the design solution shown in Figure 5.6(b), is actually not layered since all the modules can be considered to be in the same layer.
- An important characteristic feature of a good design solution is layering of the modules.
- A layered design achieves control abstraction and is easier to understand and debug.
- In a layered design, the *top-most module in the hierarchy can be considered as a manager* that only invokes the services of the lower level module to discharge its responsibility.
- *The modules at the intermediate layers offer services to their higher layer by invoking the services* of the lower layer modules and also by doing some work themselves to a limited extent.
- The modules at the lowest layer are the **worker modules**. These do not invoke services of any module and entirely carry out their responsibilities by themselves.
- Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers (that is, the modules whose services it invokes).
- Besides, in a layered *design errors are isolated*, since an error in one module can affect only the higher layer modules. As a result, in case of any failure of a module, only the modules at the lower levels need to be investigated for the possible error.
- *Thus, debugging time reduces significantly in a layered design.*
- In the following, we discuss some important concepts and terminologies associated with a layered design:
- **Super-ordinate and sub-ordinate modules:** In a control hierarchy, a module that controls another module is said to be super-ordinate to it. Conversely, a module controlled by another module is said to be sub-ordinate to the controller.
- **Visibility:** A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.
- **Control abstraction:** In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules

at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

- **Depth and width:** Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.
- **Fan-out:** Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.
- **Fan-in:** Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

## 9. APPROACHES TO SOFTWARE DESIGN

- There are two fundamentally different approaches to software design that are in use today—
  - function-oriented design, and
  - object-oriented design.
- These two design approaches are different, they are complementary rather than competing techniques.
- The object oriented approach is a relatively newer technology and is still evolving.
- For development of large programs, the object- oriented approach is becoming increasingly popular due to certain advantages that it offers.
- Function-oriented designing is a mature technology and has a large following.

### 9.1 Function-oriented Design

The following are the salient features of the function-oriented design approach:

- **Top-down decomposition:** A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.
  - In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.
  - For example, consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge.
  - This high-level function may be refined into the following sub functions:
    - assign-membership-number
    - create-member-record
    - print-bill
  - Each of these sub-functions may be split into more detailed sub-functions and so on.
- **Centralized system state:** The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event.
  - For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system.
  - Such data in procedural programs usually have global scope and are shared by many modules.

- The system state is centralized and shared among different functions. For example, in the library management system, several functions such as the following share data such as member-records for reference and updating:
  - create-new-member
  - delete-member
  - update-member-record

A large number of function-oriented design approaches have been proposed in the past. A few of the well-established function-oriented design approaches are as following:

- Structured design by Constantine and Yourdon, [1979]
- Jackson's structured design by Jackson [1975]
- Warnier-Orr methodology [1977, 1981]
- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

## 9.2 Object-oriented Design

- In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities).
- Each object is associated with a set of functions that are called its methods.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object.
- The system state is decentralized since there is no globally shared data in the system and data is stored in each object.
  - For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data.
- The methods defined for one object cannot directly refer to or change the data of other objects.
- The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below.
- Objects decompose a system into functionally independent modules.
- Objects can also be considered as instances of abstract data types (ADTs).
- The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language.
- ADT is an important concept that forms an important pillar of object orientation.
- Let us now discuss the important concepts behind an ADT. There are, in fact, three important concepts associated with an ADT—
  - data abstraction,
  - data structure,
  - Data type.

  We discuss these in the following subsection:

- **Data abstraction:** The principle of data abstraction implies that how data is exactly stored is abstracted away.
  - This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organized, and manipulated inside the object.
  - The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object.

    - Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop.
- **Data structure:** A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organized collection of primitive data items such as integer, floating point numbers, characters, etc.
- **Data type:** A type is a programming language terminology that refers to anything that can be instantiated.
    - For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.
    - In object-orientation, classes are ADTs. But, what is the advantage of developing an application using ADTs?

# SOFTWARE ENGINEERING UNIT-3

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING.

Dr DEEPAK NEDUNURI

[ SIR C R REDDY COLLEGE OF ENGINEERING ]

ELURU.

# SOFTWARE ENGINEERING
# UNIT-3

FUNCTION-ORIENTED SOFTWARE DESIGN

- Function-oriented design techniques were proposed nearly four decades ago.
- These techniques are at the present time still very popular and are currently being used in many software development organizations.
- These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software.
- These services provided by a software (e.g., issue book, search book, etc.,) for a Library Automation Software to its users are also known as the high-level functions supported by the software.
- During the design process, these high-level functions are successively decomposed into more detailed functions.
- The term top-down decomposition is often used to denote the successive decomposition of a set of high-level functions into more detailed functions.
- After top-down decomposition has been carried out, the different identified functions are mapped to modules and a module structure is created.
- This module structure would possess all the characteristics of a good design identified in the last chapter.
- The SA/SD technique can b e used to perform the high-level design of a software. The details of SA/SD technique are discussed further.

## 1. OVERVIEW OF SA/SD METHODOLOGY

- As the name itself implies, SA/SD methodology involves carrying out two distinct activities:
  - Structured analysis (SA)
  - Structured design (SD)
- The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1.
-

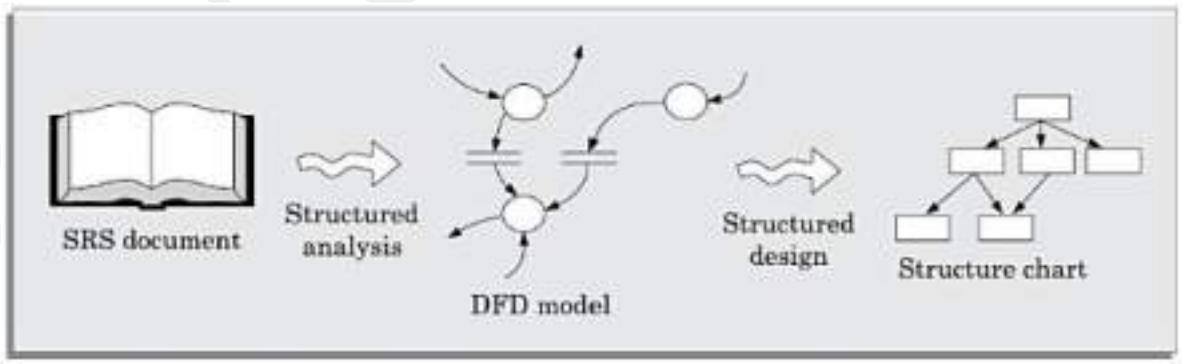


**Figure 6.1:** Structured analysis and structured design methodology.

- During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

- As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model.
- **During structured analysis**, functional decomposition of the system is achieved. That is, each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions.
- **During structured design**, all functions identified during structured analysis are mapped to a module structure.
- This **module structure** is also called the high-level design or the software architecture for the given problem.
- This is represented using a **structure chart**.
- The high-level design stage is normally followed by a detailed design stage.
- During the **detailed design** stage, the algorithms and data structures for the individual modules are designed.
- The detailed design can directly be implemented as a working system using a conventional programming language.
- It is important to understand that the purpose of structured analysis is to capture the detailed structure of the system as perceived by the user, whereas the purpose of structured design is to define the structure of the solution that is suitable for implementation in some programming language.
- The results of structured analysis can therefore, be easily understood by the user. In fact, the different functions and data in structured analysis are named using the user's terminology. The user can therefore even review the results of the structured analysis to ensure that it captures all his requirements.
- In the following section, we first discuss how to carry out structured analysis to construct the DFD model. Subsequently, we discuss how the DFD model can be transformed into structured design.

## 2 STRUCTURED ANALYSIS

- We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and the data flow among these processing tasks are represented graphically.
- The structured analysis technique is based on the following underlying principles:
- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions. Graphical representation of the analysis results using data flow diagrams (DFDs).
- DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
- A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.
- Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed.

## 3. Data Flow Diagrams (DFDs)

- The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.
- The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism— **it is simple to understand and use**.
- A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the **data flow among these functions**.

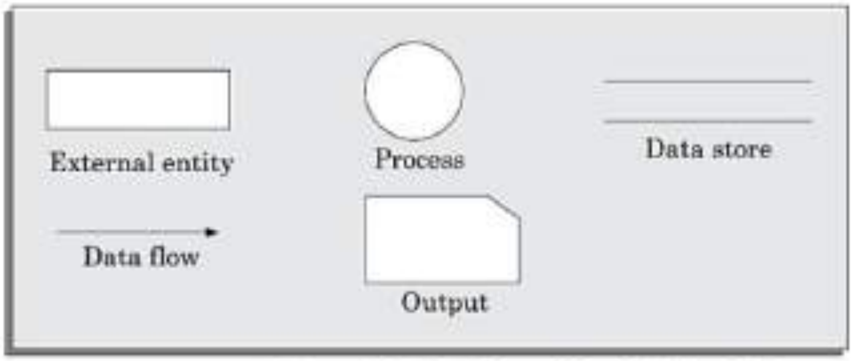


**Figure 6.2:** Symbols used for designing DFDs.

- Starting with a set of high-level functions that a system performs, a DFD model represents the sub-functions performed by the functions using a hierarchy of diagrams.
- Human mind is such that it can **easily understand any hierarchical model of a system**—because in a hierarchical model, starting with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy.
- The DFD technique is also based on a very simple set of intuitive concepts and rules.

**Primitive symbols used for constructing DFDs** : There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2.

- The meaning of these symbols are explained as follows: **Figure 6.2:** Symbols used for designing DFDs.
- **1. Function symbol:** A function is represented using a circle.

This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).

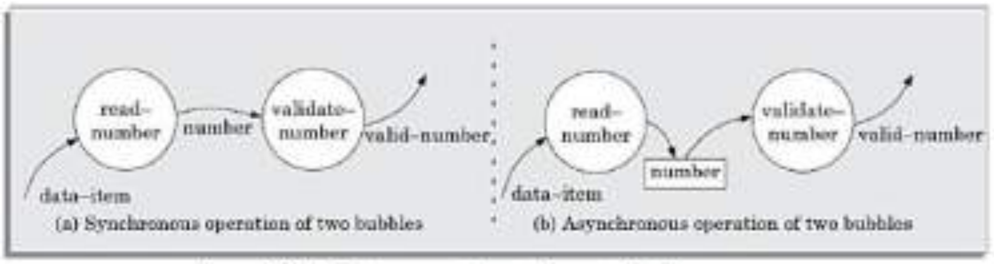
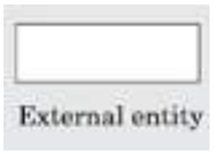


**Figure 6.3:** Synchronous and asynchronous data flow.

- **2. External entity symbol:** An external entity such as a librarian, a library member, etc. is represented by a rectangle.



The external entities are essentially those **physical entities external to the software system which interact with the system** by inputting data to the system or by consuming the data produced by the system.

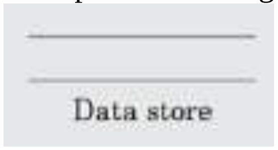
In addition to the human users, the external entity symbols can be used to **represent external hardware and software** such as another application software that would interact with the software being modeled.

- **3. Data flow symbol:** A directed arc (or an arrow) is used as a data flow symbol.

A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.

Data flow symbols are usually annotated with the corresponding data names. For example the DFD in Figure 6.3(a) shows three data flows—the data item number flowing from the process read-number to validate-number, data item flowing into read-number, and valid-number flowing out of validate-number.

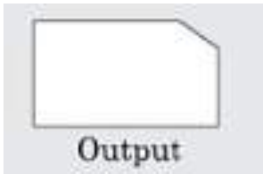
- **4. Data store symbol:** A data store is represented using two parallel lines.



It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol.
The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting t o a data store need not be annotated with the name of the corresponding data items. As an example of a data store, number is a data store in Figure 6.3(b).

DEPT OF CSE RGAN

- **5. Output symbol:** The output symbol is as shown in Figure 6.2. The output symbol is used when a hard copy is produced.



- **Important concepts associated with constructing DFD models** Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

## <u>Synchronous and asynchronous operations :</u>

- If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figure 6.3(a).

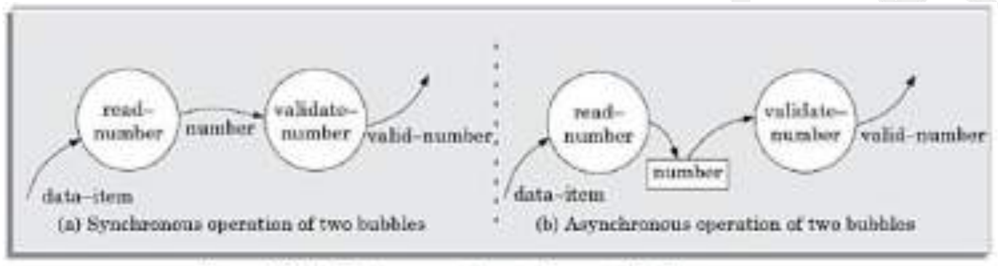


**Figure 6.3:** Synchronous and asynchronous data flow.

- Here, the validate-number bubble can start processing only after the read number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.
- However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

## <u>Data dictionary:</u>

- Every DFD model of a system must be accompanied by a data dictionary.
- A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model.
- The DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc., as shown in Figure 6.
- However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.
- For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.
  *grossP ay = regularP ay + overtimePay*

- For the smallest units of data items, the data dictionary simply lists their name and their type.
- Composite data items are expressed in terms of the component data items using certain operators.
- The operators using which a composite data item can be expressed in terms of its component data items.
- The dictionary plays a very important role in any software development process, especially for the following reasons:
- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project.
- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities.
- For large systems, the data dictionary can become extremely complex and voluminous.
- Moderate-sized projects can have thousands of entries in the data dictionary. It becomes extremely difficult to maintain a voluminous dictionary manually. Computer-aided software engineering (CASE) tools come handy to overcome this problem.
- Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary. As a result, the designers do not have to spend almost any effort in creating the data dictionary.
- These CASE tools also support some query language facility to query about the definition and usage of data items.
- For example, queries may be formulated to determine which data item affects which processes, or a process affects which data items, or the definition and usage of specific data items, etc. Query handling is facilitated by storing the data dictionary in a relational database management system (RDBMS).

**Data definition :**

- Composite data items can be defined in terms of primitive data items using the following data definition operators.

**+** : denotes composition of two data items, e.g. a+b represents data a and b.

**[,,]** : represents selection, i.e. any one of the data items listed inside the square bracket can occur For example, [a,b] represents either a occurs or b occurs.

**( )** : the contents inside the bracket represent optional data which may or may not appear. a+(b) represents either a or a+b occurs.

**{}** : represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.

**=** : represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.

**/* */** : Anything appearing within /* and */ is considered as comment.

## 3.1 DEVELOPING THE DFD MODEL OF A SYSTEM

- A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- The DFD model of a problem consists of many of DFDs and a single data dictionary.
- The DFD model of a system i s constructed by using a hierarchy of DFDs (see Figure 6.4).

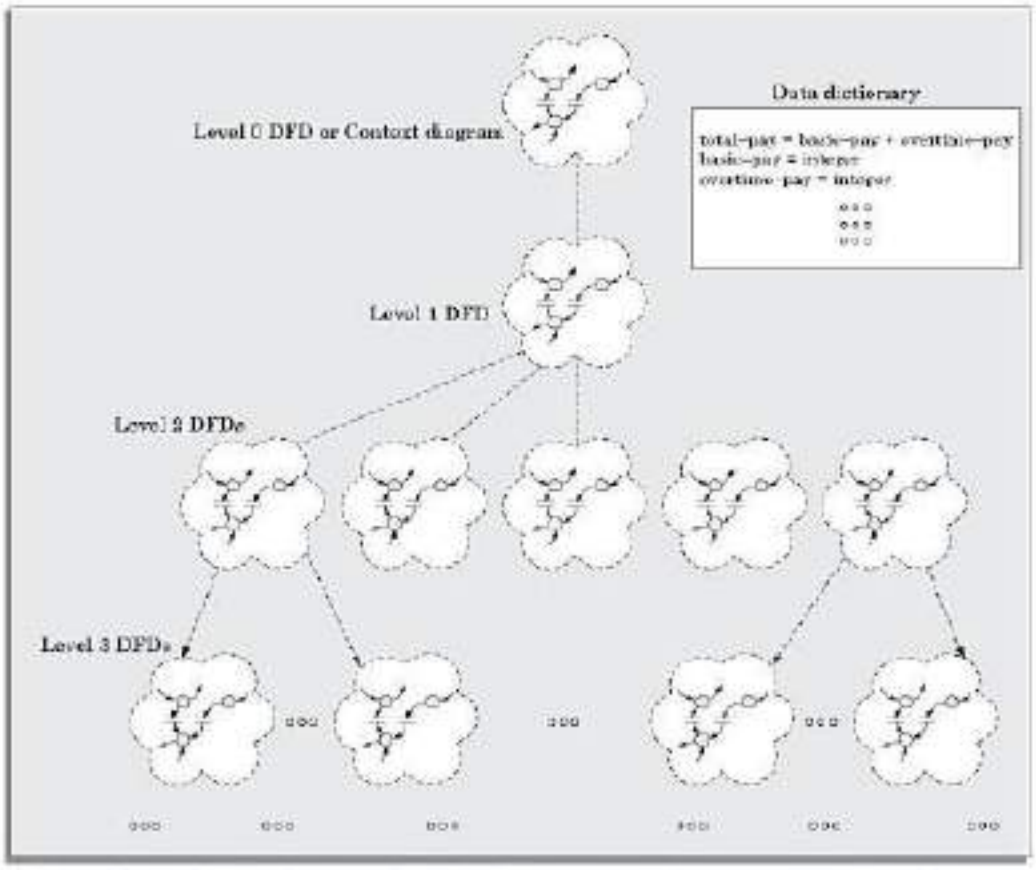


**Figure 6.4:** DFD model of a system consists of a hierarchy of DFDs and a single data dictionary

- The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand. At each successive lower level DFD s, more and more details are gradually introduced.
- To develop a higher-level DFD model, processes are decomposed into their sub-processes and the data flow among these sub-processes is identified.
- To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each.
- Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on.
- However, there is only a single data dictionary for the entire DFD model. All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

## Context Diagram :

- The context diagram is the most abstract (highest level) data flow representation of a system.
- It represents the entire system as a single bubble.
- The bubble in the context diagram is annotated with the name of the software system being developed. (usually a noun).
- This is the only bubble in a DFD model, where a noun is used for naming the bubble.
- The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble.
- This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality. As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure 6.10). The context diagram has been labelled as 'Supermarket software'.

