Lesson No. : 1

Introduction to Software Engineering

1. 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?

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

Over runs, 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 The Software Crisis

There are lots of meanings of term software crisis. First of all we go in history and what was the scenario at that time.

Hardware Oriented World: There was more focus on the hardware and software as people were more focusing on that how instructions can be executed fast. Not on how bugs should be removed frequently.

Cost: Hardware vs. Software

Rapid increase in software cost and Rapid fall in hardware cost need more resources and well defined and systematic model for software development.

Inexperienced Developer:

Developer were hardly one year of experienced old so quite difficult for them to fulfill all coding based requirement.

Failures in Big project:

Big failure in project cost government billions of dollars like

- **1) the North East blackout in 2003**- has been major power system failures in the history of north which involves 100 power plants, 50 million customer faced problem, $ 6 million dollar financial loss.

- **2) Year 2000(Y2k**) refers to the widespread snags in processing date after the Year 2000. In 1960-80 when shortened the four digit date format like 1972 to a 2 digit format like 72 because of "Limited Memory. Because of that 2000 was shortened to 00. Million dollar were spent to handle this problem.

• **3) Arian- 5 Space Rocket:** In 1996, developed at cost of $7000 Million Dollarsover a period of 10 years was destroyed within less than 1 minutes after its launch. As there was software bugs in rocket guidance system.

• **4) "Dollar 924 lakhs":** In 1996, US bank credit accounts of nearly 800 customers with dollar 924 lakhs. This problem was due to main programming bug in the banking system.

• **5)1991 during Gulf War,** The USA use patriot missiles as a defense against Iraqi scud missile . However, patriot failed to hit the scud many times with cost life of 28 USA soldiers. In an inquiry it is found that a small bug had resulted in miscalculation of missile path.

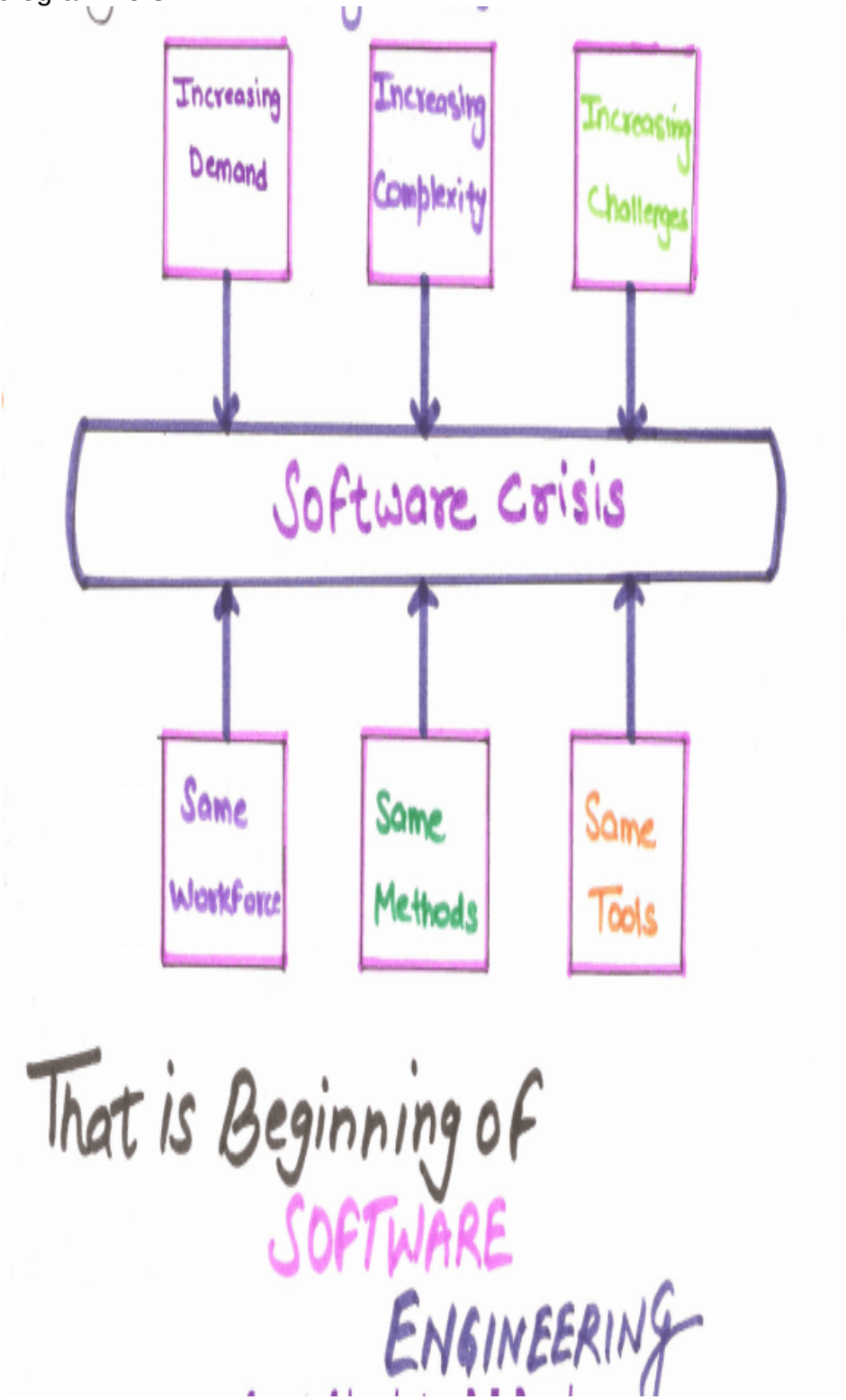
User changing requirement:

As user changes its requirement slightly as world and technology changing rapidly.

What is Software Crisis?

The difficulty of writing the code for a computer program which is correct and understandable is referred to as software crisis. Or Software crisis is also referred to as inability to hire enough qualified

programmers.



1.3 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.

1.4 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.

1.5 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.

