INDEX

Lesson No. : 1 **Writer: Dr. Rakesh Kumar**

Introduction to Software Engineering **Vetter: Sh. Naresh Mann**

1.0 Objectives

The objective of this lesson is to make the students acquainted with the introductory concepts of software engineering. To make them familiar with the problem of software crisis this has ultimately resulted into the development of software engineering. After studying this lesson, the students will:

1. Understand what is software crisis?
2. What are software engineering and its importance?
3. What are the quality factors of software?

1.1 Introduction

In order to develop a software product, user needs and constraints must be determined and explicitly stated; the product must be designed to accommodate implementers, users and maintainers; the source code must be carefully implemented and thoroughly tested; and supporting documents must be prepared. Software maintenance tasks include analysis of change request, redesign and modification of the source code, thorough testing of the modified code, updating of documents to reflect the changes and the distribution of modified work products to the appropriate user. The need for systematic approaches to development and maintenance of software products became apparent in the 1960s. Many software developed at that time were subject to cost overruns, schedule slippage, lack of reliability, inefficiency, and lack of user acceptance. As computer systems become larger and complex, it became apparent that the demand for computer software was growing faster than our ability to produce and maintain it. As a result the field of software engineering has evolved into a technological discipline of considerable importance.

1.2 Presentation of contents

1.2.1 The Software Crisis

The headlines have been screaming about the Y2K Software Crisis for years now. Lurking behind the Y2K crisis is the real root of the problem: The Software Crisis. After five decades of progress, software development has remained a craft and has yet to emerge into a science.

What is the Software Crisis?

Is there a crisis at all? As you stroll through the aisles of neatly packaged software in your favorite computer discount store, it wouldn't occur to you that there's a problem. You may be surprised to learn that those familiar aisles of software represent only a small share of the software market--of the $90 Billion software market, a mere 10% of software products are "shrink wrapped" packages for personal computers. The remaining 90% of the market is comprised of large software products developed to specific customer specifications.

By today's definition, a "large" software system is a system that contains more than 50,000 lines of high-level language code. It's those large systems that bring the software crisis to light. You know that in large projects the work is done in teams consisting of project managers, requirements analysts, software engineers, documentation experts, and programmers. With so many professionals collaborating in an organized manner on a project, what's the problem?

Why is it that the team produces fewer than 10 lines of code per day over the average lifetime of the project?

Why are sixty errors found per every thousand lines of code?

Why is one of every three large projects scrapped before ever being completed?

Why is only 1 in 8 finished software projects considered "successful?"

And more:

- The cost of owning and maintaining software in the 1980’s was twice as expensive as developing the software.
- During the 1990’s, the cost of ownership and maintenance increased by 30% over the 1980’s.
- In 1995, statistics showed that half of surveyed development projects were operational, but were not considered successful.
- The average software project overshoots its schedule by half.
- Three quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer’s requirements.

Software projects are notoriously behind schedule and over budget. Over the last twenty years many different paradigms have been created in attempt to make software development more predictable and controllable. There is no single solution to the crisis. It appears that the Software Crisis can be boiled down to two basic sources:

- Software development is seen as a craft, rather than an engineering discipline.
- The approach to education taken by most higher education institutions encourages that "craft" mentality.

Software development today is more of a craft than a science. Developers are certainly talented and skilled, but work like craftsmen, relying on their talents and skills and using techniques that cannot be measured or reproduced. On the other hand, software engineers place emphasis on reproducible, quantifiable techniques–the marks of science. The software industry is still many years away from becoming a mature engineering discipline. Formal software engineering processes exist, but their use is not widespread. A crisis similar to the software crisis is not seen in the hardware industry, where well-documented, formal processes are tried and true. To make matters worse, software technology is constrained by hardware technology. Since hardware develops at a much faster pace than software, software developers are constantly trying to catch up and take advantage of hardware improvements. Management often encourages ad hoc software development in an attempt to get products out on time for the new hardware architectures. Design, documentation, and evaluation are of secondary importance and are omitted or completed after the fact. However, as the statistics show, the ad hoc approach just doesn't work. Software developers have classically accepted a certain number of errors in their work as inevitable and part of the job. That mindset becomes increasingly unacceptable as software becomes embedded in more and more consumer electronics. Sixty errors per thousand lines of code is unacceptable when the code is embedded in a toaster, automobile, ATM machine or razor.

1.2.2 Mature Software

As we have seen, most software projects do not follow a formal process. The result is a product that is poorly designed and documented. Maintenance becomes problematic because without a design and documentation, it’s difficult or impossible to predict what sort of effect a simple change might have on other parts of the system.

Fortunately there is an awareness of the software crisis, and it has inspired a worldwide movement towards process improvement. Software industry leaders are beginning to see that following a formal software process consistently leads to better quality products, more efficient teams and individuals, reduced costs, and better morale.

The SEI (Software Engineering Institute) uses a Capability Maturity Model (CMM) to assess the state of an organization’s development process. Such models are nothing new–they have been routinely applied to industrial engineering disciplines. What’s new is the application to software development. The SEI Software CMM has become a de facto standard for assessing and improving software processes. Ratings range from Maturity Level 1, which is characterized by ad hoc development and lack of a formal software development process, up to Maturity Level 5, at which an organization not only has a formal process, but also continually refines and improves it. Each maturity level is further broken down into key process areas that indicate the areas an organization should focus on to improve its software process (e.g. requirement analysis, defect prevention, or change control).

Level 5 is very difficult to attain. In early 1995, only two projects, one at Motorola and another at Loral (the on-board space shuttle software project), had earned Maturity Level 5. Another study showed that only 2% of reviewed projects rated in the top two Maturity Levels, in spite of many of those projects placing an extreme emphasis on software process improvement. Customers contracting large projects will naturally seek organizations with high CMM ratings, and that has prompted increasingly more organizations to investigate software process improvement.

Mature software is also reusable software. Artisans are not concerned with producing standardized products, and that is a reason why there is so little interchangeability in software components. Ideally, software would be standardized to such an extent that it could be marketed as a "part", with its own part number and revision, just as though it were a hardware part. The software component interface would be compatible with any other software system. Though it would seem that nothing less than a software development revolution could make that happen, the National Institute of Standards and Technology (NIST) founded the Advanced Technology Program (ATP), one purpose of which was to encourage the development of standardized software components.

The consensus seems to be that software has become too big to treat as a craft. And while it may not be necessary to apply formal software processes to daily programming tasks, it is important in the larger scheme of things, in that it encourages developers to think like engineers.

1.2.3 Software Engineering

Software Engineering (SE) is the design, development, and documentation of software by applying technologies and practices from computer science, project management, engineering, application domains, interface design, digital asset management and other fields.

The term software engineering was popularized after 1968, during the 1968 NATO Software Engineering Conference (held in Garmisch, Germany) by its chairman F.L. Bauer, and has been in widespread use since.

The term software engineering has been commonly used with a variety of distinct meanings:

- As the informal contemporary term for the broad range of activities that was formerly called programming and systems analysis;
- As the broad term for all aspects of the practice of computer programming, as opposed to the theory of computer programming, which is called computer science;
- As the term embodying the advocacy of a specific approach to computer programming, one that urges that it be treated as engineering discipline rather than an art or a craft, and advocates the codification of recommended practices in the form of software engineering methodologies.
- Software engineering is "(1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software," and "(2) the study of approaches as in (1)." – IEEE Standard 610.12

- Software engineering is defined as the systematic and scientific approach to develop, operate, maintain and to retire the software product. Software product means software for a large/medium size and complex problem. We get the real advantage of software engineering when it is applied to a project. Though it can also be used for the development of programs/small assignments.
- Software engineering is the application of science and mathematics, by which the computer hardware is made useful to the user via software (computer programs, procedures, and associated documentation).

1.2.4 Scope and Focus

Software engineering is concerned with the conception, development and verification of a software system. This discipline deals with identifying, defining, realizing and verifying the required characteristics of the resultant software. These software characteristics may include: functionality, reliability, maintainability, availability, testability, ease-of-use, portability, and other attributes. Software engineering addresses these characteristics by preparing design and technical specifications that, if implemented properly, will result in software that can be verified to meet these requirements.

Software engineering is also concerned with the characteristics of the software development process. In this regard, it deals with characteristics such as cost of development, duration of development, and risks in development of software.

1.2.5 The Need for Software Engineering

Software is often found in products and situations where very high reliability is expected, even under demanding conditions, such as monitoring and controlling nuclear power plants, or keeping a modern airliner aloft Such applications contain millions of lines of code, making them comparable in complexity to the most complex modern machines. For example, a modern airliner has several million physical parts (and the space shuttle about ten million parts), while the software for such an airliner can run to 4 million lines of code.

1.2.6 Technologies and practices

Software engineers advocate many different technologies and practices, with much disagreement. Software engineers use a wide variety of technologies and practices. Practitioners use a wide variety of technologies: compilers, code repositories, word processors. Practitioners use a wide variety of practices to carry out and coordinate their efforts: pair programming, code reviews and daily stand up meetings. The goal of every software engineer should be to bring an idea out of a previous planned model, which should be transparent and well documented.

In spite of the enormous economic growth and productivity gains enabled by software, persistent complaints about the quality of software remain.

1.2.7 Nature of Software Engineering

Software engineering resembles many different fields in many different ways. The following paragraphs make some simple comparisons.

1.2.7.1 Mathematics

Programs have many mathematical properties. For example the correctness and complexity of many algorithms are mathematical concepts that can be rigorously proven. Programs are finite, so in principle, developers could know many things about a program in a rigorous mathematical way. The use of mathematics within software engineering is often called formal methods. However, computability theory shows that not everything useful about a program can be proven. Mathematics works best for small pieces of code and has difficulty scaling up.

1.2.7.2 Engineering

Software Engineering is considered by many to be an engineering discipline because there are pragmatic approaches and expected characteristics of engineers. Proper analysis, documentation, and commented code are signs of an engineer. It is argued that software engineering is engineering. Programs have many properties that can be measured. For example, the performance and scalability of programs under various workloads can be measured. The effectiveness of caches, bigger processors, faster networks, newer databases are engineering issues. Mathematical equations can sometimes be deduced from the measurements. Mathematical approaches work best for system-wide analysis, but often are meaningless when comparing different small fragments of code.

1.2.7.3 Manufacturing

Programs are built in as a sequence of steps. By properly defining and carrying out those steps, much like a manufacturing assembly line, advocates hope to improve the productivity of developers and the quality of final programs. This approach inspires the many different processes and methodologies.

1.2.7.4 Project management

Commercial (and many non-commercial) software projects require management. There are budgets and schedules to set. People to hire and lead. Resources (office space, computers) to acquire. All of this fits more appropriately within the purview of management.

1.2.7.5 Audio and Visual art

Programs contain many artistic elements, like to writing or painting. User interfaces should be aesthetically pleasing and provide optimal audio and visual communication to end-users. What is considered "good design" is often subjective, and may be decided by one's own sense of aesthetics. Because graphic artists create graphic elements for graphical user interfaces, graphic design often overlaps interaction design in the position of an interface designer. User interfaces may require technical understanding including graphical integration with code, computer animation technology, automation of graphic production, integrating graphics with sound editing technology, and mathematical application. One could say that "audiovisual engineering" is required. User interfaces with user-read text and voice may also be enhanced from professional copywriting and technical writing. Code should be aesthetically pleasing to programmers. Even the decision of whether a variable name or class name is clear and simple is an artistic question.

1.2.7.6 Performance

The act of writing software requires that developers summon the energy to find the answers they need while they are at the keyboard. Creating software is a performance that resembles what athletes do on the field, and actors and musicians do on stage. Some argue that Software Engineering need inspiration to spark the creation of code. Sometimes a creative spark is needed to create the architecture or to develop a unit of code to solve a particularly intractable problem. Others argue that discipline is the key attribute. Pair programming emphasizes this point of view.

1.2.8 Branch of Which Field?

Is Software Engineering a branch of programming, a branch of computer science, a branch of traditional engineering, or a field that stands on its own?

1.2.8.1 Branch of programming

Programming emphasizes writing code, independent of projects and customers. Software engineering emphasizes writing code in the context of projects and customers by making plans and delivering applications. As a branch of programming, Software Engineering would probably have no significant licensing or professionalism issues.

1.2.8.2 Branch of computer science

Many believe that software engineering is a part of computer science, because of their close historical connections and their relationship to mathematics. They advocate keeping Software engineering a part of computer science. Both computer science and software engineering care about programs. Computer science emphasizes the theoretical, eternal truths while software engineering emphasizes practical, everyday usefulness. Some argue that computer science is to software engineering as physics and chemistry are to traditional engineering. As a branch of computer science, Software Engineering would probably have few licensing or professionalism concerns.

1.2.8.3 Branch of engineering

Some Software Engineering academics and practitioners have advocated treating Software Engineering an engineering discipline. Advocates for this view argue that the practice of engineering involves the use of mathematics, science, and the technology of the day, to build trustworthy products that are "fit for purpose", a description that applies as well to Software Engineering as to any other engineering discipline. As a branch of engineering, Software Engineering would probably adopt the engineering model of licensing and professionalism.

1.2.8.4 Freestanding field

Recently, software engineering has been finding its own identity and emerging as an important freestanding field. Practitioners are slowly realizing that they form a huge community in their own right. Software engineering may need to create a form of regulation/licensing appropriate to its own circumstances.

1.2.8.5 Debate over the term 'Engineering'

Some people believe that software development is a more appropriate term than software engineering for the process of creating software. Pete McBreen, (author of "Software Craftsmanship: The New Imperative" (ISBN 0-201-73386-2)), argues that the term Software Engineering implies levels of rigor and proven processes that are not appropriate for all types of software development. He argues strongly for 'craftsmanship' as a more appropriate metaphor because that term brings into sharper focus the skills of the developer as the key to success instead of the "manufacturing" process. Using a more traditional comparison, just as not everyone who works in construction is a civil engineer, not everyone who can write code is a software engineer.

Some people dispute the notion that the field is mature enough to warrant the title "engineering". Opposition also comes from the traditional engineering disciplines, whose practitioners usually object to the use of the title "engineer" by anyone who has not gone through an accredited program of engineering education. In each of the last few decades, at least one radical new approach has entered the mainstream of software development (e.g. Structured Programming, Object Orientation, ...), implying that the field is still changing too rapidly to be considered an engineering discipline. Other people would argue that the supposedly radical new approaches are actually evolutionary rather than revolutionary, the mere introduction of new tools rather than fundamental changes.

1.2.9 Software Characteristics

The fundamental difference between a software and hardware is that software is a conceptual entity while hardware is physical entity. When the hardware is built, the process of building a hardware results in a physical entity, which can be easily measured. Software being a logical has the different characteristics that are to be understood.

- **Software is developed or engineered but it is not manufactured in the classical sense.**

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. 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.

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

If you will use hardware, you will observe wear and tear with the passage of time. But software being a conceptual entity will not wear with the passage of time.

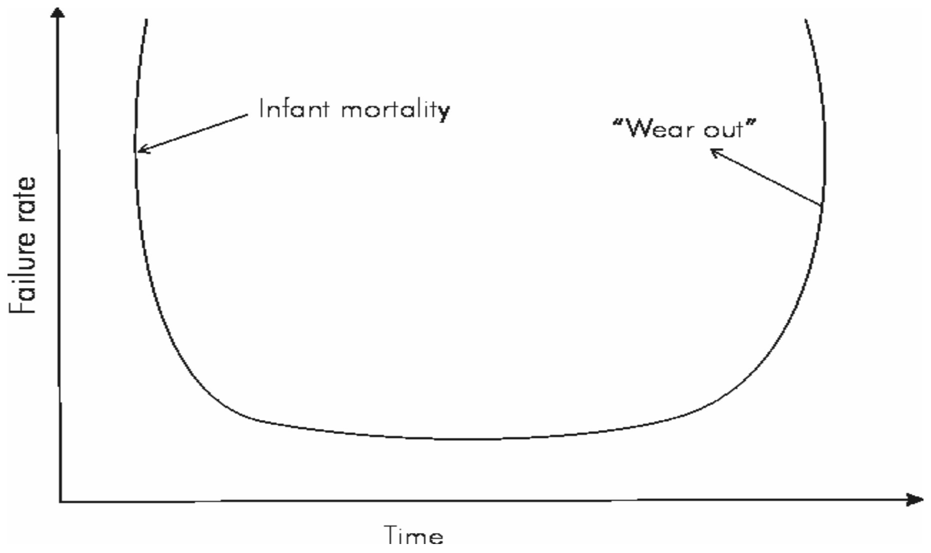


FIGURE 1.1 HARDWARE FAILURE CURVE

Above figure 1.1 shows 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 (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in following Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown.

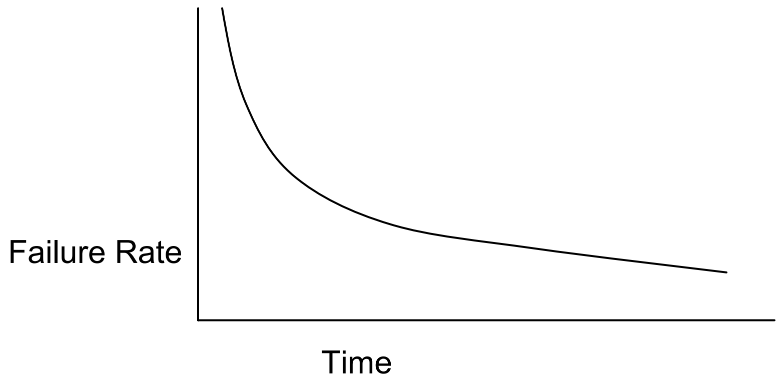


Figure 1.2 Failure curve for software

This seeming contradiction can best be explained by considering the "actual curve" shown in following Figure 1.3. During its life, software will undergo change (maintenance). As the changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise-the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. If any software fails then it indicates an error in design or an error in the process through which design was translated into machine executable code then it means some compilation error. So it is very much clear that, software maintenance involves more complexity than hardware maintenance or we can say that software maintenance is a more complex process than hardware maintenance.

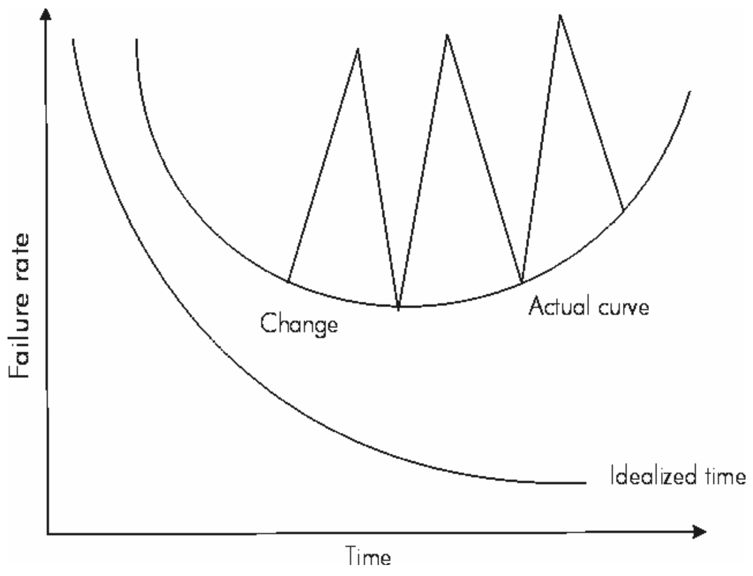


FIGURE 1.3 SOFTWARE IDEALIZED AND ACTUAL FAILURE CURVES

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

Consider the manner in which the control hardware for a computer-based product is designed and built: The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an IC or a chip) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

According to the standard engineering discipline, 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. In the end, we can say that software design is a complex and sequential process.

A software component should be designed and implemented so that it can be reused in different programs since it is a better approach, according to finance and manpower. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

Software components: If you will observe the working of mechanical /electrical /civil engineers, you will see the frequent use reusable components. To built a computer, they will not have to start from the scratch. They will take the components like monitor, keyboard, mouse, hard disk etc. and assemble them together. In the hardware world, component reuse is a natural part of the engineering process.

Reusability of the components has also become the most desirable characteristic in software engineering also. If you have to design software, don’t start from the scratch, rather first check for the reusable components and assemble them. A software component should be designed and implemented so that t can be reused in may different applications. In the languages like C and Pascal we are seeing the presence of a number of library functions (The functions which are frequently required such as to compute the square root etc, are provided in the library and those can be used as such.). With the advent of Object oriented languages such as C++ and Java, reusability has become a primary issue. Reusable components prepared using these languages, encapsulate data as well as procedure. Availability of reusable components can avoid two major problems in the software development: (1) Cost overrun and (2) schedule slippage. If every time we will start from scratch, these problems are inevitable, as we have already realized in procedure oriented approach.

In fourth generation languages also, we are not suppose to specify the procedural detail rather we specify the desired result and supporting software translates the specification of result into a machine executable program.

1.2.10 Software Applications

Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined. Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example, many business applications use highly structured input data (e.g., a database) and produce formatted "reports." Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

Information determinacy refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm(s) without interruption, and produces resultant data in report or graphical format. Such applications are determinate. A multi-user operating system, on the other hand, accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions, and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate.

System software: System software is a collection of programs and utilities for providing service to other programs. Other system applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Real-time software: Software for the monitors/analyzes/controls real-world events as they occur is called real time. Elements of real-time software include a data-gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response can be maintained.

Business software: Business information processing is the largest single software application area. In a broad sense, business software is an integrated software and has many components related to a particular field of the business. Discrete "systems" for example, payroll, accounts receivable/payable, inventory have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision-making. In addition to conventional data processing application, business software applications also encompass interactive computing.

Engineering and scientific software: Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software: Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Personal computer software: The personal computer is the type of computer, which gave revolution to the information technology. The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

Web-based software: The Web pages processed by the browser are the software that incorporates executable instructions (e.g., CGI, HTML, PERL, or Java), and data (e.g. hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

Artificial intelligence software: Artificial intelligence (AI) software is the software, which thinks and behaves like a human. AI software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

1.2.11 Software Quality Attributes

Software quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, implicit characteristics that are expected of all professionally developed software.

The above definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for ease of user). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

"Good Software" needs to be fit for its purpose i.e. it does what it is intended to do. Software has various attributes that lend towards it being good or bad.

External quality attributes are visible to anyone using the software. Reliability is an external quality attribute, which can be measured during or after implementation by testing how the software product relates to its environment.

The internal quality of the software can be measured in terms of its technical attributes such as coupling and cohesion. Some may question the importance of internal quality attributes especially if the software seems to work well and the client is satisfied with it. It can be reasoned though that the internal quality attributes have an impact upon the external quality of the software. Low cohesion for example can lead to messy code, which may be very hard to understand and maintain. The ability to maintain the software system is of course an important factor to be considered when determining the overall quality of the software product.

1.2.11.1 ISO 9126

The ISO 9126 is a Quality Model outlines the factors that contribute to software being good or not so good. The International Organization for Standardization (ISO) 9126 model outlines the following quality attributes:

- Functionality - how well the software system caters for the client's needs
  - Suitability
  - Accuracy: The precision of computations and control.
  - Interoperability
  - Compliance
  - Security: The availability of mechanisms that control or protect programs and data.
- Reliability - how capable the software system is to maintain a desired level of performance for a certain duration of time
  - Maturity
  - Recoverability
  - Fault Tolerance
- Usability - how much effort is needed on the part of users to properly use the software system
  - Learnability
  - Understandability

- Operability

➢ Efficiency - how well the software system performs depending on the resources it requires

- Time Behaviour
- Resource Behaviour

➢ Maintainability - how easily future changes can be made to the software system

- Stability
- Analysability
- Changeability
- Testability

➢ Portability - how well the systems can be transported from one environment to another

- Installability
- Conformance
- Replaceability
- Adaptability

1.2.11.2 McCall's Quality Model

According to McCall There are three dimensions of a software product (as shown in figure 1.4) dealing with different quality factors. These are discussed below:

➢ **Product Operation:** Its is concerned with those aspects of the software when the software is in operation.

- **Correctness:** It is defined as the extent to which a program satisfies its specification.
- **Reliability:** Informally, software is reliable if the user can depend on it. The specialized literature on software reliability defines reliability in terms of statistical behavior-the probability that the software will operate as expected over a specified time interval. For the purpose of this chapter, however, the informal definition is sufficient.
- **Efficiency:** It is concerned with memory requirement, response time etc.
- **Integrity:**
- **Usability:** It may be defined, as efforts required learning software to operate it.

➢ **Product Transition:** Periodically we have to move the software from one platform to another. This dimension is concerned with those factors, which are concerned with transition.

- **Portability:** It may be defined, as the efforts required moving the software from one hardware platform to another.
- **Reusability:** It is the extent to which parts of the software can be reused in other related applications.
- **Interoperability**: It is the effort required to couple the system with other systems.

➢ **Product Revision:** Maintenance is an important phase of software life. This dimension is concerned with all those factors that are concerned with the modification of the software.

- **Maintainability**: It is defined, as the efforts required identifying the bugs in the program and removing them.
- **Flexibility**: It may be defined, as the efforts required modifying an operational program.
- **Testability**: It is defined, as the efforts required testing a program so that it can be ensured that it is performing the intended functions.

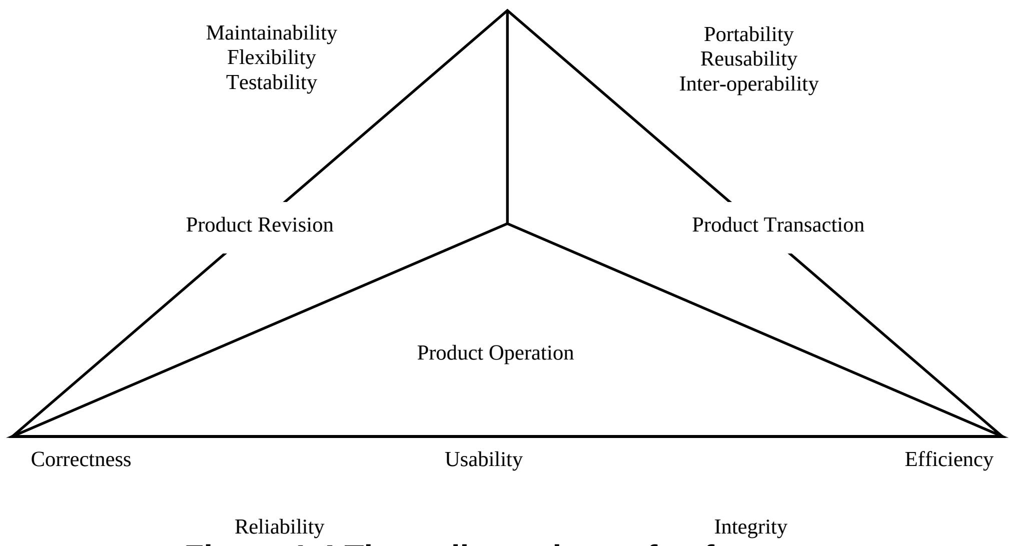


Figure 1.4 Three dimensions of software

1.3 Summary

This lesson has provided an introduction to the basic concepts of software engineering. In early 1960, it became apparent that there is a need of systemetic approach to software development. The software developed in that age faced a number of problems such as cost overru, schedule slippage, poor quality etc. It resulted into a problem coined as softwrae crisis. Software Engineering is the solution to these problems. A number of definitions were presented. An important one is reproduced here “Software engineering is "(1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software," and "(2) the study of approaches as in (1)." – IEEE Standard 610.12”.

This lesson gives an overview of the quality attributes of software. And two quality models were discussed: McCall’s quality model and ISO 9126. According to McCall There are three dimensions of a software product dealing with different quality factors: Product operation, Product transition and product revision. According to ISO 9126, the quality attributes are classified as functionality, reliability, usability, efficiency, maintainability, and portability.

1.4 Keywords

Software quality: It is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, implicit characteristics that are expected of all professionally developed software.

ISO 9126: The ISO 9126 is a Quality Model that outlines the factors that contribute to software being good or not so good.

Software Engineering: It is the design, development, and documentation of software by applying technologies and practices from computer science, project management, engineering, application domains, interface design, digital asset management and other fields.

1.5 Self Assessment Questions

1. What do you understand by software crisis? What are the factors responsible for that? Explain.

2. Define software engineering. What is the need of a systemetic approach to software development? Explain.
3. Define Software quality and discuss the ISO 9126 model of software quality.
4. Develop a list of software quality attributes. Provide a concise definition for each of the quality attributes.
5. What qualities are to be there in software? Discuss the McCall's quality model.

1.6 References/Suggested readings

1. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
2. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
3. Software Engineering by Sommerville, Pearson Education.
4. Software Engineering – A Practitioner's Approach by Roger S Pressman, McGraw-Hill.

Lesson number: 2	**Writer: Dr. Rakesh Kumar**
Software Metrics	**Vetter: Sh. Naresh Mann**

2.1 Objectives

The objective of this lesson is to introduce the students with the concept of software measurement. After studying this lesson they will be familiar with different types of metrics such as Function Points (FP), Source Line Of Code (SLOC), Cyclomatic complexity, etc and theirs advantages and drawbacks.

2.2 Introduction

The IEEE Standard Glossary of Software Engineering Terms define metric as “ a quantitative measure of the degree to which a system, component, or process possesses a given attribute”. A software metric is a measure of some property of a piece of software or its specifications. Since quantitative methods have proved so powerful in the other sciences, computer science practitioners and theoreticians have worked hard to bring similar approaches to software development. Tom DeMarco stated, "You can't control what you can't measure" in DeMarco, T. (1982) Controlling Software Projects: Management, Measurement & Estimation, Yourdon Press, New York, USA, p3. Ejiogu suggested that a metric should possess the following characteristics: (1) simple and computable: It should be easy to learn how to derive the metric and its computation should not be effort and time consuming. (2) Empirically and intuitively persuasive: The metric should satisfy the engineer's intuitive notion about the product under consideration. The metric should behave in certain ways, rising and falling appropriately under various conditions (3) consistent and Objective: The metric should always yield results that are unambiguous. The third party would be able to derive the same metric value using the same information (4) consistent in its use of units and dimensions: It uses only those measures that do not lead to bizarre combinations of units (5) Programming language independent (6) an effective mechanism for quality feedback**.** In addition to the above-mentioned characteristics, Roche suggests that metric should be defined in an unambiguous manner. According to Basili Metrics should be tailored to best accommodate specific products and processes. Software metric domain can be partitioned into process, project, and product metrics. Process metrics are used for software process improvement such as defect rates, errors found during development. Project metrics are used by software project manager to adapt project work flows.

2.3 Presentation of Contents

2.3.1 Common software metrics

Common software metrics include:

- Source lines of code
- Cyclomatic complexity
- Function point analysis
- Code coverage
- Number of classes and interfaces
- Cohesion
- Coupling

2.3.2 Limitations

The assessment of "how much" software there is in a program, especially making prediction of such prior to the detail design, is very difficult to satisfactorily define or measure. The practical utility of software metrics has thus been limited to narrow domains where the measurement process can be stabilized.

Management methodologies such as the Capability Maturity Model or ISO 9000 have therefore focused more on process metrics which assist in monitoring and controlling the processes that produce the software.

Examples of process metrics affecting software:

- Number of times the program failed to rebuild overnight
- Number of defects introduced per developer hour

- Number of changes to requirements
- Hours of programmer time available and spent per week
- Number of patch releases required after first product ship

2.3.3 Criticisms

Potential weaknesses and criticism of the metrics approach:

- **Unethical**: It is said to be unethical to reduce a person's performance to a small number of numerical variables and then judge him/her by that measure. A supervisor may assign the most talented programmer to the hardest tasks on a project; which means it may take the longest time to develop the task and may generate the most defects due to the difficulty of the task. Uninformed managers overseeing the project might then judge the programmer as performing poorly without consulting the supervisor who has the full picture.
- **Demeaning**: "Management by numbers" without regard to the quality of experience of the employees, instead of "managing people."
- **Skewing:** The measurement process is biased by the act of measurement by employees seeking to maximize management's perception of their performances. For example, if lines of code are used to judge performance, then employees will write as many separate lines of code as possible, and if they find a way to shorten their code, they may not use it.
- **Inaccurate**: No known metrics are both meaningful and accurate. Lines of code measure exactly what is typed, but not of the difficulty of the problem. Function points were developed to better measure the complexity of the code

or specification, but they require personal judgment to use well. Different estimators will produce different results. This makes function points hard to use fairly and unlikely to be used well by everyone.

2.3.4 Gaming Metrics

Industry experience suggests that the design of metrics will encourage certain kinds of behaviour from the people being measured. The common phrase applied is "you get what you measure".

A simple example that is actually quite common is the cost-per-function-point metric applied in some Software Process Improvement programs as an indicator of productivity. The simplest way to achieve a lower cost-per-FP is to make function points arbitrarily smaller. Since there is no standard way of measuring function points, the metric is wide open to gaming - that is, cheating.

One school of thought on metrics design suggests that metrics communicate the real intention behind the goal, and that people should do exactly what the metric tells them to do. This is a spin-off of Test-driven Development, where developers are encouraged to write the code specifically to pass the test. If that's the wrong code, then they wrote the wrong test. In the metrics design process, gaming is a useful tool to test metrics and help make them more robust, as well as for helping teams to more clearly and effectively articulate their real goals.

It should be noted that there are very few industry-standard metrics that stand up to even moderate gaming.

2.3.5 Balancing Metrics

One way to avoid the "be careful what you wish for" trap is to apply a suite of metrics that balance each other out. In software projects, it's advisable to have at least one metric for each of the following:

- Schedule
- Risk
- Cost
- Quality

Too much emphasis on any one of these aspects of performance is likely to create an imbalance in the team's motivations, leading to a dysfunctional project.

The Balanced scorecard is a useful tool for managing a suite of metrics that address multiple performance perspectives.

2.3.6 Software Measurement

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind. (Lord Kelvin, Popular Lectures and Addresses, 1889)

Software plays an important role in our life. We want products which affect our lives has quality attributes. We need quality software. In order to determine quality of software we must have some metrics to measure quality. The key point here is quality of the same product may be change. Software is not an exception. So if we determine quality attributes of the software we can also have more precise, predictable and repeatable control over the software development process and product. If software engineer know what he/she will do, then we can "measure" software more easily. Most of time we do not know exactly what is the problem. With only a small understanding of the desired software system, estimations of costs begin.

In the early days of computing, software costs represented a small percentage of the overall cost of a computer-based system. Therefore, a sizable error in estimates of software cost had relatively little impact. Today, software is the most expensive element in many computer-based systems. Therefore, a large cost-estimation can make the difference between profit and loss.

Software measurement enables us to estimate cost and effort devoted to project. Software measurement enables us to determine quality of software. Software measurement enables us to predict maintainability of software. Software measurement enables us to validate the best practices of software development.

Area of software measurement in software engineering is active more than thirty years. There is a huge collection of researches, but still no a concrete software cost estimation model.

If we want to estimate cost-effort of a software project. We need to know the size of the software. One of the first software metric to measure the size of the software as length is the LOC (Line of Code) The LOC measure is used extensively in COCOMO, cost estimation model. Another size metric is Function points (FP) that reflect the user's view of a system's functionality and gives size as functionality. A unit (the function point) expresses the amount of information processing that an application offers the user. The unit is separate from the way in which the information processing is carried out technically.

Because software is a high-level notion made up of many different attributes, there can never be a single measure of software complexity. Most of the complexity metrics are also restricted to code. The best knowns are Halstead's Software Science and McCabes's cyclomatic number. Halstead defined a range of metrics based on the operators and operands in a program. McCabe's metrics is derived from the program's control flow graph.

2.3.7 Halstead's Software Science

The Software Science developed by M. H. Halstead principally attempts to estimate the programming effort.

The measurable and countable properties are:

- n_1 = number of unique or distinct operators appearing in that implementation
- n_2 = number of unique or distinct operands appearing in that implementation
- N_1 = total usage of all of the operators appearing in that implementation
- N_2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead defines:

- The vocabulary n as $n = n_1 + n_2$
- The implementation length N as $N = N_1 + N_2$

Operators can be "+" and "*" but also an index "[...]" or a statement separation "..;..". The number of operands consists of the numbers of literal expressions, constants and variables.

Length Equation

It may be necessary to know about the relationship between length N and vocabulary n. Length Equation is as follows. " ' " on N means it is calculated rather than counted :

$N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$

It is experimentally observed that N ' gives a rather close agreement to program length.

Quantification of Intelligence Content

The same algorithm needs more consideration in a low level programming language. It is easier to program in Pascal rather than in assembly. The intelligence Content determines how much is said in a program. In order to find Quantification of Intelligence Content we need some other metrics and formulas:

Program Volume: This metric is for the size of any implementation of any algorithm.

$V = N \log_2 n$

Program Level: It is the relationship between Program Volume and Potential Volume. Only the clearest algorithm can have a level of unity.

$L = V^* / V$

Program Level Equation: is an approximation of the equation of the Program Level. It is used when the value of Potential Volume is not known because it is possible to measure it from an implementation directly.

$L' = n^*_1 n_2 / n_1 N_2$

Intelligence Content

$I = L' \times V = (2n_2 / n_1N_2) \times (N_1 + N_2) \log2 (n_1 + n_2)$

In this equation all terms on the right-hand side are directly measurable from any expression of an algorithm. The intelligence content is correlated highly with the potential volume. Consequently, because potential volume is independent of the language, the intelligence content should also be independent.

Programming Effort

The programming effort is restricted to the mental activity required to convert an existing algorithm to an actual implementation in a programming language. In order to find Programming effort we need some metrics and formulas: **Potential Volume**: is a metric for denoting the corresponding parameters in an algorithm's shortest possible form. Neither operators nor operands can require repetition.

$V' = (n^*_1 + n^*_2) \log_2 (n^*_1 + n^*_2)$

Effort Equation

The total number of elementary mental discriminations is:

$E = V / L = V^2 / V'$

If we express it: The implementation of any algorithm consists of N selections (nonrandom > of a vocabulary n. a program is generated by making as many mental comparisons as the program volume equation determines, because the program volume V is a measure of it. Another aspect that influences the effort equation is the program difficulty. Each mental comparison consists of a number of elementary mental discriminations. This number is a measure for the program difficulty.

Time Equation

A concept concerning the processing rate of the human brain, developed by the psychologist John Stroud, can be used. Stroud defined a moment as the time required by the human brain to perform the most elementary discrimination. The Stroud number S is then Stroud's moments per second with 5 <= S <= 20. Thus we can derive the time equation where, except for the Stroud number S, all of the parameters on the right are directly measurable:

T ' = (n1N2 (n1log2n1 + $n_2 log_2 n_2$) $log_2 n$) / $2n_2 S$

Advantages of Halstead:

- Do not require in-depth analysis of programming structure.
- Predicts rate of error.
- Predicts maintenance effort.
- Useful in scheduling and reporting projects.
- Measure overall quality of programs.
- Simple to calculate.
- Can be used for any programming language.
- Numerous industry studies support the use of Halstead in predicting programming effort and mean number of programming bugs.

Drawbacks of Halstead

- It depends on completed code.
- It has little or no use as a predictive estimating model. But McCabe's model is more suited to application at the design level.

2.3.8 McCabe's Cyclomatic number: Cyclomatic complexity

Cyclomatic complexity is a software metric (measurement) in computational complexity theory. It was developed by Thomas McCabe and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to the commands of a program. A directed edge connects two nodes if the second command might be executed immediately after the first command.

Definition

$M = E - N + P$

where

M = cyclomatic complexity

E = the number of edges of the graph

N = the number of nodes of the graph

P = the number of connected components.

"M" is alternatively defined to be one larger than the number of decision points (IFs, UNTILs, ENDs...) in a module (function, procedure, chart node, etc.), or more generally a system.

Alternative definition

$v(G) = e - n + 2$

G is a program's flow graph

e is the number of arcs in the flow graph

n is the number of nodes in the flow graph

Alternative way

There is another simple way to determine the cyclomatic number. This is done by counting the number of closed loops in the flow graph, and incrementing that number by one.

i.e.

M = Number of closed loops + 1

Where

M = Cyclomatic number.

Implications for Software Testing

- M is a lower bound for the number of possible paths through the control flow graph.
- M is an upper bound for the number of test cases that are necessary to achieve complete branch coverage.

For example, consider a program that consists of two sequential if-then-else statements.

```
if (c1)    { f1(); }
else
           { f2(); }
if (c2)    { f3(); }
else
           { f4();}
```

- To achieve complete branch coverage, two test cases are sufficient here.
- For complete path coverage, four test cases are necessary.

- The cyclomatic number M is three, falling in the range between these two values, as it does for any program.

Key Concept

The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.

Cyclomatic complexity is normally calculated by creating a graph of the source code with each line of source code being a node on the graph and arrows between the nodes showing the execution pathways. As some programming languages can be quite terse and compact, a source code statement when developing the graph may actually create several nodes in the graph (for instance when using the C and C++ language "?" conditional operator (also known as the "ternary operator") within a function call interface).

In general, in order to fully test a module all execution paths through the module should be exercised. This implies a module with a high complexity number requires more testing effort than a module with a lower value since the higher complexity number indicates more pathways through the code. This also implies that a module with higher complexity is more difficult for a programmer to understand since the programmer must understand the different pathways and the results of those pathways.

One would also expect that a module with higher complexity would tend to have lower cohesion (less than functional cohesion) than a module with lower complexity. The possible correlation between higher complexity measure with a lower level of cohesion is predicated on a module with more decision points generally implementing more than a single well defined function. However there are certain types of modules that one would expect to have a high complexity number, such as user interface (UI) modules containing source code for data validation and error recovery.

The results of multiple experiments (G.A. Miller) suggest that modules approach zero defects when McCabe's Cyclomatic Complexity is within 7 ± 2. A study of PASCAL and FORTRAN programs found that a Cyclomatic Complexity between 10 and 15 minimized the number of module changes.

Advantages of McCabe Cyclomatic Complexity

- It can be used as a ease of maintenance metric.
- Used as a quality metric, gives relative complexity of various designs.
- It can be computed early in life cycle than of Halstead's metrics.
- Measures the minimum effort and best areas of concentration for testing.
- It guides the testing process by limiting the program logic during development.
- Is easy to apply.

Drawbacks of McCabe Cyclomatic Complexity

- The cyclomatic complexity is a measure of the program's control complexity and not the data complexity
- the same weight is placed on nested and non-nested loops. However, deeply nested conditional structures are harder to understand than non-nested structures.
- It may give a misleading figure with regard to a lot of simple comparisons and decision structures. Whereas the fan-in fan-out method would probably be more applicable as it can track the data flow

2.3.9 Fan-In Fan-Out Complexity - Henry and Kafura's

Henry and Kafura (1981) identified a form of the fan in - fan out complexity, which maintains a count of the number of data flows from a component plus the number of global data structures that the program updates. The data flow count includes updated procedure parameters and procedures called from within a module.

Complexity = Length x (Fan-in x Fan-out)^2

Length is any measure of length such as lines of code or alternatively McCabe's cyclomatic complexity is sometimes substituted.

Henry and Kafura validated their metric using the UNIX system and suggested that the measured complexity of a component allowed potentially faulty system components to be identified. They found that high values of this metric were often measured in components where there had historically been a high number of problems.

Advantages of Henry's and Kafura's Metic

- it takes into account data-driven programs

- it can be derived prior to coding, during the design stage

Drawbacks of Henry's and Kafura's Metic

- it can give complexity values of zero if a procedure has no external interactions

2.3.10 Source lines of code (SLOC)

The basis of the Measure SLOC is that program length can be used as a predictor of program characteristics such as effort and ease of maintenance. The LOC measure is used to measure size of the software. Source lines of code (SLOC) is a software metric used to measure the amount of code in a software program. SLOC is typically used to estimate the amount of effort that will be required to develop a program, as well as to estimate productivity or effort once the software is produced.

There are versions of LOC:

DSI (Delivered Source Instructions)

It is used in COCOMO'81 as KDSI (Means thousands of Delivered Source Instructions). DSI is defined as follows:

- Only Source lines that are DELIVERED as part of the product are included -- test drivers and other support software is excluded
- SOURCE lines are created by the project staff -- code created by applications generators is excluded
- One INSTRUCTION is one line of code or card image
- Declarations are counted as instructions
- Comments are not counted as instructions

Advantages of LOC

- Simple to measure

Drawbacks of LOC

- It is defined on code. For example it cannot measure the size of specification.
- It characterize only one specific view of size, namely length, it takes no account of functionality or complexity
- Bad software design may cause excessive line of code
- It is language dependent
- Users cannot easily understand it

Because of the critics above there have been extensive efforts to characterize other products size attributes, notably complexity and functionality.

Measuring SLOC

Many useful comparisons involve only the order of magnitude of lines of code in a project. Software projects can vary between 100 to 100,000,000 lines of code. Using lines of code to compare a 10,000 line project to a 100,000 line project is far more useful than when comparing a 20,000 line project with a 21,000 line project. While it is debatable exactly how to measure lines of code, wide discrepancies in 2 different measurements should not vary by an order of magnitude.

There are two major types of SLOC measures: physical SLOC and logical SLOC. Specific definitions of these two measures vary, but the most common definition of physical SLOC is a count of lines in the text of the program's source code including comment lines. Blanks lines are also included unless the lines of code in a section consist of more than 25% blank lines. In this case blank lines in excess of 25% are not counted toward lines of code.

Logical SLOC measures attempt to measure the number of "statements", but their specific definitions are tied to specific computer languages (one simple logical SLOC measure for C-like languages is the number of statement-terminating semicolons). It is much easier to create tools that measure physical SLOC, and physical SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions. Unfortunately, SLOC measures are often stated without giving their definition, and logical SLOC can often be significantly different from physical SLOC.

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC:

```
for (i=0; i<100; ++i) printf("hello"); /* How many lines of code is this? */
```

In this example we have:

- 1 Physical Lines of Code LOC
- 2 Logical Line of Code lLOC (for statement and printf statement)
- 1 Comment Line

Depending on the programmer and/or coding standards, the above "line of code" could be, and usually is, written on many separate lines:

```
for (i=0; i<100; ++i)
{
    printf("hello");
} /* Now how many lines of code is this? */
```

In this example we have:

- ➢ 4 Physical Lines of Code LOC (Is placing braces work to be estimated?)
- ➢ 2 Logical Line of Code lLOC (What about all the work writing non-statement lines?)
- ➢ 1 Comment Line (Tools must account for all code and comments regardless of comment placement.)

Even the "logical" and "physical" SLOC values can have a large number of varying definitions. Robert E. Park (while at the Software Engineering Institute) et al. developed a framework for defining SLOC values, to enable people to carefully explain and define the SLOC measure used in a project. For example, most software systems reuse code, and determining which (if any) reused code to include is important when reporting a measure.

Origins of SLOC

At the time that people began using SLOC as a metric, the most commonly used languages, such as FORTRAN and assembler, were line-oriented languages. These languages were developed at the time when punch cards were the main form of data entry for programming. One punch card usually represented one line of code. It was one discrete object that was easily counted. It was the visible output of the programmer so it made sense to managers to count lines of code as a measurement of a programmer's productivity. Today, the most commonly used computer languages allow a lot more leeway for formatting. One line of text no longer necessarily corresponds to one line of code.

Usage of SLOC measures

SLOC measures are somewhat controversial, particularly in the way that they are sometimes misused. Experiments have repeatedly confirmed that effort is highly correlated with SLOC, that is, programs with larger SLOC values take more time to develop. Thus, SLOC can be very effective in estimating effort. However, functionality is less well correlated with SLOC: skilled developers may be able to develop the same functionality with far less code, so one program with less SLOC may exhibit more functionality than another similar program. In particular, SLOC is a poor productivity measure of individuals, since a developer can develop only a few lines and yet be far more productive in terms of functionality than a developer who ends up creating more lines (and generally spending more effort). Good developers may merge multiple code modules into a single module, improving the system yet appearing to have negative productivity because they remove code. Also, especially skilled developers tend to be assigned the most difficult tasks, and thus may sometimes appear less "productive" than other developers on a task by this measure.