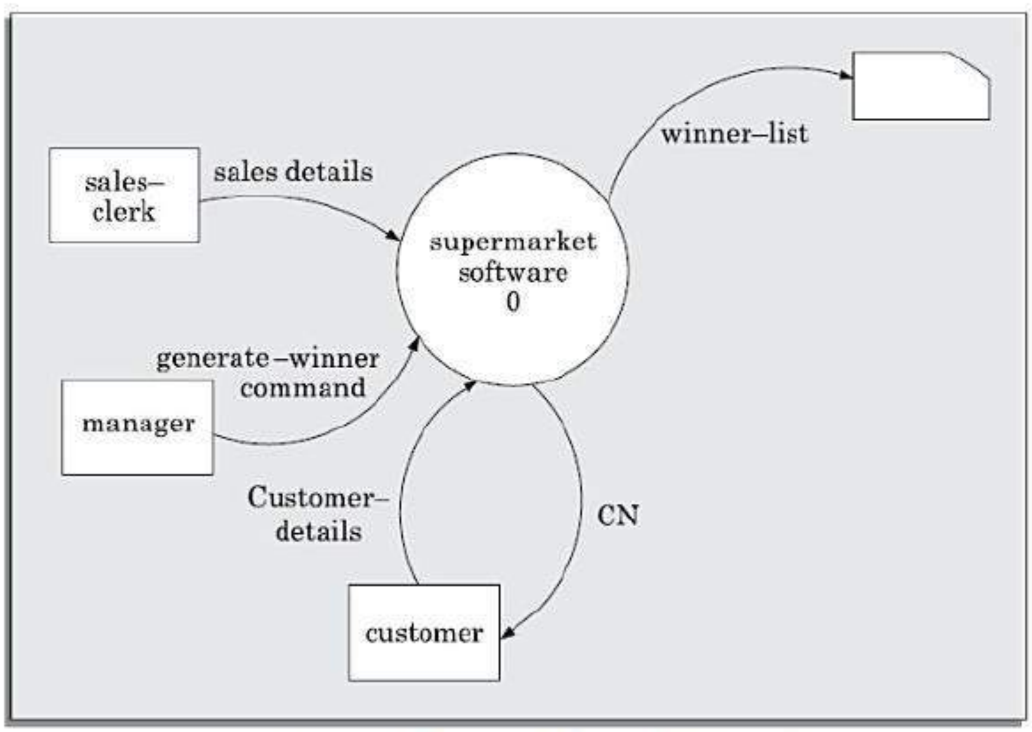


**Figure 6.10:** Context diagram for Example 6.3.

- **Figure 6.4:** DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

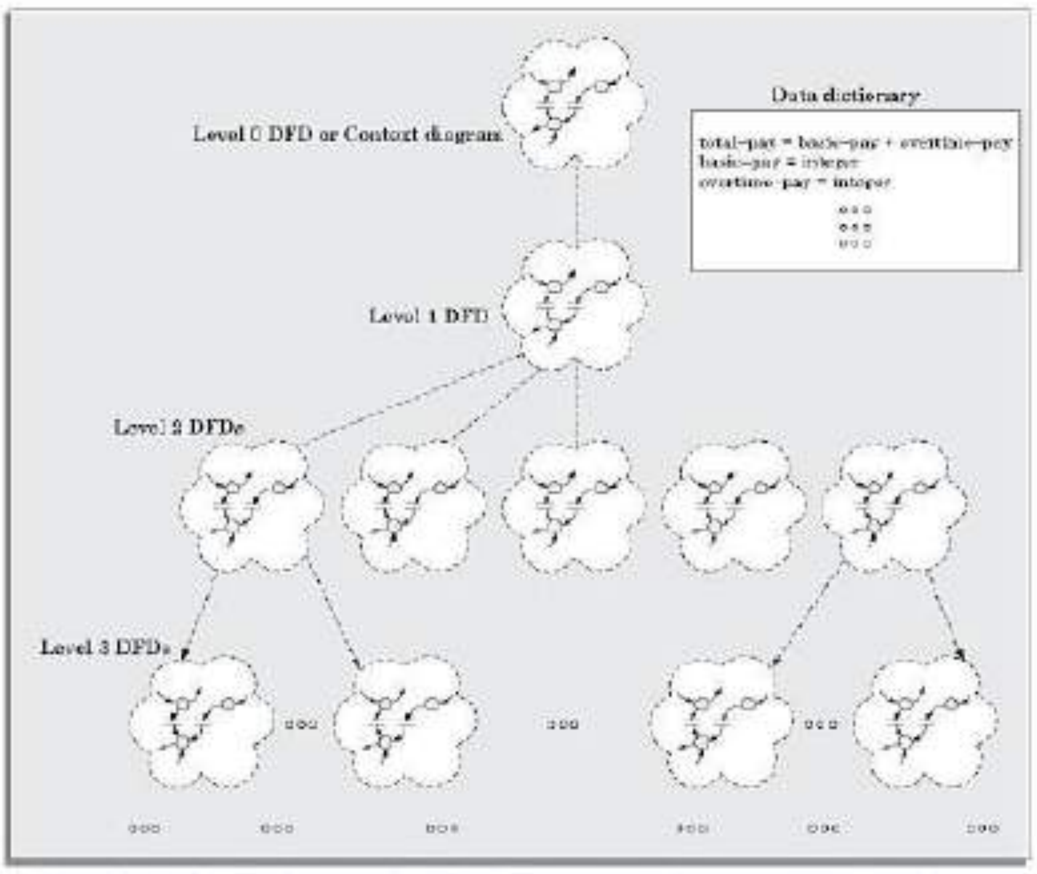


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary

- The context diagram establishes the context in which the system operates; that is,
  - who are the users,
  - what data do they input to the system, and
  - what data they received by the system.
- The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system
- The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented.
- The data input to the system and the data output from the system are represented as incoming and outgoing arrows.
- These data flow arrows should be annotated with the corresponding data names.
- **To develop the context diagram of the system, we have to analyse the SRS document to identify the different types of users**
- Here, the term users of the system also includes any external systems which supply data to or receive data from the system.

## <u>Level 1 DFD :</u>

- The level 1 DFD usually contains three to seven bubbles.
- That is, the system is represented as performing three to seven important functions.
- To develop the level 1 DFD, examine the high-level functional requirements in the SRS document.
- If there are three to seven high level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD.
- Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.
- What if a system has more than seven high-level requirements identified in the SRS document? In this case, some of the related requirements have to be combined and represented as a single bubble i n the level 1 DFD.
- These can be split appropriately in the lower DFD levels. If a system has less than three high-level functional requirements, then some of the high-level requirements need to be split into their sub-functions so that we have roughly about five to seven bubbles represented on the diagram. Level 1 DFDs Examples are shown in 6.1 to 6.4.

## Decomposition:

- Each bubble in the DFD represents a function performed by the system.
- The bubbles are decomposed into sub-functions.
- Decomposition of a bubble is also known as factoring or exploding a bubble.
- Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles.
- For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes repetitive. On the other hand, too many bubbles (i.e. more than seven bubbles) at any level of a DFD makes the DFD model hard to understand.
- Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

## Developing the DFD model of a system more systematically.

- **1. Construction of context diagram:** Examine the SRS document to determine:
  - Different high-level functions that the system needs to perform.
  - Data input to every high-level function.
  - Data output from every high-level function.
  - Interactions (data flow) among the identified high-level functions.
  - Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level data flow diagram (DFD), usually called the DFD 0.
- **2. Construction of level 1 diagram:** Examine the high-level functions described in the SRS document.
  - If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble.
  - If there are more than seven bubbles, then some of them have to be combined.
  - If there are less than three bubbles, then some of these have to be split.
- **3. Construction of lower-level diagrams:** Decompose each high-level function into its constituent sub-functions through the following set of activities:
  - Identify the different sub-functions of the high-level function.

- Identify the data input to each of these sub-functions.
- Identify the data output from each of these sub-functions.
- Identify the interactions (data flow) among these sub-functions.
- Represent these aspects in a diagrammatic form using a DFD.
- Recursively repeat Step 3 for each sub-function until a sub-function can be represented by using a simple algorithm.

## Numbering of bubbles

- It is necessary to number the different bubbles occurring in the DFD.
- These numbers help in uniquely identifying any bubble in the DFD from its bubble number.
- The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD.
- Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc.
- When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc.
- In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

## Balancing DFDs :

- The DFD model of a system usually consists of many DFDs that are organized in a hierarchy.
- In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD.
- The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.
- Balancing a DFD shown in Figure 6.5.

DEPT OF CSE RGAN

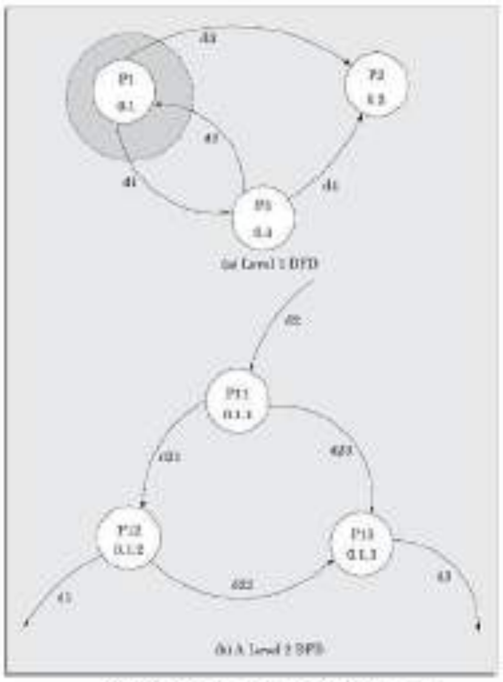


Figure 6.5: An example showing balanced decomposition

- In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle).
- In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1,0.1.2,0.1.3).
- The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.
- Dangling arrows (d1,d2,d3) represent the data flows into or out of a diagram.

  **Illustration 1.** A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message is generated. While developing the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in Figure 6.6) to indicate that the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD.

  **Figure 6.6:** It is incorrect to show control information on a DFD.

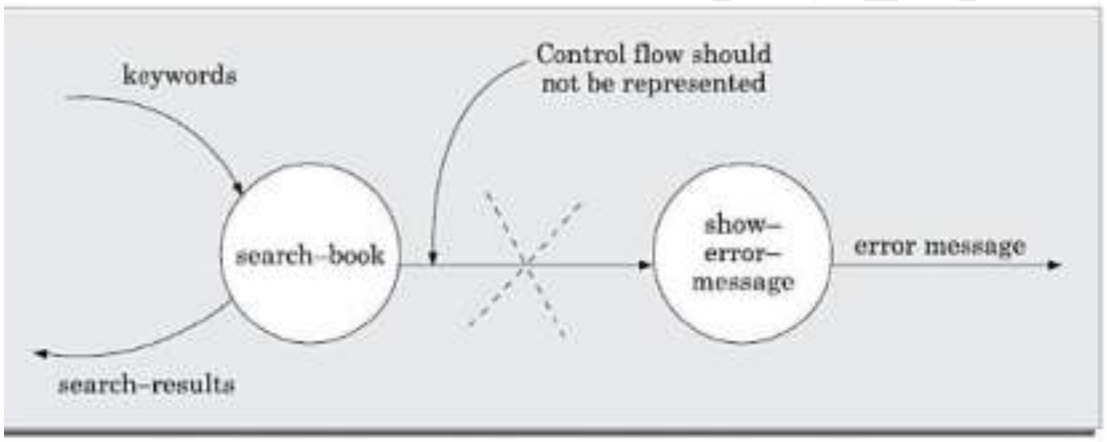


**Figure 6.6:** It is incorrect to show control information on a DFD.

**Example-1 (RMS Calculating Software)**

- A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display it.
- In this example, the context diagram is simple to draw.
- The system accepts three integers from the user and returns the result to him.
- This has been shown in Figure 6.8(a).

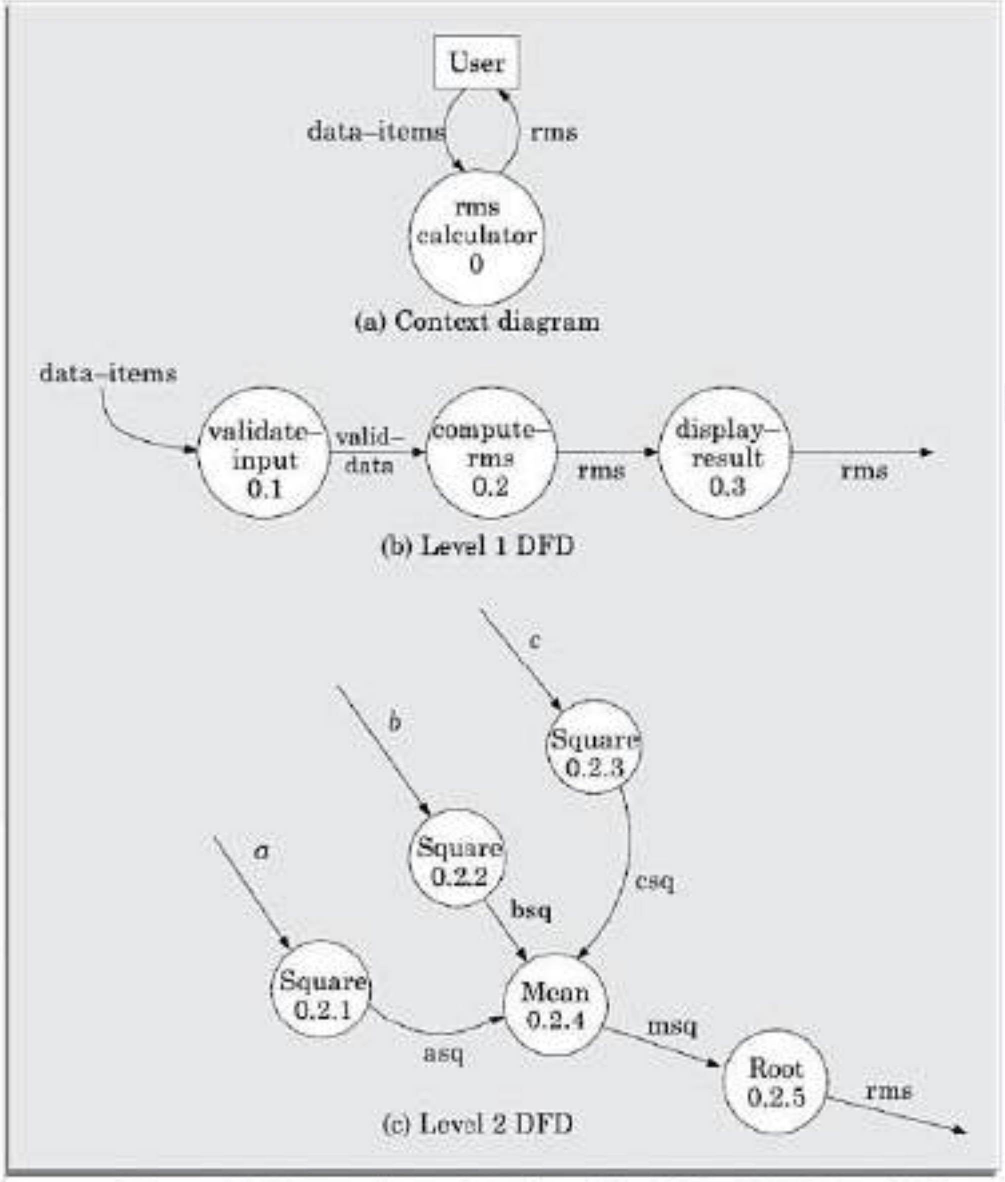


**Figure 6.8:** Context diagram, level 1, and level 2 DFDs for Example 6.1.

## 4. STRUCTURED DESIGN

- The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a **structure chart**.
- **A structure chart represents the software architecture.**
- The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules.
- The structure chart representation can be easily implemented using some programming language.
- The main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.
- The basic building blocks using which structure charts are designed are as following:
- **Rectangular boxes:** A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.
- **Module invocation arrows:** An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow.

    - However, just by looking at the structure chart, we cannot say whether a module calls another module just once or many times.
    - Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.
- **Data flow arrows:** These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.
- **Library modules:** A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.
- **Selection:** The diamond symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.
- **Repetition:** A loop around the control flow arrows denotes that the respective modules are invoked repeatedly. In any structure chart, there should be one and only one module at the top, called the root. There should be at most one control relationship between any two modules in the structure chart. This means that if module **A invokes module B**, **module B cannot invoke module A**. The main reason behind this restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules. However, it is possible for two higher-level modules to invoke the same lower-level module.
- An example of a properly layered design and another of a poorly layered design are shown in Figure 6.18.

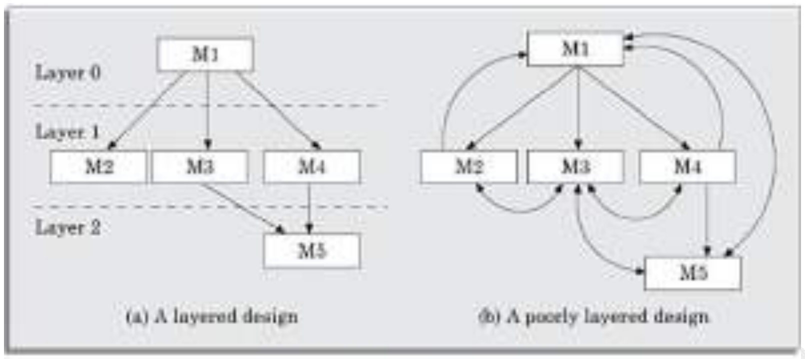


**Figure 6.18:** Examples of properly and poorly layered designs.

**Figure 6.18:** Examples of properly and poorly layered designs.

## Flow chart *versus* structure chart

- Flow chart is a convenient technique to represent the flow of control in a program.
- A structure chart differs from a flow chart in three principal ways:
    - It is usually difficult to identify the different modules of a program from its flow chart representation.
    - Data interchange among different modules is not represented in a flow chart.
    - Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.

## Transformation of a DFD Model into Structure Chart

- Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart.
- Structured design provides two strategies to guide transformation of a DFD into a structure chart:

    Transform analysis

    Transaction analysis
- Level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs.
- At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.
- Whether to apply transform or transaction processing?
- Given a specific DFD of a model, how does one decide whether to apply transform analysis or transaction analysis?
- For this, one would have to examine the data input to the diagram. The data input to the diagram can be easily spotted because they are represented by dangling arrows.
- If all the data flow into the diagram are processed in similar ways (i.e. if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable. Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very simple processing.
- The bubbles are decomposed until it represents a very simple processing that can be implemented using only a few lines of code.
- Therefore, transform analysis is normally applicable at the lower levels of a DFD model. Each different way in which data is processed corresponds to a separate transaction. Each

DEPT OF CSE RGAN

transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

**Transform analysis**

- Transform analysis identifies the primary functional components (modules) and the input and output data for these components.
- The first step in transform analysis is to divide the DFD into three types of parts:
  • Input.
  • Processing.
  • Output.
- The input portion in the DFD includes processes that transform input data from physical (e.g, character from terminal) to logical form (e.g. internal tables, lists, etc.).
- Each input portion is called an 'afferent branch' (not comparable branch).
- The output portion of a DFD transforms output data from logical form to physical form.
- Each output portion is called an efferent branch. The remaining portion of a DFD is called central transform.
- In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.
- Identifying the input and output parts requires experience and skill. One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone. Processes which validate input are not central transforms. Processes which sort input or filter data from it are central transforms.
- The first level o f structure chart is produced by representing each input and output unit as a box and each central transform as a single box.
- In the third step of transform analysis, the structure chart is refined by adding sub functions required by each of the high-level functional components.
- Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying consumer modules etc.
- The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

**Example-6 Draw the structure chart for the RMS software of Example-1.**

- By observing the level 1 DFD of Figure 6.8,

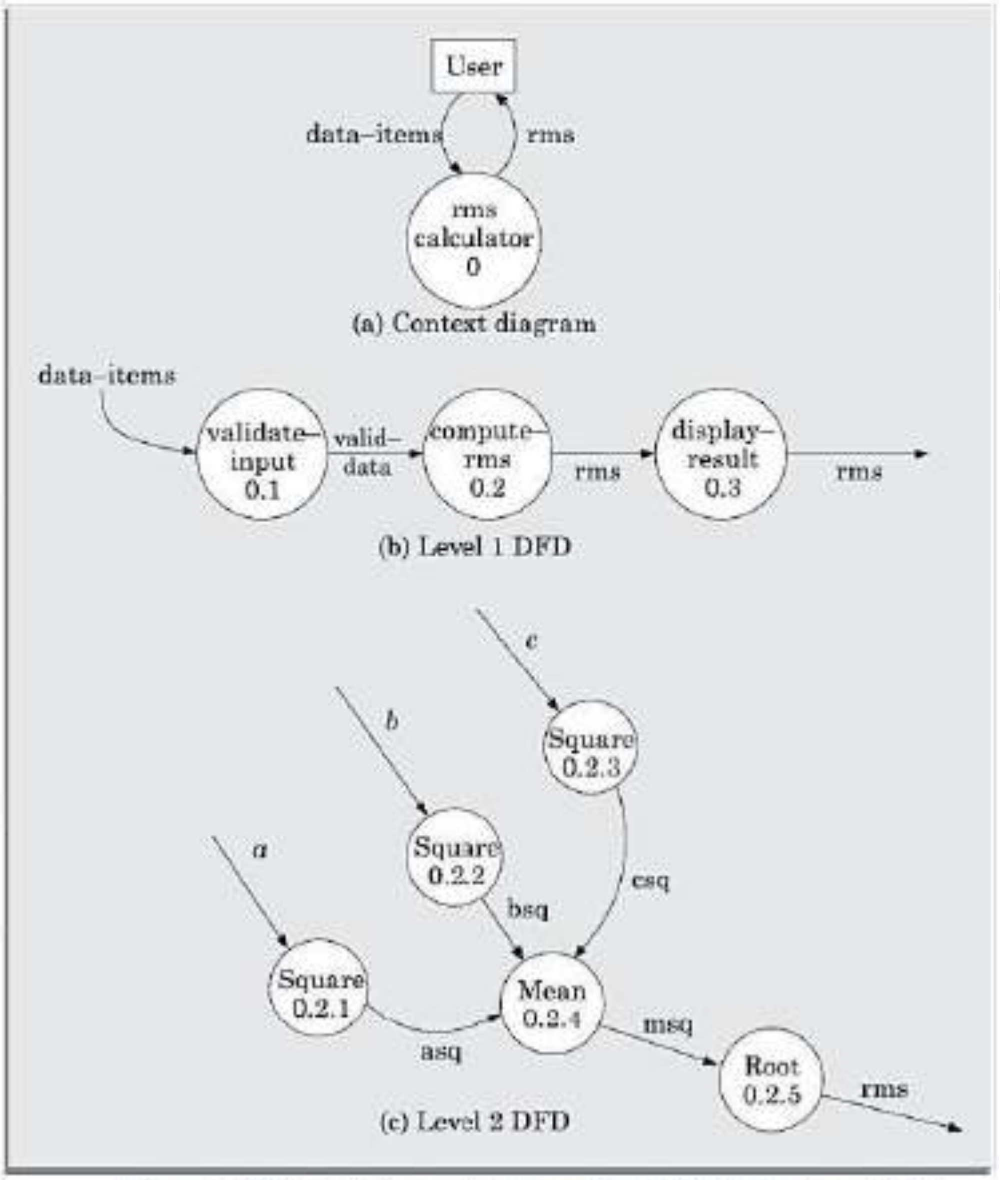


Figure 6.8: Context diagram, level 1, and level 2 DFDs for Example 6.1.

- we can identify validate-input as the afferent branch and write-output as the efferent branch. The remaining (i.e., compute-rms) as the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in Figure 6.19.

