

➤ What is Software Engineering?

The term software engineering is the product of two words, software, and engineering.

The software is a collection of integrated programs. Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages.

Computer programs and related documentation such as requirements, design models and user manuals

Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc. **Software Engineering** is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Why is Software Engineering required?

Software Engineering is required due to the following reasons:

- To manage Large software ○ For more Scalability ○ Cost Management
- To manage the dynamic nature of software
- For better quality Management Need of Software Engineering
- The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.
- **Huge Programming:** It is simpler to manufacture a wall than to a house or building, similarly, as the measure of programming become extensive engineering has to step to give it a scientific process.
- **Adaptability:** If the software procedure were not based on scientific and engineering ideas, it would be simpler to re-create new software than to scale an existing one.
- **Cost:** As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the proper process is not adapted.
- **Dynamic Nature:** The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one.
- **Quality Management:** Better procedure of software development provides a better and quality software product.

Characteristics of a good software engineer

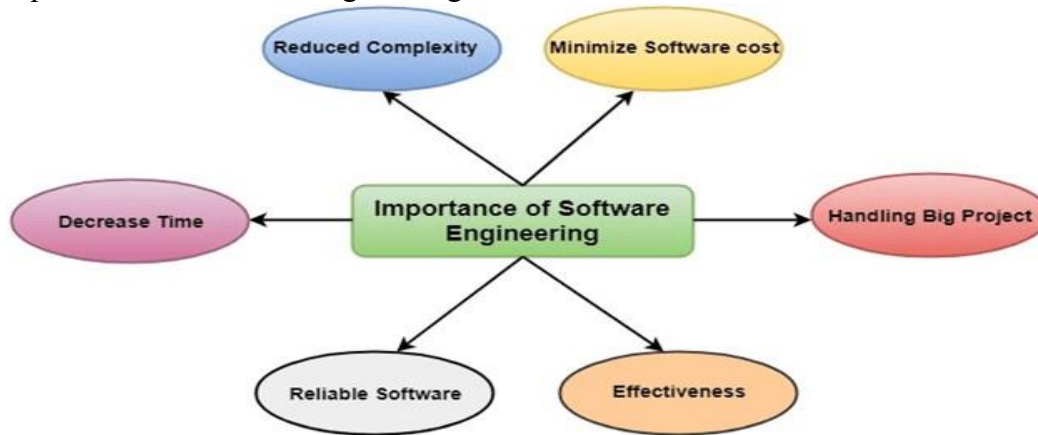
Exposure to systematic methods, i.e., familiarity with software engineering principles. □

Good technical knowledge of the project range (Domain knowledge).

- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team



Importance of Software Engineering



The importance of Software engineering is as follows:

1. **Reduces complexity:** Big software is always complicated and challenging to progress. Software engineering has a great solution to reduce the complication of any project. Software engineering divides big problems into various small issues. And then start solving each small issue one by one. All these small problems are solved independently to each other.
2. **To minimize software cost:** Software needs a lot of hardwork and software engineers are highly paid experts. A lot of manpower is required to develop software with a large number of codes. But in software engineering, programmers project everything and decrease all those things that are not needed. In turn, the cost for software productions becomes less as compared to any software that does not use software engineering method.
3. **To decrease time:** Anything that is not made according to the project always wastes time. And if you are making great software, then you may need to run many codes to get the definitive running code. This is a very time-consuming procedure, and if it is not well handled, then this can take a lot of time. So if you are making your software according to the software engineering method, then it will decrease a lot of time.
4. **Handling big projects:** Big projects are not done in a couple of days, and they need lots of patience, planning, and management. And to invest six and seven months of any company, it requires heaps of planning, direction, testing, and maintenance. No one can say that he has given four months of a company to the task, and the project is still in its first stage. Because the company has provided many resources to the plan and it should be completed. So to handle a big project without any problem, the company has to go for a software engineering method.
5. **Reliable software:** Software should be secure, means if you have delivered the software, then it should work for at least its given time or subscription. And if any bugs come in the software, the company is responsible for solving all these bugs. Because in software engineering, testing and maintenance are given, so there is no worry of its reliability.
6. **Effectiveness:** Effectiveness comes if anything has made according to the standards. Software standards are the big target of companies to make it more effective. So Software becomes more effective in the act with the help of software engineering.

