LECTURE NOTES

ON

SOFTWARE ENGINEERING

Contents

UNIT-1

INTRODUCTION TO SOFTWARE ENGINEERING

1.1 BASIC TERMINOLOGY:

The term ***software engineering*** is composed of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

So, we can define ***software engineering*** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

IEEE defines software engineering as:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: ***abstraction*** and ***decomposition*****.**

The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem. The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

1.2 NEED OF SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software -** It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

- **Quality Management-** Better process of software development provides better and quality software product.

1.3 CHARACTERISTICS OF GOOD SOFTWARE

Software

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

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability

- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

1.4 SOFTWARE CHARACTERISTICS

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

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

Software is engineered or constructed

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.

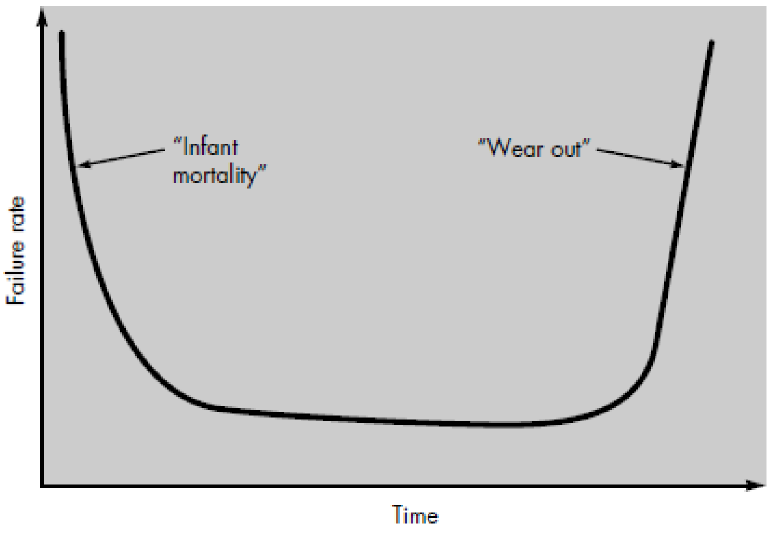


Figure 1.1: Failure curve for hardware

2. Software doesn't "wear out."

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a 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 effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Figure below. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate! This seeming contradiction can best be explained by considering the “actual curve” shown in Figure below. During its life, software will undergo change (maintenance).

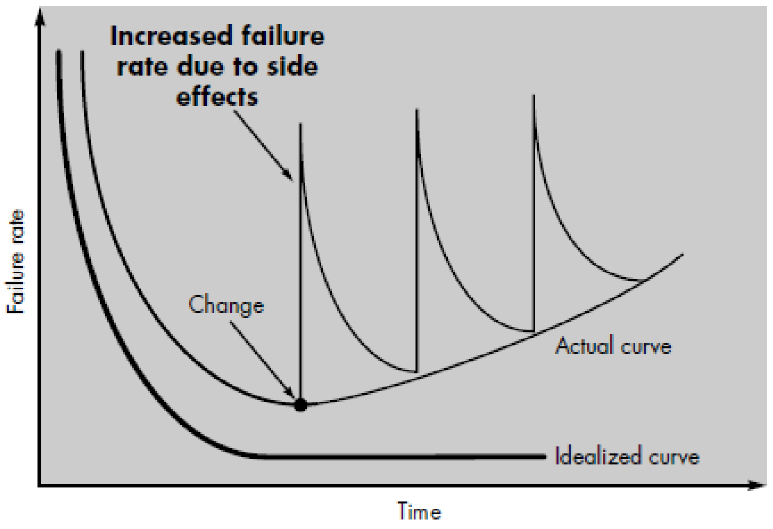


Figure 1.2: Idealized and actual failure curves for software

As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.2. Before the curve can return to the original steady- state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change. Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

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

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

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. 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.

1.5 SOFTWARE CRISIS

Software crisis is the situation resulted due to the catastrophic failure of software development that leads to the incomplete and degrading performance of software products. The term Software crisis was coined in the year 1968 AD. Software crisis is the situation resulted due to the rapid increase in computer power and the complexity of the problems that could not be tackled.

Causes of software crisis:

1) Due to the Projects running over-budget

2) Due to the Projects running over-time

3) Software was very inefficient and was of low quality

4) Software often did not meet requirements

5) Code was difficult to maintain and Projects were a mess and unmanageable.

The problems like Aircraft failure, Destruction of large plant due to software error are not the examples of software crisis. They are just the software failure problems.

1.6 SOFTWARE PROCESSES and CHARACTERISTICS

The process that deals with the technical and management issues of software development is called a **software process**. A software development project must have at least development activities and project management activities. The fundamental objectives of a process are the same as that of software engineering (after all, the process is the main vehicle of satisfying the software engineering objectives), viz. optimality and scalability.

Optimality means that the process should be able to produce high-quality software at low cost, and scalability means that it should also be applicable for large software projects. To achieve these objectives, a process should have some properties. **Predictability** of a process determines how accurately the outcome of following a process in a project can be predicted before the project is completed. Predictability can be considered a fundamental property of any process, In fact, if a process is not predictable, it is of limited use.

One of the important objectives of the development project should be to produce software that is easy to maintain. And the process should be such that it ensures this maintainability. Testing consumes the most resources during development. Underestimating the testing effort often causes the planners to allocate insufficient resources for testing, which, in turn, results in unreliable software or schedule slippage.

The goal of the process should not be to reduce the effort of design and coding, but to reduce the cost of maintenance. Both testing and maintenance depend heavily on the design and coding of software, and these costs can be considerably reduced if the software is designed and coded to make testing and maintenance easier. Hence, during the early phases of the development process the prime issues should be "can it be easily tested" and "can it be easily modified". Errors can occur at any stage during development.

However error detection and correction should be a continuous process that is done throughout software development. Detecting errors soon after they have been introduced is clearly an objective that should be supported by the process. A process is also not a static entity.

As the productivity (and hence the cost of a project) and quality are determined largely by the process to satisfy the engineering objectives of quality improvement and cost reduction, the software process must be improved. Having process improvement as a basic goal of the software process implies that the software process used is such that is supports its improvement.

1.7 SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

1.7.1 THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.

When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example.

Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure. A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

1.7.2 DIFFERENT LIFE CYCLE MODELS

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

1.7.2.1 CLASSICAL WATERFALL MODEL

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases as shown in fig.2.1 (next page)

1. **Feasibility study** - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.

- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

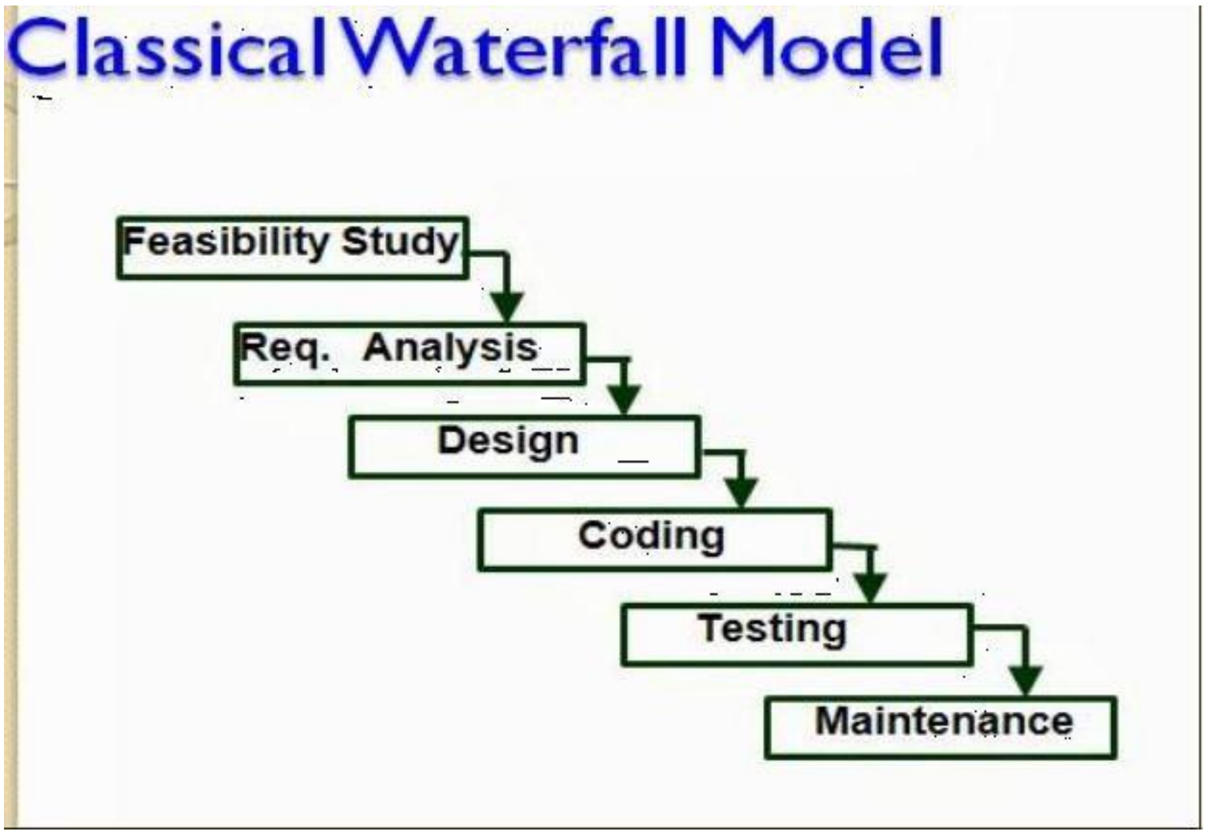


Fig. 1.3

2. **Requirements analysis and specification: -** The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

    - Requirements gathering and analysis
    - Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system.

Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

3. **Design: -** The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

- **Traditional design approach -**Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

- **Object-oriented design approach** -In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

4. **Coding and unit testing:-**The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.
5. **Integration and system testing: -**Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

- α – testing: It is the system testing performed by the development team.
- β –testing: It is the system testing performed by a friendly set of customers.
- Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed,

specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

6. **Maintenance: -**Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software

product to its maintenance effort is roughly in the 40:60 ratios. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

Shortcomings of the classical waterfall model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle.

The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle.

For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

1.7.2.2 ITERATIVE WATERFALL MODEL

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model. Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules. The block diagram of Iterative Waterfall model is shown in the fig. 1.2

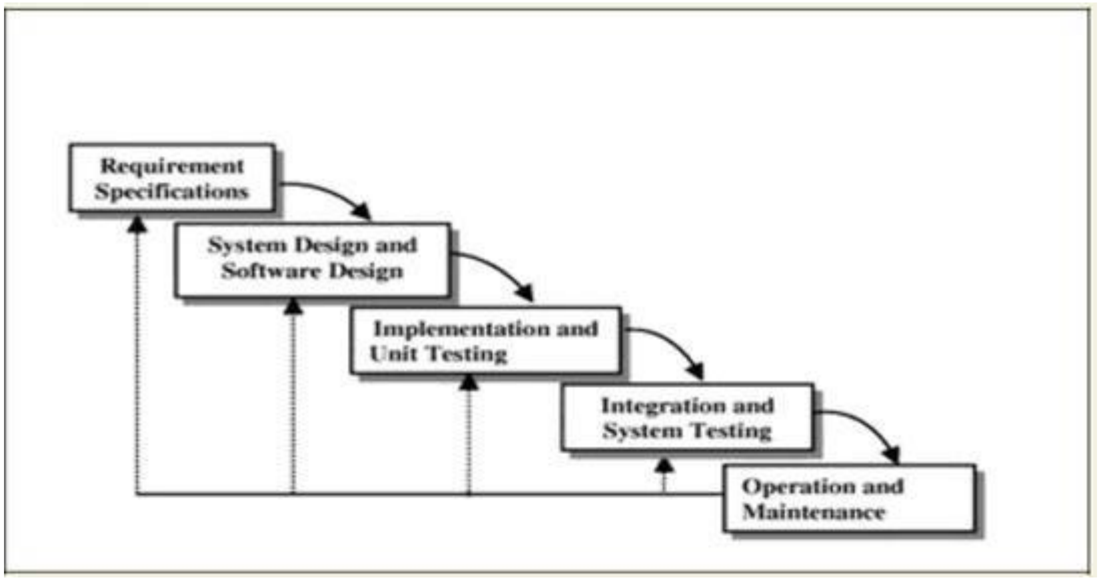

Requirement Specifications
System Design and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Fig. 1.4 Iterative Waterfall Model

1.7.2.3 PROTOTYPING MODEL

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions.

The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system. The block diagram of Prototyping model is shown in Fig. 1.3

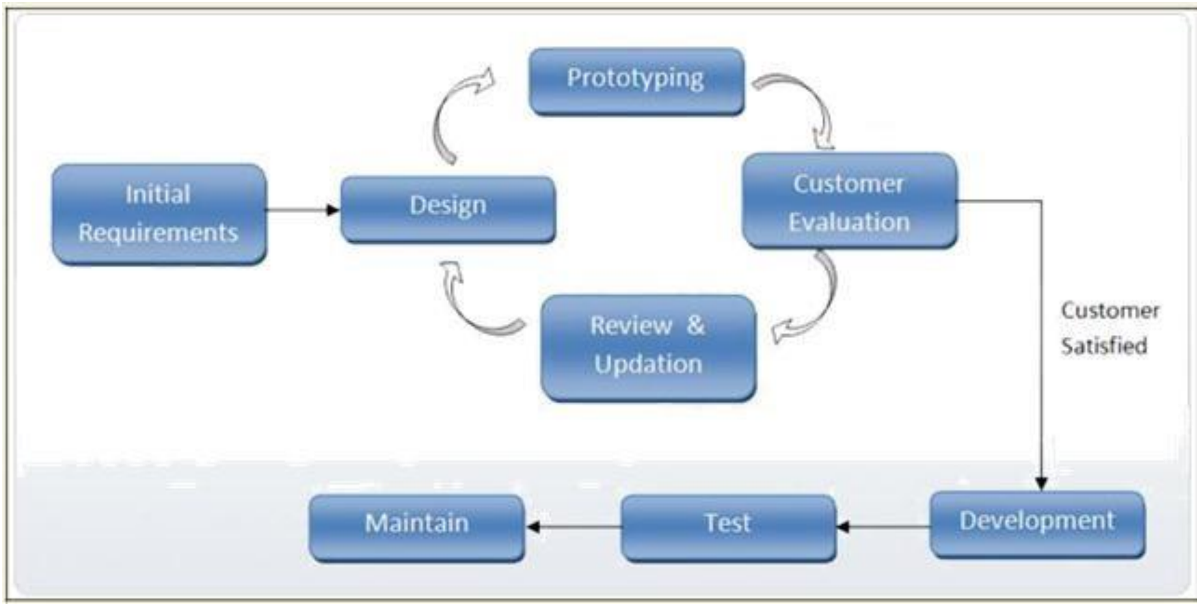


Fig. 1.5 Prototype Model

Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product. A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

A prototype of the actual product is preferred in situations such as:

• User requirements are not complete

• Technical issues are not clear

1.7.2.4 EVOLUTIONARY MODEL

It is also called *successive versions model* or *incremental model*. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

- Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.
- Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Advantages:

- User gets a chance to experiment partially developed system
- Reduce the error because the core modules get tested

thoroughly. Disadvantages:

- It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.

The block diagram of Evolutionary Model is shown in the Fig. 1.4

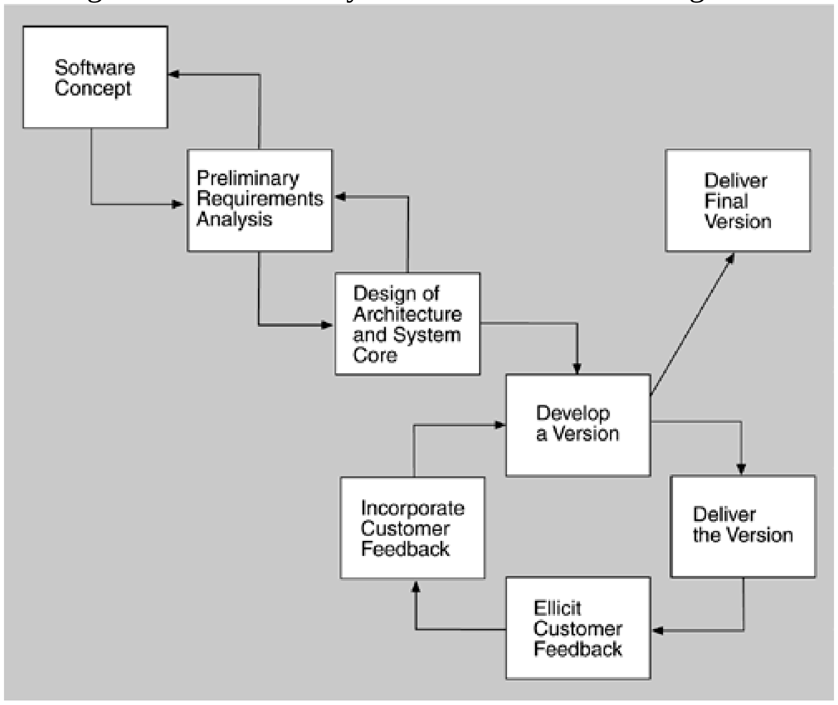

Software Concept
Preliminary Requirements Analysis
Design of Architecture and System Core
Develop a Version
Deliver Final Version
Deliver the Version
Ellicit Customer Feedback
Incorporate Customer Feedback

Fig. 1.6 Evolutionary Model

1.7.2.5 SPIRAL MODEL

The Spiral model of software development is shown in fig. 1.5. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 1.5. The following activities are carried out during each phase of a spiral model.

First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the phase.Examine the risks associated with these objectives

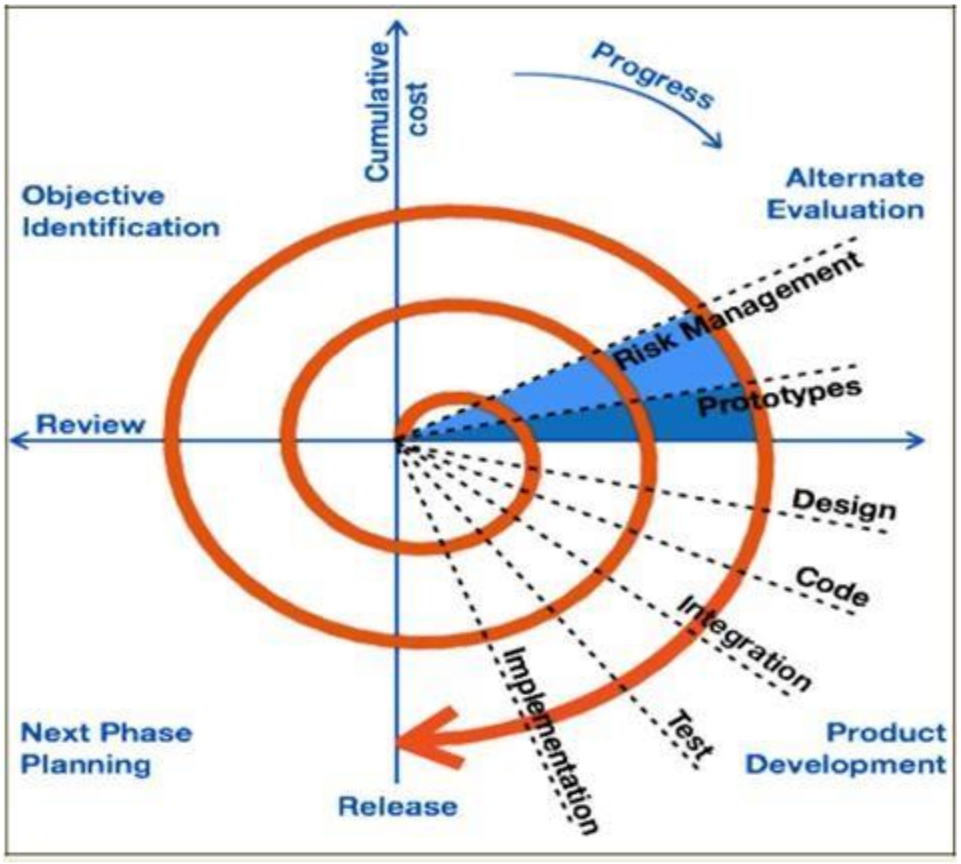


Fig. 1.7 Spiral Model

Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.

- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration around the spiral.

- Progressively more complete version of the software gets built with each iteration around the spiral.

Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

Comparison of different life-cycle models

The **classical waterfall model** can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The **iterative waterfall model** is probably the most widely used software development model evolved so far. This model is simple to understand and use. However this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The **prototyping model** is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The **evolutionary approach** is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The **spiral model** is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment.

On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

1.8 REQUIREMENT ANALYSIS AND SPECIFICATION

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as ***system analysts***.

The analyst starts *requirements gathering and analysis* activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?

- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness. When the analyst detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.

1.9 REQUIREMENT ENGINEERING

The process to gather the software requirements from client, analyze and document them is known as requirement engineering.

The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

1.9.1 REQUIREMENT ENGINEERING PROCESS

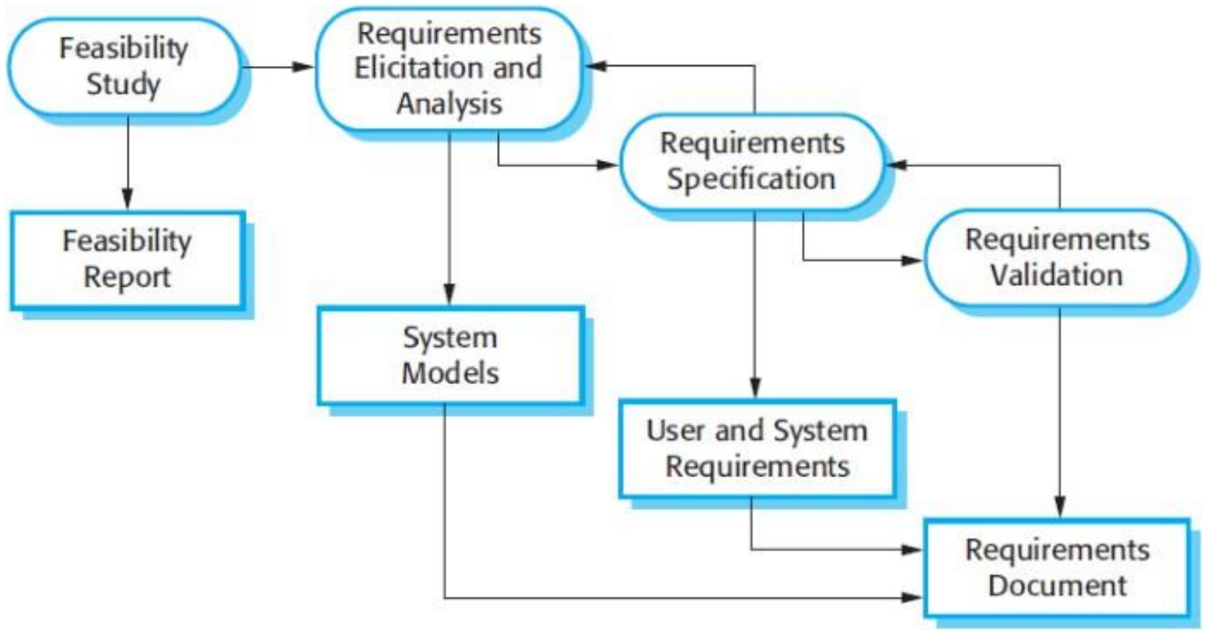


Fig. 1.8 Requirement Engineering Process

It is a four step process, which includes –

- Feasibility Study
- Requirement Gathering
- Software Requirement Specification
- Software Requirement Validation

Feasibility study

When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.