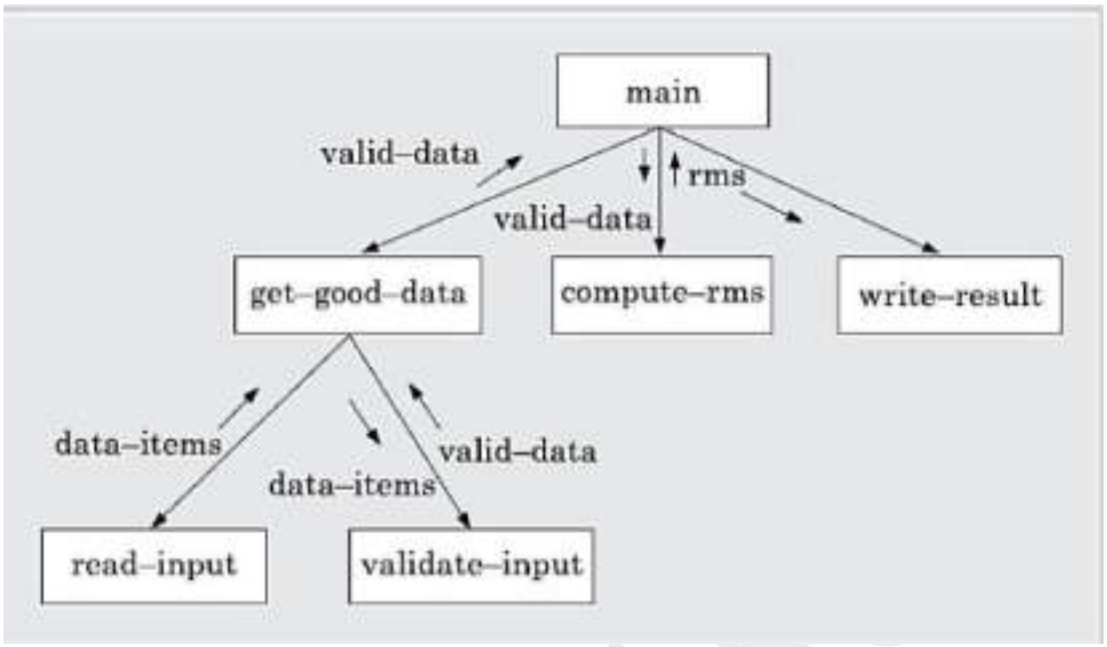


**Figure 6.19:** Structure chart for Example-1.

## 5. DETAILED DESIGN

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart.
- These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

## 6. DESIGN REVIEW

- After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team.
- Normally, members of the team who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents especially for the following aspects:
- **Traceability:** Whether each bubble of the DFD can be traced to some module in the structure chart a nd vice versa. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and vice versa.
- **Correctness:** Whether all the algorithms and data structures of the detailed design are correct.
- **Maintainability:** Whether the design can be easily maintained in future.
- **Implementation:** Whether the design can be easily and efficiently be implemented.
- After the points raised by the reviewers are addressed by the designers,

the design document becomes ready for implementation.

# 7. USER INTERFACE DESIGN

- The user interface portion of a software product is responsible for all interactions with the user.
- Almost every software product has a user interface.
- In the early days of computer, no software product had any user interface.
- The computers those days were batch systems and no interactions with the users were supported.
- Now, we know that things are very different—almost every software product is highly interactive.
- The user interface part of a software product is responsible for all interactions with the end-user. Consequently, the user interface part of any software product is of direct concern to the end-users.
- No wonder then that many users often judge a software product based on its user interface.
- An interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction.
- Users become particularly irritated when a system behaves in an unexpected ways, i.e., issued commands do not carry out actions according to the intuitive expectations of the user.
- Normally, when a user starts using a system, he builds a mental model of the system and expects the system behavior to conform to it.
- For example, if a user action causes one type of system activity and response under some context, then the user would expect similar system activity and response to occur for similar user actions in similar contexts.
- Therefore, sufficient care and attention should be paid to the design of the user interface of any software product.
- Development of a good user interface usually takes significant portion of the total system development effort. For many interactive applications, as much as 50 per cent of the total development effort is spent on developing the user interface part. Unless the user interface is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously.
- Therefore, it is necessary to carefully study various concepts associated with user interface design and understand various systematic techniques available for the development of user interface.
- In this chapter, we first discuss some common terminologies and concepts associated with development of user interfaces.
- Then, we classify the different types of interfaces commonly being used. We also provide some guidelines for designing good interfaces, and discuss some tools for development of graphical user interfaces (GUIs). Finally, we present a GUI development methodology.

## 7.1 CHARACTERISTICS OF A GOOD USER INTERFACE

- It is important to identify the different characteristics that are usually desired of a good user interface.
- Unless we know what exactly is expected of a good user interface, we cannot possibly design one.

- In the following subsections, we identify a few important characteristics of a good user interface:
- **Speed of learning:** A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. **A good user interface should not require its users to memorize commands.** Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:
  - **Use of metaphors and intuitive command names:** If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, **users can immediately relate to it.** Another popular metaphor is a shopping cart. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses the shopping cart metaphor for designing the interaction style for a situation where similar types of choices have to be made, then the users can easily understand and learn to use the interface.
  - **Consistency:** Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.
  - **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with. This can be achieved if the interfaces of different applications are developed using some standard user interface components.
- **Speed of use:** Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.
  - This characteristic of the interface is sometimes referred to as productivity support of the interface. It indicates how fast the users can perform their intended tasks.
  - The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users.
  - The most frequently used commands should have the smallest length or be available at the top of a menu to minimize the mouse movements necessary to issue commands.
- **Speed of recall:** Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.
- **Error prevention:** A good user interface should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily

determined by monitoring the errors committed by an average users while using the interface.

- **Aesthetic and attractive:** A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.
- **Consistency:** The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another.
- **Feedback:** A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request.
- **Support for multiple skill levels:** A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects.
- **Error recovery (undo facility):** While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors they commit while using software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.
- **User guidance and on-line help:** Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

## 8. BASIC CONCEPTS

- **1. User Guidance and On-line Help:** Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system.
  - **On-line help system:** Users expect the on-line help messages to be tailored to the context in which they invoke the “help system”. Therefore, a good online help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way.
  - **Guidance messages:** The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc. A good guidance system should have different levels of sophistication for different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface
  - **Error messages:** Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc.

- **2. <u>Mode-based</u> versus <u>Modeless</u> Interface:**
  - A **mode** is a state or collection of states in which only a subset of all user interaction tasks can be performed.
  - a **modeless** interface, the same set of commands can be invoked at any time during the running of the software.
  - Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software.
  - On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is, i.e., the mode at any instant is determined by the sequence of commands already issued by the user.
  - A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode.
  - Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.
- **3 Graphical User Interface (GUI) versus Text-based User Interface:**
  - In a **GUI** multiple windows with different information can simultaneously be displayed on the user screen.
  - This is perhaps one of the biggest advantages of GUI over text- based interfaces since the user has the flexibility to simultaneously interact with several related items at any time
  - Iconic information representation and symbolic information manipulation is possible in a GUI.
  - Symbolic information manipulation such as dragging an icon representing a file to a trash for deleting is intuitively very appealing and the user can instantly remember it.
  - A GUI usually supports command selection using an attractive and user-friendly menu selection system.
  - In a GUI, a **pointing device** such as a mouse or a **light pen** can be used for issuing commands.
  - The use of a **pointing device** increases the efficacy of command issue procedure.
  - A GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse.
  - **A text-based user interface** can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals.
  - However, display terminals with graphics capability with bitmapped high-resolution displays and significant amount of local processing power have become affordable and over the years have replaced text-based terminals on all desktops.

## 9. <u>TYPES OF USER INTERFACES</u>

- Broadly speaking, user interfaces can be classified into the following three categories:

  **Command language-based interfaces**

  **Menu-based interfaces**

  **Direct manipulation interfaces**

- Each of these categories of interfaces has its own characteristic advantages and disadvantages.
- Therefore, most modern applications use a careful combination of all these three types of user interfaces for implementing the user command repertoire.
- It is very difficult to come up with a simple set of guidelines as to which parts of the interface should be implemented using what type of interface.
- This choice is to a large extent dependent on the experience and discretion of the designer of the interface.

**1. Command Language-based Interface**

- A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands.
- The user is expected to frame the appropriate commands in the language and type them appropriately whenever required.
- A simple command language-based interface might simply assign unique names to the different commands.
- However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands.
- A command language-based interface can be made concise requiring minimal typing by the user.
- Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.
- Among the three categories of interfaces, the command language interface allows for most efficient command issue procedure requiring minimal typing.
- Further, a command language-based interface can be implemented even on cheap alphanumeric terminals.
- Also, a command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interfaces because compiler writing techniques are well developed.
- One can systematically develop a command language interface by using the standard compiler writing tools LEX and YACC.
- However, command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are **difficult to learn** and require the user to **memorize the set of primitive commands.**
- Also, most **users make errors while formulating commands in the command language** and also while typing them.
- Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse.

- Obviously, for casual and inexperienced users, **command language-based interfaces are not suitable**.

**2. Menu-based Interface**

- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.
- A menu-based interface is based on recognition of the command names, rather than recollection.
- Humans are much better in recognizing something than recollecting it.
- Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device.
- This factor is an important consideration for the occasional user who cannot type fast.
- However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands.
- Composing commands in a menu based interface is not possible.
- Also, if the number of choices is large, it is difficult to design a menu-based interface.
- Moderate-sized software might need hundreds or thousands of different menu choices.
- In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms.
- In the following, the techniques available to structure a large number of menu items:
- **Scrolling menu:** Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required.
    - This would enable the user to view and select the menu items that cannot be accommodated on the screen.
    - In a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs.
    - This is important since the user cannot see all the commands at any one time.
    - An example situation where a scrolling menu is frequently used is font size selection in a document processor (see Figure 9.1).

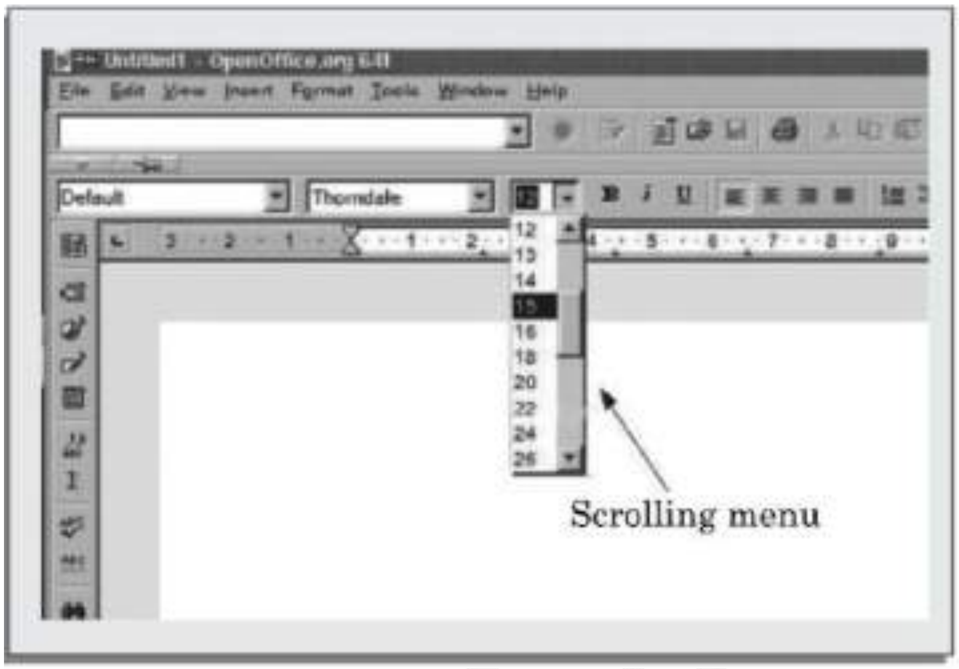

Untitled1 - OpenOffice.org 6.0
File Edit View Insert Format Tools Window Help
Default
Thorndale
12
13
14
15
16
18
20
22
24
26
Scrolling menu

**Figure 9.1:** Font size selection using scrolling menu.

- **Walking menu:** Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu. An example of a walking menu is shown in Figure 9.2.

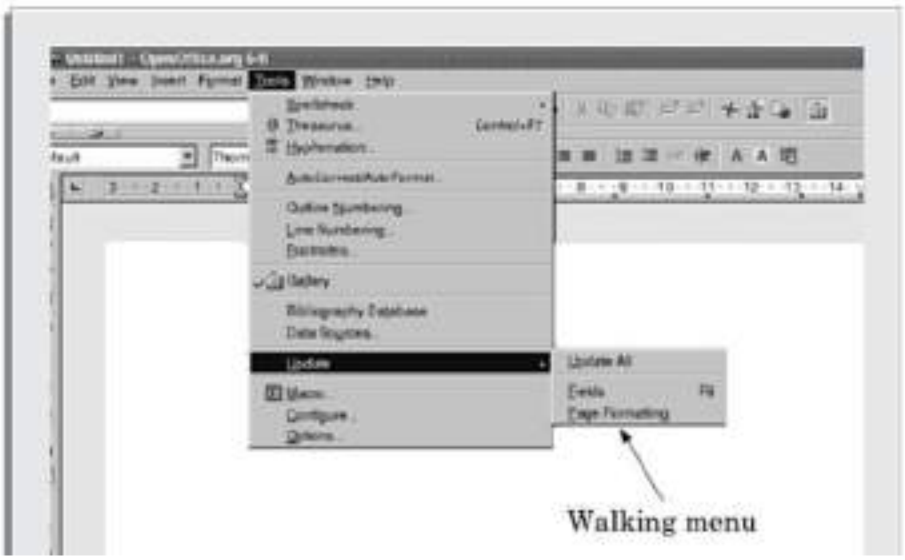
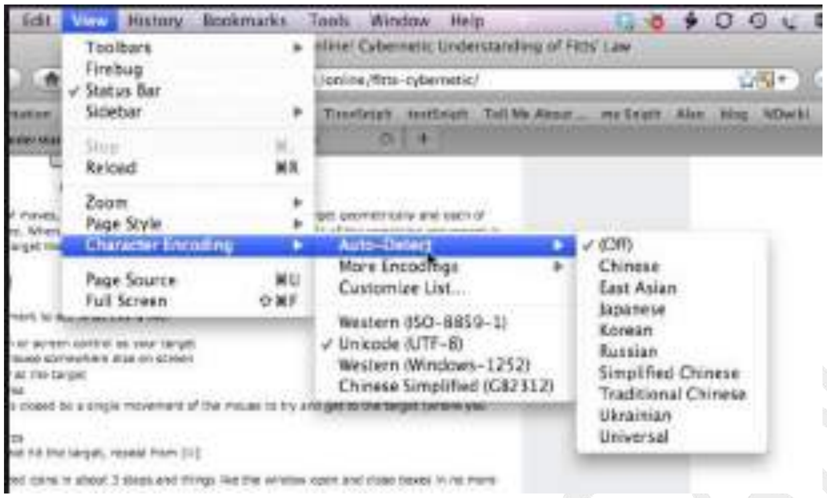


**Figure 9.2:** Example of walking menu.

- **Hierarchical menu:** This type of menu is suitable for small screens with limited display area such as that in mobile phones. In a hierarchical menu, the menu items are organized in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus in this case, one can consider the menu and its various submenu to form a hierarchical tree-like structure



- .

**3. Direct Manipulation Interfaces**

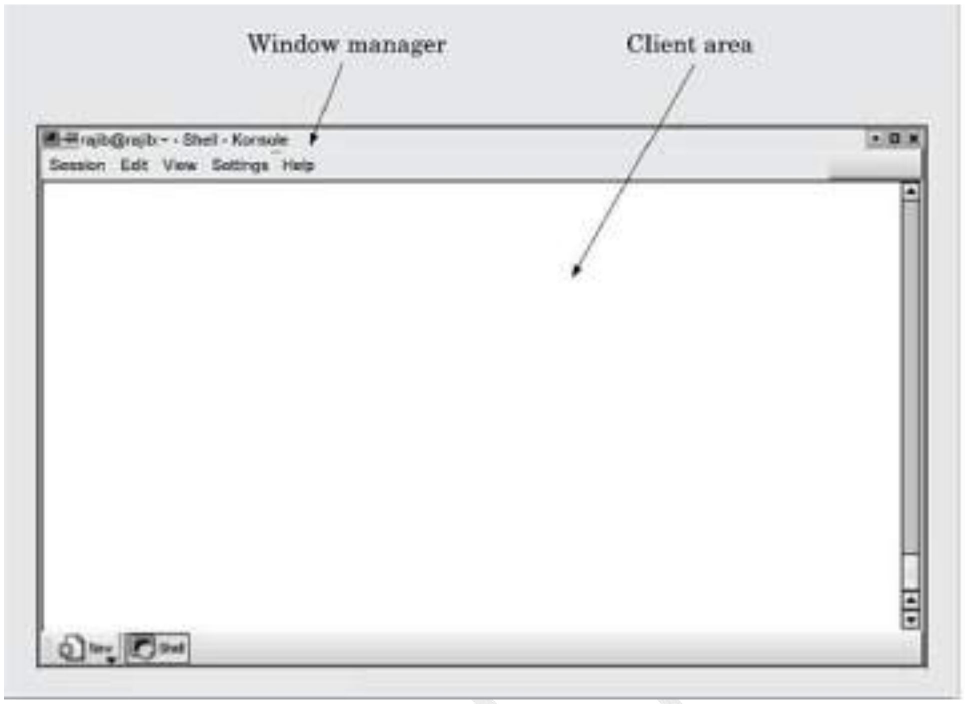
- Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons2 or objects).
- For this reason, direct manipulation interfaces are sometimes called as iconic interfaces. In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon representing a file into an icon representing a trash box, for deleting the file.
- Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language independent.

**10. <u>FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT</u>**

- Graphical user interfaces became popular in the 1980s.
- The main reason why there were very few GUI-based applications prior to the eighties is that graphics terminals were too expensive.
- For example, the price of a graphics terminal those days was much more than what a high-end personal computer costs these days.
- The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.
- In the following subsections, an overview of the window management system, the component-based development style, and visual programming.

**10.1 Window System**

- Most modern graphical user interfaces are developed using some window system. A window system can generate displays through a set of windows. Since a window is the basic entity in such a graphical user interface, we need to first discuss what exactly a window is.
- **Window:** A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc.


Window manager
Client area
rajib@rajib:~ - Shell - Konsole
Session Edit View Settings Help
New
Shell
**Figure 9.3:** Window with client and user areas marked.

- A window can be divided into two parts—**client part**, and **non-client part**.
- The client area makes up the whole of the window, except for the borders and scroll bars.
- The client area is the area available to a client application for display.
- The non-client-part of the window determines the look and feel of the window.
- The look and feel defines a basic behavior for all windows, such as creating, moving, resizing, and iconifying of the windows.
- The window manager is responsible for managing and maintaining the non-client area of a window. A basic window with its different parts is shown in Figure 9.3.

- **Window Management System (WMS) :** A graphical user interface typically consists of a large number of windows.
  - Therefore, it is necessary to have some systematic way to manage these windows.
  - Most graphical user interface development environments do this through a window management system (WMS).
  - A window management system is primarily a resource manager.
  - It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS) —which not only does resource management, but also provides the basic behavior to the windows and provides several utility routines to the application programmer for user interface development.
  - A WMS simplifies the task of a GUI designer to a great extent by providing the basic behavior to the various windows such as move, resize, iconify, etc. A WMS consists of two parts (see Figure 9.4): a **window manager**, and a **window system**.

- **Window manager and window system:** The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system.
    - The window manager and not the window system determine how the windows look and behave.
    - The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system.
    - The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in Figure 9.4.
    - This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manger invoke services of the window manager.
    - Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.

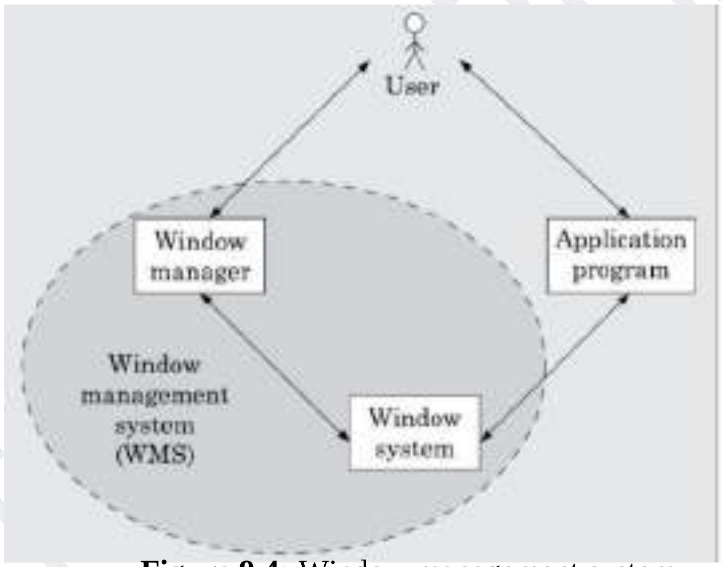


**Figure 9.4:** Window management system.

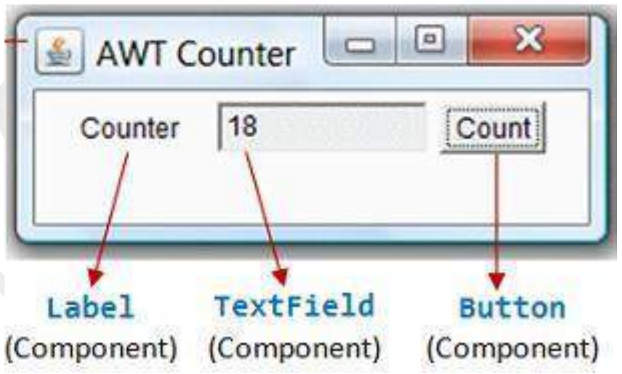
## 10.2 Component-based development

- A development style based on widgets is called component-based (or widget-based ) GUI development style.
- There are several important advantages of using a widget-based design style.
- One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast. In this style of development, the user interfaces for different applications are built from the same basic components.
- **Visual programming:** Visual programming is the drag and drop style of program development.
    - In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment.
    - The application programmer can easily develop the user interface by dragging the required component types (e.g., menu, forms, etc.) from the displayed icons and placing them wherever required.