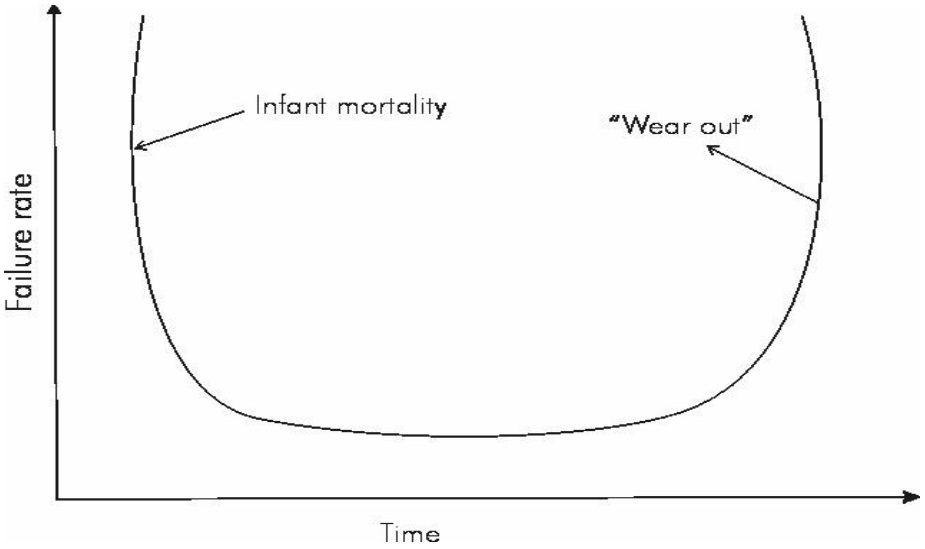


FIGURE 1.1 HARDWARE FAILURE CURVE

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.

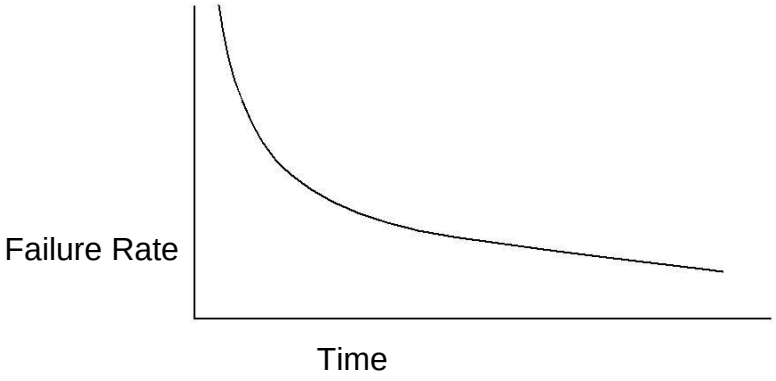


Figure 1.2 Failure curve for software

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.

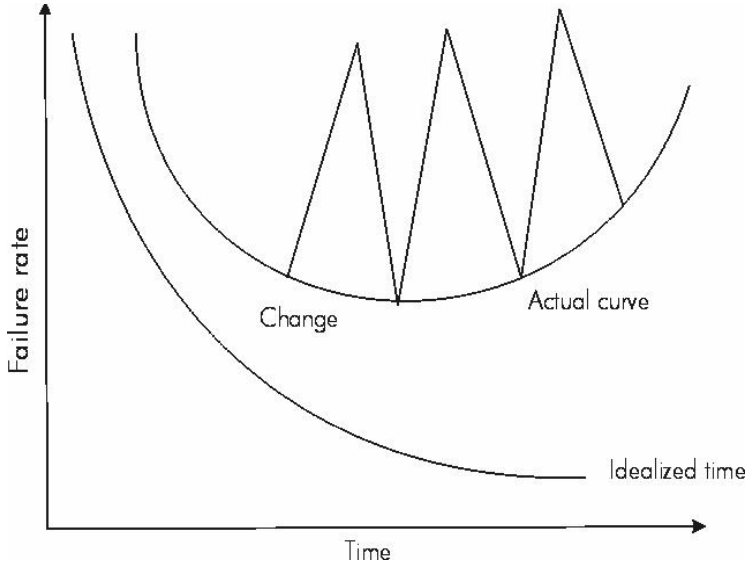


FIGURE 1.3 SOFTWARE IDEALIZED AND ACTUAL FAILURE CURVES

- **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 build 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 many 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.

Lesson -2 Software Metrics

2.1 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". 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.

2.2 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:

$T' = (n1N2\ (n1\log2n1 + n_2\log_2 n_2)\ \log_2 n) / 2n_2S$

Advantages of Halstead:

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

Drawbacks of Halstead

1. It depends on completed code.
2. 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 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.

2.4 Function Points (FP)

Function points are 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.

Lesson -3 Software life cycle model

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 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".

Usage of the waterfall model

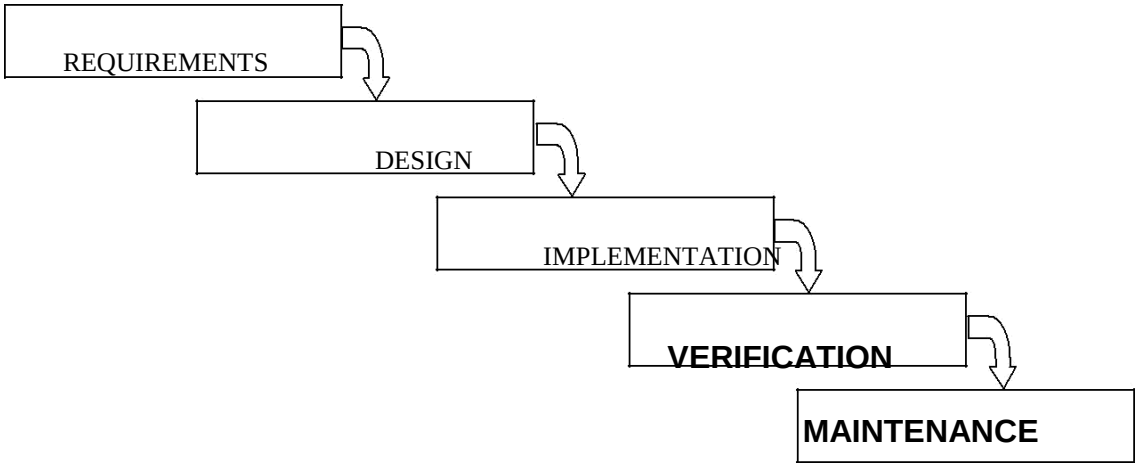


Figure 3.1 Water Fall Model

The unmodified "waterfall model". Progress flows from the top to the bottom, like a waterfall. In Royce's original waterfall model, the following phases are followed 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 is 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.

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.

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

  - Management Implications

  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.

  - Product Implications

  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

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.

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.

3.4 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.

3.5 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.

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.