Referencing to this information, the analysts does a detailed study about whether the desired system and its functionality are feasible to develop.

This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Requirement Gathering

If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

Software Requirement Specification

SRS is a document created by system analyst after the requirements are collected from various stakeholders. SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

Software Requirement Validation

After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions -

- If they can be practically implemented
- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

1.10 REQUIREMENT ELICITATION PROCESS

Requirement elicitation process can be depicted using the following diagram:

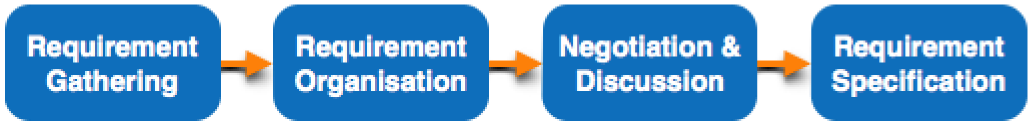


Fig. 1.9

- **Requirements gathering -** The developers discuss with the client and end users and know their expectations from the software.

- **Organizing Requirements** - The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.

  The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.

- **Documentation** - All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

1.10.1 Requirement Elicitation Techniques

Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others who have a stake in the software system development.

There are various ways to discover requirements

Interviews

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- Structured (closed) interviews, where every single information to gather is decided in advance, they follow pattern and matter of discussion firmly.
- Non-structured (open) interviews, where information to gather is not decided in advance, more flexible and less biased.
- Oral interviews
- Written interviews
- One-to-one interviews which are held between two persons across the table.
- Group interviews which are held between groups of participants. They help to uncover any missing requirement as numerous people are involved.

Surveys

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

Questionnaires

A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.

A shortcoming of this technique is, if an option for some issue is not mentioned in the questionnaire, the issue might be left unattended.

Task analysis

Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

Domain Analysis

Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

Brainstorming

An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

Prototyping

Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps giving better idea of requirements. If there is no software installed at client's end for developer's reference and the client is not aware of its own requirements, the developer creates a prototype based on initially mentioned requirements. The prototype is shown to the client and the feedback is noted. The client feedback serves as an input for requirement gathering.

Observation

Team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. They observe the workflow at client's end and how execution problems are dealt. The team itself draws some conclusions which aid to form requirements expected from the software.

1.10.2 SOME MORE ELICITATION TECHNIQUES

1.10.2.1 FACILITATED APPLICATION SPECIFICATION TECHNIQUE (FAST):

It's objective is to bridge the expectation gap – difference between what the developers think they are supposed to build and what customers think they are going to get. A team oriented approach is developed for requirements gathering. Each attendee is asked to make a list of objects that are-

1. Part of the environment that surrounds the system
2. Produced by the system
3. Used by the system

Each participant prepares his/her list, different lists are then combined, redundant entries are eliminated, team is divided into smaller sub-teams to develop mini-specifications and finally a draft of specifications is written down using all the inputs from the meeting.

1.10.2.2 QUALITY FUNCTION DEPLOYMENT (QFD):

In this technique, customer's satisfaction is the prime concern; hence it gives more stress on the requirements which are of more value to the customer. Three types of requirements are:

- **Normal requirements –** In this the objective and goals of the proposed software are discussed with the customer. Example – normal requirements for a result management system may be entry of marks, calculation of results etc
- **Expected requirements –** These requirements are so obvious that the customer need not explicitly state them. Example – protection from unauthorised access.
- **Exciting requirements –** It includes features that are beyond customer's expectations and prove to be very satisfying when present. Example – when an unauthorised access is detected, it should backup and shutdown all processes.

The major steps involved in this procedure are –

1. Identify all the stakeholders, eg. Users, developers, customers etc
2. List out all requirements from customer.
3. A value indicating degree of importance is assigned to each requirement.
4. In the end the final list of requirements is categorised as –

- It is possible to achieve
- It should be deferred and the reason for it

- It is impossible to achieve and should be dropped off.

1.10.2.3 USE CASE APPROACH:

This technique combines text and pictures to provide a better understanding of the requirements. The use cases describe the 'what', of a system and not 'how'. Hence they only give a functional view of the system.

The components of the use case design include three major things – Actor, Use cases, use case diagram.

1. **Actor –** It is the external agent that lies outside the system but interacts with it in some way. An actor maybe a person, machine etc. It is represented as a stick figure. Actors can be primary actors or secondary actors.

- Primary actors – It requires assistance from the system to achieve a goal.
- Secondary actor – It is an actor from which the system needs assistance.

2. **Use cases –** They describe the sequence of interactions between actors and the system. They capture who(actors) do what(interaction) with the system. A complete set of use cases specifies all possible ways to use the system.
3. **Use case diagram –** A use case diiagram graphically represents what happens when an actor interacts with a system. It captures the functional aspect of the system.

- A stick figure is used to represent an actor.
- An oval is used to represent a use case.
- A line is used to represent a relationship between an actor and a use case.

1.10.2.4 DATA FLOW DIAGRAM

Data flow diagram is a graphical representation of data flow in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

Types of DFD

Data Flow Diagrams are either Logical or Physical.

- **Logical DFD** - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.

- **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components:

DFD can represent Source, destination, storage and flow of data using the following set of components:

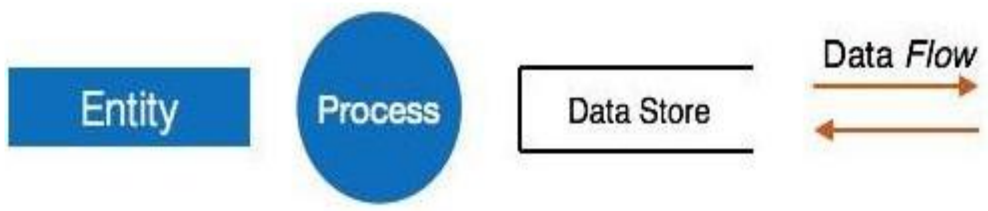


Fig. 1.10

- **Entities** - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.

- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.

- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.

- **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.

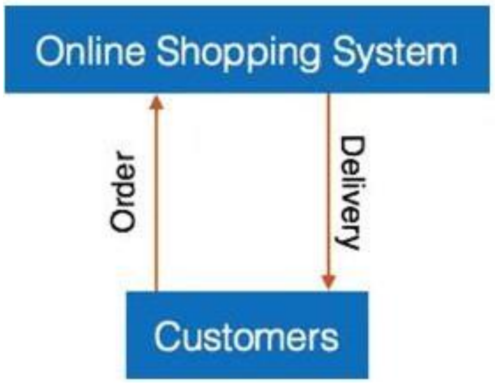


Fig. 1.11 Level 0 DFD of Online shopping

- **Level 1 -** The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.

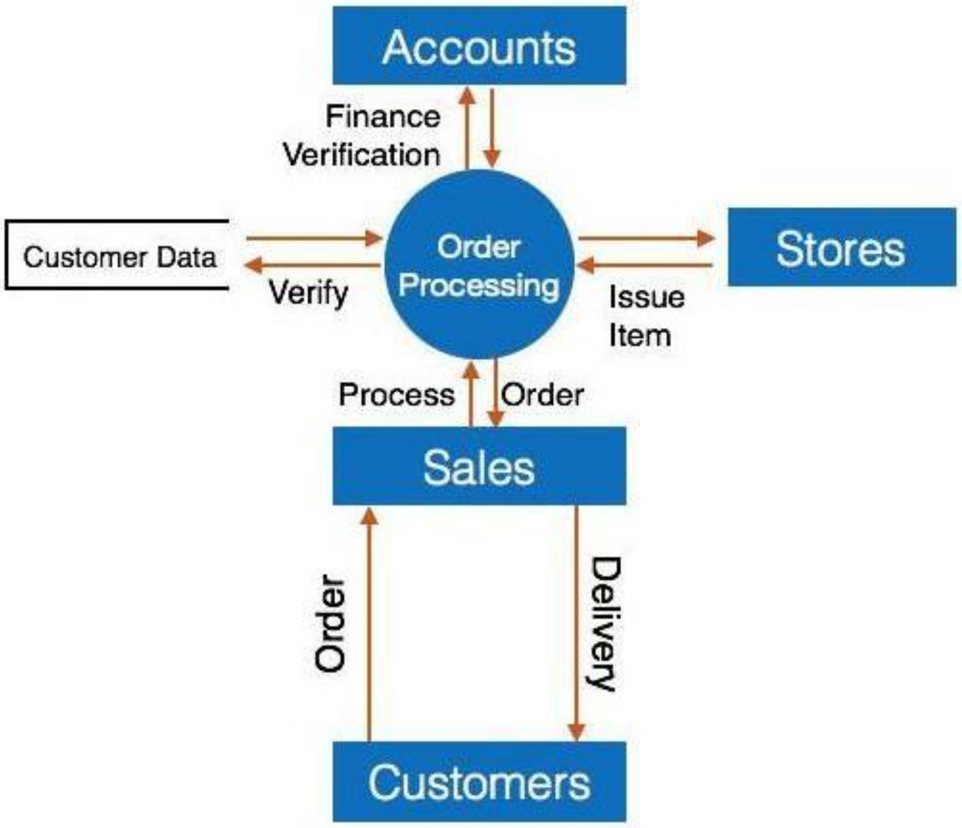


Fig. 1.12 Level 1 DFD of Online shopping

- **Level 2** - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

Structure Chart

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD. A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks which are used to design structure charts are the following:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from on one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

Structure Chart vs. Flow Chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of the software from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.

- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

1.10.2.5 DECISION TABLES

A Decision table represents conditions and the respective actions to be taken to address them, in a structured tabular format. It is a powerful tool to debug and prevent errors. It helps group similar information into a single table and then by combining tables it delivers easy and convenient decision-making.

Creating Decision Table

To create the decision table, the developer must follow basic four steps:

- Identify all possible conditions to be addressed
- Determine actions for all identified conditions
- Create Maximum possible rules
- Define action for each rule

Decision Tables should be verified by end-users and can lately be simplified by eliminating duplicate rules and actions. Example: Let us take a simple example of day-to-day problem with our Internet connectivity. We begin by identifying all problems that can arise while starting the internet and their respective possible solutions. All possible problems are listed under column conditions and the prospective actions under column actions.

	Conditions/Actions	**Rules**							
	Shows Connected	N	N	N	N	Y	Y	Y	Y
Conditions	Ping is Working	N	N	Y	Y	N	N	Y	Y
	Opens Website	Y	N	Y	N	Y	N	Y	N

Actions	Check network cable	X							
	Check internet router	X				X	X	X	
	Restart Web Browser							X	
	Contact Service provider		X	X	X	X	X	X	

Table 1.1: Decision Table – In-house Internet Troubleshooting

1.9.2.6 ENTITY RELATIONSHIP MODEL

Entity-Relationship model is a type of database model based on the notion of real world entities and relationship among them. We can map real world scenario onto ER database model. ER Model creates a set of entities with their attributes, a set of constraints and relation among them.

ER Model is best used for the conceptual design of database. ER Model can be represented as follows:

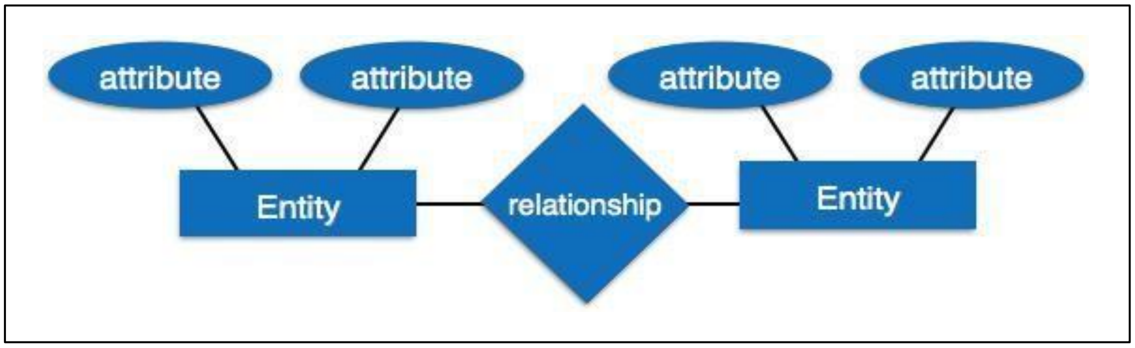


Fig 1.13 : ER Model

- **Entity** - An entity in ER Model is a real world being, which has some properties called ***attributes***. Every attribute is defined by its corresponding set of values, called ***domain***.

For example, consider a school database. Here, a student is an entity. Student has various attributes like name, id, age and class etc.

- **Relationship** - The logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of associations between two entities.

Mapping cardinalities:

- one to one
- one to many
- many to one
- many to many

EXAMPLE OF ER- DIAGRAM

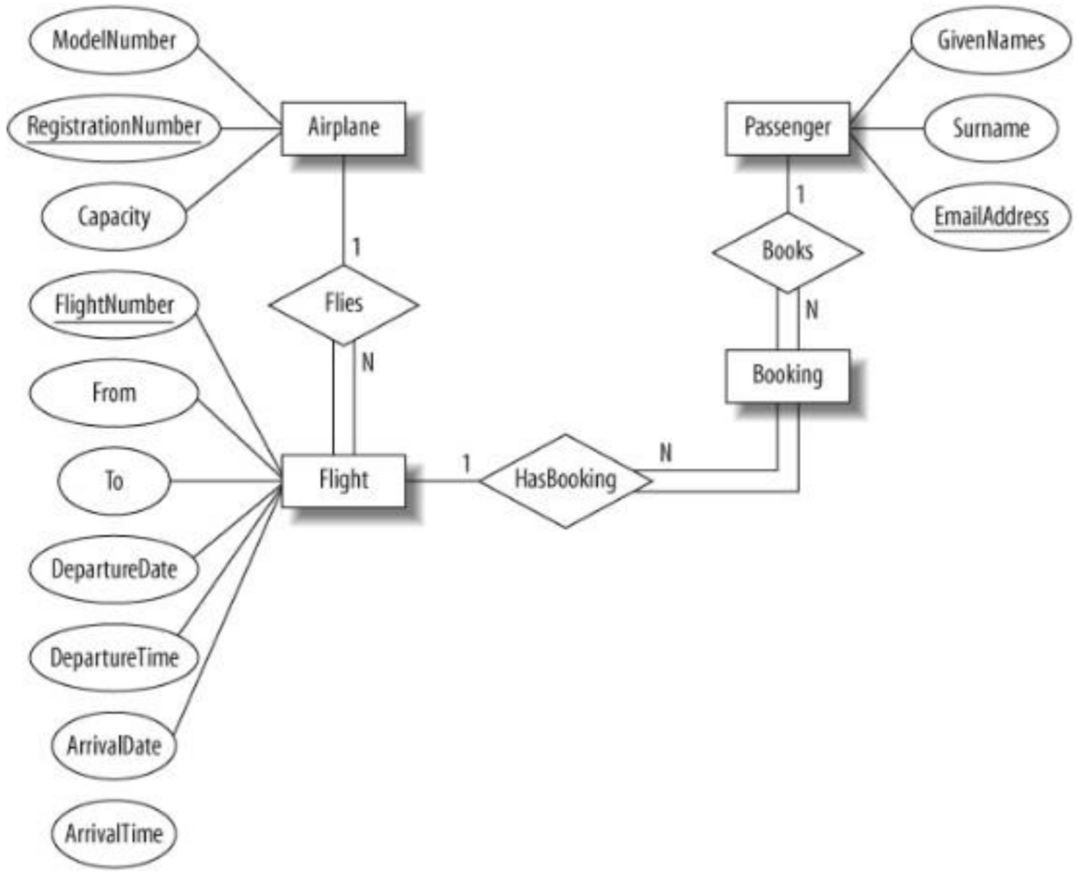


Fig. 1.14 The ER diagram of the flight database

1.10 DATA DICTIONARY

Data dictionary is the centralized collection of information about data. It stores meaning and origin of data, its relationship with other data, data format for usage etc. Data dictionary has rigorous definitions of all names in order to facilitate user and software designers.

Data dictionary is often referenced as meta-data (data about data) repository. It is created along with DFD (Data Flow Diagram) model of software program and is expected to be updated whenever DFD is changed or updated.

Requirement of Data Dictionary

The data is referenced via data dictionary while designing and implementing software. Data dictionary removes any chances of ambiguity. It helps keeping work of programmers and designers synchronized while using same object reference everywhere in the program.

Data dictionary provides a way of documentation for the complete database system in one place. Validation of DFD is carried out using data dictionary.

Contents

Data dictionary should contain information about the following

- Data Flow
- Data Structure
- Data Elements
- Data Stores
- Data Processing

Data Flow is described by means of DFDs as studied earlier and represented in algebraic form as described.

Example

Address = House No + (Street / Area) + City + State

Course ID = Course Number + Course Name + Course Level + Course Grades

=	**Composed of**
{}	Repetition
()	Optional
+	And
[/]	Or

Table 1.2

Data Elements

Data elements consist of Name and descriptions of Data and Control Items, Internal or External data stores etc. with the following details:

- Primary Name
- Secondary Name (Alias)
- Use-case (How and where to use)
- Content Description (Notation etc.)
- Supplementary Information (preset values, constraints etc.)

Data Store

It stores the information from where the data enters into the system and exists out of the system. The Data Store may include -

- **Files**
    - Internal to software.
    - External to software but on the same machine.
    - External to software and system, located on different machine.
- **Tables**
    - Naming convention
    - Indexing property

Data Processing

There are two types of Data Processing:

- **Logical:** As user sees it

- **Physical:** As software sees it.

1.11 REQUIREMENTS DOCUMENTATION:

Projects cannot be implemented without the proper requirements. A requirements document is a compilation of a variety of documents together. Requirements documentation in project management describes how each requirement meets the business needs for the project. Requirements should be measurable, traceable, consistent, complete and acceptable to the stakeholders.

There are many benefits of this particular project management output. It captures the project requirements and ensures that they meet the expectations of the stakeholders. It helps understand what is needed to produce the desired result of the project or the final product. Moreover, the requirements also help the project teamplan on how they should implement the quality control for the project.

There are several components of the requirements documentation that project managers need to know and these include the traceability of the business and project objectives, rules for the organization and its guiding principles. The requirements documentation needs to follow a specific format. The format can range from a simple document that lists the requirements that are categorized by the stakeholder or it can be elaborate and contain a detailed description of the project, an executive summary, and several attachments to support the project.

How to Write a Requirements Document?

If you are working for a software development company or other similar employer, you may need to come up with a requirements document for an IT product. This kind of document specifies what a future software application or IT product might look like, and more importantly, how it will be used and how it needs to be built. This is done by showing various markets for product development, along with other essential data. If you need to write a requirements document, these basic steps will assist in detailing what is needed.

STEPS TO WRITE REQUIREMENTS DOCUMENT:

1. **Create a comprehensive explanation of what is needed for a product**. The requirements document will need to fully develop the context around a product and what it must look like to help developers implement their work.

2. **Interview various sources.** Get information for the requirements document from business leaders, engineers, developers, sales reps, customers or anyone else with important information about needs for product development.

3. **List system requirements or properties.** One of the important elements of requirements is the system requirements, or how the product will interact with a given system for a workstation or network.
4. **Identify any constraints for the project.** Explaining restrictions or constraints within the requirements document will help further guide those who are working on the software or IT product.

5. **Consider any interface requirements.** Interface requirements are an important part of this document because they determine how the end-user will view the product. They often have a critical influence on the user-friendliness of a product.
   - Identify color schemes, command button requirements and any other part of a successful interface.
   - Keep in mind the programming tools that will be used to develop the project or product when listing interface requirements. This will provide more guidance for developers and others.
6. **Identify parameters like cost and scheduling.** These practical considerations should also be part of a requirements document.
7. **Work up a development plan.** Along with the above information, a good requirements document will often include further instructions about development of the product. This can take one of a number of forms, and may require a bit of creativity.
8. **Insert visuals.** Graphic mockups of a product keep the project fresh and appealing to the eye, while providing more detail about how something will look.
9. **Categorize and organize requirements**. Find a way to neatly fit each of these categories of requirements into a single document.
10. **Write the requirements document.** Utilize good document planning strategies in constructing something that reads well, where individual areas of the requirements document are easy to access.

1.12 SRS DOCUMENT

SRS stands for Software Requirements Specification, which is a document that fully describes the expected behavior of a software system.

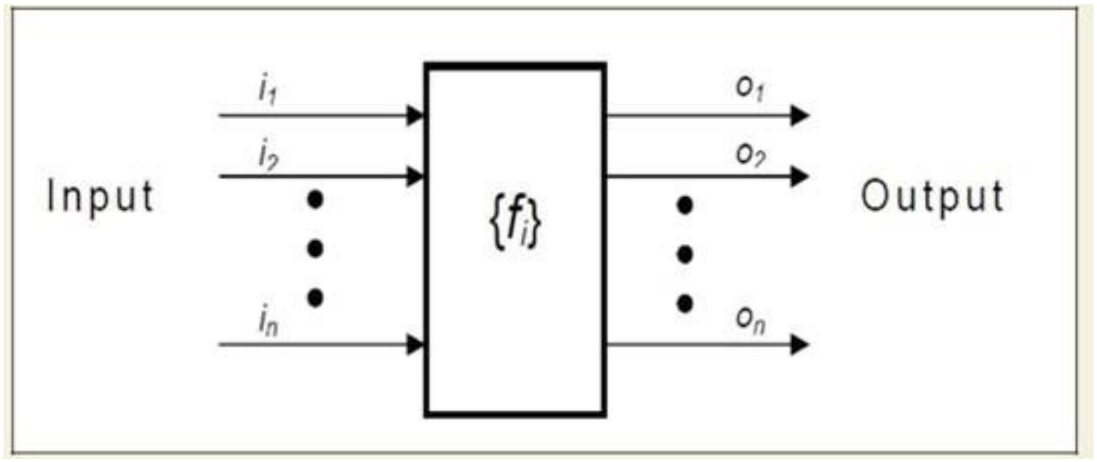
1.12.1 PARTS OF AN SRS DOCUMENT/ORGANIZATION OF SRS DOCUMENT:

The important parts of SRS document are:

- Functional requirements of the system
- Non-functional requirements of the system, and
- Goals of implementation

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in **fig.1.6.** Each function f_i of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.