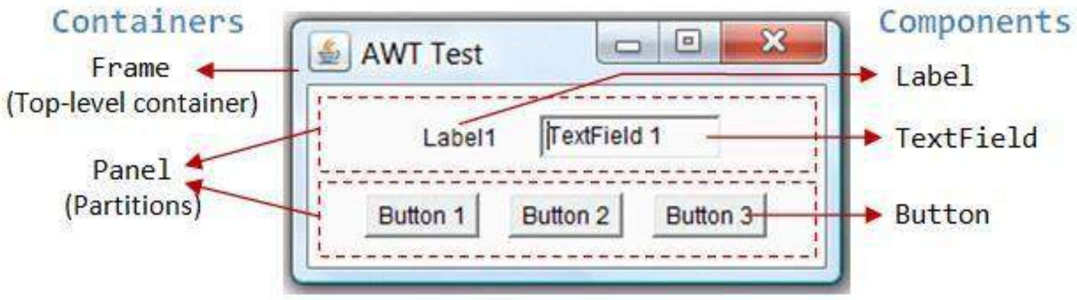
- Thus, visual programming can be considered as program development through manipulation of several visual objects.
- Reuse of program components in the form of visual objects is an important aspect of this style of programming.
- Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g., factory design), simulation, etc.
- User interface development using a visual programming language greatly reduces the effort required to develop the interface.
- Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with window based user interfaces for Microsoft Windows environments.

## 10.3 Types of Widgets

- Widget is an application, or a component of an interface, that enables a user to perform a function or access a service.
- Different interface programming packages support different widget sets.
- However, a surprising number of them contain similar kinds of widgets.
- The following widgets we have chosen as representatives of this generic class.
- **Label widget:** This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

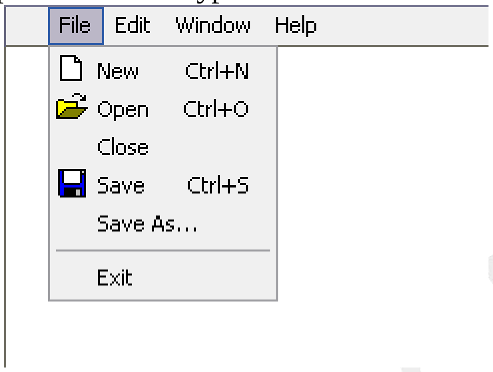


- **Container widget:** These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.



DEPT OF CSE RGAN

- **Pop-up menu:** These are transient and task specific. A pop-up menu appears upon pressing the mouse right button, irrespective of the mouse position.
- **Pull-down menu:** These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.



- **Dialog boxes:** We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Though most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.
- **Push button:** A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (. . . ). A push button with an ellipsis generally indicates that another dialog box will appear.
- **Radio buttons:** A set of radio buttons are used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.
- **Combo boxes:** A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

## 10.4 An Overview of X-Window/MOTIF

- One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that **it allows development of portable GUIs.**
- **Applications developed using the X-window system are device independent.**
- Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent GUI operation has been schematically represented in Figure 9.5.

DEPT OF CSE RIAN

- Here, A is the computer application in which the application is running. B can be any computer on the network from where you can interact with the application.
- Network independent GUI was pioneered by the X-window system in the mid-eighties at MIT (Massachusetts Institute of Technology) with support from DEC (Digital Equipment Corporation). Now-a-days many user interface development systems support network-independent GUI development, e.g., the AWT and Swing components of Java.

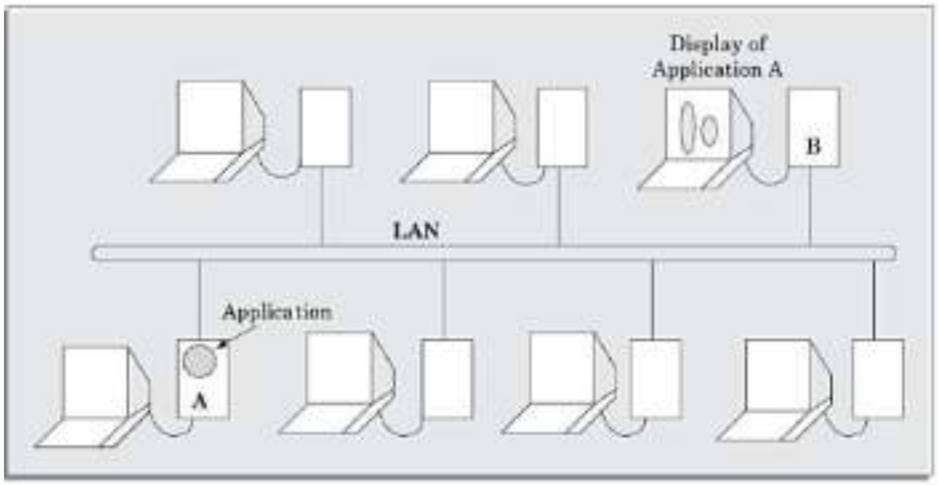


**Figure 9.5:** Network-independent GUI.

- The X-window functions are low level functions written in C language which can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines.

## 10.5 X Architecture

- The X architecture is pictorially depicted in Figure 9.6. The different terms used in this diagram are explained as follows:

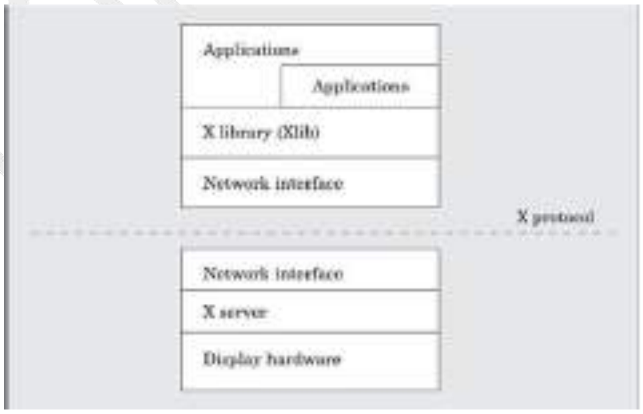


**Figure 9.6:** Architecture of the X System.

- **Xserver:** The X server runs on the hardware to which the display and the key board are attached. The X server performs low-level graphics, manages window, and user input functions. The X server controls accesses to a bit-mapped graphics display resource and manages it.
- **X protocol.** The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network protocol, and the programming language used.

- **X library (Xlib).** The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server. Xlib provides low level primitives for developing an user interface, such as displaying a window, drawing characters and graphics on the window, waiting for specific events, etc.
- **Xtoolkit (Xt).** The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, menu bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface. In order to develop a user interface, the designer has to put together the set of components (widgets) he needs, and then he needs to define the characteristics (called resources) and behavior of these widgets by using the intrinsic routines to complete the development of the interface. Therefore, developing an interface using Xtoolkit is much easier than developing the same interface using only X library.

## 11. A USER INTERFACE DESIGN METHODOLOGY

- At present, no step-by-step methodology is available which can be followed by rote to come up with a good user interface.
- What we present in this section is a set of recommendations which you can use to complement your ingenuity. Even though almost all popular GUI design methodologies are user-centered, this concept has to be clearly distinguished from a user interface design by users.
- Before we start discussing about the user interface design methodology, let us distinguish between a user-centered design and a design by users.
- User-centered design is the theme of almost all modern user interface design techniques.
- However, user-centered design does not mean design by users. One should not get the users to design the interface, nor should one assume that the user's opinion of which design alternative is superior is always right.

### 11.1 Implications of Human Cognition Capabilities on User Interface Design

- An area of human-computer interaction where extensive research has been conducted is how human cognitive capabilities and limitations influence the way an interface should be designed. In the following subsections, we discuss some of the prominent issues that have been extensively reported in the literature.
- **Limited memory:** Humans can remember at most seven unrelated items of information for short periods of time. Therefore, the GUI designer should not require the user to remember too many items of information at a time. It is the GUI designer's responsibility to anticipate what information the user will need at what point of each task and to ensure that the relevant information is displayed for the user to see.
- **Frequent task closure:** Doing a task (except for very trivial tasks) requires doing several subtasks. When the system gives a clear feedback to the user that a task has been successfully completed, the user gets a sense of achievement and relief. The user can clear out information regarding the completed task from memory. This is known as task closure.

- **Recognition rather than recall:** Information recall incurs a larger memory burden on the users and is to be avoided as far as possible. On the other hand, recognition of information from the alternatives shown to him is more acceptable.
- **Procedural versus object-oriented:** Procedural designs focus on tasks, prompting the user in each step of the task, giving them very few options for anything else. This approach is best applied in situations where the tasks are narrow and well-defined or where the users are inexperienced, such as a bank ATM. An object-oriented interface on the other hand focuses on objects. This allows the users a wide range of options.

**11.2 A GUI Design Methodology**

- The GUI design methodology we present here is based on the seminal work of Frank Ludolph. Our user interface design methodology consists of the following important steps:
- Detailed presentation and graphics design. GUI construction. Usability evaluation.

**Examining the use case model**

- We now elaborate the above steps in GUI design. The starting point for GUI design is the use case model.
- This captures the important tasks the users need to perform using the software. As far as possible, a user interface should be developed using one or more metaphors.
- **Metaphors** help in interface development at lower effort and reduced costs for training the users.
- Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc.
- A solution based on metaphors is easily understood by the users, reducing learning time and training costs.
- Some commonly used metaphors are the following:
  White board, Shopping cart, Desktop, Editor's work bench, White page, Yellow page
  Office cabinet, Post box, Bulletin board, Visitor's Book.

**Task and object modelling**

- A task is a human activity intended to achieve some goals. Examples of task goals can be as follows:
  Reserve an airline seat
  Buy an item
  Transfer money from one account to another
  Book a cargo for transmission to an address

- A task model is an abstract model of the structure of a task.
- A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal.
- Each task can be modeled as a hierarchy of subtasks.
- A task model can be drawn using a graphical notation similar to the activity network model.
- Tasks can be drawn as boxes with lines showing how a task is broken down into subtasks. An underlined task box would mean that no further decomposition of the task is required.
- An example of decomposition of a task into subtasks is shown in Figure 9.7.

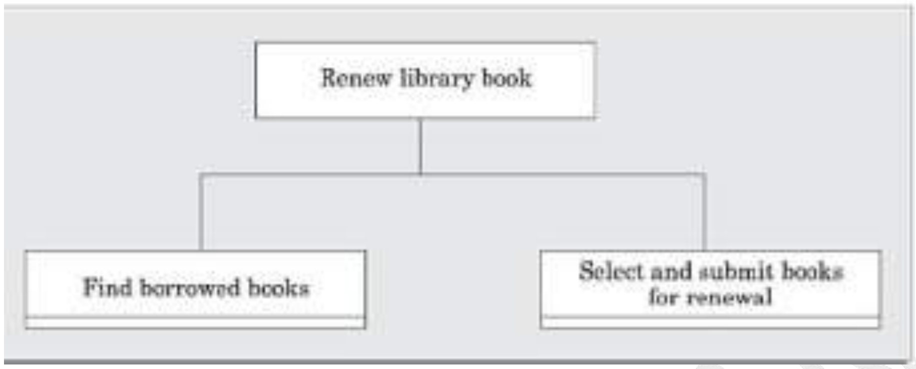


**Figure 9.7:** Decomposition of a task into subtasks.

- Identification of the user objects forms the basis of an object-based design.
- A user object model is a model of business objects which the end-users believe that they are interacting with.
- The objects in a library software may be books, journals, members, etc.
- The objects in the supermarket automation software may be items, bills, indents, shopping list, etc.
- The state diagram for an object can be drawn using a notation similar to that used by UML..
- The state diagram of an object model can be used to determine which menu items should be dimmed in a state.
- An example state chart diagram for an order object is shown in Figure 9.8.

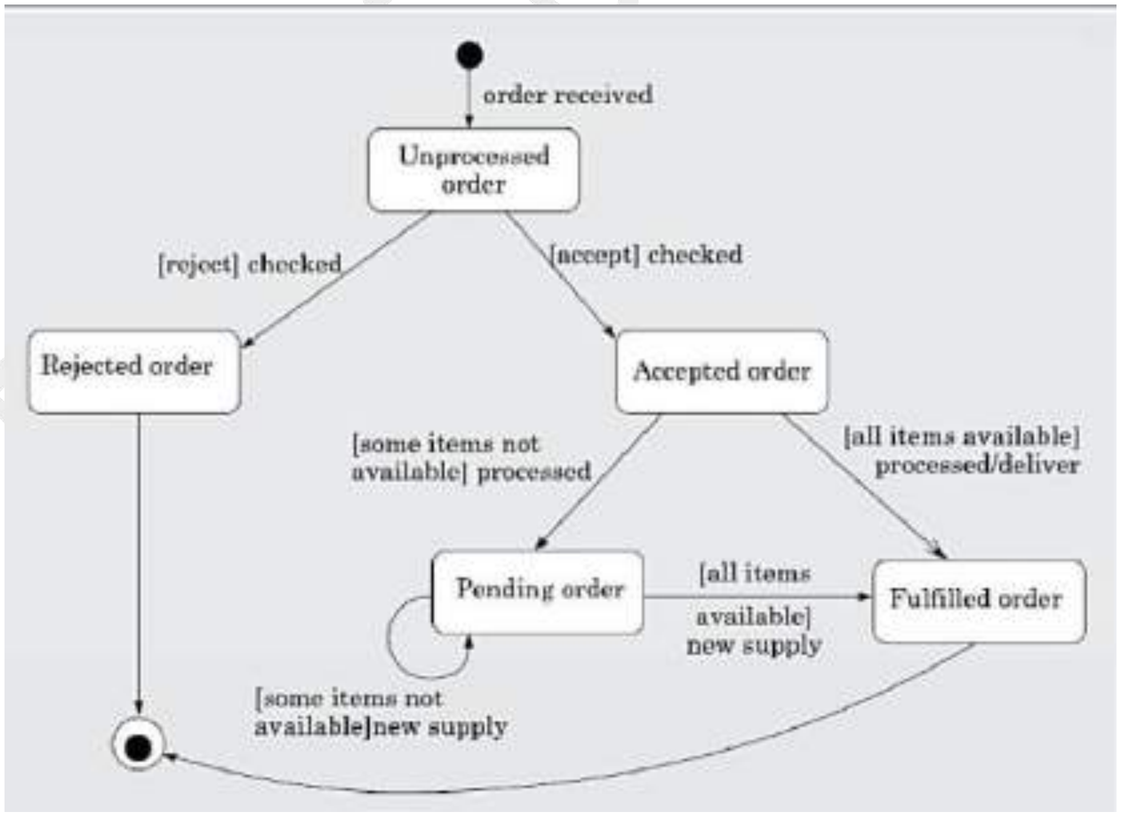

order received
Unprocessed order
[reject] checked
[accept] checked
Rejected order
Accepted order
[some items not available] processed
[all items available] processed/deliver
Pending order
[all items available] new supply
Fulfilled order
[some items not available]new supply

**Figure 9.8:** State chart diagram for an order object.

**Metaphor selection**

- The first place one should look for while trying to identify the **candidate metaphors is the set of parallels to objects**, tasks, and terminologies of the use cases. If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects.
- **Example 9.1** We need to develop the interface for a web-based pay-order shop, where the users can examine the contents of the shop through a web browser and can order them. Several metaphors are possible for different parts of this problem as follows: Different items can be picked up from racks and examined. The user can request for the **catalog** associated with the items by clicking on the item. Related items can be picked from the drawers of an item cabinet. The items can be organized in the form of a book, similar to the way information about electronic components is organized in a semiconductor hand book. Once the users make up their mind about an item they wish to buy, they can put them into a shopping cart.
- **Interaction design and rough layout:** The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc. This involves making a choice from a set of available components that would best suit the subtask. Rough layout concerns how the controls, another widgets to be organized in windows.
- **Detailed presentation and graphics design:** Each window should represent either an object or many objects that have a clear relationship to each other. At one extreme, each object view could be in its own window. But, this is likely to lead to too much window opening, closing, moving, and resizing. At the other extreme, all the views could be placed in one window side-by-side, resulting in a very large window. This would force the user to move the cursor around the window to look for different objects.
- **GUI construction:** Some of the windows have to be defined as modal dialogs. When a window is a modal dialog, no other windows in the application are accessible until the current window is closed. When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked. Modal dialogs are commonly used when an explicit confirmation or authorization step is required for an action (e.g., confirmation of delete). Though use of modal dialogs is essential in some situations, overuse of modal dialogs reduces user flexibility. In particular, sequences of modal dialogs should be avoided.
- **User interface inspection: B**uild a check list of points which can be easily checked for an interface.
- The check list details are as follows.
- **Visibility of the system status:** The system should as far as possible keep the user informed about the status of the system and what is going on.
- **Match between the system and the real world:** The system should speak the user's language with words, phrases, and concepts familiar to that used by the user, rather than using system-oriented terms.
- **Undoing mistakes:** The user should feel that he is in control rather than feeling helpless or to be at the control of the system. An important step toward this is that the users should be able to undo and redo operations.

- **Consistency:** The users should not have to wonder whether different words, concepts, and operations mean the same thing in different situations.
- **Recognition rather than recall:** The user should not have to recall information which was presented in another screen. All data and instructions should be visible on the screen for selection by the user.
- **Support for multiple skill levels:** Provision of accelerators for experienced users allows them to efficiently carry out the actions they most frequently require to perform.
- **Aesthetic and minimalist design:** Dialogs and screens should not contain information which are irrelevant and are rarely needed. Every extra unit of information in a dialog or screen competes with the relevant units and diminishes their visibility.
- **Help and error messages:** These should be expressed in plain language (no codes), precisely indicating the problem, and constructively suggesting a solution.
- **Error prevention:** Error possibilities should be minimized. A key principle in this regard is to prevent the user from entering wrong values.

# SOFTWARE ENGINEERING UNIT-4

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING.

Dr DEEPAK NEDUNURI

[ SIR C R REDDY COLLEGE OF ENGINEERING ]

ELURU.

# SOFTWARE ENGINEERING
# UNIT-4

1. Coding and Testing: Coding,
2. Code Review,
3. Software Documentation,
4. Testing,
5. Unit Testing,
6. Black-Box Testing,
7. White-Box Testing,
8. Debugging,
9. Program Analysis Tool,
10. Integration Testing,
11. Testing Object-Oriented Programs,
12. System Testing,
13. Some General Issues Associated with Testing

# 1. <u>**CODING AND TESTING**</u> ✷

- **Coding** is undertaken once the design phase is complete and the design documents have been successfully reviewed.
- In the coding phase, every module specified in the design document is coded and unit tested.
- During unit testing, each module is tested in isolation from other modules.
- That is, a module is tested independently as and when its coding is complete.
- After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.
- Integration and testing of modules is carried out according to an integration plan.
- The integration plan, according to which different modules are integrated together.
- During each integration step, a number of modules are added to the partially integrated system and the resultant system is tested.
- **The full product takes shape only after all the modules have been integrated together.**
- System testing is conducted on the full product.
- During system testing, the product is tested against its requirements as recorded in the SRS document.

- **Testing** is an important phase in software development.
- Testing of professional software is carried out using a large number of test cases.
- It is usually the case that many of the different test cases can be executed in parallel by different team members.
- Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed.
- In a typical development organization, at any time, the maximum number of software engineers can be found to be engaged in testing activities.
- It is not very surprising then that in the software industry there is always a large demand for software test engineers.
- Now Software testers are looked upon as masters of specialized concepts, techniques, and tools.
- Testing a software product is as much challenging as initial development activities such as specifications, design, and coding.
- Moreover, testing involves a lot of creative thinking.

## CODING

- The input to the coding phase is the design document.
- Design document contains both <u>module structure</u> (e.g., a structure chart) and <u>detailed design.</u>
- The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified.
- During the coding phase, different modules identified in the design document are coded according to their respective module specifications.
- The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.
- A <u>**coding standard**</u> gives a uniform appearance to the codes written by different engineers.
- It facilitates code understanding and code reuse.
- It promotes good programming practices.

- A coding standard lists several rules to be followed during coding.
- Besides the coding standards, several coding guidelines are also prescribed by software companies.
- But, what is the Difference between a coding guideline and a coding standard? It is **Mandatory for the programmers to follow the coding standards**.
- Coding guidelines provide some general suggestions regarding the coding style to be followed.
- After a module has been coded, usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing.
- It is important to detect as many errors as possible during code reviews.