3.6 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:

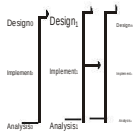


Figure 3.2 The Iterative Enhancement Model

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.

3.7 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.

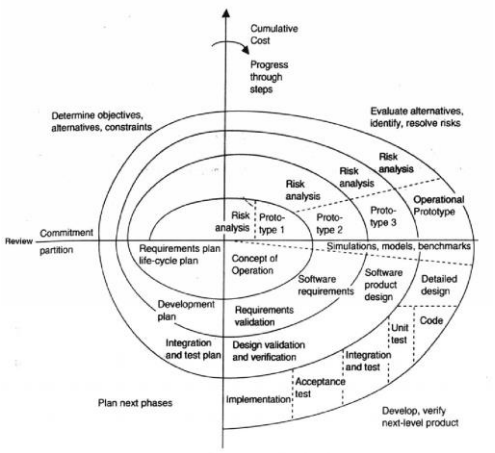


Figure 3.3 Boehm's spiral model of the software process

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.

Lesson -4 Project Planning

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 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.

4.3 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.

4.3.1 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 4.1 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 4.2.

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 4.2 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.

4.3.2 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 4.3.

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 4.3 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 4.4 Coefficients for intermediate COCOM

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 * $906^{0.38}$

4.3.3 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 4.5.

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 4.5 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

Lesson -5

Software Requirement Analysis &Specification

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. Software system requirements

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

5.2.1. 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.

5.2.2 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.

5.3 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:

- **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

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

5.4 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.

5.5 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.

Lesson-6 Software Design

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 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.
- Un maintainable 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 unturned 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.

6.3 Qualities of a Good Design

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

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.

6.4 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.

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 modelling, 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.

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.

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 understand ability 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 extraneous 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):

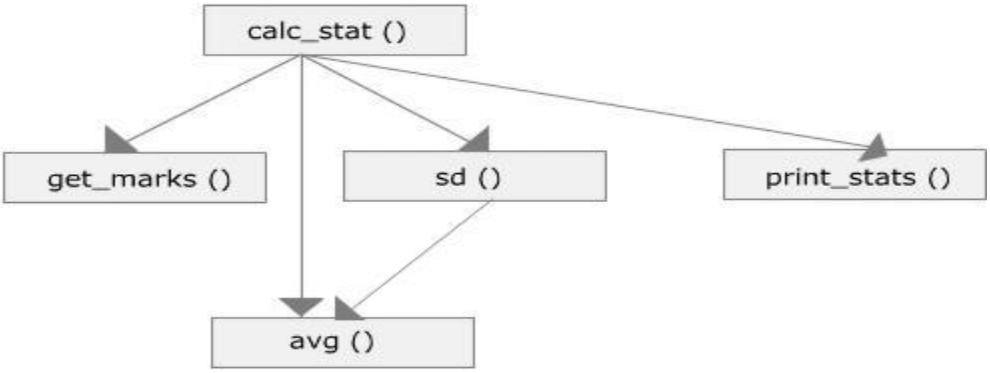


Figure 6.1

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;
}
```

Lesson-7 Design-II

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 modelling, modular decomposition. In a structured approach to design, the system is decomposed into a set of interacting functions.

7.2 Structured Programming

The goal of structured programming is to linearism 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:

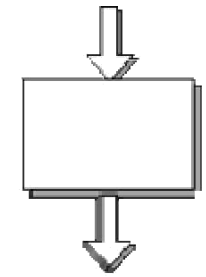
Structure Rule One: Code Block

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 7.1. 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 7.1



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 7.2.

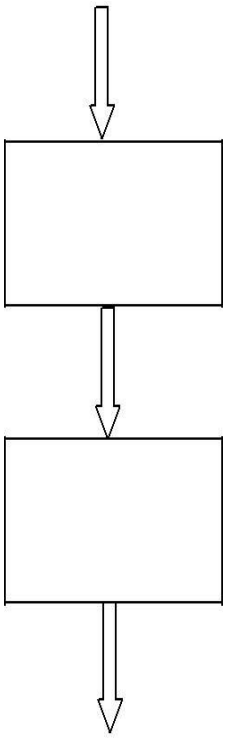


Figure 7.2 Rule 2: A sequence of code blocks is structured

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 7.3.

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.

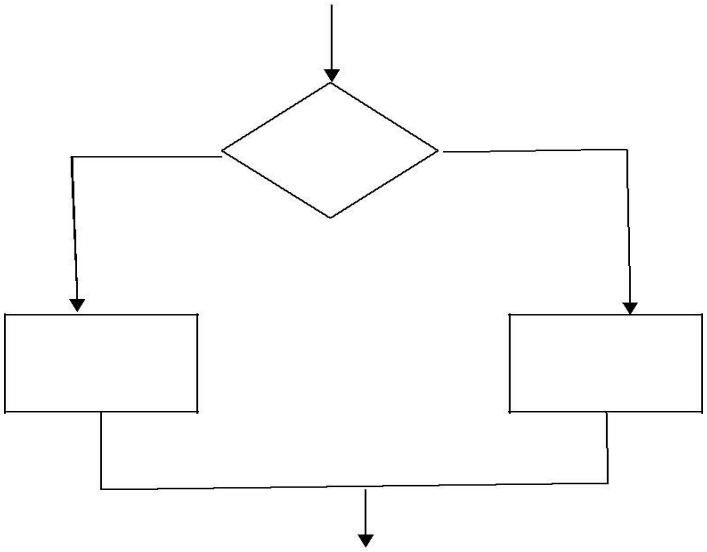


Figure 7.3 Rule 3: An alternation of code blocks is structured

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 7.4.

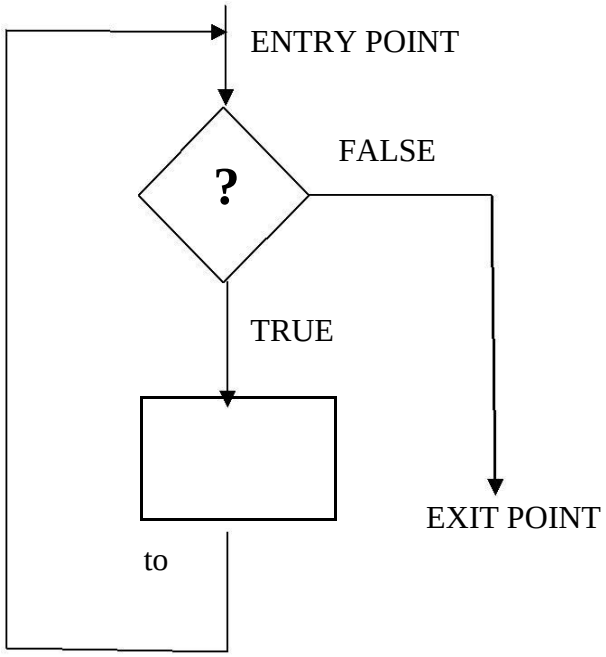


Figure 7.4 Rule 4: The iteration of code block is structured

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 7.5.

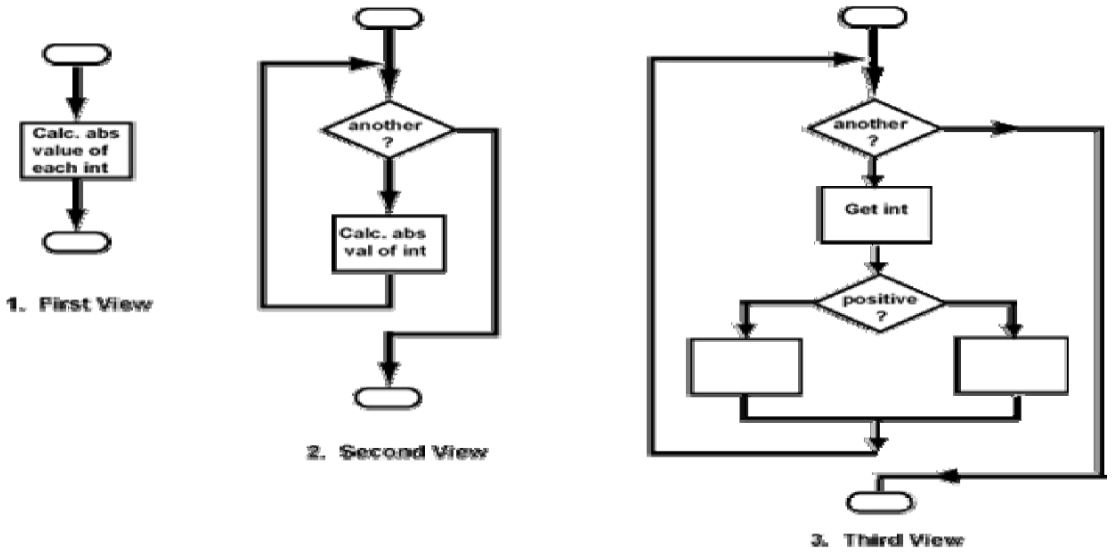


Figure 7.5

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

Lesson-8 Coding

8.0 Objectives

The objective of this lesson is to make the students familiar

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 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:

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.

- **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.

- **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.

- **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;
```