. **Fig. 1.15 View of a system performing a set of functions**

Nonfunctional requirements:-

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Example: - Consider the case of the library system, where –

F1: Search Book function

Input: an author's name

Output: details of the author's books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high- level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Documenting functional requirements:

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw- cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM

R1: withdraw cash

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

R1.1 select withdraw amount option
  Input: "withdraw amount" option
  Output: user prompted to enter the account type
R1.2: select account type
  Input: user option
  Output: prompt to enter amount
R1.3: get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed

1.12.2 PROPERTIES OF A GOOD SRS DOCUMENT

The important properties of a good SRS document are the following:

- **Concise.** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.
- **Structured.** It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.
- **Black-box view/ Right level of abstraction**: If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the SRS. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

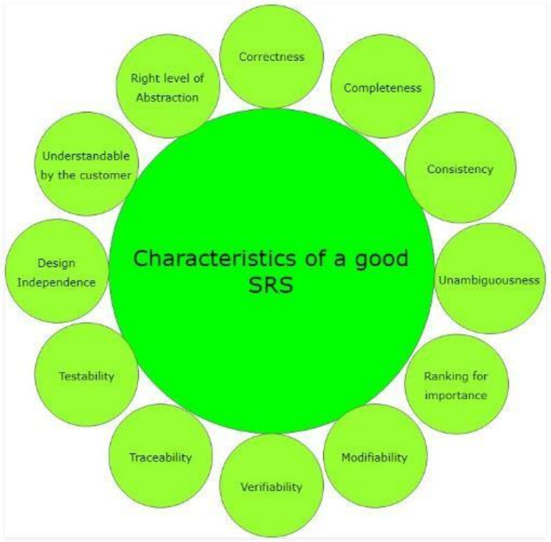


Fig. 1.16 Characteristics of a good SRS

- **Conceptual integrity.** It should show conceptual integrity so that the reader can easily understand it.

- **Response to undesired events.** It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

- **Verifiable.** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

- **Correctness:** User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

- **Completeness:** Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

- **Consistency:** Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

- **Unambiguousness:** An SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

- **Ranking for importance and stability:** There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

- **Modifiability:** SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.

- **Verifiability:** An SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement stating that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

- **Traceability:** One should be able to trace a requirement to a design component and then to a code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases

- **Design Independence:** There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

- **Testability:** An SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

- **Understandable by the customer:** An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

1.12.3 PROBLEMS WITHOUT AN SRS DOCUMENT

The important problems that an organization would face if it does not develop a SRS document are as follows:

- Without developing the SRS document, the system would not be implemented according to customer needs.
- Software developers would not know whether what they are developing is what exactly required by the customer.
- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
- It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Problems with an unstructured specification

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent

UNIT-2

SOFTWARE PROJECT MANAGEMENT CONCEPTS

2.1 SOFTWARE PROJECT

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

2.2 NEED OF SOFTWARE PROJECT MANAGEMENT

Software is said to be an intangible product. Software development is a kind of all new stream in world business and there's very little experience in building software products. Most software products are tailor made to fit client's requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.

2.3 SOFTWARE PROJECT MANAGER

A software project manager is a person who undertakes the responsibility of executing the software project. Software project manager is thoroughly aware of all the phases of SDLC that the software would go through. Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

A project manager closely monitors the development process, prepares and executes various plans, arranges necessary and adequate resources, maintains communication among all team members in order to address issues of cost, budget, resources, time, quality and customer satisfaction.

2.4 THE MANAGEMENT SPECTRUM:

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavour will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the product.

2.4.1 The People The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software

Engineering Institute has developed a people management capability maturity model (PM-CMM), “to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability” .The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices. The PM-CMM is a companion to the software capability maturity model that guides organizations in the creation of a mature software process.

People of a project includes from manager to developer, from client to finish user. However, principally people of a project highlight the developers. it's thus vital to own extremely skillful and intended developers that the software package Engineering Institute has developed land Management Capability Maturity Model (PM-CMM), “to enhance the readiness of software package organizations to undertake progressively advanced applications by serving to to draw in, grow, motivate, deploy, and retain the talent required to boost their software package development power”. Organizations that brought home the bacon high levels of maturity within the people management space have a better chance of implementing effective software package engineering practices.

The software process (and every software project) is populated by players who can be categorized into one of five constituencies

- **Senior Managers**: Senior Managers define the business issue.
- **Project Managers**: Project Managers plan,motivate, Organize and control the practitioners who do the Software work.
- **Practitioners**: Practitioners deliver the technical skills that are necessary to engineer a product or application.
- **Customers**: Customer specifies the requirements for the software to be developed.
- **End Users**: End Users interact with the software once it is released.

Every software project is populated by people who fall within this taxonomy. To be effective, the project team must be organized in a way that maximizes each person’s skills and abilities. And that’s the job of the team leader.

Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don’t have the right mix of people skills. And yet, as Edgemon states: “Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers.”

In an excellent book of technical leadership, Jerry Weinberg [WEI86] suggests a MOI model of leadership:

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

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

Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application. Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

Another view [EDG95] of the characteristics that define an effective project manager emphasizes four key traits:

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

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

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

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

The Software Team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The following options are available for applying human resources to a project that will require n people working for k years:

1. n individuals are assigned to m different functional tasks, relatively little combined work occurs; coordination is the responsibility of a software manager who may have six other projects to be concerned with.

2. n individuals are assigned to m different functional tasks (m < n) so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among teams is the responsibility of a software manager.

3. n individuals are organized into t teams; each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project; coordination is controlled by both the team and a software project manager.

The “best” team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [MAN81] suggests three generic team organizations:

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

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

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

Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability— are facts of life. To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively noninteractive and impersonal communication channels". Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis. Kraul and Streeter examine a collection of project coordination techniques that are categorized in the following manner:

Formal, impersonal approaches include software engineering documents and deliverables (including source code), technical memos, project milestones, schedules, and project control tools, change requests and related documentation, error tracking reports, and repository data.

Formal, interpersonal procedures focus on quality assurance activities applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures include group meetings for information dissemination and problem solving and "collocation of requirements and development staff."

Electronic communication encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.

Interpersonal networking includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

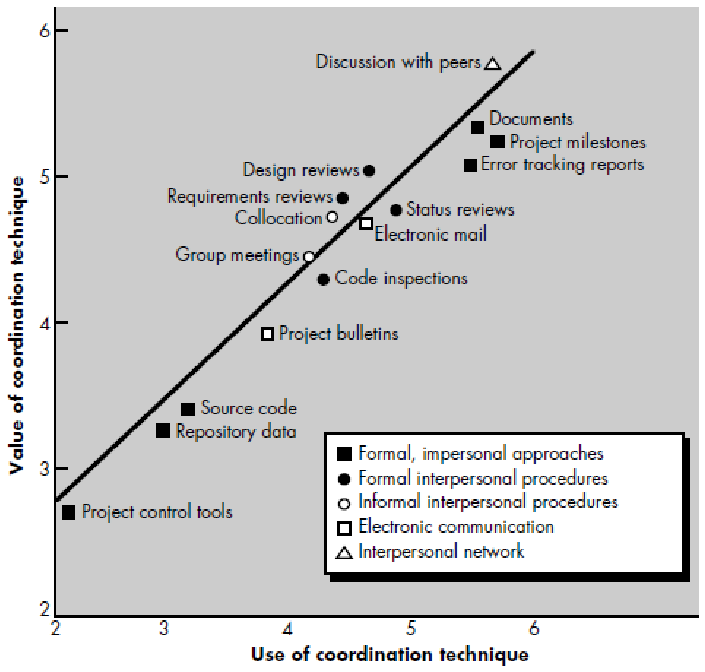


Fig. 2.1 Value and Use of Coordination And Communication Techniques

To assess the efficacy of these techniques for project coordination, Kraul and Streeter studied 65 software projects involving hundreds of technical staff. Figure above (adapted from [KRA95]) expresses the value and use of the coordination techniques just noted.

Referring to the above figure, the perceived value (rated on a seven point scale) of various coordination and communication techniques is plotted against their frequency of use on a project. Techniques that fall above the regression line were "judged to be relatively valuable, given the amount that they were used" [KRA95]. Techniques that fell below the line were perceived to have less value. It is interesting to note that interpersonal networking was rated the technique with highest coordination and communication value. It is also important to note that early software quality assurance mechanisms (requirements and design reviews) were perceived to have more value than later evaluations of source code (code inspections).

2.4.2 THE PRODUCT: Product is any package that needs to be developed. Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered and technical & management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress. The software developer and customer must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements analysis Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved. Scope identifies the primary data, functions and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner. Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

2.4.3 THE PROCESS: A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

2.4.4 THE PROJECT: We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. Here, the manager needs to perform some job. The project contains all and everything of the entire development method and to avoid project failure the manager needs to take some steps, needs to fret concerning some common warnings etc. In order to manage a successful software project, we must understand what can go wrong and how to do it right. A project is a series of steps where we need to make accurate decision so as to make a successful project. In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a common sense approach for planning, monitoring and controlling the project.

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right. In an excellent paper on software projects, John Reel [REE99] defines ten signs that indicate that an information systems project is in jeopardy:

1. Software people don't understand their customer's needs.

2. The product scope is poorly defined.

3. Changes are managed poorly.

4. The chosen technology changes.

5. Business needs change [or are ill-defined].

6. Deadlines are unrealistic.

7. Users are resistant.

8. Sponsorship is lost [or was never properly obtained].

9. The project team lacks people with appropriate skills.

10. Managers [and practitioners] avoid best practices and lessons learned.

Jaded industry professionals often refer to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes the other 90 percent of the allotted effort and time [ZAH94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceeding list.

But enough negativity! How does a manager act to avoid the problems just noted? Reel [REE99] suggests a five-part commonsense approach to software projects:

1. Start on the right foot. This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objects and expectations for everyone who will be involved in the project. It is reinforced by building the right team and giving the team the autonomy, authority, and technology needed to do the job.

2. Maintain momentum. Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.

3. Track progress. For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures can be collected and used to assess progress against averages developed for the software development organization.

4. Make smart decisions. In essence, the decisions of the project manager and the software team should be to “keep it simple.” Whenever possible, decide to use commercial off-the-shelf software or existing software components, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you’ll need every minute).

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

2.5 RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

SKILLS NECESSARY FOR SOFTWARE PROJECT MANAGEMENT

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

2.6 PROJECT PLANNING

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts.

2.6.1 What are the steps in Project Planning?

Project Planning spans across the various aspects of the Project. Generally Project Planning is considered to be a process of estimating, scheduling and assigning the projects resources in order to deliver an end product of suitable quality. However it is much more as it can assume a very strategic role, which can determine the very success of the project. A Project Plan is one of the crucial steps in Project Planning in General!

Typically Project Planning can include the following types of project Planning:

1) Project Scope Definition and Scope Planning
2) Project Activity Definition and Activity Sequencing
3) Time, Effort and Resource Estimation
4) Risk Factors Identification
5) Cost Estimation and Budgeting
6) Organizational and Resource Planning
7) Schedule Development
8) Quality Planning
9) Risk Management Planning
10) Project Plan Development and Execution
11) Performance Reporting
12) Planning Change Management
13) Project Rollout Planning

We now briefly examine each of the above steps:

1) **Project Scope Definition and Scope Planning:**
   In this step we document the project work that would help us achieve the project goal. We document the assumptions, constraints, user expectations, Business Requirements, Technical requirements, project deliverables, project objectives and everything that defines the final product requirements. This is the foundation for a successful project completion.

2) Quality Planning

The relevant quality standards are determined for the project. This is an important aspect of Project Planning. Based on the inputs captured in the previous steps such as the Project Scope, Requirements, deliverables, etc. various factors influencing the quality of the final product are determined. The processes required to deliver the Product as promised and as per the standards are defined.

3) Project Activity Definition and Activity Sequencing:

In this step, we define all the specific activities that must be performed to deliver the product by producing the various product deliverables. The Project Activity sequencing identifies the interdependence of all the activities defined.

4) Time, Effort and Resource Estimation:

Once the Scope, Activities and Activity interdependence is clearly defined and documented, the next crucial step is to determine the effort required to complete each of the activities. See the article on "Software Cost Estimation" for more details. The Effort can be calculated using one of the many techniques available such as Function Points, Lines of Code, Complexity of Code, Benchmarks, etc. This step clearly estimates and documents the time, effort and resource required for each activity.

5) Risk Factors Identification:

"Expecting the unexpected and facing it"
It is important to identify and document the risk factors associated with the project based on the assumptions, constraints, user expectations, specific circumstances, etc.

6) Schedule Development:

The time schedule for the project can be arrived at based on the activities, interdependence and effort required for each of them. The schedule may influence the cost estimates, the cost benefit analysis and so on.

Project Scheduling is one of the most important task of Project Planning and also the most difficult tasks. In very large projects it is possible that several teams work on developing the project. They may work on it in parallel. However their work may be interdependent.

Again various factors may impact in successfully scheduling a project

- o Teams not directly under our control
- o Resources with not enough experience

Popular Tools can be used for creating and reporting the schedules such as Gantt Charts

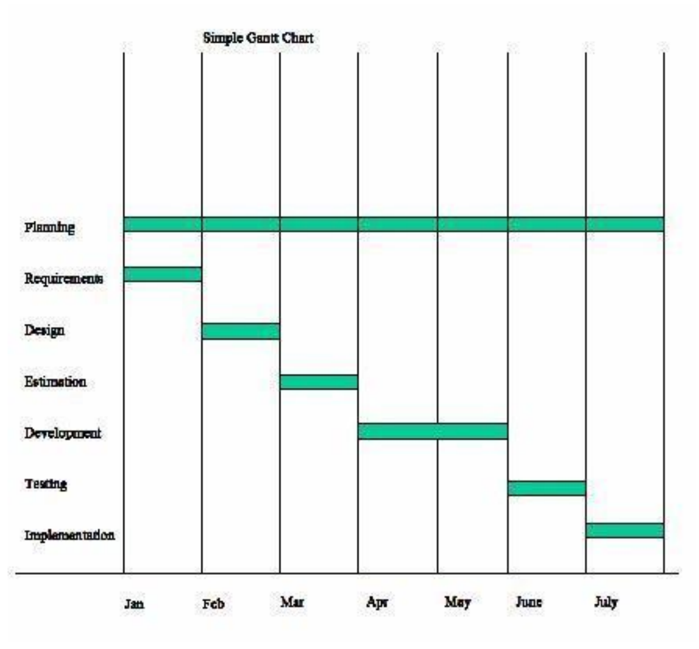


Fig. 2.2

7) Cost Estimation and Budgeting:

Based on the information collected in all the previous steps it is possible to estimate the cost involved in executing and implementing the project. See the article on "Software Cost Estimation" for more details. A Cost Benefit Analysis can be arrived at for the project. Based on the Cost Estimates Budget allocation is done for the project.

8) Organizational and Resource Planning

Based on the activities identified, schedule and budget allocation resource types and resources are identified. One of the primary goals of Resource planning is to ensure that the project is run efficiently. This can only be achieved by keeping all the project resources fully utilized as possible. The success depends on the accuracy in predicting the resource demands that will be placed on the project. Resource planning is an iterative process and necessary to optimize the use of resources throughout the project life cycle thus making the project execution more efficient. There are various types of resources – Equipment, Personnel, Facilities, Money, etc.

9) Risk Management Planning:

Risk Management is a process of identifying, analyzing and responding to a risk. Based on the Risk factors Identified a Risk resolution Plan is created. The plan analyses each of the risk factors and their impact on the project. The possible responses for each of them can be planned. Throughout the lifetime of the project these risk factors are monitored and acted upon as necessary.

10) Project Plan Development and Execution:

Project Plan Development uses the inputs gathered from all the other planning processes such as Scope definition, Activity identification, Activity sequencing, Quality Management Planning, etc. A detailed Work Break down structure comprising of all the activities identified is used. The tasks are scheduled based on the inputs captured in the steps previously described. The Project Plan documents all the assumptions, activities, schedule, timelines and drives the project.
Each of the Project tasks and activities are periodically monitored. The team and the stakeholders are informed of the progress. This serves as an excellent communication mechanism. Any delays are analyzed and the project plan may be adjusted accordingly.

11) Performance Reporting:

As described above the progress of each of the tasks/activities described in the Project plan is monitored. The progress is compared with the schedule and timelines documented in the Project Plan. Various techniques are used to measure and report the project performance such as EVM (Earned Value Management) A wide variety of tools can be used to report the performance of the project such as PERT Charts, GANTT charts, Logical Bar Charts, Histograms, Pie Charts, etc.

12) Planning Change Management:

Analysis of project performance can necessitate that certain aspects of the project be changed. The Requests for Changes need to be analyzed carefully and its impact on the project should be studied. Considering all these aspects the Project Plan may be modified to accommodate this request for Change.

Change Management is also necessary to accommodate the implementation of the project currently under development in the production environment. When the new product is implemented in the production environment it should not negatively impact the environment or the performance of other applications sharing the same hosting environment.

13) **Project Rollout Planning:**

In Enterprise environments, the success of the Project depends a great deal on the success of its rollout and implementations. Whenever a Project is rolled out it may affect the technical systems, business systems and sometimes even the way business is run. For an application to be successfully implemented not only the technical environment should be ready but the users should accept it and use it effectively. For this to happen the users may need to be trained on the new system. All this requires planning.

2.6.2 ACTIVITIES INVOLVED IN PROJECT PLANNING:

Project planning consists of the following essential activities:

- Estimating the following attributes of the project:
    - **Project size**: What will be problem complexity in terms of the effort and time required to develop the product?
    - **Cost**: How much is it going to cost to develop the project?
    - **Duration**: How long is it going to take to complete development?
    - **Effort**: How much effort would be required?

  The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.
- Scheduling manpower and other resources.
- Staff organization and staffing plans.
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

2.6.3 PRECEDENCE ORDERING AMONG PROJECT PLANNING ACTIVITIES

Different project related estimates done by a project manager have already been discussed. Fig. 30.1 shows the order in which important project planning activities may be undertaken. From fig. 30.1 it can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.

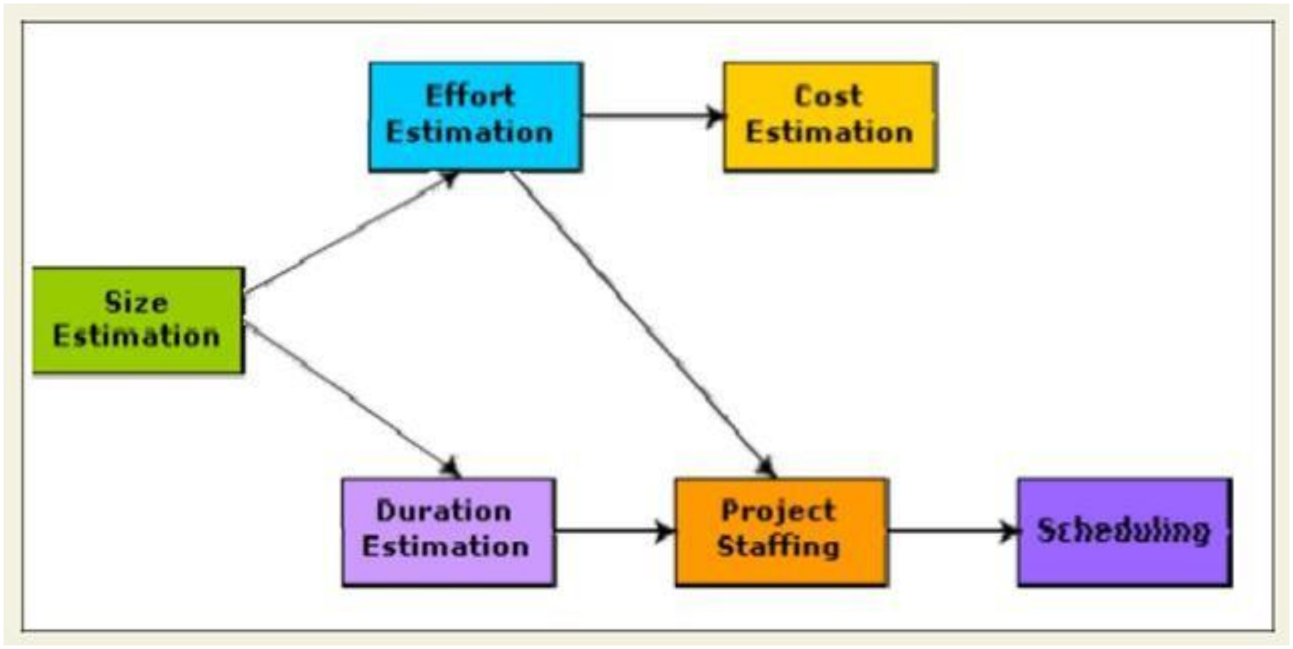


Fig. 2.3 Project Planning Activities

SLIDING WINDOW PLANNING

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However, Project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases. After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

2.7 SOFTWARE PROJECT MANAGEMENT PLAN (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below. This list can be used as a possible organization of the SPMP document.

Organization of the Software Project Management Plan (SPMP) Document

1. **Introduction**
   (a) Objectives
   (b) Major Functions
   (c) Performance Issues
   (d) Management and Technical Constraints

2. **Project Estimates**
   (a) Historical Data Used
   (b) Estimation Techniques Used
   (c) Effort, Resource, Cost, and Project Duration Estimates

3. **Schedule**
   (a) Work Breakdown Structure
   (b) Task Network Representation
   (c) Gantt Chart Representation
   (d) PERT Chart Representation

4. **Project Resources**
   (a) People
   (b) Hardware and Software
   (c) Special Resources

5. **Staff Organization**
   (a) Team Structure
   (b) Management Reporting

6. **Risk Management Plan**
   (a) Risk Analysis
   (b) Risk Identification

(c) Risk Estimation

(d) Risk Abatement Procedures

7. **Project Tracking and Control Plan**

8. **Miscellaneous Plans**

(a) Process Tailoring

(b) Quality Assurance Plan

(c) Configuration Management Plan

(d) Validation and Verification

(e) System Testing Plan

(f) Delivery, Installation, and Maintenance Plan

2.8 PROJECT SIZE ESTIMATION METRICS

The size of a program is neither the number of bytes that the source code occupies nor the byte size of the executable code but it is an indicator of the effort and time required to develop the program. In other words, the size of a program indicates the development complexity. There are several metrics to measure effort, time and cost of a planned software project. These are:

2.8.1 Size Metrics - LOC (Lines of Code), mostly calculated in thousands of delivered source code lines, denoted as KLOC. Function Point Count is measure of the functionality provided by the software. Function Point count defines the size of functional aspect of software.