## 1.1 Coding Standards and Guidelines

- Good software development organizations usually develop their own coding standards and guidelines depending on what suits their organization best and based on the specific types of software they develop.

### Representative coding standards

- **Rules for limiting the use of global:** These rules list what types of data can be declared global.
- **Standard headers for different modules:** The header of different modules should have standard format and information for ease of understanding and maintenance.
- The following is an example of header format that is being used in some companies:
  Name of the module.
  Date on which the module was created.
  Author’s name.
  Modification history.
  Synopsis of the module.
- **Naming conventions for global variables, local variables, and constant identifiers:** A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g., Global Data) and local variable names start with small letters (e.g., Local Data). Constant names should be formed using capital letters only (e.g., CONSTDATA).
- **Conventions regarding error return values and exception handling mechanisms:** The way error conditions are reported by different functions in a program should be standard within an organization. For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code. This facilitates reuse and debugging.
- **Representative coding guidelines:** The following are some representative coding guidelines that are recommended by many software development organizations. Wherever necessary, the rationale behind these guidelines is also mentioned.
- **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.
- **Avoid obscure side effects:** The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure ( indistinct ) side effect is one that is not obvious from a

casual examination of the code. Obscure side effects make it difficult to understand a piece of code.

- **Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. For example, some programmers make use of a temporary loop variable for also computing and storing the final result.
- **Code should be well-documented:** As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.
- **Length of any function should not exceed 10 source lines:** A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
- **Do not use GO TO statements:** Use of GO TO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

## 2. CODE REVIEW ✶

- Testing is an effective defect removal mechanism.
- However, testing is applicable to only executable code.
- **Review** is a very effective technique to remove defects from source code.
- In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing.
- Code review for a module is undertaken after the module successfully compiles.
- That is, all the syntax errors have been eliminated from the module.
- Obviously, code review does not target to design syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors.
- Code review has been recognized as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.
- Reviews directly detect errors.
- Testing only helps detect failures
- Normally, the following two types of reviews are carried out on the code of a module:
  - Code Inspection.
  - Code Walkthrough.

### 2.1 Code Walkthrough

- Code walkthrough is an informal code analysis technique.
- In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated.
- A few members of the development team are given the code a couple of days before the walkthrough meeting.
- Each member selects some test cases and simulates execution of the code by hand.
- The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.
- Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective.
- These guidelines are based on personal experience, common sense, several other subjective factors.

## 2.2 Code Inspection

- During code inspection, the code is examined for the presence of some common **programming errors**.
- The principal aim of code inspection is to check common types of errors that usually occur in the code due to **programmer mistakes.**
- The inspection process has several beneficial side effects, other than finding errors.
- The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques.
- As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter.
- It is more likely that such an error can be discovered by specifically looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the code.
- In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.
- Following is a list of some classical programming errors which can be checked during code inspection:
  Use of uninitialised variables.
  Jumps into loops.
  Non-terminating loops.
  Incompatible assignments.
  Array indices out of bounds.
  Improper storage allocation and deallocation.
  Mismatch between actual and formal parameter in procedure calls.
  Use of incorrect logical operators or incorrect precedence among operators.
  Improper modification of loop variables.
  Comparison of equality of floating point values.

## 2.3 Clean Room Testing

- Clean room testing was implemented at IBM.
- This type of testing applies on walkthroughs, inspection, and formal verification.
- The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler.
- It is interesting to note that the term clean room was first coined at IBM by drawing design to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.
- This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.
- The main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time consuming for detecting all simple errors.

## 3. SOFTWARE DOCUMENTATION ✷

- When a software is developed,
  - the executable files and the source code,
  - several kinds of documents such as users' manual,
  - software requirements specification (SRS) document,
  - design document,
  - test document,

    - Installation manual, etc., are developed as part of the software engineering process.
- All these documents are considered a vital part of any good software development practice.
- Good documents are helpful in the following ways:
    - Good documents help enhance understandability of code.
    - Documents help the users to understand and effectively use the system.
    - Good documents help to effectively tackle the manpower turnover problem.
    - Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
    - Production of good documents helps the manager to effectively track the progress of the project. The project manager would know that some measurable progress has been achieved, if the results of some pieces of work have been documented and the same has been reviewed.
- Different types of software documents can broadly be classified into the following:
    - **Internal documentation:** These are provided in the source code itself.
    - **External documentation:** These are the supporting documents such as SRS document, installation document, user manual, design document, and test document. We discuss these two types of documentation in the next section.

## 3.1 Internal Documentation

- Internal documentation is the code comprehension features provided in the source code itself.
- Internal documentation can be provided in the code in several forms.
- The important types of internal documentation are the following:
    - Comments embedded in the source code.
    - Use of meaningful variable names.
    - Module and function headers.
    - Code indentation.
    - Code structuring (i.e., code decomposed into modules and functions).
    - Use of enumerated types.
    - Use of constant identifiers.
    - Use of user-defined data types.
- Careful experiments suggest that out of all types of internal documentation, meaningful variable names are most useful while trying to understand a piece of code.
- A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments.

## 3.2 External Documentation

- External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc.
- A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.
- An important feature that is required of any good external documentation is consistency with the code.
- If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the software.

# 4. TESTING ✶

- **Testing** is an important phase in software development.
- The aim of program testing is to help realize identify all defects in a program.
- Testing of professional software is carried out using a large number of test cases.
- It is usually the case that many of the different test cases can be executed in parallel by different team members.
- Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed.
- In a typical development organization, at any time, the maximum number of software engineers can be found to be engaged in testing activities.
- It is not very surprising then that in the software industry there is always a large demand for software test engineers.
- Now Software testers are looked upon as masters of specialized concepts, techniques, and tools.
- Testing a software product is as much challenging as initial development activities such as specifications, design, and coding.
- Moreover, testing involves a lot of creative thinking.

## 4.1 Basic Concepts and Terminologies

### How to test a program?

- Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected.
- If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction.
- A highly simplified view of program testing is schematically shown in Figure 10.1.

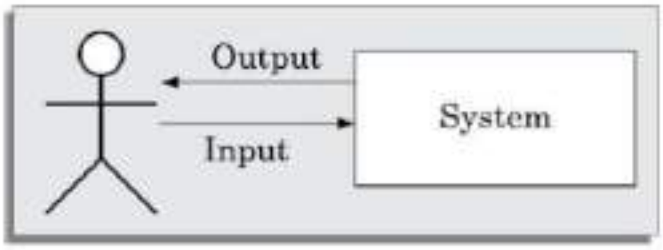


**Figure 10.1:** A simplified view of program testing.

- The tester has been shown as a stick icon, which inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs.
- Unless the conditions under which software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers.
- For examples, software might fail for a test case only when a network connection is enabled.

### Terminologies

- As is true for any specialized domain, the area of software testing has come to be associated with its own set of terminologies. In the following, we discuss a few important terminologies that have been standardized by the IEEE Standard Glossary of Software Engineering Terminology :
- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any

development activity. For example, during coding a programmer might commit the mistake of

- Not initializing a certain variable, or
- Might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation.
- Both these mistakes can lead to an incorrect result.

- An **error** is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. One example of an error is a call made to a wrong function. The terms error, fault, bug, and defect are considered to be synonyms in the area of program testing.
  **Example 10.2** Can a designer's mistake give rise to a program error? Give an example of a designer's mistake and the corresponding program error.
  **Answer:** Yes, a designer's mistake give rise to a program error. For example, a requirement might be overlooked by the designer, which can lead to it being overlooked in the code as well.
- A **failure** of a program essentially denotes an incorrect behavior exhibited by the program during its execution. An incorrect behavior is observed either as an incorrect result produced or as an inappropriate activity carried out by the program. Every failure is caused by some bugs present in the program. In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:
  – The result computed by a program is 0, when the correct result is 10.
  – A program crashes on an input.
  – A robot fails to avoid an obstacle and collides with it.
  It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.
  **Example 10.3** Give an example of a program error that may not cause any failure.
  **Answer:** Consider the following C program segment:
  In the above code, if the variable roll assumes zero or some negative value under some circumstances, then an array index out of bound type of error would result. However, it may be the case that for all allowed input values the variable roll is always assigned positive values. Then, the else clause is unreachable and no failure would occur. Thus, even if an error is present in the code, it does not show up as an error since it is unreachable for normal input values.
  **Explanation:** An array index out of bound type of error is said to occur, when the array index variable assumes a value beyond the array bounds.
- A **test case** is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its execution mode. As an example, consider the different execution modes of certain text editor software.
- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed.
- A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.

  - A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality.
  - A test case is said to be **negative test case**, if it is designed to test whether the software carries out something that is not required of the system.
  - As one example each of a positive test case and a negative test case, consider a program to manage user login.
  - A positive test case can be designed to check if a login system validates a user with the correct user name and password.
  - A negative test case in this case can be a test case that checks whether the login functionality validates and admits a user with wrong or bogus login user name or password.
- A **test suite** is the set of all tests that have been designed by a tester to test a given program.
- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.
  **Example 10.4** Suppose two programs have been written to implement essentially the same functionality. How can you determine which of these is more testable?
  **Answer:** A program is more testable, if it can be adequately tested with less number of test cases. Obviously, a less complex program is more testable. The complexity of a program can be measured using several types of metrics such as number of decision statements used in the program. Thus, a more testable program should have a lower structural complexity metric.
- A **failure mode** of software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.
- **Verification versus validation**
  **(Follow the Running Notes – Material has been given to all the students)**

## 4.2 Testing Activities

- Testing involves performing the following main activities:
- **Test suite design:** The set of test cases using which a program is to be tested is designed possibly using several test case design techniques. We discuss a few important test case design techniques later in this Chapter.
- **Running test cases and checking the results to detect failures:** Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.
- **Locate error:** In this activity, the failure symptoms are analyzed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

- **Error correction:** After the error is located during debugging, the code is appropriately changed to correct the error. The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed; the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.
  **Figure 10.2:** Testing process.

### 4.3 Testing in the Large versus Testing in the Small

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

Unit testing
Integration testing
System testing

- During unit testing, the individual functions (or units) of a program are tested.
- Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.
- After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing).
- Finally, the fully integrated system is tested (system testing).
- Integration and system testing are known as testing in the large.

## 5. UNIT TESTING ✶

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

### Driver and stub modules

- In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module.
- That is, besides the module under test, the following are needed to test the module:
  - o The procedures belonging to other modules that the module under test calls.
  - o Non-local data structures that the module accesses.
  - o A procedure to call the functions of the module under test with appropriate parameters.
- Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested.
- In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

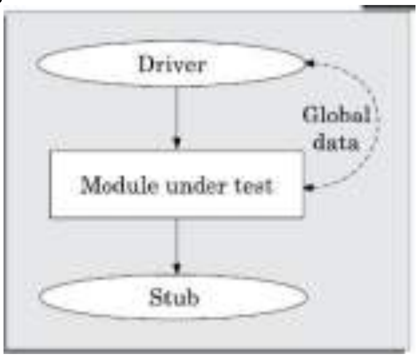


**Figure 10.3:** Unit testing with the help of driver and stub modules.

**Stub:** The role of stub and driver modules is pictorially shown in Figure 10.3.

A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behavior.

For example, a stub procedure may produce the expected behavior using a simple table look up mechanism.

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

## 6. BLACK-BOX TESTING ✶

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

In the following subsections, we will elaborate these two test case design techniques.

### 6.1 Equivalence Class Partitioning

- In the equivalence class partitioning approach, the domain of **input values** to the program under test is partitioned into a set of equivalence classes.
- The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.
- The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.
- Equivalence classes for a unit under test can be designed by examining the input data and output data.
- The following are two general guidelines for designing the equivalence classes:
- 1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.

### 6.2 Boundary Value Analysis

- A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs.
- The reason behind programmers committing such errors might purely be due to psychological factors.
- Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes.
- For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.
- Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.
- To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values.

- For those equivalence classes that are not a range of values (i.e., consist of a discrete collection of values) no boundary value test cases can be defined.
- For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0, 1, 10, 11}.
- **Example 10.9** For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.
  **Answer:** There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1, 5000, 5001}.

## 7. WHITE-BOX TESTING ✷

- White-box testing is an important type of unit testing.
- A large number of white-box testing strategies exist.
- Each testing strategy essentially designs test cases based on **analysis of some aspect of source code** and is based on some heuristic.

### 7.1 Basic Concepts

- A white-box testing strategy can either be coverage-based or fault based.
- **Fault-based testing:** A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitute the **fault model** of the strategy.
- **Coverage-based testing:** A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.
- **Testing criterion for coverage-based testing:** A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures. The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.
- **Stronger versus weaker testing:** compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.
  - A white-box testing strategy is said to be **stronger** than another strategy, if the stronger testing strategy covers **all program elements** covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.
  - When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies.
  - The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa. If a stronger testing has been performed, then a weaker testing need not be carried out.

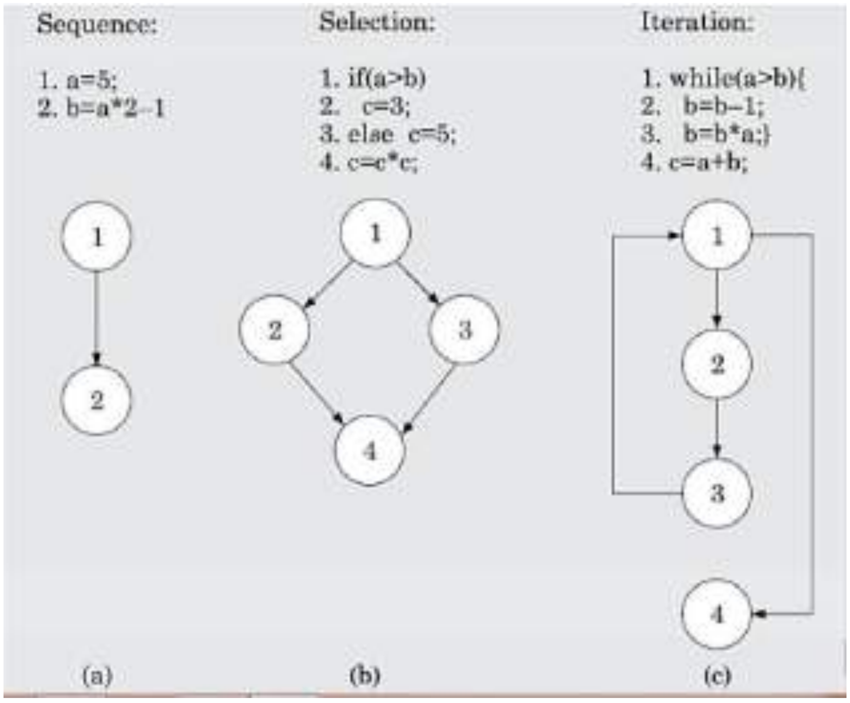


**Figure 10.6:** Illustration of stronger, weaker, and complementary testing strategies.

A test suite should, however, be enriched by using various complementary testing strategies.

## 7.2 Statement Coverage

- The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.
- **Example 10.11** Design statement coverage-based test suite for the following Euclid's GCD computation program:

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

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

## 7.3 Branch Coverage

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.
- Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

**Example 10.12** For the program of Example 10.11, determine a test suite to achieve branch coverage.
**Answer:** The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage. It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

## 7.4 Multiple Condition Coverage

- In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2 ).or.c3).
- A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

  **Example 10.13** Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.
  **Answer:** Consider the following C program segment:

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

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

## 7.5 Path Coverage

- A test suite achieves path coverage if it executes each linearly independent paths ( o r basis paths ) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

### Control flow graph (CFG)

- A control flow graph describes how the control flows through the program. We can define a control flow graph as the following: A control flow graph describes the sequence in which the different instructions of a program get executed. In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph (see Figure 10.5).

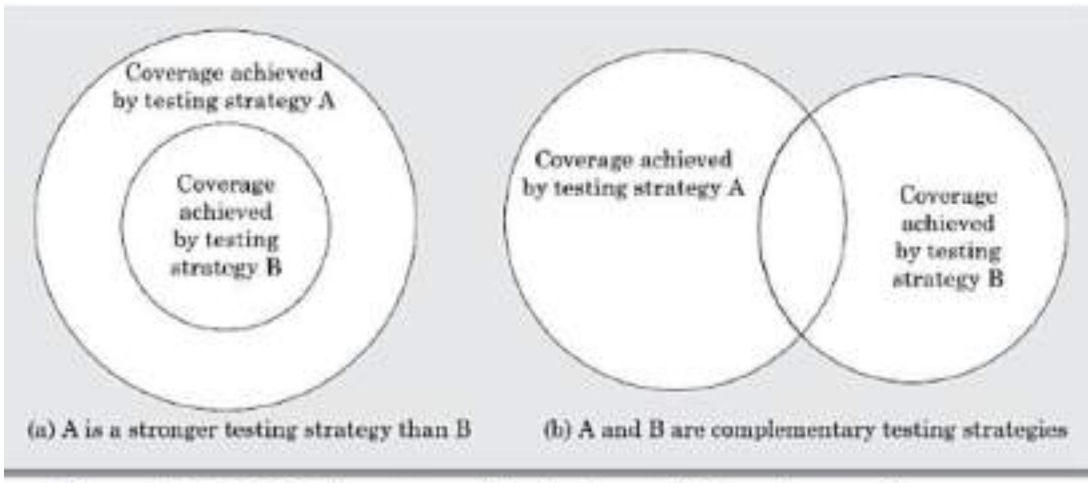
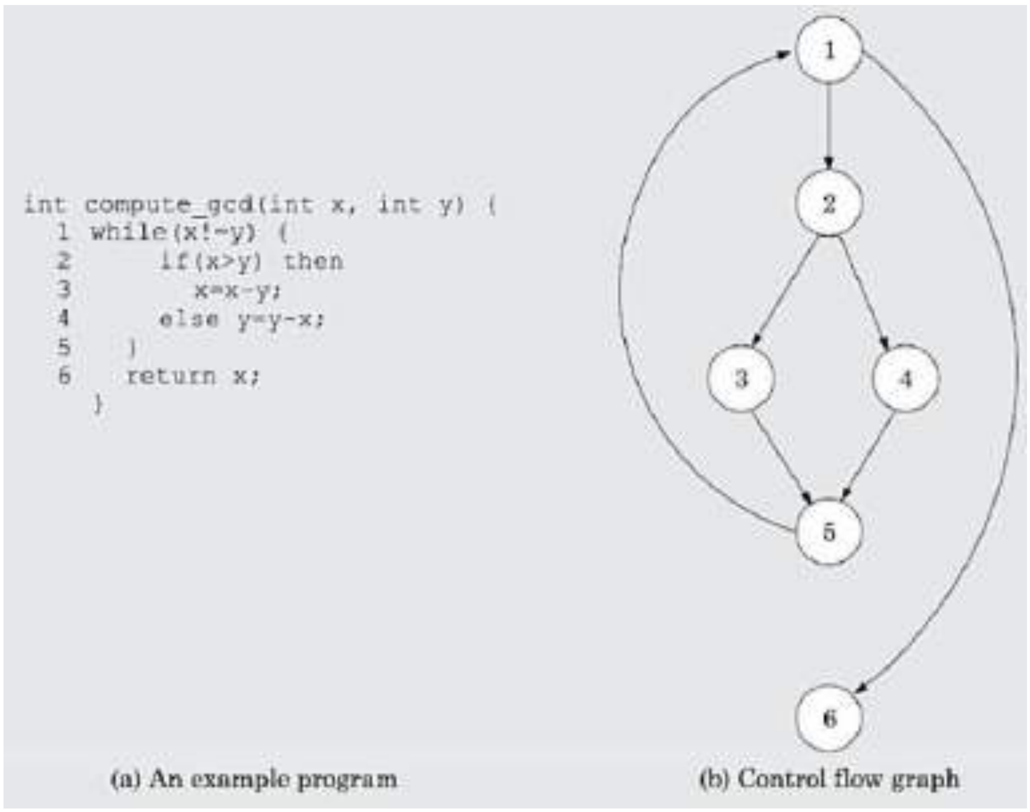

Coverage achieved by testing strategy A
Coverage achieved by testing strategy B
(a) A is a stronger testing strategy than B
Coverage achieved by testing strategy A
Coverage achieved by testing strategy B
(b) A and B are complementary testing strategies

**Figure 10.5:** CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

- There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.
- More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.
- We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only. Figure 10.5 summarizes how the CFG for these three types of constructs can be drawn.
- The CFG representation of the sequence and decision types of statements is straight forward. Please note carefully how the CFG for the loop (iteration) construct can be drawn.
- For iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop.
- That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop. Using these basic ideas, the CFG of the program given in Figure 10.7(a) can be drawn as shown in Figure 10.7(b).



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

**Figure 10.7:** Control flow diagram of an example program.

**Path**

- A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Please note that a program can have more than one terminal node when it contains multiple exits or return type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops.
- For example, in Figure 10.5(c), there can be an infinite number of paths such as 12314, 12312314, 12312312314, etc. If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent p a t h s ( o r basis paths ). Let us now discuss what linearly independent paths are and how to determine these in a program.