- **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.

- **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”.

- **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.

- **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.

- **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
```

- **Examine routines having more than five formal parameters**

Parameters are used to exchange the information among the functions or routines. Use of more than five formal parameters gives a feeling that probably

the function is complete. 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.

8.3 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.

- **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.

- **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;

- **Avoid 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
```

- **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.

- **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.

8.4 Software Verification and Validation 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.

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.

Lesson-9 Software 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.

9.2.1 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.

It 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.

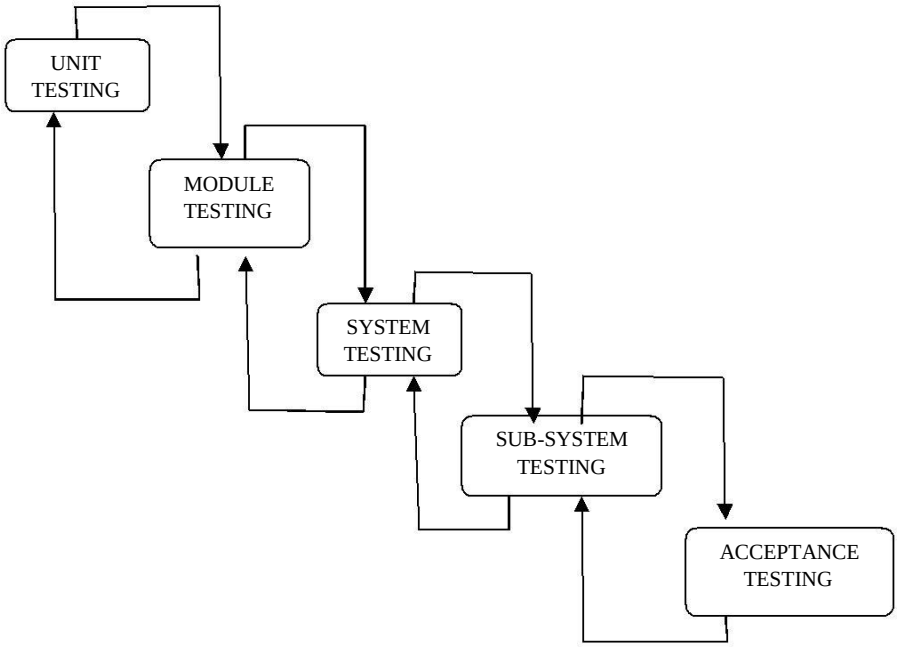


Figure 9.1 Test levels

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.

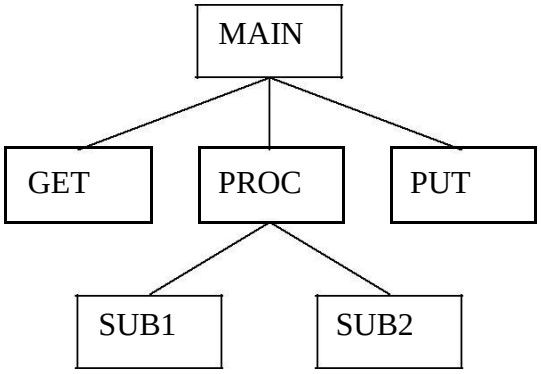


Figure 9.2

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 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.