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

2.8.2 Function point (FP)

This metric overcomes many of the shortcomings of the LOC metric. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed. The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. The size is also dependent on the number of files and the number of interfaces.

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

UFP = (<u>Number of inputs</u>)*4 + (<u>Number of outputs</u>)*5 + (<u>Number of inquiries</u>)*4 + (<u>Number of files</u>)*10 + (<u>Number of interfaces</u>)*10

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance. Inquiries are counted separately.

Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally,

FP=UFP*TCF

Shortcomings of LOC metric

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.

- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.

- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.

- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

Shortcomings of Feature Point Metric

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration.

2.9 PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

2.9.1 Empirical Estimation Techniques

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: ***Expert judgment technique*** and ***Delphi cost estimation.***

Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

Delphi Cost Estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

2.9.2 Heuristic Technique

Heuristic techniques assume that the relationships among the different project parameters can be modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

$$\textbf{Estimated Parameter} = c_1 * e^{d}{}_1$$

In the above expression, e is the characteristic of the software which has already been estimated (independent variable). Estimated Parameter is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc. c_1 and d_1 are constants. The values of the constants c_1 and d_1 are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\textbf{Estimated Resource} = c_1 * e_1{}^{d}{}_1 + c_2 * e_2{}^{d}{}_2 + \ldots$$

Where e_1, e_2, … are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, … are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the constants c_1, c_2, d_1, d_2, …. Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

2.9.3 Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique.

Halstead’s software science can be used to derive some interesting results starting with a few simple assumptions. Halstead’s software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

2.10 COCOMO MODEL

Organic, Semidetached and Embedded software projects

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm’s [1981] definition of organic, semidetached, and embedded systems are elaborated below.

Organic: A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

2.10.1 BASIC COCOMO

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ Months}$$

Where

• KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

• a_1, a_2, b_1, b2 are constants for each category of software products,

• Tdev is the estimated time to develop the software, expressed in months,

• Effort is the total effort required to develop the software product, expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in fig. below). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve (as shown in fig. below).

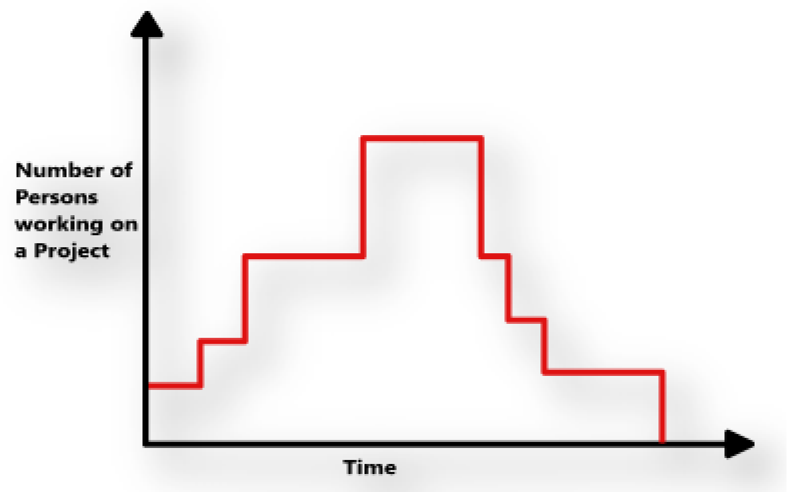


Fig. Person Month Curve

Fig. 2.4

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a_1, a_2, b_1, b_2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] as explained before. He derived the above expressions by examining historical data collected from a large number of actual projects.

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : Effort = $2.4(\text{KLOC})^{1.05}$ PM

Semi-detached : Effort = $3.0(\text{KLOC})^{1.12}$ PM

Embedded : Effort = $3.6(\text{KLOC})^{1.20}$ PM

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = $2.5(\text{Effort})^{0.38}$ Months

Semi-detached : Tdev = $2.5(\text{Effort})^{0.35}$ Months

Embedded : Tdev = $2.5(\text{Effort})^{0.32}$ Months

some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. 11.4 shows a plot of estimated effort versus product size. From fig. 11.4, we can observe that the effort is somewhat super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

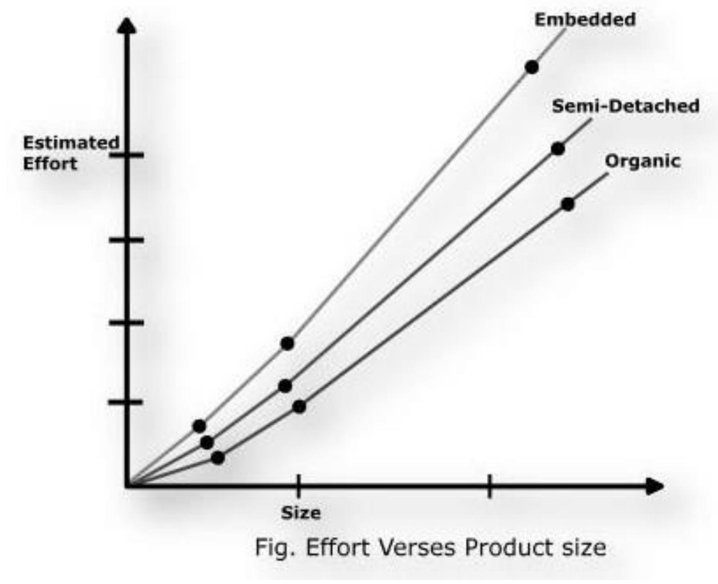


Fig. 2.5 Effort versus product size

The development time versus the product size in KLOC is plotted in fig. below. From fig. Given below, it can be observed that the development time is a sublinear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig. Given below, it can be observed that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

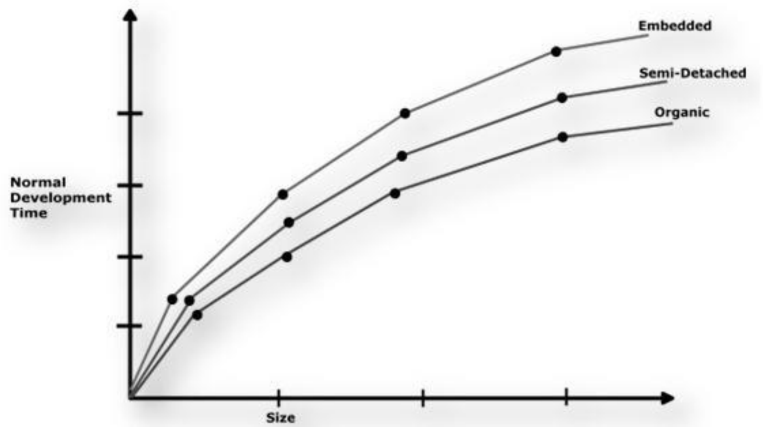


Fig 2.6 Development Time versus Size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

For Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time?

From the basic COCOMO estimation formula for organic software:

Effort = $\mathbf{2.4 \times (32)^{1.05} = 91}$ **PM**
Nominal development time = $\mathbf{2.5 \times (91)^{0.38} = 14}$ **months**
Cost required to develop the product = **14 x 15,000**
= Rs. 210,000/

2.10.2 INTERMEDIATE COCOMO MODEL

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.

Development Environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

2.10.3 COMPLETE COCOMO MODEL

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These subsystems may have widely different characteristics. For example, some subsystems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

- Database part
- Graphical User Interface (GUI) part
- Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

2.11 STAFFING LEVEL ESTIMATION

Once the effort required to develop a software has been determined, it is necessary to determine the staffing requirement for the project. Putnam first studied the problem of what should be a proper staffing pattern for software projects. He extended the work of Norden who had earlier investigated the staffing pattern of research and development (R&D) type of projects. In order to appreciate the staffing pattern of software projects, Norden's and Putnam's results must be understood.

2.11.1 NORDEN'S WORK

Norden studied the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve. Norden represented the Rayleigh curve by the following equation:

$$E = K/t^2_d * t * e^{-t^2 / 2 t^2_d}$$

Where E is the effort required at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R & D projects and were not meant to model the staffing pattern of software development projects.

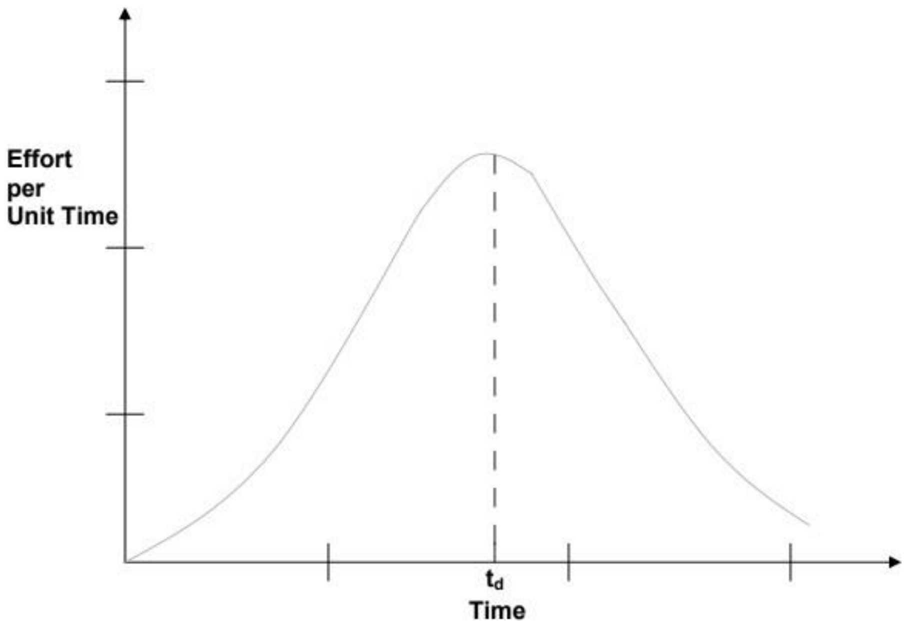


Fig. 2.7 Rayleigh curve

2.11.2 PUTNAM'S WORK

Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration testing. Therefore, t_d can be approximately considered as the time required developing the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of $C_k = 2$ for poor development environment (no methodology, poor documentation, and review, etc.), $C_k = 8$ for good software development environment (software engineering principles are adhered to), $C_k = 11$ for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from the historical data of the organization developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls.

However, the staff build-up should not be carried out in large instalments. The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve. Experience shows that a very rapid build-up of project staff any time during the project development correlates with schedule slippage.

It should be clear that a constant level of manpower throughout the project duration would lead to wastage of effort and increase the time and effort required to develop the product. If a constant number of engineers are used over all the phases of a project, some phases would be overstaffed and the other phases would be understaffed causing inefficient use of manpower, leading to schedule slippage and increase in cost.

2.12 ORGANIZATION STRUCTURE

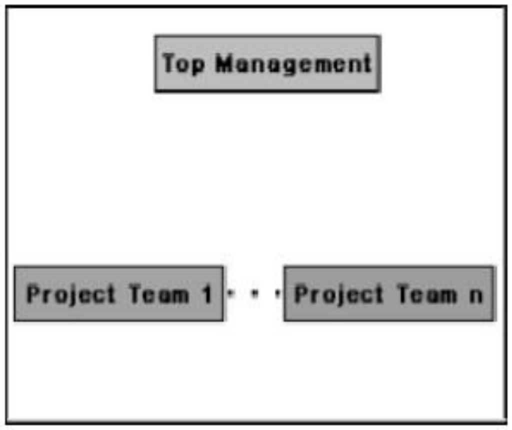
Usually every software development organization handles several projects at any time. Software organizations assign different teams of engineers to handle different software projects. Each type of organization structure has its own advantages and disadvantages so the issue “how is the organization as a whole structured?” must be taken into consideration so that each software project can be finished before its deadline.

Functional format vs. project format

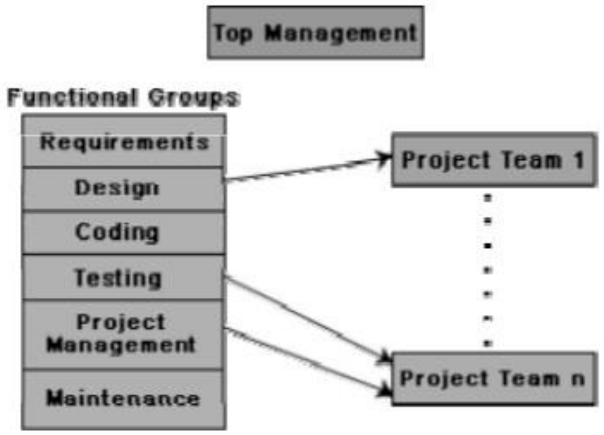
There are essentially two broad ways in which a software development organization can be structured: functional format and project format. In the project format, the project development staff are divided based on the project for which they work (as shown in figure below). In the functional format, the development staff are divided based on the functional group to which they belong. The different projects borrow engineers from the required functional groups for specific phases to be undertaken in the project and return them to the functional group upon the completion of the phase.

In the functional format, different teams of programmers perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the project evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood by the subsequent teams working on the project. This requires good quality documentation to be produced after every activity.

In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. Obviously, the functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams.



(a) Project Organization



(b) Functional Organization

Fig. 2.8 Schematic representation of the functional and project organization

.

2.13 ADVANTAGES OF FUNCTIONAL ORGANIZATION OVER PROJECT ORGANIZATION

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages.

The main advantages of a functional organization are:

- Ease of staffing
- Production of good quality documents
- Job specialization
- Efficient handling of the problems associated with manpower turnover.

The functional organization allows the engineers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organization also provides an efficient solution to the staffing problem. We have already seen that the staffing pattern should approximately follow the Rayleigh distribution for efficient utilization of the personnel by minimizing their wait times.

The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organization. A project organization structure forces the manager to take in almost a constant number of engineers for the entire duration of his project. This results in engineers idling in the initial phase of the software development and is under tremendous pressure in the later phase of the development. A further advantage of the functional organization is that it is more effective in handling the problem of manpower turnover. This is because engineers can be brought in from the functional pool when needed. Also, this organization mandates production of good quality documents, so new engineers can quickly get used to the work already done.

2.14 UNSUITABILITY OF FUNCTIONAL FORMAT IN SMALL ORGANIZATIONS

In spite of several advantages of the functional organization, it is not very popular in the software industry. The apparent paradox is not difficult to explain. The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, coder, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organizations to fill in slots for some roles such as maintenance, testing, and coding groups. Also, another problem with the functional organization is that if an organization handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects. Also, for obvious reasons the functional format is not suitable for small organizations handling just one or two projects.

2.15 TEAM STRUCTURES

Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized. There are mainly three formal team structures: chief programmer, democratic, and the mixed team organizations although several other variations to these structures are possible. Problems of different complexities and sizes often require different team structures for chief solution. Chief Programmer Team In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in figure given below. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

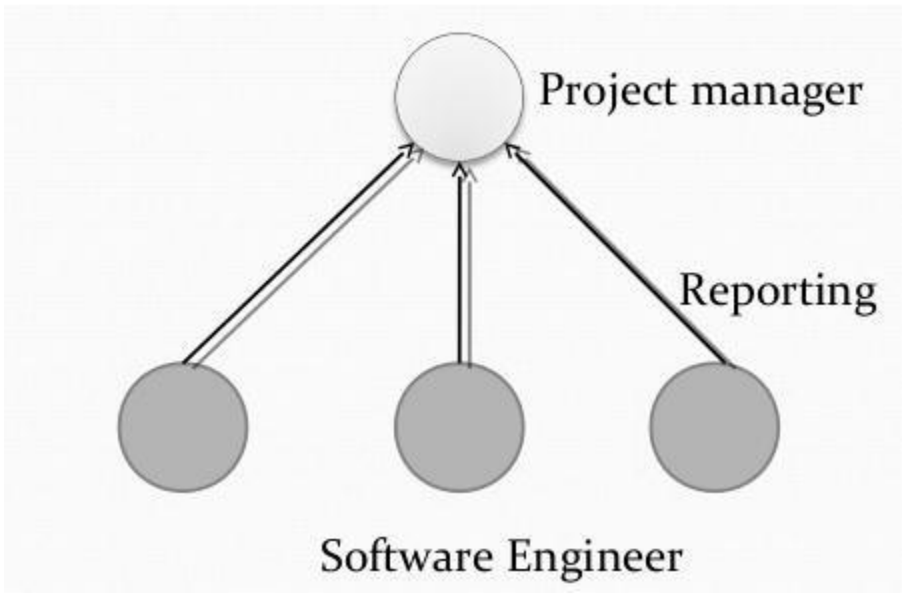


Fig. 2.9 Chief programmer team structure

The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can work out a satisfactory design and ask the programmers to code different modules of his design solution. For example, suppose an organization has successfully completed many simple MIS projects. Then, for a similar MIS project, chief programmer team structure can be adopted. The chief programmer team structure works well when the task is within the intellectual grasp of a single individual. However, even for simple and well-understood problems, an organization must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early project completion outweighs other factors such as team morale, personal developments, life-cycle cost etc.

2.15.1 DEMOCRATIC TEAM

The democratic team structure, as the name implies, does not enforce any formal team hierarchy (as shown in figure given below). Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

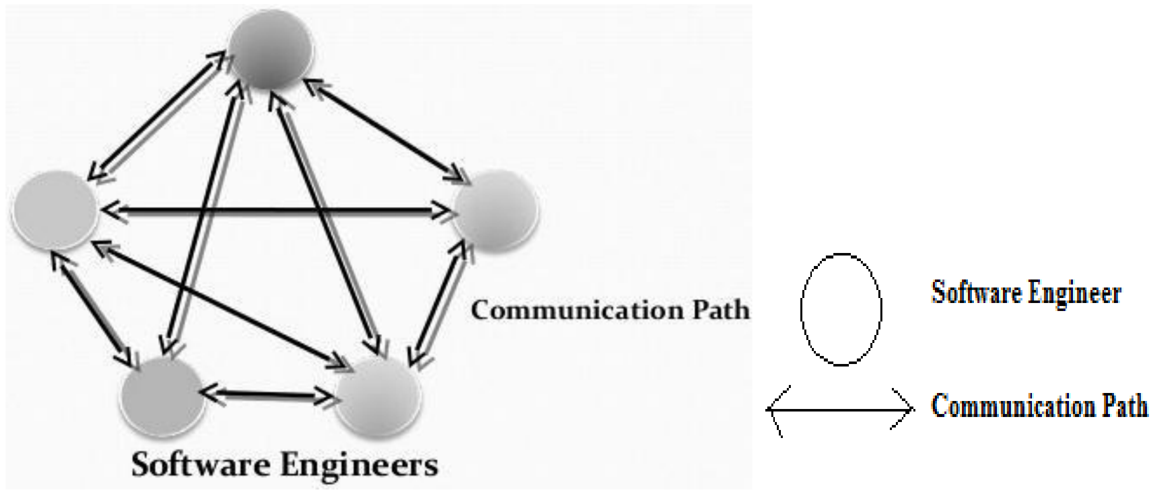


Fig. 2.10 Democratic team structure

The democratic organization leads to higher morale and job satisfaction. Consequently, it suffers from less man-power turnover. Also, democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic. The democratic team organization encourages egoless programming as programmers can share and review one another’s work.

2.15.2 MIXED CONTROL TEAM ORGANIZATION

The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization. The mixed control team organization is shown pictorially in fig. 12.4. This team organization incorporates both hierarchical reporting and democratic set up. In fig. 12.4, the democratic connections are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organization is suitable for large team sizes. The democratic arrangement at the senior engineers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organization is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.

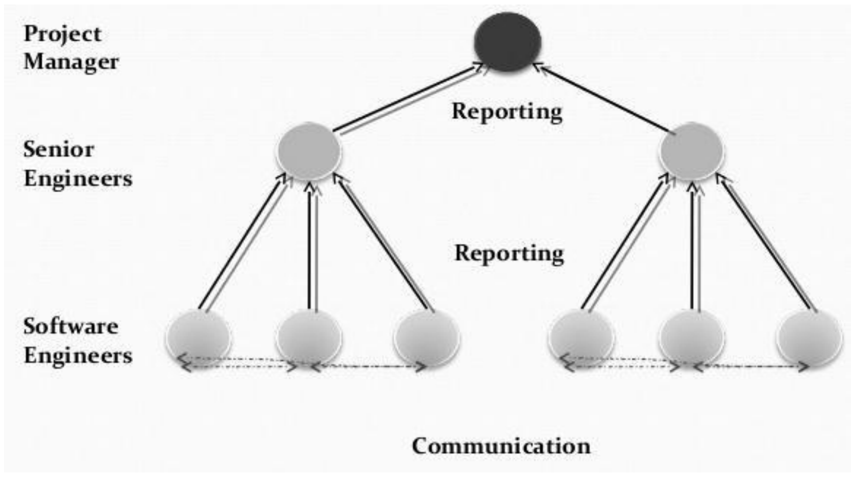


Fig. 2.11 Mixed team structure

2.16 RISK ANALYSIS AND MANAGEMENT

Risk is the chance that an event would occur which will lead to change in the project circumstances that were assumed while forecasting the project costs and benefits and will have an impact on project objectives. To ensure that these events do not lead to failure of the projects, there is a need to manage the risks associated with the projects through adoption of appropriate risk management framework. In order to successfully manage the risk, it is necessary to know: what event will trigger the risk, the probability (or likelihood) of occurrence of the risk event, and the consequences of the risk event if it occurs. The concept of risk management, therefore, deals with identifying the risks associated with the project, assessing their probability of occurrence and their potential impact on critical project performance measures, and employing direct and indirect means for either reducing the exposure of the underlying project activities to these risks or shifting some of the exposure to other.

2.16.1 RISK MANAGEMENT PROCESS

A software project can be affected by a large variety of risks. In order to be able to systematically identify the important risks which might affect a software project, it is necessary to categorize risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project:

1. **Project risks.** Project risks concern varies forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen. For any manufacturing project, such as manufacturing of cars, the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus he can easily assess the progress of the work and control it. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

2. **Technical risks**. Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due to the development team's insufficient knowledge about the project.

3. **Business risks.** This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.

Risk management is an ongoing process which continues through the lifecycle of a project. The risk management process takes place in the following stages:

1. **Risk identification**: The process of identifying all the risks associated with the project, whether during its development phase, or its construction or operational phase.
2. **Risk assessment:** The process of determining the likelihood of the identified risks materialising and the magnitude of their consequences if they do materialise.

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

- The likelihood of a risk coming true (denoted as r).
- The consequence of the problems associated with that risk (denoted as s).

Based on these two factors, the priority of each risk can be computed:

p = r * s

Where, p is the priority with which the risk must be handled, r is the probability of the risk becoming true, and s is the severity of damage caused due to the risk becoming true. If all identified risks are prioritized, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for these risks.