**Linearly independent set of paths (or basis path set)**

- A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set.
- If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.
- According to the above definition of a linearly independent set of paths, for any path in the set, its sub path cannot be a member of the set.

## 7.6 McCabe's Cyclomatic Complexity Metric

- McCabe obtained his results by applying graph-theoretic techniques to the control flow graph of a program. McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

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

  $V(G) = E – N + 2$

  Where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph. For the CFG of example shown in Figure 10.7, E = 7 and N = 6. Therefore, the value of the Cyclomatic complexity = 7 – 6 + 2 = 3.

**Steps to carry out path coverage-based testing**

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

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

  Test using a randomly designed set of test cases. Perform dynamic analysis to check the path coverage achieved. until at least 90 per cent path coverage is achieved.

## 7.7 Data Flow-based Testing

- Data flow based testing method selects test paths of a program according to the definitions and uses of different variables in a program. Consider a program P. For a statement numbered S of P , let DEF(S) = {X /statement S contains a definition of X } and USES(S)= {X /statement S contains a use of X }
- For the statement S: a=b+c; DEF(S) = {a}, USES(S) = {b, c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X. All definitions criterion is a test coverage criterion that requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition.

## 7.8 Mutation Testing

- All white-box testing strategies that we have discussed so far are coverage-based testing techniques.
- In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program.
- In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed.
- After the initial testing is complete, mutation testing can be taken up.
- The idea behind mutation testing is to make a few arbitrary changes to a program at a time.
- Each time the program is changed, it is called a mutated program and the change effected is called a mutant.
- An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of simple errors.
- A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement.
- A mutant may or may not cause an error in the program.
- If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.

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

## 8. DEBUGGING ✶

- After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed. In this Section, we shall summarize the important approaches that are available to identify the error locations. Each of these approaches has its own advantages and disadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

### 8.1 Debugging Approaches

- The following are some of the approaches that are popularly adopted by the programmers for debugging:

#### Brute force method

- This is the most common method of debugging but is the least efficient method.
- In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.
- Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

#### Backtracking

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

#### Cause elimination method

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

#### Program slicing

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

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

## 9. PROGRAM ANALYSIS TOOLS ✷

- A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, adequacy of testing, etc.
- Program analysis tools are classified into the following two broad categories:
  - Static analysis tools
  - Dynamic analysis tools
- These two categories of program analysis tools are discussed in the following

### 9.1 Static Analysis Tools

- Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyze the source code to compute certain metrics characterizing the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions. These also check the conformance of the code with the prescribed coding standards. In this context, it displays the following analysis results:
- To what extent the coding standards have been adhered to?
- Whether certain programming errors such as uninitialized variables, mismatch between actual and formal parameters, variables that are declared but never used, etc., exist?
- A list of all such errors is displayed.

### 9.2 Dynamic Analysis Tools

- Dynamic program analysis tools can be used to evaluate several program characteristics based on an analysis of the run time behavior of a program.
- These tools usually record and analyze the actual behavior of a program while it is being executed.
- A dynamic program analysis tool (also called a dynamic analyzer) usually collects execution trace information by instrumenting the code.
- Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program.
- The instrumented code when executed, records the behavior of the software for different test cases.
- An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.
- After software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the coverage that has been achieved by the complete test suite for the program.

## 10. INTEGRATION TESTING ✷

- Integration testing is carried out after all (or at least some of ) the modules have been unit tested.
- Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters).
- For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module.

- Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module.
- The objective of integration testing is to check whether the different modules of a program interface with each other properly.
- During integration testing, different modules of a system are integrated. During integration testing, different modules of a system are integrated in a planned manner using an integration plan.
- The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested.
- An important factor that guides the integration plan is the module dependency graph.
- Thus, by examining the structure chart, the integration plan can be developed.
- Any one (or a mixture) of the following approaches can be used to develop the test plan:
    - Big-bang approach to integration testing.
    - Bottom-up approach to integration testing Mixed (also called sandwiched) approach to integration testing.
    - Top-down approach to integration testing

## Big-bang approach to integration testing

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

## Bottom-up approach to integration testing

- Large software products are often made up of several subsystems.
- A subsystem might consist of many modules which communicate among each other through well-defined interfaces.
- In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.
- The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily.
- The test cases must be carefully chosen to exercise the interfaces in all possible manners.
- In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems.
- The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously.
- Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level

modules do I/O and other critical functions, testing the low-level modules thoroughly increase the reliability of the system.

- A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

**Top-down approach to integration testing**

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

**Mixed approach to integration testing**

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

### 10.1 Phased versus Incremental Integration Testing

- Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

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

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

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

## 11. TESTING OBJECT-ORIENTED PROGRAMS ✶

- During the initial years of object-oriented programming, it was believed that object-orientation would, to a great extent, reduce the cost and effort incurred on testing.

- This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc., thereby chances of errors in the code is minimized.
- However, it was soon realized that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs.
- The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that are present in procedural programs.
- Therefore additional test cases are needed to be designed to detect these. We examine these issues as well as some other basic issues in testing object-oriented programs in the following subsections.

## 11.1 What is a Suitable Unit for Testing Object-oriented Programs?

- For procedural programs, we had seen that procedures are the basic units of testing.
- That is, first all the procedures are unit tested. Then various tested procedures are integrated together and tested.
- Thus, as far as procedural programs are concerned, procedures are the basic units of testing.
- Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing?

## 11.2 Do Various Object-orientation Features Make Testing Easy?

- The implications of different object-orientation features in testing are as follows.
- **Encapsulation:** The encapsulation feature helps in data abstraction, error isolation, and error prevention.
    - Encapsulation prevents the tester from accessing the data internal to an object.
    - The encapsulation feature though makes testing difficult, the difficulty can be overcome to some extent through use of appropriate state reporting methods.
- **Inheritance:** The inheritance feature helps in code reuse and was expected to simplify testing.
    - It was expected that if a class is tested thoroughly, then the classes that are derived from this class would need only incremental testing of the added features.
    - However, this is not the case. Even if the base class class has been thoroughly tested, the methods inherited from the base class need to be tested again in the derived class.
    - The reason for this is that the inherited methods would work in a new context (new data and method definitions).
    - As a result, correct behaviour of a method at an upper level, does not guarantee correct behaviour at a lower level. Therefore, retesting of inherited methods needs to be followed as a rule, rather as an exception.
- **Dynamic binding:** Dynamic binding was introduced to make the code compact, elegant, and easily extensible. However, as far as testing is concerned all possible bindings of a method call have to be identified and tested. This is not easy since the bindings take place at run-time.
- **Object states:** In contrast to the procedures in a procedural program, objects store data permanently. As a result, objects do have significant states. The behaviour of an

object is usually different in different states. That is, some methods may not be active in some of its states. Also, a method may act differently in different states.

### 11.3 Grey-Box Testing of Object-oriented Programs

- Model-based testing is important for object oriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.
- For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs.
- These are called the grey-box test cases.
- The following are some important types of grey-box testing that can be carried on based on UML models:
- **State-model-based testing**
  **State coverage:** Each method of an object is tested at each state of the object.
  **State transition coverage:** It is tested whether all transitions depicted in the state model work satisfactorily.
  **State transition path coverage:** All transition paths in the state model are tested.
- **Use case-based testing**
  **Scenario coverage:** Each use case typically consists of a mainline scenario and several alternate scenarios. For each use case, the mainline and all alternate sequences are tested to check if any errors show up.
- **Class diagram-based testing**
  **Testing derived classes:** All derived classes of the base class have to be instantiated and tested. In addition to testing the new methods defined in the derived class, the inherited methods must be retested.
  **Association testing:** All association relations are tested.
  **Aggregation testing:** Various aggregate objects are created and tested.
  **Sequence diagram-based testing**
  **Method coverage:** All methods depicted in the sequence diagrams are covered. **Message path coverage:** All message paths that can be constructed from the sequence diagrams are covered.

### 11.5 Integration Testing of Object-oriented Programs

There are two main approaches to integration testing of object-oriented programs:
• Thread-based
• Use based

- **Thread-based approach:** In this approach, all classes that need to collaborate to realize the behavior of a single use case are integrated and tested. After all the required classes for a use case are integrated and tested, another use case is taken up and other classes (if any) necessary for execution of the second use case to run are integrated and tested. This is continued till all use cases have been considered.
- **Use-based approach:** Use-based integration begins by testing classes that either needs no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

## 12. SYSTEM TESTING ✶

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

- The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation (procedural or object oriented) is immaterial. There are essentially three main kinds of system testing depending on who carries out testing:
- 1. **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- 2. **Beta Testing:** Beta testing is the system testing performed by a select group of friendly customers.
- 3. **Acceptance Testing:** Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

## 12.1 Smoke Testing

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

## 12.2 Performance Testing

- Performance testing is an important type of system testing.
- Performance testing is carried out to check whether the system meets the nonfunctional requirements identified in the SRS document.
- There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the type of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document.
- All performance tests can be considered as black-box tests.

### Stress testing

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

### Volume testing

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

### Configuration testing

- Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements.
- Sometimes systems are built to work in different configurations for different users.
- For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing.

- The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.

**Compatibility testing**

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

**Regression testing**

- This type of testing is required when software is maintained to fix some bugs or enhance functionality, performance, etc. Regression testing is also discussed in Section 10.13.

**Recovery testing**

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

**Maintenance testing**

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

**Documentation testing**

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

**Usability testing**

- Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested. A GUI being just being functionally correct is not enough. Therefore, the GUI has to be checked against the checklist we discussed in Sec. 9.5.6.

**Security testing**

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

## 12.3 Error Seeding

- Sometimes customers specify the maximum number of residual errors that can be present in the delivered software. These requirements are often expressed in terms of maximum number of allowable errors per line of source code. The error seeding technique can be used to estimate the number of residual errors in software. Error seeding, as the name implies, it involves seeding the code with some known errors. In other words, some artificial errors are introduced (seeded) into the program. The number of these seeded errors that are detected in the course of standard testing is determined. These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program:

  - The number of errors remaining in the product.
  - The effectiveness of the testing strategy.
  - Let N be the total number of defects in the system, and let n of these defects be found by testing.
  - Let S be the total number of seeded defects, and let s of these defects be found during testing.

- Therefore, we get: Defects still remaining in the program after testing can be given by: Error seeding works satisfactorily only if the kind seeded errors and their frequency of occurrence matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in software. To some extent, the different categories of errors that are latent and their frequency of occurrence can be estimated by analyzing historical data collected from similar projects. That is, the data collected is regarding the types and the frequency of latent errors for all earlier related projects.
- This gives an indication of the types (and the frequency) of errors that are likely to have been committed in the program under consideration. Based on these data, the different types of errors with the required frequency of occurrence can be seeded.

## 13. SOME GENERAL ISSUES ASSOCIATED WITH TESTING ✶

- In this section, we shall discuss two general issues associated with testing. These are—how to document the results of testing and how to perform regression testing.

### Test documentation

- A piece of documentation that is produced towards the end of testing is the test summary report.
- This report normally covers each subsystem and represents a summary of tests which have been applied to the subsystem and their outcome.
- It normally specifies the following: What is the total number of tests that were applied to a subsystem. Out of the total number of tests how many tests were successful.
- How many were unsuccessful, and the degree to which they were unsuccessful, e.g., whether a test was an outright failure or whether some of the expected results of the test were actually observed.

### Regression testing

- Regression testing spans unit, integration, and system testing. Instead, it is a separate dimension to these three forms of testing.
- Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix.
- However, if only a few statements are changed, then the entire test suite need not be run — only those test cases that test the functions and are likely to be affected by the change need to be run.
- Whenever software is changed to either fix a bug, or enhance or remove a feature, regression testing is carried out.

# SOFTWARE ENGINEERING UNIT-5

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING.

Dr DEEPAK NEDUNURI

[ SIR C R REDDY COLLEGE OF ENGINEERING ]

ELURU.

# SOFTWARE ENGINEERING
# UNIT-5

1. Software Reliability And Quality Management: Software Reliability,
2. Statistical Testing,
3. Software Quality,
4. Software Quality Management System,
5. ISO 9000,
6. SEI Capability Maturity Model.
7. Computer Aided Software Engineering: Case and its Scope,
8. Case Environment,
9. Case Support in Software Life Cycle,
10. Other Characteristics of Case Tools,
11. Towards Second Generation CASE Tool,
12. Architecture of a Case Environment

## SOFTWARE RELIABILITY AND QUALITY MANAGEMENT

- Reliability of a software product is an important concern for most users.
- Users not only want the products they purchase to be highly reliable product.
- This may especially be true for safety-critical and embedded software products.
- It is very difficult to accurately measure the reliability of any software product.
- One of the main problems encountered while quantitatively measuring the reliability of a software product is the fact that reliability is observer-dependent. That is, different groups of users may arrive at different reliability estimates for the same product.
- Besides this, several other problems (such as frequently changing reliability values due to bug corrections) make accurate measurement of the reliability of a software product difficult.
- Even though no entirely satisfactory metric to measure the reliability of a software product exists,
- Software quality assurance (SQA) has emerged as one of the most talked about topics in recent years in software industry circle.
- The major aim of SQA is to help an organization **develop high quality software products** in a repeatable manner.
- A software development organization can be called *repeatable* when its software development process is person-independent. That is, the success of a project does not depend on who exactly are the team members of the project.
- Besides, the quality of the developed software and the cost of development are important issues addressed by SQA.

### 1. SOFTWARE RELIABILITY

- The reliability of a software product essentially denotes its *trustworthiness or dependability*.
- **The reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.**
- **Intuitively**, it is obvious that a **software product** having a large number of defects **is unreliable**.
- It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced.
- It would have been very nice if we could mathematically characterize this relationship between reliability and the number of bugs present in the system using a simple closed form expression.
- Unfortunately, it is very difficult to characterize the observed reliability of a system in terms of the number of latent defects in the system using a simple mathematical expression.
- Removing errors from those parts of a software product that are very infrequently executed, makes little difference to the perceived reliability of the product.
- It has been experimentally observed by analyzing the behavior of a large number of programs that 90 per cent of the execution time of a typical program is spent in executing only 10 per cent of the instructions in the program.
- The *most used* 10 per cent instructions are often called the *core*1 of a program.
- The rest 90 per cent of the program statements are called *non-core*.
- Reliability also depends upon how the product is used, or on its *execution profile*.

- If the users execute only those features of a program that are “correctly” implemented, none of the errors will be exposed and the perceived reliability of the product will be high.
- if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low.
- Different categories of users of a software product typically execute different functions of a software product.
- Software reliability more difficult to measure than hardware reliability: The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

**1.1 Hardware versus Software Reliability**

- Hardware components fail due to very different reasons as compared to software components.
- Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.
- A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part.
- In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug.
- For this reason, when a hardware part is repaired its reliability would be maintained at the level that existed before the failure occurred;
- whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug fix introduces new errors).
- To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, the inter-failure times remain constant).
- On the other hand, the aim of software reliability study would be reliability growth (that is, increase in inter-failure times).
- A comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product is sketched in Figure 11.1.

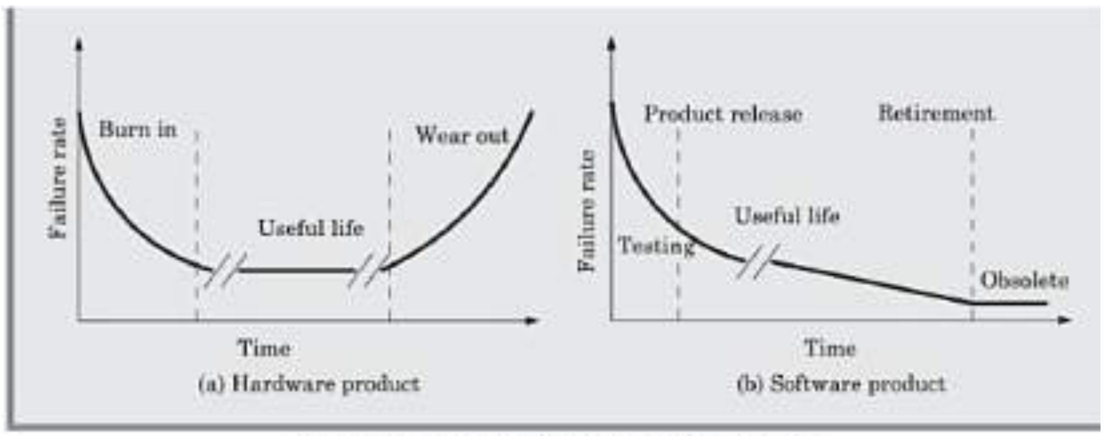


**Figure 11.1:** Change in failure rate of a product.

- Observe that the plot of change of reliability with time for a hardware component (Figure 11.1(a)) appears like a “bath tub”. For a software component the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced.

- The system then enters its useful life, where the rate of failure is almost constant.
- After some time (called product life time) the major components wear out, and the failure rate increases.
- The initial failures are usually covered through manufacturer's warranty.
- That is, one need not feel happy to buy a ten year old car at one tenth of the price of a new car, since it would be near the rising edge of the bath tub curve, and one would have to spend unduly large time, effort, and money on repairing and end up as the loser.
- In contrast to the hardware products, the software product shows the highest failure rate just after purchase and installation (see the initial portion of the plot in Figure 11.1 (b)).
- As the system is used, more and more errors are identified and removed resulting in reduced failure rate.
- This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no more error correction occurs and the failure rate remains unchanged.

## 1.2 Reliability Metrics of Software Products

- The reliability requirements for different categories of software products may be different.
- For this reason, it is necessary that the level of reliability required for a software product should be specified in the software requirements specification (SRS) document.
- In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product.
- A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has.
- Six metrics that show the reliability as follows:
- **Rate of occurrence of failure (ROCOF):** ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.
- **Mean time to failure (MTTF):** MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures.
    - Let the failures occur at the time instants t1, t2 ... tn. Then, MTTF can be calculated as. It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc. is not taken into account in the time measurements and the clock is stopped at these times.
- **Mean time to repair (MTTR):** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
- **Mean time between failure (MTBF):** The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF

- **Probability of failure on demand (POFOD):** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made.
    - For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.
    - POFOD metric is very appropriate for software products that are not required to run continuously.
- **Availability:** Availability of a system is a measure of how likely would the system is available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.
    - This metric is important for systems such as telecommunication systems, and operating systems, and embedded controllers, etc. which are supposed to be never down and where repair and restart time are significant and loss of service during that time cannot be overlooked.

**Shortcomings of reliability metrics of software products:**

- All the above reliability metrics suffer from several shortcomings as far as their use in software reliability measurement is concerned.
    - One of the reasons is that these metrics are centered on the probability of occurrence of system failures but take no account of the consequences of failures.
    - That is, these reliability models do not distinguish the relative severity of different failures.
    - Failures which are transient and whose consequences are not serious are in practice of little concern in the operational use of a software product.

- A scheme of classification of failures is as follows:

**Transient:** Transient failures occur only for certain input values while invoking a function of the system.
**Permanent:** Permanent failures occur for all input values while invoking a function of the system.
**Recoverable:** When a recoverable failure occurs, the system can recover without having to shutdown and restart the system (with or without operator intervention).
**Unrecoverable:** In unrecoverable failures, the system may need to be restarted.
**Cosmetic:** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the situation **where the mouse button has to be clicked twice instead of once to invoke a given function** through the graphical user interface.

**1.3 Reliability Growth Modelling**

- A reliability growth model **is a mathematical model** of how software reliability improves as errors are detected and repaired.
- A reliability growth model **can be used to predict when a particular level of reliability** is likely to be attained.
- Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.
- Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.
- **Jelinski and Moranda model :** The simplest reliability growth model is a **step function model** where it is assumed that the reliability increases by a constant

increment each time an error is detected and repaired. Such a model is shown in Figure 11.2.

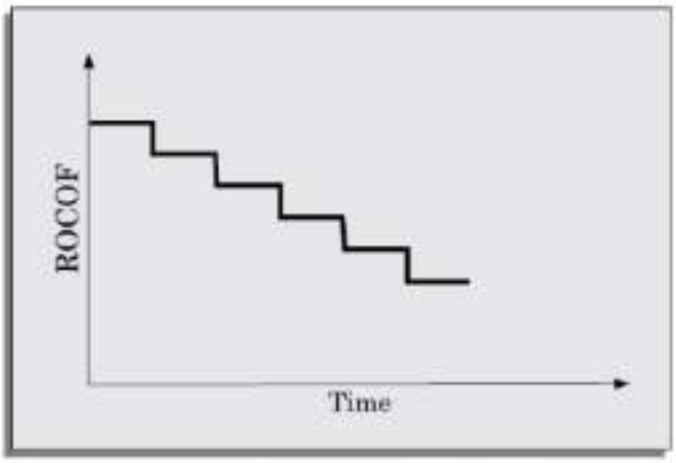


**Figure 11.2:** Step function model of reliability growth.

However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth which is highly unrealistic since we already know that correction of different errors contribute differently to reliability growth.

**Figure 11.2:** Step function model of reliability growth.

- **Littlewood and Verall's model :** This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors.
    - It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases.
    - It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution.
    - This distribution models the fact that error corrections with large contributions to reliability growth are removed first.
    - This represents diminishing return as test continues.