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

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

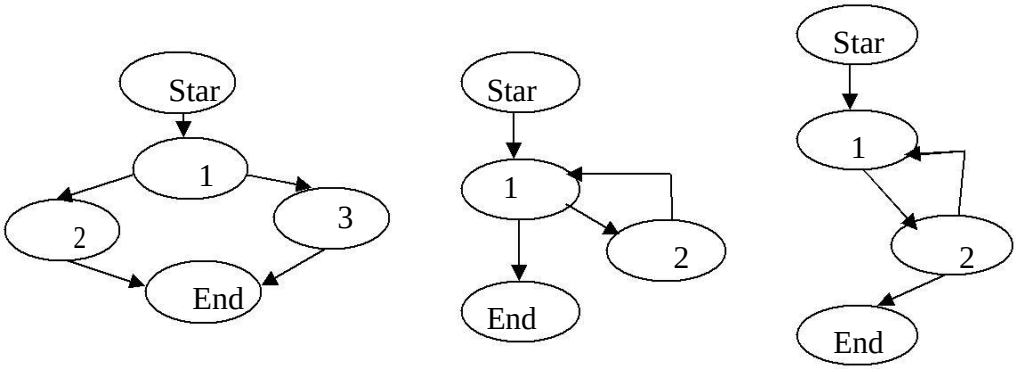
acknowledges that the order in which the btanches are executed during a test (the path traversed) is an important factor in determining the test outcome. So tester attempts to find a set of test cases that ensure the traversal of each logical path in the control flow graph.

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 9.3.1:



If loop **While Loop** **Do While Loop**

Figure 9.3.1

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 9.3.2. 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].

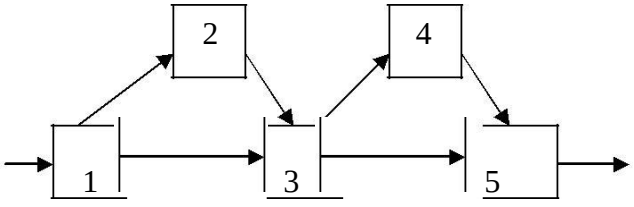


Figure 9.3.2

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.

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'.

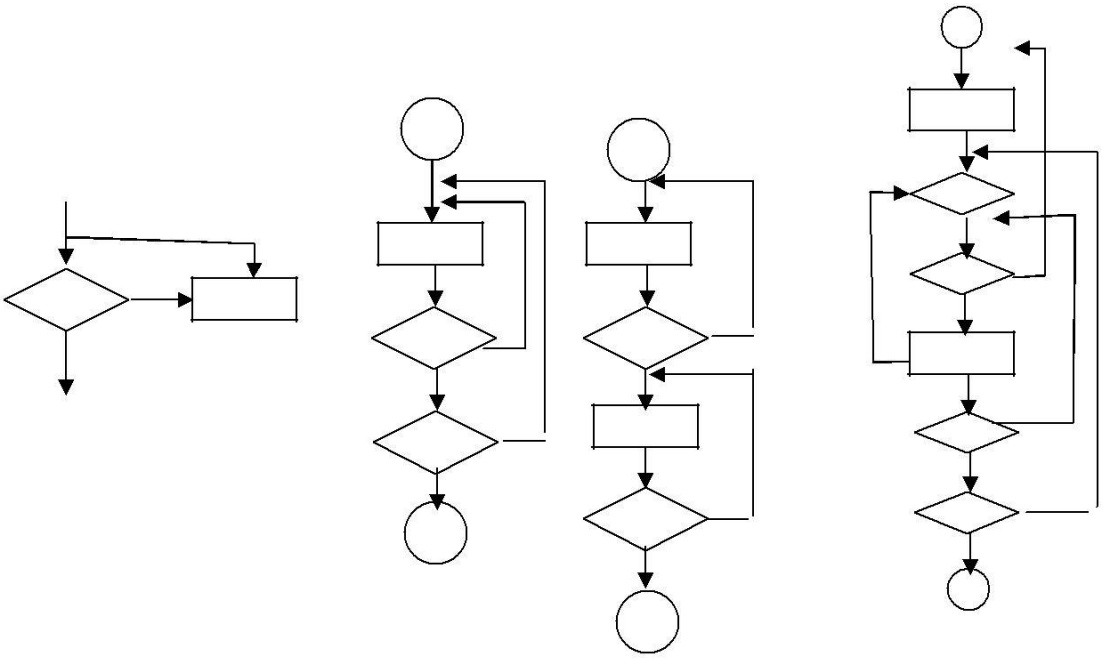
One simple data flow testing strategy is to require that every DU chain be covered at least once. This strategy is known as DU testing strategy.

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 9.3.3.

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 9.3.3

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.

9.4 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.

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:

<table>
<tr><td>-2, -1, 0</td><td>1,…..6</td><td>7,……12</td><td>13, 14, 15</td></tr>
<tr><td rowspan="2">Invalid Partition 1</td><td>P1</td><td>P2</td><td rowspan="2">Invalid Partition 2</td></tr>
<tr><td colspan="2">Valid Partition</td></tr>
</table>

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.

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.

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_1. 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 9.3.4. 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 9.3.1). 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).

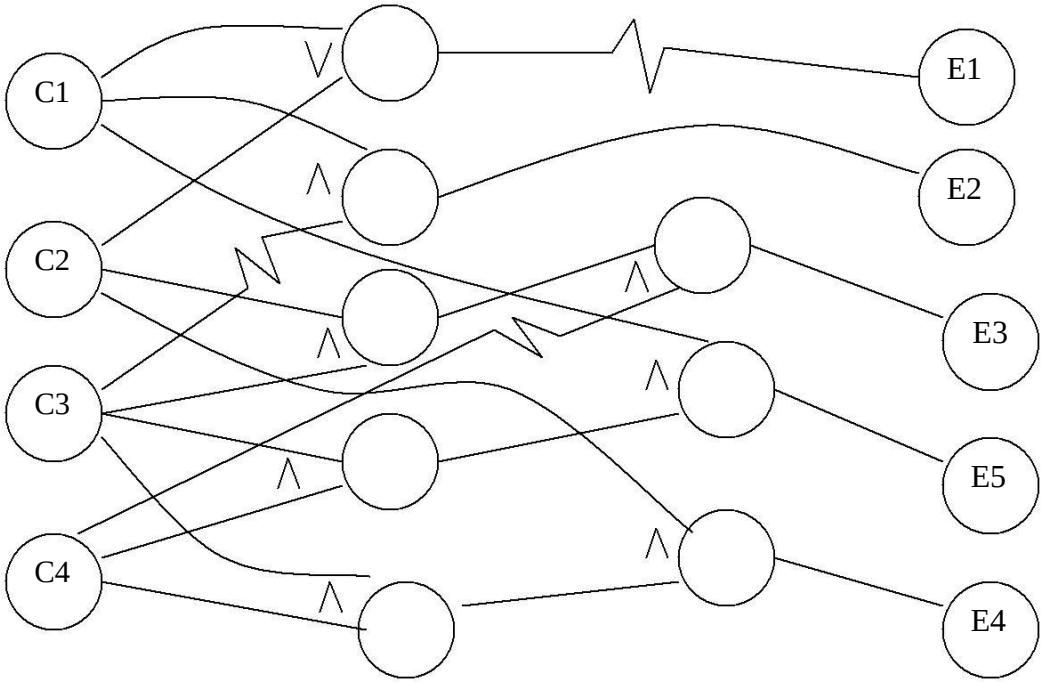


Figure 9.3.4 The Cause Effect Graph

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 9.3.1 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.