3. **Risk containment:** After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. Different risks require different containment procedures. In fact, most risks require ingenuity on the part of the project manager in tackling the risk.

There are three main strategies to plan for risk containment:

- **Avoid the risk:** This may take several forms such as discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.
- **Transfer the risk:** This strategy involves getting the risky component developed by a third party, buying insurance cover, etc.
- **Risk reduction:** This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

4. **Risk leverage:** To choose between the different strategies of handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this the risk leverage of the different risks can be computed. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

Risk Leverage = (risk exposure before reduction – risk exposure after reduction) / (cost of reduction)

2.16.2 RISK RELATED TO SCHEDULE SLIPPAGE

Even though there are three broad ways to handle any risk, but still risk handling requires a lot of ingenuity on the part of a project manager. As an example, it can be considered the options available to contain an important type of risk that occurs in many software projects – that of schedule slippage. Risks relating to schedule slippage arise primarily due to the intangible nature of software. Therefore, these can be dealt with by increasing the visibility of the software product. Visibility of a software product can be increased by producing relevant documents during the development process wherever meaningful and getting these documents reviewed by an appropriate team. Milestones should be placed at regular intervals through a software engineering process to provide a manager with regular indication of progress. Completion of a phase of the development process before followed need not be the only milestones. Every phase can be broken down to reasonable-sized tasks and milestones can be scheduled for these tasks too. A milestone is reached, once documentation produced as part of a software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to set a milestone every 10 to 15 days.

The successful implementation of the various stages of the risk management process requires putting in place an effective plan for communication and consultation with both the project's external and internal stakeholders in order to ensure that those responsible for implementing risk management and those with vested interest understand on what basis decisions are made and why particular actions are required. This consultative approach help to define the context appropriately, to help ensure risks are identified effectively, bringing different areas of expertise together in analyzing risks, ensuring different views are appropriately considered in evaluating risks. This approach also adds a sense of ownership of risk to the managers and the stakeholders.

2.17 PROJECT SCHEDULING

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different engineers. The work breakdown structure formalism helps the manager to breakdown the tasks systematically.

After the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. In general, the task dependencies define a partial ordering among tasks, i.e. each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities are represented in the form of an activity network.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT chart representation is developed. The PERT chart representation is suitable for program monitoring and control. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

2.17.1 WORK BREAKDOWN STRUCTURE

Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem. The root of the tree is labelled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. The next figure represents the WBS of an MIS (Management Information System) software.

While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

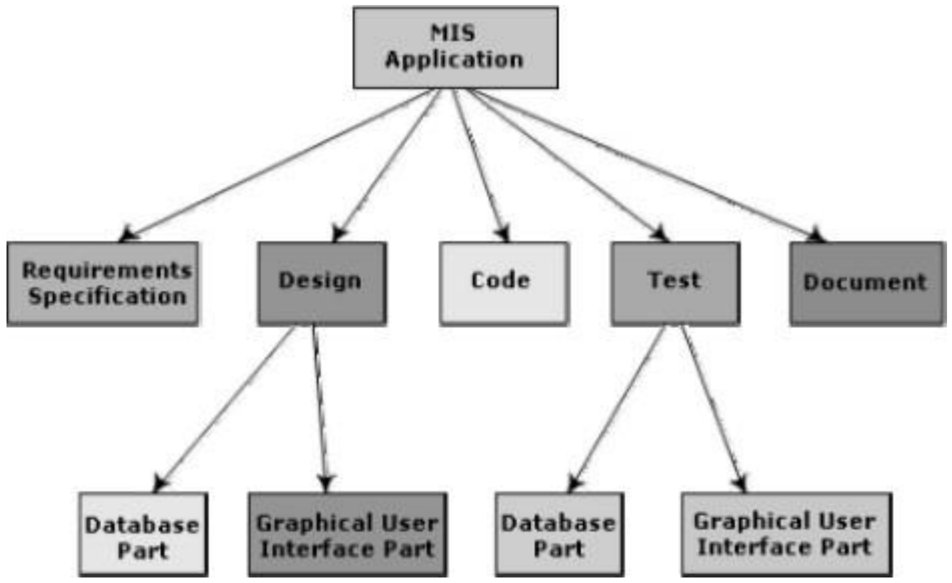

MIS Application
Requirements Specification
Design
Code
Test
Document
Database Part
Graphical User Interface Part
Database Part
Graphical User Interface Part

Fig. 2.12 Work breakdown structure of an MIS problem

2.17.2 ACTIVITY NETWORKS AND CRITICAL PATH METHOD

WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies (as shown in figure given below). Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task. Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however is not a good idea, because software engineers often resent such unilateral decisions. A possible alternative is to let engineer himself estimate the time for an activity he can assigned to. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the engineers to do a better and faster job. However, careful experiments have shown that unrealistically aggressive schedules not only cause engineers to compromise on intangible quality aspects, but also are a cause for schedule delays. A good way to achieve accurately in estimation of the task durations without creating undue schedule pressures is to have people set their own schedules.

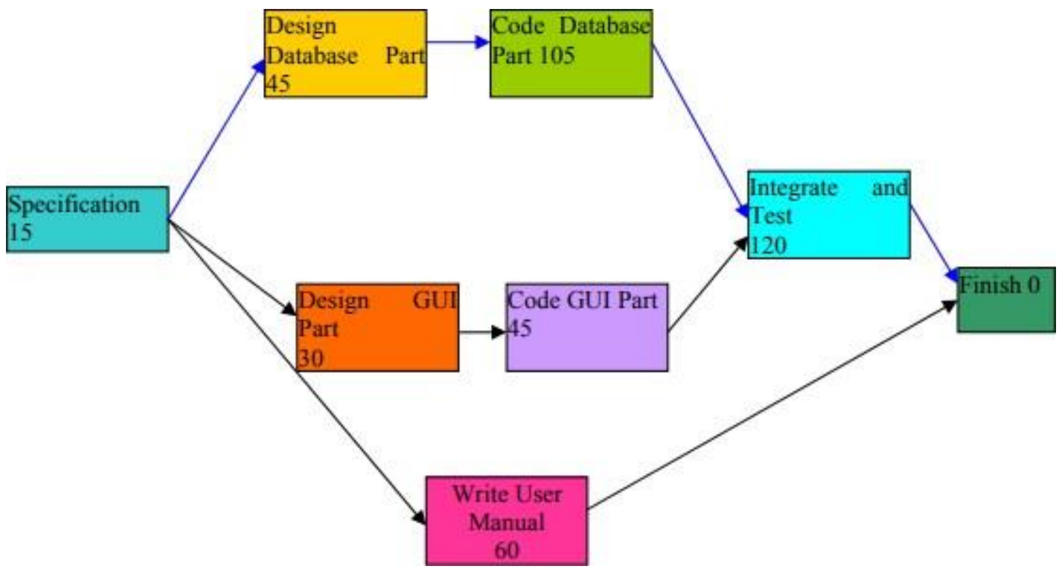


Fig. 2.13 Activity network representation of the MIS problem

2.17.3 CRITICAL PATH METHOD (CPM)

From the activity network representation following analysis can be made. The minimum time (MT) to complete the project is the maximum of all paths from start to finish. The earliest start (ES) time of a task is the maximum of all paths from the start to the task. The latest start time is the difference between MT and the maximum of all paths from this task to the finish. The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task. The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT. The slack time (ST) is LS – EF and equivalently can be written as LF – EF. The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the "flexibility" in starting and completion of tasks. A critical task is one with a zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path. These parameters for different tasks for the MIS problem are shown in the following table.

Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code database	60	165	60	165	0
Code GUI part	45	90	120	165	75
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

Table2.1: Different Parameters of Tasks

The critical paths are all the paths whose duration equals MT. The critical path in the activity network figure above is shown with a blue arrow.

2.17.4 GANTT CHART

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity. Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of figure Activity network representation of the MIS problem given above is shown in the figure given below.

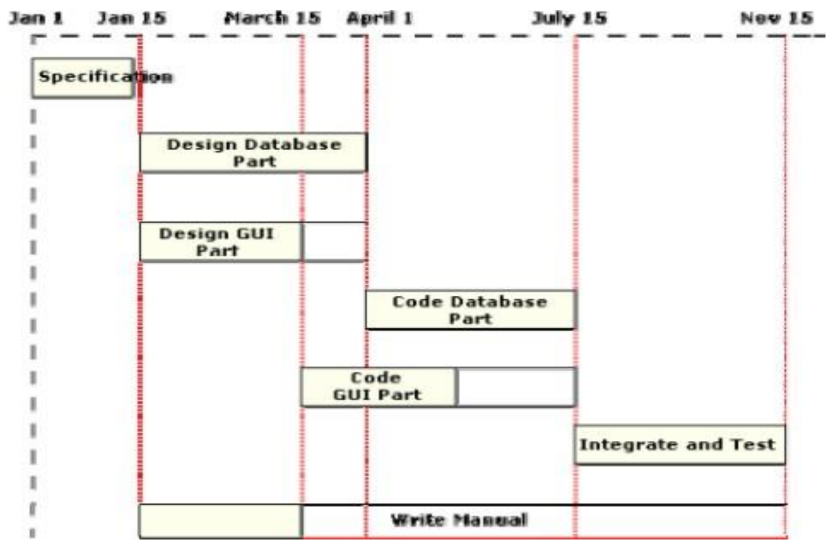


Fig. 2.14 Gantt chart representation of the MIS problem

2.17.5 PERT CHART

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of fig. above i.e Activity network representation of the MIS problem is shown in figure given below. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited.

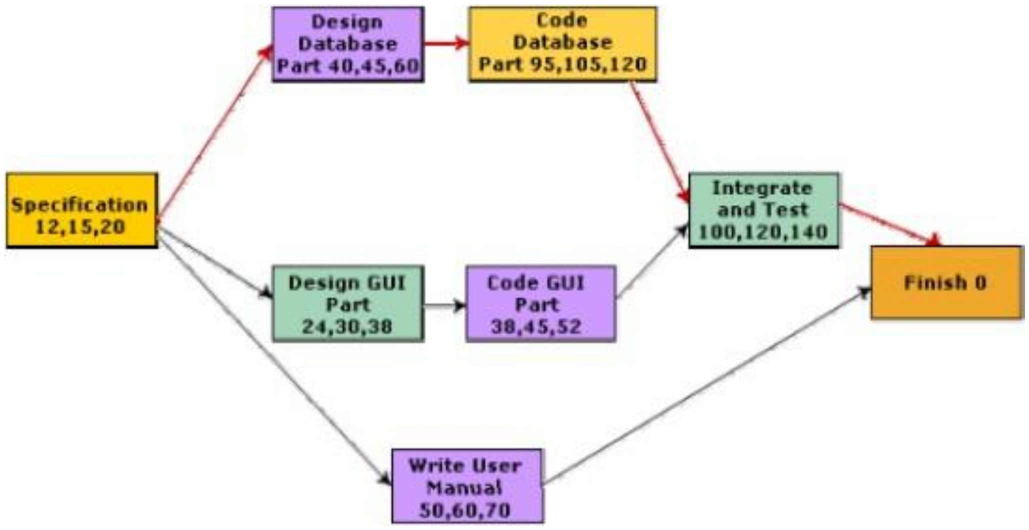

Design Database Part 40,45,60
Code Database Part 95,105,120
Specification 12,15,20
Integrate and Test 100,120,140
Finish 0
Design GUI Part 24,30,38
Code GUI Part 38,45,52
Write User Manual 50,60,70

Fig.2.15 PERT chart representation of the MIS problem

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.

2.18 SOFTWARE DESIGN AND IMPLEMENTATION

The implementation phase is the process of converting a system specification into an executable system. If an incremental approach is used, it may also involve refinement of the software specification. A software design is a description of the structure of the software to be implemented, data models, interfaces between system components, and maybe the algorithms used. The software designers develop the software design iteratively; they add formality and detail and correct the design as they develop their design. Here's an abstract model of the design process showing the inputs, activities, and the documents to be produced as output.

The diagram suggests that the stages of the design process are sequential. In fact, they are interleaved. A feedback from one stage to another and rework can't be avoided in any design process.

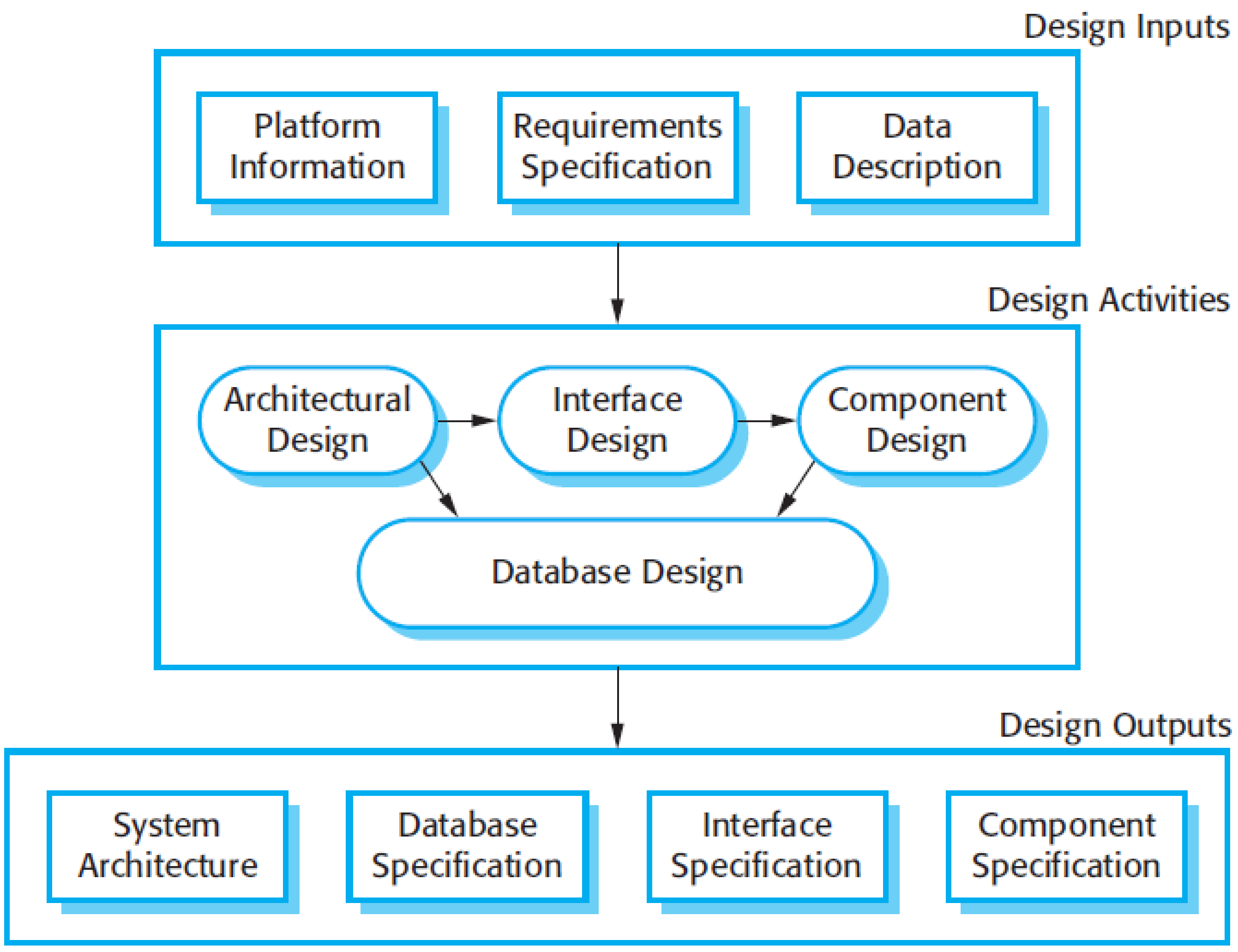


Fig. 2.16 The software design process

These activities can vary depending on the type of the system needs to be developed. We've showed four main activities that may be part of the design process for information systems, and they are:

1. **Architectural design:** It defines the overall structure of the system, the main components, their relationships.
2. **Interface design:** It defines the interfaces between these components. The interface specification must be clear. Therefore, a component can be used without having to know it's implemented. Once the interface specification are agreed, the components can be designed and developed concurrently.

3. **Component design:** Take each component and design how it will operate, with the specific design left to the programmer, or a list of changes to be made to a *reusable* component.
4. **Database design:** The system data structures are designed and their representation in a database is defined. This depends on whether an existing database is to be reused or a new database to be created.

These activities lead to a set of design outputs. The detail and representation vary based on the system being developed. For example, in critical systems, detailed design documents giving a precise and accurate description of the system must be produced. These outputs may be graphical models of the system, and in many cases, automatically generating code from these models.

UNIT-3
SOFTWARE DESIGN

3.1 INTRODUCTION

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

3.2 SOFTWARE DESIGN LEVELS

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

- **High-level Design-** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

MODULARIZATION:

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantages of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

CONCURRENCY

Back in time, all softwares were meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, software has multiple modules, and then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other. It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution. Example: The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

3.3 COUPLING & COHESION

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are a set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

3.3.1 COHESION

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.

- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

3.3.2 COUPLING

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better is the program.

There are five levels of coupling, namely -

- **Content coupling -** When a module can directly access or modify or refer to the contents of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

3.4 DESIGN VERIFICATION

The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements. The next phase, which is the implementation of software, depends on all outputs mentioned above.

It is then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of design phase are in formal notation form, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.

By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is import ant for good software design, accuracy and quality.

3.5 SOFTWARE DESIGN STRATEGIES

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution. There are multiple variants of software design. These variants are:

3.5.1 STRUCTURED DESIGN

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on 'divide and conquer' strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

Cohesion - grouping of all functionally related elements.

Coupling - communication between different modules.

A good structured design has ***high*** cohesion and ***low*** coupling arrangements.

3.5.2 FUNCTION ORIENTED DESIGN:

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used. This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions change the data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

3.5.3 OBJECT ORIENTED DESIGN

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.
  In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.

- Class hierarchy and relation among them are defined.
- Application framework is defined.

3.6 SOFTWARE DESIGN APPROACHES

There are two generic approaches for software designing:

3.6.1 TOP DOWN DESIGN

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their one set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

3.6.2 BOTTOM-UP DESIGN

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

3.7 SOFTWARE METRICS AND MEASURES

A **measure** provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or a process. **Measurement** is the act of determining a measure. Once measures are collected they are converted into metrics for use. IEEE defines **metric** as 'a quantitative measure of the degree to which a system, component, or process possesses a given attribute.' It can also be defined as “The continuous application of measurement based techniques to the software development process and its product to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products". A measure may be the number of errors found in a module during testing. Measurement is the result of such data collection. Software metric should relate the individual measures in some way like: average number of errors found per hour of the testing.

3.7.1 WHAT AND WHY?

The goal of software metrics is to identify and control essential parameters that affect software development. Other objectives of using software metrics are listed below:

- Measuring the size of the software quantitatively.
- Assessing the level of complexity involved.
- Assessing the strength of the module by measuring coupling.
- Assessing the testing techniques.
- Specifying when to stop testing.
- Determining the date of release of the software.
- Estimating cost of resources and project schedule.
- Software metrics help project managers to gain an insight into the efficiency of the software process, project, and product. This is possible by collecting quality and productivity data and then analyzing and comparing these data with past averages in order to know whether quality improvements have occurred. Also, when metrics are applied in a consistent manner, it helps in project planning and project management activity.

SCOPE OF SOFTWARE METRICS

- ➢ Software metrics contains many activities which include the following −
- ➢ Cost and effort estimation
- ➢ Productivity measures and model
- ➢ Data collection
- ➢ Quantity models and measures
- ➢ Reliability models
- ➢ Performance and evaluation models
- ➢ Structural and complexity metrics
- ➢ Capability – maturity assessment
- ➢ Management by metrics
- ➢ Evaluation of methods and tools

Software measurement is a diverse collection of these activities that range from models predicting software project costs at a specific stage to measures of program structure.

Software Metrics provide measures for various aspects of software process and software product. Software metrics is a standard of measure that contains many activities which involve some degree of measurement. It can be classified into three categories: product metrics, process metrics, and project metrics.

3.7.2 CATEGORIES OF METRICS:

- **Product metrics** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability and quality level.
- **Process metrics** can be used to improve software development and maintenance. Examples include effort required in the process, time to produce the product, the effectiveness of defect removal during development, number of defects found during testing, the pattern of testing defect arrival, and the response time of the fix process.
- **Project metrics** describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

Software Measures can be understood as a process of quantifying and symbolizing various attributes and aspects of software. Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics.

Software measures are fundamental requirement of software engineering. They not only help to control the software development process but also aid to keep quality of ultimate product excellent.

3.8 SOFTWARE METRICS

3.8.1 TOKEN COUNT

The drawback in LOC size measure of treating all lines alike can be solved by giving more weight to those lines, which are difficult to code and have more "stuff". One natural solution to this problem may be to count the basic symbols used in a line instead of lines themselves. These basic symbols are called "tokens". Such a scheme was used by Halstead in his theory of software science.

HALSTEAD’S SOFTWARE METRICS

A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands. Halstead’s metrics are included in a number of current commercial tools that count software lines of code. By counting the tokens and determining which operators are and which are operands, the following base measures can be collected:

n1 = Number of distinct operators.
n2 = Number of distinct operands.

N1 = Total number of occurrences of operators.

N2 = Total number of occurrences of operands.

In addition to the above, Halstead defines the following:

n1* = Number of potential operators.

n2* = Number of potential operands.

Halstead refers to n1* and n2* as the minimum possible number of operators and operands for a module and a program respectively. This minimum number would be embodied in the programming language itself, in which the required operation would already exist (for example, in C language, any program must contain at least the definition of the function main()), possibly as a function or as a procedure: n1* = 2, since at least 2 operators must appear for any function or procedure : 1 for the name of the function and 1 to serve as an assignment or grouping symbol, and n2* represents the number of parameters, without repetition, which would need to be passed on to the function or the procedure.

Halstead metrics –

Halstead metrics are:

- **Halstead Program Length –** The total number of operator occurrences and the total number of operand occurrences.
  N = N1 + N2
  And estimated program length is, N^ = n1log2n1 + n2log2
- **Halstead Vocabulary –** The total number of unique operator and unique operand occurrences.
  n = n1 + n2
- **Program Volume –** Proportional to program size, represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The properties V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.
  V = Size * (log2 vocabulary) = N * log2(n)
  The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used. And error = Volume / 3000
- **Program Difficulty –** This parameter shows how difficult to handle the program is.
  D = (n1 / 2) * (N2 / n2)
  D = 1 / L

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

- **Programming Effort –** Measures the amount of mental activity needed to translate the existing algorithm into implementation in the specified program language. E = V / L = D * V = Difficulty * Volume

- **Language Level –** Shows the algorithm implementation program language level. The same algorithm demands additional effort if it is written in a low level program language. For example, it is easier to program in Pascal than in Assembler. L' = V / D / D
  And estimated program level is L^ =2 * (n2) / (n1)(N2)

- **Intelligence Content –** Determines the amount of intelligence presented (stated) in the program This parameter provides a measurement of program complexity, independently of the program language in which it was implemented. I = V / D.