Software myths are misleading attitudes that have caused serious problems for managers and technical people alike.

➤ Software Myths

Software myths propagate misinformation and confusion. There are three kinds of software myths: **1) Management myths:** Managers with software responsibility are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Following are the Management Myths:

- **Myth:** We already have a book that's full of standards and procedures for building software; won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but isn't used. Most software practitioners aren't aware of its existence. Also, it doesn't reflect modern software engineering practices and is also complete. □ **Myth:** My people have state-of-the-art software development tools; after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

- **Myth:** If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

- **Myth:** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

2) Customer myths: Customer myths lead to false expectations (by the customer) and ultimately,

dissatisfaction with the developer. Following are the customer myths: □

Myth: A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the functions, behavior, performance, interfaces, design constraints, and validation criteria is essential.

□ **Myth:** Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause heavy additional costs. Change, when requested after software is in production, can be much more expensive than the same change requested earlier

3) Practitioner's myths: Practitioners have following myths

Myth: Once we write the program and get it to work, our job is done. **Reality:** Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program “running” I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many element. Documentation provides a foundation for successful engineering and more importantly, guidance for software support.

Myth: Software engineering will make us creates voluminous and unnecessary documentation and will invariably slow us down.

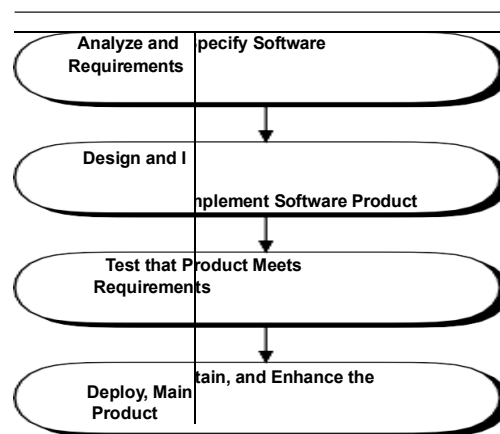
Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced reworks results in faster delivery times.

➤ Software Engineering Processes

In order for software to be consistently well engineered, its development must be conducted in an orderly process. It is sometimes possible for a small software product to be developed without a well-defined process. However, for a software project of any substantial size, involving more than a few people, a good process is essential. The process can be viewed as a road map by which the project participants understand where they are going and how they are going to get there. There is general agreement among software engineers on the major steps of a software process. Figure 1 is a graphical depiction of these steps. As discussed in Chapter 1, the first three steps in the process diagram coincide with the basic steps of the problem solving process, as shown in Table 4. The fourth step in the process is the postdevelopment phase, where the product is deployed to its users, maintained as necessary, and enhanced to meet evolving requirements.

The first two steps of the process are often referred to, respectively, as the "what and how" of software development. The "Analyze and Specify" step defines *what* the problem is to be solved; the "Design and Implement" step entails *how* the problem is solved.

Figure 1: Diagram of general software process steps.



Problem-Solving Phase	Software Process Step
Define the Problem	Analyze and Specify Software Requirements
Solve the Problem	Design and Implement Software Product
Verify the Solution	Test that Product Meets Requirements
	Deploy, Maintain, and Enhance the Product

Table 4: Correspondence between problem-solving and software processes.

While these steps are common in most definitions of software process, there are wide variations in how process details are defined. The variations stem from the kind of software being developed and the people doing the development. For example, the process for developing a well-understood business application with a highly experienced team can be quite different from the process of developing an experimental artificial intelligence program with a group of academic researchers.

Among authors who write about software engineering processes, there is a good deal of variation in process details. There is variation in terminology, how processes are structured, and the emphasis placed on different aspects of the process. This chapter will define key process terminology and present a specific process that is generally applicable to a range of end-user software. The chapter will also discuss alternative approaches to defining software engineering processes.