## 2. STATISTICAL TESTING

- Statistical testing is a testing process whose objective is to determine the reliability of the product rather than discovering errors.
- The test cases designed for statistical testing with an entirely different objective from those of conventional testing. To carry out statistical testing, we need to first define the operation profile of the product.
- **Operation profile:** Different categories of users may use a software product for very different purposes.
    - For example, a librarian might use the Library Automation Software to create member records, delete member records, add books to the library, etc.,
    - whereas a library member might use software to query about the availability of a book, and to issue and return books.
    - Formally, we can define the operation profile of a software as the probability of a user selecting the different functionalities of the software.
    - If we denote the set of various functionalities offered by the software by {fi}, the operational profile would associate with each function {fi} with the probability with which an average user would select {fi} as his next function

to use. Thus, we can think of the operation profile as assigning a probability value pi to each functionality fi of the software.

- **How to define the operation profile for a product? :** We need to divide the input data into a number of input classes.
    - For example, for graphical editor software, we might divide the input into data associated with the edit, print, and file operations.
    - We then need to assign a probability value to each input class; to signify the probability for an input value from that class to be selected.
    - The operation profile of a software product can be determined by observing and analyzing the usage pattern of the software by a number of users.

**2.1 Steps in Statistical Testing**

- The first step is to determine the operation profile of the software.
- The next step is to generate a set of test data corresponding to the determined operation profile.
- The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.
- For accurate results, statistical testing requires some fundamental assumptions to be satisfied.
- It requires a statistically significant number of test cases to be used.

- **Pros and cons of statistical testing:** Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used.
- Therefore, it results in a system that the users can find to be more reliable (than actually it is!).
- Also, the reliability estimation arrived by using statistical testing is more accurate compared to those of other methods discussed.
- However, it is not easy to perform the statistical testing satisfactorily due to the following two reasons. There is no simple and repeatable way of defining operation profiles. Also, the number of test cases with which the system is to be tested should be statistically significant.

## 3. SOFTWARE QUALITY

- Traditionally, **the quality of a product is defined in terms of its fitness of purpose**.
- That is, a good quality product does exactly what the users want it to do, since for almost every product, fitness of purpose is interpreted in terms of satisfaction of the requirements laid down in the SRS document.
- Although "fitness of purpose" is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc.—"fitness of purpose" is not a wholly satisfactory definition of quality for software products.
- To give an example of why this is so, consider a software product that is functionally correct. That is, it correctly performs all the functions that have been specified in its SRS document.
- Even though it may be functionally correct, we cannot consider it to be a quality product, if it has an almost unusable user interface.
- The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:

- **Portability:** A software product is said to be portable, if it can be easily made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.
- **Usability:** A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.
- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

**McCall's quality factors:**

- McCall distinguishes two levels of quality attributes [McCall]. The higher level attributes, known as quality factor s or external attributes can only be measured indirectly.
- The second-level quality attributes are called quality criteria.
- Quality criteria can be measured directly, either objectively or subjectively.
- By combining the ratings of several criteria, we can either obtain a rating for the quality factors, or the extent to which they are satisfied.
- For example, the reliability cannot be measured directly, but by measuring the number of defects encountered over a period of time. Thus, reliability is a higher-level quality factor and number of defects is a low-level quality factor.

**ISO 9126:**

- ISO 9126 defines a set of hierarchical quality characteristics.

## 4. <u>SOFTWARE QUALITY MANAGEMENT SYSTEM</u>

- A quality management system (often referred to as quality system) is the principal methodology used by organisations to ensure that the products they develop have the desired quality.
- In the following, some of the important issues associated with a quality system:
- **Managerial structure and individual responsibilities :** A quality system is the responsibility of the organisation as a whole. However, every organisation has a separate quality department to perform several quality system activities. The quality system of an organisation should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.
- **Quality system activities :** The quality system activities encompass the following:
    - Auditing of projects to check if the processes are being followed.
    - Collect process and product metrics and analyze them to check if quality goals are being met.
    - Review of the quality system to make it more effective.
    - Development of standards, procedures, and guidelines.
    - Produce reports for the top management summarizing the effectiveness of the quality system in the organization.
    - A good quality system must be well documented.

## 4.1 Evolution of Quality Systems:

- Quality systems have rapidly evolved over the last six decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products.
    - For example, a company manufacturing nuts and bolts would inspect its finished goods and would reject those nuts and bolts that are outside certain specified tolerance range.
    - Since that time, quality systems of organizations have undergone four stages of evolution as shown in Figure 11.3.
    - The initial product inspection method gave way to quality control (QC) principles. Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

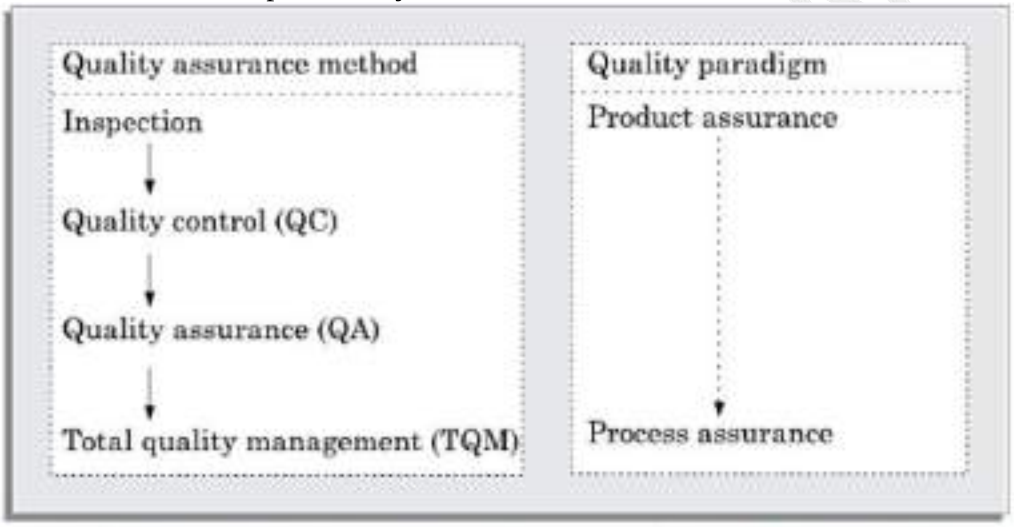


**Figure 11.3:** Evolution of qualty system and corresponding shft in the qualty paradigm.

- Thus, quality control aims at correcting the causes of errors and not just rejecting the defective products.
- The next breakthrough in quality systems was the development of the quality assurance (QA) principles.
- The basic premise of modern quality assurance is that if an organization's processes are good then the products are bound to be of good quality.
- The modern quality assurance paradigm includes guidance for recognizing, defining, analyzing, and improving the production process.
- Total quality management (TQM) advocates that the process followed by an organization must continuously be improved through process measurements.
- TQM goes a step further than quality assurance and aims at continuous process improvement.
- TQM goes beyond documenting processes to optimizing them through redesign.

## 4.2 Product Metrics versus Process Metrics:

- All modern quality systems lay emphasis on collection of certain product and process metrics during product development.
- Let us first understand the basic differences between product and process metrics.
- Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.

# 5. <u>ISO 9000</u>

- International Organization for Standards (ISO) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987.

## 5.1 What is ISO 9000 Certification? :

- ISO 9000 certification serves as a reference for contract between independent parties.
- In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether the vendor has obtained ISO 9000 certification or not.
- In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system.
- We have already seen that the quality system of an organization applies to all its activities related to its products or services.
- The ISO standard addresses both operational aspects (that is, the process) and organizational aspects such as responsibilities, reporting, etc.
- It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.
- ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO 9003.
- The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.

The types of software companies to which the different ISO standards apply are as follows:

- **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.
- **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in production.
- **ISO 9003:** This standard applies to organizations involved only in installation and testing of products.

## 5.2 ISO 9000 for Software Industry

- ISO 9000 is a generic standard that is applicable to big industries, starting from a steel manufacturing industry to a service rendering company.
- Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very difficult to interpret them in the context of software development organizations.
- But in every big industries software plays very important role.
- So ISO 9000 is applicable to Software Industries.

## 5.3 Why Get ISO 9000 Certification?

Some of the benefits that accrue to organizations obtaining ISO certification:

- Confidence of customers in an organization increases when the organization qualifies for ISO 9001 certification.
- This is especially true in the international market.
- In fact, many organizations awarding international software development contracts insist that the development organization have ISO 9000 certification.
- For this reason, it is vital for software organizations involved in software export to obtain ISO 9000 certification.

- ISO 9000 requires a well-documented software production process to be in place.
- A well- documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient, and cost effective.
- ISO 9000 certification points out the weak points of organizations and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and TQM.

**5.4 How to Get ISO 9000 Certification?**

- An organization intending to obtain ISO 9000 certification applies to a ISO 9000 registrar for registration.
- The ISO 9000 registration process consists of the following stages:
- **Application stage:** Once an organization decides to go for ISO 9000 certification, it applies to a registrar for registration.
- **Pre-assessment:** During this stage the registrar makes a rough assessment of the organization.
- **Document review and adequacy audit:** During this stage, the registrar reviews the documents submitted by the organization and makes suggestions for possible improvements.
- **Compliance audit:** During this stage, the registrar checks whether the suggestions made by it during review have been complied to by the organization or not.
- **Registration:** The registrar awards the ISO 9000 certificate after successful completion of all previous phases.
- **Continued surveillance:** The registrar continues monitoring the organization periodically.
- ISO mandates that a certified organization can use the certificate for corporate advertisements but cannot use the certificate for advertising any of its products. This is probably due to the fact that the ISO 9000 certificate is issued for an organization's process and not to any specific product of the organization.

**5.5 Salient Features of ISO 9001 Requirements**

- Salient features all the requirements as follows:
- **Document control:** All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.
- **Planning:** Proper plans should be prepared and then progress against these plans should be monitored.
- **Review:** Important documents across all phases should be independently checked and reviewed for effectiveness and correctness.
- **Testing:** The product should be tested against specification.
- **Organizational aspects:** Several organizational aspects should be addressed e.g., management reporting of the quality team.

**5.6 ISO 9000-2000**

- ISO revised the quality standards in the year 2000 to fine tune the standards.
- The major changes include a mechanism for continuous process improvement.
- There is also an increased emphasis on the role of the top management, including establishing measurable objectives for various roles and levels of the organization. The new standard recognizes that there can be many processes in an organization.

## 6. SEI CAPABILITY MATURITY MODEL

- **"SEI CMM - Software Engineering Institute Capability Maturity Model "** was proposed by Software Engineering Institute of the Carnegie Mellon University, USA.
- CMM is patterned after the pioneering work of Philip Crosby who published his maturity grid of five evolutionary stages in adopting quality practices in his book "Quality is Free" [Crosby79].
- The Unites States Department of Defence (US DoD) is the largest buyer of software product.
- It often faced difficulties in vendor performances, and had to many times live with low quality products, late delivery, and cost escalations. In this context, SEI CMM was originally developed to assist the U.S. Department of Defense (DoD) in software acquisition.
- Most of the major DoD contractors began CMM-based process improvement initiatives as they vied for DoD contracts.
- It was observed that the SEI CMM model helped organizations to improve the quality of the software they developed and therefore adoption of SEI CMM model had significant business benefits.
- Gradually many commercial organizations began to adopt CMM as a framework for their own internal improvement initiatives.
- SEI CMM classifies software development industries into the following five maturity levels:

**Level 1: Initial**

- A software development organization at this level is characterized by ad hoc activities.

**Level 2: Repeatable**

- At this level, the basic project management practices such as tracking cost and schedule are established. Configuration management tools are used on items identified for configuration control. Size and cost estimation techniques such as function point analysis are used.

**Level 3: Defined**

- At this level, the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities.

**Level 4: Managed**

- At this level, the focus is on software metrics. Both process and product metrics are collected. Quantitative quality goals are set for the products and at the time of completion of development.

**Level 5: Optimizing**

- At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement.
-

### 6.1 Comparison between ISO 9000 Certification and SEI/CMM

- Let us compare some of the key characteristics of ISO 9000 certification and the SEI CMM model for quality appraisal:
- ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organization in official documents, communication with external parties, and in tender quotations.

- However, SEI CMM assessment is purely for internal use. SEI CMM was developed specifically for software industry and therefore addresses many issues which are specific to software industry alone.
- Thus, it provides a way for achieving gradual quality improvement. In contrast, an organization adopting ISO 9000 either qualifies for it or does not qualify.

**6.2 Capability Maturity Model Integration (CMMI)**

- Capability maturity model integration (CMMI) is the successor of the capability maturity model (CMM).
- The CMM was developed from 1987 until 1997. In 2002, CMMI Version 1.1 was released. Version 1.2 followed in 2006.
- CMMI aimed to improve the usability of maturity models by integrating many different models into one framework.
- After CMMI was first released in 1990, it was adopted and used in many domains. For example, CMMs were developed for disciplines such as systems engineering (SE-CMM), people management (PCMM), software acquisition (SA-CMM), and others.
- Although many organizations found these models to be useful, they also struggled with problems caused by overlap, inconsistencies, and integrating the models.
- In this context, CMMI is generalized to be applicable to many domains.
- For example, the word "software" does not appear in definitions of CMMI. This unification of various types of domains into a single model makes CMMI extremely abstract. The CMMI, like its predecessor, describes five distinct levels of maturity.

# 7. COMPUTER AIDED SOFTWARE ENGINEERING (CASE)

- CASE tools help to improve software development effort and maintenance effort.
- CASE has emerged as a much talked topic in software industries.
- Software is becoming the costliest component in any computer installation. Even though hardware prices keep dropping like never and falling below even the most optimistic expectations, software prices are becoming costlier due to increased manpower costs.
- CASE tools promise effort and cost reduction in software development and maintenance.
- Therefore, deployment and development of CASE tools have become pet subjects for most software project managers.
- For software engineers, CASE tools promises to reduce the burden of routine jobs, and help develop better quality products more efficiently.

**CASE AND ITS SCOPE**

- A CASE tool is a generic term used to denote any form of automated support for software engineering,
- In a more restrictive sense a CASE tool can mean any tool used to automate some activity associated with software development.
- Many CASE tools are now available.
- Some of these tools assist in phase-related tasks such as
  - specification,
  - structured analysis,
  - design,
  - coding,
  - testing, etc. and

  - Others to non-phase activities such as project management and configuration management.
- The primary objectives in using any CASE tool are:
  - To increase productivity.
  - To help produce better quality software at lower cost.

## 8. <u>CASE ENVIRONMENT</u>

- CASE tools are not integrated, then the data generated by one tool would have to input to the other tools.
- This may also involve format conversions as the tools developed by different vendors are likely to use different formats.
- CASE tools are characterized by the stage or stages of software development life cycle on which they focus.
- The central repository all the CASE tools in a CASE environment share common information among them.
- Thus a CASE environment facilitates the automation of the step-by-step methodologies for software development.
- In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development.
- The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger.
- The different tools are integrated to the extent that once the compiler detects an error, the editor takes automatically goes to the statements in error and the error statements are highlighted.
- Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc.
- A schematic representation of a CASE environment is shown in Figure 12.1.

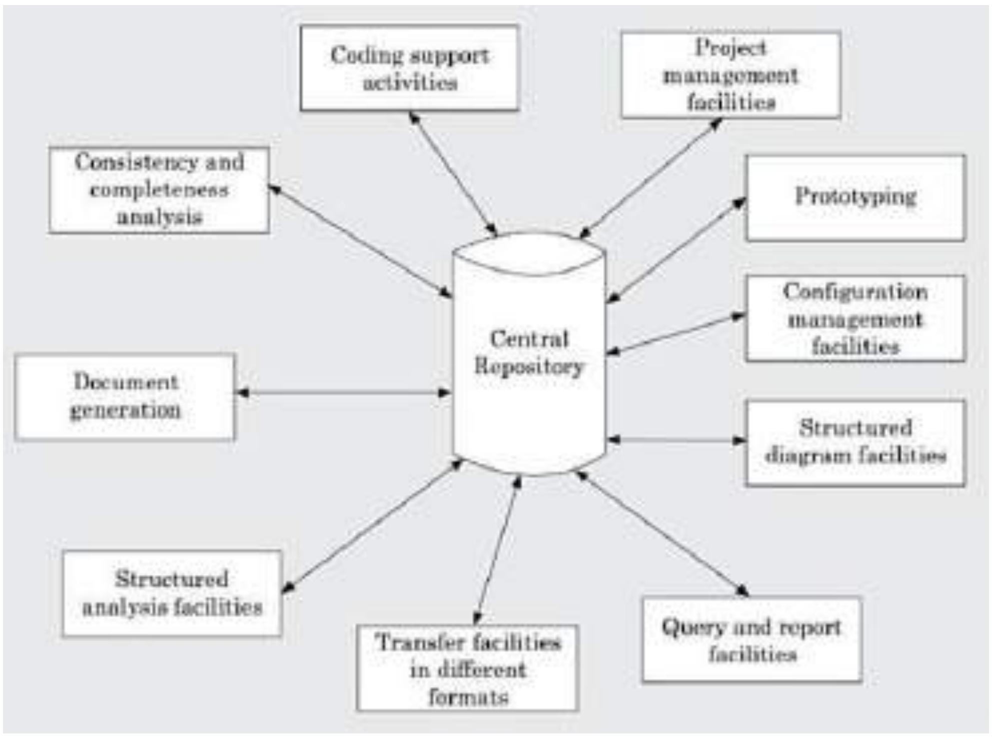

Coding support activities
Project management facilities
Consistency and completeness analysis
Prototyping
Configuration management facilities
Central Repository
Document generation
Structured diagram facilities
Structured analysis facilities
Transfer facilities in different formats
Query and report facilities

**Figure 12.1:** A CASE environment.

- The standard programming environments such as Turbo C, Visual C++, etc. come equipped with a program editor, compiler, debugger, linker, etc.,
- All these tools are integrated. If you click on an error reported by the compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

### 8.1 Benefits of CASE

Let us examine some of these benefits:

- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases.
- Use of CASE tools leads to considerable improvements in quality.
- CASE tools help produce high quality and consistent documents.
- CASE tools reduce the complexity in a software engineer's work.
- CASE tools have led to revolutionary cost saving in software maintenance efforts.

## 9. CASE SUPPORT IN SOFTWARE LIFE CYCLE

- Let us examine the various types of support that CASE provides during the different phases of a software life cycle.
- CASE tools should support a development methodology, help enforce the same, and provide certain amount of consistency checking between different phases.
- Some of the possible support that CASE tools usually provide in the software development life cycle is discussed below.

### 9.1 Prototyping Support

- We have already seen that prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on.
- The prototyping CASE tool's requirements are as follows:
    - Define user interaction.
    - Define the system control flow.
    - Store and retrieve data required by the system.
    - Incorporate some processing logic.
- There are several stand alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype.

A good prototyping tool should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, a prototyping CASE tool should support the user to create a GUI using a graphics editor.
- The user should be allowed to define all data entry forms, menus and controls. It should integrate with the data dictionary of a CASE environment.
- The user should be able to define the sequence of states through which a created prototype can run.
- The user should also be allowed to control the running of the prototype.

### 9.2 Structured Analysis and Design

- A CASE tool should support one or more of the structured analysis and design technique.
- The CASE tool should support effortlessly drawing analysis and design diagrams.
- The CASE tool should support drawing fairly complex diagrams and preferably through a hierarchy of levels.

- It should provide easy navigation through different levels and through design and analysis.
- The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy.

### 9.3 Code Generation

- More pragmatic support expected from a CASE tool during code generation phase are the following:
    - The CASE tool should support generation of module skeletons or templates in one or more popular languages.
    - It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.
    - The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular programming languages.
    - It should generate database tables for relational database management systems.

### 9.4 Test Case Generator

- The CASE tool for test case generation should have the following features:
    - It should support both design and requirement testing
    - It should generate test set reports in ASCII format which can be directly imported into the test plan document.

## 10. OTHER CHARACTERISTICS OF CASE TOOLS

- The characteristics listed in this section are not central to the functionality of CASE tools but significantly enhance the effectively and usefulness of CASE tools.

### 10.1 Hardware and Environmental Requirements

- In most cases, it is the existing hardware that would place constraints upon the CASE tool selection.
- Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities.
- Therefore, we have to emphasize on selecting the most optimal CASE tool configuration for a given hardware configuration.

### 10.2 Documentation Support

- The deliverable documents should be organized graphically and should be able to incorporate text and diagrams from the central repository.
- This helps in producing up-to-date documentation.
- The CASE tool should integrate with one or more of the commercially available desktop publishing packages.
- It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

### 10.3 Pro ject Management

- It should support collecting, storing, and analysing information on the software project’s progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

### 10.4 External Interface

- The tool should allow exchange of information for reusability of design.
- The information which is to be exported by the tool should be preferably in ASCII format and support open architecture.

### 10.5 Reverse Engineering Support

- The tool should support generation of structure charts and data dictionaries from the existing source codes.
- It should populate the data dictionary from the source code.
- If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

### 10.6 Data Dictionary Interface

- The data dictionary interface should provide view and update access to the entities and relations stored in it.
- It should have print facility to obtain hard copy of the viewed screens.
- It should provide analysis reports like cross-referencing, impact analysis, etc.
- Ideally, it should support a query language to view its contents.