- **Programming Time –** Shows time (in minutes) needed to translate the existing algorithm into implementation in the specified program language. T = E / (f * S).

Example:

Consider a program having

- **Number of distinct operator : 12**
- **Number of operands : 5**
- **Total number of operator occurrences : 20**
- **Total number of operand occurrences :**

  Calculate the different Halstead software metrics for above programs.

Solution.

Number of distinct operator (η_1) = 12

Number of distinct operands (η_2) = 05

Total No. of operator occurence (N_1) = 20

Total No. of operand occurence (N_2) = 15

Program length (N)

$N = N_1 + N_2 = 20+15$

$N = 35$

Program vocabulary (η)

$\eta = \eta_1 + \eta_2$

$\eta = 12 + 05$

$\eta = 17$

Volume of Program (v) $V = N*\log_2 \eta$

$V = 25*\log_2 17 = 24 \times 4.08 = 102$

Potential volume of Program (V*):

$V* = (2 + \eta_2*) \log_2 (2 + \eta_2*)$

$= (2 + 12) \log_2 (2 + 12)$

$= 14 \log_2 14 = 14 \times 3.80 = 53.2$

Program level (L) $L = V*/V = 53.2/102 = 0.54$

Counting rules for C language –

1. Comments are not considered.
2. The identifier and function declarations are not considered
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Functions calls are considered as operators.
7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.
8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.
10. All the brackets, commas, and terminators are considered as operators.

11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrence of "+" and "-" are dealt separately. Similarly "*" (multiplication operator) are dealt separately.
13. In the array variables such as "array-name [index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
14. In the structure variables such as "struct-name, member-name" or "struct-name -> member-name", struct-name, member-name are taken as operands and '.', '->' are taken as operators. Some names of member elements in different structure variables are counted as unique operands.
15. All the hash directives are ignored.

3.8.2 QUALITY METRICS

Defects, their types and causes, consequence, intensity of severity and their implications define the quality of product.
The number of defects found in development process and number of defects reported by the client after the product is installed or delivered at client-end, define quality of product.

3.8.3 PROCESS METRICS - In various phases of SDLC, the methods and tools used, the company standards and the performance of development are software process metrics.

3.8.4 RESOURCE METRICS - Effort, time and various resources used, represents metrics for resource measurement.

3.8.5 COMPLEXITY METRICS - McCabe's Cyclomatic complexity quantifies the upper bound of the number of independent paths in a program, which is perceived as complexity of the program or its modules. It is represented in terms of graph theory concepts by using control flow graph. McCabe suggests that complexity does not depend on the number of statements. Instead it depends only on the decision structure of the program – the number of if, while and similar statements. To calculate the cyclomatic complexity of a program, count the number of conditions and add one. For example, the program fragment:

```
x = y;
if ( a = = b)
   c = = d;
else
   e = f;
p=q
```

has a complexity of 2, because there are two independent paths through the program. Similarly a while and a repeat each count one towards the complexity count. Compound conditions like:

```
if a > b and c > d
```

then count two because this if statement could be rewritten as two, nested if statements. Note that a program that consists only of a sequence of statements, has a cyclomatic complexity of 1, however long it is. Thus the smallest value of this metric is 1.

There are two ways of using McCabe's measure. First, if we had two algorithms that solve the same problem, we could use this measure to select the simpler. Second, McCabe suggests that if the cyclomatic complexity of a component is greater than 10, then it is too complex. In such a case, it should either be rewritten or else broken down into several smaller components. Cyclomatic complexity is a useful attempt to quantify complexity, and it is claimed that it has been successfully applied. It is, however, open to several criticisms as follows. First, why is the value of 10 adopted as the limit? This figure for the maximum allowed complexity is somewhat arbitrary and unscientific.

Second, the measure makes no allowance for the sheer length of a module, so that a one- page module (with no decisions) is rated as equally complex as a thousand-page module (with no decisions).

Third, the measure depends only on control flow, ignoring, for example, references to data. One program might only act upon a few items of data, while another might involve operations on a variety of complex objects. (Indirect references to data, say via pointers, are an extreme case.)

So McCabe's measure is a crude attempt to quantify the complexity of a software component. But it suffers from obvious flaws and there are various suggestions for devising an improved measure. However, McCabe's complexity measure has become famous and influential as a starting point for work on metrics.

Cyclomatic complexity = $E – N + 2 * P$
where,
E = number of edges in a flow graph
N = number of nodes in a flow graph
P = number of nodes that have exit points

Example of Cyclomatic complexity

```
If A = 10 then
  If B > C then
    A = B
```

```
ELSE
    A = C
  ENDIF
ENDIF
PRINT A
PRINT B
PRINT C
```

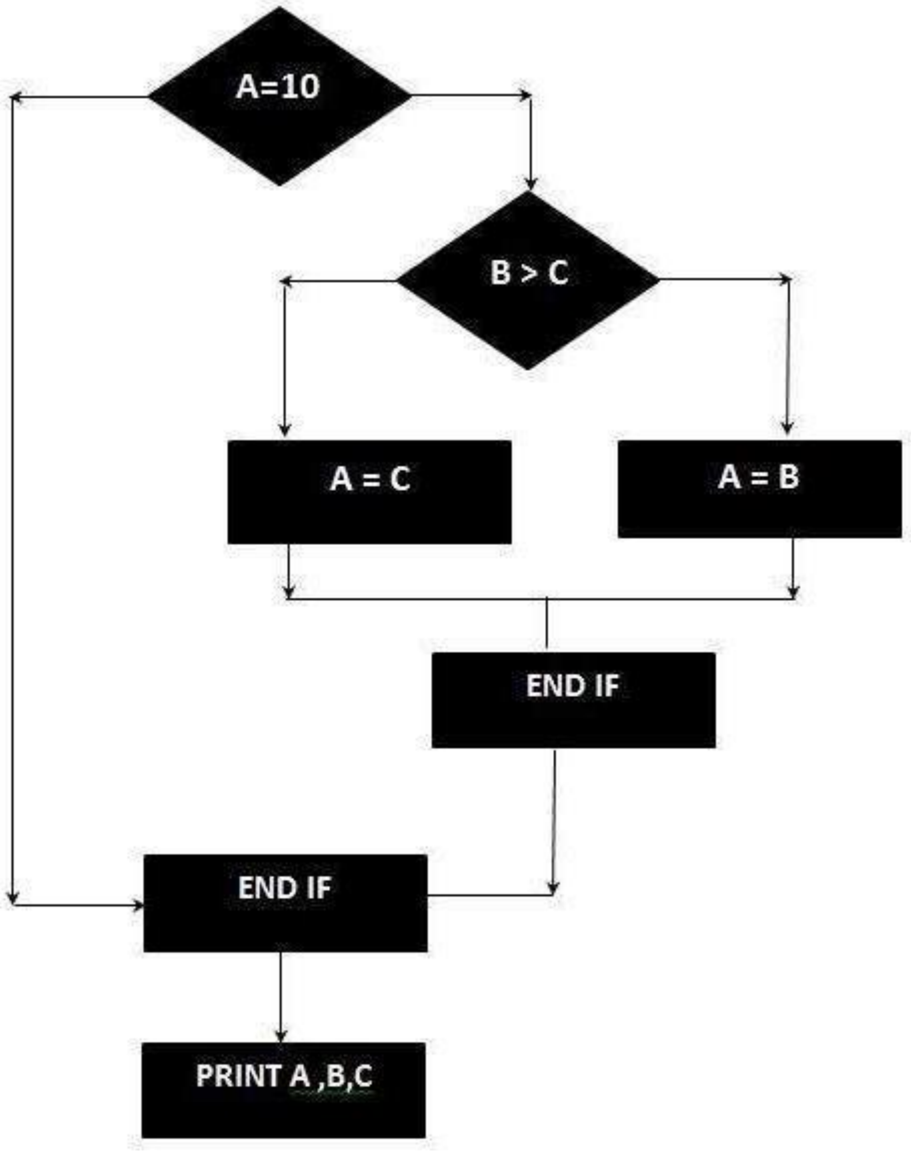


Fig. 3.1 Flow Graph

The Cyclomatic complexity is calculated using the above control flow diagram that shows seven nodes (shapes) and eight edges (lines), hence the cyclomatic complexity is 8 - 7 + 2 =3

3.9 DESIGN METRICS

Line of Code, Function Point, and Token Count are important metrics to find out the effort and time required to complete the project. There are some Data Structure metrics to compute the effort and time required to complete the project. There metrics are:

- The Amount of Data.
- The Usage of data within a Module.
- Program weakness.
- The sharing of Data among Modules.

1. **The Amount of Data**: To measure Amount of Data, there are further many different metrics and these are:
    - Number of variable (VARS): In this metric, Number of variables used in the program are counted.
    - Number of Operands (η2): In this metric, Number of operand used in the program are counted.

    $$\eta 2 = \text{VARS} + \text{Constants} + \text{Labels}$$

    - Total number of occurrence of variable (N2): In this metric, total number of occurrence of variables are computed
2. **The Usage of data within a Module**:
    The measure this metric, average number of live variables are computed. A variable is live from its first to its last references with in procedure.

    $$\text{Average no of Live variables (LV)} = \frac{\text{Sum of count live variables}}{\text{Sum of count of executable statements}}$$

3. **Program weakness**: Program weakness depends on its Modules weakness. If Modules are weak(less Cohesive), then it increase the effort and time metrics required to complete the project.

    $$\text{Average life of variables } (\gamma) = \frac{\text{Sum of count of live variables}}{\text{No. of unique variables}}$$

    Module Weakness (WM) = LV* γ

    Here, LV: average no. of live variables.

    γ: average life of variables.

    Program Weakness (WP) = (∑WM)/m

    Here, WM: weakness of module.

m: number of modules in the program.

4. The Sharing of Data among Module: As the data sharing between the Modules increases (higher Coupling), no parameter passing between Modules also increased, as a result more effort and time are required to complete the project. So Sharing Data among Module is an important metrics to calculate effort and time.

There are a number of reasons why software engineers and project managers require design metrics which measure quantitatively the quality of developed software designs. The first is as an aid for software project management. The ability to discover complex designs is useful for two reasons.

First, it enables project management to discover bad designs well before implementation and testing take place and enables redesign to be initiated.

Second, it enables designs which are complex to be discovered and adequate resources allocated to their implementation later in the software life-cycle.

The ability to quantify designs is also important because of the current shortage of experienced software designers. One of the important qualities that new software designers do not possess is an intuitive feel for good or bad software design. Design metrics would be able partly to replace intuition while the inexperienced software designer is learning.

Design metrics can act as an aid, for the evaluation of the claims made for the now very large numbers of software design methods. Claims such as: “Structured design is a set of proposed general program design considerations and techniques for making coding, debugging and modification easier, faster and less expensive by reducing complexity” and “effective system design requires a balance between data decomposition and process decomposition” must be explored, not only in terms of reported experience with software developed using the various design methods, but also in terms of universally accepted design metrics.

Another reason for establishing design metrics is as an additional criterion for customer acceptance of a software system. The current criteria are that a piece of software must meet its required functional and performance specifications. In cases where the software is to be maintained by the customer an additional criterion should be a design metric's acceptable value. A final reason for the use of metrics is concerned with evaluating designer productivity.

UNIT-4

SOFTWARE TESTING

4.1 SOFTWARE TESTING FUNDAMENTALS

Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to "demolish" the software that has been built. In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.

Software engineers are by their nature constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.

4.1.1 TESTING OBJECTIVES

The testing objectives are as follows:

1. Testing is a process of executing a program with the intent of finding an error.

2. A good test case is one that has a high probability of finding an as-yet undiscovered error.

3. A successful test is one that uncovers an as-yet-undiscovered error.

The objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort. If testing is conducted successfully, it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met. In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.

But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

4.1.2 TESTING PRINCIPLES

The testing principles are as follows:

• **All tests should be traceable to customer requirements.**

As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer’s point of view) are those that cause the program to fail to meet its requirements.

• **Tests should be planned long before testing begins.**
Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
• **The Pareto principle applies to software testing.**
The Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
• **Testing should begin “in the small” and progress toward testing “in the large.”**
The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
• **Exhaustive testing is not possible.**
The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
• **To be most effective, testing should be conducted by an independent third party.**
By *most effective*, we mean testing that has the highest probability of finding errors (the primary objective of testing). Thus, the software engineer who created the system is not the best person to conduct all tests for the software.

4.1.3 TESTABILITY

Software testability is simply how easily [a computer program] can be tested. Sometimes, testability is used to mean how adequately a particular set of tests will cover the product. It's also used by the military to mean how easily a tool can be checked and repaired in the field. Those two meanings are not the same as *software testability*. The checklist that follows provides a set of characteristics that lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."
- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development and testing).

Observability. "What you see is what you test."
- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).

- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

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

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

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

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

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

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

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

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

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

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

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

The **attributes of a "good" test** are as follows:

1. A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

2. A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).
For example, a module of the *SafeHome* software is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.

3. A good test should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

4. A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.
5.

4.2 TEST CASE DESIGN

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.
Any engineered product (and most other things) can be tested in one of two ways:
(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

The first test approach is called black-box testing and the second, white-box testing.
Black-box testing alludes to tests that are conducted at the software interface. Although they are designed to uncover errors, black-box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information (e.g., a database) is maintained. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

4.3 WHITE-BOX TESTING

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

A reasonable question might be posed at this juncture: "Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?" Stated another way, why don't we spend all of our energy on black-box tests? The answer lies in the nature of software defects:

• *Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.*
Errors tend to creep into our work when we design and implement function, conditions, or control that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.
• *We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.*
The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered
only once path testing commences.
• *Typographical errors are random.*
When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others
may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.
Each of these reasons provides an argument for conducting white-box tests.

4.4 BASIS PATH TESTING

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

4.4.1 Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 4.1. Each structured construct has a corresponding flow graph symbol.

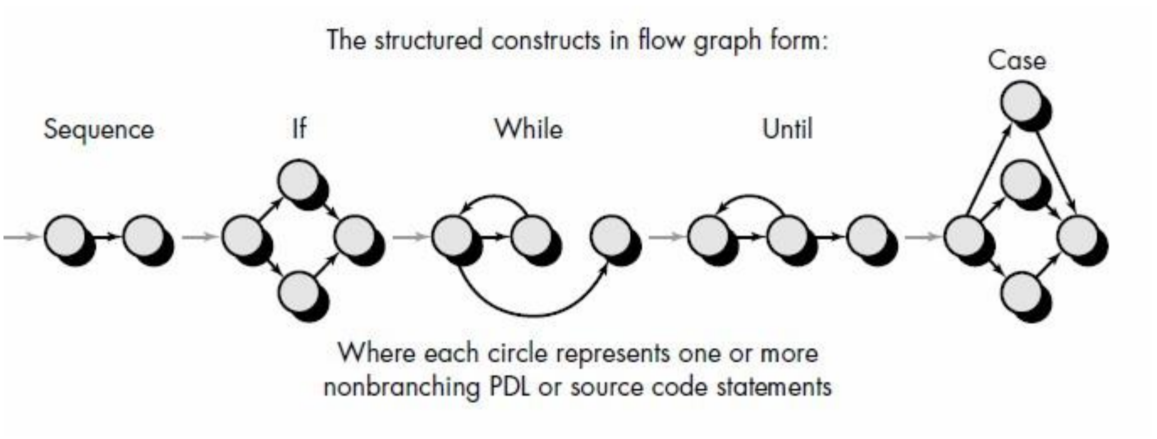

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Figure 4.1: Flow Graph Notation

To illustrate the use of a flow graph, we consider the procedural design representation in Figure 4.2 (a). Here, a flowchart is used to depict program control structure.

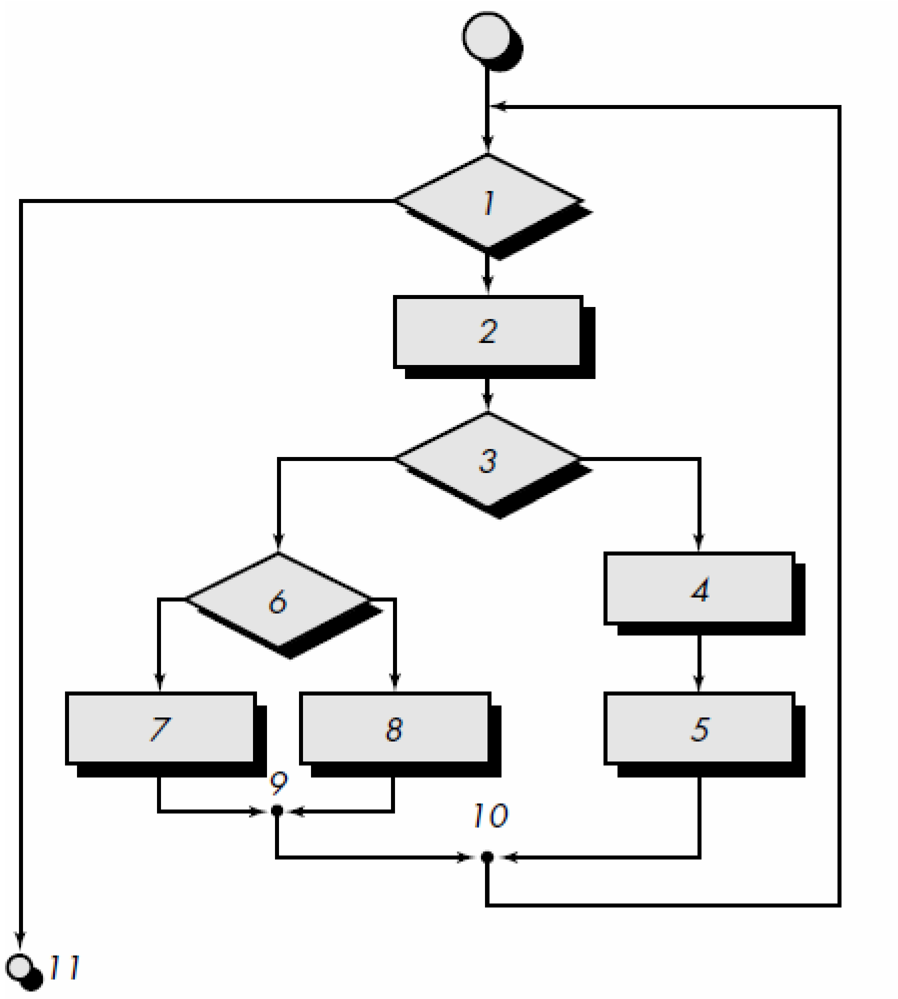


Figure 4.2 (a): Flow Chart

Figure 4.2 (b) maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to

Figure 4.2 (b), each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.

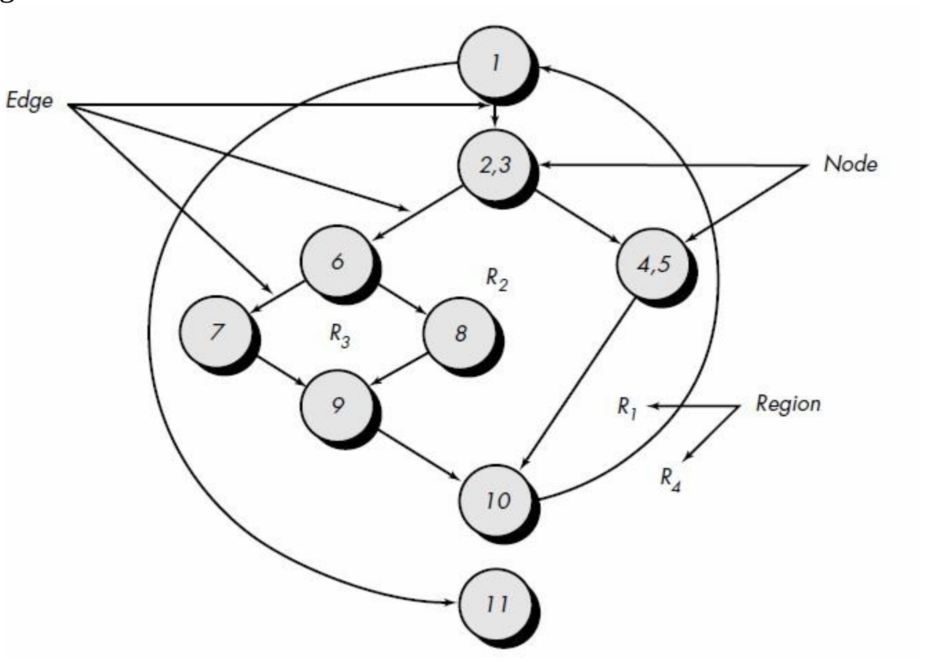


Figure 4.2 (b): Flow Graph

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 4.3, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

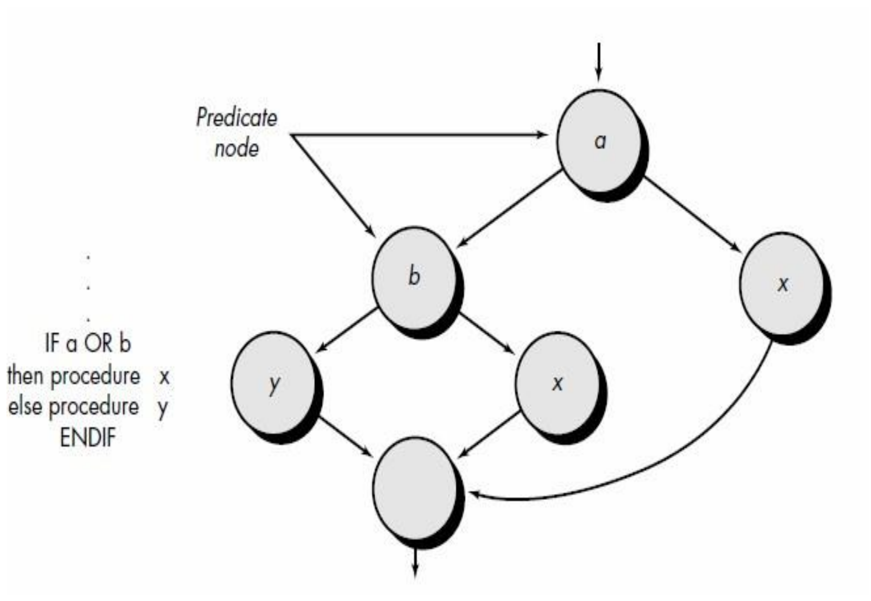


Figure 4.3: Compound Logic

4.4.2 Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 4.2 (b) is

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11

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

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in Figure 4.2 (b). That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

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

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

1. The number of regions of the flow graph correspond to the cyclomatic complexity.

2. Cyclomatic complexity, $V(G)$, for a flow graph, G, is defined as

$$V(G) = E - N + 2$$

where E is the number of flow graph edges, N is the number of flow graph nodes.