Independent of technical details, there are general quality criteria that apply to any good process. These criteria include the following:

1. The process is suited to the people involved in a project and the type of software being developed.

2. All project participants clearly understand the process, or at minimum the part of the process in which they are directly involved.
3. If possible, the process is defined based on the experience of engineers who have participated in successful projects in the past, in an application domain similar to the project at hand.
4. The process is subject to regular evaluation, so that adjustments can be made as necessary during a project, and so the process can be improved for future projects. With neat graphs and tables, the software development process is intended to appear quite orderly. In actual practice, the process can get messy. Developing software often involves people of diverse backgrounds, varying skills, and differing viewpoints on the product to be developed. Added to this are the facts that software projects can take a long time to complete and cost a lot of money. Given these facts, software development can be quite challenging, and at times trying for those doing it.

Having a well-defined software process can help participants meet the challenges and minimize the trying times. However, any software process must be conducted by people who are willing and able to work effectively with one another. Effective human communication is absolutely essential to any software development project, whatever specific technical process is employed.

General Concepts of Software Processes

Before defining the process followed in the book, some general process concepts are introduced. These concepts will be useful in understanding the definition, as well as in the discussion of different approaches to defining software processes.

Process Terminology

The following terminology will be used in the presentation and discussion of this chapter:

- **software process:** a hierarchical collection of *process steps*; hierarchical means that a process step can in turn have *sub-steps*
- **process step:** one of the activities of a software process, for example "**Analyze and Specify Software Requirements**" is the first step in Figure 1 ; for clarity and consistency of definition, process steps are named with verbs or verb phrases
- **software artifact:** a software work product produced by a process step; for example, a requirements specification document is an artifact produced by the "**Analyze and Specify**" step; for clarity and consistency, process artifacts are named with nouns or noun phrases
- **ordered step:** a process step that is performed in a particular order in relation to other steps; the steps shown in Figure 1 are ordered, as indicated by the arrows in the diagram
- **pervasive step:** a process step that is performed continuously or at regularly-scheduled intervals throughout the ordered process; for example, process steps to perform project management tasks are pervasive, since management is a continuous ongoing activity
- **process enactment:** the activity of performing a process; most process steps are enacted by people, but some can be automated and enacted by a software development tool
- **step precondition:** a condition that must be true before a process step is enacted; for example, a precondition for the "**Design and Implement**" step could be that the requirements specification is signed off by the customer
- **step post condition:** a condition that is true after a process step is enacted; for example, a post condition for the "**Design and Implement**" step is that the implementation is complete and ready to be tested for final delivery.

In addition to these specific terms, there is certain general terminology that is used quite commonly in software engineering textbooks and literature. In particular, the terms "analyze", "specify", "design", and "implement" appear nearly universally. While the use of these terms is widespread, their definitions are not always the same. In this book, these terms are given specific definitions in the context of the process that is

defined later in this chapter. This book's definitions here are consistent with mainstream usage, however the reader should be aware that specific definitions of these terms can vary among authors.

Process Structure

There are a variety of ways to depict a process. A typical graphical depiction uses a diagram with boxes and arrows, as shown in Figure 1. In this style of diagram, a process step is shown as a rounded box, and the order of steps is depicted with the arrowed lines. Process sub-steps are typically shown with a box expansion notation. For example, Figure 2 shows the expansion of the "**Analyze and Specify**" step. The activities of the first sub-step include general aspects of requirements analysis, such as defining the overall problem, identifying personnel, and studying related products. The second sub-step defines functional requirements for the way the software actually works, i.e., what functions it performs. The last sub-step defines non-functional requirements, such as how much the product will cost to develop and how long it will take to develop. This expansion is an over-simplification for now, since there are more than three substeps in "**Analyze and Specify**". A complete process expansion is coming up a bit later in this chapter.