SLOC is particularly ineffective at comparing programs written in different languages unless adjustment factors are applied to normalize languages. Various computer languages balance brevity and clarity in different ways; as an extreme example, most assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL. The following example shows a comparison of a "Hello World" program written in C, and the same program written in COBOL - a language known for being particularly verbose.

Program in C

```
#include <stdio.h>
int main(void) {
   printf("Hello World");
   return 0;
}
```

Lines of code: 5 (excluding white space)

Program in COBOL

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID.   HELLOWORLD.
000300
000400*
000500  ENVIRONMENT DIVISION.
000600  CONFIGURATION          SECTION.
000700  SOURCE-COMPUTER. RM-COBOL.
000800  OBJECT-COMPUTER. RM-COBOL.
000900
001000  DATA    DIVISION.
001100  FILE SECTION.
```

```
001200
100000  PROCEDURE DIVISION.
100100
100200  MAIN-LOGIC SECTION.
100300  BEGIN.
100400  DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500  DISPLAY "Hello world!" LINE 15 POSITION 10.
100600  STOP    RUN.
100700  MAIN-LOGIC-EXIT.
100800  EXIT.
```

Lines of code: 17 (excluding white space)

Another increasingly common problem in comparing SLOC metrics is the difference between auto-generated and hand-written code. Modern software tools often have the capability to auto-generate enormous amounts of code with a few clicks of a mouse. For instance, GUI builders automatically generate all the source code for a GUI object simply by dragging an icon onto a workspace. The work involved in creating this code cannot reasonably be compared to the work necessary to write a device driver, for instance.

There are several cost, schedule, and effort estimation models which use SLOC as an input parameter, including the widely-used Constructive Cost Model (COCOMO) series of models by Barry Boehm et al and Galorath's SEER-SEM. While these models have shown good predictive power, they are only as good as the estimates (particularly the SLOC estimates) fed to them. Many have advocated the use of function points instead of SLOC as a measure of functionality, but since function points are highly correlated to SLOC (and cannot be automatically measured) this is not a universally held view.

According to Andrew Tanenbaum, the SLOC values for various operating systems in Microsoft's Windows NT product line are as follows:

Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	6
1994	Windows NT 3.5	10
1996	Windows NT 4.0	16
2000	Windows 2000	29
2002	Windows XP	40
2005	Windows Vista Beta 2	50

David A. Wheeler studied the Red Hat distribution of the GNU/Linux operating system, and reported that Red Hat Linux version 7.1 (released April 2001) contained over 30 million physical SLOC. He also determined that, had it been developed by conventional proprietary means, it would have required about 8,000 person-years of development effort and would have cost over $1 billion (in year 2000 U.S. dollars).

A similar study was later made of Debian GNU/Linux version 2.2 (also known as "Potato"); this version of GNU/Linux was originally released in August 2000. This study found that Debian GNU/Linux 2.2 included over 55 million SLOC, and if developed in a conventional proprietary way would have required 14,005 person-years and cost $1.9 billion USD to develop. Later runs of the tools used report that the following release of Debian had 104 million SLOC, and as of year 2005, the newest release is going to include over 213 million SLOC.

Operating System	**SLOC (Million)**
Red Hat Linux 6.2	17
Red Hat Linux 7.1	30
Debian 2.2	56
Debian 3.0	104
Debian 3.1	213
Sun Solaris	7.5
Mac OS X 10.4	86
Linux kernel 2.6.0	6.0

Graphics Program	**SLOC (Million)**
Blender 2.42	~1
Gimp-2.3.8	0.65

SLOC and relation to security faults

"The central enemy of reliability is complexity" Geer et al.

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight." ~ Bill Gates.

A number of experts have claimed a relationship between the number of lines of code in a program and the number of bugs that it contains. This relationship is not simple, since the number of errors per line of code varies greatly according to the language used, the type of quality assurance processes, and level of testing, but it does appear to exist. More importantly, the number of bugs in a program has been directly related to the number of security faults that are likely to be found in the program.

This has had a number of important implications for system security and these can be seen reflected in operating system design. Firstly, more complex systems are likely to be more insecure simply due to the greater number of lines of code needed to develop them. For this reason, security focused systems such as OpenBSD grow much more slowly than other systems such as Windows and Linux. A second idea, taken up in both OpenBSD and many Linux variants, is that separating code into different sections which run with different security environments (with or without special privileges, for example) ensures that the most security critical segments are small and carefully audited.

Advantages

- Scope for Automation of Counting: Since Line of Code is a physical entity; manual counting effort can be easily eliminated by automating the counting process. Small utilities may be developed for counting the LOC in a program. However, a code counting utility developed for a specific language cannot be used for other languages due to the syntactical and structural differences among languages.
- An Intuitive Metric: Line of Code serves as an intuitive metric for measuring the size of software due to the fact that it can be seen and the effect of it can be visualized. Function Point is more of an objective metric which cannot be

imagined as being a physical entity, it exists only in the logical space. This way, LOC comes in handy to express the size of software among programmers with low levels of experience.

Disadvantages

- Lack of Accountability: Lines of code measure suffers from some fundamental problems. Some think it isn't useful to measure the productivity of a project using only results from the coding phase, which usually accounts for only 30% to 35% of the overall effort.
- Lack of Cohesion with Functionality: Though experiments have repeatedly confirmed that effort is highly correlated with LOC, functionality is less well correlated with LOC. That is, skilled developers may be able to develop the same functionality with far less code, so one program with less LOC may exhibit more functionality than another similar program. In particular, LOC is a poor productivity measure of individuals, since a developer can develop only a few lines and still be more productive than a developer creating more lines of code.
- Adverse Impact on Estimation: As a consequence of the fact presented under point (a), estimates done based on lines of code can adversely go wrong, in all possibility.
- Developer’s Experience: Implementation of a specific logic differs based on the level of experience of the developer. Hence, number of lines of code differs from person to person. An experienced developer may implement

certain functionality in fewer lines of code than another developer of relatively less experience does, though they use the same language.

- Difference in Languages: Consider two applications that provide the same functionality (screens, reports, databases). One of the applications is written in C++ and the other application written a language like COBOL. The number of function points would be exactly the same, but aspects of the application would be different. The lines of code needed to develop the application would certainly not be the same. As a consequence, the amount of effort required to develop the application would be different (hours per function point). Unlike Lines of Code, the number of Function Points will remain constant.
- Advent of GUI Tools: With the advent of GUI-based languages/tools such as Visual Basic, much of development work is done by drag-and-drops and a few mouse clicks, where the programmer virtually writes no piece of code, most of the time. It is not possible to account for the code that is automatically generated in this case. This difference invites huge variations in productivity and other metrics with respect to different languages, making the Lines of Code more and more irrelevant in the context of GUI-based languages/tools, which are prominent in the present software development arena.
- Problems with Multiple Languages: In today's software scenario, software is often developed in more than one language. Very often, a number of languages are employed depending on the complexity and requirements. Tracking and reporting of productivity and defect rates poses a serious problem in this case since defects cannot be attributed to a particular

language subsequent to integration of the system. Function Point stands out to be the best measure of size in this case.

- Lack of Counting Standards: There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? – These are the questions that often arise. Though organizations like SEI and IEEE have published some guidelines in an attempt to standardize counting, it is difficult to put these into practice especially in the face of newer and newer languages being introduced every year.

2.3.11 Function Points (FP)

Function points is basic data from which productivity metrics could be computed.

FP data is used in two ways:

- as an estimation variable that is used to "size" each element of the software,
- as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

The approach is to identify and count a number of unique function types:

- External inputs (e.g. file names)
- External outputs (e.g. reports, messages)
- Queries (interactive inputs needing a response)
- External files or interfaces (files shared with other software systems)
- Internal files (invisible outside the system)

Each of these is then individually assessed for complexity and given a weighting value, which varies from 3 (for simple external inputs) to 15 (for complex internal files).

Unadjusted function points (UFP) is calculated as follows:
The sum of all the occurrences is computed by multiplying each function count with a weighting and then adding up all the values. The weights are based on the complexity of the feature being counted. Albrecht's original method classified the weightings as:

Function Type	**Low**	**Average**	**High**
External Input	x3	x4	x6
External Input	x4	x5	x7
Logical Internal File	x7	x10	x15
External Interface File	x5	x7	x10
External Inquiry	x3	x4	x6

Low, average and high decision can be determined with this table:

	1-5 Data element types	6-19 Data element types	20+ Data element types
0-1 File types referenced	Low	Low	Average
2-3 File types referenced	Low	Average	High
4+ File types referenced	Average	High	High

In order to find adjusted FP, UFP is multiplied by technical complexity factor (TCF) which can be calculated by the formula:

TCF = 0.65 + (sum of factors) / 100

There are 14 technical complexity factors. Each complexity factor is rated on the basis of its degree of influence, from no influence to very influential:

1. Data communications
2. Performance
3. Heavily used configuration
4. Transaction rate
5. Online data entry
6. End user efficiency
7. Online update
8. Complex processing
9. Reusability
10. Installation ease
11. Operations ease
12. Multiple sites
13. Facilitate change
14. Distributed functions

Then FP = UFP x TCF

Function points are recently used also for real time systems.

Advantages of FP

- It is not restricted to code
- Language independent
- The necessary data is available early in a project. We need only a detailed specification.
- More accurate than estimated LOC

Drawbacks of FP

- Subjective counting
- Hard to automate and difficult to compute
- Ignores quality of output
- Oriented to traditional data processing applications
- Effort prediction using the unadjusted function count is often no worse than when the TCF is added.

Organizations such as the International Function Point Users Group IFPUG have been active in identifying rules for function point counting to ensure that counts are comparable across different organizations.

At IFPUG's Message Board Home Page you can find solutions about practical use of FP.

If sufficient data exists from previous programs, function points can reasonably be converted to an LOC estimate.

2.4 Summary

Software metrics have been partitioned into process metric, project metric and product metric. These metrics help the managers in improving the software processes; assist in the planning, tracking and control of a software project; and assess the quality of the product. There are a number of software metrics such as Source lines of code, Cyclomatic complexity, Function point analysis, Code coverage, Number of classes and interfaces, Cohesion, Coupling etc. Line of Code (LOC) is a measure of size. It is very easy to compute but have limited applicability. Cyclomatic complexity is a measure of the complexity of the program, based on the control constructs available in the program. Halstead measure computes the complexity of the software on the basis of operators and operands present in the program. Function Point (FP) is also a measure of size that can be used in the early stage of the software development. It reflects the size on the basis of input, outputs, external interfaces, queries, and number of files required. Metrics are the tools that help in better monitoring and control.

2.5 Key words

Cyclomatic complexity: The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code.

Function Point: Function points is basic data from which productivity metrics could be computed, based on external inputs/outputs, files, interfaces, and filrs.

Halstead software science: The Software Science developed by M. H. Halstead principally that attempts to estimate the programming effort.

Software metric: A software metric is a measure of some property of a piece of software or its specifications.

Lines of Code (LOC): It is a software metric that measures the size of software in terms of lines in the program.

2.6 Self-Assessment Questions

1. What do you understand by complexity of software? What is cyclomatic complexity? Write a program for bubble sort and compute its cyclomatic complexity.
2. What are the different approaches to compute the cyclomatic complexity? Explain using suitable examples.
3. Define metric. What are the desirable characteristics of a metric?
4. What do you understand by SLOC? What are its advantages and disadvantages? Explain.
5. What is Halstead software science? What are the different measures available in it? Explain.

2.7 References/Suggested readings

5. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
6. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
7. Software Engineering by Sommerville, Pearson Education.
8. Software Engineering – A Practitioner’s Approach by Roger S Pressman, McGraw-Hill.

Lesson number: 3	**Writer: Dr. Rakesh Kumar**
Software Life Cycle Models	**Vetter: Sh. Naresh Mann**

3.0 Objectives

The objective of this lesson is to introduce the students to the concepts of software life cycle models. After studying this lesson, they will:

- Understand a number of process models like waterfall model, spiral model, prototyping and iterative enhancement.
- Come to know the merits/demerits, and applicability of different models.

3.1 Introduction

A software process is a set of activities and associated results which lead to the production of a software product. There are many different software processes; there are fundamental activities which are common to all software processes. These are:

- **Software specification:** The functionality of the software and constraints on its operations are defined.
- **Software design and implementation:** The software to meet the specification is produced.
- **Software validation:** The software must be validated to ensure that it does what the customer wants.
- **Software evolution:** The software must evolve to meet changing customer needs.

Software process model is an abstract representation of a software process. A number of software models are discussed in this lesson.

3.2 Presentation of contents

3.2.1 Software Project Planning

Lack of planning is the primary cause of schedule slippage, cost overrun, poor quality, and high maintenance cost. So a careful planning is required to avoid such problems. The steps required to plan a software project are given below:

Planning a software project
Defining the problem 1. Develop a statement of the problem including the description of present

situation, problem constraints, and a statement of the goals to be achieved.

2. Identify the functions to be provided, constraints, harware/software/people subsystem.
3. Determine system level goals and the requirements for development process and the work products.
4. Decide the acceptance criteria for system.

Develop a solution strategy

1. Outline several solution strategies.
2. Conduct a feasibility study for each strategy.
3. Recommend a solution strategy.
4. Develop a list of priorities for product characteristics.

Planning the development process

1. Define a life cycle model and organization structure.
2. Plan the configuration management, quality assurance, and validation activities.
3. Decide the phase dependent tools, techniques and notations.
4. Establish cost estimate.
5. Estable development schedule.
6. Establish staffing estimate.
7. Develop estimate of the computing resources required to operate and maintain the system.
8. Prepare a glossary of terms.
9. Identify source of information.

As stated above, planning the development process includes a very important consideration i.e. defining a life cycle model. The software life cycle includes all the activities required to define, develop, test, deliver, operate, and maintain a system. A number of models are discussed here.

3.2.2 Water fall model

The waterfall model is a sequential software development model in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The origin of the term "waterfall" is often cited to be an article published in 1970 by W. W. Royce; ironically, Royce himself advocated an iterative approach to software development and did not even use the term "waterfall". Royce originally described what is now known as the waterfall model as an example of a method that he argued "is risky and invites failure".

3.2.2.1 History of the waterfall model

In 1970 Royce proposed what is now popularly referred to as the waterfall model as an initial concept, a model which he argued was flawed. His paper then explored how the initial model could be developed into an iterative model, with feedback from each phase influencing previous phases, similar to many methods used widely and highly regarded by many today. Ironically, it is only the initial model that received notice; his own criticism of this initial model has been largely ignored. The "waterfall model" quickly came to refer not to Royce's final, iterative design, but rather to his purely sequentially ordered model.

Despite Royce's intentions for the waterfall model to be modified into an iterative model, use of the "waterfall model" as a purely sequential process is still popular, and, for some, the phrase "waterfall model" has since come to refer to any approach to software creation which is seen as inflexible and non-iterative.

3.2.2.2 Usage of the waterfall model

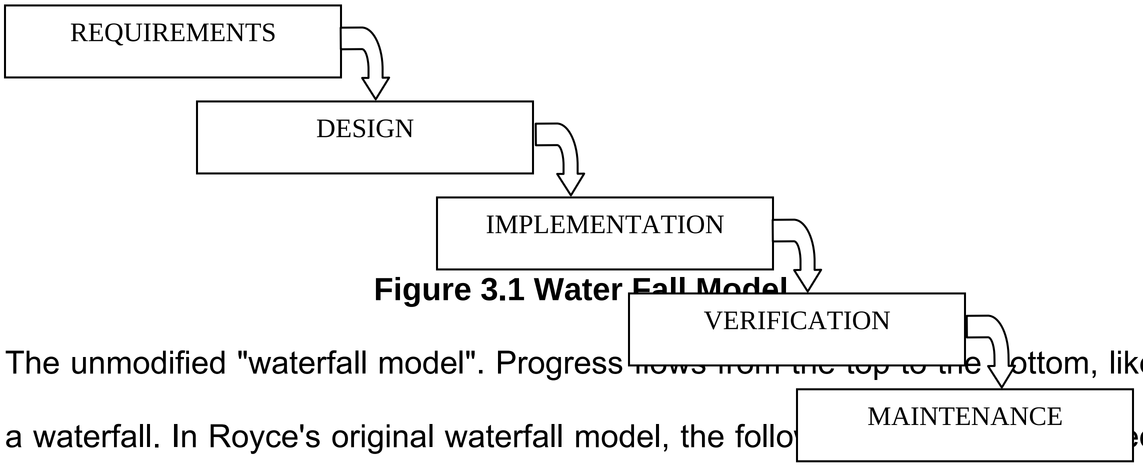


Figure 3.1 Water Fall Model

The unmodified "waterfall model". Progress flows from the top to the bottom, like a waterfall. In Royce's original waterfall model, the follo[illegible]ed perfectly in order:

1. Requirements specification
2. Design
3. Construction (implementation or coding)
4. Integration
5. Testing and debugging (verification)
6. Installation
7. Maintenance

To follow the waterfall model, one proceeds from one phase to the next in a purely sequential manner. For example, one first completes "requirements specification" — they set in stone the requirements of the software. When and only when the requirements are fully completed, one proceeds to design. The software in question is designed and a "blueprint" is drawn for implementers (coders) to follow — this design should be a plan for implementing the requirements given. When and only when the design is fully completed, an implementation of that design is made by coders. Towards the later stages of this implementation phase, disparate software components produced by different teams are integrated. After the implementation and integration phases are complete, the software product is tested and debugged; any faults introduced in earlier phases are removed here. Then the software product is installed, and later maintained to introduce new functionality and remove bugs.

Thus the waterfall model maintains that one should move to a phase only when its preceding phase is completed and perfected. Phases of development in the waterfall model are thus discrete, and there is no jumping back and forth or overlap between them.

However, there are various modified waterfall models that may include slight or major variations upon this process.

3.2.2.3 Arguments for the water fall model

Time spent early on in software production can lead to greater economy later on in the software lifecycle; that is, it has been shown many times that a bug found in the early stages of the production lifecycle (such as requirements specification or design) is more economical (cheaper in terms of money, effort and time) to fix than the same bug found later on in the process. (it is said that "a requirements defect that is left undetected until construction or maintenance will cost 50 to 200 times as much to fix as it would have cost to fix at requirements time.") This should be obvious to some people; if a program design is impossible to implement, it is easier to fix the design at the design stage than to realize months down the track when program components are being integrated that all the work done so far has to be scrapped because of a broken design.

This is the central idea behind Big Design Up Front (BDUF) and the waterfall model - time spent early on making sure that requirements and design are absolutely correct is very useful in economic terms (it will save you much time and effort later). Thus, the thinking of those who follow the waterfall process goes, one should make sure that each phase is 100% complete and absolutely correct before proceeding to the next phase of program creation. Program requirements should be set in stone before design is started (otherwise work put into a design based on "incorrect" requirements is wasted); the programs design should be perfect before people begin work on implementing the design (otherwise they are implementing the "wrong" design and their work is wasted), etc.

A further argument for the waterfall model is that it places emphasis on documentation (such as requirements documents and design documents) as well as source code. More "agile" methodologies can de-emphasize documentation in favor of producing working code - documentation however can be useful as a "partial deliverable" should a project not run far enough to produce any substantial amounts of source code (allowing the project to be resumed at a later date). An argument against agile development methods, and thus partly in favour of the waterfall model, is that in agile methods project knowledge is stored mentally by team members. Should team members leave, this knowledge is lost, and substantial loss of project knowledge may be difficult for a project to recover from. Should a fully working design document be present (as is the intent of Big Design Up Front and the waterfall model) new team members or even entirely new teams should theoretically be able to bring themselves "up to speed" by reading the documents themselves. With that said, agile methods do attempt to compensate for this. For example, extreme programming (XP) advises that project team members should be "rotated" through sections of work in order to familiarize all members with all sections of the project (allowing individual members to leave without carrying important knowledge with them).

As well as the above, some prefer the waterfall model for its simple and arguably more disciplined approach. Rather than what the waterfall adherent sees as "chaos" the waterfall model provides a structured approach; the model itself progresses linearly through discrete, easily understandable and explainable "phases" and is thus easy to understand; it also provides easily markable "milestones" in the development process. It is argued that the waterfall model and Big Design Up Front in general can be suited to software projects which are stable (especially those projects with unchanging requirements, such as with "shrink wrap" software) and where it is possible and likely that designers will be able to fully predict problem areas of the system and produce a correct design before implementation is started. The waterfall model also requires that implementers follow the well made, complete design accurately, ensuring that the integration of the system proceeds smoothly.

The waterfall model is widely used, including by such large software development houses as those employed by the US Department of Defense and NASA and upon many large government projects. Those who use such methods do not always formally distinguish between the "pure" waterfall model and the various modified waterfall models, so it can be difficult to discern exactly which models are being used to what extent.

Steve McConnell sees the two big advantages of the pure waterfall model as producing a "highly reliable system" and one with a "large growth envelope", but rates it as poor on all other fronts. On the other hand, he views any of several modified waterfall models (described below) as preserving these advantages while also rating as "fair to excellent" on "work with poorly understood requirements" or "poorly understood architecture" and "provide management with progress visibility", and rating as "fair" on "manage risks", being able to "be constrained to a predefined schedule", "allow for midcourse corrections", and "provide customer with progress visibility". The only criterion on which he rates a modified waterfall as poor is that it requires sophistication from management and developers.

3.2.2.4 Criticism of the waterfall model

The waterfall model however is argued by many to be a bad idea in practice, mainly because of their belief that it is impossible to get one phase of a software product's lifecycle "perfected" before moving on to the next phases and learning from them (or at least, the belief that this is impossible for any non-trivial program). For example clients may not be aware of exactly what requirements they want before they see a working prototype and can comment upon it - they may change their requirements constantly, and program designers and implementers may have little control over this. If clients change their requirements after a design is finished, that design must be modified to accommodate the new requirements, invalidating quite a good deal of effort if overly large amounts of time have been invested into "Big Design Up Front". (Thus methods opposed to the naive waterfall model, such as those used in Agile software development advocate less reliance on a fixed, static requirements document or design document). Designers may not be aware of future implementation difficulties when writing a design for an unimplemented software product. That is, it may become clear in the implementation phase that a particular area of program functionality is extraordinarily difficult to implement. If this is the case, it is better to revise the design than to persist in using a design that was made based on faulty predictions and which does not account for the newly discovered problem areas.

Steve McConnell in Code Complete (a book which criticizes the widespread use of the waterfall model) refers to design as a "wicked problem" - a problem whose requirements and limitations cannot be entirely known before completion. The implication is that it is impossible to get one phase of software development "perfected" before time is spent in "reconnaissance" working out exactly where and what the big problems are.

To quote from David Parnas' "a rational design process and how to fake it ":

“Many of the [systems] details only become known to us as we progress in the [systems] implementation. Some of the things that we learn invalidate our design and we must backtrack.”

The idea behind the waterfall model may be "measure twice; cut once", and those opposed to the waterfall model argue that this idea tends to fall apart when the problem being measured is constantly changing due to requirement modifications and new realizations about the problem itself. The idea behind those who object to the waterfall model may be "time spent in reconnaissance is seldom wasted".

In summary, the criticisms of waterfall model are as follows:

- **Changing Software Requirements:** Software techniques and tools exist for identifying ambiguous and missing software requirements. These problems are important factors in the development of any software system. However, the problems are further complicated with changing software requirements. The development length of large-scale software systems is such that changing requirements are a significant problem that leads to increased development costs. Software requirements formulation and analysis is even more difficult in complex application domains

  <u>Management Implications</u>

  The problems that changing requirements introduce into the software life cycle are reflected in schedule slippages and cost overruns. One argument is that more time spent upstream in the software life cycle results in less turmoil downstream in the life cycle. The more time argument is typically false when the software requirements and specification technique is a natural language.

  <u>Product Implications</u>

  The product implications are quality-based aspects of the system during software development and maintenance. The requirements problems listed above have a ripple effect throughout the development of a software system.

Even with this advanced technology, changing requirements are to be expected, but there would be the environment for control and discipline with the changes.

- Unless those who specify requirements and those who design the software system in question are highly competent, it is difficult to know exactly what is needed in each phase of the software process before some time is spent in the phase "following" it. That is, feedback from following phases is needed to complete "preceding" phases satisfactorily. For example, the design phase may need feedback from the implementation phase to identify problem design areas. The counter-argument for the waterfall model is that experienced designers may have worked on similar systems before, and so may be able to accurately predict problem areas without time spent prototyping and implementing.
- Constant testing from the design, implementation and verification phases is required to validate the phases preceding them. Constant "prototype design" work is needed to ensure that requirements are non-contradictory and possible to fulfill; constant implementation is needed to find problem areas and inform the design process; constant integration and verification of the implemented code is necessary to ensure that implementation remains on track. The counter-argument for the waterfall model here is that constant implementation and testing to validate the design and requirements is only needed if the introduction of bugs is likely to be a problem. Users of the waterfall model may argue that if designers follow a disciplined process and

do not make mistakes that there is no need for constant work in subsequent phases to validate the preceding phases.

- Frequent incremental builds (following the "release early, release often" philosophy) are often needed to build confidence for a software production team and their client.
- It is difficult to estimate time and cost for each phase of the development process without doing some "recon" work in that phase, unless those estimating time and cost are highly experienced with the type of software product in question.
- The waterfall model brings no formal means of exercising management control over a project and planning control and risk management are not covered within the model itself.
- Only a certain number of team members will be qualified for each phase; thus to have "code monkeys" who are only useful for implementation work do nothing while designers "perfect" the design is a waste of resources. A counter-argument to this is that "multiskilled" software engineers should be hired over "specialized" staff.

3.2.2.5 Modified waterfall models

In response to the perceived problems with the "pure" waterfall model, many modified waterfall models have been introduced. These models may address some or all of the criticisms of the "pure" waterfall model. While all software development models will bear at least some similarity to the waterfall model, as all software development models will incorporate at least some phases similar to those used within the waterfall model, this section will deal with those closest to the waterfall model. For models which apply further differences to the waterfall model, or for radically different models seek general information on the software development process.

3.2.2.6 Royce’s final model

Royce's final model, his intended improvement upon his initial "waterfall model", illustrated that feedback could (should, and often would) lead from code testing to design (as testing of code uncovered flaws in the design) and from design back to requirements specification (as design problems may necessitate the removal of conflicting or otherwise unsatisfiable / undesignable requirements). In the same paper Royce also advocated large quantities of documentation, doing the job "twice if possible", and involving the customer as much as possible—now the basis of participatory design and of User Centered Design, a central tenet of Extreme Programming.

3.2.2.7 The Sashimi model

The sashimi model (so called because it features overlapping phases, like the overlapping fish of Japanese sashimi) was originated by Peter DeGrace. It is sometimes simply referred to as the "waterfall model with overlapping phases" or "the waterfall model with feedback". Since phases in the sashimi model overlap, information of problem spots can be acted upon during phases of the waterfall model that would typically "precede" others in the pure waterfall model. For example, since the design and implementation phases will overlap in the sashimi model, implementation problems may be discovered during the "design and implementation" phase of the development process. This helps alleviate many of the problems associated with the Big Design Up Front philosophy of the waterfall model.

3.2.3 Software Prototyping and Requirements Engineering

The conventional waterfall software life cycle model (or software process) is used to characterize the phased approach for software development and maintenance. Software life cycle phase names differ from organization to organization. The software process includes the following phases:

- requirements formulation and analysis,
- specification,
- design,
- coding,
- testing, and
- Maintenance.

Alternative software life cycle models have been proposed as a means to address the problems that are associated with the waterfall model. One alternative software life cycle model uses prototyping as a means for providing early feedback to the end user and developer.

The waterfall model allows for a changing set of means for representing an evolving software system. These documents then provide a basis for introducing errors during the software life cycle. The user often begins to receive information concerning the actual execution of the system after the system is developed. During the development of large-scale software systems, the end user, developer, and manager can become frustrated with ambiguous, missing, or changing software requirements.

3.2.3.1 Prototyping Software Systems

Software customers find it very difficult to express their requirements. Careful requirement analysis with systematic review help to reduce the uncertainty about what the system should do. However there is no substitute for trying out a requirement before agreeing to it. This is possible if the prototype is available. A software prototype supports two requirement engineering process activities:

- Requirement elicitation: System prototype helps the user in identifying his requirements better. He can experiment with it to see how the system supports their work. In this process they can get new ideas and find strength and weakness in the software.
- Requirement validation: The prototype may reveal errors and omissions in the requirements which have been proposed.

A significant risk in software development is requirement errors and omissions and prototyping help in risk analysis and reduction. So prototyping is part of requirement engineering process. But now prototyping has been introduced throughout the conventional, waterfall software life cycle model. Now a day many systems are developed using an evolutionary approach where an initial version is created quickly and modified to produce a final system.

Two forms of life cycle models, Throwaway prototyping and evolutionary prototyping, have emerged around prototyping technology.

3.2.3.2 Conventional Model and related Variations

The conventional life cycle model can allow for prototyping within any of the phases. Unfortunately, this approach can be difficult to control when, for example, coding of a user interface takes place during requirements formulation and analysis.

3.2.3.3 Evolutionary Prototyping

In evolutionary prototyping the focus is on achieving functionality for demonstrating a portion of the system to the end user for feedback and system growth. The prototype emerges as the actual system moves downstream in the life cycle. With each iteration in development, functionality is added and then translated to an efficient implementation. Also of interest is functional programming and relational programming as a means for accomplishing evolutionary prototyping.

3.2.3.4 Prototyping Pitfalls

Prototyping has not been as successful as anticipated in some organizations for a variety of reasons. Training, efficiency, applicability, and behavior can each have a negative impact on using software prototyping techniques.

Learning Curve

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

Tool Efficiency

Prototyping techniques outside the domain of conventional programming languages can have execution inefficiencies with the associated tools. The efficiency question was argued as a negative aspect of prototyping.

Applicability

Application domain has an impact on selecting a prototyping technique. There would be limited benefit to using a technique not supporting real-time features in a process control system. The control room user interface could be described, but not integrated with sensor monitoring deadlines under this approach.

Undefined Role Models for Personnel

This new approach of providing feedback early to the end user has resulted in a problem related to the behavior of the end user and developers. An end user with a previously unfortunate system development effort can be biased in future interactions with development teams.

3.2.3.5 Throwaway prototyping

The objective of throwaway prototyping is to facilitate the designers in better understanding the requirements of the user. In this model, a prototype is developed rapidly by using shortcuts and gives to the user to use. So that he will be able to identify his requirements better. To expedite the matters, functionalities may be stripped from the throwaway prototype where these functions are well understood, quality standards may be relaxed, and performance criteria ignored. Once the software requirement specification document is ready, the prototype will be discarded and the design phase will start.

3.2.3.6 Prototyping Opportunities

Not to prototype at all should simply not be an option in software development. The end user can not throw the software needs (stated in natural language) over the transom and expect the development team to return the finished software system after some period of time with no problems in the deliverables.

- Existing Investment in Maintained Systems

One of the major problems with incorporating this technology is the large investment that exists in software systems currently in maintenance. The idea of completely reengineering an existing software system with current technology is not feasible. There is, however, a threshold that exists where the expected life span of a software system justifies that the system would be better maintained after being reengineered in this technology. Total reengineering of a software system should be a planned for effort rather than as a reaction to a crisis situation. At a minimum, prototyping technology could be used on critical portions of an existing software system. This minimal approach could be used as a means to transition an organization to total reengineering.

- Adding Investment in Fully Exploiting the Technology

In many cases, an organization will decide to incorporate this advanced software prototyping technology, but the range of support for the concept varies widely. Software prototyping, as a development technique, must be integrated within an organization through training, case studies, and library development. In situations where this full range of commitment to the technology is lacking, e.g., only developer training provided, when problems begin to arise in using the technology a normal reaction of management is to revert back to what has worked in the past.

- Developer to End User Pass Off

Finally, the end user involvement becomes enhanced when changes in requirements can first be prototyped and agreed to before any development proceeds. Similarly, during development of the actual system or even later out into maintenance should the requirements change, the prototype is enhanced and agreed to before the actual changes become confirmed.

3.2.4 Iterative Enhancement

The iterative enhancement model counters the third limitation of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the water- fall model. Furthermore, as in prototyping, the increments provide feedback to the client that is useful for determining the final requirements of the system.

In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system. A project control list is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far the project is at any given step from the final system.

Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called the design phase, implementation phase, and analysis phase. The process is iterated until the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement process model is shown in Figure 3.2:

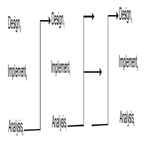


Figure 3.2 The Iterative Enhancement Model

The project control list guides the iteration steps and keeps track of all the tasks that must be done. Based on the analysis, one of the tasks in the list can include redesign, of defective components or redesign of the entire system. However, redesign of the system will generally occur only in the initial steps. In the later steps, the design would have stabilized and there is less chance of redesign. Each entry in the list is a task that should be performed in one step of the iterative enhancement process and should be simple enough to be completely understood. Selecting tasks in this manner will minimize the chances of error and reduce the redesign work. The design and implementation phases of each step can be performed in a top-down manner or by using some other technique.

One effective use of this type of model is product development, in which the developers themselves provide the specifications and, therefore, have a lot of control on what specifications go in the system and what stay out. In fact, most products undergo this type of development process. First, a version is released that contains some capability. Based on the feedback from users and experience with this version, a list of additional features and capabilities is generated. These features form the basis of enhancement of the software, and are included in the next version. In other words, the first version contains some core capability and then more features are added to later versions.

However, in a customized software development, where the client has to essentially provide and approve the specifications, it is not always clear how this process can be applied. Another practical problem with this type of development project comes in generating the business contract-how will the cost of additional features be determined and negotiated, particularly because the client organization is likely to be tied to the original vendor who developed the first version. Overall, in these types of projects, this process model can be useful if the "core" of the application to be developed is well understood and the "increments" can be easily defined and negotiated. In client-oriented projects, this process has the major advantage that the client's organization does not have to pay for the entire software together; it can get the main part of the software developed and, perform cost-benefit analysis for it before enhancing the software with more capabilities.

3.2.5 The Spiral Model

This model was originally proposed by Bohem (1988). As it is clear from the name, the activities in this model can be organized like a spiral that has many cycles. The radial dimension represents the cumulative cost incurred in accomplishing the steps done so far, and the angular dimension represents the progress made in completing each cycle of the spiral. The model is shown in Figure 3.3.

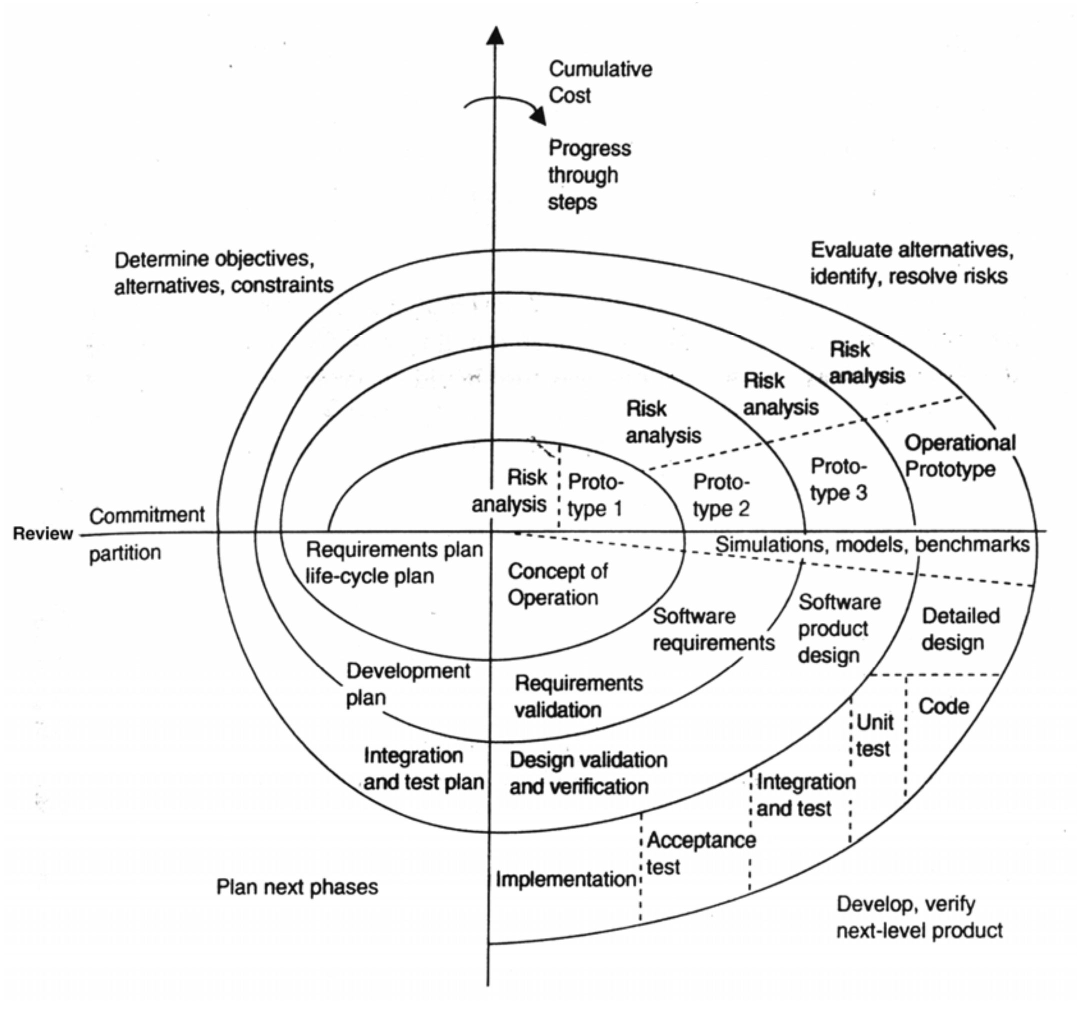


Figure 3.3 Boehm's spiral model of the software process

Each cycle in the spiral is split into four sectors:

- **Objective setting:** Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist. This is the first quadrant of the cycle (upper-left quadrant).
- **Risk Assessment and reduction:** The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project. Risks reflect the chances that some of the objectives of the project may not be met.
- **Development and validation:** The next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping.
- **Planning:** Next, the software is developed, keeping in mind the risks. Finally, the next stage is planned. The project is reviewed and a decision made whether to continue with a further cycle of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The development step depends on the remaining risks. For example, if performance or user-interface risks are considered more important than the program development risks, the next step may be an evolutionary development that involves developing a more detailed prototype for resolving the risks. On the other hand, if the program development risks dominate and the previous prototypes have resolved all the user-interface and performance risks, the next step will follow the basic waterfall approach.

The risk-driven nature of the spiral model allows it to accommodate any mixture of a specification-oriented, prototype-oriented, simulation-oriented, or some other type of approach. An important feature of the model is that each cycle of the spiral is completed by a review that covers all the products developed during that cycle, including plans for the next cycle. The spiral model works for development as well as enhancement projects.

In a typical application of the spiral model, one might start with an extra round zero, in which the feasibility of the basic project objectives is studied. These project objectives may or may not lead to a development/enhancement project. Such high-level objectives include increasing the efficiency of code generation of a compiler, producing a new full-screen text editor and developing an environment for improving productivity. The alternatives considered in this round are also typically very high-level, such as whether the organization should go for in-house development, or contract it out, or buy an existing product. In round one, a concept of operation might be developed. The objectives are stated more precisely and quantitatively and the cost and other constraints are defined precisely. The risks here are typically whether or not the goals can be met within the constraints. The plan for the next phase will be developed, which will involve defining separate activities for the project. In round two, the top-level requirements are developed. In succeeding rounds, the actual development may be done.

This is a relatively new model; it can encompass different development strategies. In addition to the development activities, it incorporates some of the management and planning activities into the model. For high-risk projects, this might be a preferred model.

3.3 Summary

Software processes are the activities involved in producing a software system. Software process models are abstract representation of these processes. There are a number of models with their merits and demerits such as water fall model, prototyping, iterative enhancement, and spiral model. The central idea behind waterfall model - time spent early on making sure that requirements and design are absolutely correct, is very useful in economic terms. A further argument for the waterfall model is that it places emphasis on documentation as well as source code; it helps the new team members to be able to bring themselves "up to speed" by reading the documents themselves. Waterfall model is preferred for its simple and arguably more disciplined approach. The waterfall model however is argued by many to be a bad idea in practice, mainly because of their belief that it is impossible to get one phase of a software product's lifecycle "perfected" before moving on to the next phases and learning from them. Clients may not be aware of exactly what requirements they want before they see a working prototype and can comment upon it - they may change their requirements constantly, and program designers and implementers may have little control over this. To sort out the limitations of water fall model, other models are proposed such as prototyping that helps the user in identifying their requirements better.

Throwaway prototyping involves developing a prototype to understand the system requirements. In evolutionary prototyping, a prototype evolves through several versions to the final system.

3.4 Keywords

Iterative enhancement: In this model, the software is developed in increments, each increment adding some functional capability to the system until the full system is implemented.

Prototyping: To avoid requirement errors and omission, prototyping is a technique facilitating requirement elicitation and validation.

Spiral model: This model was proposed by Bohem and in it the activities can be organized like a spiral that has many cycles.

Waterfall model: It is a sequential software development model in which development is seen as flowing like a waterfall through the phases of analysis, design, implementation, testing and maintenance.

3.5 Self Assessment Questions

1. What are the advantages of using waterfall model?
2. What are the limitations of water fall model? Explain.
3. What do you understand by prototyping? What is the need of it?
4. Differentiate between evolutionary and throwaway prototyping and discuss their merits and demerits.
5. Why, for large systems, it is recommended that prototypes should be throw-away prototype?
6. When should one use the iterative enhancement model?

3.6 Suggested readings/References

9. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
10. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
11. Software Engineering by Sommerville, Pearson Education.
12. Software Engineering – A Practitioner’s Approach by Roger S Pressman, McGraw-Hill.

Lesson Number: 4 **Writer: Dr. Rakesh Kumar**

SOFTWARE PROJECT PLANNING **Vetter: Dr. Yogesh Chaba**

4.0 Objectives

Project planning is an important issue in the successful completion of a software project. The objective of this lesson is to make the students familiar with the factors affecting the cost of the software, different versions of COCOMO and the problems and criteria to evaluate the models.

4.1 Introduction

Software cost estimation is the process of predicting the amount of effort required to build a software system. Software cost estimation is one of the most difficult and error prone task in software engineering. Cost estimates are needed throughout the software lifecycle. Preliminary estimates are required to determine the feasibility of a project. Detailed estimates are needed to assist with project planning. The actual effort for individual tasks is compared with estimated and planned values, enabling project managers to reallocate resources when necessary.

Analysis of historical project data indicates that cost trends can be correlated with certain measurable parameters. This observation has resulted in a wide range of models that can be used to assess, predict, and control software costs on a real-time basis. Models provide one or more mathematical algorithms that compute cost as a function of a number of variables.

4.2 Presentation of Contents

4.2.2 Cost factor

There are a number of factors affecting the cost of the software. The major one are listed below:

- **Programmer ability:** Results of the experiments conducted by Sackman show a significant difference in individual performance among the programmers. The difference between best and worst performance were factors of 6 to I in program size, 8 to 1 in execution time, 9 to 1 in development time, 18 to 1 in coding time, and 28 to 1 in debugging time.
- **Product Complexity:** There are generally three acknowledged category of the software: application programs, utility programs and system programs. According to Brook utility programs are three times as difficult to write as application programs, and that system programs are three times as difficult to write as utility programs. So it is a major factor influencing the cost of software.
- **Product Size:** It is obvious that a large software product will be more expensive than a smaller one.
- **Available time:** It is generally agreed that software projects require more total efforts if development time is compressed or expanded from the optimal time.
- **Required reliability:** Software reliability can be defined as the probability that a program will perform a required function under stated conditions for a stated period of time. Reliability can be improved in a software, but there is a cost associated with the increased level of analysis, design, implementation, verification and validation efforts that must be exerted to ensure high reliability.

- **Level of technology:** The level of technology is reflected by the programming language, abstract machine, programming practices and software tools used. Using a high level language instead of assembly language will certainly improve the productivity of programmer thus resulting into a decrease in the cost of software.

Size is a primary cost factor in most models. There are two common ways to measure software size: lines of code and function points.

Lines of Code

The most commonly used measure of source code program length is the number of lines of code (LOC). The abbreviation NCLOC is used to represent a non-commented source line of code. NCLOC is also sometimes referred to as effective lines of code (ELOC). NCLOC is therefore a measure of the uncommented length. The commented length is also a valid measure, depending on whether or not line documentation is considered to be a part of programming effort. The abbreviation CLOC is used to represent a commented source line of code. By measuring NCLOC and CLOC separately we can define:

total length (LOC) = NCLOC + CLOC

KLOC is used to denote thousands of lines of code.

Function Points

Function points (FP) measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an unadjusted function point count (UFC). Counts are made for the following categories:

- External inputs – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)
- External outputs – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
- External inquiries – interactive inputs requiring a response
- External files – machine-readable interfaces to other systems
- Internal files – logical master files in the system

Once this data has been collected, a complexity rating is associated with each count according to Table 4.1.

	Weighting Factor		
Item	Simple	Average	Complex
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

Table 4.1 Function point complexity weights.

Each count is multiplied by its corresponding complexity weight and the results are summed to provide the UFC. The adjusted function point count (FP) is calculated by multiplying the UFC by a technical complexity factor (TCF). Components of the TCF are listed in Table 4.2.

F1	Reliable back-up and recovery	F2	Data communications
F3	Distributed functions	F4	Performance
F5	Heavily used configuration	F6	Online data entry
F7	Operational ease	F8	Online update
F9	Complex interface	F10	Complex processing
F11	Reusability	F12	Installation ease
F13	Multiple sites	F14	Facilitate change

Table 4.2 Components of the technical complexity factor

Each component is rated from 0 to 5, where 0 means the component has no influence on the system and 5 means the component is essential. The TCF can then be calculated as:

TCF = 0.65 + 0.01(SUM (Fi))

The factor varies from 0.65 (if each Fi is set to 0) to 1.35 (if each Fi is set to 5). The final function point calculation is:

FP = UFC x TCF

4.2.3 Types of Models

There are two types of models that have been used to estimate cost: cost models and constraint models.

Cost Models

Cost models provide direct estimates of effort. These models typically have a primary cost factor such as size and a number of secondary adjustment factors or cost drivers. Cost drivers are characteristics of the project, process, products, or resources that influence effort. Cost drivers are used to adjust the preliminary estimate provided by the primary cost factor. A typical cost model is derived using regression analysis on data collected from past software projects. Effort is plotted against the primary cost factor for a series of projects. The line of best fit is then calculated among the data points. If the primary cost factor were a perfect predictor of effort, then every point on the graph would lie on the line of best fit. In reality however, there is usually a significant residual error. It is therefore necessary to identify the factors that cause variation between predicted and actual effort. These parameters are added to the model as cost drivers.

The overall structure of regression-based models takes the form:

$E = A + B \times S^C$

Where A, B, and C are empirically derived constants, E is effort in person months, and S is the primary input (typically either LOC or FP). The following are some examples of cost models using LOC as a primary input:

$E = 5.2 \times (KLOC)^{0.91}$	Walston-Felix Model
$E = 5.5 + 0.73 \times (KLOC)^{1.16}$	Bailey-Basili Model
$E = 3.2 \times (KLOC)^{1.05}$	COCOMO Basic Model
$E = 5.288 \times (KLOC)^{1.047}$	Doty Model for KLOC > 9

Table 4.3 Cost models using LOC as a primary input

$E = -12.39 + 0.0545\ FP$	Albrecht and Gaffney Model
$E = 60.62 \times 7.728 \times 10^{-8}\ FP^3$	Kemerer Model
$E = 585.7 + 15.12\ FP$	Matson, Barnett, and Mellichamp Model

Table 4.4 Cost models using FP as a primary input include:

Constraint Models

Constraint models demonstrate the relationship over time between two or more parameters of effort, duration, or staffing level. The RCA PRICE S model and Putnam’s SLIM model are two examples of constraint models.

Most of the work in the cost estimation field has focused on algorithmic cost modeling. In this process costs are analyzed using mathematical formulas linking costs or inputs with metrics to produce an estimated output. The formulae used in a formal model arise from the analysis of historical data. The accuracy of the model can be improved by calibrating the model to your specific development environment, which basically involves adjusting the weightings of the metrics. Generally there is a great inconsistency of estimates. Kemerer conducted a study indicating that estimates varied from as much as 85 - 610 % between predicated and actual values. Calibration of the model can improve these figures; however, models still produce errors of 50-100%.

4.2.4 SLIM (Software Life Cycle Management)

Putnam's SLIM is one of the first algorithmic cost model. It is based on the Norden / Rayleigh function and generally known as a macro estimation model (It is for large projects). SLIM enables a software cost estimator to perform the following functions:

- **Calibration** Fine tuning the model to represent the local software development environment by interpreting a historical database of past projects.
- **Build** an information model of the software system, collecting software characteristics, personal attributes, and computer attributes etc.
- **Software sizing** SLIM uses an automated version of the lines of code (LOC) costing technique.

The algorithm used is:

$K = (LOC / (C * t^{4/3})) * 3$

K is the total life cycle effort in working years, t is development and the C is the technology constant, combining the effect of using tools, languages, methodology and quality assurance (QA), time in years.

The value of technology constant varies from 610 to 57314. For easy, experienced projects technology constant is high.

SLIM is not widespreadly used but there is a SLIM tool.

Advantages of SLIM

- Uses linear programming to consider development constraints on both cost and effort.
- SLIM has fewer parameters needed to generate an estimate over COCOMO'81 and COCOMO'II

Drawbacks of SLIM

- Estimates are extremely sensitive to the technology factor
- Not suitable for small projects

4.2.5 COCOMO’81

Boehm's COCOMO model is one of the mostly used models commercially. The first version of the model delivered in 1981 and COCOMO II is available now. COCOMO 81 is a model designed by Barry Boehm to give an estimate of the number of man-months it will take to develop a software product. This "COnstructive COst MOdel" is based on a study of about sixty projects at TRW, a Californian automotive and IT company, acquired by Northrop Grumman in late 2002. The programs examined ranged in size from 2000 to 100,000 lines of code, and programming languages used ranged from assembly to PL/I.

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms.

- Basic COCOMO - is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code.
- Intermediate COCOMO - computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes.
- Detailed COCOMO - incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

4.2.5.1 Basic COCOMO 81

Basic COCOMO is a form of the COCOMO model. COCOMO may be applied to three classes of software projects. These give a general impression of the software project.

- **Organic projects** – These are relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements.
- **Semi-detached projects** – These are intermediate (in size and complexity) software projects in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements.
- **Embedded projects** – These are software projects that must be developed within a set of tight hardware, software, and operational constraints.

	Size	Innovation	Deadline/constraints	Dev. Environment
Organic	Small	Little	Not tight	Stable
Semi-detached	Medium	Medium	Medium	Medium
Embedded	Large	Greater	Tight	Complex H/W

				/customer interfaces

Table 4.5 Three classes of S/W projects for COCOMO

The basic COCOMO equations take the form

$E=a\ (KLOC)^{b}$

$D=c\ (E)^{d}$

$P=E/D$

where E is the effort applied in person-months, D is the development time in chronological months, KLOC is the estimated number of delivered lines of code for the project (expressed in thousands), and P is the number of people required. The coefficients a_b, b_b, c_b and d_b are given in the table 4.6.

Software project	a	b	c	D
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4.6 Coefficients for Basic COCOMO

Basic COCOMO is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is necessarily limited because of its lack of factors to account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and other project attributes known to have a significant influence on software costs.

4.2.5.2 Intermediate COCOMO 81

The Intermediate COCOMO is an extension of the Basic COCOMO model, and is used to estimate the programmer time to develop a software product. This extension considers a set of "cost driver attributes" that can be grouped into four major categories, each with a number of subcategories:

- Product attributes
  - Required software reliability (RELY)
  - Size of application database (DATA)
  - Complexity of the product (CPLX)
- Hardware attributes
  - Execution-time constraints (TIME)
  - Main Storage Constraints (STOR)
  - Volatility of the virtual machine environment (VIRT)
  - Required turnabout time (TURN)
- Personnel attributes
  - Analyst capability (ACAP)
  - Programmer capability (PCAP)
  - Applications experience (AEXP)
  - Virtual machine experience (VEXP)
  - Programming language experience (LEXP)
- Project attributes
  - Use of software tools (TOOL)
  - Modern Programming Practices (MODP)
  - Required development schedule (SCED)

Each of the 15 attributes is rated on a 6-point scale that ranges from "very low" to "extra high" (in importance or value). Based on the rating, an effort multiplier is determined from the table below. The product of all effort multipliers results in an 'effort adjustment factor (EAF). Typical values for EAF range from 0.9 to 1.4 as shown in table 4.7.

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	1.15	1.40	
DATA		0.94	1.00	1.08	1.16	
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
TIME			1.00	1.11	1.30	1.66
STOR			1.00	1.06	1.21	1.56
VIRT		0.87	1.00	1.15	1.30	
TURN		0.87	1.00	1.07	1.15	
ACAP	1.46	1.19	1.00	0.86	0.71	
PCAP	1.29	1.13	1.00	0.91	0.82	
AEXP	1.42	1.17	1.00	0.86	0.70	
VEXP	1.21	1.10	1.00	0.90		
LEXP	1.14	1.07	1.00	0.95		
TOOL	1.24	1.10	1.00	0.91	0.82	
MODP	1.24	1.10	1.00	0.91	0.83	
SCED	1.23	1.08	1.00	1.04	1.10	

Table 4.7 Effort adjustment factor

The Intermediate COCOMO formula now takes the form...

$$E=a\,(KLOC)^{(b)}.EAF$$

Where E is the effort applied in person-months, KLOC is the estimated number of thousands of delivered lines of code for the project and EAF is the factor calculated above. The coefficient a and the exponent b are given in the next table.

Software project	a	b
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Table 4.8 Coefficients for intermediate COCOMO

The Development time D is calculated from E in the same way as with Basic COCOMO.

The steps in producing an estimate using the intermediate model COCOMO'81 are:

1. Identify the mode (organic, semi-detached, or embedded) of development for the new product.
2. Estimate the size of the project in KLOC to derive a nominal effort prediction.
3. Adjust 15 cost drivers to reflect your project.
4. Calculate the predicted project effort using first equation and the effort adjustment factor (EAF)
5. Calculate the project duration using second equation.

Example estimate using the intermediate COCOMO'81

Mode is organic

Size = 200KDSI

Cost drivers:

- Low reliability => .88
- High product complexity => 1.15
- Low application experience => 1.13
- High programming language experience => .95
- Other cost drivers assumed to be nominal => 1.00

C = .88 * 1.15 * 1.13 * .95 = 1.086

Effort = 3.2 * ($200^{1.05}$) * 1.086 = 906 MM

Development time = 2.5 * $906^{0.38}$

4.2.5.3 Detailed COCOMO

The Advanced COCOMO model computes effort as a function of program size and a set of cost drivers weighted according to each phase of the software lifecycle. The Advanced model applies the Intermediate model at the component level, and then a phase-based approach is used to consolidate the estimate.

The 4 phases used in the detailed COCOMO model are: requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT), and integration and test (IT). Each cost driver is broken down by phase as in the example shown in Table 4.9.

Cost Driver	Rating	RPD	DD	CUT	IT
ACAP	Very Low	1.80	1.35	1.35	1.50
	Low	0.85	0.85	0.85	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	0.75	0.90	0.90	0.85
	Very High	0.55	0.75	0.75	0.70

Table 4.9 Analyst capability effort multiplier for Detailed COCOMO

Estimates made for each module are combined into subsystems and eventually an overall project estimate. Using the detailed cost drivers, an estimate is determined for each phase of the lifecycle.

Advantages of COCOMO'81

- COCOMO is transparent; you can see how it works unlike other models such as SLIM.
- Drivers are particularly helpful to the estimator to understand the impact of different factors that affect project costs.

Drawbacks of COCOMO'81

- It is hard to accurately estimate KDSI early on in the project, when most effort estimates are required.
- KDSI, actually, is not a size measure it is a length measure.
- Extremely vulnerable to mis-classification of the development mode
- Success depends largely on tuning the model to the needs of the organization, using historical data which is not always available

4.2.6 COCOMO II (Constructive Cost Model)

Researches on COCOMO II are started in late 90s because COCOMO'81 is note enough to apply to newer software development practices.

Differences between COCOMO’81 and COCOMO’II

COCOMO'II differs from COCOMO'81 with such differences:

- COCOMO'81 requires software size in KSLOC as an input, but COCOMO'II provides different effort estimating models based on the stage of development of the project.
- COCOMO'81 provides point estimates of effort and schedule, but COCOMO'II provides likely ranges of estimates that represent one standard deviation around the most likely estimate.
- COCOMO'II adjusts for software reuse and reengineering where automated tools are used for translation of existing software, but COCOMO'81 made little accommodation for these factors
- COCOMO'II accounts for requirements volatility in its estimates.

- The exponent on size in the effort equations in COCOMO'81 varies with the development mode. COCOMO'II uses five scale factors to generalize and replace the effects of the development mode.

COCOMO II has three different models:

- The Application Composition Model
- The Early Design Model
- The Post-Architecture Model

4.2.6.1 The Application Composition Model