3. Cyclomatic complexity, $V(G)$, for a flow graph, G, is also defined as

$$V(G) = P + 1$$

where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 4.2 (b), the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. $V(G)$ = 11 edges _ 9 nodes + 2 = 4.

3. $V(G)$ = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 4.2 (b) is 4.

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

4.4.3 Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to derive the basis set:

1. **Using the design or code as a foundation, draw a corresponding flow graph.**
   A flow graph is created using the symbols and construction rules.

2. **Determine the cyclomatic complexity of the resultant flow graph.**

The cyclomatic complexity, $V(G)$, is determined by applying the algorithms. $V(G)$ can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1. In Figure 4.4,

$V(G)$ = 6 regions

$V(G)$ = 17 edges -13 nodes + 2 = 6

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

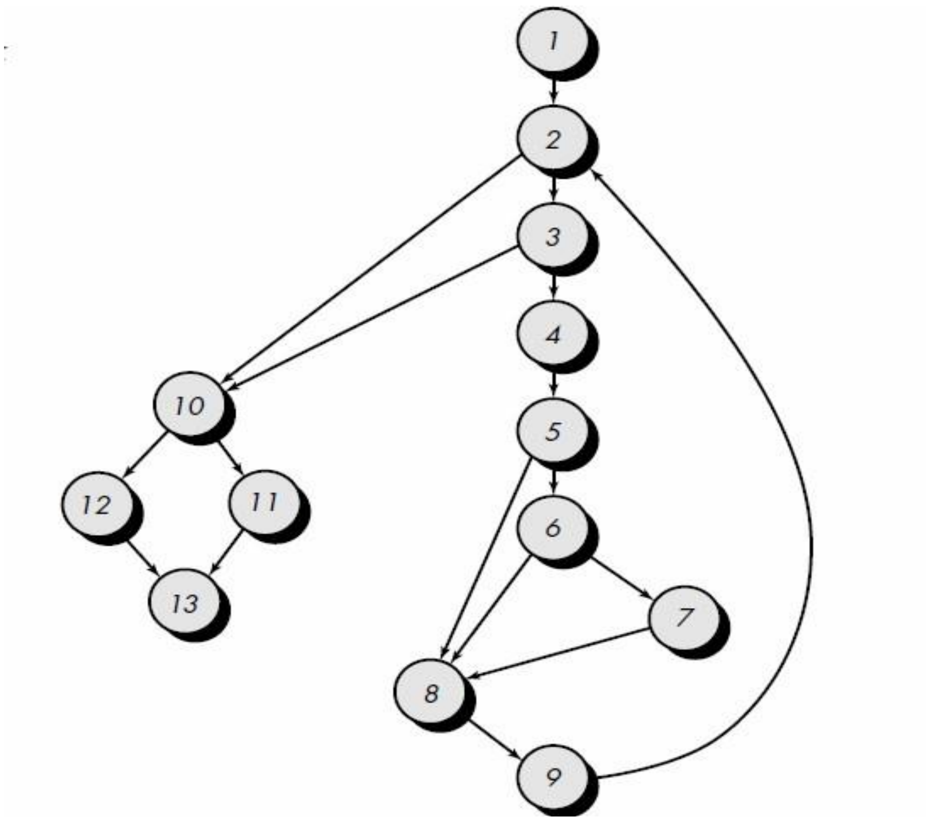


Figure 4.5: Flow Graph for the procedure average

3. Determine a basis set of linearly independent paths.

The value of *V*(*G*) provides the number of linearly independent paths through the program control
structure. In the case of procedure *average*, we expect to specify six paths:

path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

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

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

4. Prepare test cases that will force execution of each path in the basis set.

Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set just described are

Path 1 test case:

value(k) = valid input, where $k < i$ for $2 \leq i \leq 100$

value(i) = -999 where $2 \leq i \leq 100$

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

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = -999

Expected results: Average = -999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

Path 4 test case:

value(i) = valid input where i < 100

value(k) < minimum where $k < i$

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

Path 5 test case:

value(i) = valid input where $i < 100$

value(k) > maximum where $k <= i$

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

Path 6 test case:

value(i) = valid input where $i < 100$

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

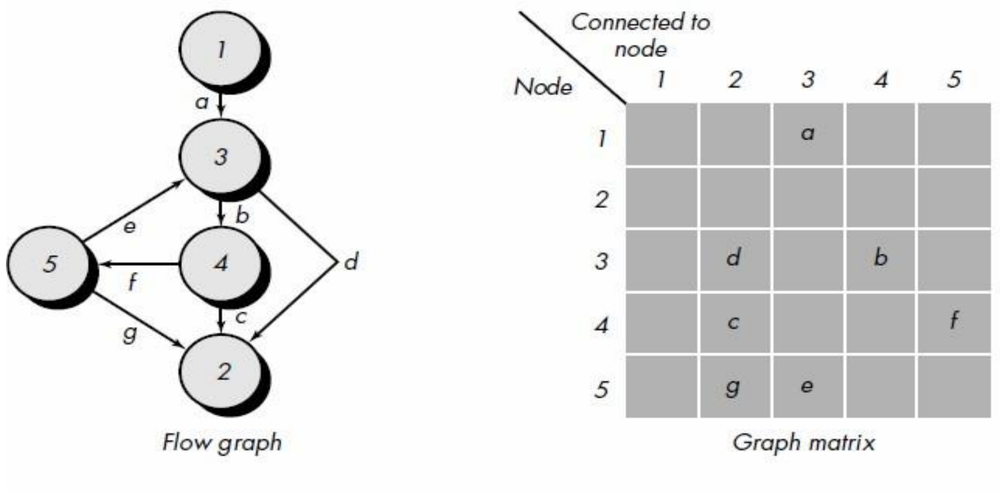
Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

4.4.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a *graph matrix,* can be quite useful.

A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix is shown in Figure 4.5. Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*.



Node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Figure 4.5: Graph Matrix

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

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

• The probability that a link (edge) will be executed.

• The processing time expended during traversal of a link.

• The memory required during traversal of a link.

• The resources required during traversal of a link.

To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in Figure 4.5 is redrawn as shown in Figure 4.6. Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity). Represented in this form, the graph matrix is called a *connection matrix.*

Referring to Figure 4.6, each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity.

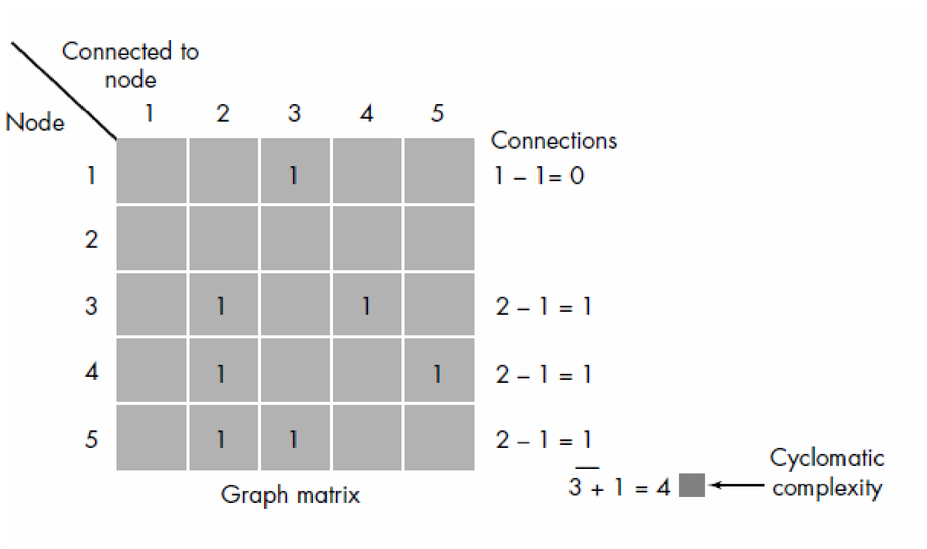


Figure 4.6: Connection Matrix

4.5 CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself.

4.5.1 Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E1 <relational-operator> *E2*

where *E1* and *E2* are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. Assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬). A condition without relational expressions is referred to as a *Boolean expression.*

Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression. If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

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

• Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies generally have two advantages. First, measurement of test coverage of a condition is simple. Second, the test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

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

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

Domain testing requires three or four tests to be derived for a relational expression. For a relational expression of the form

E1 <relational-operator> *E2*

three tests are required to make the value of *E1* greater than, equal to, or less than that of *E2*. If <relational-operator> is incorrect and *E1* and *E2* are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in *E1* and *E2*, a test that makes the value of *E1* greater or less than that of *E2* should make the difference between these two values as small
as possible.

For a Boolean expression with *n* variables, all of 2*n* possible tests are required ($n > 0$). This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if *n* is small.

4.5.2 Data Flow Testing

The *data flow testing* method selects test paths of a program according to the locations of definitions and uses of variables in the program. Assume that each statement in a program is 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* at statement *S* is said to be *live* at

statement *S'* if there exists a path from statement *S* to statement *S'* that contains no other definition of *X*.
A *definition-use* (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 statement *S* is live at statement *S'*.
One simple data flow testing strategy is to require that every DU chain be covered at least once. DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then- else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

4.5.3 Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.(Figure 4.7)

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.
1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. *m* passes through the loop where $m < n$.
5. *n* _1, *n*, *n* + 1 passes through the loop.

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Some suggestions to reduce the number of tests are:
1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

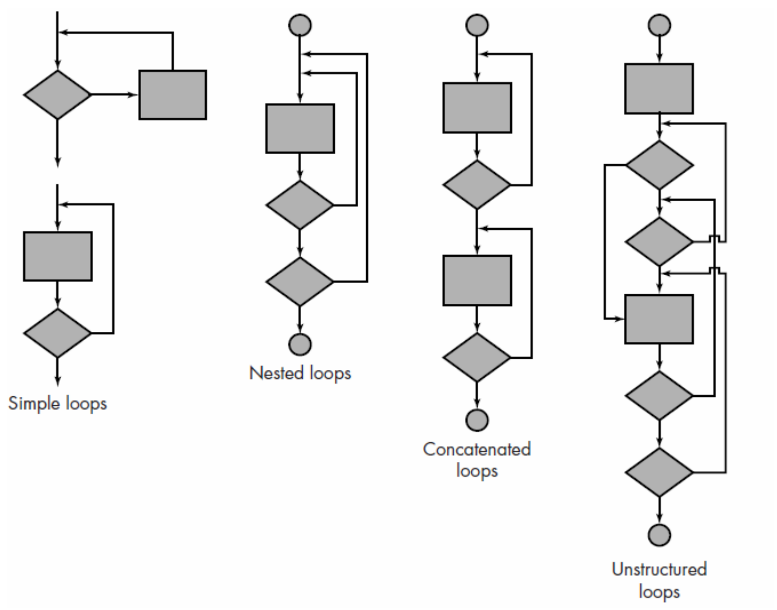


Figure 4.7: Classes of Loops

4.6 BLACK-BOX TESTING

Black-box testing, also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques.

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

4.6.1 Graph-Based Testing Methods

The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next

step is to define a series of tests that verify "all objects have the expected relationship to one another." Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.
To accomplish these steps, the software engineer begins by creating a *graph*—a collection of *nodes* that represent objects; *links* that represent the relationships between objects; *node weights* that describe the properties of a node (e.g., a specific data value or state behavior); and *link weights* that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure 4.8. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

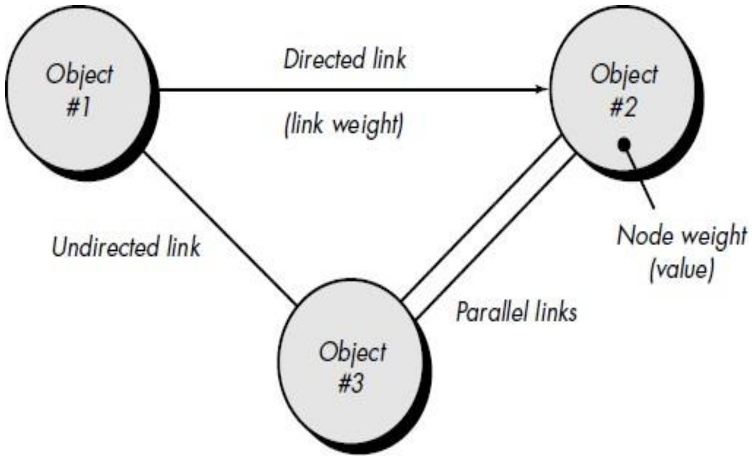


Figure 4.8: Graph Notation

Beizer describes a number of behavioral testing methods that can make use of graphs:

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

Finite state modeling. The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **order-information**

is verified during *inventory- availability look-up* and is followed by **customer-billing-information input**). The state transition diagram can be used to assist in creating graphs of this type.

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

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

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

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

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

4.6.2 Equivalence Partitioning

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

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean,* one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application. The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form

area code	—	blank or three-digit number
prefix	—	three-digit number not beginning with 0 or 1
suffix	—	four-digit number
password	—	six digit alphanumeric string
commands	—	check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking application can be specified as

area code: Input condition, *Boolean*—the area code may or may not be present.
Input condition, *range*—values defined between 200 and 999, with specific exceptions.

prefix: Input condition, *range*—specified value >200
Input condition, value—four-digit length

password: Input condition, *Boolean*—a password may or may not be present.
Input condition, *value*—six-character string.

command: Input condition, *set*—containing commands.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

4.6.3 Boundary Value Analysis

A greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

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

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

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

4.6.4 Comparison Testing

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

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

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

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

4.7 TESTING FOR SPECIALIZED ENVIRONMENTS, ARCHITECTURES, AND APPLICATIONS

As computer software has become more complex, the need for specialized testing approaches has also grown. The white-box and black-box testing methods are applicable across all environments, architectures, and applications, but unique guidelines and approaches to testing are sometimes warranted.

4.7.1 Testing GUIs

Graphical user interfaces (GUIs) present interesting challenges for software engineers. Because of reusable components provided as part of GUI development environments, the creation of the user interface has become less time consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI.

Due to the large number of permutations associated with GUI operations, testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.

4.7.2 Testing of Client/Server Architectures

Client/server (C/S) architectures represent a significant challenge for software testers. The distributed nature of client/server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of C/S architectures and the software that reside within them considerably more difficult than stand-alone applications.

4.7.3 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. Testing must also extend to the third element of the software configuration—documentation.
Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or an on-line help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that that documentation testing should be a meaningful part of every software test plan.
Documentation testing can be approached in two phases. The first phase, *review and inspection*, examines the document for editorial clarity. The second phase, *live test*, uses the documentation in conjunction with the use of the actual program. A live test for documentation can be approached using techniques that are analogous to many of the black- box testing methods

4.7.4 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test case designer have to consider white- and black-box test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN) with no error when the machine is making copies (in the "copying" state). These same operator interrupts, if input when the machine is in the "jammed" state, cause a display of the diagnostic code indicating the location of the jam to be lost (an error).

Task testing. The first step in the testing of real-time software is to test each task independently. That is, white-box and black-box tests are designed and executed for each task. Each task is executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.

Behavioral testing. Using system models created with CASE tools, it is possible to simulate the behavior of a real-time system and examine its behaviour as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built.

Intertask testing. Once errors in individual tasks and in system behaviour have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
System testing. Software and hardware are integrated and a full range of system tests are conducted in an attempt to uncover errors at the software/hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential.

4.8 SOFTWARE TESTING STRATEGIES

4.8.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as *verification and validation* (V&V). *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:
Verification: "Are we building the product right?"
Validation: "Are we building the right product?"
The definition of V&V encompasses many of the activities that we have referred to as *software quality assurance* (SQA).

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing.

4.8.2 A Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure 4.9. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

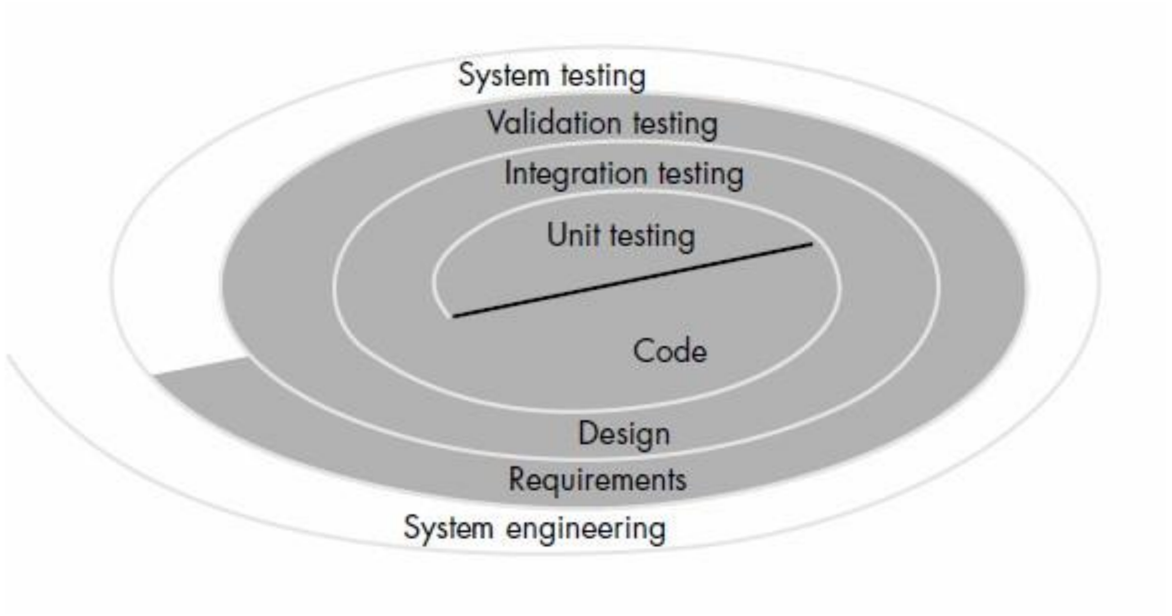


Figure 4.9: Testing Strategy

A strategy for software testing may also be viewed in the context of the spiral. *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter *validation testing,* where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the context sequentially. The steps are shown in Figure 4.10. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing*. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are the most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high- order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements. Black-box testing techniques are used exclusively during validation.

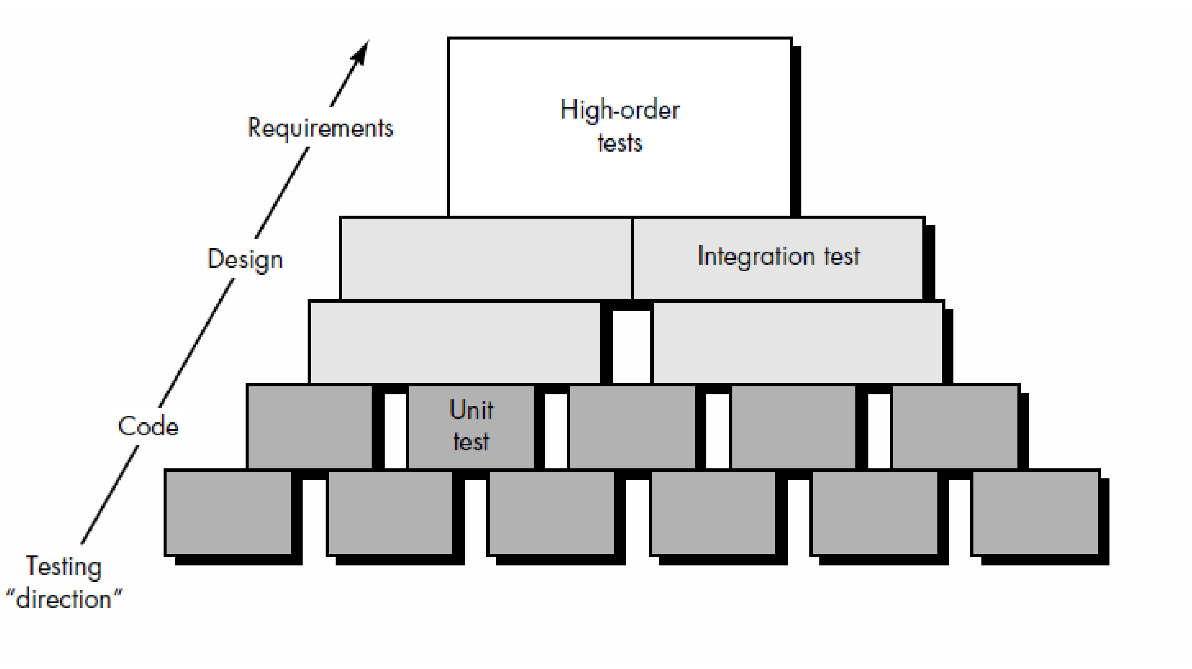


Figure 4.10: Software Testing Steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

4.9 UNIT TESTING

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

4.9.1 Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure 4.11. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

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

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

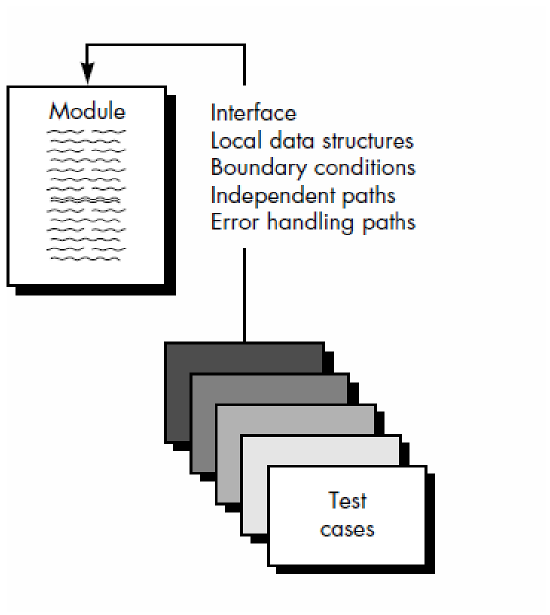


Figure 4.11: Unit Test

4.9.2 Unit Test Procedures

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins.