- A more compact process notation uses mostly text, with indentation and small icons to depict sub-step expansion. Figure 3 shows a textual version of the general process, with the first step partially expanded, and other steps unexpanded. Right-pointing arrowheads depict an unexpanded process step. Down-

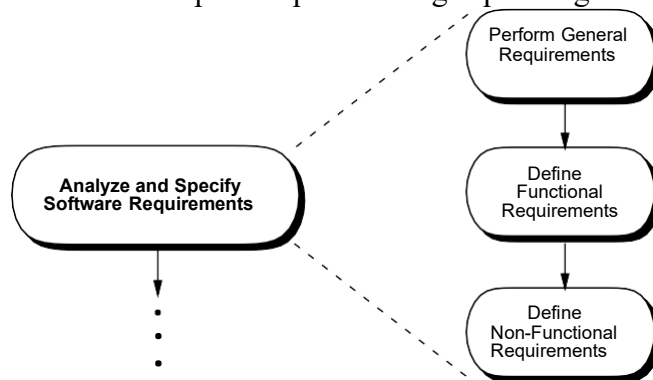


Figure 2: Expansion of the “Analyze and Specify” Step.

Analyze and Specify Software Requirements

Perform General Requirements Analysis

State Problem to be Solved Identify People Involved Analyze Operational
Setting Analyze Impacts Identify Positive Impacts Identify Negative Impacts
Analyze Related Systems Analyze Feasibility

Define Functional Requirements Define Non-Functional Requirements

Design and Implement Software Product

Test that Product Meets Requirements Deploy,

Maintain, and Enhance the Product

pointing arrow heads depict a process step with its sub-steps expanded immediately below. A round bullet depicts a process step that has no sub-steps.

Depending on the context, one or the other form of process depiction can be useful. When the emphasis is on the flow of the process, the graphical depiction can be most useful. To show complete process details, the textual depiction is generally more appropriate.

An important property of the textual depiction is that it can be considered unordered in terms of process step enactment. In the graphical process depiction, the directed lines connote a specific ordering of steps and sub-steps. The textual version can be considered more abstract, in that the top-to-bottom order of steps does not necessarily depict the specific order in which steps are enacted.

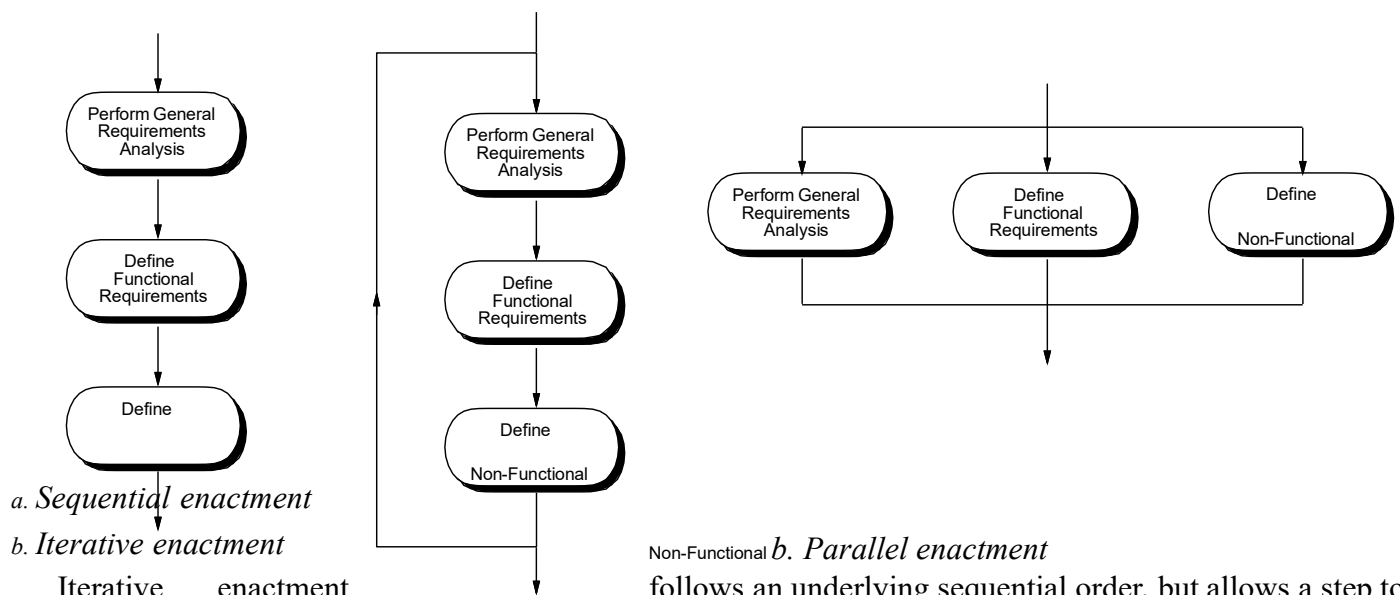
Given its abstractness, the textual depiction of a process can be considered the *canonical form*. Canonical form is a mathematical term meaning the standard or most basic form of something, for which other forms can exist. In the case of a software process, the canonical process form is the one most typically followed. The process can vary from its canonical form in terms of the order in which the steps are followed, and the number of times steps may be repeated.