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.

Lesson -10 Software Maintenance

Software maintenance is a part of Software Development Life Cycle. Its main purpose is to modify and update software application after delivery to correct faults and to improve performance. Software is a model of the real world. When the real world changes, the software requires alteration wherever possible.

Basics of Software Maintenance

Software does not wear out or get tired. However, it needs to be upgraded and enhanced to meet new user requirements. For such modifications in the software system, software maintenance is performed. IEEE defines maintenance as 'a process of modifying a software system or component after delivery to correct faults, to improve performance or other attributes or to adapt the product to a changed environment.' The objective is to ensure that the software is able to accommodate changes after the system has been delivered and deployed.

To understand the concept of maintenance properly, let us consider an example of a car. When a car is 'used', its components wear out due to friction in the mechanical parts, unsuitable use, or by external conditions. The car owner solves the problem by changing its components when they become totally unserviceable and by using trained mechanics to handle complex faults during the car's lifetime. Occasionally, the owner gets the car serviced at a service station. This helps in preventing future wear and tear of the car. Similarly, in software engineering the software needs to be 'serviced' so that it is able to meet the changing environment (such as business and user needs) where it functions. This servicing of software is commonly referred to as software maintenance, which ensures that the software system continues to perform according to the user

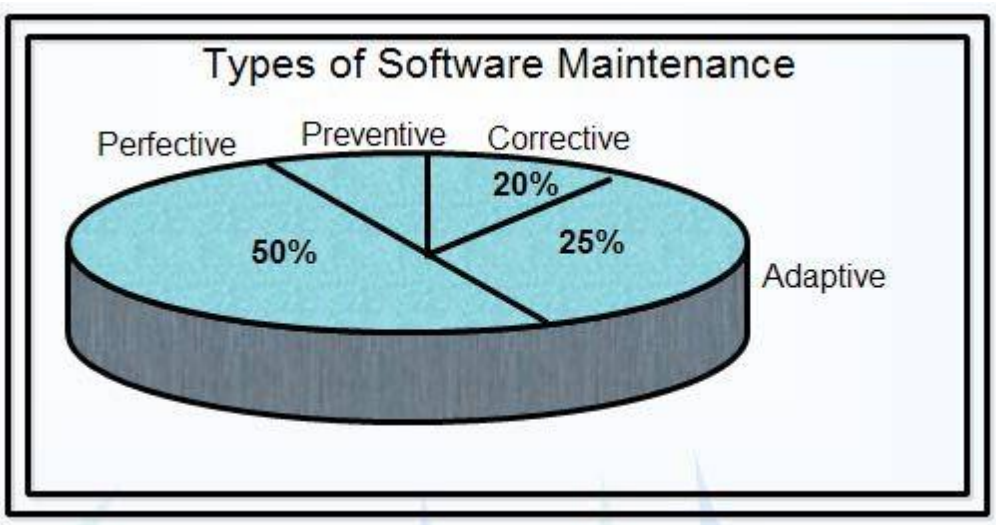
requirements even after the proposed changes have been incorporated. In addition, software maintenance serves the following purposes.

1. **Providing continuity of service:** The software maintenance process focuses on fixing errors, recovering from failures such as hardware failures or incompatibility of hardware with the software, and accommodating changes in the operating system and the hardware.

2. **Supporting mandatory upgrades:** Software maintenance supports up gradations, if required, in a software system. Up gradations may be required due to changes in government regulations or standards. For example, if a web-application system with multimedia capabilities has been developed, modification may be necessary in countries where screening of videos (over the Internet) is prohibited. The need for up gradations may also be felt to maintain competition with other software that exist in the same category.

3. **Improving the software to support user requirements:** Requirements may be requested to enhance the functionality of the software, to improve performance, or to customize data processing functions as desired by the user. Software maintenance provides a framework, using which all the requested changes can be accommodated.

4. **Facilitating future maintenance work:** Software maintenance also facilitates future maintenance work, which may include restructuring of the software code and the database used in the software.



Adaptive Maintenance

Adaptive maintenance is the implementation of changes in a part of the system, which has been affected by a change that occurred in some other part of the system. Adaptive maintenance consists of adapting software to changes in the environment such as the hardware or the operating system. The term environment in this context refers to the conditions and the influences which act (from outside) on the system. For example, business rules, work patterns, and government policies have a significant impact on the software system.

For instance, a government policy to use a single 'European currency' will have a significant effect on the software system. An acceptance of this change will require banks in various member countries to make significant changes in their software systems to accommodate this currency. Adaptive maintenance accounts for 25% of all the maintenance activities.

Perfective Maintenance

Perfective maintenance mainly deals with implementing new or changed user requirements. Perfective maintenance involves making functional enhancements to the system in addition to the activities to increase the system's performance even when the changes have not been suggested by faults. This includes enhancing both the function and efficiency of the code and changing the functionalities of the system as per the users' changing needs.

Examples of perfective maintenance include modifying the payroll program to incorporate a new union settlement and adding a new report in the sales analysis system. Perfective maintenance accounts for 50%, that is, the largest of all the maintenance activities.

Preventive Maintenance

Preventive maintenance involves performing activities to prevent the occurrence of errors. It tends to reduce the software complexity thereby improving program

understandability and increasing software maintainability. It comprises documentation updating, code optimization, and code restructuring. Documentation updating involves modifying the documents affected by the changes in order to correspond to the present state of the system. Code optimization involves modifying the programs for faster execution or efficient use of storage space. Code restructuring involves transforming the program structure for reducing the complexity in source code and making it easier to understand.

Preventive maintenance is limited to the maintenance organization only and no external requests are acquired for this type of maintenance. Preventive maintenance accounts for only 5% of all the maintenance activities.