The Application Composition model is used in prototyping to resolve potential high-risk issues such as user interfaces, software/system interaction, performance, or technology maturity. Object points are used for sizing rather than the traditional LOC metric.

An initial size measure is determined by counting the number of screens, reports, and third-generation components that will be used in the application. Each object is classified as simple, medium, or difficult using the guidelines shown in following Tables 4.10 and table 4.11.

	Number and source of data tables		
Number of views contained	Total <4	Total <8	Total 8+
<3	Simple	simple	medium
3-7	Simple	medium	difficult
8+	Medium	difficult	difficult

Table 4.10 Object point complexity levels for screens.

	Number and source of data tables		
Number of views contained	Total <4	Total <8	Total 8+
<3	Simple	simple	medium
3-7	Simple	medium	difficult
8+	Medium	difficult	difficult

Table 4.11 Object point complexity levels for reports.

The number in each cell is then weighted according to Table 4.12. The weights represent the relative effort required to implement an instance of that complexity level.

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

Table 4.12 Complexity weights for object points

The weighted instances are summed to provide a single object point number. Reuse is then taken into account. Assuming that r% of the objects will be reused from previous projects; the number of new object points (NOP) is calculated to be:

NOP = (object points) x (100 – r) / 100

A productivity rate (PROD) is determined using Table 4.13.

Developers' experience and capability	Very Low	Low	Nominal	High	Very High
ICASE maturity and capability	Very Low	Low	Nominal	High	Very High
PROD	4	7	13	25	50

Table 4.13 Average productivity rates based on developer's experience and the ICASE maturity/capability

Effort can then be estimated using the following equation:

E = NOP / PROD

4.2.6.2 The Early Design Model

The Early Design model is used to evaluate alternative software/system architectures and concepts of operation. An unadjusted function point count (UFC) is used for sizing. This value is converted to LOC using tables such as those published by Jones, excerpted in Table 4.14.

Language	**Level**	**Min**	**Mode**	**Max**

Machine language	0.10	-	640	-
Assembly	1.00	237	320	416
C	2.50	60	128	170
RPGII	5.50	40	58	85
C++	6.00	40	55	140
Visual C++	9.50	-	34	-
PowerBuilder	20.00	-	16	-
Excel	57.00	-	5.5	-

Table 4.14 Programming language levels and ranges of source code statements per function point

The Early Design model equation is:

E = aKLOC x EAF

where a is a constant, provisionally set to 2.45.

The effort adjustment factor (EAF) is calculated as in the original COCOMO model using the 7 cost drivers shown in Table 4.15. The Early Design cost drivers are obtained by combining the Post-Architecture cost drivers.

Cost Driver	Description	Counterpart Combined Post-Architecture Cost Driver
RCPX	Product reliability & complexity	RELY, DATA, CPLX, DOCU
RUSE	Required reuse	RUSE
PDIF	Platform difficulty	TIME, STOR, PVOL
PERS	Personnel capability	ACAP, PCAP, PCON
PREX	Personnel experience	AEXP, PEXP, LTEX
FCIL	Facilities	TOOL, SITE
SCED	Schedule	SCED

Table 4.15 Early Design cost drivers.

4.2.6.3 The Post-Architecture Model

The Post-Architecture model is used during the actual development and maintenance of a product. Function points or LOC can be used for sizing, with modifiers for reuse and software breakage. Boehm advocates the set of guidelines proposed by The Software Engineering Institute in counting lines of code. The Post-Architecture model includes a set of 17 cost drivers and a set of 5 factors determining the projects scaling component. The 5 factors replace the development modes (organic, semidetached, embedded) of the original COCOMO model.

The Post-Architecture model equation is:

E = aKLOC^b x EAF

Where a is set to 2.55 and b is calculated as:

b = 1.01 + 0.01 x SUM (Wi)

Where W is the set of 5 scale factors shown in Table 4.16:

W(i)	Very Low	Low	Nominal	High	Very High	Extra High
Precedentedness	4.05	3.24	2.42	1.62	0.81	0.00
Development/Flexibility	6.07	4.86	3.64	2.43	1.21	0.00
Architecture/Risk Resolution	4.22	3.38	2.53	1.69	0.84	0.00
Team Cohesion	4.94	3.95	2.97	1.98	0.99	0.00
Process Maturity	4.54	3.64	2.73	1.82	0.91	0.00

Table 4.16 COCOMO II scale factors

The EAF is calculated using the 17 cost drivers shown in Table 4.17.

Cost Driver	Description	Rating					
		Very Low	Low	Nominal	High	Very High	Extra High
Product							
RELY	Required S/W reliability	0.75	0.88	1.00	1.15	1.39	-
DATA	Database size	-	0.93	1.00	1.09	1.19	-
CPLX	Product complexity	0.70	0.88	1.00	1.15	1.30	1.66
RUSE	Required reusability		0.91	1.00	1.14	1.29	1.49
DOCU	Documentation		0.95	1.00	1.06	1.13	
Platform							
TIME	Execution time constraint	-	-	1.00	1.11	1.31	1.67
STOR	Main storage constraint	-	-	1.00	1.06	1.21	1.57
PVOL	Platform volatility	-	0.87	1.00	1.15	1.30	-
Personnel							
ACAP	Analyst capability	1.50	1.22	1.00	0.83	0.67	-
PCAP	Programmer capability	1.37	1.16	1.00	0.87	0.74	-
PCON	Personnel continuity	1.24	1.10	1.00	0.92	0.84	-
AEXP	Applications experience	1.22	1.10	1.00	0.89	0.81	-

PEXP	Platform experience	1.25	1.12	1.00	0.88	0.81	-
LTEX	Language & tool experience	1.22	1.10	1.00	0.91	0.84	
Project							
TOOL	Software Tools	1.24	1.12	1.00	0.86	0.72	-
SITE	Multisite development	1.25	1.10	1.00	0.92	0.84	0.78
SCED	Development Schedule	1.29	1.10	1.00	1.00	1.00	-

Table 4.17 Post-Architecture cost drivers.

4.2.7 The Norden-Rayleigh Curve

The Norden-Rayleigh curve represents manpower as a function of time. Norden observed that the Rayleigh distribution provides a good approximation of the manpower curve for various hardware development processes.

SLIM uses separate Rayleigh curves for design and code, test and validation, maintenance, and management. A Rayleigh curve is shown in following Figure.

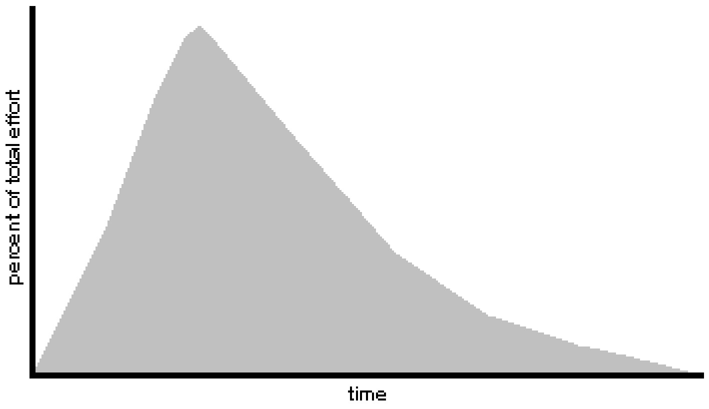


Figure 4.1 A Rayleigh curve.

Development effort is assumed to represent only 40 percent of the total life cycle cost. Requirements specification is not included in the model. Estimation using SLIM is not expected to take place until design and coding.

Several researchers have criticized the use of a Rayleigh curve as a basis for cost estimation. Norden's original observations were not based in theory but rather on observations. Moreover his data reflects hardware projects. It has not been demonstrated that software projects are staffed in the same way. Software projects sometimes exhibit a rapid manpower buildup which invalidate the SLIM model for the beginning of the project.

4.2.8 The Software Equation

Putnam used some empirical observations about productivity levels to derive the software equation from the basic Rayleigh curve formula. The software equation is expressed as:

$\text{Size} = CE^{1/3}(t^{4/3})$

Where C is a technology factor, E is the total project effort in person years, and t is the elapsed time to delivery in years.

The technology factor is a composite cost driver involving 14 components. It primarily reflects:

- Overall process maturity and management practices
- The extent to which good software engineering practices are used
- The level of programming languages used
- The state of the software environment
- The skills and experience of the software team
- The complexity of the application

The software equation includes a fourth power and therefore has strong implications for resource allocation on large projects. Relatively small extensions in delivery date can result in substantial reductions in effort.

4.2.9 The Manpower-Buildup Equation

To allow effort estimation, Putnam introduced the manpower-buildup equation:

$D = E / t^3$

Where D is constant called manpower acceleration, E is the total project effort in years, and t is the elapsed time to delivery in years.

The manpower acceleration is 12.3 for new software with many interfaces and interactions with other systems, 15 for standalone systems, and 27 for reimplementations of existing systems.

Using the software and manpower-buildup equations, we can solve for effort:

$E = (S / C)^{9/7} (D^{4/7})$

This equation is interesting because it shows that effort is proportional to size to the power 9/7 or ~1.286, which is similar to Boehm's factor which ranges from 1.05 to 1.20.

4.2.10 Criteria for Evaluating a Model

Boehm provides the following criteria for evaluating cost models:

- Definition – Has the model clearly defined the costs it is estimating, and the costs it is excluding?
- Fidelity – Are the estimates close to the actual costs expended on the projects?
- Objectivity – Does the model avoid allocating most of the software cost variance to poorly calibrated subjective factors (such as complexity)? Is it hard to adjust the model to obtain any result you want?
- Constructiveness – Can a user tell why the model gives the estimates it does? Does it help the user understand the software job to be done?

- Detail – Does the model easily accommodate the estimation of a software system consisting of a number of subsystems and units? Does it give (accurate) phase and activity breakdowns?
- Stability – Do small differences in inputs produce small differences in output cost estimates?
- Scope – Does the model cover the class of software projects whose costs you need to estimate?
- Ease of Use – Are the model inputs and options easy to understand and specify?
- Prospectiveness – Does the model avoid the use of information that will not be well known until the project is complete?
- Parsimony – Does the model avoid the use of highly redundant factors, or factors which make no appreciable contribution to the results?

4.2.11 Problems with Existing Models

There is some question as to the validity of existing algorithmic models applied to a wide range of projects. It is suggested that a model is acceptable if 75 percent of the predicted values fall within 25 percent of their actual values. Unfortunately most models are insufficient based on this criterion. Kemerer reports average errors (in terms of the difference between predicted and actual project effort) of over 600 percent in his independent study of COCOMO. The reasons why existing modeling methods have fallen short of their goals include model structure, complexity, and size estimation.

4.2.11.1 Structure

Although most researchers and practitioners agree that size is the primary determinant of effort, the exact relationship between size and effort is unclear. Most empirical studies express effort as a function of size with an exponent b and a multiplicative term a. However the values of a and b vary from data set to data set.

Most models suggest that effort is proportional to size, and b is included as an adjustment factor so that larger projects require more effort than smaller ones. Intuitively this makes sense, as larger projects would seem to require more effort to deal with increasing complexity. However in practice, there is little evidence to support this. Banker and Kemerer analyzed seven data sets, finding only one that was significantly different from 1 (with a level of significance of p=0.05). Table 4.18 compares the adjustment factors of several different models.

Model	Adjustment Factor
Walston-Felix	0.91
Nelson	0.98
Freburger-Basili	1.02
COCOMO (organic)	1.05
Herd	1.06
COCOMO (semi-detached)	1.12
Bailey-Basili	1.16
Frederic	1.18
COCOMO (embedded)	1.20
Phister	1.275

Putnam	1.286
Jones	1.40
Halstead	1.50
Schneider	1.83

Table 4.18 Comparison of effort equation adjustment factors

There is also little consensus about the effect of reducing or extending duration. Boehm's schedule cost driver assumes that increasing or decreasing duration increases project effort. Putnam's model implies that decreasing duration increases effort, but increasing duration decreases effort (Fenton, 1997). Other studies have shown that decreasing duration decreases effort, contradicting both models.

Most models work well in the environments for which they were derived, but perform poorly when applied more generally. The original COCOMO is based on a data set of 63 projects. COCOMO II is based on a data set of 83 projects. Models based on limited data sets tend to incorporate the particular characteristics of the data. This results in a high degree of accuracy for similar projects, but restricts the application of the model.

4.2.11.2 Complexity

An organization's particular characteristics can influence its productivity. Many models include adjustment factors, such as COCOMO's cost drivers and SLIM's technology factor to account for these differences. The estimator relies on adjustment factors to account for any variations between the model's data set and the current estimate. However this generalized approach is often inadequate.

Kemerer has suggested that application of the COCOMO cost drivers does not always improve the accuracy of estimates. The COCOMO model assumes that the cost drivers are independent, but this is not the case in practice. Many of the cost drivers affect each other, resulting in the over emphasis of certain attributes. The cost drivers are also extremely subjective. It is difficult to ensure that the factors are assessed consistently and in the way the model developer intended.

Calculation of adjustment factor is also often complicated. The SLIM model is extremely sensitive to the technology factor, however this is not an easy value to determine. Calculation of the EAF for the detailed COCOMO model can also be somewhat complex, as it is distributed between phases of the software lifecycle.

4.2.11.3 Size Estimation

Most models require an estimate of product size. However size is difficult to predict early in the development lifecycle. Many models use LOC for sizing, which is not measurable during requirements analysis or project planning. Although function points and object points can be used earlier in the lifecycle, these measures are extremely subjective.

Size estimates can also be very inaccurate. Methods of estimation and data collection must be consistent to ensure an accurate prediction of product size. Unless the size metrics used in the model are the same as those used in practice, the model will not yield accurate results.

4.3 Summary

Software cost estimation is an important part of the software development process. Models can be used to represent the relationship between effort and a primary cost factor such as size. Cost drivers are used to adjust the preliminary estimate provided by the primary cost factor. Although models are widely used to predict software cost, many suffer from some common problems. The structure of most models is based on empirical results rather than theory. Models are often complex and rely heavily on size estimation. Despite these problems, models are still important to the software development process. Model can be used most effectively to supplement and corroborate other methods of estimation.

4.4 Keywords

Software cost estimation: is the process of predicting the amount of effort required to build a software system.

COCOMO: It is a model to give an estimate of the number of man-months to develop a software product.

Basic COCOMO: It is model that computes software development effort as a function of program size expressed in estimated lines of code.

Intermediate COCOMO: It computes software development effort as function of program size and a set of "cost drivers".

Detailed COCOMO: It incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step of the software engineering process.

4.5 Self Assessment Questions

1. What are the factors affecting the cost of the software?
2. Differentiate between basic, intermediate, and advanced COCOMO.
3. What are the differences between COCOMOII and COCOMO 81? Explain.
4. What do you understand by cost models and constraint models?

3.6 References/Suggested readings

13. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
14. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
15. Software Engineering by Sommerville, Pearson Education.
16. Software Engineering – A Practitioner's Approach by Roger S Pressman, McGraw-Hill.

Lesson No.: 5 **Writer: Dr. Rakesh Kumar**

Software Requirement Analysis & Specification Vetter: Dr Yogesh Chaba

5.0 Objectives

The objectives of this lesson are to get the students familiar with software requirements. After studying this lesson the students will be able:

- To understand the concepts of requirements.
- To differentiate between different types of requirements.
- To know the structure of software requirement specification.
- Characteristics of SRS.

5.1 Introduction

The analysis phase of software development is concerned with project planning and software requirement definition. To identify the requirements of the user is a tedious job. The description of the services and constraints are the requirements for the system and the process of finding out, analyzing, documenting, and checking these services is called requirement engineering. The goal of requirement definition is to completely and consistently specify the requirements for the software product in a concise and unambiguous manner, using formal notations as appropriate. The software requirement specification is based on the system definition. The requirement specification will state the “what of” the software product without implying “how”. Software design is concerned with specifying how the product will provide the required features.

5.2 Presentation of contents

5.2.1 Software system requirements

Software system requirements are classified as functional requirements and non-functional requirements.

5.2.1.1 Functional requirements

The functional requirements for a system describe the functionalities or services that the system is expected to provide. They provide how the system should react to particular inputs and how the system should behave in a particular situation.

5.2.1.2 Non-functional requirements

These are constraints on the services or functionalities offered by the system. They include timing constraints, constraints on the development process, standards etc. These requirements are not directly concerned with the specific function delivered by the system. They may relate to such system properties such as reliability, response time, and storage. They may define the constraints on the system such as capabilities of I/O devices and the data representations used in system interfaces.

The objective of this phase is to identify the requirements of the clients and generate a formal requirement document.

5.2.2 Software requirement specification

It is the official document of what is required of the system developers. It consists of user requirements and detailed specification of the system requirements. According to Henninger there are six requirements that an SRS should satisfy:

1. It should specify only external system behavior.
2. It should specify constraints on the implementation.
3. It should be easy to change.

4. It should serve as a reference tool for system maintainers.
5. It should record forethought about the life cycle of the system.
6. It should characterize acceptable response to undesired events.

The IEEE standard suggests the following structure for SRS:

1. Introduction

1.1 Purpose of the requirement document.

1.2 Scope of the product

1.3 Definitions, acronyms, and abbreviations

1.4 References

1.5 Overview of the remainder of the document

2. General description

2.1 Product perspective

2.2 Product functions

2.3 User characteristics

2.4 General constraints

2.5 Assumption and dependencies

3. Specific requirements covering functional, non-functional and interface requirements.
4. Appendices
5. Index

5.2.2.1 Characteristics of SRS

The desirable characteristics of an SRS are following:

- **Correct:** An SRS is correct if every requirement included in the SRS represents something required in the final system.
- **Complete:** An SRS is complete if everything software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.
- **Unambiguous:** An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.
- **Verifiable:** An SRS is verifiable if and only if every specified requirement is verifiable i.e. there exists a procedure to check that final software meets the requirement.
- **Consistent:** An SRS is consistent if there is no requirement that conflicts with another.
- **Traceable:** An SRS is traceable if each requirement in it must be uniquely identified to a source.
- **Modifiable:** An SRS is modifiable if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency.
- **Ranked:** An SRS is ranked for importance and/or stability if for each requirement the importance and the stability of the requirements are indicated.

5.2.2.2 Components of an SRS

An SRS should have the following components:

(i) Functionality

(ii) Performance

(iii) Design constraints

(iv) External Interfaces

Functionality

Here functional requirements are to be specified. It should specify which outputs should be produced from the given input. For each functional requirement, a detailed description of all the inputs, their sources, range of valid inputs, the units of measure are to be specified. All the operation to be performed on input should also be specified.

Performance requirements

In this component of SRS all the performance constraints on the system should be specified such as response time, throughput constraints, number of terminals to be supported, number of simultaneous users to be supported etc.

Design constraints

Here design constraints such as standard compliance, hardware limitations, Reliability, and security should be specified. There may be a requirement that system will have to use some existing hardware, limited primary and/or secondary memory. So it is a constraint on the designer. There may be some standards of the organization that should be obeyed such as the format of reports. Security requirements may be particularly significant in defense systems. It imposes a restriction sometimes on the use of some commands, control access to data, require the use of passwords and cryptography techniques etc.

External Interface requirements

Software has to interact with people, hardware, and other software. All these interfaces should be specified. User interface has become a very important issue now a day. So the characteristics of user interface should be precisely specified and should be verifiable.

5.2.3 Problem Analysis

The objective of problem analysis is to obtain the clear understanding of the user's requirements. There are the different approaches to problem analysis which have been discussed in the following section.

Data flow modeling

It is also known as structured analysis. Here the aim is to identify the functions performed in the problem and the data consumed and produced by these function.

What is a model?

Douglas T. Ross that a model answers questions; that the definition of a model is: M models A if M answers questions about A

Why model?

We need to model complex systems in the real world in order to understand them. For example: we create computerized models of the real world to manipulate large amounts of data and hence derive information which can assist in decision making.

An analyst will create diagrammatic models of a target or proposed system in order to:

- Understand the system, and
- Communicate:
    - to demonstrate, or clarify, understanding of the existing system and/or to obtain feedback from users/clients;
    - to describe unambiguously the proposed computer system to users/clients and to the programming team.

Modeling techniques are extremely useful in tackling the complexity which is found when attempting to analyze and understand a system. Models are also extremely useful communication tools; i.e.: complex ideas and concepts can be captured on paper and can be shown to users and clients for clarification and feedback; or for distribution to other professionals, team members, contractors etc. In this respect, the final models created in the Design and Development phases of a system are essentially paper based prototypes.

5.2.3.1 Modeling Techniques

The three most important modeling techniques used in analyzing and building information systems are:

- Data Flow Diagramming (DFDs): Data Flow Diagrams (DFDs) model events and processes (i.e. activities which transform data) within a system. DFDs examine how data flows into, out of, and within the system. (Note: 'data' can be understood as any 'thing' (eg: raw materials, filed information, ideas, etc.) which is processed within the system as shown in Figure 5.1.

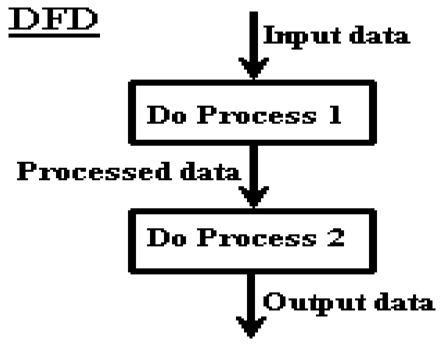


Figure 5.1

- Logical Data Structure modelling (LDSs): Logical Data Structures (LDSs) represent a system's information and data in another way. LDSs map the underlying data structures as entity types, entity attributes, and the relationships between the entities as shown in figure 5.2.

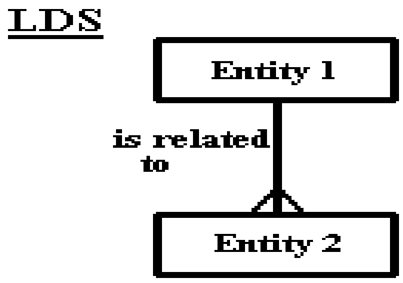


Figure 5.2

- Entity Life Histories (ELHs): Entity Life Histories (ELHs) describe the changes which happen to 'things' (entities) within the system as shown in figure 5.3.

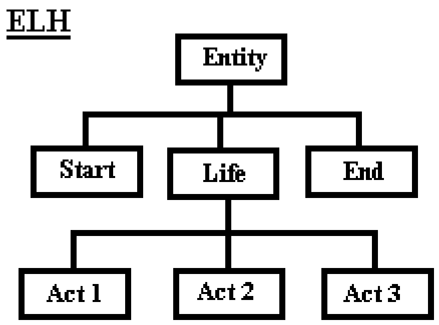


Figure 5.3

These three techniques are common to many methodologies and are widely used in system analysis. Notation and graphics style may vary across methodologies, but the underlying principles are generally the same.

In SSADM (Structured Systems Analysis and Design Methodology - which has for a number of years been widely used in the UK) systems analysts and modelers use the above techniques to build up three, inter-related, views of the target system, which are cross-checked for consistency.

5.2.3.2 Data Flow Diagrams (DFDs)

SSADM uses different sets of Data Flow Diagram to describe the target system in different ways; eg:

- WHAT the system does
- HOW it does it
- WHAT it should do
- HOW it should do it

Another way of looking at it is that, in SSADM, DFDs are used to answer the following data-oriented questions about a target system:

What processing is done? When? How? Where? By whom?

What data is needed? By whom? for what? When?

5.2.3.2.1 Data Flow Diagrams show

- The processes within the system.
- The data stores (files) supporting the system's operation.
- The information flows within the system.

- The system boundary.
- Interactions with external entities.

However, we are not interested, here, in the development process in detail, only in the general modeling technique. Essentially, DFDs describe the information flows within a system.

5.2.3.2.2 DFD Principles

- The general principle in Data Flow Diagramming is that a system can be decomposed into subsystems, and subsystems can be decomposed into lower level subsystems, and so on.
- Each subsystem represents a process or activity in which data is processed. At the lowest level, processes can no longer be decomposed.
- Each 'process' in a DFD has the characteristics of a system.
- Just as a system must have input and output (if it is not dead), so a process must have input and output.
- Data enters the system from the environment; data flows between processes within the system; and data is produced as output from the system

5.2.3.2.3 Basic DFD Notations

In a DFD, a process may be shown as a circle, an oval, or (typically) a rectangular box; and data are shown as arrows coming to, or going from the edge of the process box.

(SSADM) DFD Notations

SSADM uses 4 diagramming notations in DFDs as shown in figure 5.4:

- Processes transform or manipulate data. Each box has a unique number as identifier (top left) and a unique name (an imperative - eg: 'do this' - statement in the main box area) The top line is used for the location of, or the people responsible for, the process.
- Data Flows depict data/information flowing to or from a process. The arrows used to represent the flows must either start and/or end at a process box.
- Data Stores are some location where data is held temporarily or permanently.
- External Entities, also known as 'External source/recipients, are things (e.g.: people, machines, organizations etc.) which contribute data or information to the system or which receive data/information from it.

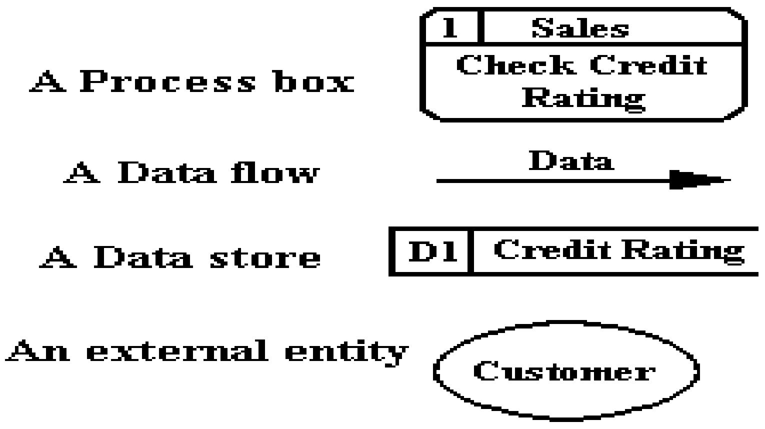


Figure 5.4

5.2.3.2.4 General Data Flow Rules

- Entities are either 'sources of' or 'sinks' for data input and outputs - i.e. they are the originators or terminators for data flows.
- Data flows from Entities must flow into Processes
- Data flows to Entities must come from Processes

- Processes and Data Stores must have both inputs and outputs (What goes in must come out!)
- Inputs to Data Stores only come from Processes.
- Outputs from Data Stores only go to Processes.

5.2.3.2.5 DFD Levels

The 'Context Diagram ' is an overall, simplified, view of the target system, which contains only one process box, and the primary inputs and outputs as shown in figure 5.5.

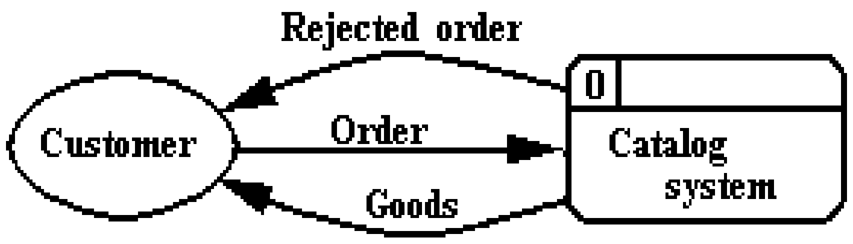


Figure 5.5

Context diagram 2

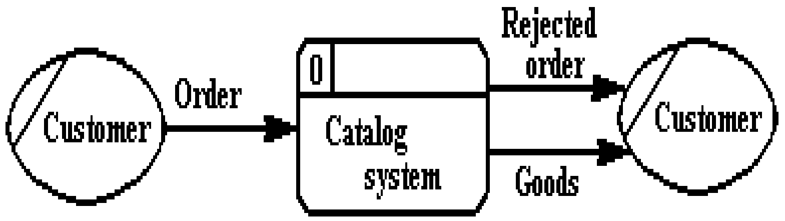


Figure 5.6

Both the above figure 5.5 and 5.6 say the same thing. The second makes use of the possibility in SSADM of including duplicate objects. (In context diagram 2 the duplication of the Customer object is shown by the line at the left hand side.

Drawing the diagram in this way emphasizes the Input-Output properties of a system.)

The Context diagram above, and the one which follows (Figure 5.7), are a first attempt at describing part of a 'Home Catalogue' sales system. In the modeling process it is likely that diagrams will be reworked and amended many times - until all parties are satisfied with the resulting model. A model can usefully be described as a co-ordinated set of diagrams.

The Top (1st level) DFD

The Top or 1st level DFD, describes the whole of the target system. The Top level DFD 'bounds' the system -and shows the major processes which are included within the system.

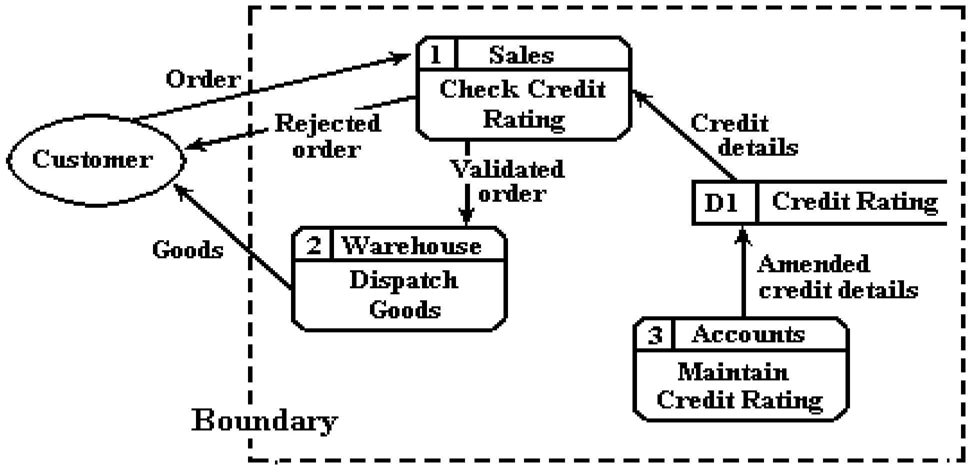


Figure 5.7

The next step - the Next Level(s)

Each Process box in the Top Level diagram may itself be made up of a number of processes, and where this is the case, the process box will be decomposed as a second level diagram as shown in figure 5.8.

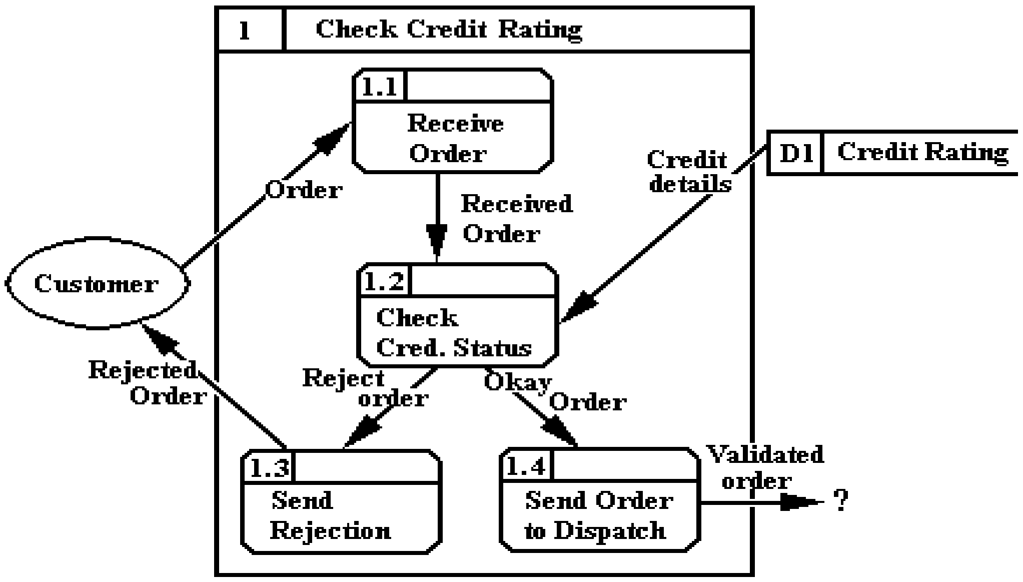


Figure 5.8

Each box in a diagram has an identification number derived from the parent. Any box in the second level decomposition may be decomposed to a third level. Very complex systems may possibly require decomposition of some boxes to further levels.

Decomposition stops when a process box can be described with an Elementary Function Description using, for example, Pseudocode as shown in figure 5.5.

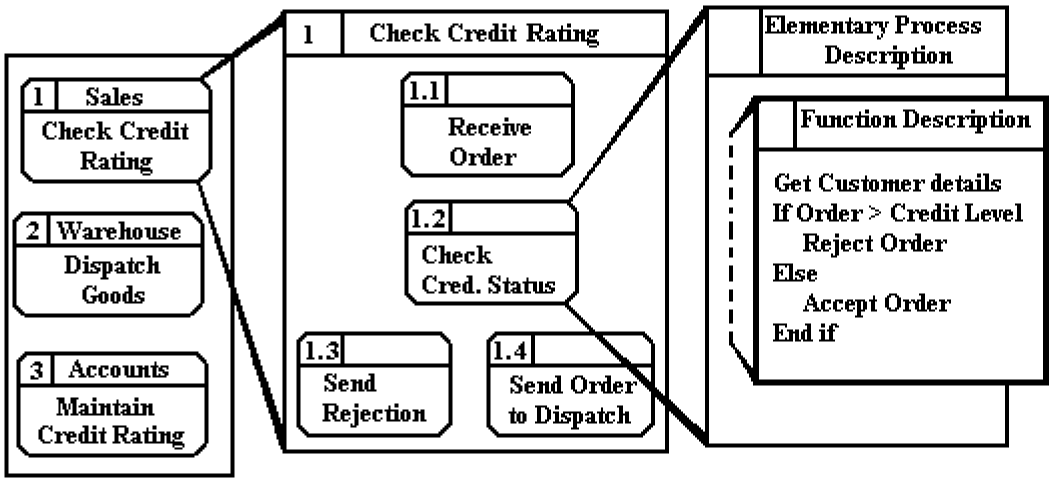


Figure 5.9

Each box in a diagram has an identification number (derived from the parent - the Context level is seen as box 0) in the top left corner.

Every page in a DFD should contain fewer than 10 components. If a process has more than 10 components, then one or more components (typically a process) should be combined into one and another DFD be generated that describes that component in more detail. Each component should be numbered, as should each subcomponent, and so on. So for example, a top level DFD would have components 1, 2, 3, 4, 5, the subcomponent DFD of component 3 would have components 3.1, 3.2, 3.3, and 3.4; and the subsubcomponent DFD of component 3.2 would have components 3.2.1, 3.2.2, and 3.2.3

SSADM uses different sets of Data Flow Diagram to describe the target system in different ways, moving from analysis of the current system to specification of the required system:

WHAT the system does	- Current Physical DFD
HOW it does it	- Current Logical DFD
WHAT it should do	- Required Logical DFD
HOW it should do it	- Required Physical DFD

Table 5.1

5.2.3.2.6 Developing a DFD

In the following section, two approaches to prepare the DFD are proposed.

Top-Down Approach

1. The system designer makes a context level DFD, which shows the interaction (data flows) between the system (represented by one process) and the system environment (represented by terminators).
2. The system is decomposed in lower level DFD into a set of processes, data stores, and the data flows between these processes and data stores.
3. Each process is then decomposed into an even lower level diagram containing its subprocesses.
4. This approach then continues on the subsequent subprocesses, until a necessary and sufficient level of detail is reached.

Event Partitioning Approach

This approach was described by Edward Yourdon in "Just Enough Structured Analysis",

1. Construct detail DFD.
    1. The list of all events is made.
    2. For each event a process is constructed.
    3. Each process is linked (with incoming data flows) directly with other processes or via data stores, so that it has enough information to respond to given event.
    4. The reaction of each process to a given event is modeled by an outgoing data flow.

5.2.3.2 Structured Analysis and Design Techniques (SADT)

SADT was developed by D.T. Ross. It incorporates a graphical language. An SADT model consists of an ordered set of SA (Structured Analysis) diagrams. Each diagram must contain 3 to 6 nodes plus interconnecting arcs. Two basic types of SA diagram are Actigram (activity diagram) and datagram (data diagram). In actigram the nodes denote activities and arcs specify the data flow between activities while in datagrams nodes specify the data objects and arcs denote activities. The following figure shows the formats of actigram and datagram. It is important to note that there are four distinct types of arcs. Arcs coming into the left side of a node show inputs and arcs leaving the right side of a node convey output. Arcs entering the top of a node convey control and arcs entering the bottom specify mechanism. Following Figures 5.10 and 5.11 illustrate the activity diagrams and data diagrams. As shown, in actigram arc coming into the left side shows the input data on that activity works, arc coming from right indicates the data produced by the activity. Arc entering the top of the node specifies the control data for the activity arc entering the bottom specify the processor.

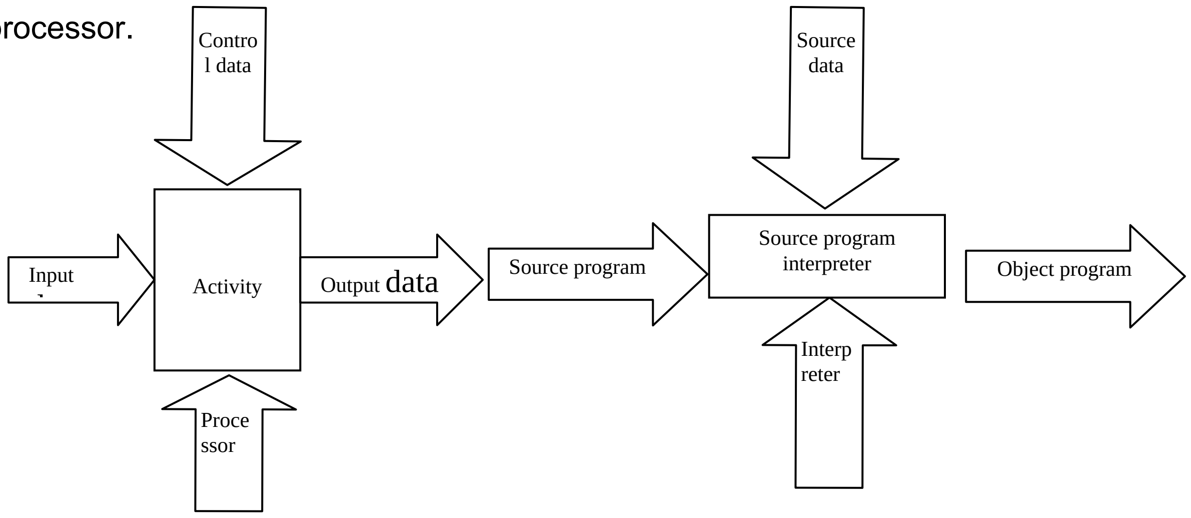


Figure 5.10 Activity diagram

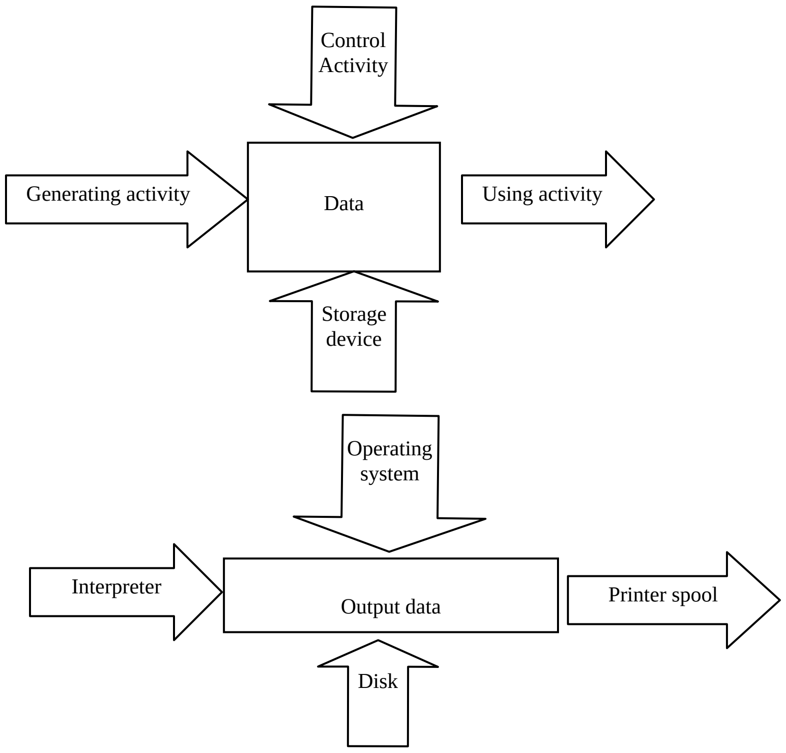


Figure 5.11 Data diagram components

Following SADT diagram (Figure 5.12) illustrate a simple payroll system. It is making the use of activity diagrams.

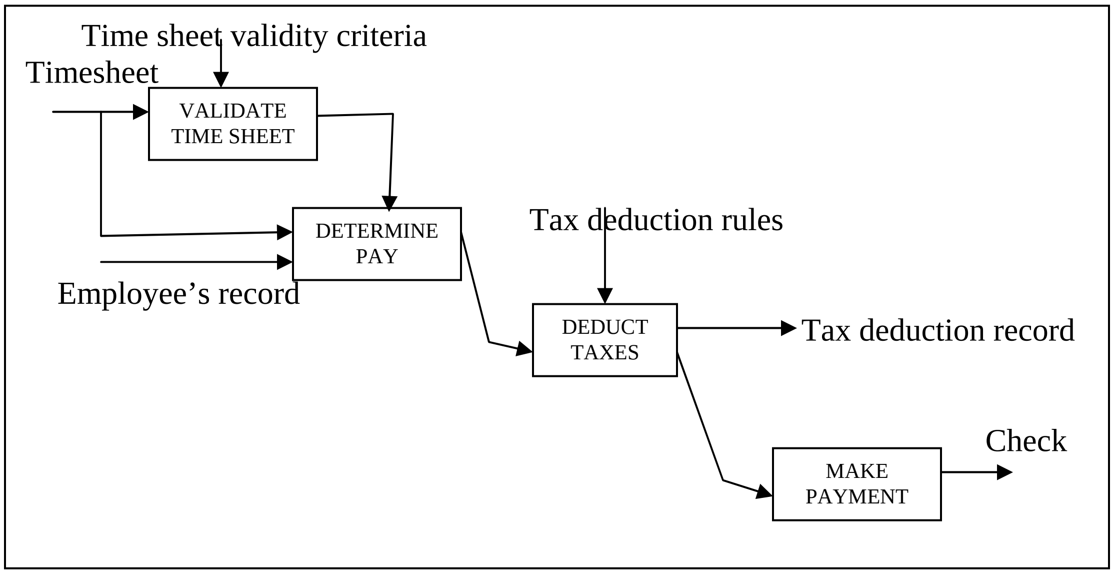


Figure 5.12

5.2.3.3 Prototyping

Sometimes when the system is a totally new system, the users and clients do not have a good idea of their requirements. In this type of cases, prototyping can be a good option. The idea behind prototyping is that clients and users can assess their needs much better if they can see the working of a system, even if the system is a partial system. So an actual experience with a prototype that implements part of the whole system helps the user in understanding their requirements. So in prototyping approach, first of all a prototype is built and then delivered to the user to use it.

There are two variants of prototyping: (i) Throwaway prototyping and (ii) evolutionary prototyping. Throwaway prototyping is used with the objective that prototype will be discarded after the requirements have been identified. In evolutionary prototyping, the idea is that prototype will be eventually converted in the final system. Gradually the increments are made to the prototype by taking into the consideration the feedback of clients and users.

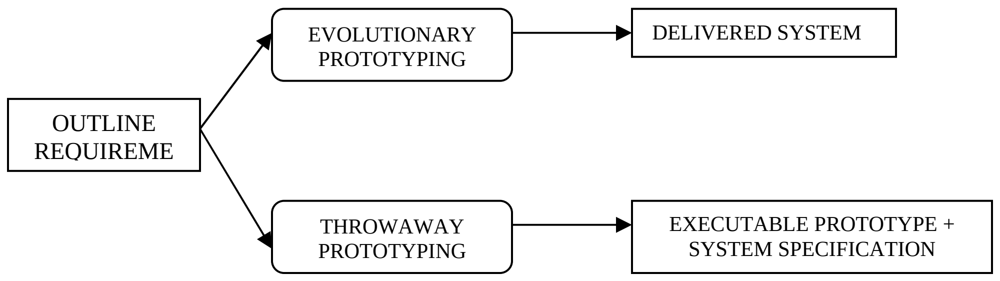


Figure 5.13

Evolutionary prototyping

It is the only way to develop the system where it is difficult to establish a detailed system specification. But this approach has following limitations:

(i) Prototype evolves so quickly that it is not cost effective to produce system documentation.

(ii) Continual changes tend to corrupt the structure of the prototype system. So maintenance is likely to be difficult and costly.

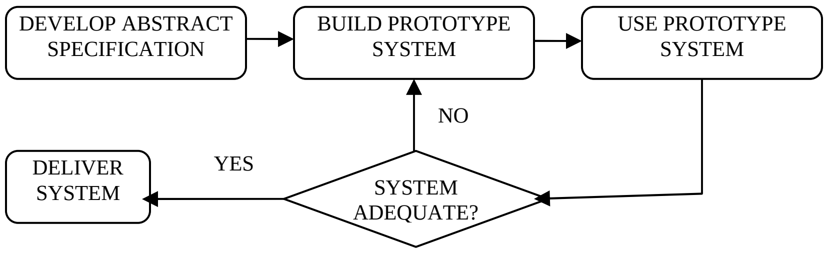


Figure 5.14

Throwaway prototyping

The principal function of the prototype is to clarify the requirements. After evaluation the prototype is thrown away as shown in figure 5.15.

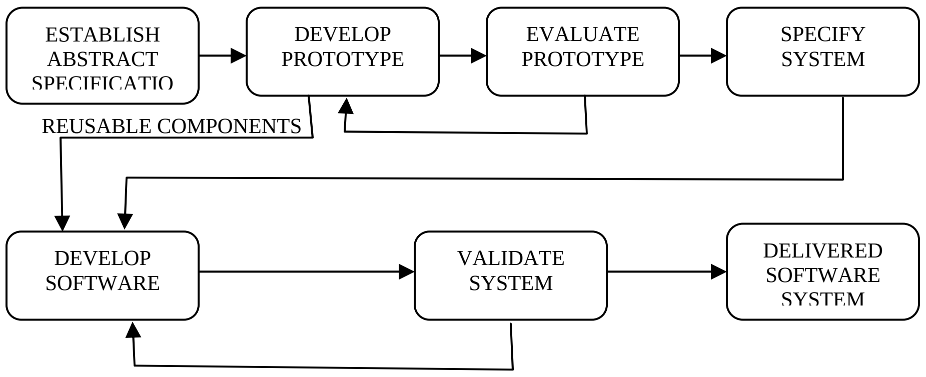


Figure 5.15

Customers and end users should resist the temptation to turn the throwaway prototype into a delivered system. The reason for this are:

(i) Important system characteristics such as performance, security, reliability may have been ignored during prototype development so that a rapid implementation could be developed. It may be impossible to turn the prototype to meet these non-functional requirements.

(ii) The changes made during prototype development will probably have degraded the system structure. So the maintenance will be difficult and expensive.

5.2.4 Specification languages

Requirement specification necessitates the use of some specification language. The language should possess the desired qualities like modifiability, understandability, unambiguous etc. But the language should be easy to use. This requirement is sometimes difficult to meet. For example to avoid the unambiguity, one should use formal language, which is difficult to use and learn. Natural language is quite easy to use but tends to be ambiguous. Some of the commonly used specification languages are discussed below

5.2.4.1 Structured English

The natural languages are most easy to use but it has some drawbacks and the most important one is that requirements specified in natural language are imprecise and ambiguous. To remove this drawback some efforts have been made and one is the use of structured English. In structure English the requirements are broken into sections and paragraphs. Each paragraph is broken into sub paragraphs. Many organizations specify the strict use of some words and restrict the use of others to improve the precision.

5.2.4.2 Regular expressions

Regular expressions are used to specify the syntactic structure of symbol strings. Many software products involve processing of symbol strings. Regular expressions provide a powerful notation for such cases. The rules for regular expressions are:

1. Atoms: A basic symbol in the alphabet of interest.
2. Alternations: If R1 and R2 are regular expressions then (R1|R2) is a regular expression. (R1|R2) denotes the union of the languages specified by R1 and R2.
3. Composition: If R1 and R2 are regular expressions then (R1 R2) is a regular expression. (R1 R2) denotes the language formed by concatenating strings from R2 onto strings from R1.
4. Closure: If R1 is a regular expression then (R1)* is a regular expression. (R1)* denotes the language formed by concatenating zero or more strings from R1 with zero or more strings from R1. A commonly used notation is (R1)+, which denotes one or more concatenation of elements in R1.
5. Completeness: Nothing else is a regular expression.

For example the requirement, a valid data stream must start with an “a”, followed by “b”s and “c”s in any order but always interleaved by a and terminated by “b” or “c”, may be represented by following regular expression: ((a (b|c)))+.

5.2.4.3 Decision tables

It is a mechanism for recording complex decision logics. A decision table consists of four quadrants: condition stub, condition entries, action stub, and action entries. In the condition stub, all the conditions are specified. Condition entries are used to combine conditions into decision rules. Action stub specifies the actions to be taken in response to decision rules. The action entry relates decision rules to actions as shown in table 5.2.

	Decision Rules			
	Rule 1	Rule 2	Rule 3	Rule 4
Condition stub		Condition entries		
Action Stub		Action entries		

Table 5.2 Decision table

Following is the decision table (Table 5.3) to find the largest of three numbers.

	Rule 1	Rule 2	Rule 3
A>B	Y	Y	N
A>C	Y	N	N
B>C	-	N	Y
A is largest	X		
B is largest			X
C is largest		X	

Table 5.3

The above decision table is an example of a limited entry decision table in which the entries are limited to Y, N, -, X where Y denotes yes, N denotes No, - denotes don’t care, and X denotes perform action. If more than one decision rule has identical (Y, N, -) entries, the table is said to be ambiguous. Ambiguous pair of decision rules that specify identical actions are said to be redundant and those specifying different actions are contradictory. The contradictory rules permits specification of nondeterministic and concurrent actions.

	Rule 1	Rule 2	Rule 3	Rule 4
C1	Y	Y	Y	Y
C2	Y	N	N	N
C3	N	N	N	N
A1	X			
A2		X		
A3			X	X

Table 5.4

The above decision table (Table 5.4) illustrates redundant rules (R3 and R4) and contradictory rules (R2 and R3, R2 and R4).

A decision table is complete if every possible set of conditions has a corresponding action prescribed. There are 2^n combinations of conditions in a decision table that has n conditions.

5.2.4.4 Event tables

They specify actions to be taken when events occur under different sets of conditions. A 2-dimensional event table relates action to operating conditions and event of interest. For example, following table 5.5 specifies that if condition c1 is there and event E1 occurs, then one must take A1 action. A "-" entry indicates that no action is required. "X" indicates impossible system configuration. Two actions separated by “,” (A2, A3) denotes concurrent activation while separated by “;” (A4; A5) denotes A5 follows A4.

Conditions	Event				
	E1	E2	E3	E4	E5
C1	A1	-	A4; A5		
C2	X	A2, A3			
C3					
C4					

Table 5.5

5.2.4.5 Transition table

These are used to specify changes in the state of a system as a function of driving forces. Following figure shows the format of a transition table (Table 5.6).

Current state	Current input	
	A	B
S0	S0	S1
S1	S1	S0

Table 5.6 Transition Table

It indicates that if B is the input in S0 state then a transition will take place to s1 state. Transition tables are the representation of finite state automata.

Summary

Requirements state what the system should do and define constraints on its operations and implementation. The requirements may be classified as functional and non-functional requirements providing the information about the functionalities and constraints on the system. The requirements are to be specified in an unambiguous manner. So a number of tools and techniques are used such as DFD, decision table, event table, transition table, regular expressions etc. The SRS is the agreed statement of the system requirements. It should be organized so that both clients and developers can use it. To satisfy its goal, the SRS should have certain desirable characteristics such as consistency, verifiability, modifiability, traceability etc. To ensure the characteristic completeness, the SRS should consist of the components: Functionality, Performance, Design constraints, and External interfaces. There are different approaches to problem analysis. The aim of problem analysis is to have the clear understanding of the requirements of the user. The approaches discussed are data flow modeling using DFD, Structured analysis and Design Technique (SADT) and prototyping. Data flow modeling and SADT focus mainly on the functions performed in the problem domain and the input consumed and produced by these functions.

Keywords

Functional requirements: These are the functionalities or services that the system is expected to provide.

Non-functional requirements: These are constraints on the services or functionalities offered by the system.

SRS: It is the official document of what is required of the system developers.

DFD: They model events and processes and examine how data flows into, out of, and within the system.

Transition table: These are used to specify changes in the state of a system as a function of driving forces.

Event tables: They specify actions to be taken when events occur under different sets of conditions.

Decision Table: It is a mechanism for recording complex decision logics consisting of four quadrants: condition stub, condition entries, action stub, and action entries.

Self-assessment questions

1. What do you understand by requirements? Differentiate between functional and non-functional requirements using suitable examples.
2. What do you understand by SRS (Software Requirement Specification)? What are the components of it?
3. What are the desirable characteristics of Software Requirement Specification? Explain.
4. Why do we need the specification languages to generate the software requirement specification? What are the merits and demerits of using the specification languages? Explain.

5. Explain the graphic and text notations used in Data Flow Diagrams (DFDs).
6. What are the principles of Data Flow Diagram (DFD) modeling?
7. Why is DFD modeling useful?

1.6 References/Suggested readings

17. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
18. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
19. Software Engineering by Sommerville, Pearson Education.
20. Software Engineering – A Practitioner’s Approach by Roger S Pressman, McGraw-Hill.

Lesson number: 6	**Writer: Dr. Rakesh Kumar**
Software Design - I	**Vetter: Dr. PK Bhatia**

6.0 Objectives

The objective of this lesson is to make the students familiar with the concepts of design, design notations and design concepts. After studying the lesson students will be acquainted with:

1. Design and its quality.
2. Design fundamental concepts
3. Modularization criteria
4. Design notations

6.1 Introduction

Design is an iterative process of transforming the requirements specification into a design specification. Consider an example where Mrs. & Mr. XYZ want a new house. Their requirements include,

- a room for two children to play and sleep
- a room for Mrs. & Mr. XYZ to sleep
- a room for cooking
- a room for dining
- a room for general activities

and so on. An architect takes these requirements and designs a house. The architectural design specifies a particular solution. In fact, the architect may produce several designs to meet this requirement. For example, one may maximize children’s room, and other minimizes it to have large living room. In addition, the style of the proposed houses may differ: traditional, modern and two-storied. All of the proposed designs solve the problem, and there may not be a “best” design.

Software design can be viewed in the same way. We use requirements specification to define the problem and transform this to a solution that satisfies all the requirements in the specification. Design is the first step in the development phase for any engineered product. The designer goal is to produce a model of an entity that will later be built.

6.2.1 Definitions for Design

- "Devising artifacts to attain goals" [H.A. Simon, 1981].
- "The process of defining the architecture, component, interfaces and other characteristics of a system or component" [IEEE 160.12].
- The process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization.

Without Design, System will be

- Unmanageable since there is no concrete output until coding. Therefore it is difficult to monitor & control.
- Inflexible since planning for long term changes was not given due emphasis.
- Unmaintainable since standards & guidelines for design & construction are not used. No reusability consideration. Poor design may result in tightly coupled modules with low cohesion. Data disintegrity may also result.
- Inefficient due to possible data redundancy and untuned code.
- Not portable to various hardware / software platforms.

Design is different from programming. Design brings out a representation for the program – not the program or any component of it. The difference is tabulated below.

Design	Programming
Abstractions of operations & data("What to do")	Device algorithms and data representations
Establishes interfaces	Consider run-time environments
Choose between design alternatives Make trade-offs w.r.t.constraints etc	Choose functions, syntax of language
Devices representation of program	Construction of program

6.2.2 Qualities of a Good Design

Functional: It is a very basic quality attribute. Any design solution should work, and should be constructable.

Efficiency: This can be measured through

- run time (time taken to undertake whole of processing task or transaction)
- response time (time taken to respond to a request for information)
- throughput (no. of transactions / unit time)
- memory usage, size of executable, size of source, etc

Flexibility: It is another basic and important attribute. The very purpose of doing design activities is to build systems that are modifiable in the event of any changes in the requirements.

Portability & Security: These are to be addressed during design - so that such needs are not “hard-coded” later.

Reliability: It tells the goodness of the design - how it work successfully (More important for real-time and mission critical and on-line systems).

Economy: This can be achieved by identifying re-usable components.

Usability: Usability is in terms of how the interfaces are designed (clarity, aesthetics, directness, forgiveness, user control, ergonomics, etc) and how much time it takes to master the system.

6.2.3 Design Constraints

Typical Design Constraints are:

- Budget
- Time
- Integration with other systems
- Skills
- Standards
- Hardware and software platforms

Budget and Time cannot be changed. The problems with respect to integrating to other systems (typically client may ask to use a proprietary database that he is using) has to be studied & solution(s) are to be found. ‘Skills’ is alterable (for example, by arranging appropriate training for the team). Mutually agreed upon standards has to be adhered to. Hardware and software platforms may remain a constraint.

Designer try answer the “How” part of “What” is raised during the requirement phase. As such the solution proposed should be contemporary. To that extent a designer should know what is happening in technology. Large, central computer systems with proprietary architecture are being replaced by distributed network of low cost computers in an open systems environment We are moving away from conventional software development based on hand generation of code (COBOL,

C) to Integrated programming environments. Typical applications today are internet based.

The process of design involves conceiving and planning out in the mind" and “making a drawing, pattern, or sketch of". In software design, there are three distinct types of activities: external design, architectural design, and detailed design. Architectural and detailed designs are collectively referred to as internal design. External design of software involves conceiving, planning out, and specifying the externally observable characteristics of a software product. These characteristics include user displays and report formats, external data sources and data sinks, and the functional characteristics, performance requirements, and high-level process structure for the product. External design begins during the analysis phase and continues in the design phase. In practice, it is not possible to perform requirements definition without doing some preliminary design. Requirements definition is concerned with specifying the external, functional, and performance requirements for a system, as well as exception handling and the other items. External design is concerned with refining those requirement and establishing the high level structural view of the system, Thus, the distinction between requirements definition and external design is not sharp, but is rather a gradual shift in emphasis from detailed "what" to high-level 'how".

Internal design involves conceiving, planning out, and specifying the internal structure and processing details of the software product. The goals of internal design are to specify internal structure and processing details, to record design decisions and indicate why certain alternatives and trade-offs were chosen, to elaborate the test plan, and to provide a blueprint for implementation, testing, and maintenance activities. The work products of internal design include a specification of architectural structure, the details of algorithms and data structures, and the test plan.

Architectural design is concerned with refining the conceptual view of the system, identifying the internal processing functions, decomposing the high level functions into sub functions, defining internal data streams and data stores. Issues of concern during detailed design include specification of algorithm, concrete data structures that implement the data stores, interconnection among the functions etc.

The test plan describes the objectives of testing, the test completion criteria, the integration plan (strategy, schedule, and responsible individuals), particular tools and techniques to be used, and the actual test cases and expected results. Functional tests and Performance tests are developed during requirements analysis and are refined during the design phase. Tests that examine the internal structure of the software product and tests that attempt to break the system (stress tests) are developed during detailed design and implementation.

External design and architectural design typically span the Period from Software Requirements Review (SRR) to Preliminary Design Review (PDR). Detailed design spans the period from preliminary design review to Critical Design Review (CDR).

Phases →	Analysis	Design	Implementation
Activities →	Planning Requirement Definition	External Architectural Detailed	Coding Debugging Testing

Reviews →	SRR	PDR	CDR

6.2.4 FUNDAMENTAL DESIGN CONCEPTS

Fundamental concepts of software design include abstraction, structure, information hiding, modularity, concurrency, verification, and design aesthetics.

6.2.4.1 Abstraction

Abstraction is the intellectual tool that allows us to deal with concepts apart from particular instances of those concepts. During requirements definition and design, abstraction permits separation of the conceptual aspects of a system from the implementation details. We can, for example, specify the FIFO property of a queue or the LIFO property of a stack without concern for the representation scheme to be used in implementing the stack or queue. Similarly, we can specify the functional characteristics of the routines that manipulate data structures (e.g., NEW, PUSH, POP, TOP, EMPTY) without concern for the algorithmic details of the routines.

During software design, abstraction allows us to organize and channel our thought processes by postponing structural considerations and detailed algorithmic considerations until the functional characteristics, data streams, and data stores have been established. Structural considerations are then addressed prior to consideration of algorithmic details. This approach reduces the amount of complexity that must be dealt with at any particular point in the design process.

Architectural design specifications are models of software in which the functional and structural attributes of the system are emphasized. During detailed design, the architectural structure is refined into implementation details. Design is thus a process of proceeding from abstract considerations to concrete representations. Three widely used abstraction mechanisms in software design are functional abstraction, data abstraction, and control abstraction. These mechanisms-allow us to control the complexity of the design process by systematically proceeding from abstract to the concrete. Functional abstraction involves the use of parameterized sub programs. The ability to parameterize a subprogram and to bind different values on different invocations of the subprogram is a powerful abstraction mechanism. Functional abstraction can be generalized to collection of subprograms i.e. groups. Within a group, certain routines have the visible property, which allows them to be used by routines in other groups. Routines without the visible property are hidden from other groups. A group thus provides a functional abstraction in which the visible routines communicate with other groups and the hidden routines exist to support the visible ones.

Data abstraction involves specifying a data type or data object by specifying legal operations on objects. Many modern programming languages provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.

Control abstraction is the third commonly used abstraction mechanism in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details.

IF statements and WHILE statements in modern programming languages are abstractions of machine code implementations that involve conditional jump instructions. A statement of the form leaves unspecified the sorting technique, the nature of S, the nature of the files, and how "for all I in S" is to be handled. Another example of control abstraction is the monitor construct, which is a control abstraction for concurrent programming; implementation details of the operator are hidden inside the construct. At the architectural design level, control abstraction permits specification of sequential subprograms, exception handlers, and co-routines and concurrent program units without concern for the exact details of implementation.

6.2.4.2 Information Hiding

Information hiding is a fundamental design concept for software. When a software system is designed using the information hiding approach, each module in the system hides the internal details of its processing activities and modules communicates only thru well defined interfaces. Design should begin with a list of difficult design decisions and design decisions that are likely to change. Each module is designed to hide such a decision from the other modules. Because these design decisions transcend execution time, design modules may not correspond to processing steps in the implementation of the system. In addition to hiding of difficult and changeable design decisions, other candidates for information hiding include:

- A data structure, its internal linkage, and the implementation details of the procedures that manipulate it (this is the principle of data abstraction)

- The format of control blocks such as those for queues in an operating system (a "control-block" module)
- Character codes, ordering of character sets, and other implementation details
- Shifting, masking, and other machine dependent details

Information hiding can be used as the principal design technique for architectural design of a system, or as a modularization criterion in conjunction with other design

6.2.4.3 Modularity

There are many definitions of the term "module." They range from "a module is a FORTRAN subroutine" to "a module is an Ada package" to "a module is a work assignment for an individual programmer". All of these definitions are correct, in the sense that modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved. Modular systems consist of well-defined, manageable units with well-defined interfaces among the units. Desirable properties of a modular system include:

- Each processing abstraction is a well-defined subsystem that is potentially useful in other applications.
- Each function in each abstraction has a single, well-defined purpose.
- Each function manipulates no more than one major data structure.
- Functions share global data selectively. It is easy to identify all routines that share a major data structure.

- Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

6.2.5 Modularization criteria

Architectural design has the goal of producing well-structured, modular software systems. In this section of the text, we consider a software module to be a named entity having the following characteristics:

- Modules contain instructions, processing logic, and data structures.
- Modules can be separately compiled and stored in a library.
- Modules can be included in a program.
- Module segments can be used by invoking a name and some parameters.
- Modules can use other modules.

Examples of modules include procedures, subroutines, and functions; functional groups of related procedures, subroutines, and functions; data abstraction groups; utility groups; and concurrent processes. Modularization allows the designer to decompose a system into functional units, to impose hierarchical ordering on function usage, to implement data abstraction, to develop independently useful subsystems. In addition, modularization can be used to isolate machine dependencies, to improve the performance of a software product, or to ease debugging, testing, integration, tuning, and modification of the system.

There are numerous criteria that can be used to guide the modularization of a system. Depending on the criteria used, different system structures may result. Modularization criteria include the conventional criterion, in which each module and its sub modules correspond to a processing step in the execution sequence; the information hiding criterion, in which each module hides a difficult or changeable design decision from the other modules; the data abstraction criterion, in which each module hides the representation details of a major data structure behind functions that access and modify the data structure; levels of abstraction, in which modules and collections of modules provide a hierarchical set of increasingly complex services; coupling-cohesion, in which a system is structured to maximize the cohesion of elements in each module and to minimize the coupling between modules; and problem modeling, in which the modular structure of the system matches the structure of the problem being solved. There are two versions of problem modeling: either the data structures match the problem structure and the visible functions manipulate the data structures, or the modules form a network of communicating processes where each process corresponds to a problem entity.

Coupling and cohesion

A fundamental goal of software design is to structure the software product so that the number and complexity of interconnection between modules is minimized. A good heuristic for achieving this goal involves the concepts of coupling and cohesion.

6.2.5.1 Coupling

Coupling is the measure of strength of association established by a connection from one module to another. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system (‘ripple effect’). The use of global variables can result in an enormous number of connections between the modules of a program. The degree of coupling between two modules is a function of several factors: (1) How complicated the connection is, (2) Whether the connection refers to the module itself or something inside it, and (3) What is being sent or received. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Low coupling means that one module does not have to be concerned with the internal implementation of another module, and interacts with another module with a stable interface. With low coupling, a change in one module will not require a change in the implementation of another module. Low coupling is a sign of a well structured computer system.

However, in order to achieve maximum efficiency, a highly coupled system is sometimes needed. In modern computing systems, performance is often traded for lower coupling; the gains in the software development process are greater than the value of the running performance gain.

Low-coupling / high-cohesion is a general goal to achieve when structuring computer programs, so that they are easier to understand and maintain.

The concepts are usually related: low coupling implies high cohesion and vice versa. In the field of object-oriented programming, the connection between classes tends to get lower (low coupling), if we group related methods of a class together (high cohesion). The different types of coupling, in order of lowest to highest, are as follows:

- Data coupling
- Stamp coupling
- Control coupling
- External coupling
- Common coupling
- Content coupling

Where data coupling is most desirable and content coupling least.

Data Coupling

Two modules are data coupled if they communicate by parameters (each being an elementary piece of data).E.g. sin (theta) returning sine value, calculate_interest (amount, interest rate, term) returning interest amt.

Stamp Coupling (Data-structured coupling)

Two modules are stamp coupled if one passes to other a composite piece of data (a piece of data with meaningful internal structure). Stamp coupling is when modules share a composite data structure, each module not knowing which part of the data structure will be used by the other (e.g. passing a student record to a function which calculates the student's GPA)

Control Coupling

Two modules are control coupled if one passes to other a piece of information intended to control the internal logic of the other. In Control coupling, one module controls logic of another, by passing it information on what to do (e.g. passing a what-to-do flag).

External coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Common coupling

Two modules are common coupled if they refer to the same global data area. Instead of communicating through parameters, two modules use a global data

Content coupling

Two modules exhibit content coupled if one refers to the inside of the other in any way (if one module 'jumps' inside another module). E.g. Jumping inside a module violate all the design principles like abstraction, information hiding and modularity.

In object-oriented programming, subclass coupling describes a special type of coupling between a parent class and its child. The parent has no connection to the child class, so the connection is one way (i.e. the parent is a sensible class on its own). The coupling is hard to classify as low or high; it can depend on the situation.

We aim for a 'loose' coupling. We may come across a (rare) case of module A calling module B, but no parameters passed between them (neither send, nor received). This is strictly should be positioned at zero point on the scale of coupling (lower than Normal Coupling itself). Two modules A &B are normally coupled if A calls B – B returns to A – (and) all information passed between them is by means of parameters passed through the call mechanism. The other two types of coupling (Common and content) are abnormal coupling and not desired. Even in Normal Coupling we should take care of following issues:

- Data coupling can become complex if number of parameters communicated between is large.
- In Stamp coupling there is always a danger of over-exposing irrelevant data to called module. (Beware of the meaning of composite data. Name represented as an array of characters may not qualify as a composite data. The meaning of composite data is the way it is used in the application NOT as represented in a program)
- "What-to-do flags" are not desirable when it comes from a called module ('inversion of authority'): It is alright to have calling module (by virtue of the fact, is a boss in the hierarchical arrangement) know internals of called module and not the other way around.

In general, use of tramp data and hybrid coupling is not advisable. When data is passed up and down merely to send it to a desired module, the data will have no meaning at various levels. This will lead to tramp data. Hybrid coupling will result when different parts of flags are used (misused?) to mean different things in different places (Usually we may brand it as control coupling – but hybrid coupling complicate connections between modules). Two modules may be coupled in more than one way. In such cases, their coupling is defined by the worst coupling type they exhibit.

In object-oriented programming, coupling is a measure of how strongly one class is connected to another.

Coupling is increased between two classes A and B if:

- A has an attribute that refers to (is of type) B.
- A calls on services of a B object.
- A has a method which references B (via return type or parameter).
- A is a subclass of (or implements) B.

Disadvantages of high coupling include:

- A change in one class forces a ripple of changes in other classes.
- Difficult to understand a class in isolation.
- Difficult to reuse or test a class because dependent class must also be included.

One measure to achieve low coupling is functional design: it limits the responsibilities of modules. Modules with single responsibilities usually need to communicate less with other modules, and this has the virtuous side-effect of reducing coupling and increasing cohesion in many cases.

6.2.5.2 Cohesion

Designers should aim for loosely coupled and highly cohesive modules. Coupling is reduced when the relationships among elements not in the same module are minimized. Cohesion on the other hand aims to maximize the relationships among elements in the same module. Cohesion is a good measure of the maintainability of a module. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand. The types of cohesion, in order of lowest to highest, are as follows:

1. Coincidental Cohesion (Worst)
2. Logical Cohesion
3. Temporal Cohesion
4. Procedural Cohesion
5. Communicational Cohesion
6. Sequential Cohesion
7. Functional Cohesion (Best)

Coincidental cohesion (worst)

Coincidental cohesion is when parts of a module are grouped arbitrarily; the parts have no significant relationship (e.g. a module of frequently used functions).

Logical cohesion

Logical cohesion is when parts of a module are grouped because of a slight relation (e.g. using control coupling to decide which part of a module to use, such as how to operate on a bank account).

Temporal cohesion

In a temporally bound (cohesion) module, the elements are related in time. Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Procedural cohesion

Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Communicational cohesion

Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a method updateStudentRecord which operates on a student record, but the actions which the method performs are not clear).

Sequential cohesion

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part (e.g. a function which reads data from a file and processes the data).

Functional cohesion (best)

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (a perfect module).

Since cohesion is a ranking type of scale, the ranks do not indicate a steady progression of improved cohesion. Studies by various people including Larry Constantine and Edward Yourdon as well as others indicate that the first two types of cohesion are much inferior to the others and that module with communicational cohesion or better tend to be much superior to lower types of cohesion. The seventh type, functional cohesion, is considered the best type.

However, while functional cohesion is considered the most desirable type of cohesion for a software module, it may not actually be achievable. There are many cases where communicational cohesion is about the best that can be attained in the circumstances. However the emphasis of a software design should be to maintain module cohesion of communicational or better since these types of cohesion are associated with modules of lower lines of code per module with the source code focused on a particular functional objective with less extranous or unnecessary functionality, and tend to be reusable under a greater variety of conditions.

Example: Let us create a module that calculates average of marks obtained by students in a class:

```
calc_stat(){read (x[]); a = average (x); print a}
average (m){sum=0; for i = 1 to N { sum = sum + x[i]; } return (sum/N);}
```

In average() above, all of the elements are related to the performance of a single function. Such a functional binding (cohesion) is the strongest type of binding. Suppose we need to calculate standard deviation also in the above problem, our pseudo code would look like:

```
calc_stat(){ read (x[]); a = average (x); s = sd (x, a); print a, s;}
average(m) // function to calculate average
{sum =0; for i = 1 to N { sum = sum + x[i]; } return (sum/N);}
sd (m, y) //function to calculate standard deviation
{ …}
```

Now, though average () and sd () are functionally cohesive, calc_stat() has a sequential binding (cohesion). Like a factory assembly line, functions are arranged in sequence and output from average () goes as an input to sd(). Suppose we make sd () to calculate average also, then calc_stat() has two functions related by a reference to the same set of input. This results in communication cohesion.

Let us make calc-stat() into a procedure as below:

```
calc_stat(){
sum = sumsq = count = 0
for i = 1 to N
read (x[i])
sum = sum + x[i]
sumsq = sumsq + x[i]*x[i]
…}
a = sum/N
s = … // formula to calculate SD
print a, s
}
```

Now, instead of binding functional units with data, calc-stat() is involved in binding activities through control flow. calc-stat() has made two statistical functions into a procedure. Obviously, this arrangement affects reuse of this module in a different context (for instance, when we need to calculate only average not std. dev.). Such cohesion is called procedural.

A good design for calc_stat () could be (Figure 6.1):

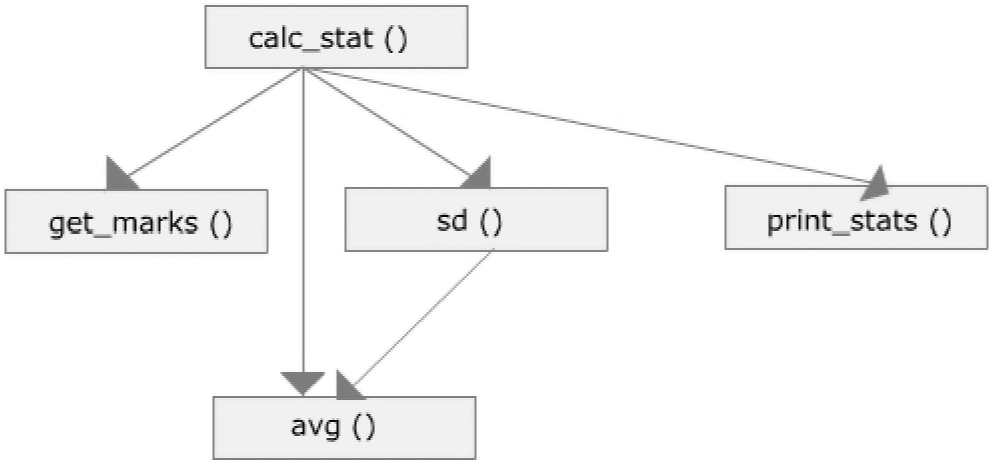


Figure 6.1

A logically cohesive module contains a number of activities of the same kind. To use the module, we may have to send a flag to indicate what we want (forcing various activities sharing the interface). Examples are a module that performs all input and output operations for a program. The activities in a logically cohesive module usually fall into same category (validate all input or edit all data) leading to sharing of common lines of code (plate of spaghetti?). Suppose we have a module with possible statistical measures (average, standard deviation). If we want to calculate only average, the call to it would look like calc_all_stat (x[], flag). The flag is used to indicate out intent i.e. if flag=0 then function will return average, and if flag=1, it will return standard deviation.

```
calc_stat(){ read (x[]); a = average (x); s = sd (x, a); print a, s;}
calc_all_stat(m, flag)
{
If flag=0{sum=0; for i = 1 to N { sum = sum + x[i]; }return (sum/N);}
If flag=1{ …….; return sd;}
```

}

6.2.5.3 Other Modularization Criteria

Additional criteria for deciding which functions to place in which modules of software system include: hiding difficult and changeable design decisions, limiting the physical size of modules, structuring the system to improve the observability, and testability, isolating machine dependencies to few routines, easing likely changes, providing general purpose utility functions, developing an acceptable overlay structure in a machine with limited memory capacity, minimizing page faults in a virtual memory machine, and reducing the call return overhead of excessive subroutine calls. For each software product, the designer must weigh these factors and develop a consistent set of modularization criteria to guide the design process. Efficiency of the resulting implementation is a concern that frequently arises when decomposing a system into modules. A large number of small modules having data coupling and functional cohesion implies a large execution time overhead for establishing run-time linkages between the modules. The preferred technique for optimizing the efficiency of a system is to first design and implements the system in a highly modular fashion. System performance is then measured, and bottlenecks are removed by reconfiguring and recombining modules, and by hand coding certain critical linkages and critical routines in assembly language if necessary. In these situations, the modular source code should be retained as documentation for the assembly language routines. The soundness of this technique is based on two observations. First, most software systems spend a large portion of processing time in a small portion of the code; typically 80 percent or more of execution time is spent in 20 percent or less of the code. Furthermore, the region of code where the majority of time is spent is usually not predictable until the program is implemented and actual performance is measured. Second, it is relatively easy to reconfigure and recombine small modules into larger units if necessary for better performance; however, failure to initially decompose a system far enough may prevent identification of a function that can be used in other contexts.

6.2.6 Popular Design Methods

Popular Design Methods include

1) Modular decomposition

- Based on assigning functions to components.
- It starts from functions that are to be implemented and explain how each component will be organized and related to other components.

2) Event-oriented decomposition

- Based on events that the system must handle.
- It starts with cataloging various states and then describes how transformations take place.

3) Object-oriented design

- Based on objects and their interrelationships
- It starts with object types and then explores object attributes and actions.

Structured Design – It uses modular decomposition

6.2.7 Design Notation

In software design the representation schemes used are of fundamental importance. Good notation can clarify the interrelationships and interactions of interest, while poor notation can complicate and interfere with good design practice. At least, three levels of design specifications exist; external design specifications, which describe the external characteristics of software systems; architectural design specifications, which describe the structure of the system; and detailed design specifications, which describe control flow, data representation, and other algorithmic details within the modules.

During the design phase, there are two things of interest: the design of the system, producing which are the basic objective of this phase, and the process of designing itself. It is for the latter that principles and methods are needed. In addition, while designing, a designer needs to record his thoughts and decisions, to represent the design so that he can view it and play with it. For this, design notations are used.

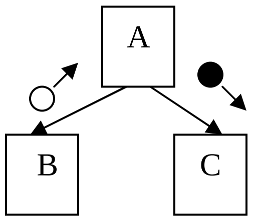
Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. They are meant largely for the designer so that he can quickly represent his decisions in a compact manner that he can evaluate and modify. These notations are frequently graphical.

Once the designer is satisfied with the design he has produced, the design is to be precisely specified in the form of a document. To specify the design, specification languages are used. Producing the design specification is the ultimate objective of the design phase. The purpose of this design document is quite different from that of the design notation. Whereas a design represented using the design notation is largely to be used by the designer, a design specification has to be so precise and complete that it can be used as a basis of further development by other programmers. Generally, design specification uses textual structures, with design notation helping in understanding.

Here, we first describe a design notation structure charts that can be used to represent a function-oriented design. Then, we describe a simple design language to specify a design. Though the design document, the final output of the design activity, typically also contains other things like design decisions taken and background, its primary purpose is to document the design itself. We will focus on this aspect only.

6.2.7.1 Structure Charts

Structure charts, a graphic representation of the structure, are used during architectural design to document hierarchical structure, parameters, and interconnections in a system. The structure of a program is made up of the modules of that program together with the interconnections between modules. Every computer program has a structure, and given a program, its structure can be determined. In a structure chart, a box represents a module with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the subordinate of A, and A is called the super-ordinate of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail).



Unlike flowcharts, structure chart do not represent the structural information. So generally decision boxes are not there. However, there are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can also be represented in a structure chart. For example, let us consider a situation where module A has subordinates B, C, and D, and A repeatedly calls the modules C and D. This can be represented by a looping arrow around the arrows joining the subordinates C and D to A, as shown in figure 6.2. All the subordinate modules activated within a common loop are enclosed in the same looping arrow.

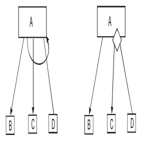


FIGURE 6.2 ITERATION AND DECISION REPRESENTATION

Major decisions can be represented, similarly. For example, if the invocation of modules C and D in module A depends on the outcome of some decision, that is represented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond, as shown in above Figure.

Modules in a system can be categorized into few classes. There are some modules that obtain information from their subordinates and then pass it to their super-ordinate. This kind of module is an input module. Similarly, there are output modules that take information from their super-ordinate and pass it on to its subordinates. As the name suggests, the input and output modules are, typically, used for input and output of data, from and to the environment. The input modules get the data from the sources and get it ready to be processed, and the output modules take the output produced and prepare it for proper presentation to the environment. Then, there are modules that exist solely for the sake of transforming data into some other form. Such a module is called a transform module. Most of the computational modules typically fall in this category. Finally, there are modules whose primary concern is managing the flow of data to and from different subordinates. Such modules are called coordinate modules. The structure chart representation of the different types of modules is shown in following Figure.

A module can perform functions of more than one type of module. For example, the composite module in Figure 6.3 is an input module, from the point of view of its super-ordinate, as it feeds the data Y to the super-ordinate. Internally, A is a coordinate module and views its job as getting data X from one subordinate and passing it to another subordinate, who converts it to Y. Modules in actual systems are often composite modules.

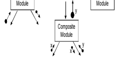


FIGURE 6.3 DIFFERENT TYPES OF MODULES.

A structure chart is a nice representation mechanism for a design that uses functional abstraction. It shows the modules and their call hierarchy, the interfaces between the modules, and what information passes between modules. It is a convenient and compact notation that is very useful while creating the design. That is, a designer can make effective use of structure charts to represent the model he is creating while he is designing. However, it is not very useful for representing the final design, as it does not give all the information needed about the design. For example, it does not specify the scope, structure of data, specifications of each module, etc. Hence, it is generally not used to convey design to the implementer.

We have seen how to determine the structure of an existing program. But, once the program is written, its structure is fixed; little can be done about altering the structure. However, for a given set of requirements many different programs can be written to satisfy the requirements, and each program can have a different structure. That is, although the structure of a given program is fixed, for a given set of requirements, programs with different structures can be obtained. The objective of the design phase using function-oriented method is to control the eventual structure of the system by fixing the structure during design.

6.2.7.2 Data Flow diagrams (DFD)

DFD is a directed graph in which node specifies process and arcs specify data items transmitted between processing nodes. Unlike flowcharts, DFD do not indicate decision logic under which various processing nodes in the diagram might be activated. DFD can be used during requirement analysis as well as in design phase to specify external and top-level internal design specification. The following symbols(Figure 6.4) are used to construct a DFD:

Data Flow	→
Process	○
Data Stores	⊏
Data Source/Sink	▭

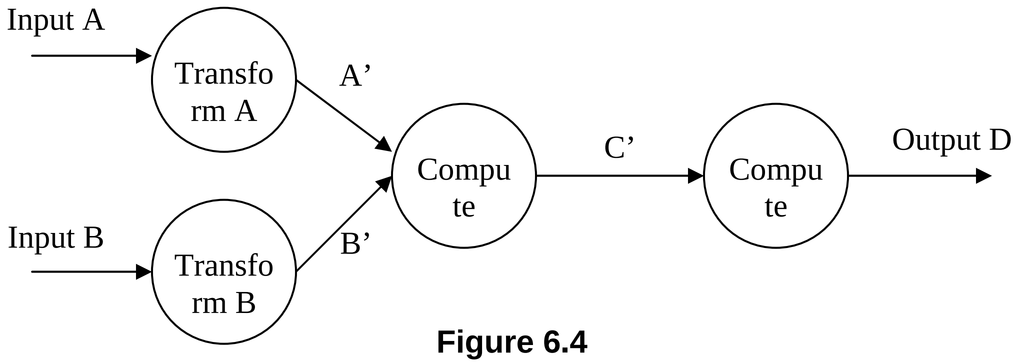


Figure 6.4

An informal Data Flow Diagram

6.2.7.3 Pseudocode

Pseudocode notation can be used in architectural as well as detailed design phase. In it the designer describes system characteristics using short, concise, English language phrases that are structured by keywords such as If-Then-else, While-Do, and End. Keywords and indentation describes the flow of control, while the English phrases describe processing actions. E.g.