Consider the three major sub-steps of under **Analyze and Specify** in Figure 3. The normal order of these steps is as listed in the figure. This means that "Perform General Requirements Analysis", is normally performed before "Define Functional Requirements" and "Define Non-Functional Requirements". However in some cases, it may be appropriate to analyze the non-functional requirements before the other steps, or to iterate through all three of the steps in several passes. The important point is that in abstracting out a particular enactment order, the textual process depiction allows the basic structure of the process to be separated from the order of enactment.

Styles of Process Enactment

Once the steps of a software process are defined, they can be enacted in different ways. The three general forms of ordered enactment are *sequential*, *iterative*, and *parallel*. These are illustrated in Figure 4 for the three sub-steps of the **Analyze and Specify** step.

Sequential enactment means that the steps are performed one after the other in a strictly sequential order. A preceding step must be completed before the following step begins. For the three steps in Figure a, this means that the general analysis is completed first, followed by functional requirements, followed by non-functional requirements.



Iterative enactment

Non-Functional

follows an underlying sequential order, but allows a step to be only partially completed before the following step begins. Then at the end of a sequence, the steps can be re-enacted to complete some additional work. When each step is fully completed, the entire sequence is done. In Figure b, some initial work on general analysis can be completed, enough to start the function requirements analysis. After some functional requirements are done, work on the non-functional requirements can begin. Then the three steps are repeated until each is complete.

Parallel enactment allows two or more steps to be performed at the same time, independent of one another. When the work of each step is completed, the process moves on to the subsequent steps.

Which of these enactment styles to use is determined by the mutual dependencies among the steps? For some projects, the determination may be made that a complete understanding of the general requirements is necessary before the functional and non-functional requirements begin. In this case, a strictly sequential order

is followed. In other projects, it may be determined that general requirements need only be partially understood initially, in which case an iterative order is appropriate.

In this particular example that deals with analysis, a purely parallel order is probably not appropriate, since at least some understanding of the general requirements is necessary before functional and non-functional requirements are analyzed. Given this, a hybrid process order can be employed, such as shown in Figure 5. In this hybrid style of enactment, a first pass at general analysis is performed. Then the functional and nonfunctional analysis proceed in parallel. The process then iterates back to refine the general requirements and then proceed with further functional and non-functional refinements.

The three styles of process enactment discussed so far apply to process steps that are performed in some order relative to one another. A fourth kind of enactment is *pervasive*. A pervasive process step is performed continuously throughout the entire process, or at regularly scheduled points in time. A good example of pervasive process steps are those related to project management. A well managed project will have regularly-scheduled meetings that occur on specific scheduled dates, independent of what specific ordered step developers may be conducting. The steps of the process dealing with project supervision occur essentially continuously, as the supervisors oversee developer's work, track progress, and ensure

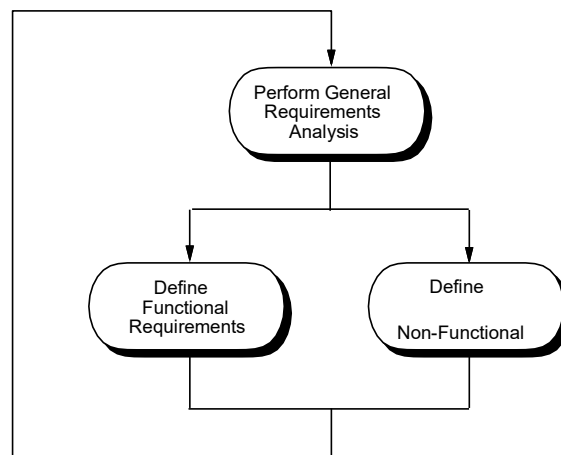


Figure 5: Hybrid process enactment. that the process is on schedule.