Lesson -11 Software Quality and Maintenance

11.1 **Software Engineering | Capability maturity model (CMM)**

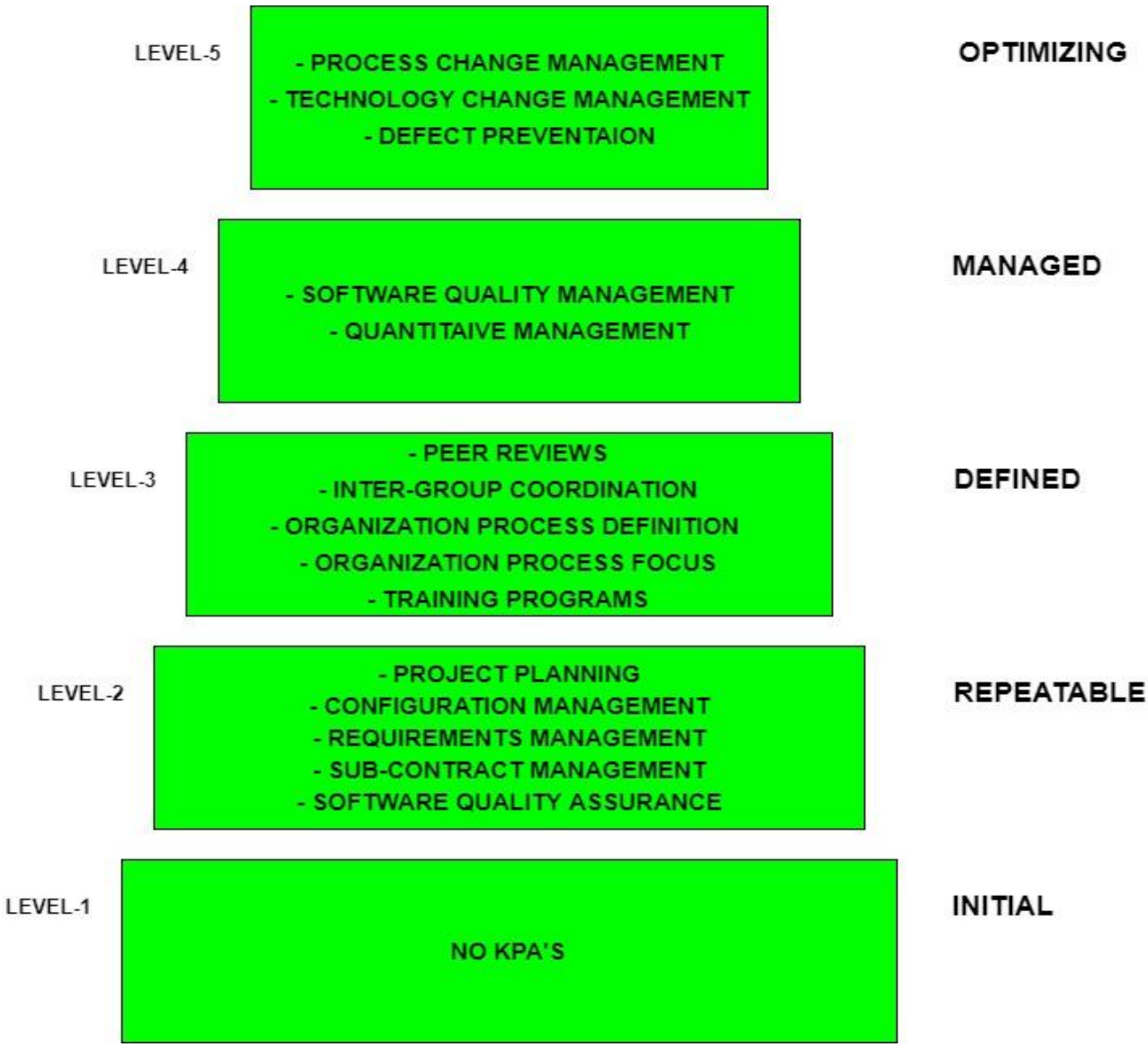
CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987.

- It is not a software process model. It is a framework which is used to analyse the approach and techniques followed by any organization to develop a software product.
- It also provides guidelines to further enhance the maturity of those software products.
- It is based on profound feedback and development practices adopted by the most successful organizations worldwide.
- This model describes a strategy that should be followed by moving through 5 different levels.
- Each level of maturity shows a process capability level. All the levels except level-1 are further described by Key Process Areas (KPA's).

Key Process Areas (KPA's):

Each of these KPA's defines the basic requirements that should be met by a software process in order to satisfy the KPA and achieve that level of maturity.

Conceptually, key process areas form the basis for management control of the software project and establish a context in which technical methods are applied, work products like models, documents, data, reports, etc. are produced, milestones are established, quality is ensured and change is properly managed.



Level-1: Initial –

- No KPA's defined.
- Processes followed are adhoc and immature and are not well defined.
- Unstable environment for software dvelopment.
- No basis for predicting product quality, time for completion, etc.

Level-2: Repeatable –

- Focuses on establishing basic project management policies.

- Experience with earlier projects is used for managing new similar natured projects.

KPA's:

- Project Planning- It includes defining resources required, goals, constraints, etc. for the project. It presents a detailed plan to be followed systematically for successful completion of a good quality software.
- Configuration Management- The focus is on maintaining the performance of the software product, including all its components, for the entire lifecycle.
- Requirements Management- It includes the management of customer reviews and feedback which result in some changes in the requirement set. It also consists of accommodation of those modified requirements.
- Subcontract Management- It focuses on the effective management of qualified software contractors i.e. it manages the parts of the software which are developed by third parties.
- Software Quality Assurance- It guarantees a good quality software product by following certain rules and quality standard guidelines while development.

Level-3: Defined –

- At this level, documentation of the standard guidelines and procedures takes place.
- It is a well defined integrated set of project specific software engineering and management processes.

KPA's:

- Peer Reviews- In this method, defects are removed by using a number of review methods like walkthroughs, inspections, buddy checks, etc.
- Intergroup Coordination- It consists of planned interactions between different development teams to ensure efficient and proper fulfilment of customer needs.
- Organization Process Definition- It's key focus is on the development and maintenance of the standard development processes.
- Organization Process Focus- It includes activities and practices that should be followed to improve the process capabilities of an organization.
- Training Programs- It focuses on the enhancement of knowledge and skills of the team members including the developers and ensuring an increase in work efficiency.

Level-4: Managed –

- At this stage, quantitative quality goals are set for the organization for software products as well as software processes.
- The measurements made help the organization to predict the product and process quality within some limits defined quantitatively.