```
INITIALIZE tables and counters
OPEN files
```

```
READ the first ten records
WHILE there are more records DO
    WHILE there are more words in the text record DO
    --------
    --------
    ENDWHILE at the end of the text record
ENDWHILE when all the text records have been processed
PRINT ……
CLOSE file
TERMINATE the program
```

6.3 Summary

Design is the bridge between software requirements and implementation that specifies those requirements. The goal of architectural design is to specify a system structure that satisfies the requirements, the external design specification, and the implementation constraints. Detailed design provides the algorithmic details, data representation etc. Design is an important phase. Design principles and concepts establish a foundation for the creation of the design model that encompasses representation of data, architecture, interface, and procedures. Design principles and concepts guide the software engineer. Concept of modularity helps the designer in producing a design, which is simple and modifiable. Two important criteria for modularity are coupling and cohesion. Coupling is a measure of interconnection among modules. Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion and coupling are closely related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is. Structured design uses the modular decomposition. Design notations discussed in this chapter are data flow diagrams, structured chart and Pseudocode. Structure chart is a good notation to represent the structured design. The structure chart of a program is a graphic representation of its structure. In a structure chart, a box represents a module with the module name written in the box. An arrow from module A to module B represents that module A invokes module B.

6.4 Key words

Abstraction, coupling, cohesion, Design, Data Flow Diagram, Information hiding, Modularity, Structured Chart, Pseudocode.

6.5 Self-assessment questions

1. Define design. What are the desirable qualities of a good design? Explain.
2. What is a module? What are the advantages of a modular design?
3. What do you understand by coupling and cohesion? What is the relationship between them?
4. Define coupling. What are the different types of coupling? Explain.
5. Define cohesion. Discuss the different types of cohesion using suitable examples.
6. What can be the other criteria for modularization apart from coupling and cohesion?

7. What do you understand by design notations? Discuss the difference between flowchart and data flow diagrams.

6.6 References/Suggested readings

21. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
22. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
23. Software Engineering by Sommerville, Pearson Education.
24. Software Engineering – A Practitioner’s Approach by Roger S Pressman, McGraw-Hill.

Lesson number: 7	**Writer: Dr. Rakesh Kumar**
Software Design - II	**Vetter: Dr. PK Bhatia**

7.0 Objectives

The objective of this lesson is to get the students acquainted with the design activities, to provide a systematic approach for the derivation of the design - the blueprint from which software is constructed. This lesson will help them in understanding how a software design may be represented as a set of functions. This lesson introduces them with the notations, which may be used to represent a function-oriented design.

7.1 Introduction

Design is a process in which representations of data structure, program structure, interface characteristics, and procedural details are synthesized from information requirements. During design a large system can be decomposed into sub-systems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design. In architectural design process the activities carried out are system structuring (system is decomposed into sub-systems and communications between them are identified), control modeling, modular decomposition. In a structured approach to design, the system is decomposed into a set of interacting functions.

7.2 Presentation of contents

7.2.1 Transition from Analysis to Design

7.2.1 Transition from Analysis to Design

The flow of information during design is shown in the following figure (7.1). The data design transforms the data model created during analysis into the data structures that will be required to implement the software.

The architectural design defines the relationship between major structural elements of the software, the design patterns that can be used to achieve the requirements and the constraints that affect the implementation.

The interface design describes how the software communicates within itself, and with humans who use it.

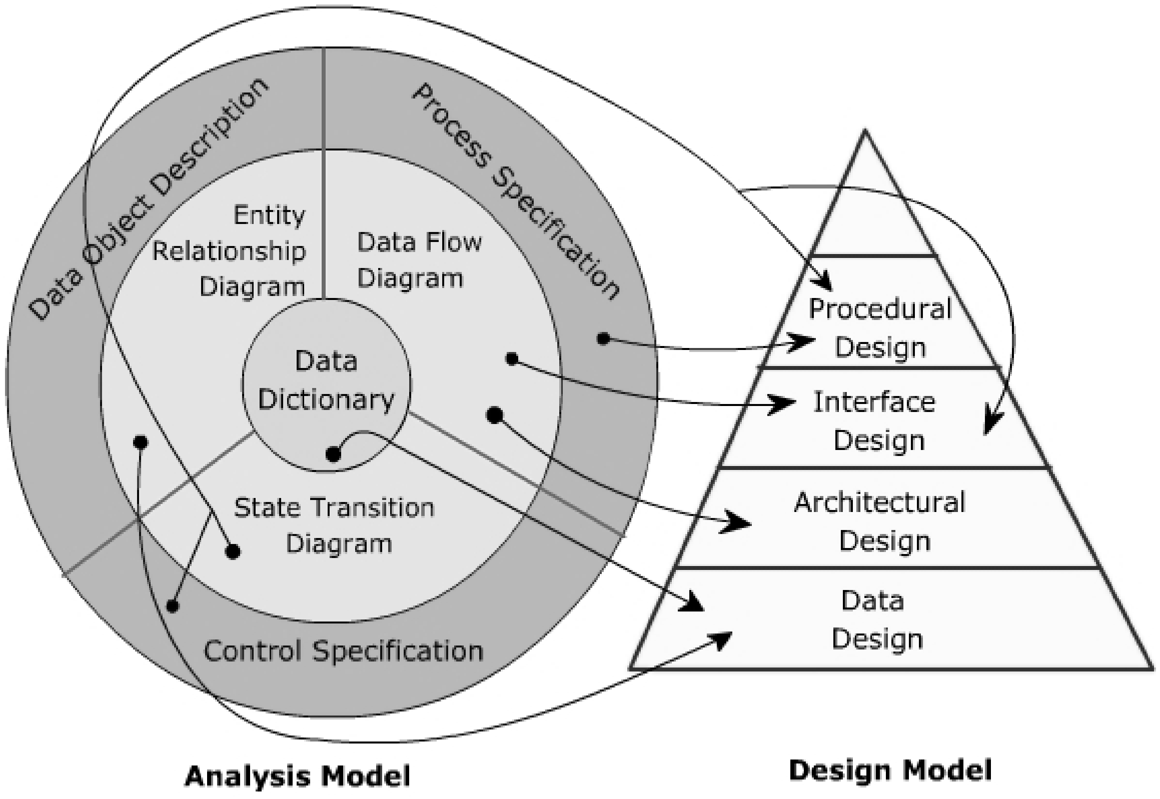


Figure 7.1

The Procedural design (typically, Low Level Design) elaborates structural elements of the software into procedural (algorithmic) description.

7.2.2 High Level Design Activities

Broadly, High Level Design includes Architectural Design, Interface Design and Data Design.

7.2.2.1 Architectural Design

Software architecture is the first step in producing a software design. Architecture design associates the system capabilities with the system components (like modules) that will implement them. The architecture of a system is a comprehensive framework that describes its form and structure, its components and how they interact together. Generally, a complete architecture plan addresses the functions that the system provides, the hardware and network that are used to develop and operate it, and the software that is used to develop and operate it. An architecture style involves its components, connectors, and constraints on combining components. Shaw and Garlan describe seven architectural styles. Commonly used styles include

- Pipes and Filters
- Call-and-return systems
    - Main program / subprogram architecture
- Object-oriented systems
- Layered systems
- Data-centered systems
- Distributed systems
    - Client/Server architecture

In Pipes and Filters, each component (filter) reads streams of data on its inputs and produces streams of data on its output. Pipes are the connectors that transmit output from one filter to another. E.g. Programs written in UNIX shell.

In Call-and-return systems, the program structure decomposes function into a control hierarchy where a “main” program invokes (via procedure calls) a number of program components, which in turn may invoke still other components. E.g.

Structure Chart is a hierarchical representation of main program and subprograms.

In Object-oriented systems, component is an encapsulation of data and operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message calls.

In Layered systems, each layer provides service to the one outside it, and acts as a client to the layer inside it. They are arranged like an "onion ring". E.g. OSI ISO model.

Data-centered systems use repositories. Repository includes a central data structure representing current state, and a collection of independent components that operate on the central data store. In a traditional database, the transactions, in the form of an input stream, trigger process execution. E.g. Database.

A popular form of distributed system architecture is the Client/Server where a server system responds to the requests for actions / services made by client systems. Clients access server by remote procedure call.

The following issues are also addressed during architecture design:

- Security
- Data Processing: Centralized / Distributed / Stand-alone
- Audit Trails
- Restart / Recovery
- User Interface

7.2.2.2 Architectural design process

Data flow oriented design is an architectural design method that allows a convenient transition from the analysis model to a design description of program structure. The strategy for converting the DFD (representation of information flow) into Structure Chart is discussed below:

- Break the system into suitably tractable units by means of transaction analysis
- Convert each unit into a good structure chart by means of transform analysis
- Link back the separate units into overall system implementation

7.2.2.3 Transaction Analysis

The transaction is identified by studying the discrete event types that drive the system. For example, with respect to railway reservation, a customer may give the following transaction stimulus (Figure 7.2):

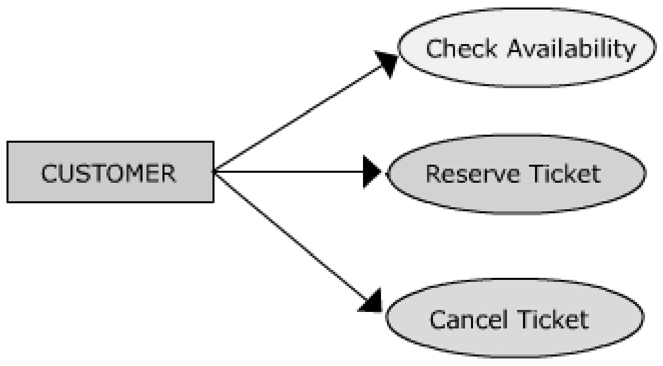


Figure 7.2

The three transaction types here are: Check Availability (an enquiry), Reserve Ticket (booking) and Cancel Ticket (cancellation). On any given time we will get customers interested in giving any of the above transaction stimuli. In a typical situation, any one stimulus may be entered through a particular terminal. The human user would inform the system her preference by selecting a transaction type from a menu. The first step in our strategy is to identify such transaction types and draw the first level breakup of modules in the structure chart, by creating separate module to co-ordinate various transaction types. This is shown as follows (Figure 7.3):

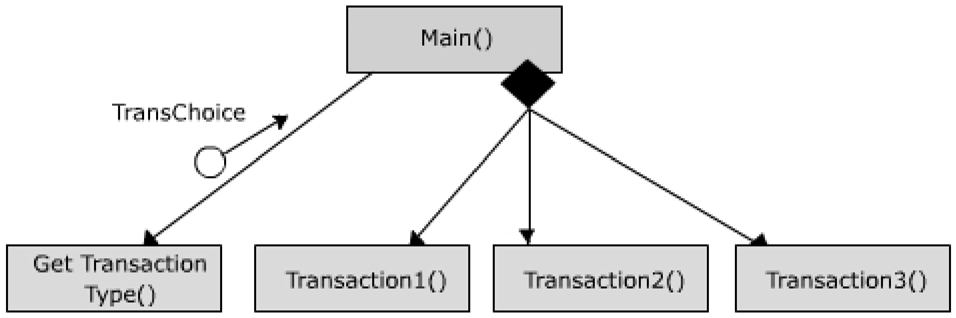


Figure 7.3

The Main ( ) which is a over-all coordinating module, gets the information about what transaction the user prefers to do through TransChoice. The TransChoice is returned as a parameter to Main ( ). Remember, we are following our design principles faithfully in decomposing our modules. The actual details of how GetTransactionType ( ) is not relevant for Main ( ). It may for example, refresh and print a text menu and prompt the user to select a choice and return this choice to Main ( ). It will not affect any other components in our breakup, even when this module is changed later to return the same input through graphical interface instead of textual menu. The modules Transaction1 ( ), Transaction2 ( ) and Transaction3 ( ) are the coordinators of transactions one, two and three respectively. The details of these transactions are to be exploded in the next levels of abstraction.

We will continue to identify more transaction centers by drawing a navigation chart of all input screens that are needed to get various transaction stimuli from the user. These are to be factored out in the next levels of the structure chart (in exactly the same way as seen before), for all identified transaction centers.

7.2.2.4 Transform Analysis

Transform analysis is strategy of converting each piece of DFD (may be from level 2 or level 3, etc.) for all the identified transaction centers. In case, the given system has only one transaction (like a payroll system), then we can start transformation from level 1 DFD itself. Transform analysis is composed of the following five steps:

1. Draw a DFD of a transaction type (usually done during analysis phase)
2. Find the central functions of the DFD
3. Convert the DFD into a first-cut structure chart
4. Refine the structure chart
5. Verify that the final structure chart meets the requirements of the original DFD

Let us understand these steps through a payroll system example:

7.2.2.4.1 Identifying the central transform

The central transform is the portion of DFD that contains the essential functions of the system and is independent of the particular implementation of the input and output. One way of identifying central transform is to identify the centre of the DFD by pruning off its afferent and efferent branches. Afferent stream is traced from outside of the DFD to a flow point inside, just before the input is being transformed into some form of output (For example, a format or validation process only refines the input – does not transform it). Similarly an efferent stream is a flow point from where output is formatted for better presentation.

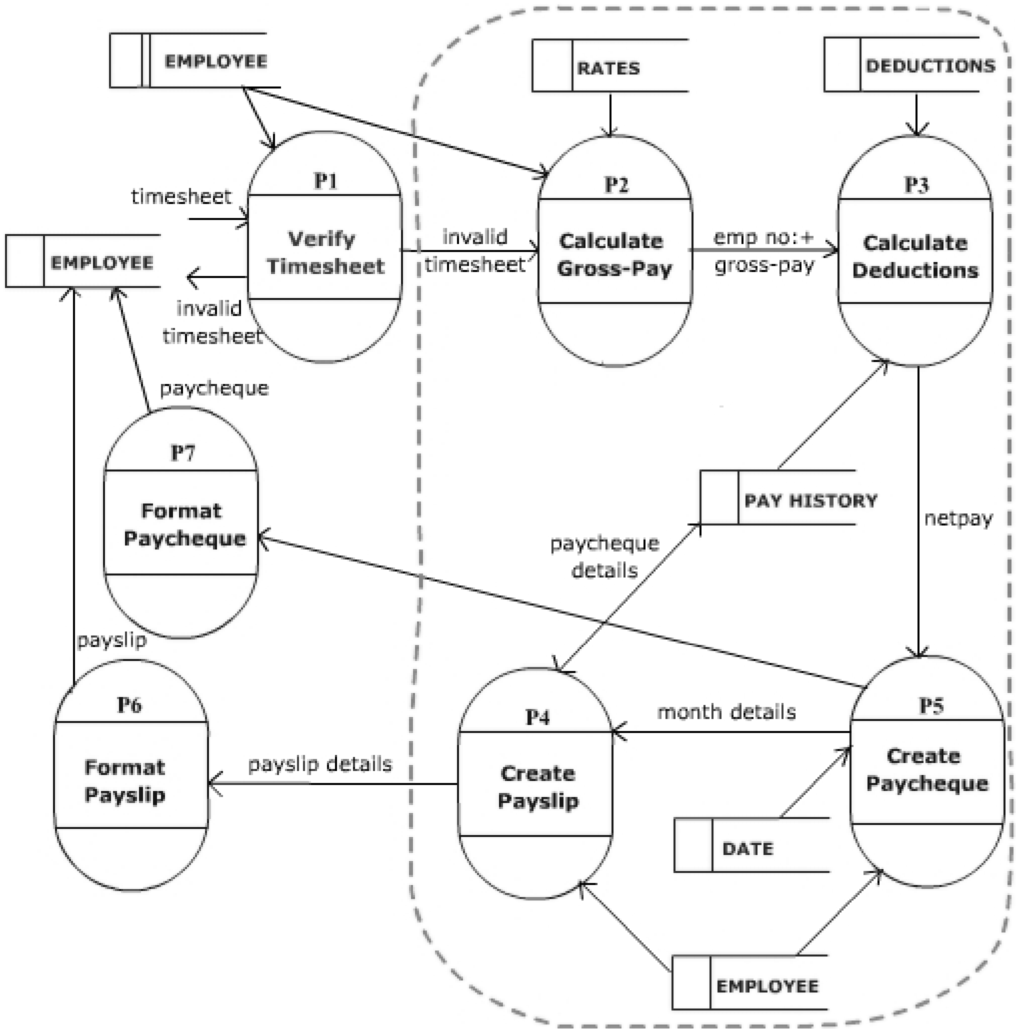


Figure 7.4

The processes between afferent and efferent stream represent the central transform (marked within dotted lines above in figure 7.4). In the above example, P1 is an input process, and P6 & P7 are output processes. Central transform processes are P2, P3, P4 & P5 - which transform the given input into some form of output.

7.2.2.4.2 First-cut Structure Chart

To produce first-cut (first draft) structure chart, first we have to establish a boss module. A boss module can be one of the central transform processes. Ideally, such process has to be more of a coordinating process (encompassing the essence of transformation). In case we fail to find a boss module within, a dummy-coordinating module is created

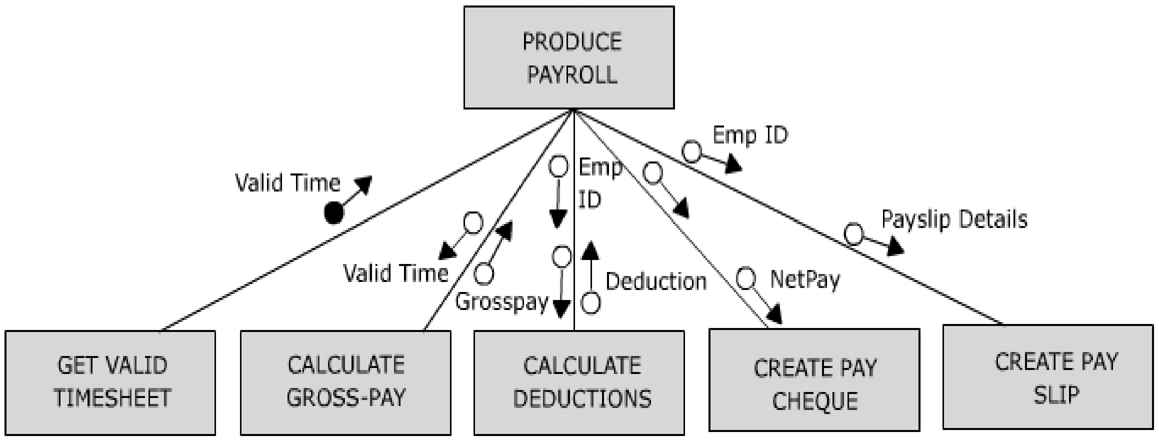


Figure 7.5

In the above illustration in figure 7.5, we have a dummy boss module "Produce Payroll" – which is named in a way that it indicate what the program is about. Having established the boss module, the afferent stream processes are moved to left most side of the next level of structure chart; the efferent stream process on the right most side and the central transform processes in the middle. Here, we moved a module to get valid timesheet (afferent process) to the left side. The two central transform processes are move in the middle. By grouping the other two central transform processes with the respective efferent processes, we have created two modules– essentially to print results, on the right side.

The main advantage of hierarchical (functional) arrangement of module is that it leads to flexibility in the software. For instance, if "Calculate Deduction" module is to select deduction rates from multiple rates, the module can be split into two in the next level – one to get the selection and another to calculate. Even after this change, the "Calculate Deduction" module would return the same value.

7.2.2.4.3 Refine the Structure Chart

Expand the structure chart further by using the different levels of DFD. Factor down till you reach to modules that correspond to processes that access source / sink or data stores. Once this is ready, other features of the software like error handling, security, etc. has to be added. A module name should not be used for two different modules. If the same module is to be used in more than one place, it will be demoted down such that "fan in" can be done from the higher levels. Ideally, the name should sum up the activities done by the module and its sub-ordinates.

7.2.2.4.4 Verify Structure Chart vis-à-vis with DFD

Because of the orientation towards the end-product, the software, the finer details of how data gets originated and stored (as appeared in DFD) is not explicit in Structure Chart. Hence DFD may still be needed along with Structure Chart to understand the data flow while creating low-level design.

Some characteristics of the structure chart as a whole would give some clues about the quality of the system. Page-Jones suggest following guidelines for a good decomposition of structure chart:

- Avoid decision splits - Keep span-of-effect within scope-of-control: i.e. A module can affect only those modules which comes under it's control (All sub-ordinates, immediate ones and modules reporting to them, etc.)

- Error should be reported from the module that both detects an error and knows what the error is.
- Restrict fan-out (number of subordinates to a module) of a module to seven. Increase fan-in (number of immediate bosses for a module). High fan-ins (in a functional way) improve reusability.

7.2.2.5 User Interface Design

The design of user interfaces draws heavily on the experience of the designer. Three categories of Human Computer Interface (HCI) design guidelines are

1. General interaction
2. Information display
3. Data entry

7.2.2.5.1 General Interaction

Guidelines for general interaction often cross the boundary into information /display, data entry and overall system control. They are, therefore, all encompassing and are ignored at great risk. The following guidelines focus on general interaction.

- **Be consistent:** Use a consistent formats for menu selection, command input, data display and the myriad other functions that occur in a HCI.
- **Offer meaningful feedback:** Provide the user with visual and auditory feedback to ensure that two way communications (between user and interface) is established.
- **Ask for verification of any nontrivial destructive action:** If a user requests the deletion of a file, indicates that substantial information is to be overwritten,

or asks for the termination of a program, an “Are you sure …” message should appear.

- **Permit easy reversal of most actions:** UNDO or REVERSE functions have saved tens of thousands of end users from millions of hours of frustration. Reversal should be available in every interactive application.
- **Reduce the amount of information that must be memorized between actions:** The user should not be expected to remember a list of numbers or names so that he or she can re-use them in a subsequent function. Memory load should be minimized.
- **Seek efficiency in dialog, motion, and thought:** Keystrokes should be minimized, the distance a mouse must travel between picks should be considered in designing screen layout; the user should rarely encounter a situation where he or she asks, "Now what does this mean."
- **Forgive mistakes:** The system should protect itself from errors that might cause it to fail (defensive programming)
- **Categorize activities by functions and organize screen geography accordingly:** One of the key benefits of the pull down menu is the ability to organize commands by type. In essence, the designer should strive for "cohesive" placement of commands and actions.
- **Provide help facilities that are context sensitive**
- **Use simple action verbs or short verb phrases to name commands:** A lengthy command name is more difficult to recognize and recall. It may also take up unnecessary space in menu lists.

7.2.2.5.2 Information Display

If information presented by the HCI is incomplete, ambiguous or unintelligible, the application will fail to satisfy the needs of a user. Information is "displayed" in many different ways with text, pictures and sound, by placement, motion and size, using color, resolution, and even omission. The following guidelines focus on information display.

- Display only information that is relevant to the current context: The user should not have to wade through extraneous data, menus and graphics to obtain information relevant to a specific system function.
- Don’t bury the user with data; use a presentation format that enables rapid assimilation of information: Graphs or charts should replace voluminous tables.
- Use consistent labels, standard abbreviations, and predictable colors: The meaning of a display should be obvious without reference to some outside source of information.
- Allow the user to maintain visual context: If computer graphics displays are scaled up and down, the original image should be displayed constantly (in reduced form at the corner of the display) so that the user understands the relative location of the portion of the image that is currently being viewed.
- Produce meaningful error messages
- Use upper and lower case, indentation, and text grouping to aid in understanding: Much of the information imparted by a HCI is textual, yet, the

layout and form of the text has a significant impact on the ease with which information is assimilated by the user.

- Use windows to compartmentalize different types of information: Windows enable the user to "keep" many different types of information within easy reach.
- Use “analog” displays to represent information that is more easily assimilated with this form of representation: For example, a display of holding tank pressure in an oil refinery would have little impact if a numeric representation were used. However, thermometer-like displays were used; vertical motion and color changes could be used to indicate dangerous pressure conditions. This would provide the user with both absolute and relative information.
- Consider the available geography of the display screen and use it efficiently: When multiple windows are to be used, space should be available to show at least some portion of each. In addition, screen size should be selected to accommodate the type of application that is to be implemented.

7.2.2.5.3 Data Input

Much of the user's time is spent picking commands, typing data and otherwise providing system input. In many applications, the keyboard remains the primary input medium, but the mouse, digitizer and even voice recognition systems are rapidly becoming effective alternatives. The following guidelines focus on data input:

- Minimize the number of input actions required of the user: Reduce the amount of typing that is required. This can be accomplished by using the mouse to select from pre-defined sets of input; using a "sliding scale" to specify input data across a range of values; using "macros" that enable a single keystroke to be transformed into a more complex collection of input data.

- Maintain consistency between information display and data input: The visual characteristics of the display (e.g., text size, color, and placement) should be carried over to the input domain.
- Allow the user to customize the input: An expert user might decide to create custom commands or dispense with some types of warning messages and action verification. The HCI should allow this.
- Interaction should be flexible but also tuned to the user's preferred mode of input: The user model will assist in determining which mode of input is preferred. A clerical worker might be very happy with keyboard input, while a manager might be more comfortable using a point and pick device such as a mouse.
- Deactivate commands that are inappropriate in the context of current actions: This protects the user from attempting some action that could result in an error.
- Let the user control the interactive flow: The user should be able to jump unnecessary actions, change the order of required actions (when possible in the context of an application), and recover from error conditions without exiting from the program.
- Provide help to assist with all input actions

- Eliminate “Mickey mouse” input: Do not let the user to specify units for engineering input (unless there may be ambiguity). Do not let the user to type .00 for whole number dollar amounts, provide default values whenever possible, and never let the user to enter information that can be acquired automatically or computed within the program

7.2.2.6 Procedural Design

Procedural design occurs after data, architectural, and interface designs have been established. Procedural design specifies procedural details unambiguously. It is concerned with specifying algorithmic details, concrete data representations, interconnections among functions, and data structure. Detailed design is strongly influenced by implementation language, but it is not the same as implementation, detailed design is more concerned with semantic issues and less concerned with syntactic details than is implementation. Implementation addresses issues of programming language syntax, coding style, and internal documentation. Detailed design permits design of algorithm and data representation at a higher level of abstraction and notation than the implementation language provides. Detailed design should be carried to a level where each statement in the design notation will result in a few statements in the implementation language.

7.2.2.7 Structured Programming

The goal of structured programming is to linearize control flow through a computer program so that the execution sequence follows the sequence in which the code is written. The dynamic structure of the program than resemble the static structure of the program. This enhances the readability, testability, and modifiability of the program. This linear flow of control can be achieved by restricting the set of allowed program construct to single entry, single exit formats. These issues are discussed in the following section:

Structure Rule One: Code Block

If the entry conditions are correct, but the exit conditions are wrong, the bug must be in the block. This is not true if execution is allowed to jump into a block. The bug might be anywhere in the program. Debugging under these conditions is much harder.

Rule 1 of Structured Programming: A code block is structured as shown in figure 7.6. In flow-charting terms, a box with a single entry point and single exit point is structured. This may look obvious, but that is the idea. Structured programming is a way of making it obvious that program is correct.

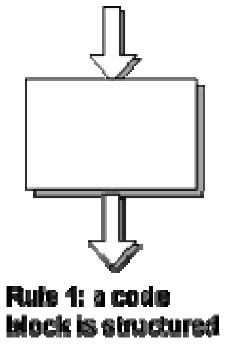


Figure 7.6

Structure Rule Two: Sequence

A sequence of blocks is correct if the exit conditions of each block match the entry conditions of the following block. Execution enters each block at the block's entry point, and leaves through the block's exit point. The whole sequence can be regarded as a single block, with an entry point and an exit point.

Rule 2 of Structured Programming: Two or more code blocks in sequence are structured as shown in figure 7.7.

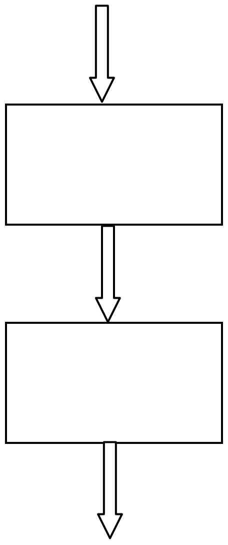


Figure 7.7 Rule 2: A sequence of code blocks is structured

Structure Rule Three: Alternation

If-then-else is sometimes called alternation (because there are alternative choices). In structured programming, each choice is a code block. If alternation is arranged as in the flowchart at right, then there is one entry point (at the top) and one exit point (at the bottom). The structure should be coded so that if the entry conditions are satisfied, then the exit conditions are fulfilled (just like a code block).

Rule 3 of Structured Programming: The alternation of two code blocks is structured as shown in figure 7.8.

An example of an entry condition for an alternation structure is: register $8 contains a signed integer. The exit condition might be: register $8 contains the absolute value of the signed integer. The branch structure is used to fulfill the exit condition.

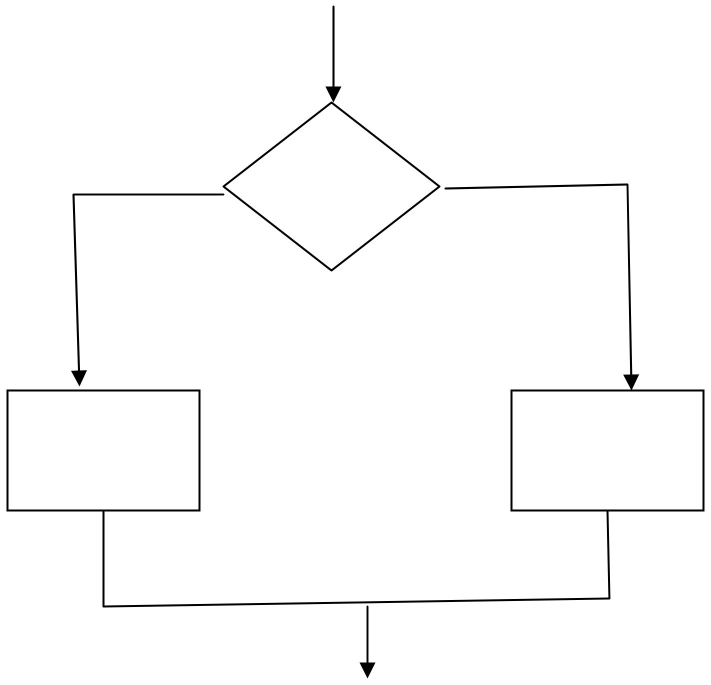


Figure 7.8 Rule 3: An alternation of code blocks is structured

Structure rule four - Iteration

Iteration (while-loop) is arranged as at right. It also has one entry point and one exit point. The entry point has conditions that must be satisfied and the exit point has conditions that will be fulfilled. There are no jumps into the structure from external points of the code.

Rule 4 of Structured Programming: The iteration of a code block is structured as shown in figure 7.9.

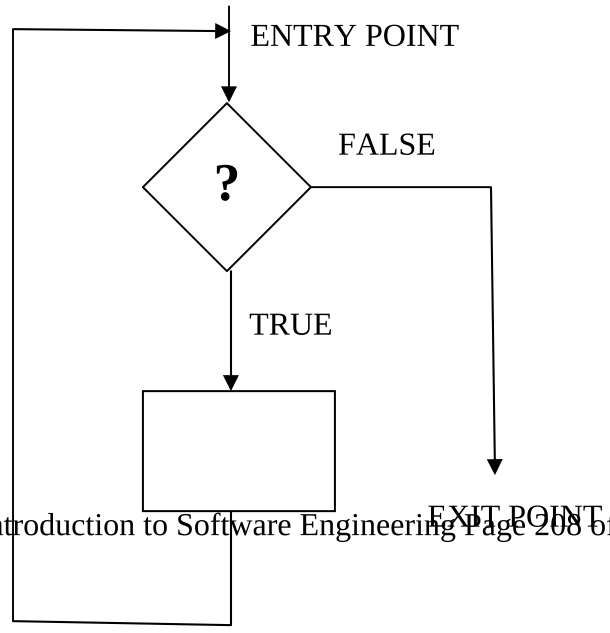


Figure 7.9 Rule 4: The iteration of code block is structured

Structure Rule Five: Nesting Structures

In flowcharting terms, any code block can be expanded into any of the structures. Or, going the other direction, if there is a portion of the flowchart that has a single entry point and a single exit point, it can be summarized as a single code block.

Rule 5 of Structured Programming: A structure (of any size) that has a single entry point and a single exit point is equivalent to a code block.

For example, say that you are designing a program to go through a list of signed integers calculating the absolute value of each one. You might (1) first regard the program as one block, then (2) sketch in the iteration required, and finally (3) put in the details of the loop body, as shown in figure 7.10.

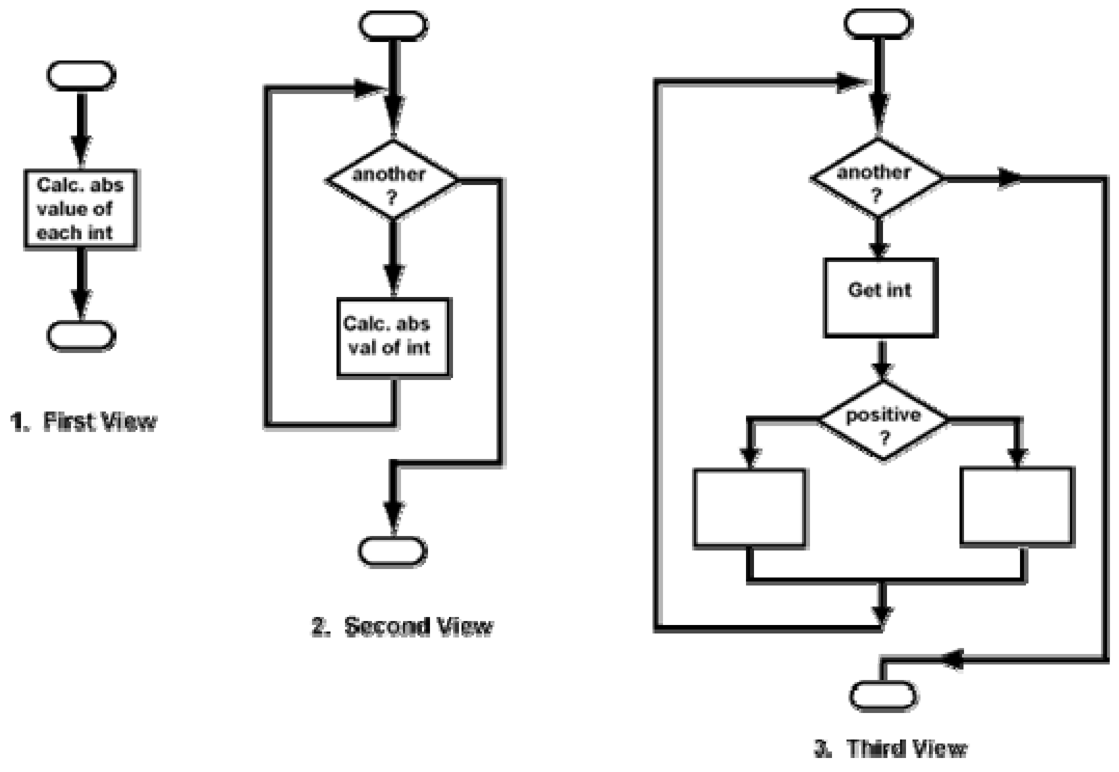


Figure 7.10

Or, you might go the other way. Once the absolute value code is working, you can regard it as a single code block to be used as a component of a larger program.

You might think that these rules are OK for ensuring stable code, but that they are too restrictive. Some power must be lost. But nothing is lost. Two researchers, Böhm and Jacopini, proved that any program could be written in a structured style. Restricting control flow to the forms of structured programming loses no computing power.

The other control structures you may know, such as case, do-until, do-while, and for are not needed. However, they are sometimes convenient, and are usually regarded as part of structured programming. In assembly language they add little convenience

7.2.2.8 Program Design Language

Program design language (PDL), also called structured English or pseudocode, is "a pidgin language, in that, it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)". PDL is used as a generic reference for a design language.

At first glance, PDL looks like a modern programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled (at least not yet). However, PDL tools currently exist to translate PDL into a programming language "skeleton" and/or a graphical representation (e.g., a flowchart) of design. These tools also produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.

A design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs, data declaration, and modularity characteristics.
- A free syntax of natural language that describes processing features.
- Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures.
- Subprogram definition and calling techniques that support various modes of interface description.

A basic PDL syntax should include constructs for subprogram definition, interface description, data declaration, techniques for block structuring, condition constructs, repetition constructs, and I/O constructs. The format and semantics for some of these POL constructs are presented in the section that follows.

It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, inter-process synchronization, and many other features. The application design, for which PDL is to be used should, dictates the final form for the design language.

7.3 Summary

Design is a process in which representations of data structure, program structure, interface characteristics, and procedural details are synthesized from information requirements. High Level Design activity includes Architectural Design, Interface Design and Data Design. The architecture of a system is a framework that describes its form and structure, its components and how they interact together. An architecture style involves its components, connectors, and constraints on combining components. Shaw and Garlan describe seven architectural styles (i) Pipes and Filters (ii) Call-and-return systems (iii) Object-oriented systems (iv) Layered systems (v) Data-centered systems (vi) Distributed systems. In the data flow oriented design, DFD representing the information flow is converted into the structure chart. The design of user interfaces involves three categories of Human Computer Interface (HCI) design guidelines (i) General interaction, (ii) Information display, (iii) Data entry. Procedural design occurs after data, architectural, and interface designs have been established. It is concerned with specifying algorithmic details, concrete data representations, interconnections among functions, and data structure. An important aspect here is structured programming emphasizing the use of single entry and single exit constructs. Using structured programming, facilitate the development of testable and maintainable code. To specify the algorithm during detailed design PDL is a good tool. PDL resembles a programming language that uses of narrative text (e.g., English) embedded directly within PDL statements. It is easy to translate a PDL code into an implementation using a programming language.

7.4 Keywords

Design It is a process in which representations of data structure, program structure, interface characteristics, and procedural details are synthesized from information requirements.

Architecture design: It defines the relationship between major structural elements of the software, the design patterns that can be used to achieve the requirements and the constraints that affect the implementation.

Structured programming: It is a technique to linearize control flow through a computer program by restricting the set of allowed program construct to single entry, single exit formats so that the execution sequence follows the sequence in which the code is written.

7.5 Self Assessment Questions

1. What do you understand by structured programming? What are the different rules of structured programming? Explain.
2. What are the different architectural styles? Give an overview of them.
3. Why should we not go directly from high level design to implementation? What is the advantage of having a detailed design in between i.e. writing the algorithms instead of directly writing the program using a high level language.
4. What are the desirable characteristics of human-comouter interface? Explain.
5. Discuss the procedure of converting a DFD representing information flow into a structured chart. Use suitable example.

7.6 References/Suggested readings

25. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
26. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
27. Software Engineering by Sommerville, Pearson Education.
28. Software Engineering – A Practitioner’s Approach by Roger S Pressman, McGraw-Hill.

Lesson No.: 8 **Writer: Dr. Rakesh Kumar**

Coding

Vetter: Dr. Yogesh Chaba

8.0 Objectives

The objective of this lesson is to make the students familier

1. With the concept of coding.
2. Programming Style
3. Verification and validations techniques.

8.1 Introduction

The coding is concerned with translating design specifications into source code. The good programming should ensure the ease of debugging, testing and modification. This is achieved by making the source code as clear and straightforward as possible. An old saying is “Simple is great”. Simplicity, clarity and elegance are the hallmarks of good programs. Obscurity, cleverness, and complexity are indications of inadequate design. Source code clarity is enhanced by structured coding techniques, by good coding style, by appropriate supporting documents, by good internal comments etc. Production of high quality software requires that the programming team should have a thorough understanding of duties and responsibilities and should be provided with a well defined set of requirements, an architectural design specification, and a detailed design description.

8.2 Presentation of contents

8.2.1 Programming style

8.2.1 Programming style

Programming style refers to the style used in writing the source code for a computer program. Most programming styles are designed to help programmers quickly read and understands the program as well as avoid making errors. (Older programming styles also focused on conserving screen space.) A good coding style can overcome the many deficiencies of a primitive programming language, while poor style can defeat the intent of en excellent language. The goal of good programming style is to provide understandable, straightforward, elegant code.

The programming style used in a particular program may be derived from the coding standards or code conventions of a company or other computing organization, as well as the preferences of the actual programmer. Programming styles are often designed for a specific programming language (or language family) and are not used in whole for other languages. (Style considered good in C source code may not be appropriate for BASIC source code, and so on.) Good style, being a subjective matter, is difficult to concretely categorize; however, there are several elements common to a large number of programming styles. Programming styles are often designed for a specific programming language and are not used in whole for other languages. So there is no single set of rules that can be applied in every situation; however there are general guidelines that are widely applicable. These are listed below:

8.2.1.1 Dos of good programming style

1. Use a few standards, agreed upon control constructs.
2. Use GOTO in a disciplined way.
3. Use user-defined data types to model entities in the problem domain.
4. Hide data structure behind access functions
5. Isolate machine dependencies in a few routines.
6. Use appropriate variable names
7. Use indentation, parentheses, blank spaces, and blank lines to enhance readability.

8.2.1.1.1 Use a few standard control constructs

There is no standard set of constructs for structured coding. For example to implement loops, a number of constructs are available such as repeat-until. While-do, for loop etc. If the implementation language does not provide structured coding constructs, a few stylistic patterns should be used by the programmers. This will make coding style more uniform with the result that programs will be easier to read, easier to understand, and easier to modify.

8.2.1.1.2 Use GOTO in a disciplined way

The best time to use GOTO statement is never. In all the modern programming languages, constructs are available which help you in avoiding the use of GOTO statement, so if you are a good programmer then you can avoid the use of GOTO statement. But if it is warranted then the acceptable uses of GOTO statements are almost always forward transfers of control within a local region of code. Don't use GOTO to achieve backward transfer of control.

8.2.1.1.3 Use user-defined data types to model entities in the problem domain

Use of distinct data types makes it possible for humans to distinguish between entities from the problem domain. All the modern programming languages provide the facilities of enumerated data type. For example, if an identifier is to be used to represent the month of a year, then instead of using integer data type to represent it, a better option can be an enumerated data type as illustrated below:

```
enum month = (jan, feb, march, april, may, june, july, aug, sep, oct, nov, dec);
month x;
```

Variables x is declared of month type. Using such types makes the program much understandable.

```
X = july;
```

is more meaningful than

```
x = 7;
```

8.2.1.1.4 Hide data structure behind access functions

It is the manifestation of the principle of information hiding. It is the approach taken in data encapsulation, wherein data structures and its accessing routines are encapsulated in a single module. So a module makes visible only those features that are required by other modules.

8.2.1.1.5 Appropriate variable names

Appropriate choices for variable names are seen as the keystone for good style. Poorly-named variables make code harder to read and understand. For example, consider the following pseudo code snippet:

```
get a b c
if a < 24 and b < 60 and c < 60
  return true
else
  return false
```

Because of the choice of variable names, the function of the code is difficult to work out. However, if the variable names are made more descriptive:

```
get hours minutes seconds
if hours < 24 and minutes < 60 and seconds < 60
  return true
else
  return false
```

the code's intent is easier to discern, namely, "Given a 24-hour time, true will be returned if it is a valid time and false otherwise."

A general guideline is “use the descriptive names suggesting the purpose of identifier”.

8.2.1.1.6 Use indentation, parentheses, blank spaces, and blank lines to enhance readability

Programming styles commonly deal with the appearance of source code, with the goal of improving the readability of the program. However, with the advent of software that formats source code automatically, the focus on appearance will likely yield to a greater focus on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without religious debates.

Indenting

Indent styles assist in identifying control flow and blocks of code. In programming languages that use indentation to delimit logical blocks of code, good indentation style directly affects the behavior of the resulting program. In other languages, such as those that use brackets to delimit code blocks, the indent style does not directly affect the product. Instead, using a logical and consistent indent style makes one's code more readable. Compare:

```
if (hours < 24 && minutes < 60 && seconds < 60){
   return true;
} else {
   return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
   return true;
}
else
{
   return false;
}
```

with something like

```
if (hours < 24 && minutes < 60 && seconds < 60) {return true;}
else {return false;}
```

The first two examples are much easier to read because they are indented well, and logical blocks of code are grouped and displayed together more clearly.

This example is somewhat contrived, of course - all the above are more complex (less stylish) than

```
return hours < 24 && minutes < 60 && seconds < 60;
```

Spacing

Free-format languages often completely ignore white space. Making good use of spacing in one's layout is therefore considered good programming style.

Compare the following examples of C code.