Testing is another part of the software process that can be considered to be pervasive. In some traditional models of software process, testing is an ordered step that comes near the end, after the implementation is complete. The process used in this book considers testing to be a pervasive step that is conducted at regularly schedule intervals, throughout all phases of development.

The people who make the determination of which style of enactment to use are those who define the process in the first place. Process definers are generally senior technical and management staff of the development team. These are the people who understand the type of software to be developed and the capabilities of the staff who will develop it. The remaining sections of this chapter contain further discussion on the rationale for choosing different styles of process enactment, as well as different overall process structures.

Defining a Software Process

This book presents and follows a specific software process. The purpose of presenting a particular process is three-fold:

- to define a process that is useful for a broad class of end-user software, including the example software system presented in the book
- to provide an organizational framework for presenting technical subject matter
- to give a concrete example of process definition, that can be used for guidance in defining other software processes

An important point to make at the outset is that this is "a" software process, not "the" process. There is in fact no single process that is universally applicable to all software. The process employed in this book is useful for a reasonably wide range of end-user products. However, this process, as any other, must be adapted to suit the needs of a particular development team working on a particular project. A good way to regard the process is as a representative example of process definition. Further discussion of process adaptation appears later in the chapter.

One of the most important things to say about software process is "use one that works". This means that technical details of a process and its methodologies are often less important than how well the process suits the project at hand. Above all, the process should be one that everyone thoroughly understands and can be productive using. There is no sense having management dictate a process from on high that the customers and technical staff cannot live with. The management and technical leaders who define a software process must understand well the people who will use it, and consult with them as necessary before, during, and after the establishment of a process. In order for all this to happen, the process must be clearly defined, which is what this chapter is about.

The top-level steps of the book's process are shown in Figure 6. These steps are a refinement of the general software process presented at the beginning of the chapter in Figure 1. The refined process has the following enhancements compared to the more general one:

- the "Analyze and Specify" step has been broken down into two separate steps;

- similarly, the "Design and Implement" step has been broken

"Design and Implement" step into two separate steps;

Ordered Steps:

Pervasive Steps:

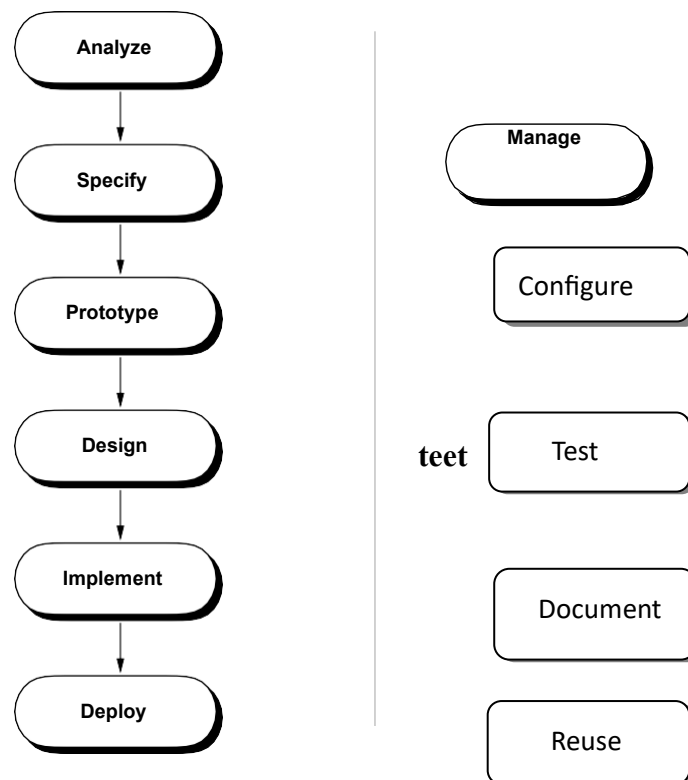


Figure 6: Top-level steps of the process used in the book.