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure 4.12. In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

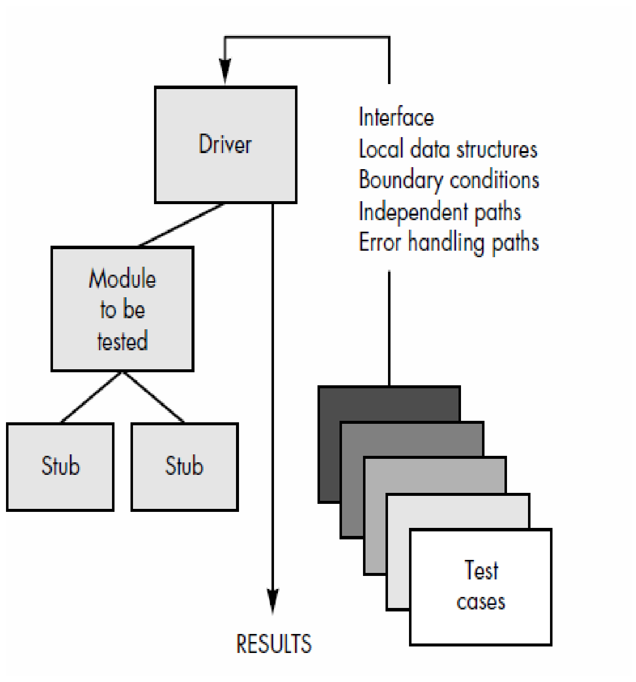


Figure 4.12: Unit Test Environment

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

4.10 INTEGRATION TESTING

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one module can have an inadvertent, adverse effect on another; sub-functions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems.

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design. There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

4.10.1 Top-down Integration

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

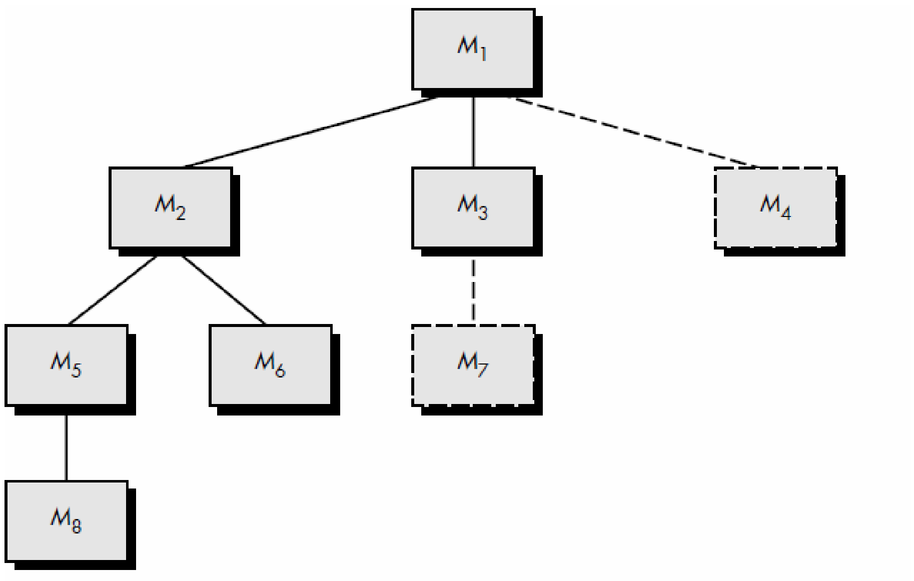


Figure 4.13: Top Down Integration

Referring to Figure 4.13, *depth-first integration* would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the lefthand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and righthand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

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

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

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

3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.
The process continues from step 2 until the entire program structure is built.

4.10.2 Bottom-up Integration

Bottom-up integration testing begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.
A bottom-up integration strategy may be implemented with the following steps:
1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

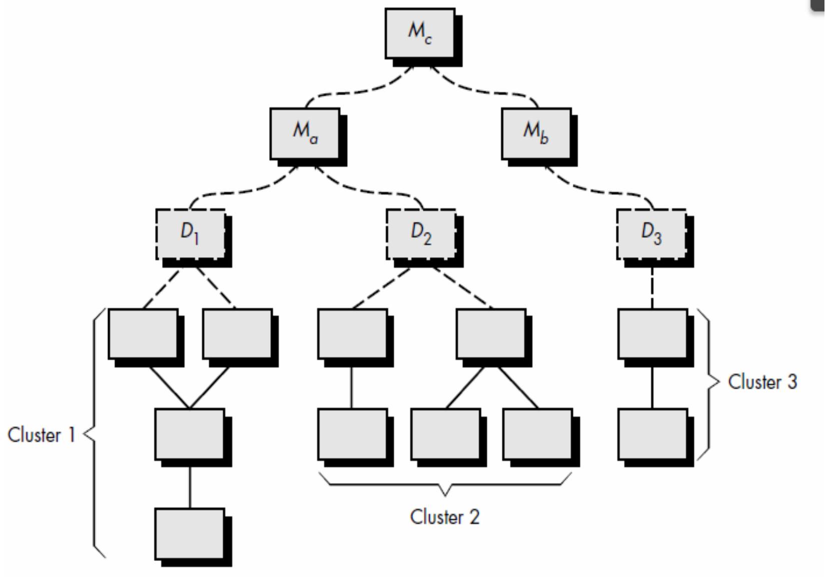


Figure 4.14: Bottom-Up Integration

Integration follows the pattern illustrated in Figure 4.14. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

4.10.3 Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.* Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

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

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

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

4.10.4 Smoke Testing

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

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

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

4.11 VALIDATION TESTING

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

4.11.1 Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance

requirements are attained, documentation is correct, and human engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
After each validation test case has been conducted, one of two possible conditions exist:
(1) The function or performance characteristics conform to specification and are accepted or (2) a deviation from specification is uncovered and a *deficiency list* is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

4.11.2 Alpha and Beta Testing

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

The *alpha test* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.
The *beta test* is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

4.12 SYSTEM TESTING

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

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

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

4.12.1 Recovery Testing

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

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

4.12.2 Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer : "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

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

4.12.3 Stress Testing

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

1. special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
2. input data rates may be increased by an order of magnitude to determine how input functions
3. will respond,
4. test cases that require maximum memory or other resources are executed,
5. test cases that may cause thrashing in a virtual operating system are designed,
6. test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called *sensitivity testing.* In some situations
(the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

4.12.4 Performance Testing

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

However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.
Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.

4.13 THE ART OF DEBUGGING

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

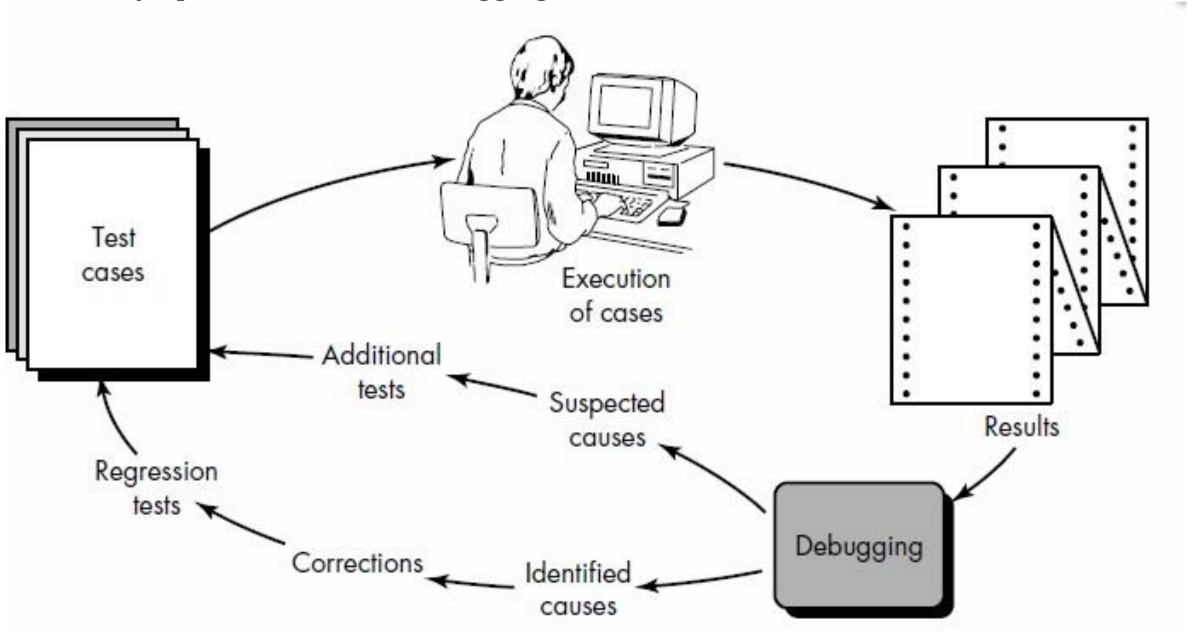


Figure 4.15: The debugging process

4.13.1 The Debugging Process

Debugging is not testing but always occurs as a consequence of testing. Referring to Figure 4.15, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non- corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

The few characteristics of bugs are:
1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, the errors are encountered that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g. the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

4.13.2 Debugging Approaches

In general, three categories for debugging approaches may be proposed:
(1) brute force, (2) backtracking, and (3) cause elimination.

The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. Brute force debugging methods are applied when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements.

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Each of these debugging approaches can be supplemented with debugging tools. We can apply a wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators, memory dumps, and cross-reference maps. However, tools are not a substitute for careful evaluation based on a complete software design document and clear source code.

4.14 SOFTWARE REENGINEERING

An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (the press of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software reengineering has been spawned by a software maintenance “iceberg” that has been building for more than three decades.

4.14.1 Software Maintenance

Thirty years ago, software maintenance was characterized as an "iceberg." In the early 1970s, the maintenance iceberg was big enough to sink an aircraft carrier. Today, it could easily sink the entire navy!

The maintenance of existing software can account for over 60 percent of all effort expended by a development organization, and the percentage continues to rise as more software is produced. Much of the software is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time [and most were not], they were created when program size and storage space were principle concerns. They were then migrated to new platforms, adjusted for changes in machine and operating system technology and enhanced to meet new user needs—all without enough regard to overall architecture.
The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running . . .

The ubiquitous nature of change underlies all software work. Change is inevitable when computer-based systems are built; therefore, we must develop mechanisms for evaluating, controlling, and making modifications.

4.14.2 A Software Reengineering Process Model

Reengineering takes time; it costs significant amounts of money; and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That’s why every organization needs a pragmatic strategy for software reengineering. A workable strategy is encompassed in a reengineering process model.

Reengineering is a rebuilding activity, and we can better understand the reengineering of information systems if we consider an analogous activity: the rebuilding of a house. Consider the following situation.
You have purchased a house in another state. You’ve never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?
• Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
• Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to “remodel” without rebuilding (at much lower cost and in much less time).

• Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
• If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
• If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

Although these principles focus on the rebuilding of a house, they apply equally well to the reengineering of computer-based systems and applications.

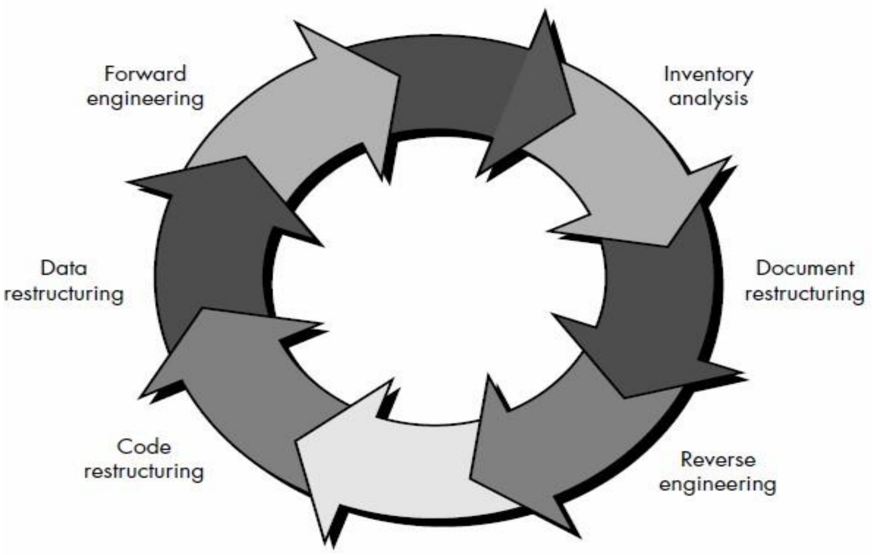


Figure 4.16: A Software Re-engineering process model

To implement these principles, a software reengineering process model that defines six activities is applied, shown in Figure 4.16. In some cases, these activities occur in a linear sequence, but this is not always the case. For example, it may be that reverse engineering (understanding the internal workings of a program) may have to occur before document restructuring can commence.
The reengineering paradigm shown in the figure is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited. For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis. Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

Document restructuring. Weak documentation is the trademark of many legacy systems. But what do we do about it? What are our options?

1. *Creating documentation is far too time consuming. If the system works, we'll live with what we have.* In some cases, this is the correct approach. It is not possible to re-create documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!

2. *Documentation must be updated, but we have limited resources. We'll use a "document when touched" approach.* It may not be necessary to fully re-document an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.

3. *The system is business critical and must be fully redocumented.* Even in this case, an intelligent approach is to pare documentation to an essential minimum. Each of these options is viable. A software organization must choose the one that is most appropriate for each case.

Reverse engineering. The term *reverse engineering* has its origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets." These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained. But these documents are proprietary and unavailable to the company doing the reverse engineering. In essence, successful reverse engineering derives one or more design and manufacturing specifications for a product by examining actual specimens of the product.

Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring. The most common type of reengineering is code restructuring. Some legacy systems have a relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured. To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured (this can be done automatically). The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring. A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program that the source code itself. Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Data objects and attributes are identified, and existing data structures are reviewed for quality.
When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.
Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward engineering. In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality.

Forward engineering, also called *renovation* or *reclamation*, not only recovers design information from existing software, but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software reimplements the function of the existing system and also adds new functions and/or improves overall performance.

4.15 REVERSE ENGINEERING

Reverse engineering conjures an image of the "magic slot." We feed an unstructured, undocumented source listing into the slot and out the other end comes full documentation for the computer program. Unfortunately, the magic slot doesn't exist. Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable.

The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations (a low-level abstraction), program and data structure information (a somewhat higher level of abstraction), data and control flow models (a relatively high level of abstraction), and entity relationship models (a high level of abstraction). As the abstraction level increases, the software engineer is provided with information that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple data flow representations may also be derived, but it is far more difficult to develop a complete set of data flow diagrams or entity-relationship models.

Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

If the *directionality* of the reverse engineering process is one way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.

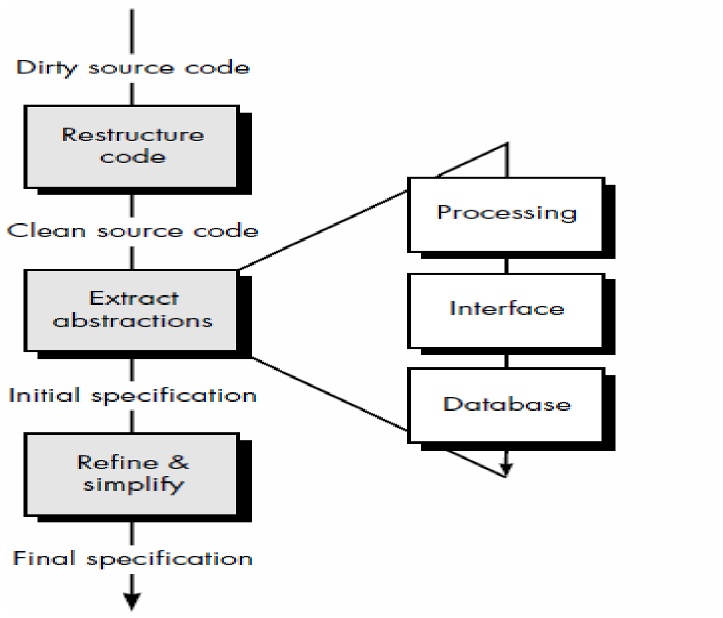


Figure 4.17: The Reverse Engineering process

The reverse engineering process is represented in Figure 4.17. Before reverse engineering activities can commence, unstructured (“dirty”) source code is *restructured* so that it contains only the structured programming constructs. This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.

The core of reverse engineering is an activity called *extract abstractions.* The engineer must evaluate the old program and from the (often undocumented) source code, extract a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

4.15.1 Reverse Engineering to Understand Processing

The first real reverse engineering activity begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application system must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within the system.

4.15.2 Reverse Engineering to Understand Data

Reverse engineering of data occurs at different levels of abstraction. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems).

Internal data structures. Reverse engineering techniques for internal program data focus on the definition of classes of objects. This is accomplished by examining the program code with the intent of grouping related program variables

Database structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

4.15.3 Reverse Engineering User Interfaces

Sophisticated GUIs have become de rigueur for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur. To fully understand an existing user interface (UI), the structure and behavior of the interface must be specified.

4.16 RESTRUCTURING

Software restructuring modifies source code and/or data in an effort to make it amenable to future changes. In general, restructuring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules. If the restructuring effort extends beyond module boundaries and encompasses the software architecture, restructuring becomes forward engineering.
Arnold defines a number of benefits that can be achieved when software is restructured:

• Programs have higher quality—better documentation, less complexity, and conformance to modern software engineering practices and standards.
• Frustration among software engineers who must work on the program is reduced, thereby improving productivity and making learning easier.
• Effort required to perform maintenance activities is reduced.
• Software is easier to test and debug.

4.16.1 Code Restructuring

Code restructuring is performed to yield a design that produces the same function but with higher quality than the original program. In general, code restructuring techniques model program logic using Boolean algebra and then apply a series of transformation rules that yield restructured logic. The objective is to take "spaghetti-bowl" code and derive a procedural design that conforms to the structured programming philosophy.

Other restructuring techniques have also been proposed for use with reengineering tools. A *resource exchange diagram* maps each program module and the resources (data types, procedures and variables) that are exchanged between it and other modules. By creating representations of resource flow, the program architecture can be restructured to achieve minimum coupling among modules.

4.16.2 Data Restructuring

Before data restructuring can begin, a reverse engineering activity called *analysis of source code* must be conducted. All programming language statements that contain data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to extract data items and objects, to get information on data flow, and to understand the existing data structures that have been implemented. This activity is sometimes called *data analysis*.

Once data analysis has been completed, *data redesign* commences. In its simplest form, a *data record standardization* step clarifies data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format. Another form of redesign, called *data name rationalization,* ensures that all data naming conventions conform to local standards and that aliases are eliminated as data flow through the system.

When restructuring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

4.17 FORWARD ENGINEERING

A program with control flow that is the graphic equivalent of a bowl of spaghetti, with "modules" that are 2,000 statements long, with few meaningful comment lines in 290,000 source statements and no other documentation must be modified to accommodate changing user requirements. There are the following options:

1. We can struggle through modification after modification, fighting the implicit design and source code to implement the necessary changes.
2. We can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.
3. We can redesign, recode, and test those portions of the software that require modification, applying a software engineering approach to all revised segments.
4. We can completely redesign, recode, and test the program, using CASE (reengineering) tools to assist us in understanding the current design.
There is no single "correct" option. Circumstances may dictate the first option even if the others are more desirable.

Rather than waiting until a maintenance request is received, the development or support organization uses the results of inventory analysis to select a program that (1) will remain in use for a preselected number of years, (2) is currently being used successfully, and (3) is likely to undergo major modification or enhancement in the near future. Then, option 2, 3, or 4 is applied.

The forward engineering process applies software engineering principles, concepts, and methods to re-create an existing application. In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

4.17.1 Forward Engineering for Client/Server Architectures

Over the past decade many mainframe applications have been reengineered to accommodate client/server architectures. In essence, centralized computing resources (including software) are distributed among many client platforms. Although a variety of different distributed environments can be designed, the typical mainframe application that is reengineered into a client/server architecture has the following features:

- Application functionality migrates to each client computer.
- New GUI interfaces are implemented at the client sites.
- Database functions are allocated to the server.
- Specialized functionality (e.g., compute-intensive analysis) may remain at the server site.
- New communications, security, archiving, and control requirements must be established at both the client and server sites.

It is important to note that the migration from mainframe to c/s computing requires both business and software reengineering. In addition, an “enterprise network infrastructure” should be established.
Reengineering for c/s applications begins with a thorough analysis of the business environment that encompasses the existing mainframe. Three layers of abstraction can be identified. The database sits at the foundation of a client/server architecture and manages transactions and queries from server applications. Yet these transactions and queries must be controlled within the context of a set of business rules (defined by an existing or reengineered business process). Client applications provide targeted functionality to the user community.

4.17.2 Forward Engineering for Object-Oriented Architectures

Object-oriented software engineering has become the development paradigm of choice for many software organizations. But, what about existing applications that were developed using conventional methods? In some cases, the answer is to leave such applications “as is.” In others, older applications must be reengineered so that they can be easily integrated into large, object-oriented systems.

Reengineering conventional software into an object-oriented implementation uses many of the same techniques. First, the existing software is reverse engineered so that appropriate data, functional, and behavioural models can be created. If the reengineered system extends the functionality or behaviour of the original application, use-cases are created. The data models created during reverse engineering are then used in conjunction with CRC modeling to establish the basis for the definition of classes. Class hierarchies, object-relationship models, object-behavior models, and subsystems are defined, and object-oriented design commences.

As object-oriented forward engineering progresses from analysis to design, a CBSE process model can be invoked. If the existing application exists within a domain that is already populated by many object-oriented applications, it is likely that a robust component library exists and can be used during forward engineering.

For those classes that must be engineered from scratch, it may be possible to reuse algorithms and data structures from the existing conventional application. However, these must be redesigned to conform to the object-oriented architecture.

4.17.3 Forward Engineering User Interfaces

As applications migrate from the mainframe to the desktop, users are no longer willing to tolerate arcane, character-based user interfaces. In fact, a significant portion of all effort expended in the transition from mainframe to client/server computing can be spent in the reengineering of client application user interfaces.
Merlo and his colleagues suggest the following model for reengineering user interfaces:

1. **Understand the original interface and the data that move between it and the remainder of the application.**
   The intent is to understand how other elements of a program interact with existing code that implements the interface. If a new GUI is to be developed, the data that flow between the GUI and the remaining program must be consistent with the data that currently flow between the character-based interface and the program.
2. **Remodel the behavior implied by the existing interface into a series of abstractions that have meaning in the context of a GUI.**
   Although the mode of interaction may be radically different, the business behaviour exhibited by users of the old and new interfaces (when considered in terms of a usage

scenario) must remain the same. A redesigned interface must still allow a user to exhibit the appropriate business behavior. For example, when a database query is to be made, the old interface may require a long series of text-based commands to specify the query. The reengineered GUI may streamline the query to a small sequence of mouse picks, but the intent and content of the query remain unchanged.

3. **Introduce improvements that make the mode of interaction more efficient.**
   The ergonomic failings of the existing interface are studied and corrected in the design of the new GUI.
4. **Build and integrate the new GUI.**
   The existence of class libraries and fourth generation tools can reduce the effort required to build the GUI significantly. However, integration with existing application software can be more time consuming. Care must be taken to ensure that the GUI does not propagate adverse side effects into the remainder of the application.