KPA's:

- Software Quality Management- It includes the establishment of plans and strategies to develop a quantitative analysis and understanding of the product's quality.
- Quantitative Management- It focuses on controlling the project performance in a quantitative manner.

Level-5: Optimizing –

- This is the highest level of process maturity in CMM and focuses on continuous process improvement in the organization using quantitative feedback.
- Use of new tools, techniques and evaluation of software processes is done to prevent recurrence of known defects.

KPA's:

- Process Change Management- Its focus is on the continuous improvement of organization's software processes to improve productivity, quality and cycle time for the software product.
- Technology Change Management- It consists of identification and use of new technologies to improve product quality and decrease the product development time.
- Defect Prevention- It focuses on identification of causes of defects and to prevent them from recurring in future projects by improving project defined process.

11.2 ISO 9000

ISO 9000 is a set of international standards on quality management and quality assurance developed to help companies effectively document the quality system elements to be implemented to maintain an efficient quality system. They are not specific to any one industry and can be applied to organizations of any size.

ISO 9000 can help a company satisfy its customers, meet regulatory requirements, and achieve continual improvement. However, it should be

considered to be a first step, the base level of a quality system, not a complete guarantee of quality.

ISO 9000 principles of quality management

The ISO 9000:2015 and ISO 9001:2015 standards are based on seven quality management principles that senior management can apply for organizational improvement:

1. Customer focus

- ✓ Understand the needs of existing and future customers
- ✓ Align organizational objectives with customer needs and expectations
- ✓ Meet customer requirements
- ✓ Measure customer satisfaction
- ✓ Manage customer relationships
- ✓ Aim to exceed customer expectations

2. Leadership

- ✓ Establish a vision and direction for the organization
- ✓ Set challenging goals
- ✓ Model organizational values

- ✓ Establish trust
- ✓ Equip and empower employees
- ✓ Recognize employee contribution

3. Engagement of people

- ✓ Ensure that people’s abilities are used and valued
- ✓ Make people accountable
- ✓ Enable participation in continual improvement
- ✓ Evaluate individual performance
- ✓ Enable learning and knowledge sharing
- ✓ Enable open discussion of problems and constraints

4. Process approach

- ✓ Manage activities as processes
- ✓ Measure the capability of activities
- ✓ Identify linkages between activities
- ✓ Prioritize improvement opportunities
- ✓ Deploy resources effectively

5. Improvement

- ✓ Improve organizational performance and capabilities

- ✓ Align improvement activities
- ✓ Empower people to make improvements
- ✓ Measure improvement consistently
- ✓ Celebrate improvements

6. Evidence-based decision making

- ✓ Ensure the accessibility of accurate and reliable data
- ✓ Use appropriate methods to analyze data
- ✓ Make decisions based on analysis
- ✓ Balance data analysis with practical experience

7. Relationship management

- ✓ Identify and select suppliers to manage costs, optimize resources, and create value
- ✓ Establish relationships considering both the short and long term
- ✓ Share expertise, resources, information, and plans with partners
- ✓ Collaborate on improvement and development activities
- ✓ Recognize supplier successes

11.3 Six Sigma

Six Sigma is a highly disciplined process that helps us focus on developing and delivering near-perfect products and services.

Features of Six Sigma

- ✓ Six Sigma's aim is to eliminate waste and inefficiency, thereby increasing customer satisfaction by delivering what the customer is expecting.
- ✓ Six Sigma follows a structured methodology, and has defined roles for the participants.
- ✓ Six Sigma is a data driven methodology, and requires accurate data collection for the processes being analyzed.
- ✓ Six Sigma is about putting results on Financial Statements.
- ✓ Six Sigma is a business-driven, multi-dimensional structured approach for –
    - Improving Processes
    - Lowering Defects
    - Reducing process variability
    - Reducing costs
    - Increasing customer satisfaction
    - Increased profits

The word *Sigma* is a statistical term that measures how far a given process deviates from perfection.

The central idea behind Six Sigma: If you can measure how many "defects" you have in a process, you can systematically figure out how to eliminate them and get as close to "zero defects" as possible and specifically it means a failure rate of 3.4 parts per million or 99.9997% perfect.

Key Concepts of Six Sigma

At its core, Six Sigma revolves around a few key concepts.

- **Critical to Quality** − Attributes most important to the customer.
- **Defect** − Failing to deliver what the customer wants.
- **Process Capability** − What your process can deliver.
- **Variation** − What the customer sees and feels.
- **Stable Operations** − Ensuring consistent, predictable processes to improve what the customer sees and feels.
- **Design for Six Sigma** − Designing to meet customer needs and process capability.

Our Customers Feel the Variance, Not the Mean. So Six Sigma focuses first on reducing process variation and then on improving the process capability.

Myths about Six Sigma

There are several myths and misunderstandings surrounding Six Sigma. Some of them few are given below −

- Six Sigma is only concerned with reducing defects.
- Six Sigma is a process for production or engineering.

- Six Sigma cannot be applied to engineering activities.
- Six Sigma uses difficult-to-understand statistics.
- Six Sigma is just training.

Benefits of Six Sigma

Six Sigma offers six major benefits that attract companies −

- Generates sustained success
- Sets a performance goal for everyone
- Enhances value to customers
- Accelerates the rate of improvement
- Promotes learning and cross-pollination
- Executes strategic change

Origin of Six Sigma

- Six Sigma originated at Motorola in the early 1980s, in response to achieving 10X reduction in product-failure levels in 5 years.
- Engineer Bill Smith invented Six Sigma, but died of a heart attack in the Motorola cafeteria in 1993, never knowing the scope of the craze and controversy he had touched off.
- Six Sigma is based on various quality management theories (e.g. Deming's 14 point for management, Juran's 10 steps on achieving quality).

11.4 Configuration Management

Software configuration management (SCM) is a software engineering discipline consisting of standard processes and techniques often used by organizations to manage the changes introduced to its software products. SCM helps in identifying individual elements and configurations, tracking changes, and version selection, control, and baselining.

SCM is also known as software control management. SCM aims to control changes introduced to large complex software systems through reliable version selection and version control.

The SCM system has the following advantages:

- Reduced redundant work.
- Effective management of simultaneous updates.
- Avoids configuration-related problems.
- Facilitates team coordination.
- Helps in building management; managing tools used in builds.
- Defect tracking: It ensures that every defect has traceability back to its source.