## 11. TOWARDS SECOND GENERATION CASE TOOL

- An important feature of the second generation CASE tool is the direct support of any adapted methodology.
- This would necessitate the function of a CASE administrator for every organization, who can tailor the CASE tool to a particular methodology.
- In addition, we may look forward to the following features in the second generation CASE tool:

**Intelligent diagramming support:**

- The fact that diagramming techniques are useful for system analysis and design is well established. The future CASE tools would provide help to draw the diagrams.

**Integration with implementation environment:**

- The CASE tools should provide integration between design and implementation.

**Data dictionary standards:**

- It is highly unlikely that any one vendor will be able to deliver a total solution.
- Moreover, a preferred tool would require tuning up for a particular system.
- Thus the user would act as a system integrator. This is possible only if some standard on data dictionary emerges.

## 12. ARCHITECTURE OF A CASE ENVIRONMENT

- The architecture of CASE environment is shown in Figure 12.2.
- The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository.

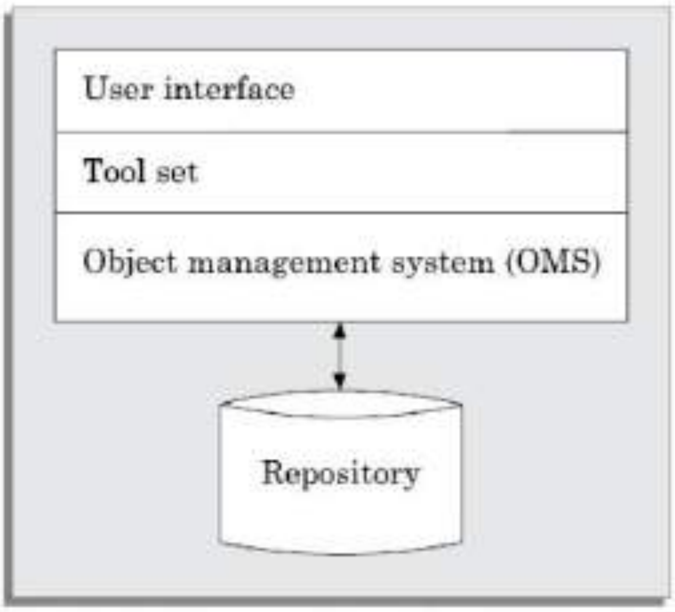


**Figure 12.2:** Architecture of a modern CASE environment.

**User interface**

- The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

**Object management system and repository**

- Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc.
- The object management system maps these logical entities into the underlying storage management system (repository).
- The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records.
- There are a few types of entities but large number of instances.
- By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each.
- Thus the object management system takes care of appropriately mapping these entities into the underlying storage management system.

# SOFTWARE ENGINEERING UNIT-6

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING.

Dr DEEPAK NEDUNURI

[ SIR C R REDDY COLLEGE OF ENGINEERING ]

ELURU.

# SOFTWARE ENGINEERING
# UNIT-6

1. Software Maintenance: Software maintenance,
2. Maintenance Process Models,
3. Maintenance Cost,
4. Software Configuration Management.
5. Software Reuse: what can be Reused?
6. Why almost No Reuse So Far?
7. Basic Issues in Reuse Approach,
8. Reuse at Organization Level.

# 1. SOFTWARE MAINTENANCE

- The mention of the word maintenance brings up the image of a screw driver, wielding mechanic with soiled hands holding onto a bagful of spare parts.
- It would be the objective of this chapter to give idea of the software maintenance projects, and to familiarize you with the latest techniques in software maintenance.
- Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is necessity for almost any kind of product.
- However, most products need maintenance due to the wear and tear caused by use.
- On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

## 1.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE

- In this section, we first classify the different maintenance efforts into a few classes.

**Types of Software Maintenance**

- There are three types of software maintenance, which are described as follows:
- **Corrective:** Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

**Characteristics of Software Evolution**

- Lehman and Belady have studied the characteristics of evolution of several software products [1980].
- They have expressed their observations in the form of laws.
- Their important laws are presented in the following subsection. But a word of caution here is that these are generalizations and may not be applicable to specific cases and also most of these observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.
- **Lehman's first law:** A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts.
- **Lehman's second law:** The structure of a program tends to degrade as more and more maintenance is carried out on it.
- **Lehman's third law:** Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified.
- Therefore this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

## 1.2 SOFTWARE REVERSE ENGINEERING

- Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.

- The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.
- Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

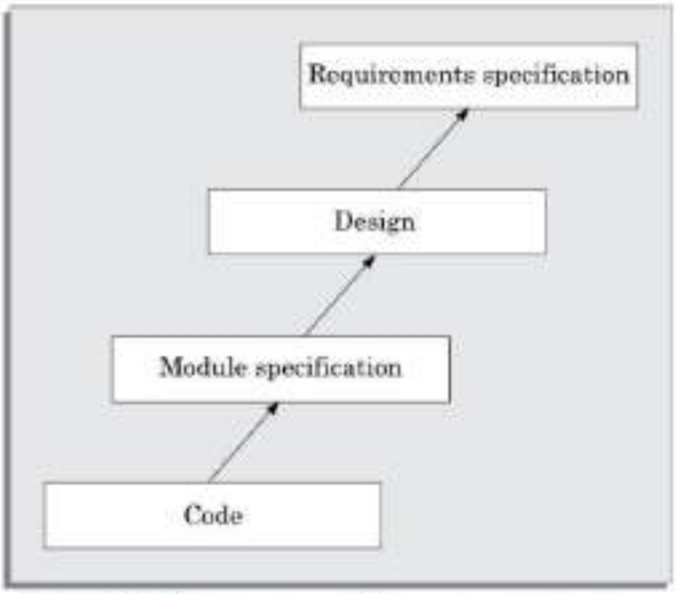


**Figure 13.1:** A process model for reverse engineering.

- A way to carry out these cosmetic changes is shown schematically in Figure 13.1.
- After the cosmetic changes have been carried out on legacy software, the process of extracting the code, design, and the requirements specification can begin.
- These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code.
- The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

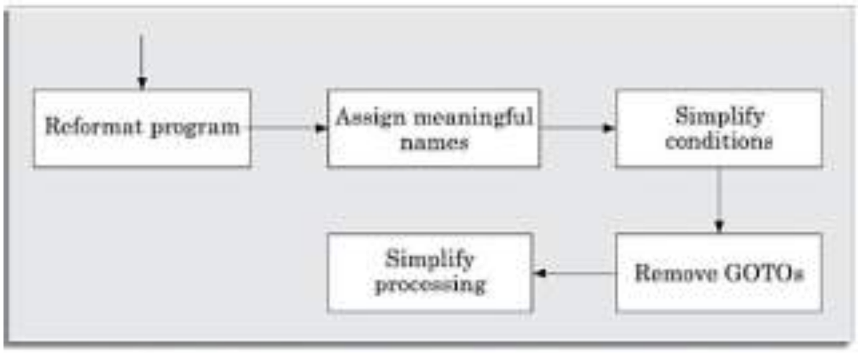


**Figure 13.2:** Cosmetic changes carried out before reverse engineering.

## 2. SOFTWARE MAINTENANCE PROCESS MODELS

- The activities involved in a software maintenance project are
  - (i) the extent of modification to the product required,
  - (ii) the resources available to the maintenance team,
  - (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.),
  - (iv) the expected project risks, etc.
- When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.
- Two broad categories of process models can be proposed.

### First model

- The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later.
- This maintenance process is graphically presented in Figure 13.3.

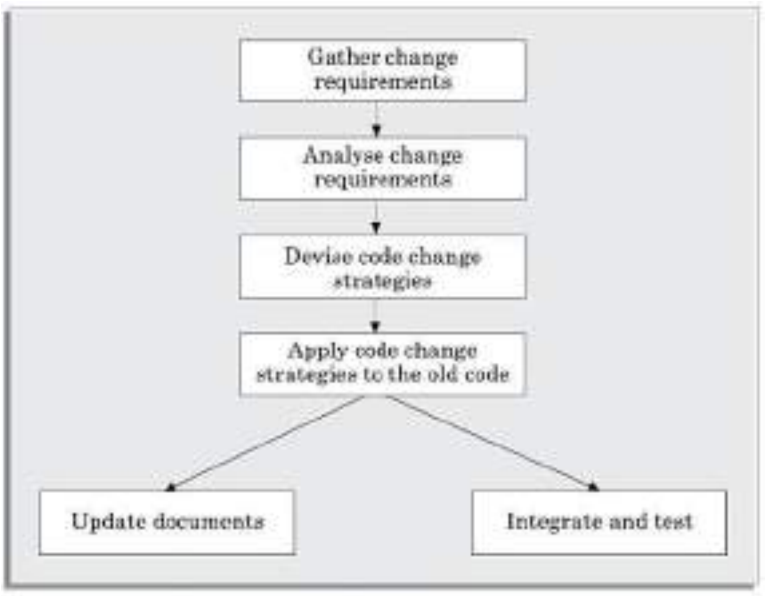

Gather change requirements
Analyse change requirements
Devise code change strategies
Apply code change strategies to the old code
Update documents
Integrate and test

**Figure 13.3:** Maintenance process model 1.

- In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change.
- At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code.
- The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system.
- Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

## Second model

- The second model is preferred for projects where the amount of rework required is significant.
- This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering. This process model is depicted in Figure 13.4.

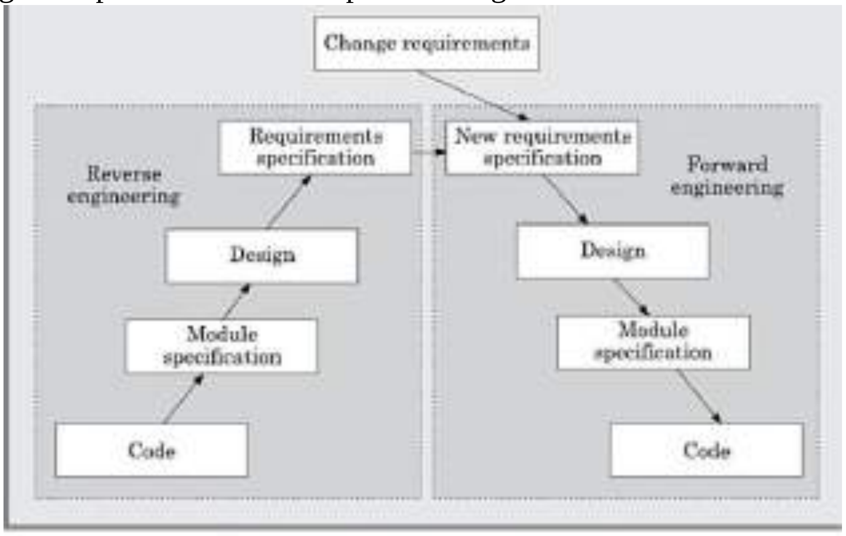


**Figure 13.4:** Maintenance process model 2.

- The reverse engineering cycle is required for legacy products.
- During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications.
- The module specifications are then analyzed to produce the design.
- The design is analyzed (abstracted) to produce the original requirements specification.
- The change requests are then applied to this requirements specification to arrive at the new requirements specification.
- At this point a forward engineering is carried out to produce the new code.

## 3. <u>ESTIMATION OF MAINTENANCE COST</u>

- We had earlier pointed out that maintenance efforts require about 60 per cent of the total life cycle cost for a typical software product.
- However, maintenance costs vary widely from one application domain to another.
- For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.
- Boehm [1981] proposed a formula for estimating maintenance costs as part of his **Co**nstructive **Co**st **Mo**del - COCOMO cost estimation model.
- Boehm's maintenance cost estimation is made in terms of a quantity called the **A**nnual **C**hange **T**raffic (ACT).
- Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion. Where, KLOC added is the total kilo lines of source code added during maintenance.
- KLOC deleted is the total KLOC deleted during maintenance.
- Thus, the code that is changed should be counted in both the code added and code deleted.
- The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

  Maintenance cost = ACT × Development cost

- Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

## 4. SOFTWARE CONFIGURATION MANAGEMENT

- The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g., source code, design document, SRS document, test document, user's manual, etc.
- These objects are usually referred to and modified by a number of software developers throughout the life cycle of the software.
- The state of each deliverable object changes as development progresses and also as bugs is detected and fixed.
- The configuration of the software is the state of all project deliverables at any point of time; and software configuration management deals with effectively tracking and controlling the configuration of software during its life cycle.

**Software revision versus version**

- A new version of software is created when there is significant change in functionality, technology, or the hardware it runs on, etc.
- On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc.
- Even the initial delivery might consist of several versions and more versions might be added later on.
- For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on.
- As software is released and used by the customer, errors are discovered that need correction.
- Enhancements to the functionalities of the software may also be needed.
- A new release of software is an improved system intended to replace an old one.

### 4.1 Necessity of Software Configuration Management

- There are several reasons for putting an object under configuration management. The following are some of the important problems.
- **Problems associated with concurrent access:** Possibly the most important reason for configuration management is to control the access to the different deliverable objects.
- **Providing a stable development environment:** When a project work is underway, the team members need a stable environment to make progress.
- **System accounting and maintaining status information:** System accounting denotes keeping track of who made a particular change to an object and when the change was made.
- **Handling variants:** Existence of variants (software versions) of a software product causes some peculiar problems. Suppose you have several variants of the same module, and find that a bug exists in one of them. Then, it has to be fixed in all versions and revisions.
- To do it efficiently, you should not have to fix it in each and every version and revision of the software separately. Making a change to one program should be reflected appropriately in all relevant versions and revisions.

### 4.2 Configuration Management Activities

- Configuration management is carried out through two principal activities:
- **Configuration identification:** It involves deciding which parts of the system should be kept track.
- **Configuration control:** Configuration control is the process of managing changes to controlled objects. It ensures that changes to a system happen smoothly. A *configuration management tool* helps to keep track the current state of the project. The configuration management tool enables the developer to change various components in a controlled manner.
- **Source code control system (SCCS) and RCS:** SCCS and RCS are two popular **Configuration management tools** available on most UNIX systems.

## 5. SOFTWARE REUSE

- Software products are expensive. Therefore, software project managers are always worried about the high cost of software development.
- A possible way to reduce development cost is to reuse parts from previously developed software.
- In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality.
- A reuse approach that is of late gaining prominence is component-based development. Component-based software development is different from the traditional software development in the sense that software is developed by assembling software from off-the-shelf components.
- Software development with reuse is very similar to a modern hardware engineer building an electronic circuit by using standard types of ICs and other components.

### 5.1 WHAT CAN BE REUSED?

- Before discussing the details of reuse techniques, it is important to deliberate about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

  Requirements specification
  Design
  Code
  Test cases
  Knowledge
- Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts, reuse of knowledge occurs automatically without any conscious effort in this direction.
- A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

## 6. WHY ALMOST NO REUSE SO FAR?

- Engineers working in software development organizations often have a feeling that the current system that they are developing **is similar to the last few systems built**.

- However, no attention is paid on how not to duplicate what can be reused from previously developed systems.
- Everything is being built from the old system.
- The current system falls behind schedule and no one has time to figure out how the similarity between the current system and the systems developed in the past can be exploited.
- Even those organizations which start the process of reusing programs.
- Creation of components that are reusable in different applications is a difficult problem.
- It is very difficult to anticipate the exact components that can be reused across different applications.
- But, even when the reusable components are carefully created and made available for reuse, programmers prefer to create their own, because the available components are difficult to understand and adapt to the new applications.
- In this context, the following observation is significant:
    - The routines of mathematical libraries are being reused very successfully by almost every programmer.
    - No one in their mind would think of writing a routine to compute sine or cosine.
    - Let us investigate why reuse of commonly used mathematical functions is so easy.
    - Everyone has clear ideas about what kind of argument should implement the type of processing to be carried out and the results returned.
    - Secondly, mathematical libraries have a small interface.

## 7. <u>BASIC ISSUES IN ANY REUSE PROGRAM</u>

- The following are some of the basic issues that must be clearly understood for starting any reuse program:
  Component creation.
  Component indexing and storing.
  Component search.
  Component understanding.
  Component adaptation.
  Repository maintenance.

**Component creation:**

- For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important.

**Component indexing and storing**

- Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse.
- The components need to be stored in a *relational database management system* (RDBMS) or an *object-oriented database system* (ODBMS) for efficient access when the number of components becomes large.

**Component searching**

- The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

**Component understanding**

- The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component.
- To facilitate understanding, the components should be well documented and should do something simple.

**Component adaptation**

- Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand.
- However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

**Repository maintenance**

- A component repository once is created requires continuous maintenance.
- New components, as and when created have to be entered into the repository.
- The faulty components have to be tracked.
- Further, when new applications emerge, the older applications become . In this case, the obsolete components might have to be removed from the repository.

7.1 A REUSE APPROACH

- A promising approach that is being adopted by many organizations is to introduce a building block approach into the software development process. For this, the reusable components need to be identified after every development project is completed.
- The reusability of the identified components has to be enhanced and these have to be cataloged into a component library.
- It must be clearly understood that an issue crucial to every reuse effort is the identification of reusable components.
- Domain analysis is a promising approach to identify reusable components.
- The domain analysis approach to create reusable components.

**Domain Analysis**

- The aim of domain analysis is to identify the reusable components for a problem domain.

**Reuse domain**

- A reuse domain is a technically related set of application areas.

**Evolution of a reuse domain**

- The ultimate results of domain analysis are development of problem oriented languages. The problem-oriented languages are also known as *application generators*.

**Component Classification**

- Components need to be properly classified in order to develop an effective indexing and storage scheme. We have already remarked that hardware reuse has been very successful.

**Searching**

- The domain repository may contain thousands of reuse items. In such large domains, what is the most efficient way to search an item that one is looking for?

**Repository Maintenance**

- Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search.

**Reuse without Modifications**

- Once standard solutions emerge, no modifications to the program parts may be necessary. One can directly plug in the parts to develop his application. Reuse without modification is much more useful than the classical program libraries.
- Application generators have been applied successfully to data processing application, user interface, and compiler development. Application generators are less successful with the development of applications with close interaction with hardware such as real-time systems.

## 8. REUSE AT ORGANISATION LEVEL

- Reusability should be a standard part in all software development activities including specification, design, implementation, test, etc.
- Ideally, there should be a steady flow of reusable components. In practice, however, things are not so simple.
- Extracting reusable components from projects that were completed in the past presents an important difficulty not encountered while extracting a reusable component from an ongoing project—typically; the original developers are no longer available for consultation.
- Development of new systems leads to a collection of related products, since reusability ranges from items whose reusability is immediate to those items whose reusability is highly improbable.
- Achieving organization-level reuse requires adoption of the following steps:

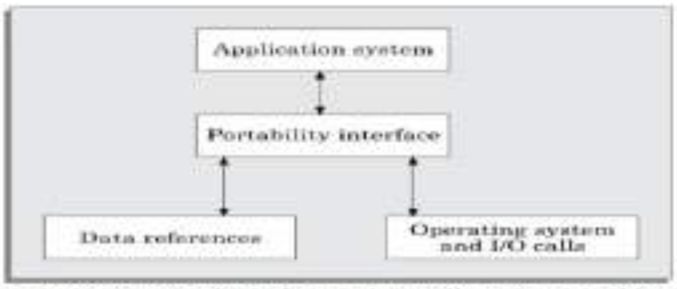
Assess of an item's potential for reuse.
Refine the item for greater reusability.
Enter the product in the reuse repository.

**Assessing a product's potential for reuse**

- Assessment of a components reuse potential can be obtained from an analysis of a questionnaire circulated among the developers.
- The questionnaire can be devised to assess a component's reusability.
- The programmers working in similar application domain can be used to answer the questionnaire about the product's reusability.
- Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored.
- A sample questionnaire to assess a component's reusability is the following:
    - Is the component's functionality required for implementation of systems in the future?
    - How common is the component's function within its domain?
    - Would there be a duplication of functions within the domain if the component is taken up?
    - Is the component hardware dependent?
    - Is the design of the component optimized enough?
    - If the component is non-reusable, then can it be decomposed to yield some reusable components?
    - Can we parameterize a non-reusable component so that it becomes reusable?

**Refining products for greater reusability**

- For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localized using data encapsulation techniques.
- The following refinements may be carried out:
- **Name generalization:** The names should be general, rather than being directly related to a specific application.
- **Operation generalization:** Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.
- **Exception generalization:** This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.
- **Handling portability problems:** Programs typically make some assumption regarding the representation of information in the underlying machine.
- These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be the same on all machines.
- Also, programs use some function libraries, which may not be available on all host machines.
- A portability solution to overcome these problems is shown in Figure 14.1.



- Figure 14.1: Improving reusability of a component by using a portability interface.

- The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program.
- Also, all platform-related calls should be routed through the portability interface.

**Current State of Reuse**

- In spite of all the shortcomings of the state-of-the-art reuse techniques, it is the experience of several organizations that most of the factors inhibiting an effective reuse program are non-technical.
- Some of these factors are the following: Need for commitment from the top management.
    - Adequate documentation to support reuse.
    - Adequate incentive to reward those who reuse. Both the people
    - Contributing new reusable components and those reusing the existing components should be rewarded to start a reuse program and keep it going.
    - Providing access to and information about reusable components.
- Organizations are often hesitant to provide an open access to the reuse repository for the fear of the reuse components finding a way to their competitors.