```
 int count;
 for(count=0;count<10;count++)
 {
   printf("%d",count*count+count);
 }
```

with

```
 int count;
 for( count = 0; count < 10; count++ )
 {
   printf( "%d", count * count + count);
 }
```

In the C-family languages, it is also recommended to avoid using tab characters in the middle of a line as different text editors render their width differently.

Python uses indentation to indicate control structures, so correct indentation is required. By doing this, the need for bracketing with curly braces ({ and }) is eliminated, and readability is improved while not interfering with common coding styles. However, this frequently leads to problems where code is copied and pasted into a Python program, requiring tedious reformatting. Additionally, Python code is rendered unusable when posted on a forum or webpage that removes white space.

8.2.1.1.7 Boolean values in decision structures

Some programmers think decision structures such as the above, where the result of the decision is merely computation of a Boolean value, are overly verbose and even prone to error. They prefer to have the decision in the computation itself, like this:

```
return hours < 12 && minutes < 60 && seconds < 60;
```

The difference is often purely stylistic and syntactic, as modern compilers produce identical object code for both forms.

8.2.1.1.8 Looping and control structures

The use of logical control structures for looping adds to good programming style as well. It helps someone reading code to understand the program's sequence of execution (in imperative programming languages). For example, in pseudocode:

```
 count = 0
 while count < 5
   print count * 2
   count = count + 1
 endwhile
```

The above snippet obeys the two aforementioned style guidelines, but the following use of the "for" construct makes the code much easier to read:

```
 for count = 0, count < 5, count=count+1
   print count * 2
```

In many languages, the often used "for each element in a range" pattern can be shortened to:

```
 for count = 0 to 5
   print count * 2
```

8.2.1.1.9 Examine routines having more than five formal parameters

Parameters are used to exachange the information among the functions or routines. Use of more than five formal parameters gives a feeling that probably the function is comples. So it is to be carefully examined. The choice of number five is not arbitrary. It is well known that human beings can deal with approximately seven items at one time and ease of understanding a subprogram call or the body of subprogram is a function of the number of parameters.

8.2.1.2 Don'ts of good programming style

1. Don't be too clever.
2. Avoid null Then statement
3. Avoid Then If statement
4. Don't nest too deeply.
5. Don't use an identifier for multiple purposes.
6. Examine routines having more than five formal parameters.

8.2.1.2.1 Don't be too clever

There is an old saying "Simple engineering is great engineering". We should try to keep our program simple. By making the use of tricks and showing cleverness, sometimes the complexity is increased. This can be illustrated using following example:

```
//Code to swap the values of two integer variables.
A=A+B;
B=A-B;
A=A-B;
```

You can observe the obscurity in the above code. The better approach can be:

```
T=A;

A=B;

B=T;
```

The second version to swap the values of two inegers is more clear and simple.

8.2.1.2.2 Avoid null then statement

A null then statement is of the form

```
If B then ; else S;
```

Which is equivalent to

```
If (not B) the S;
```

8.2.1.2.3 Avod then_If statement

A then_if statement is of the form

```
If(A>B) then
          if(X>Y) then
                      A=X
            Else
                      B=Y
            Endif
Else
          A=B
Endif
```

Then_if statement tend to obscure the conditions under which various actions are performed. It can be rewritten in the following form:

```
If(A<B) then

          A=B
Elseif (X>Y) then
          B=Y
Else
          A=X
endif
```

8.2.1.2.4 Don’t nest too deeply

Consider the following code

```
While X loop
 If Y then
  While Y loop
    While Z loop
      If W then S
```

In the above code, it is difficult to identify the conditions under which statement S will be executed. As a general guideline, nesting of program constructs to depths greater than three or four levels should be avoided.

8.2.1.2.5 Don't use an identifier for multiple purposes

Using an identifier for multiple purposes is a dangerous practice because it makes your program highly sensitive to future modification. Moreover the variable names should be descriptive suggesting their purposes to make the program understandable. This is not possible if the identifier is used for multiple purposes.

8.2.2 Software Verification and Validation

8.2.2.1 Concepts and Definitions

Software Verification and Validation (V&V) is the process of ensuring that software being developed or changed will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification). The differences between verification and validation (shown in table 8.1) are unimportant except to the theorist;

practitioners' use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required.

V&V is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the development process to ensure software requirements are complete and testable and that design, code, documentation, and data satisfy those requirements.

Table 8.1 Difference between verification and validation

Validation	**Verification**
Am I building the right product?	Am I building the product right?
Determining if the system complies with the requirements and performs functions for which it is intended and meets the organization's goals and user needs. It is traditional and is performed at the end of the project.	The review of interim work steps and interim deliverables during a project to ensure they are acceptable. To determine if the system is consistent, adheres to standards, uses reliable techniques and prudent practices, and performs the selected functions in the correct manner.
Am I accessing the right data (in terms of the data required to satisfy the requirement)	Am I accessing the data right (in the right place; in the right way).
High level activity	Low level activity
Performed after a work product is produced against established	Performed during development on key artifacts, like walkthroughs, reviews and

criteria ensuring that the product integrates correctly into the environment	inspections, mentor feedback, training, checklists and standards
Determination of correctness of the final software product by a development project with respect to the user needs and requirements	Demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle.

Activities

The two major V&V activities are reviews, including inspections and walkthroughs, and testing.

8.2.2.2 Reviews, Inspections, and Walkthroughs

Reviews are conducted during and at the end of each phase of the life cycle to determine whether established requirements, design concepts, and specifications have been met. Reviews consist of the presentation of material to a review board or panel. Reviews are most effective when conducted by personnel who have not been directly involved in the development of the software being reviewed.

Informal reviews are conducted on an as-needed basis. The developer chooses a review panel and provides and/or presents the material to be reviewed. The material may be as informal as a computer listing or hand-written documentation.

Formal reviews are conducted at the end of each life cycle phase. The acquirer of the software appoints the formal review panel or board, who may make or affect a go/no-go decision to proceed to the next step of the life cycle. Formal reviews include the Software Requirements Review, the Software Preliminary Design Review, the Software Critical Design Review, and the Software Test Readiness Review.

An inspection or walkthrough is a detailed examination of a product on a step-by-step or line-of-code by line-of-code basis. The purpose of conducting inspections and walkthroughs is to find errors. The group that does an inspection or walkthrough is composed of peers from development, test, and quality assurance.

8.2.2.3 Testing

Testing is the operation of the software with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between expected and actual results. There are varied levels of software tests, ranging from unit or element testing through integration testing and performance testing, up to software system and acceptance tests.

8.2.2.3.1 Informal Testing

Informal tests are done by the developer to measure the development progress. "Informal" in this case does not mean that the tests are done in a casual manner, just that the acquirer of the software is not formally involved, that witnessing of the testing is not required, and that the prime purpose of the tests is to find errors. Unit, component, and subsystem integration tests are usually informal tests.

Informal testing may be requirements-driven or design- driven. Requirements-driven or black box testing is done by selecting the input data and other parameters based on the software requirements and observing the outputs and reactions of the software. Black box testing can be done at any level of integration. In addition to testing for satisfaction of requirements, some of the objectives of requirements-driven testing are to ascertain:

- Computational correctness.
- Proper handling of boundary conditions, including extreme inputs and conditions that cause extreme outputs.
- State transitioning as expected.
- Proper behavior under stress or high load.
- Adequate error detection, handling, and recovery.

Design-driven or white box testing is the process where the tester examines the internal workings of code. Design-driven testing is done by selecting the input data and other parameters based on the internal logic paths that are to be checked. The goals of design-driven testing include ascertaining correctness of:

- All paths through the code. For most software products, this can be feasibly done only at the unit test level.
- Bit-by-bit functioning of interfaces.
- Size and timing of critical elements of code.

8.2.2.3.2 Formal Testing

Formal testing demonstrates that the software is ready for its intended use. A formal test should include an acquirer- approved test plan and procedures, quality assurance witnesses, a record of all discrepancies, and a test report.

Formal testing is always requirements-driven, and its purpose is to demonstrate that the software meets its requirements.

Each software development project should have at least one formal test, the acceptance test that concludes the development activities and demonstrates that the software is ready for operations.

In addition to the final acceptance test, other formal testing may be done on a project. For example, if the software is to be developed and delivered in increments or builds, there may be incremental acceptance tests. As a practical matter, any contractually required test is usually considered a formal test; others are "informal."

After acceptance of a software product, all changes to the product should be accepted as a result of a formal test. Post acceptance testing should include regression testing. Regression testing involves rerunning previously used acceptance tests to ensure that the change did not disturb functions that have previously been accepted.

8.2.2.4 Verification and Validation during the Software Acquisition Life Cycle

The V&V Plan should cover all V&V activities to be performed during all phases of the life cycle. The V&V Plan Data Item Description (DID) may be rolled out of the Product Assurance Plan DID contained in the SMAP Management Plan Documentation Standard and DID.

8.2.2.4.1 Software Concept and Initiation Phase

The major V&V activity during this phase is to develop a concept of how the system is to be reviewed and tested. Simple projects may compress the life cycle steps; if so, the reviews may have to be compressed. Test concepts may involve simple generation of test cases by a user representative or may require the development of elaborate simulators and test data generators. Without an adequate V&V concept and plan, the cost, schedule, and complexity of the project may be poorly estimated due to the lack of adequate test capabilities and data.

8.2.2.4.2 Software Requirements Phase

V&V activities during this phase should include:

- Analyzing software requirements to determine if they are consistent with, and within the scope of system requirements.
- Assuring that the requirements are testable and capable of being satisfied.
- Creating a preliminary version of the Acceptance Test Plan, including a verification matrix, which relates requirements to the tests used to demonstrate that requirements are satisfied.
- Beginning development, if needed, of test beds and test data generators.
- The phase-ending Software Requirements Review (SRR).

8.2.2.4.3 Software Architectural (Preliminary) Design Phase

V&V activities during this phase should include:

- Updating the preliminary version of the Acceptance Test Plan and the verification matrix.
- Conducting informal reviews and walkthroughs or inspections of the preliminary software and data base designs.
- The phase-ending Preliminary Design Review (PDR) at which the allocation of requirements to the software architecture is reviewed and approved.

8.2.2.4.4 Software Detailed Design Phase

V&V activities during this phase should include:

- Completing the Acceptance Test Plan and the verification matrix, including test specifications and unit test plans.
- Conducting informal reviews and walkthroughs or inspections of the detailed software and data base designs.
- The Critical Design Review (CDR), which completes the software, detailed design phase.

8.2.2.4.5 Software Implementation Phase

V&V activities during this phase should include:

- Code inspections and/or walkthroughs.
- Unit testing software and data structures.
- Locating, correcting, and retesting errors.
- Development of detailed test procedures for the next two phases.

8.2.2.4.6 Software Integration and Test Phase

This phase is a major V&V effort, where the tested units from the previous phase are integrated into subsystems and then the final system. Activities during this phase should include:

- Conducting tests per test procedures.
- Documenting test performance, test completion, and conformance of test results versus expected results.
- Providing a test report that includes a summary of non-conformances found during testing.
- Locating, recording, correcting, and retesting non-conformances.
- The Test Readiness Review (TRR), confirming the product's readiness for acceptance testing.

8.2.2.4.7 Software Acceptance and Delivery Phase

V&V activities during this phase should include:

- By test, analysis, and inspection, demonstrating that the developed system meets its functional, performance, and interface requirements.
- Locating, correcting, and retesting nonconformance.
- The phase-ending Acceptance Review (AR).

8.2.2.4.8 Software Sustaining Engineering and Operations Phase

Any V&V activities conducted during the prior seven phases are conducted during this phase as they pertain to the revision or update of the software.

8.2.2.5 Independent Verification and Validation

Independent Verification and Validation (IV&V) is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software. The IV&V agent should have no stake in the success or failure of the software. The IV&V agent's only interest should be to make sure that the software is thoroughly tested against its complete set of requirements.

The IV&V activities duplicate the V&V activities step-by-step during the life cycle, with the exception that the IV&V agent does no informal testing. If there is an IV&V agent, the formal acceptance testing may be done only once, by the IV&V agent. In this case, the developer will do a formal demonstration that the software is ready for formal acceptance.

8.2.2.6 Techniques and Tools

Perhaps more tools have been developed to aid the V&V of software (especially testing) than any other software activity. The tools available include code tracers, special purpose memory dumpers and formatters, data generators, simulations, and emulations. Some tools are essential for testing any significant set of software, and, if they have to be developed, may turn out to be a significant cost and schedule driver.

An especially useful technique for finding errors is the formal inspection. Michael Fagan of IBM developed formal inspections. Like walkthroughs, inspections involve the line-by-line evaluation of the product being reviewed. Inspections, however, are significantly different from walkthroughs and are significantly more effective. A team, each member of which has a specific role, does inspections. A moderator, who is formally trained in the inspection process, leads the team.

The team includes a reader, who leads the team through the item; one or more reviewers, who look for faults in the item; a recorder, who notes the faults; and the author, who helps explain the item being inspected.

This formal, highly structured inspection process has been extremely effective in finding and eliminating errors. It can be applied to any product of the software development process, including documents, design, and code. One of its important side benefits has been the direct feedback to the developer/author, and the significant improvement in quality that results.

8.3 Summary

A primary goal of software implementation is production of source code that is easy to read and understand. Clarity of source code ease debugging, testing and modification and these activities consume a large portion of most software budgets.

In this lesson, structured coding techniques, coding styles, standards and guidelines have been discussed. The essential aspect of structured programming is linearity of control flow. Linearity is assured by use of single entry and single exit program constructs.

Programming style is manifest in the patterns of choice made amomg alternative ways of expressing an algorithm. Consistent programming style among different programmers enhance project communication and eases debugging, testing and modification of the source code. Several guidelines for good programming style were presented. Dos and don’ts of good programming style were also illustrated.

8.4 Keywords

Coding: It is concerned with translating design specifications into source code.

Programming style: It refers to the style used in writing the source code for a computer program.

Software Validation: It is the process of ensuring that software being developed will satisfy functional and other requirements.

Verification: it is the process to ensure that each step in the process of building the software yields the right products.

Walkthrough: It is a detailed examination of a product on a step-by-step basis to find errors.

8.5 Self Assessment Questions

1. What do you understand by structured programming? Why should we do structured programming?
2. What constructs should be there in a structured programming language?
3. Why should we avoid the use of GOTO statement?
4. What are the advantages of structured prgramming?
5. What do you understand by programming style? What are the dos and don’ts of good programming style?
6. Define verification and validation. What are the differences between them? Explain.

8.6 References/Suggested readings

29. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
30. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.

31. Software Engineering by Sommerville, Pearson Education.

32. Software Engineering – A Practitioner's Approach by Roger S Pressman, McGraw-Hill.

Lesson number: 9 **Writer: Dr. Rakesh Kumar**

Software Testing - I **Vetter: Sh. Dinesh Kumar**

9.0 Objectives

The objective of this lesson is to make the students familiar with the concepts and activities carried out during testing phase. After studying this lesson the students will be familiar with following:

- Testing Fundamentals
- Test Cases and Test Criteria
- Psychology of testing
- Test Plan Activities During Testing
- Strategic Issues in Testing
- Unit testing
- Integration Testing
- Acceptance testing

9.1 Introduction

Until 1956 it was the debugging oriented period, where testing was often associated to debugging: there was no clear difference between testing and debugging. From 1957-1978 there was the demonstration oriented period where debugging and testing was distinguished now - in this period it was shown, that software satisfies the requirements. The time between 1979-1982 is announced as the destruction oriented period, where the goal was to find errors. 1983-1987 is classified as the evaluation oriented period: intention here is that during the software lifecycle a product evaluation is provided and measuring quality. From 1988 on it was seen as prevention oriented period where tests were to demonstrate that software satisfies its specification, to detect faults and to prevent faults.

Software testing is the process used to help identify the correctness, completeness, security, and quality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. This includes the process of executing a program or application with the intent of finding errors. Quality is not an absolute; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behaviour of the product against a specification. An important point is that software testing should be distinguished from the separate discipline of software quality assurance, which encompasses all business process areas, not just testing.

There are many approaches to software testing, but effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following routine procedure. One definition of testing is "the process of questioning a product in order to evaluate it", where the "questions" are operations the tester attempts to execute with the product, and the product answers with its behavior in reaction to the probing of the tester. Although most of the intellectual processes of testing are nearly identical to that of review or inspection, the word testing is connoted to mean the dynamic analysis of the product—putting the product through its paces. The quality of the application can, and normally does, vary widely from system to system but some of the common quality attributes include capability, reliability, efficiency, portability, maintainability, compatibility and usability. A good test is sometimes described as one which reveals an error; however, more recent thinking suggests that a good test is one which reveals information of interest to someone who matters within the project community.

9.2 Presentation of contents

9.2.1 Error, fault and failure

In general, software engineers distinguish software faults from software failures. In case of a failure, the software does not do what the user expects. A fault is a programming bug that may or may not actually manifest as a failure. A fault can also be described as an error in the correctness of the semantic of a computer program. A fault will become a failure if the exact computation conditions are met, one of them being that the faulty portion of computer software executes on the CPU. A fault can also turn into a failure when the software is ported to a different hardware platform or a different compiler, or when the software gets extended.

The term error is used to refer to the discrepancy between a computed, observed or measured value and the true, specified or theoretically correct value. Basically it refers to the difference between the actual output of a program and the correct output.

Fault is a condition that causes a system to fail in performing its required functions.

Failure is the inability of a system to perform a required function according to its specification. In case of a failure the observed behavior of the system is different from the specified behavior. Whenever there is a failure, there is a fault in the system but vice-versa may not be true. That is, sometimes there is a fault in the software but failure is not observed. Fault is just like an infection in the body. Whenever there is fever there is an infection, but sometimes body has infection but fever is not observed,

9.2.2 Software Testing Fundamentals

Software testing may be viewed as a sub-field of software quality assurance but typically exists independently (and there may be no SQA areas in some companies). In SQA, software process specialists and auditors take a broader view on software and its development. They examine and change the software engineering process itself to reduce the amount of faults that end up in the code or deliver faster.

Regardless of the methods used or level of formality involved the desired result of testing is a level of confidence in the software so that the developers are confident that the software has an acceptable defect rate. What constitutes an acceptable defect rate depends on the nature of the software.

A problem with software testing is that the number of defects in a software product can be very large, and the number of configurations of the product larger still. Bugs that occur infrequently are difficult to find in testing. A rule of thumb is that a system that is expected to function without faults for a certain length of time must have already been tested for at least that length of time. This has severe consequences for projects to write long-lived reliable software.

A common practice of software testing is that it is performed by an independent group of testers after the functionality is developed but before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Another common practice is for test suites to be developed during technical support escalation procedures. Such tests are then maintained in regression testing suites to ensure that future updates to the software don't repeat any of the known mistakes.

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

It is commonly believed that the earlier a defect is found the cheaper it is to fix it.

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process unit tests are written first, by the programmers. Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed.

Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process).

The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Testing is the process of finding the differences between the expected behavior specified by system models and the observed behavior of the system. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior.

9.2.3 A sample testing cycle

Although testing varies between organizations, there is a cycle to testing:

1. Requirements Analysis: Testing should begin in the requirements phase of the software development life cycle.
2. Design Analysis: During the design phase, testers work with developers in determining what aspects of a design are testable and under what parameter those tests work.
3. Test Planning: Test Strategy, Test Plan(s), Test Bed creation.
4. Test Development: Test Procedures, Test Scenarios, Test Cases, Test Scripts to use in testing software.
5. Test Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.

6. Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
7. Retesting the Defects

9.2.4 Testing Objectives

Glen Myres states a number of rules that can serves as testing objectives:

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

9.2.5 Testing principles

Davis suggested the following testing principles:

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.
- The Pareto principle applies to software testing. According to this principle 80 percent of all errors uncovered during testing will likely to be traceable to 20 percent of all program modules. The problem is to isolate these 20 percent modules and test them thoroughly.
- Testing should begin “in the small” and progress toward testing “in the large”.
- Exhaustive testing is not possible.

- To be most effective, testing should be conducted by an independent third party.

9.2.6 Psychology of Testing

“Testing cannot show the absence of defects, it can only show that software errors are present”. So devising a set of test cases that will guarantee that all errors will be detected is not feasible. Moreover, there are no formal or precise methods for selecting test cases. Even though, there are a number of heuristics and rules of thumb for deciding the test cases, selecting test cases is still a creative activity that relies on the ingenuity of the tester. Due to this reason, the psychology of the person performing the testing becomes important.

The aim of testing is often to demonstrate that a program works by showing that it has no errors. This is the opposite of what testing should be viewed as. The basic purpose of the testing phase is to detect the errors that may be present in the program. Hence, one should not start testing with the intent of showing that a program works; but the intent should be to show that a program does not work. With this in mind, we define testing as follows: testing is the process of executing a program with the intent of finding errors.

This emphasis on proper intent of testing is a trivial matter because test cases are designed by human beings, and human beings have a tendency to perform actions to achieve the goal they have in mind. So, if the goal is to demonstrate that a program works, we may consciously or subconsciously select test cases that will try to demonstrate that goal and that will beat the basic purpose of testing. On the other hand, if the intent is to show that the program does not work, we will challenge our intellect to find test cases towards that end, and we are likely to detect more errors. Testing is the one step in the software engineering process that could be viewed as destructive rather than constructive. In it the engineer creates a set of test cases that are intended to demolish the software. With this in mind, a test case is "good" if it detects an as-yet-undetected error in the program, and our goal during designing test cases should be to design such "good" test cases.

Due to these reasons, it is said that the creator of a program (i.e. programmer) should not be its tester because psychologically you cannot be destructive to your own creation. Many organizations require a product to be tested by people not involved with developing the program before finally delivering it to the customer. Another reason for independent testing is that sometimes errors occur because the programmer did not understand the specifications clearly. Testing of a program by its programmer will not detect such errors, whereas independent testing may succeed in finding them.

9.2.7 Test Levels

- Unit testing: It tests the minimal software item that can be tested. Each component is tested independently.
- Module testing: A module is a collection of dependent components. So it is component integration testing and it exposes defects in the interfaces and interaction between integrated components.

- Sub-system testing: It involves testing collection of modules which have been integrated into sub-systems. The sub-system test should concentrate on the detection of interface errors.
- System testing: System testing tests an integrated system to verify that it meets its requirements. It is concerned with validating that the system meets its functional and non-functional requirements.
- Acceptance testing: Acceptance testing allows the end-user or customer to decide whether or not to accept the product.

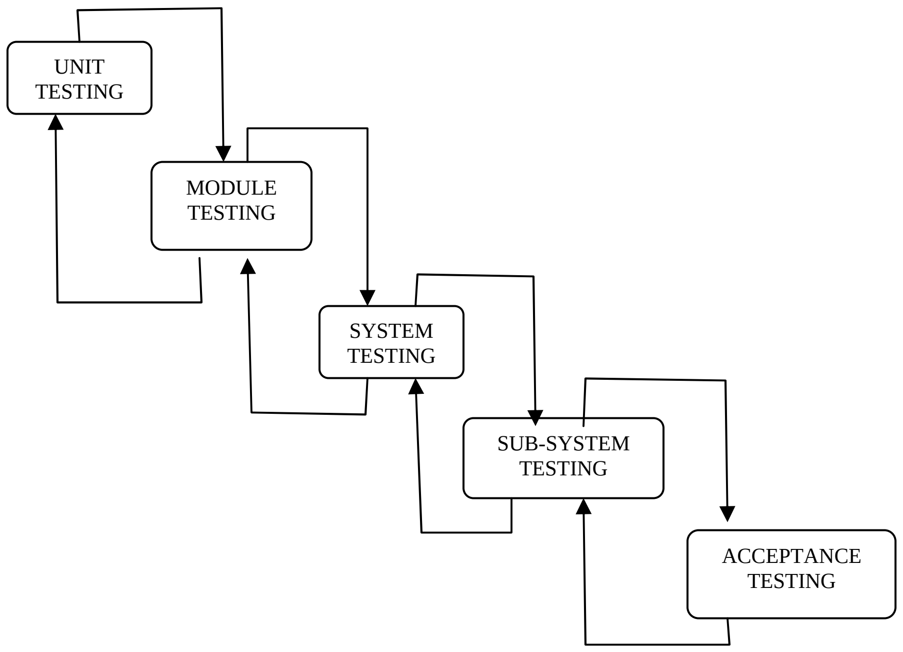


Figure 9.1 Test levels

9.2.8 SYSTEM TESTING

System testing involves two kinds of activities: integration testing and acceptance testing. Strategies for integrating software components into a functioning product include the bottom-up strategy, the top-down strategy, and the sandwich strategy. Careful planning and scheduling are required to ensure that modules will be available for integration into the evolving software product when needed. The integration strategy dictates the order in which modules must be available, and thus exerts a strong influence on the order in which modules are written, debugged, and unit tested.

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests to verify that the implemented system satisfies its requirements. Acceptance tests are typically performed by the quality assurance and/or customer organizations. Depending on local circumstances, the development group may or may not be involved in acceptance testing. Integration testing and acceptance testing are discussed in the following sections.

9.2.8.1 Integration Testing

Three are two important variants of integration testing, (a) Bottom-up integration and top-down integration, which are discussed in the following sections:

9.2.8.1.1 Bottom-up integration

Bottom-up integration is the traditional strategy used to integrate the components of a software system into a functioning whole. Bottom-up integration consists of unit testing, followed by subsystem testing, followed by testing of the entire system. Unit testing has the goal of discovering errors in the individual modules of the system. Modules are tested in isolation from one another in an artificial environment known as a "test harness," which consists of the driver programs and data necessary to exercise the modules. Unit testing should be as exhaustive as possible to ensure that each representative case handled by each module has been tested. Unit testing is eased by a system structure that is composed of small, loosely coupled modules. A subsystem consists of several modules that communicate with each other through well-defined interfaces. Normally, a subsystem implements a major segment of the total system. The primary purpose of subsystem testing is to verify the operation of the interfaces between modules in the subsystem. Both control and data interfaces must be tested. Large software may require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. In most software systems, exhaustive testing of subsystem capabilities is not feasible due to the combinational complexity of the module interfaces; therefore, test cases must be carefully chosen to exercise the interfaces in the desired manner.

System testing is concerned with subtleties in the interfaces, decision logic, control flow, recovery procedures, throughput; capacity, and timing characteristics of the entire system. Careful test planning is required to determine the extent and nature of system testing to be performed and to establish criteria by which the results will be evaluated.

Disadvantages of bottom-up testing include the necessity to write and debug test harnesses for the modules and subsystems, and the level of complexity that result from combining modules and subsystems into larger and larger units. The extreme case of complexity results when each module is unit tested in isolation and all modules are then linked and executed in one single integration run. This is the "big bang" approach to integration testing. The main problem with big-bang integration is the difficulty of isolating the sources of errors.

Test harnesses provide data environments and calling sequences for the routines and subsystems that are being tested in isolation. Test harness preparation can amount to 50 percent or more of the coding and debugging effort for a software product.

9.2.8.1.2 Top-down integration

Top-down integration starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level "skeleton" has been thoroughly tested, it becomes the test harness for its immediately subordinate routines. Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.

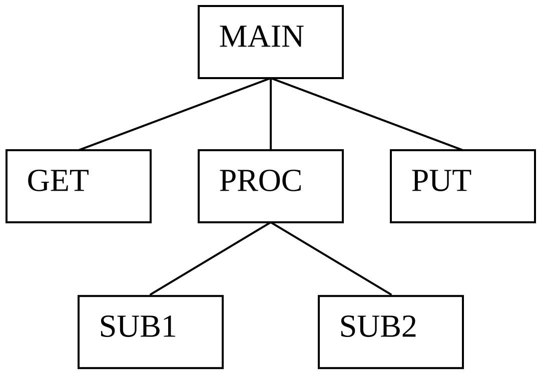


Figure 9.2

1. Test MAIN module, stubs for GET, PROC, and PUT are required.
2. Integrate GET module and now test MAIN and GET
3. Integrate PROC, stubs for SUBI, SUB2 are required.
4. Integrate PUT, Test MAIN, GET, PROC, PUT
5. Integrate SUB1 and test MAIN, GET, PROC, PUT, SUBI

6. Integrate SUB2 and test MAIN, GET, PROC, PUT, SUBI, SUB2

Above Figure 9.2 illustrates integrated top-down integration testing.

Top-down integration offers several advantages:

1. System integration is distributed throughout the implementation phase. Modules are integrated as they are developed.
2. Top-level interfaces are tested first and most often.
3. The top-level routines provide a natural test harness for lower-Level routines.
4. Errors are localized to the new modules and interfaces that are being added.

While it may appear that top-down integration is always preferable, there are many situations in which it is not possible to adhere to a strict top-down coding and integration strategy. For example, it may be difficult to find top-Level input data that will exercise a lower level module in a particular desired manner. Also, the evolving system may be very expensive to run as a test harness for new routines; it may not be cost effective to relink and re-execute a system of 50 or 100 routines each time a new routine is added. Significant amounts of machine time can often be saved by testing subsystems in isolation before inserting them into the evolving top-down structure. In some cases, it may not be possible to use program stubs to simulate modules below the current level (e.g. device drivers, interrupt handlers). It may be necessary to test certain critical low-level modules first.

The sandwich testing strategy may be preferred in these situations. Sandwich integration is predominately top-down, but bottom-up techniques are used on some modules and subsystems. This mix alleviates many of the problems encountered in pure top-down testing and retains the advantages of top-down integration at the subsystem and system level.

9.2.8.2 Regression testing

After modifying software, either for a change in functionality or to fix defects, a regression test re-runs previously passing tests on the modified software to ensure that the modifications haven't unintentionally caused a regression of previous functionality. Regression testing can be performed at any or all of the above test levels. These regression tests are often automated.

In integration testing also, each time a module is added, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. Hence, there is the need of regression testing.

Regression testing is any type of software testing which seeks to uncover regression bugs. Regression bugs occur whenever software functionality that previously worked as desired stops working or no longer works in the same way that was previously planned. Typically regression bugs occur as an unintended consequence of program changes.

Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have reemerged.

Experience has shown that as software is developed, this kind of reemergence of faults is quite common. Sometimes it occurs because a fix gets lost through poor revision control practices (or simple human error in revision control), but just as often a fix for a problem will be "fragile" - i.e. if some other change is made to the program, the fix no longer works. Finally, it has often been the case that when some feature is redesigned, the same mistakes will be made in the redesign that were made in the original implementation of the feature.

Therefore, in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools. Such a 'test suite' contains software tools that allows the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically re-run all regression tests at specified intervals and report any regressions. Common strategies are to run such a system after every successful compile (for small projects), every night, or once a week.

Regression testing is an integral part of the extreme programming software development methodology. In this methodology, design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage in the software development cycle.

Uses of regression testing

Regression testing can be used not only for testing the correctness of a program, but it is also often used to track the quality of its output. For instance in the design of a compiler, regression testing should track the code size, simulation time, and compilation time of the test suites.

System testing is a series of different tests and each test has a different purpose but all work to verify that all system elements have been properly integrated and perform allocated functions. In the following part a number of other system tests have been discussed.

9.2.8.3 Recovery testing

Many systems must recover from faults and resume processing within a specified time. Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

9.2.8.4 Stress testing

Stress tests are designed to confront programs with abnormal situations. Stress testing executes a program in a manner that demands resources in abnormal quantity, frequency, or volume. For example, a test case that may cause thrashing in a virtual operating system.

9.2.8.5 Performance Testing

For real time and embedded systems, performance testing is essential. In these systems, the compromise on performance is unacceptable. Performance testing is designed to test run-time performance of software within the context of an integrated system.

9.2.9 Acceptance testing

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests in order to demonstrate that the implemented system satisfies its requirements. Stress tests are performed to test the limitations of the systems. For example, a compiler may be tested to determine the effect of symbol table overflow.

Acceptance test will incorporate test cases developed during unit testing and integration testing. Additional test cases are added to achieve the desired level of functional, performance and stress testing of the entire system.

9.2.9.1 Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

9.2.9.2 Beta testing

Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the company. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

9.3 Summary

A high quality software product satisfies user needs, conforms to its requirements, and design specifications and exhibits an absence of errors. Techniques for improving software quality include systematic quality assurance procedures, walkthroughs, inspections, static analysis, unit testing integration testing, acceptance testing etc. Testing plays a critical role in quality assurance for software. Testing is a dynamic method for verification and validation. In it the system is executed and the behavior of the system is observed. Due, to this testing observes the failure of the system, from which the presence of faults can be deduced.

The goal of the testing is to detect the errors so there are different levels of testing. Unit testing focuses on the errors of a module while integration testing tests the system design. There are a number of approaches of integration testing with their own merits and demerits such as top-down integration, and bottom up integration. To goal of the acceptance testing is to test the system against the requirements. It comprises of alpha testing and beta testing.

The primary goal of verification and validation is to improve the quality of all the work products generated during software development and modification. Although testing is an important technique, but high quality cannot be achieved by it only. High quality is best achieved by careful attention to the details of planning, analysis, design, and implementation.

9.4 Key words

Fault: It is a programming bug that may or may not actually manifest as a failure.

Error: It is the discrepancy between a computed, observed or measured value and the true, specified or theoretically correct value.

Failure: It is the inability of a system to perform a required function according to its specification.

Unit testing: It tests the minimal software item that can be tested.

Regression testing: It is the re-running of previously passing tests on the modified software to ensure that the modifications haven't unintentionally caused a regression of previous functionality.

Acceptance testing: It is done to demonstrate that the implemented system satisfies its requirements.

Alpha testing: It is operational testing by a test team at the developers' site. It is a form of internal acceptance testing.

Beta testing: In this testing, the software is released to a limited audience outside of the company for further testing to ensure the product has few faults or bugs.

9.5 Self-Assessment Questions

1. Differentiate between

- ➢ Alpha testing and beta testing
- ➢ Top down integration and bottom up integration

2. Why the programmer of a program is not supposed to be its tester? Explain.
3. Does simply presence of fault mean software failure? If no, justify your answer with proper example.
4. What do you understand by regression testing and where do we use it?
5. Define testing. What characteristics are to be there in a good test case?
6. Explain why regression testing is necessary and how automated testing tools can assist with this type of testing.
7. Discuss whether it is possible for engineers to test their own programs in an objective way.
8. What do you understand by error, fault, and failure? Explain using suitable examples.

9.6 References/Suggested readings

33. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.

34. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing house.

35. Software Engineering by Sommerville, Pearson Education.

36. Software Engineering – A Practitioner’s Approach by Roger S Pressman, McGraw-Hill.

Lesson number: 10	**Writer: Dr. Rakesh Kumar**
Software Testing - II	**Vetter: Sh. Dinesh Kumar**

10.0 Objectives

The objective of this lesson is to make the students familiar with the process of test case design, to show them how program structure analysis can be used in the testing process. After studying this lesson the students will have the knowledge of test case design using functional and structural testing techniques.

10.1 Introduction

Testing of the software is most time and efforts consuming activity. On average 30 to 40 percent of total project efforts are consumed in testing. In some real time, embedded software figure may be higher. During the software development, errors may be introduced in any phase of the development. Because of the human inability to perform and communicate with perfection, software development is accompanied by quality assurance activity. Testing is a critical element of software quality assurance. In this chapter a number of strategies are discussed to design the test cases. According to Myres a good test is one that reveals the presence of defects in the software being tested. So a test suit does not detect defects, this means that the test chosen have not exercised the system so that defects are revealed. It does not mean that program defects do not exists.

10.2 Presentation of contents

10.2.1 Test case design

A test case is usually a single step, and its expected result, along with various additional pieces of information. It can occasionally be a series of steps but with one expected result or expected outcome. The optional fields are a test case ID, test step or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database or other common repository. In a database system, you may also be able to see past test results and who generated the results and the system configuration used to generate those results. These past results would usually be stored in a separate table.

The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Collections of test cases are sometimes incorrectly termed a test plan. They might correctly be called a test specification. If sequence is specified, it can be called a test script, scenario or procedure.

There are two basic approaches to test case design: functional (black box) and structural (white box). In functional testing, the structure of the program is not considered. Structural testing, on the other hand, is concerned with testing the implementation of the program.

10.2.2 White-box and black-box testing

White box and black box testing are terms used to describe the point of view a test engineer takes when designing test cases. Black box is an external view of the test object and white box, an internal view.

In recent years the term grey box testing has come into common usage. The typical grey box tester is permitted to set up or manipulate the testing environment, like seeding a database, and can view the state of the product after her actions, like performing a SQL query on the database to be certain of the values of columns. It is used almost exclusively of client-server testers or others who use a database as a repository of information, but can also apply to a tester who has to manipulate XML files (DTD or an actual XML file) or configuration files directly. It can also be used of testers who know the internal workings or algorithm of the software under test and can write tests specifically for the anticipated results. For example, testing a data warehouse implementation involves loading the target database with information, and verifying the correctness of data population and loading of data into the correct tables.

10.2.2.1 White box testing

White box testing (also known as clear box testing, glass box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise all paths and determines the appropriate outputs. In electrical hardware testing every node in a circuit may be probed and measured, an example is In circuit test (ICT).

Since the tests are based on the actual implementation, if the implementation changes, the tests probably will need to also. For example ICT needs updates if component values change, and needs modified/new fixture if the circuit changes. This adds financial resistance to the change process, thus buggy products may stay buggy. Automated optical inspection (AOI) offers similar component level correctness checking without the cost of ICT fixtures, however changes still require test updates.

While white box testing is applicable at the unit, integration and system levels, it's typically applied to the unit. So while it normally tests paths within a unit, it can also test paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover an overwhelming number of test cases, it might not detect unimplemented parts of the specification or missing requirements. But you can be sure that all paths through the test object are executed.

Typical white box test design techniques include:

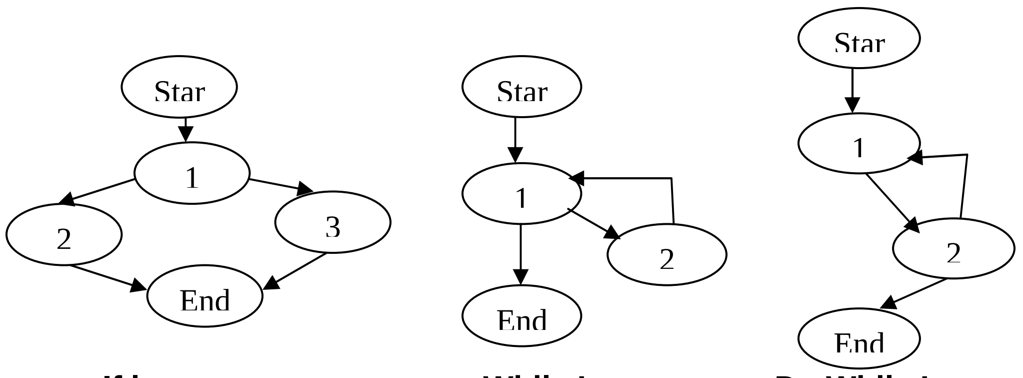
- Control flow testing
- Data flow testing

10.2.2.1.1 Code coverage

The most common structure based criteria are based on the control flow of the program. In this criterion, a control flow graph of the program is constructed and coverage of various aspects of the graph is specified as criteria. A control flow graph of program consists of nodes and edges. A node in the graph represents a block of statement that is always executed together. An edge frm node i to node j represents a possible transfer of control after executing the last statement in the block represented by node i to the first statement of the block represented by node j. Three common forms of code coverage used by testers are statement (or line) coverage, branch coverage, and path coverage. Line coverage reports on the execution footprint of testing in terms of which lines of code were executed to complete the test. According to this criterion each statement of the program to be tested should be executed at least once. Using branch coverage as the test criteria, the tester attempts to find a set of test cases that will execute each branching statement in each duirection at least once. A path coverage criterion acknowledges that the order in which the btanches are executed during a test (the path traversed) is an important factor in determining the test outcome. So tester attempts to find a set of test cases that ensure the traversal of each logical path in the control flow graph.

A Control Flow Graph (CFG) is a diagrammatic representation of a program and its execution. A CFG shows all the possible sequences of statements of a program. CFGs consist of all the typical building blocks of any flow diagrams. There is always a start node, an end node, and flows (or arcs) between nodes. Each node is labeled in order for it to be identified and associated correctly with its corresponding part in the program code.

CFGs allow for constructs to be nested in order to represent nested loops in the actual code. Some examples are given below in figure 10.1:



If loop **While Loop** **Do While Loop**

Figure 10.1

In programs where while loops exist, there are potentially an infinite number of unique paths through the program. Every path through a program has a set of associated conditions. Finding out what these conditions are allows for test data to be created. This enables the code to be tested to a suitable degree.

The conditions that exist for a path through a program are defined by the values of variable, which change through the execution of the code. At any point in the program execution, the program state is described by these variables. Statements in the code such as "x = x + 1" alter the state of the program by changing the value of a variable (in this case, x). Infeasible paths are those paths, which cannot be executed. Infeasible paths occur when no values will satisfy the path constraint.

Example:

```
//Program to find the largest of three numbers:
input a,b,c;
max=a;
if (b>max) max=b;
if(c=max) max=c;
output max;
```

The control flow graph of this program is given below in figure 10.2. In this flowgraph node 1 represents the statements [input a,b,c;max=a;if(b>max)], node 2 represents [max=b], node 3 represents [if(c>max)], node 4 represents [max=c] and node 5 represents [output max].

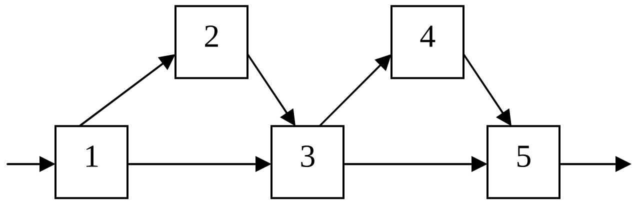


Figure 10.2

To ensure the Statement coverage [1, 2, 3, 4, 5] one test case a=5, b=10, and c=15 is sufficient.

To ensure Branch coverage [1, 3, 5] and [1, 2, 3, 4, 5], two test cases are required (i) a=5, b=10, c=15 and (ii) a=15, b=10, and c=5.

To ensure Path coverage ([1,2,3,4,5], [1,3,5], [1,2,3,5], and [1,3,4,5]), four test cases are required:

(i) a=5, b=10, c=15

(ii) a=15, b=10, and c=5.

(iii) a=5, b=10, and c=8

(iv) a=10, b=5, c=15

Path coverage criteria leads to a potentially infinite number of paths, some efforts have been made to limit the number of paths to be tested. One such approach is the cyclomatic complexity. The cyclomatic complexity of a path represents the logically independent path in a program as in the above case the cyclomatic complexity is three so three test cases are sufficient. As these are the independent paths, all other paths can be represented as a combination of these basic paths.

10.2.2.1.2 Data Flow testing

The data flow testing is based on the information about where the variables are defined and where the definitions are used. During testing the definitions of variables and their subsequent use is tested. Data flow testing looks at how data moves within a program. There are a number of associated test criteria and these should complement the control-flow criteria. Data flow occurs through assigning a value to a variable in one place accessing that a value in another place.

To illustrate the data flow based testing; let us assume that each statement in the program has been assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

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

USE(S) = { X | statement S contains a use of X}

If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X is said to be live at the statement S' if there exists a path from statement S to statement S' that does not contain any other definition of X. A Definition Use chain (DU chain) of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in the statement S is live at the statement S'.

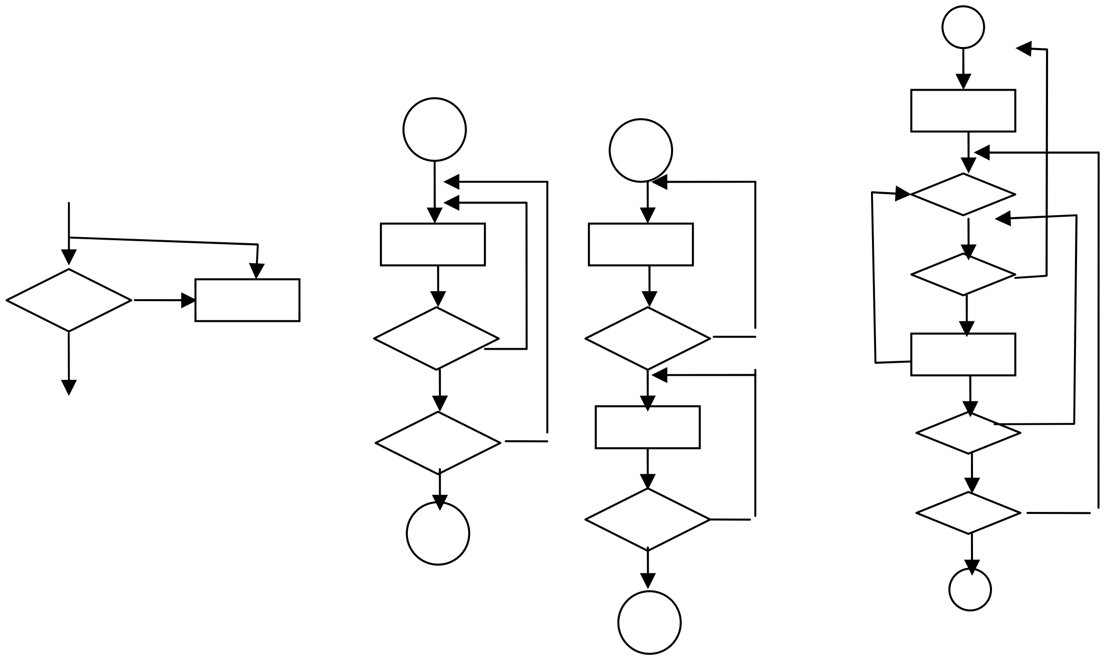
One simple data flow testing strategy is to require that every DU chain be covered at least once. This strategy is known as DU testing strategy.

10.2.2.1.3 Loop testing

Loops are very important constructs for generally all the algorithms. Loop testing is a white box testing technique. It focuses exclusively on the validity of loop constructs. Four different types of loops are: simple loop, concatenated loop, nested loop, and unstructured loop as shown in figure 10.3.

Simple loop: The following set of tests should be applied to simple loop where n is the maximum number of allowable passes thru the loop:

- Skip the loop entirely.
- Only one pass thru the loop.
- Two passes thru the loop.
- M passes thru the loop where m < n.
- N-1, n, n+1 passes thru the loop.



Simple loop Nested loop Concatenated loop Unstructured loop

Figure 10.3

Nested loop: Beizer approach to the nested loop is:

- Start at the innermost loop. Set all other loops to minimum value.
- Conduct the simple loop test for the innermost loop while holding the outer loops at their minimum iteration parameter value.

- Work outward, conducting tests for next loop, but keeping all other outer loops at minimum values and other nested loops to typical values.
- Continue until all loops have been tested.

Concatenated loops: These can be tested using the approach of simple loops if each loop is independent of other. However, if the loop counter of loop 1 is used as the initial value for loop 2 then approach of nested loop is to be used.

Unstructured loop: This class of loops should be redesigned to reflect the use of the structured programming constructs.

10.2.2.2 Black Box testing

Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.

This method of test design is applicable to all levels of development - unit, integration, system and acceptance. The higher the level, and hence the bigger and more complex the box, the more we are forced to use black box testing to simplify. While this method can uncover unimplemented parts of the specification, you can't be sure that all existent paths are tested. Some common approaches of black box testing are equivalence class partitioning, boundary value analysis etc.

10.2.2.2.1 Equivalence class partitioning

Equivalence partitioning is software testing related technique with the goal:

1. To reduce the number of test cases to a necessary minimum.

2. To select the right test cases to cover all possible scenarios.

Although in rare cases equivalence partitioning is also applied to outputs of a software component, typically it is applied to the inputs of a tested component. The equivalence partitions are usually derived from the specification of the component's behaviour. An input has certain ranges which are valid and other ranges which are invalid. This may be best explained at the following example of a function which has the pass parameter "month" of a date. The valid range for the month is 1 to 12, standing for January to December. This valid range is called a partition. In this example there are two further partitions of invalid ranges. The first invalid partition would be <= 0 and the second invalid partition would be >= 13.

-2, -1, 0	1,2,……..12	13, 14, 15
Invalid partition 1	Valid partition	Invalid partition 2

The testing theory related to equivalence partitioning says that only one test case of each partition is needed to evaluate the behaviour of the program for the related partition. In other words it is sufficient to select one test case out of each partition to check the behaviour of the program. To use more or even all test cases of a partition will not find new faults in the program. The values within one partition are considered to be "equivalent". Thus the number of test cases can be reduced considerably.

An additional effect by applying this technique is that you also find the so called "dirty" test cases. An inexperienced tester may be tempted to use as test cases the input data 1 to 12 for the month and forget to select some out of the invalid partitions. This would lead to a huge number of unnecessary test cases on the one hand, and a lack of test cases for the dirty ranges on the other hand.

The tendency is to relate equivalence partitioning to the so called black box testing which is strictly checking a software component at its interface, without consideration of internal structures of the software. But having a closer look on the subject there are cases where it applies to the white box testing as well. Imagine an interface to a component which has a valid range between 1 and 12 like in the example above. However internally the function may have a differentiation of values between 1 and 6 and the values between 7 and 12. Depending on the input value the software internally will run through different paths to perform slightly different actions. Regarding the input and output interfaces to the component this difference will not be noticed, however in your white-box testing you would like to make sure that both paths are examined. To achieve this it is necessary to introduce additional equivalence partitions which would not be needed for black-box testing. For this example this would be:

-2, -1, 0	1,…..6	7,……12	13, 14, 15
Invalid Partition 1	P1	P2	Invalid Partition 2
	Valid Partition		

To check for the expected results you would need to evaluate some internal intermediate values rather than the output interface.

Equivalence partitioning is no stand alone method to determine test cases. It has to be supplemented by boundary value analysis. Having determined the partitions of possible inputs the method of boundary value analysis has to be applied to select the most effective test cases out of these partitions.

10.2.2.2.2 Boundary value analysis

Boundary value analysis is software testing related technique to determine test cases covering known areas of frequent problems at the boundaries of software component input ranges. Testing experience has shown that especially the boundaries of input ranges to a software component are liable to defects. A programmer who has to implement e.g. the range 1 to 12 at an input, which e.g. stands for the month January to December in a date, has in his code a line checking for this range. This may look like:

```
  if (month > 0 && month < 13)
```

But a common programming error may check a wrong range e.g. starting the range at 0 by writing:

```
  if (month >= 0 && month < 13)
```

For more complex range checks in a program this may be a problem which is not so easily spotted as in the above simple example.

Applying boundary value analysis

To set up boundary value analysis test cases you first have to determine which boundaries you have at the interface of a software component. This has to be done by applying the equivalence partitioning technique. Boundary value analysis and equivalence partitioning are inevitably linked together. For the example of the month in a date you would have the following partitions:

-2, -1, 0	1,2,……..12	13, 14, 15

Invalid partition 1	Valid partition	Invalid partition 2

Applying boundary value analysis you have to select now a test case at each side of the boundary between two partitions. In the above example this would be 0 and 1 for the lower boundary as well as 12 and 13 for the upper boundary. Each of these pairs consists of a "clean" and a "dirty" test case. A "clean" test case should give you a valid operation result of your program. A "dirty" test case should lead to a correct and specified input error treatment such as the limiting of values, the usage of a substitute value, or in case of a program with a user interface, it has to lead to warning and request to enter correct data. The boundary value analysis can have 6 textcases.n,n-1,n+1 for the upper limit and n,n-1,n+1 for the lower limit.

A further set of boundaries has to be considered when you set up your test cases. A solid testing strategy also has to consider the natural boundaries of the data types used in the program. If you are working with signed values this is especially the range around zero (-1, 0, +1). Similar to the typical range check faults programmers tend to have weaknesses in their programs in this range. E.g. this could be a division by zero problems when a zero value is possible to occur although the programmer always thought the range starting at 1. It could be a sign problem when a value turns out to be negative in some rare cases, although the programmer always expected it to be positive. Even if this critical natural boundary is clearly within an equivalence partition it should lead to additional test cases checking the range around zero. A further natural boundary is the natural lower und upper limit of the data type itself. E.g. an unsigned 8-bit value has the range of 0 to 255. A good test strategy would also check how the program reacts at an input of -1 and 0 as well as 255 and 256.

The tendency is to relate boundary value analysis more to the so called black box testing which is strictly checking a software component at its interfaces, without consideration of internal structures of the software. But having a closer look on the subject there are cases where it applies also to white box testing.

After determining the necessary test cases with equivalence partitioning and the subsequent boundary value analysis it is necessary to define the combinations of the test cases in case of multiple inputs to a software component.

10.2.2.2.3 Cause-Effect Graphing

One weakness with the equivalence class partitioning and boundary value methods is that they consider each input separately. That is, both concentrate on the conditions and classes of one input. They do not consider combinations of input circumstances that may form interesting situations that should be tested. One way to exercise combinations of different input conditions is to consider all valid combinations of the equivalence classes of input conditions. This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are n different input conditions, such that any combination of the input conditions is valid, we will have 2^n test cases.

Cause-effect graphing is a technique that aids in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large. The technique starts with identifying causes and effects of the system under testing. A cause is a distinct input condition, and an effect is a distinct output condition. Each condition forms a node in the cause-effect graph. The conditions should be stated such that they can be set to either true or false. For example, an input condition can be "file is empty," which can be set to true by having an empty input file, and false by a nonempty file. After identifying the causes and effects, for each effect we identify the causes that can produce that effect and how the conditions have to be combined to make the effect true. Conditions are combined using the Boolean operators "and", "or", and "not", which are represented in the graph by Λ, V and zigzag line respectively. Then, for each effect, all combinations of the causes that the effect depends on which will make the effect true, are generated (the causes that the effect does not depend on are essentially "don't care"). By doing this, we identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effect true.

Let us illustrate this technique with a small example. Suppose that for a bank database there are two commands allowed:

```
credit  acct-number   transaction_amount
debit   acct-number   transaction_amount
```

The requirements are that if the command is credit and the acct-number is valid, then the account is credited. If the command is debit, the acct-number is valid, and the transaction_amount is valid (less than the balance), then the account is debited. If the command is not valid, the account number is not valid, or the debit amount is not valid, a suitable message is generated. We can identify the following causes and effects from these requirements:

Cause:

c_1. Command is credit

c_2. Command is debit

c_3. Account number is valid

c_4. Transaction_amt. is valid

Effects:

e_l. Print "invalid command"

e_2. Print "invalid account-number"

e_3. Print "Debit amount not valid"

e_4. Debit account

e_5. Credit account

The cause effect of this is shown in following Figure 10.4. In the graph, the cause-effect relationship of this example is captured. For all effects, one can easily determine the causes each effect depends on and the exact nature of the dependency. For example, according to this graph, the effect E_5 depends on the causes c_2, c_3, and c_4 in a manner such that the effect E_5 is enabled when all c_2, c_3, and c_4 are true. Similarly, the effect E_2 is enabled if c_3 is false.

From this graph, a list of test cases can be generated. The basic strategy is to set an effect to I and then set the causes that enable this condition. The condition of causes forms the test case. A cause may be set to false, true, or don't care (in the case when the effect does not depend at all on the cause). To do this for all the effects, it is convenient to use a decision table (Table 10.1). This table lists the combinations of conditions to set different effects. Each combination of conditions in the table for an effect is a test case. Together, these condition combinations check for various effects the software should display. For example, to test for the effect E_3, both c_2 and c_4 have to be set. That is, to test the effect "Print debit amount not valid," the test case should be: Command is debit (setting: c_2 to True), the account number is valid (setting c_3 to False), and the transaction money is not proper (setting c_4 to False).

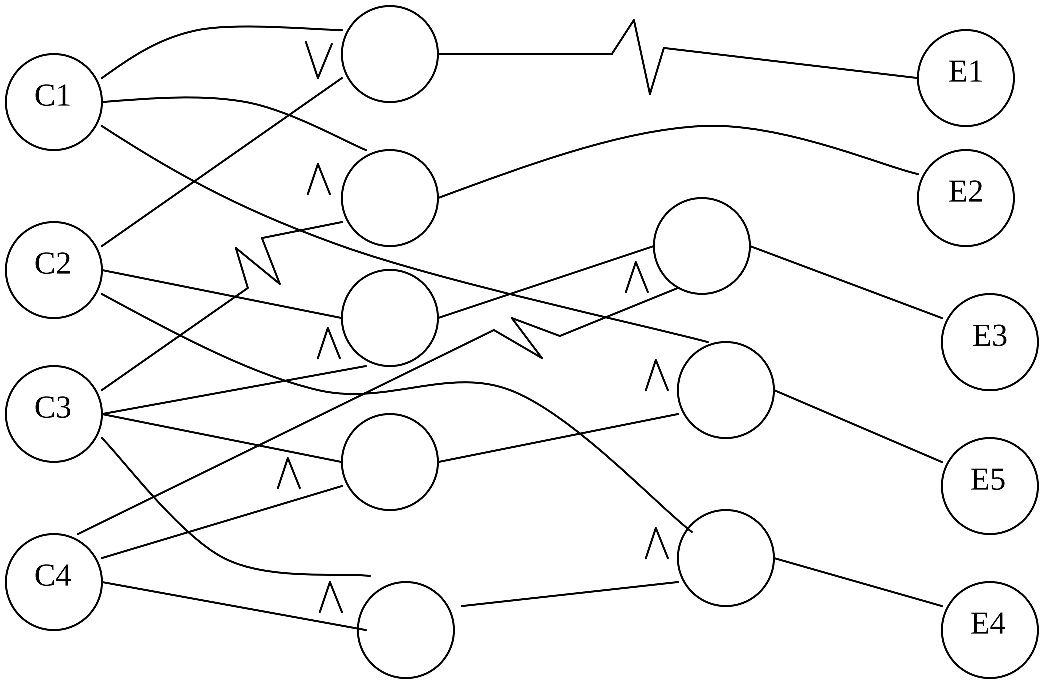


Figure 10.4 The Cause Effect Graph

SNo.	1	2	3	4	5
C_l	0	1	x	x	1
C_2	0	x	1	1	x
C_3	x	0	1	1	1

C_4	x	x	0	1	1
E_l	1				
E_2		1			
E_3			1		
E_4				1	
E_5					1

Table 10.1 Decision Table for the Cause-effect Graph

Cause-effect graphing, beyond generating high-yield test cases, also aids the understanding of the functionality of the system, because the tester must identify the distinct causes and effects. There are methods of reducing the number of test cases generated by proper traversing of the graph. Once the causes and effects are listed and their dependencies specified, much of the remaining work can also be automated.

10.2.3 Black box and white box testing compared

White box testing is concerned only with testing the software product; it cannot guarantee that the complete specification has been implemented. Black box testing is concerned only with testing the specification; it cannot guarantee that all parts of the implementation have been tested. Thus black box testing is testing against the specification and will discover faults of omission, indicating that part of the specification has not been fulfilled. White box testing is testing against the implementation and will discover faults of commission, indicating that part of the implementation is faulty. In order to fully test a software product both black and white box testing are required.

White box testing is much more expensive than black box testing. It requires the source code to be produced before the tests can be planned and is much more laborious in the determination of suitable input data and the determination if the software is or is not correct. The advice given is to start test planning with a black box test approach as soon as the specification is available. White box planning should commence as soon as all black box tests have been successfully passed, with the production of flow graphs and determination of paths. The paths should then be checked against the black box test plan and any additional required test runs determined and applied.

The consequences of test failure at this stage may be very expensive. A failure of a white box test may result in a change which requires all black box testing to be repeated and the re-determination of the white box paths. The cheaper option is to regard the process of testing as one of quality assurance rather than quality control. The intention is that sufficient quality will be put into all previous design and production stages so that it can be expected that testing will confirm that there are very few faults present, quality assurance, rather than testing being relied upon to discover any faults in the software, quality control. A combination of black box and white box test considerations is still not a completely adequate test rationale; additional considerations are to be introduced.

10.2.4 Mutation Testing

Mutation testing is another white box testing technique. Mutation testing is a fault-based testing technique that is based on the assumption that a program is well tested if all simple faults are predicted and removed; complex faults are coupled with simple faults and are thus detected by tests that detect simple faults.

Mutation testing is used to test the quality of your test suite. This is done by mutating certain statements in your source code and checking if your test code is able to find the errors. However, mutation testing is very expensive to run, especially on very large applications. There is a mutation testing tool, Jester, which can be used to run mutation tests on Java code. Jester looks at specific areas of your source code, for example: forcing a path through an if statement, changing constant values, and changing Boolean values

The idea of mutation testing was introduced as an attempt to solve the problem of not being able to measure the accuracy of test suites. The thinking goes as follows: Let’s assume that we have a perfect test suite, one that covers all possible cases. Let’s also assume that we have a perfect program that passes this test suite. If we change the code of the program (this process is called mutation) and we run the mutated program (mutant) against the test suite, we will have two possible scenarios:

1. The results of the program were affected by the code change and the test suite detects it. If this happens, the mutant is called a killed mutant.
2. The results of the program are not changed and the test suite does not detect the mutation. The mutant is called an equivalent mutant.

The ratio of killed mutants to the total mutants created measures how sensitive the program is to the code changes and how accurate the test suite is.

The effectiveness of Mutation Testing depends heavily on the types of faults that the mutation operators are designed to represent. By mutation operators we mean certain aspects of the programming techniques, the slightest change in which may cause the program to function incorrectly. For example, a simple condition check in the following code may perform viciously because of a little change in the code.

1. Original Code: for (x==1) { ….}
2. Mutated Code: for (x<=1) { ….}

In the example above the 'equality condition' was considered as a mutation operator and a little change in the condition is brought into effect as shown in the mutated code and the program is tested for its functionality.

Mutation Operators

Following could be some of the Mutation Operators for Object-Oriented languages like Java, C++ etc.

- Changing the access modifiers, like public to private etc.
- Static modifier change.
- Argument order change.
- Super keyword deletion.

Summary

A high quality software product satisfies user needs, conforms to its requirements, and design specifications and exhibits an absence of errors. Techniques for improving software quality include systematic quality assurance procedures, walkthroughs, inspections, static analysis, unit testing integration testing, acceptance testing etc. Testing plays a critical role in quality assurance for software. Testing is a dynamic method for verification and validation. In it the system is executed and the behavior of the system is observed. Due, to this testing observes the failure of the system, from which the presence of faults can be deduced.

There are two basic approaches to testing: white box testing and black box testing. In white box testing the structure of the program i.e. internal logic is considered to decide the test cases while in black box testing the examples of which are boundary value analysis, equivalence partitioning, the test cases are deduced on the basis of external specification of the program. So the functionality of the program is tested.

The goal of the testing is to detect the errors so there are different levels of testing. Unit testing focuses on the errors of a module while integration testing tests the system design. To goal of the acceptance testing is to test the system against the requirements.

The primary goal of verification and validation is to improve the quality of all the work products generated during software development and modification. Although testing is an important technique, but high quality cannot be achieved by it only. High quality is best achieved by careful attention to the details of planning, analysis, design, and implementation.

Key words

White box testing: It uses an internal perspective of the system to design test cases based on internal structure.

Mutation testing: It is used to test the quality of your test suite by mutating certain statements in your source code and checking if your test code is able to find the errors.

Boundary value analysis It is a technique to determine test cases covering known areas of frequent problems at the boundaries of software component input ranges.

Black box: It takes an external perspective of the test object to derive test cases.

Equivalence partitioning: It is a technique with the goal to reduce the number of test cases to a necessary minimum and to select the right test cases to cover all possible scenarios. By dividing the input ranges into equivalent partition and then selecting one representative from each partition.

Self-Assessment Questions

9. Differentiate between

➢ Black box testing and white box testing.

➢ Alpha testing and beta testing

➢ Top down integration and bottom up integration

➢ Control flow based and data flow based testing

10. What is boundary value analysis? Explain using suitable examples.

11. What is equivalence class partitioning? What are the advantages of using this testing technique?

12. What do you understand by structural testing? Using an example show that path coverage criteria is stronger than statement coverage and branch coverage criteria.

13. Write a module to compute the factorial of a given integer N. Design the test cases using boundary value analysis and equivalence class partitioning technique.
14. Why the programmer of a program is not supposed to be its tester? Explain.
15. What types of errors are detected by boundary value analysis and equivalence class partitioning techniques? Explain using suitable examples.
16. Using suitable examples show that:
    - Path coverage is a stronger criterion than branch coverage.
    - Branch coverage is a stronger criterion than statement coverage.
17. Does simply presence of fault mean software failure? If no, justify your answer with proper example.
18. What do you understand by regression testing and where do we use it?
19. Define testing. What characteristics are to be there in a good test case?
20. What do you understand by loop testing? Write a program for bubble sort and design the test cases using the loop testing criteria.
21. What is cyclomatic complexity? How does cyclomatic complexity number help in testing? Explain using suitable examples.

References/Suggested readings

37. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
38. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
39. Software Engineering by Sommerville, Pearson Education.

40. Software Engineering – A Practitioner's Approach by Roger S Pressman, McGraw-Hill.

Lesson no. : 11	**Writer: Dr. Rakesh Kumar**
Software Reliability	**Vetter: Dr. PK Bhatia**

11.0 Objectives

The objectives of this lesson are:

1. To introduce the concepts of reliability.
2. To discuss the metrics to measure software reliability.
3. To discuss the approaches to make software fault tolerant.
4. To make the students acquainted with the reliability growth modeling.

11.1 Introduction

With the advent of the computer age, computers, as well as the software running on them, are playing a vital role in our daily lives. We may not have noticed, but appliances such as washing machines, telephones, TVs, and watches, are having their analog and mechanical parts replaced by CPUs and software. The computer industry is booming exponentially. With a continuously lowering cost and improved control, processors and software-controlled systems offer compact design, flexible handling, rich features and competitive cost. Like machinery-replaced craftsmanship in the industrial revolution, computers and intelligent parts are quickly pushing their mechanical counterparts out of the market.

People used to believe that "software never breaks". Intuitively, unlike mechanical parts such as bolts, levers, or electronic parts such as transistors, capacitor, software will stay "as is" unless there are problems in hardware that changes the storage content or data path. Software does not age, rust, wear-out, deform or crack. There is no environmental constraint for software to operate as long as the hardware processor it runs on can operate. Furthermore, software has no shape, color, material, and mass. It cannot be seen or touched, but it has a physical existence and is crucial to system functionality.

Without being proven to be wrong, optimistic people would think that once after the software can run correctly, it will be correct forever. A series of tragedies and chaos caused by software proves this to be wrong. These events will always have their place in history.

Tragedies in Therac 25, a computer-controlled radiation-therapy machine in the year 1986, caused by the software not being able to detect a race condition, alerts us that it is dangerous to abandon our old but well-understood mechanical safety control and surrender our lives completely to software controlled safety mechanism.

Software can make decisions, but can just as unreliable as human beings. The British destroyer Sheffield was sunk because the radar system identified an incoming missile as "friendly". The defense system has matured to the point that it will not mistaken the rising moon for incoming missiles, but gas-field fire, descending space junk, etc, were also examples that can be misidentified as incoming missiles by the defense system.

Software can also have small unnoticeable errors or drifts that can culminate into a disaster. On February 25, 1991, during the Golf War, the chopping error that missed 0.000000095 second in precision in every 10th of a second, accumulating for 100 hours, made the Patriot missile fail to intercept a scud missile. 28 lives were lost.

Fixing problems may not necessarily make the software more reliable. On the contrary, new serious problems may arise. In 1991, after changing three lines of code in a signaling program which contains millions lines of code, the local telephone systems in California and along the Eastern seaboard came to a stop.

Once perfectly working software may also break if the running environment changes. After the success of Ariane 4 rocket, the maiden flight of Ariane 5 ended up in flames while design defects in the control software were unveiled by faster horizontal drifting speed of the new rocket.

There are much more scary stories to tell. This makes us wondering whether software is reliable at all, whether we should use software in safety-critical embedded applications. You can hardly ruin your clothes if the embedded software in your washing machine issues erroneous commands; and 50% of the chances you will be happy if the ATM machine miscalculates your money; but in airplanes, heart pace-makers, radiation therapy machines, a software error can easily claim people's lives. With processors and software permeating safety critical embedded world, the reliability of software is simply a matter of life and death.

11.2 Presentation of contents

11.2.1 Definition

11.2.2 Software failure mechanisms

11.2.3 The bathtub curve for Software Reliability

11.2.1 Definition

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware

Reliability, Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded.

Software Reliability is an important attribute of software quality, together with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. For example, large next-generation aircraft will have over one million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming international Space Station will have over two million lines on-board and over ten million lines of ground support software; several major life-critical defense systems will have over five million source lines of software. While the complexity of software is inversely related to software reliability, it is directly related to other important factors in software quality, especially functionality, capability, etc. Emphasizing these features will tend to add more complexity to software.

11.2.2 Software failure mechanisms

The software failure may be classified as:

- Transient failure: These failures only occur with certain inputs.

- Permanent failure: this failure occurs on all inputs.
- Recoverable failure: System can recover without operator help.
- Unrecoverable failure: System can recover with operator help only.
- Non-corruption failure: Failure does not corrupt system state or data.
- Corrupting failure: It corrupts system state or data.

Software failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems. While it is tempting to draw an analogy between Software Reliability and Hardware Reliability, software and hardware have basic differences that make them different in failure mechanisms. Hardware faults are mostly physical faults, while software faults are design faults, which are harder to visualize, classify, detect, and correct. Design faults are closely related to fuzzy human factors and the design process, which we don't have a solid understanding. In hardware, design faults may also exist, but physical faults usually dominate. In software, we can hardly find a strict corresponding counterpart for "manufacturing" as hardware manufacturing process, if the simple action of uploading software modules into place does not count. Therefore, the quality of software will not change once it is uploaded into the storage and start running. Trying to achieve higher reliability by simply duplicating the same software modules will not work, because voting cannot mask off design faults.

A partial list of the distinct characteristics of software compared to hardware is listed below:

- **Failure cause**: Software defects are mainly design defects.
- **Wear-out**: Software does not have energy related wear-out phase. Errors can occur without warning.
- **Repairable system concept**: Periodic restarts can help fix software problems.
- **Time dependency and life cycle**: Software reliability is not a function of operational time.
- **Environmental factors**: Do not affect Software reliability, except it might affect program inputs.
- **Reliability prediction**: Software reliability cannot be predicted from any physical basis, since it depends completely on human factors in design.
- **Redundancy**: Cannot improve Software reliability if identical software components are used.
- **Interfaces**: Software interfaces are purely conceptual other than visual.
- **Failure rate motivators**: Usually not predictable from analyses of separate statements.
- **Built with standard components**: Well-understood and extensively tested standard parts will help improve maintainability and reliability. But in software industry, we have not observed this trend. Code reuse has been around for some time, but to a very limited extent. Strictly speaking there are no standard parts for software, except some standardized logic structures.

11.2.3 The bathtub curve for Software Reliability

Over time, hardware exhibits the failure characteristics shown in following Figure 11.1, known as the bathtub curve. Period A, B and C stand for burn-in phase, useful life phase and end-of-life phase respectively.

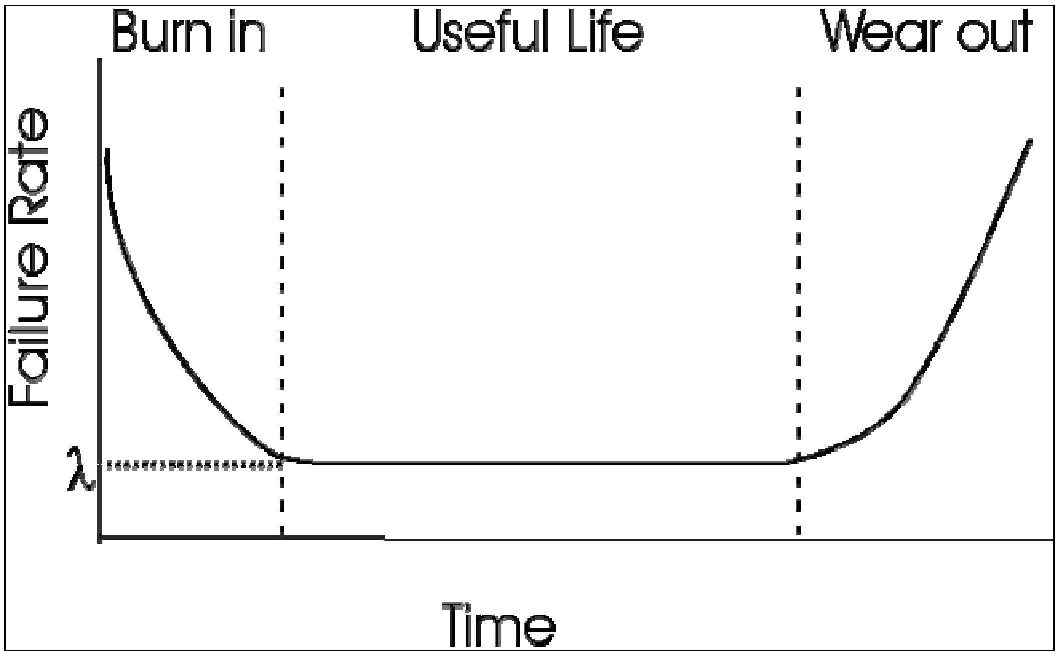


Figure 11.1 Bathtub curve for hardware reliability

Software reliability, however, does not show the same characteristics similar as hardware. A possible curve is shown in following Figure 11.2 if we projected software reliability on the same axes. There are two major differences between hardware and software curves. One difference is that in the last phase, software does not have an increasing failure rate as hardware does. In this phase, software is approaching obsolescence; there are no motivations for any upgrades or changes to the software. Therefore, the failure rate will not change. The second difference is that in the useful-life phase, software will experience a drastic increase in failure rate each time an upgrade is made. The failure rate levels off gradually, partly because of the defects found and fixed after the upgrades.

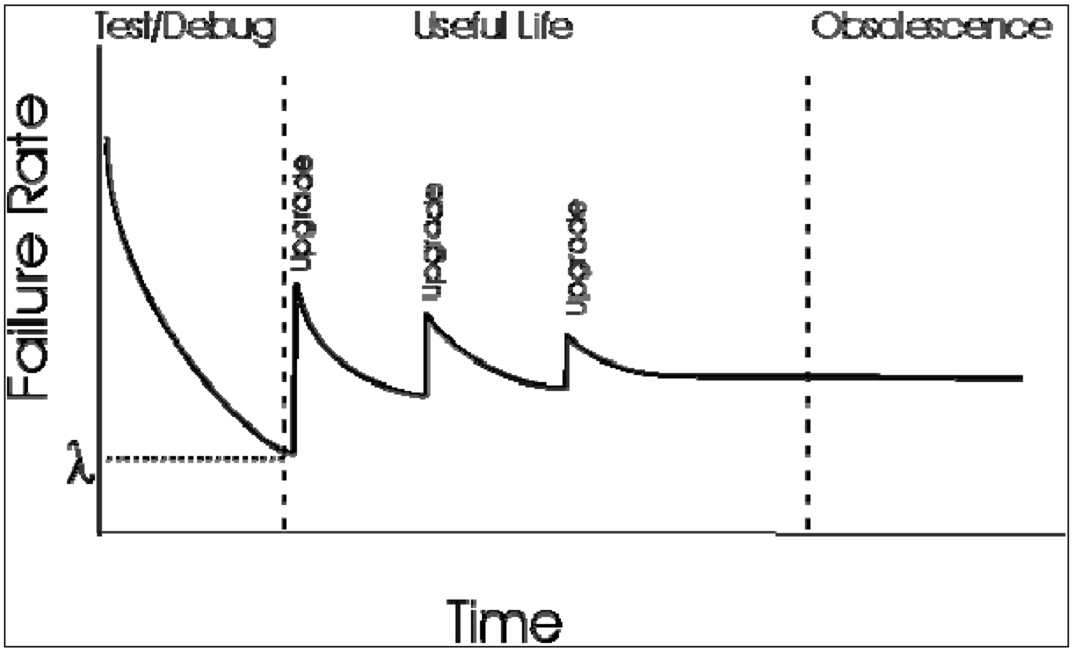


Figure 11.2 Revised bathtub curve for software reliability

The upgrades in above Figure imply feature upgrades, not upgrades for reliability. For feature upgrades, the complexity of software is likely to be increased, since the functionality of software is enhanced. Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software. For reliability upgrades, it is possible to incur a drop in software failure rate, if the goal of the upgrade is enhancing software reliability, such as a redesign or reimplementation of some modules using better engineering approaches, such as clean-room method.

Following Figure shows the testing results of fifteen POSIX compliant operating systems. From the graph we see that for QNX and HP-UX, robustness failure rate increases after the upgrade. But for SunOS, IRIX and Digital UNIX, robustness failure rate drops when the version numbers go up. Since software robustness is one aspect of software reliability, this result indicates that the upgrade of those systems shown in following Figure 11.3 should have incorporated reliability upgrades.

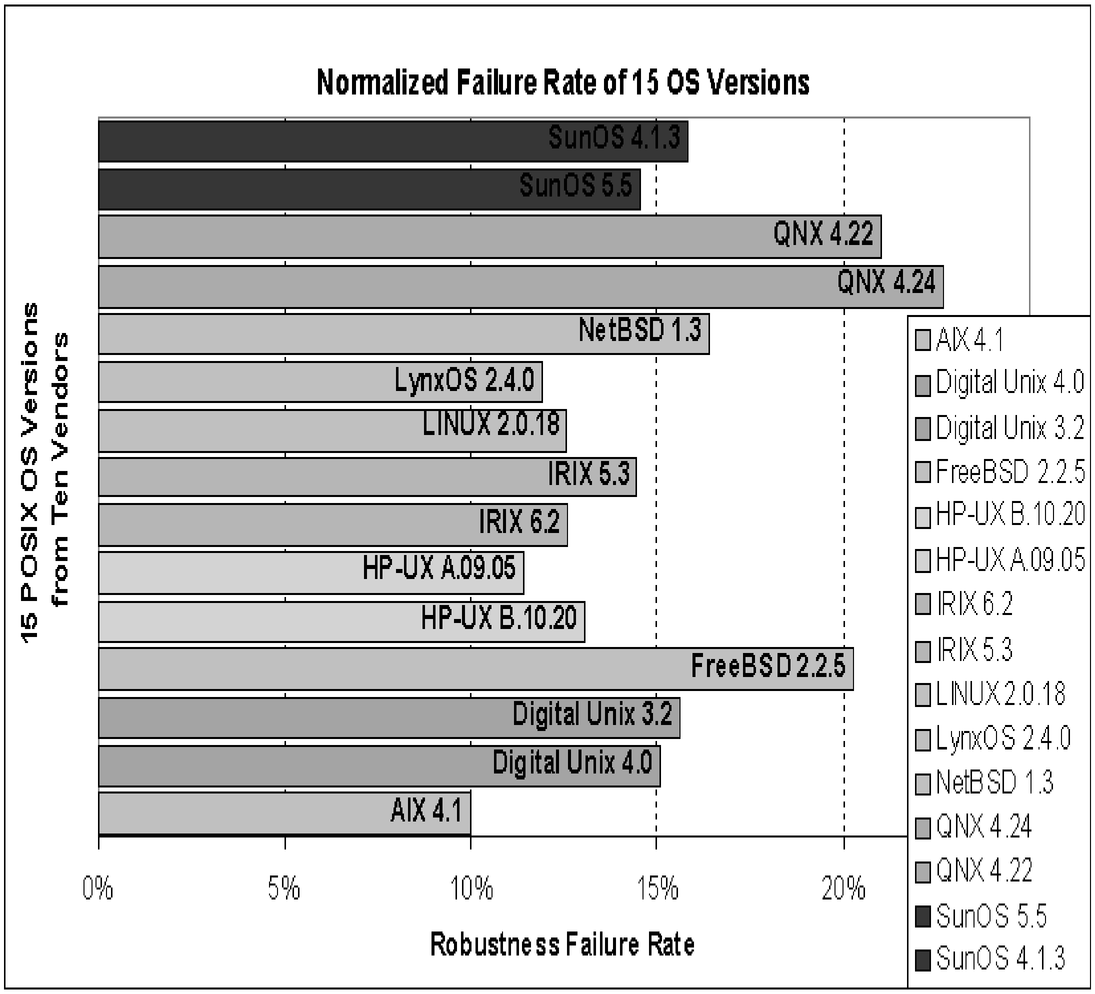


Figure 11.3

11.2.4 Available tools, techniques, and metrics

Since Software Reliability is one of the most important aspects of software quality, Reliability Engineering approaches are practiced in software field as well. Software Reliability Engineering (SRE) is the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability.

11.2.5 Software Reliability Models

A proliferation of software reliability models have emerged as people try to understand the characteristics of how and why software fails, and try to quantify software reliability. Over 200 models have been developed since the early 1970s, but how to quantify software reliability still remains largely unsolved. As many models as there are and many more emerging, none of the models can capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems.

Most software models contain the following parts: assumptions, factors, and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic.

Software modeling techniques can be divided into two subcategories: prediction modeling and estimation modeling. Both kinds of modeling techniques are based on observing and accumulating failure data and analyzing with statistical inference. The major differences of the two models are shown in following Table 11.1.

ISSUES	PREDICTION MODELS	ESTIMATION MODELS

DATA REFERENCE	Uses historical data	Uses data from the current software development effort
WHEN USED IN DEVELOPMENT CYCLE	Usually made prior to development or test phases; can be used as early as concept phase	Usually made later in life cycle (after some data have been collected); not typically used in concept or development phases
TIME FRAME	Predict reliability at some future time	Estimate reliability at either present or some future time

Table 11.1 Difference between software reliability prediction models and software reliability estimation models

Representative prediction models include Musa's Execution Time Model, Putnam's Model, and Rome Laboratory models TR-92-51 and TR-92-15, etc. Using prediction models, software reliability can be predicted early in the development phase and enhancements can be initiated to improve the reliability.

Representative estimation models include exponential distribution models, Weibull distribution model, Thompson and Chelson's model, etc. Exponential models and Weibull distribution model are usually named as classical fault count/fault rate estimation models, while Thompson and Chelson's model belong to Bayesian fault rate estimation models.

The field has matured to the point that software models can be applied in practical situations and give meaningful results and, second, that there is no one model that is best in all situations. Because of the complexity of software, any model has to have extra assumptions. Only limited factors can be put into consideration. Most software reliability models ignore the software development process and focus on the results -- the observed faults and/or failures. By doing so, complexity is reduced and abstraction is achieved, however, the models tend to specialize to be applied to only a portion of the situations and a certain class of the problems. We have to carefully choose the right model that suits our specific case. Furthermore, the modeling results cannot be blindly believed and applied.

11.2.6 Software Reliability Metrics

Measurement is commonplace in other engineering field, but not in software engineering. Though frustrating, the quest of quantifying software reliability has never ceased. Until now, we still have no good way of measuring software reliability.

Measuring software reliability remains a difficult problem because we don't have a good understanding of the nature of software. There is no clear definition to what aspects are related to software reliability. We cannot find a suitable way to measure software reliability, and most of the aspects related to software reliability. Even the most obvious product metrics such as software size have not uniform definition.

It is tempting to measure something related to reliability to reflect the characteristics, if we cannot measure reliability directly. The current practices of software reliability measurement can be divided into four categories:

11.2.6.1 Product metrics

Software size is thought to be reflective of complexity, development effort and reliability. Lines Of Code (LOC), or LOC in thousands (KLOC), is an intuitive initial approach to measuring software size. But there is not a standard way of counting. Typically, source code is used (SLOC, KSLOC) and comments and other non-executable statements are not counted. This method cannot faithfully compare software not written in the same language. The advent of new technologies of code reuse and code generation technique also cast doubt on this simple method.

Function point metric is a method of measuring the functionality of a proposed software development based upon a count of inputs, outputs, master files, inquires, and interfaces. The method can be used to estimate the size of a software system as soon as these functions can be identified. It is a measure of the functional complexity of the program. It measures the functionality delivered to the user and is independent of the programming language. It is used primarily for business systems; it is not proven in scientific or real-time applications.

Complexity is directly related to software reliability, so representing complexity is important. Complexity-oriented metrics is a method of determining the complexity of a program's control structure; by simplify the code into a graphical representation. Representative metric is McCabe's Complexity Metric.

Test coverage metrics are a way of estimating fault and reliability by performing tests on software products, based on the assumption that software reliability is a function of the portion of software that has been successfully verified or tested.

11.2.6.2 Project management metrics

Researchers have realized that good management can result in better products. Research has demonstrated that a relationship exists between the development process and the ability to complete projects on time and within the desired quality objectives. Costs increase when developers use inadequate processes. Higher reliability can be achieved by using better development process, risk management process, configuration management process, etc.

11.2.6.3 Process metrics

Based on the assumption that the quality of the product is a direct function of the process, process metrics can be used to estimate, monitor and improve the reliability and quality of software. ISO-9000 certification, or "quality management standards", is the generic reference for a family of standards developed by the International Standards Organization (ISO).

11.2.6.4 Fault and failure metrics

The goal of collecting fault and failure metrics is to be able to determine when the software is approaching failure-free execution. Minimally, both the number of faults found during testing (i.e., before delivery) and the failures (or other problems) reported by users after delivery are collected, summarized and analyzed to achieve this goal. Test strategy is highly relative to the effectiveness of fault metrics, because if the testing scenario does not cover the full functionality of the software, the software may pass all tests and yet be prone to failure once delivered. Usually, failure metrics are based upon customer information regarding failures found after release of the software. The failure data collected is therefore used to calculate failure density, Mean Time Between Failures (MTBF) or other parameters to measure or predict software reliability.

Mean Time to Failure (MTTF)

MTTF is a basic measure of reliability for non-repairable systems. It is the mean time expected until the first failure of a piece of equipment. MTTF is a statistical value and is meant to be the mean over a long period of time and large number of units. For constant failure rate systems, MTTF is the inverse of the failure rate. If failure rate is in failures/million hours, MTTF = 1,000,000 / Failure Rate for components with exponential distributions.

Technically MTBF should be used only in reference to repairable items, while MTTF should be used for non-repairable items. However, MTBF is commonly used for both repairable and non-repairable items.

Mean Time between Failures (MTBF)

MTBF is a basic measure of reliability for repairable items. It can be described as the number of hours that pass before a component, assembly, or system fails. It is a commonly-used variable in reliability and maintainability analyses.

MTBF can be calculated as the inverse of the failure rate for constant failure rate systems. For example: If a component has a failure rate of 2 failures per million hours, the MTBF would be the inverse of that failure rate.

MTBF = (1,000,000 hours) / (2 failures) = 500,000 hours

Actually MTBF is the summation of MTTF and MTRF (Mean Time To Repair).

MTBF=MTTF+MTTR

Availability

It is a measure of the time during which the system is available. It may be stated as:

Availability = MTBF / (MTBF + MMTR)

Probability of Failure on Demand (POFOD)

It is defined as the probability that the system will fail when a service is requested.

Rate of occurrence of failure (ROCOF)

It may be defined as the number of failures in unit time interval.

11.2.7 Software Reliability Improvement Techniques

Good engineering methods can largely improve software reliability.

Before the deployment of software products, testing, verification and validation are necessary steps. Software testing is heavily used to trigger, locate and remove software defects. Software testing is still in its infant stage; testing is crafted to suit specific needs in various software development projects in an ad-hoc manner. Various analysis tools such as trend analysis, fault-tree analysis, Orthogonal Defect classification and formal methods, etc, can also be used to minimize the possibility of defect occurrence after release and therefore improve software reliability.

After deployment of the software product, field data can be gathered and analyzed to study the behavior of software defects. Fault tolerance or fault/failure forecasting techniques will be helpful techniques and guide rules to minimize fault occurrence or impact of the fault on the system.

11.2.8 Software Fault Tolerance

Software fault tolerance is the ability for software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification. Software fault tolerance is a necessary component in order to construct the next generation of highly available and reliable computing systems from embedded systems to data warehouse systems.

In order to adequately understand software fault tolerance it is important to understand the nature of the problem that software fault tolerance is supposed to solve. Software faults are all design faults. Software manufacturing, the reproduction of software, is considered to be perfect. The source of the problem being solely design faults is very different than almost any other system in which fault tolerance is a desired property. This inherent issue, that software faults are the result of human error in interpreting a specification or correctly implementing an algorithm, creates issues, which must be dealt with in the fundamental approach to software fault tolerance.

Current software fault tolerance methods are based on traditional hardware fault tolerance. The deficiency with this approach is that traditional hardware fault tolerance was designed to conquer manufacturing faults primarily, and environmental and other faults secondarily. Design diversity was not a concept applied to the solutions to hardware fault tolerance, and to this end, N-Way redundant systems solved many single errors by replicating the same hardware. Software fault tolerance tries to leverage the experience of hardware fault tolerance to solve a different problem, but by doing so creates a need for design diversity in order to properly create a redundant system.

Design diversity is a solution to software fault tolerance only so far as it is possible to create diverse and equivalent specifications so that programmers can create software, which has different enough designs that they don't share similar failure modes. Design diversity and independent failure modes have been shown to be a particularly difficult problem though. The issue still remains that for a complex problem, the need for humans to solve that problem error free is not easily solvable.

Fault tolerance is defined as how to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring. Laprie argues that fault tolerance is accomplished using redundancy. This argument is good for errors which are not caused by design faults, however, replicating a design fault in multiple places will not aide in complying with a specification. It is also important to note the emphasis placed on the specification as the final arbiter of what is an error and what is not. Design diversity increases pressure on the specification creators to make multiple variants of the same specification, which are equivalent in order to aid the programmer in creating variations in algorithms for the necessary redundancy. The definition itself may no longer be appropriate for the type of problems that current fault tolerance is trying to solve, both hardware and software.

Randell argues that the difference between fault tolerance versus exception handling is that exception handling deviates from the specification and fault tolerance attempts to provide services compliant with the specification after detecting a fault. This is an important difference to realize between trying to construct robust software versus trying to construct reliable software. Reliable software will accomplish its task under adverse conditions while robust software will be able to indicate a failure correctly, (hopefully without the entire system failing.)

Software Bugs

Software faults are most often caused by design faults. Design faults occur when a programmer either misunderstands a specification or simply makes a mistake. Software faults are common for the simple reason that the complexity in modern systems is often pushed into the software part of the system. It is estimated that 60-90% of current computer errors are from software faults. Software faults may also occur from hardware; these faults are usually transitory in nature, and can be masked using a combination of current software and hardware fault tolerance techniques.

11.2.9 Software fault tolerance techniques

11.2.9.1 Recovery Blocks

The recovery block method is a simple method developed by Randell from what was observed as somewhat current practice at the time. The recovery block operates with an adjudicator, which confirms the results of various implementations of the same algorithm. In a system with recovery blocks, the system view is broken down into fault recoverable blocks. The entire system is constructed of these fault tolerant blocks. Each block contains at least a primary, secondary, and exceptional case code along with an adjudicator. The adjudicator is the component, which determines the correctness of the various blocks to try.

The adjudicator should be kept somewhat simple in order to maintain execution speed and aide in correctness. Upon first entering a unit, the adjudicator first executes the primary alternate. (There may be N alternates in a unit which the adjudicator may try.) If the adjudicator determines that the primary block failed, it then tries to roll back the state of the system and tries the secondary alternate. If the adjudicator does not accept the results of any of the alternates, it then invokes the exception handler, which then indicates the fact that the software could not perform the requested operation.

Recovery block operation still has the same dependency, which most software fault tolerance systems have: design diversity. The recovery block method increases the pressure on the specification to be specific enough to create different multiple alternatives that are functionally the same. This issue is further discussed in the context of the N-version method.

The recovery block system is also complicated by the fact that it requires the ability to roll back the state of the system from trying an alternate. This may be accomplished in a variety of ways, including hardware support for these operations. This try and rollback ability has the effect of making the software to appear extremely transactional, in which only after a transaction is accepted is it committed to the system. There are advantages to a system built with a transactional nature, the largest of which is the difficult nature of getting such a system into an incorrect or unstable state. This property, in combination with check pointing and recovery may aide in constructing a distributed hardware fault tolerant system.

11.2.9.2 N-version Software

The N-version software concept attempts to parallel the traditional hardware fault tolerance concept of N-way redundant hardware. In an N-version software system, each module is made with up to N different implementations. Each variant accomplishes the same task, but hopefully in a different way. Each version then submits its answer to voter or decider, which determines the correct answer, and returns that as the result of the module. This system can hopefully overcome the design faults present in most software by relying upon the design diversity concept. An important distinction in N-version software is the fact that the system could include multiple types of hardware using multiple versions of software. The goal is to increase the diversity in order to avoid common mode failures. Using N-version software, it is encouraged that each different version be implemented in as diverse a manner as possible, including different tool sets, different programming languages, and possibly different environments. The various development groups must have as little interaction related to the programming between them as possible. N-version software can only be successful and successfully tolerate faults if the required design diversity is met.

The dependence on appropriate specifications in N-version software, (and recovery blocks,) cannot be stressed enough. The delicate balance required by the N-version software method requires that a specification be specific enough so that the various versions are completely inter-operable, so that a software decider may choose equally between them, but cannot be so limiting that the software programmers do not have enough freedom to create diverse designs. The flexibility in the specification to encourage design diversity, yet maintain the compatibility between versions is a difficult task, however, most current software fault tolerance methods rely on this delicate balance in the specification.

11.2.9.3 N-version software and recover block - comparison

The differences between the recovery block method and the N-version method are not too numerous, but they are important. In traditional recovery blocks, each alternative would be executed serially until an acceptable solution is found as determined by the adjudicator. The recovery block method has been extended to include concurrent execution of the various alternatives. The N-version method has always been designed to be implemented using N-way hardware concurrently. In a serial retry system, the cost in time of trying multiple alternatives may be too expensive, especially for a real-time system. Conversely, concurrent systems require the expense of N-way hardware and a communications network to connect them. Another important difference in the two methods is the difference between an adjudicator and the decider. The recovery block method requires that each module build a specific adjudicator; in the N-version method, a single decider may be used. The recovery block method, assuming that the programmer can create a sufficiently simple adjudicator, will create a system, which is difficult to enter into an incorrect state. The engineering tradeoffs, especially monetary costs, involved with developing either type of system have their advantages and disadvantages, and it is important for the engineer to explore the space to decide on what the best solution for his project is.

11.2.10 Reliability Models

A reliability growth model is a mathematical model of software reliability, which predicts how software reliability should improve over time as faults are discovered and repaired. These models help the manager in deciding how much efforts should be devoted in testing. The goal of project manager is to test and debug the system until the required level of reliability is reached.

11.2.11 Jelinski-Moranda (J-M) Model

There are various models, which have been derived from reliability experiments in a number of application domains. These models are usually based on formal testing data. The quality of their results depends upon the input data, the better the outcome. The more data points available the better the model will perform. When using calendar time for large projects, you need to verify homogeneity of testing effort.

Software reliability growth models fall into two major categories:

- Time between failure models (MTBF)
- Fault count models (faults or time normalized rates)

Jelinski-Moranda model developed by Jelinski and Moranda in 1972 was one of the first software reliability growth models. It consists of some simple assumptions:

1. At the beginning of testing, there are u_0 faults in the software code with u_0 being an unknown but fixed number.

2. Each fault is equally dangerous with respect to the probability of its instantaneously causing a failure. Furthermore, the hazard rate of each fault does not change over time, but remains constant at ϕ.

3. The failures are not correlated, i.e. given u_0 and ϕ the times between failures $(\Delta t_1, \Delta t_2, \ldots \Delta t_{u0})$ are independent.

4. Whenever a failure has occurred, the fault that caused it is removed instantaneously and without introducing any new fault into the software.

As a consequence of these assumptions, the program hazard rate after removal of the (i−1)st fault is proportional to the number of faults remaining in the software with the hazard rate of one fault, $z_a(t) = \phi$, being the constant of proportionality:

$z(\Delta t \mid t_i{-}1) = \phi\, [u_0 - M_{(ti-1)}] = \phi\, [u_0 - (i - 1)]$ (1)

The Jelinski-Moranda model belongs to the binomial type of models. For these models, the failure intensity function is the product of the inherent number of faults and the probability density of the time until activation of a single fault, $f_a(t)$, i.e.:

$d_\mu(t)/dt = u_0 f_a(t) = u_0\, \phi \exp(-\phi t)$ (2)

Therefore, the mean value function is

$\mu(t) = u_0[1 - \exp(-\phi t)]$ (3)

It can easily be seen from equations (2) and (3) that the failure intensity can also be expressed as

$d_\mu(t)/dt = \phi\, [u_0 - \mu(t)]$ (4)

According to equation (4) the failure intensity of the software at time t is proportional to the expected number of faults remaining in the software; again, the hazard rate of an individual fault is the constant of proportionality. This equation can be considered the “heart” of the Jelinski-Moranda model. J-M model assumptions are hard to meet. In J-M model, reliability increases by a constant increment each time a fault is discovered and repaired.

11.2.12 Goel-Okumoto (GO) model

The model proposed by Goel and Okumoto in 1979 is based on the following assumptions:

1. The number of failures experienced by time t follows a Poisson distribution with mean value function $\mu(t)$. This mean value function has the boundary conditions $\mu(0) = 0$ and $\lim_{t \to \infty} \mu(t) = N < \infty$.
2. The number of software failures that occur in $(t, t+\Delta t]$ with $\Delta t \to 0$ is proportional to the expected number of undetected faults, $N - \mu(t)$. The constant of proportionality is ϕ.
3. For any finite collection of times $t_1 < t_2 < \cdot \cdot \cdot < t_n$ the number of failures occurring in each of the disjoint intervals $(0, t_1), (t_1, t_2), ..., (t_n-1, t_n)$ is independent.
4. Whenever a failure has occurred, the fault that caused it is removed instantaneously and without introducing any new fault into the software.

Since each fault is perfectly repaired after it has caused a failure, the number of inherent faults in the software at the beginning of testing is equal to the number of failures that will have occurred after an infinite amount of testing. According to assumption 1, $M(\infty)$ follows a Poisson distribution with expected value N. Therefore, N is the expected number of initial software faults as compared to the fixed but unknown actual number of initial software faults u_0 in the Jelinski-Moranda model. Indeed, this is the main difference between the two models.

Assumption 2 states that the failure intensity at time t is given by

$d_{\mu(t)} / d_t = \phi[N - \mu(t)]$

Just like in the Jelinski-Moranda model the failure intensity is the product of the constant hazard rate of an individual fault and the number of expected faults remaining in the software. However, N itself is an expected value.

11.2.13 Musa's basic execution time model

Musa's basic execution time model is based on an execution time model, i.e., the time taken during modeling is the actual CPU execution time of the software being modeled. This model is simple to understand and apply, and its predictive value has been generally found to be good. The model focuses on failure intensity while modeling reliability. It assumes that the failure intensity decreases with time, that is, as (execution) time increases, the failure intensity decreases. This assumption is generally true as the following is assumed about the software testing activity, during which data is being collected: during testing, if a failure is observed, the fault that caused that failure is detected and the fault is removed. Even if a specific fault removal action might be unsuccessful, overall failures lead to a reduction of faults in the software. Consequently, the failure intensity decreases. Most other models make a similar assumption, which is consistent with actual observations.

In the basic model, it is assumed that each failure causes the same amount of decrement in the failure intensity. That is, the failure intensity decreases with a constant rate with the number of failures. In the more sophisticated Musa's logarithmic model, the reduction is not assumed to be linear but logarithmic.

Musa's basic execution time model developed in 1975 was the first one to explicitly require that the time measurements be in actual CPU time utilized in executing the application under test (named "execution time" *t* in short).

Although it was not originally formulated like that the model can be classified by three characteristics:

1. The number of failures that can be experienced in infinite time is finite.
2. The distribution of the number of failures observed by time *t* is of Poisson type.
3. The functional form of the failure intensity in terms of time is exponential.

It shares these attributes with the Goel-Okumoto model, and the two models are mathematically equivalent. In addition to the use of execution time, a difference lies in the interpretation of the constant per-fault hazard rate ϕ. Musa split ϕ up in two constant factors, the linear execution frequency f and the so-called fault exposure ratio K:

$d_{\mu(t)} / d_t = f K [N - \mu(t)]$

f can be calculated as the average object instruction execution rate of the computer, *r*, divided by the number of source code instructions of the application under test, I_S, times the average number of object instructions per source code instruction, Qx: $f = r / I_S Q_x$. The fault exposure ratio relates the fault velocity $f [N - \mu(t)]$, the speed with which defective parts of the code would be passed if all the statements were consecutively executed, to the failure intensity experienced. Therefore, it can be interpreted as the average number of failures occurring per fault remaining in the code during one linear execution of the program.

11.2.14 Markov Model

This analysis yields results for both, the time dependent evolution of the system and the steady state of the system. For example, in reliability engineering, the operation of the system may be represented by a state diagram, which represents the states and rates of a dynamic system. This diagram consists of nodes (representing a possible state of the system, which is determined by the states of the individual components & sub-components) connected by arrows (representing the rate at which the system operation transitions from one state to the other state). Transitions may be determined by a variety of possible events, for example the failure or repair of an individual component. A state-to-state transition is characterized by a probability distribution. Under reasonable assumptions, the system operation may be analyzed using a Markov model.

A Markov model analysis can yield a variety of useful performance measures describing the operation of the system. These performance measures include the following:

- System reliability.
- Availability.
- Mean time to failure (MTTF).
- Mean time between failures (MTBF).
- The probability of being in a given state at a given time.
- The probability of repairing the system within a given time period (maintainability).
- The average number of visits to a given state within a given time period.

- And many other measures.

The name Markov model is derived from one of the assumptions which allows this system to be analyzed; namely the Markov property. The Markov property states: given the current state of the system, the future evolution of the system is independent of its history. The Markov property is assured if the transition probabilities are given by exponential distributions with constant failure or repair rates. In this case, we have a stationary, or time homogeneous, Markov process. This model is useful for describing electronic systems with repairable components, which either function or fail. As an example, this Markov model could describe a computer system with components consisting of CPUs, RAM, network card and hard disk controllers and hard disks.

The assumptions on the Markov model may be relaxed, and the model may be adapted, in order to analyze more complicated systems. Markov models are applicable to systems with common cause failures, such as an electrical lightning storm shock to a computer system. Markov models can handle degradation, as may be the case with a mechanical system. For example, the mechanical wear of an aging automobile leads to a non-stationary, or non-homogeneous, Markov process, with the transition rates being time dependent. Markov models can also address imperfect fault coverage, complex repair policies, multi-operational-state components, induced failures, dependent failures, and other sequence dependent events.

11.3 Summary

Software reliability is a key part in software quality. Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. The study of software reliability can be categorized into three parts: modeling, measurement and improvement.

Reliability of the software depends on the faults in the software. To assess the reliability of software, reliability models are required. To use the model, data is collected about the software. Most reliability models are based on the data obtained during the system and acceptance testing.

Software reliability modeling has matured to the point that meaningful results can be obtained by applying suitable models to the problem. There are many models exist, but no single model can capture a necessary amount of the software characteristics. Assumptions and abstractions must be made to simplify the problem. There is no single model that is universal to all the situations.

Software reliability measurement is naive. Measurement is far from commonplace in software, as in other engineering field. "How good is the software, quantitatively?" As simple as the question is, there is still no good answer. Software reliability cannot be directly measured, so other related factors are measured to estimate software reliability and compare it among products. Development process, faults and failures found are all factors related to software reliability.

Software reliability improvement is hard. The difficulty of the problem stems from insufficient understanding of software reliability and in general, the characteristics of software. Until now there is no good way to conquer the complexity problem of software. Complete testing of a moderately complex software module is infeasible. Defect-free software product cannot be assured. Realistic constraints of time and budget severely limits the effort put into software reliability improvement.

As more and more software is creeping into embedded systems, we must make sure they don't embed disasters. If not considered carefully, software reliability can be the reliability bottleneck of the whole system. Ensuring software reliability is no easy task. As hard as the problem is, promising progresses are still being made toward more reliable software. More standard components, and better process are introduced in software engineering field.

11.4 Keywords

Software Reliability: It is defined as: the probability of failure-free software operation for a specified period of time in a specified environment.

Reliability growth model: It is a mathematical model of software reliability, which predicts how software reliability should improve over time as faults are discovered and repaired.

MTTF: It is a basic measure of reliability for non-repairable systems. It is the mean time expected until the first failure of a piece of equipment.

MTBF: It is a basic measure of reliability for repairable items. It can be described as the number of hours that pass before a component, assembly, or system fails.

Availability: It is a measure of the time during which the system is available.

POFOD: It is defined as the probability that the system will fail when a service is requested.

ROCOF: It may be defined as the number of failures in unit time interval.

11.5 Self-Assessment Questions

[1] What do you understand by software reliability? Differentiate between software reliability and hardware reliability.

[2] Differentiate between fault, error and failure. Does testing observe faults or failures?

[3] What are the different categories of software failure?

[4] What are the assumptions made in Jelinski-Moranda Model? Explain the J-M model and discuss its limitations.

[5] What is the difference between software reliability and hardware reliability? Explain.

[6] What do you understand by software fault tolerance? Discuss the recovery block and N-version software techniques to fault tolerance.

[7] What is a metric? Give an overview of the different reliability metrics.

[8] What are the differences between JM model, GO model, and Musa's basic execution time model? Explain.

11.6 Suggested readings/References

41. Software Engineering concepts by Richard Fairley, Publication - Tata McGraw Hill.
42. An integrated approach to Software Engineering by Pankaj Jalote, Publication - Narosha Publishing houre.
43. Software Engineering by Sommerville, Publication - Pearson Education.

44. Software Engineering – A Practitioner’s Approach by Roger S Pressman, Publication - Tata McGraw-Hill.

Lesson No. 12 *Writer: Dr. Rakesh Kumar*

Object Oriented Design

Vetter:

12.0 Objectives

The objective of this lesson is to make the students familiar with object oriented design. Earlier in chapter 6 and 7 function oriented design was discussed. This chapter is intended to impart the knowledge of object modeling, functional modeling, and dynamic modeling. The important objective of this lesson is to get the student acquainted with OMT, a methodology for object oriented design.

12.1 Introduction

Object oriented approach for software development has become very popular in recent years. There are many advantages of using this approach over function oriented design such as reusability, permitting changes more easily, reduction in development cost and time etc. There is a fundamental difference between function oriented design and object oriented design. The former is based on procedural abstraction while later is using data abstraction. In object oriented design our focus is to identify the classes in the system and the relationship between them.

12.2 Presentation of contents

12.2.1 Object Oriented Design Methodology

12.2.5.2 System Design

12.2.5.3 Object Design

12.2.1 Object Oriented Design Methodology

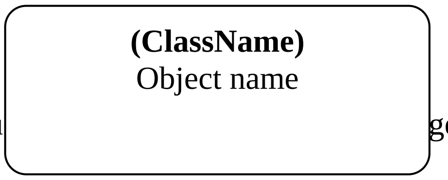
OMT (Object Modeling Technique) is a software development methodology given by James Rumbaugh et.al. This methodology describes a method for analysis, design and implementation of a system using object-oriented technique. It is a fast, intuitive approach for identifying and modeling all the objects making up a system. The static, dynamic and functional behaviors of the system are described by object model, dynamic model and functional model of the OMT. The object model describes the static, structural and data aspects of a system. The dynamic model describes the temporal, behavioral and control aspects of a system. The functional model describes the transformational and functional aspects of a system. Every system has these three aspects. Each model describes one aspect of the system but contains references to the other models.

12.2.2 Concepts and Notations for OMT methodology

12.2.2.1 Object Modeling

An object model describes the structure of objects in a system: their identity, relationship to other objects, attributes and operations. The object model is represented graphically with an object diagram. The object diagram contains classes interconnected by association lines. Each class represents a set of individual objects. The association lines establish relationships among classes. Each association line represents a set of links from the object of one class to the object of another class.

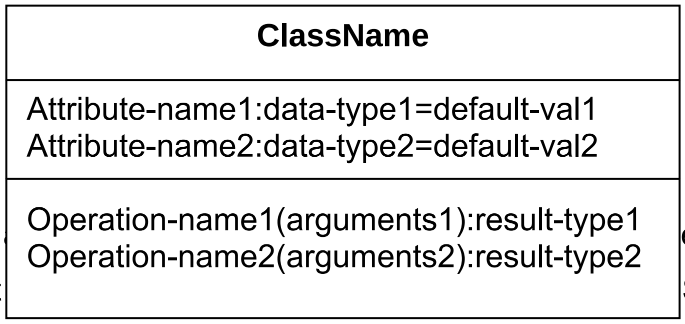
12.2.2.2 Object: An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem in hand. Objects are represented by the following icon:



12.2.2.3 Derived object: It is defined as a function of one or more objects. It is completely determined by the other objects. Derived object is redundant but can be included in the object model.

12.2.2.4 Derived attribute: A derived attribute is that which is derived from other attributes. For example, age can be derived from date of birth and current date.

12.2.2.5 Class: A class describes a group of objects with similar properties, operations and relationships to other objects. Classes are represented by the rectangular symbol and may be divided into three parts. The top part contains the name of the class, middle part attributes and bottom part operations. An attribute is a data value held by the objects in a class. For example, person is a class; Mayank is an object while name, age, and sex are its attributes. Operations are functions or transformations that may be applied to or by objects in a class. For example push and pop are operations in stack class.


ClassName
Attribute-name1:data-type1=default-val1
Attribute-name2:data-type2=default-val2
Operation-name1(arguments1):result-type1
Operation-name2(arguments2):result-type2

12.2.2.6 Links eptual connection between object So ‘flies’ is a link between Mayank and Jaguar.

An association describes a group of links with common structure and common semantics. For example a pilot flies an airplane. So here ‘flies’ is an association between pilot and airplane. All the links in an association connect objects from the same classes.

Associations are bidirectional in nature. For example, a pilot flies an airplane or an airplane is flown by a pilot.

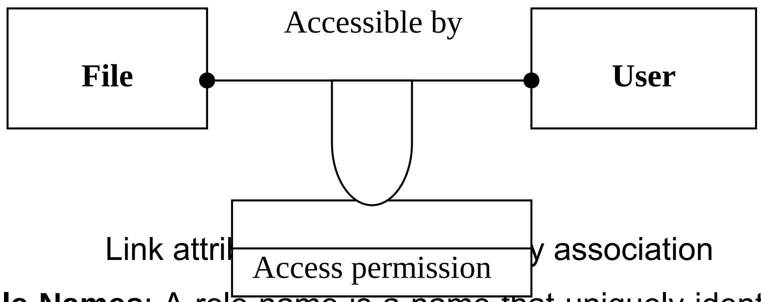
Associations may be binary, ternary or higher order. For example, a programmer develops a project in a programming language represents a ternary association

among programmer, project and programming language. Links and associations are represented by a line between objects or classes as shown in a diagram below:

12.2.2.7 Multiplicity: It specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity is described in the following manner:

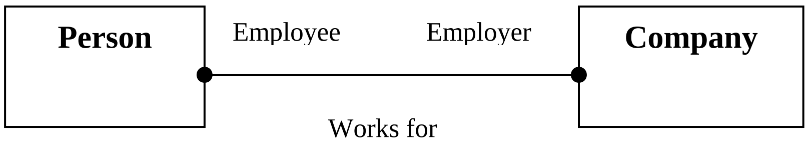
- Line without any ball indicates one-to-one association.
- Hollow ball indicates zero or one.
- Solid ball indicates zero, one or more.
- 1,2,6 indicates 1 or 2 or 6.
- 1+ indicates 1 or more

12.2.2.8 Link attributes: It is a property of the links in an association. For example, 'accessible by' is an association between class File and class User. 'Access permission' is a link attribute.



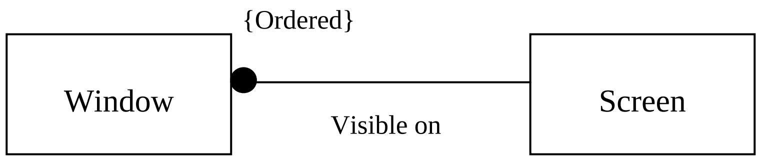
Link attri y association

12.2.2.9 Role Names: A role name is a name that uniquely identifies one end of an association. Binary association has two roles. A role name is written next to the association line near the class that plays the role. For example, consider the association 'a person works for a company', in this employee and employer are role names for the classes person and company respectively as shown in fig below.



Role names for an association

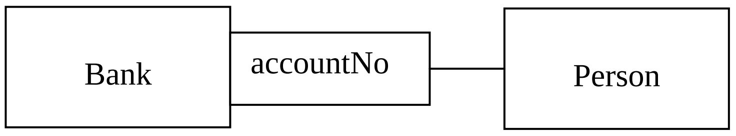
12.2.2.10 Ordering: Some times the objects on the many side of an association have order. An ordered set of objects of an association is indicated by writing {ordered} next to the multiplicity dot for the role as shown in figure below. Consider the example of association between Window class and Screen class. A screen can contain a number of windows. Windows are explicitly ordered. Only topmost window is visible on the screen at any time.



Ordered sets in an association

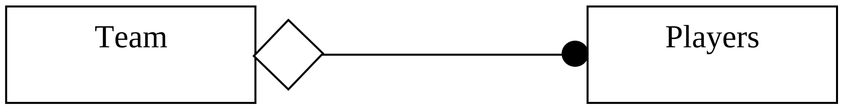
12.2.2.11 Qualification: A qualifier is an association attribute. For example, a person object may be associated to a Bank object. An attribute of this association is the accoutNo. The accountNo is the qualifier of this association.

A qualifier is shown as a small rectangle attached to the end of an association as shown in figure below. The qualifier rectangle is part of the association, not of class.



Qualified association

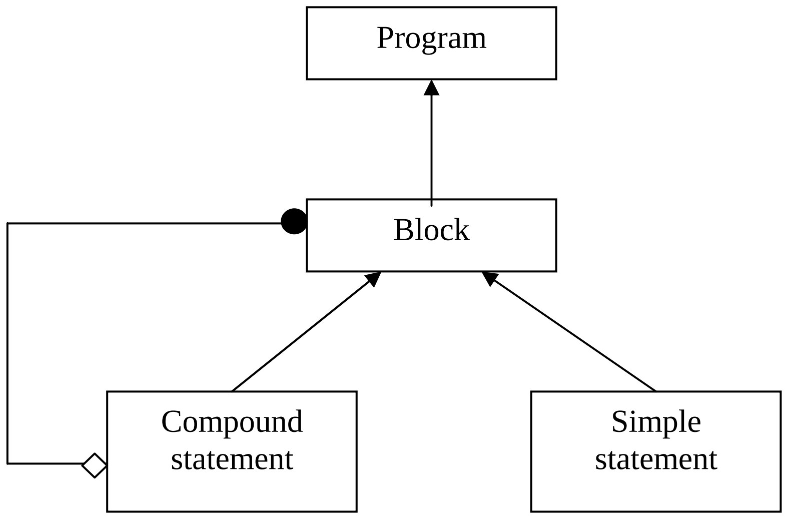
12.2.2.12 Aggregation: Aggregation is a form of association. It is the "part-whole" or "a-part-of" relationship in which objects representing the component of something are associated with an object representing the entire assembly. A hollow diamond is attached to the end of the path to indicate the aggregation. For example, a team is aggregation of players.



Aggregation

Aggregation can be fixed, variable or recursive.

- In a fixed aggregation number and subtypes are fixed i.e. predefined.
- In a variable aggregation number of parts may vary but number of levels is finite.
- A recursive aggregate contains, directly or indirectly, an instance of the same aggregate. The number of levels is unlimited. For example, a computer program is an aggregation of blocks, with optionally recursive compound statements. The recursion terminates with simple statement. Blocks can be nested to arbitrary depth.



Recursive Aggregation

12.2.2.13 Generalization: It is the relationship between a more general class and a more specific class. The general class is called as super class and specific class is called as subclass. Generalization is indicated by a triangle connecting a super class to its subclass as shown in fig below.

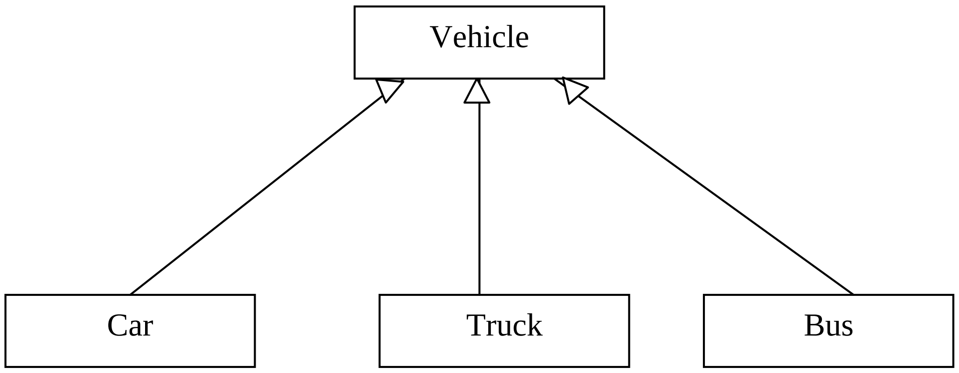
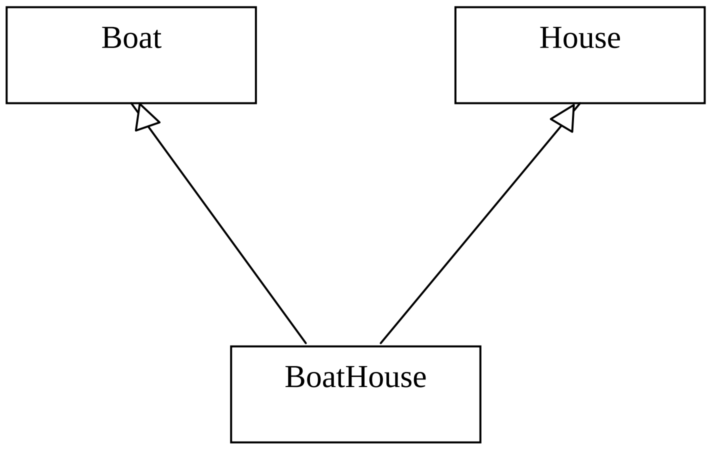


Fig. Inheritance

12.2.2.14 Multiple Inheritance: if a class inherits features from more than one superclass, it is called as multiple inheritance. A class with more than one superclass is also called as join class. For example, BoatHouse class inherits features from class Boat and House as shown in fig below:



Multiple Inheritance

12.2.2.15 Metadata: Metadata is data about data. For example, the definition of a class is metadata. Models, catalogs, blueprints, dictionary etc. are all examples of metadata.

12.2.2.16 Grouping Constructs: There are two grouping constructs: module and sheet.

Module is logical construct for grouping classes, associations and generalizations. An object model consists of one or more modules. The module name is usually listed at the top of each sheet.

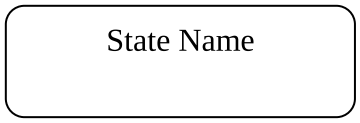
A sheet is a single printed page. Sheet is the mechanism for breaking a large object model into a series of pages. Each module is contained in one or more sheets. Sheet numbers or sheet names inside circle contiguous to a class box indicate other sheets that refer to a class.

12.2.3 Dynamic Modeling

Dynamic model describes those aspects of the system that changes with the time. It is used to specify and implement control aspects of the system. It depicts states, transitions, events and actions. The OMT state transition diagram is a network of states and events. Each state receives one or more events, at that time it makes the transition to the next state. The next state depends upon the current state as well as the events.

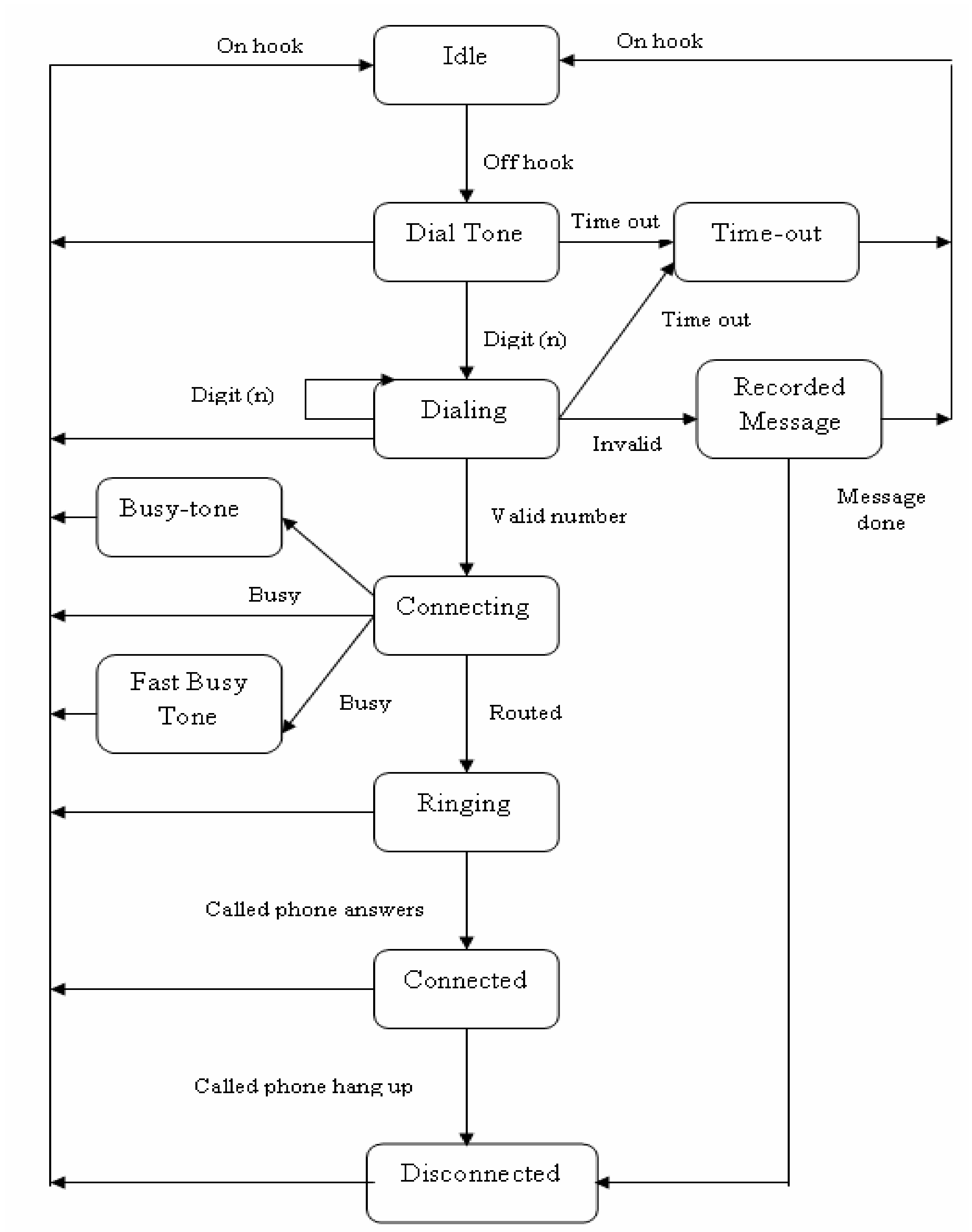
State: the attribute values and links held by an object are called its state. Stack is empty or stack is full are different states of object stack. A state corresponds to interval between two events received by an object. So a state has duration.

State Icon: A state is drawn as a rounded box containing an optional name as shown in fig below:



Event: An event is something that happens at a point in time. An individual stimulus from one object to another is an event. Press a button on mouse, airplane departs from an airport are examples of event. An event does not have duration.

State Diagram: It relates events and states. A change of state caused by an event is called a transition. Transition is drawn as an arrow from the receiving state to the target state. A state diagram is graph whose nodes are states and whose directed arcs are transitions labeled by event names. State diagram specifies the state sequence caused by an event sequence. The state diagram for the phone line is as shown below:

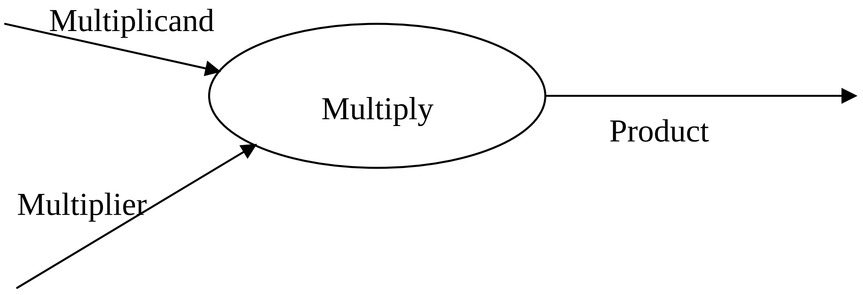

On hook
Idle
On hook
Off hook
Dial Tone
Time out
Time-out
Digit (n)
Time out
Digit (n)
Dialing
Recorded Message
Invalid
Busy-tone
Valid number
Message done
Busy
Connecting
Fast Busy Tone
Busy
Routed
Ringing
Called phone answers
Connected
Called phone hang up
Disconnected

12.2.4 Functional Modeling

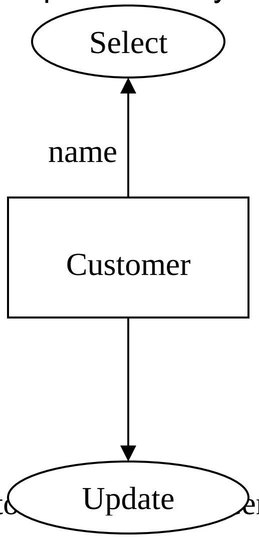
The functional model specifies the result of a computation without specifying how or when they are computed. It specifies the meaning of the operations and any constraints in the object model and the actions in the dynamic model. The functional model consists of multiple data flow diagram, which specifies the meaning of operations and constraints.

Data Flow Diagram: A data flow diagram is a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects. A data flow diagram contains four symbols: processes, actors, data stores and data flows.

Processes: A process transforms data values. It is drawn as an ellipse having a description of the transformation, generally its name (as shown in figure below). Each process has a fixed number of input and output data arrows, each of which carries a value of given type. Inputs and outputs can be labeled to show their role in computation.

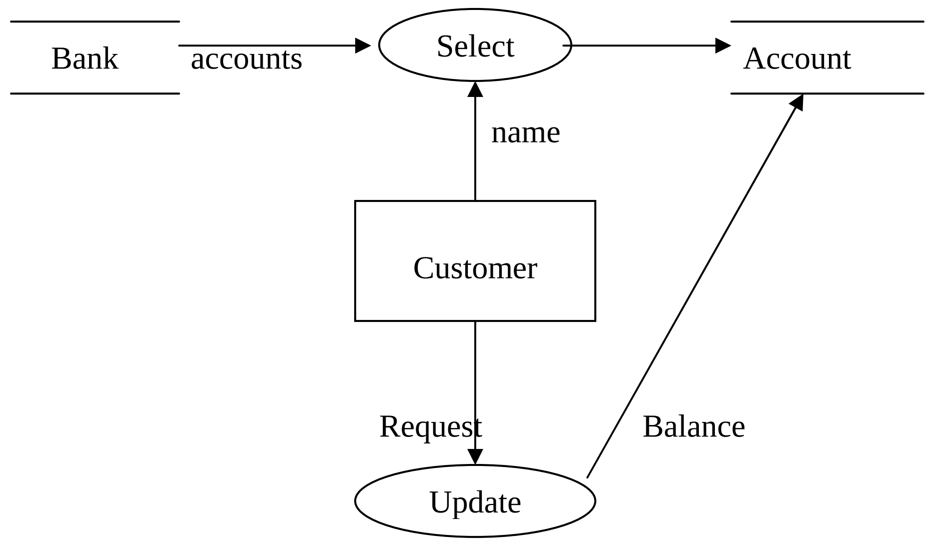


Actors: An actor is an active object that drives the data flow diagram by producing or consuming values. Actors are attached to the inputs and outputs of a data flow diagram. Actors are also called as terminators as they act as source and sink for data. An actor is represented by rectangle.

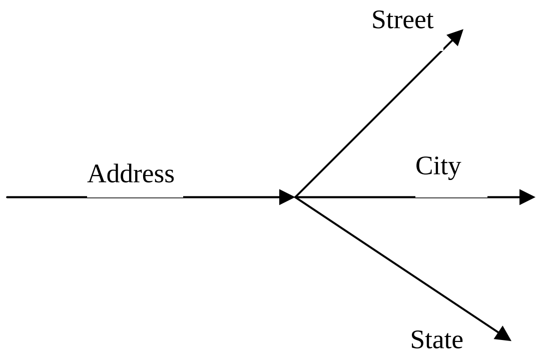


request

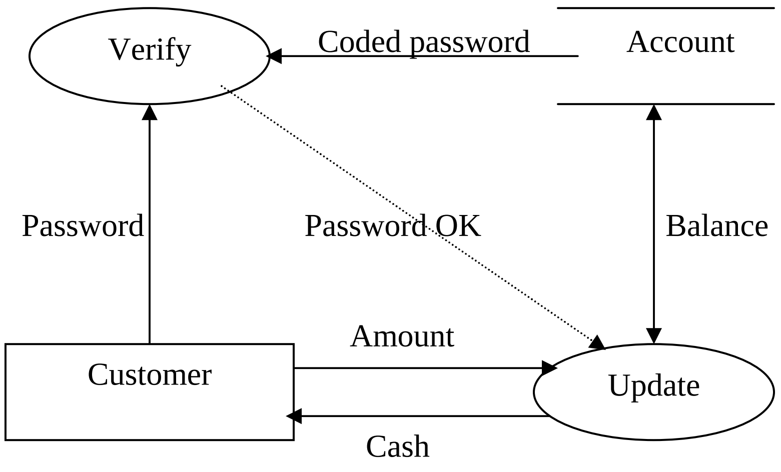
Data Stores: It stores data for later use. It does not generate any operation on its own but can respond to request. So it is a passive object in a data flow diagram. It is represented by a pair of parallel lines containing the name of store as shown in figure below. Input arrow indicates storing data in the data store and output arrow indicates accessing of data from data store.



Data Flows: A data flow connects the output of an object or process to the input of another object or process. An arrow between the producer and the consumer of the data value represents a data flow. Arrow is labeled with description of data. Sometimes an aggregate value is split into its constituents, each of which goes to a different process. A fork in the path as shown below can show this.



Control Flow: It is a Boolean value that affects whether a process is evaluated. The control flow is not an input value to the process. It represented by a dotted line from a process originating the Boolean value to the process being controlled as shown in figure below. This DFD is for a withdrawal from a bank account. The customer supplies a password and an amount. The update (withdrawal) can occur only when password is OK, which is shown as control flow in the diagram.



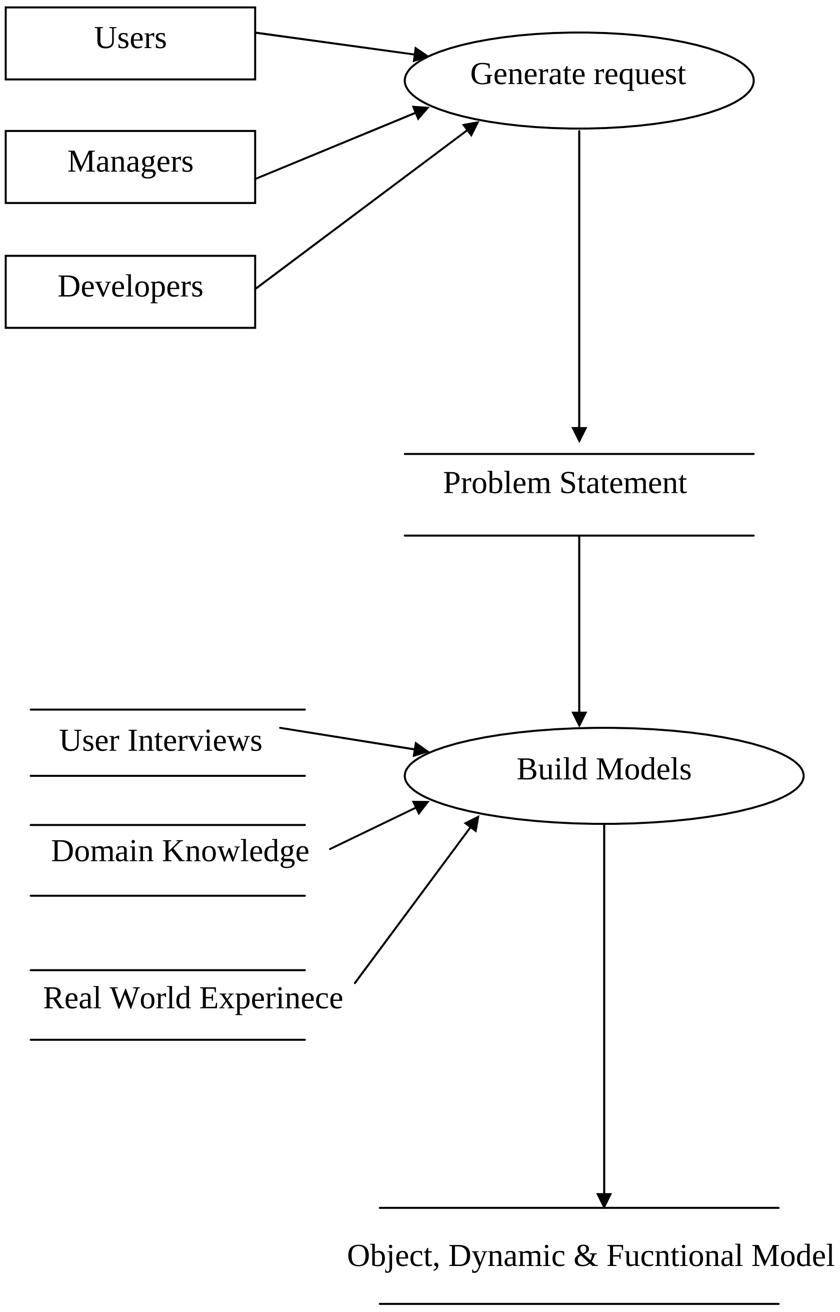
Control flow

12.2.5 OMT method

The OMT method is composed of following four phases:

- Analysis
- System design
- Object design
- Implementation

12.2.5.1 Analysis: In this step, a model of the system is prepared in reference to what the system will do. The model is expressed in terms of objects and relationships, control flow and functional transformation. In this phase three models- object model, dynamic model and functional model are prepared. These models are iterated in order to refine them. The outcome of this phase is problem statement, object model, dynamic model and functional model.



DFD for analysis process

12.2.5.1.1Object Model

It consists of object diagrams and data dictionary. Following guidelines are used to construct the object model:

- Identify objects and classes

- Begin a data dictionary containing descriptions of classes, attributes and associations.
- Add associations between classes
- Add attributes for objects and links
- Organize and simplify object classes using inheritance
- Test access paths using scenarios and iterate the above step if necessary.
- Group classes into modules, based on close coupling and related functions.

Identifying Object Classes: The first step in constructing an object model is to identify relevant object classes from the application domain. Classes generally correspond to nouns in the problem description. List all the classes that come to mind. Then identify right classes and discard others. Following are the guidelines for keeping the right classes:

- Find redundant classes: If two classes have the same information, the most descriptive name should be kept.
- Find irrelevant classes: If a class has little or nothing to do with the problem, it should be discarded.
- Find vague classes: A class should be specific. Classes with ill-defined boundaries and too broad in scope are vague and may be discarded.
- Check for attributes: check name could be an attribute not a class.
- Check for operations: If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.
- Check for roles: The name of a class should reflect its intrinsic nature and not a role that it plays in an association.
- Check for implementation constructs: A name extraneous to the real world should be eliminated from the analysis model. That may be needed later during design, but not now.

Prepare Data Dictionary: Prepare a data dictionary for all modeling entities by describing each object class. Describe the scope of the class within the current problem, including any assumptions or restrictions on its membership or use. The data dictionary also describes associations, attributes, and operations.

Identifying Associations: Any dependency between two or more classes is an association. A reference from one class to another is an association. Verb or verb phrases corresponds to associations in the problem description. For example, location (next to, part of, contained in), directed actions (drives), communication (talk to) ownership (has or part of), satisfaction of some condition (works-for, manages) etc. all associations may not be right. Unnecessary associations must be discarded. Following guidelines may be used to discard incorrect associations:

- Associations between eliminated classes: If one of the classes in the association has been eliminated, then the association must be eliminated or restated in terms of other classes.
- Irrelevant or implementation associations: Eliminate associations that are outside the problem domain or deal with implementation constructs.
- Actions. An association should describe a structural property of the application domain, not a transient event.
- Derived association: Eliminate associations that can be defined in terms of other associations.
- Add role names to associations, wherever appropriate. Qualify association if needed.
- Indicate multiplicity, if possible.
- Check for any missing association.

Identify attributes: Attributes usually corresponds to nouns followed by possessive phrases such as color of the car in the problem statement. Some guidelines for keeping right attributes:

- If independent existence of an entity is important rather than its value then it is object not attribute.
- If the value of an attribute depends on a particular context then it is qualifier not attribute.
- If the property depends upon the presence of a link then it is link attribute.
- An attribute that seems different and unrelated from other attributes may be a class. Make it a separate class.

- Minor attributes and internal attributes must be omitted in the analysis phase.

Refining with inheritance: In this step, classes are organized by using inheritance in order to share common structures. This could be done in two directions: bottom up and top down. For bottom up direction look for classes with similar attributes, associations or operations and then make a superclass to share common features. For top-down direction, look for noun phrases composed of various adjectives on the class name and consider them subclass. Enumerated sub cases in the application domain are main source of specialization. Some guidelines are listed below:

- Avoid excessive refinement.
- Multiple inheritance may be used for more sharing but only when necessary.
- Attributes and associations must be assigned to specific classes in the class hierarchy.
- Generalize the association classes, if the same association name appears more than once and almost with the same meaning.

Testing access Paths: Using object model diagram, trace access paths to see whether they give sensible result. Ask the following questions:

- Do they give unique result, when it is expected?
- Is there a way to take unique value out of many when needed?
- Are there useful questions which cannot be answered?

Iterating Object Modeling: Object model is rarely correct in a single go. It needs many iterations in order to refine it.

Grouping classes into Modules: diagrams may be divided into sheets of uniform size. A module is a set of classes. Modules may vary in size.

12.2.5.1.2 Dynamic Model:

It consists of state diagrams and event flow diagrams. Following guidelines are used to construct the dynamic model:

- Prepare scenarios of typical interaction sequences.
- Identify events between objects and prepare an event trace for each scenario.
- Develop a state diagram for each class that has important dynamic behavior.

- Check for consistency and completeness of events shared among the state diagrams.

Prepare scenarios of typical interaction sequences: A scenario is a sequence of events. Prepare scenarios for major interactions, external display formats and information exchanges.

Identify events between objects and prepare an event trace for each scenario: events are signals, inputs, decisions, interrupts, transitions and actions from users or external devices. Identify all events by using scenarios.

Event trace is an ordered list of events between different objects. Prepare each scenario as an event trace.

Develop a state diagram for each class that has important dynamic behavior: Start with the event trace diagrams that affect the class being modeled. Pick a trace showing a particular interaction and only consider the events affecting single object. Arrange the events into path whose arcs are labeled by the input and output events found along one column in the trace. The interval between any two events is state. Give each state a name. if the scenario can be repeated infinitely, close the path in the state diagram. Then merge other scenarios into the state diagram.

Check for consistency and completeness of events shared among the state diagrams: every event should have a sender and a receiver, occasionally the same object. States without predecessors or successors are suspicious; ensure they represent starting and terminating points of the interaction sequence. Make sure that corresponding events on different state diagrams are consistent.

12.2.5.1.3 Functional Model:

It consists of data flow diagrams and constraints. Following guidelines are used to construct the functional model:

- Identify input and output values.
- Build data flow diagrams.
- Describe what each function does.
- Identify constraints.
- Specify optimization criteria.

Identify input and output values: List input and output values, which are parameters of events between the system and the outside world. Check for missing input/output values through problem statement.

Build data flow diagrams: DFDs specify only dependencies among operations. They do not show sequencing or decisions. Construct DFD to show how each value is computed from input values.

Describe what each function does: After completing and refining DFDs, write description of each function in natural language, decision table, pseudo code or any other appropriate form. The description can be procedural or declarative.

Identify constraints: Constraints are functional dependencies between objects that are not related by an input-output dependency. Identify all constraints between objects. For example, account balance can not be negative.

Specify optimization criteria: Specify values to be maximized, minimized or optimized.

12.2.5.2 System Design

In this step the high level structure of the system is chosen. The outcome of this phase is structure of basic architecture for the system as well as high-level strategy decisions. Following steps are considered in this phase:

- Organize the system into subsystem.
- Identify concurrency inherent in the problem.
- Allocate subsystems to processors and tasks.
- Choose the basic strategy for implementing data stores in terms of data structures, files and databases.
- Identify global resources and determine mechanisms for controlling access to them.
- Choose an approach to implement software control.
- Consider boundary conditions.
- Establish trade off priorities.

Organize the system into subsystem: A subsystem is a collection of related classes, associations, operations, events and constraints having well defined small interfaces to other subsystems. A subsystem is identified by the services it provides. A service is group of related functions that share some common purpose. Each subsystem may be decomposed into smaller subsystems of its own. The lowest level subsystems are called modules. The relationship between two subsystems can be client-supplier or peer-to-peer. The decomposition of systems into subsystems may be organized as sequence of horizontal layers or vertical partitions.

Identify concurrency inherent in the problem: Two objects are inherently concurrent if they can receive events at the same time without interacting. The dynamic model is the guide to identifying concurrency. If events are unsynchronized, objects cannot be folded into a single thread of control. A thread of control is path through a set of state diagrams on which only a single object at a time is active.

Allocate subsystems to processors and tasks: Each concurrent subsystem must be allocated to a hardware unit. The designer must do the following:

- Estimate performance needs and resources needed to satisfy them.
- Choose H/W or S/W implementations for subsystems.
- Allocate software subsystems to processors to satisfy performance needs and minimize interprocessor communication.
- Determine the connectivity of the physical units that implement the subsystems.

Choose the basic strategy for implementing data stores in terms of data structures, files and databases: The internal and external data stores in a system provide clear separation points between subsystems. Each data store may combine data structures, files and databases implemented in primary or secondary memory.

Identify global resources and determine mechanisms for controlling access to them: global resources such as processors, tape drives, disk space, mouse buttons, file names, class names, databases etc. must be identified. A mechanism for controlling access to the must be determined.

Choose an approach to implement software control: There are two types of control flows in software: external control and internal control.

External control is the flow of externally-visible events among objects in the system. There are three types of such control: procedure-driven sequential, event driven sequential and concurrent. The type of control used depends upon the language, OS etc.

Internal control is the flow of control within a process. It exists in the implementation. Commonly used control flow are procedure calls, quasi-concurrent inter-task calls, concurrent inter-task calls etc.

Consider boundary conditions: The system designer must consider the boundary conditions of the system such as initialization, termination and failure.

Establish trade off priorities: the system designer must set priorities that will be sued to guide trade-offs during the rest of design. The entire character of a system is affected by the trade-off decisions made by the designer.

12.2.5.3 Object Design

In this phase, the analysis model is elaborated to provide a detailed basis for implementation. Following steps are considered in this phase:

- Obtain operations for the object model from the other models.
- Design algorithms to implement operations.
- Optimize access paths to data.
- Implement software control by fleshing out the approach chosen during system design.
- Adjust class structure to increase inheritance.
- Design implementations of associations.
- Determine the exact representation of object attributes.
- Package classes and associations into modules.

Obtain operations for the object model from the other models: The designer must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model.

Design algorithms to implement operations: For each operation in functional model, an algorithm must be developed. Use following guidelines:

- Choose algorithms that minimize the cost of implementation.
- Select data structures appropriate to the algorithms.
- Define new internal classes and operations as necessary.
- Assign responsibility for operations to appropriate classes.

Optimize access paths to data: The inefficient but semantically-correct analysis model can be optimized to make implementation more efficient. For optimization, the designer must:

- Add redundant associations to minimize access cost and maximize convenience.
- Rearrange the computation for greater efficiency.
- Save derived attributes to avoid recomputation of complicated expressions.

Implement software control by fleshing out the approach chosen during system design: There are three basic approaches to implementing the dynamic model: procedure-driven, event-driven and concurrent. Implement appropriate one.

Adjust class structure to increase inheritance: The definitions of classes and operations must be adjusted to increase the amount of inheritance. It is done in the following way:

- Rearrange and adjust classes and operations to increase inheritance.
- Adjust common behavior out of groups of classes.
- Use delegation to share behavior when inheritance is semantically invalid.

Design implementations of associations: We can either choose a global strategy for implementing all associations uniformly or we can select a particular technique for each association, taking into account the way it will be used in the application.

Determine the exact representation of object attributes: Designer must choose when to use primitive types in representing objects and when to combine groups of related objects. Classes can be defined in terms of other classes. But every thing in a class is implemented in terms of primitive data types.

Package classes and associations into modules: Packaging involves the following issues: Hiding internal information from outside view, Coherence entities, constructing physical modules.

12.3 Summary

This chapter is focused on how a software system can be design using objects and classes while in chapter 6 and chapter 7 the focus was on function oriented design. In object oriented approach, an object is the basic design unit. During design the classes for objects are identified. A class represents the type for the object and defines the possible state space for the objects of that class and the operations that can be performed on the objects. An object is an instance of the class. Objects do not exist in isolation but are related to each other. One f the goal of design here is to identify the relationships between the objects of different classes.

The OMT methodology describes a method for analysis, design and implementation of a system using object-oriented technique. The static, dynamic and functional behaviors of the system are described by object model, dynamic model and functional model. The object model describes the static, structural and data aspects of a system. The dynamic model describes the temporal, behavioral and control aspects of a system. The functional model describes the transformational and functional aspects of a system. OMT first creates an object model for the system, and then refines it through dynamic and functional modeling. Identifying the internal classes and optimization are the final steps in this methodology for creating a design.

12.4 Key words

Object: An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem in hand.

Class: A class describes a group of objects with similar properties, operations and relationships to other objects.

Inheritance & Multiple Inheritance: Inheritance is a relation between classes that allows for definition of one class based on the definition of existing class. If a class inherits features from more than one superclass, it is called as multiple inheritance.

Dynamic model: describes those aspects of the system that changes with the time.

12.5 Self-Assessment Questions

6. What is the difference between object model, dynamic model and functional model?
7. What do you understand by functional model? What are the guidelines to construct it?
8. What do you understand by DFD? What are the different symbols used to construct it? Explain.
9. What do you understand by state diagram? Explain it using suitable examples.

10. Define the following terms: object, class, inheritance, link, association, aggregation.
11. If an association between classes has some attributes of its own, how will you implement it?

12.6 References/Suggested readings

45. Software Engineering concepts by Richard Fairley, Tata McGraw Hill.
46. An integrated approach to Software Engineering by Pankaj Jalote, Narosha Publishing houre.
47. Software Engineering by Sommerville, Pearson Education.
48. Software Engineering – A Practitioner's Approach by Roger S Pressman, McGraw-Hill.