- step names have been shortened to single words for convenient reference;
- prototyping and deployment steps have been added, details of which are discussed shortly;

- testing has been made a pervasive step of instead an ordered step following implementation; this signifies that testing will be carried out at regularly scheduled points throughout the process, not just after the implementation is completed;
- Additional pervasive steps have been added for the process activities that manage the software project, configure software artifacts, document the artifacts, and reuse existing artifacts.

From a problem solving perspective, the **Analyze** and **Specify** steps taken together constitute the problem definition phase; the **Design** and **Implement** steps together comprise the problem solution phase. The new **Prototype** step is a "pre-solution", where the developers rapidly produce a version of the product with reduced functionality. The purpose of the prototype is to investigate key product features before all of the details are finished. The **Deploy** step elevates the process from one of plain problem solving to one that delivers a working product to the end users, once the implementation is completed.

The Software Process Model

A software process model is a specified definition of a software process, which is presented from a particular perspective. Models, by their nature, are a simplification, so a software process model is an abstraction of the actual process, which is being described. Process models may contain activities, which

are part of the software process, software product, and the roles of people involved in software engineering. Some examples of the types of software process models that may be produced are:

1. **A workflow model:** This shows the series of activities in the process along with their inputs, outputs and dependencies. The activities in this model perform human actions.
2. **A dataflow or activity model:** This represents the process as a set of activities, each of which carries out some data transformations. It shows how the input to the process, such as a specification is converted to an output such as a design. The activities here may be at a lower level than activities in a workflow model. They may perform transformations carried out by people or by computers.
3. **A role/action model:** This means the roles of the people involved in the software process and the activities for which they are responsible.

There are several various general models or paradigms of software developmenta Program for Beginners

1. **The waterfall approach:** This takes the above activities and produces them as separate process phases such as requirements specification, software design, implementation, testing, and so on. After each stage is defined, it is "signed off" and development goes onto the following stage.
2. **Evolutionary development:** This method interleaves the activities of specification, development, and validation. An initial system is rapidly developed from a very abstract specification.
3. **Formal transformation:** This method is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods to a program. These transformations are 'correctness preserving.' This means that you can be sure that the developed programs meet its specification.
4. **System assembly from reusable components:** This method assumes the parts of the system already exist. The system development process target on integrating these parts rather than developing them from scratch

➤ Software Process Models

Prescriptive process models define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

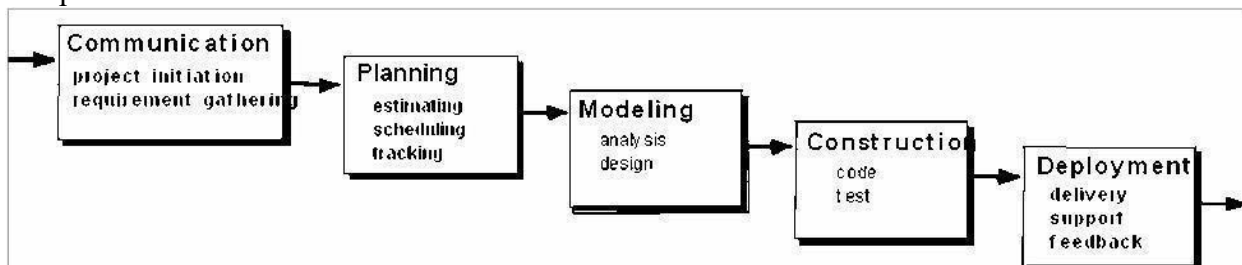
A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

THE WATERFALL MODEL:

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

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

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



The **problems** that are sometimes encountered when the waterfall model is applied are: Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

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

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

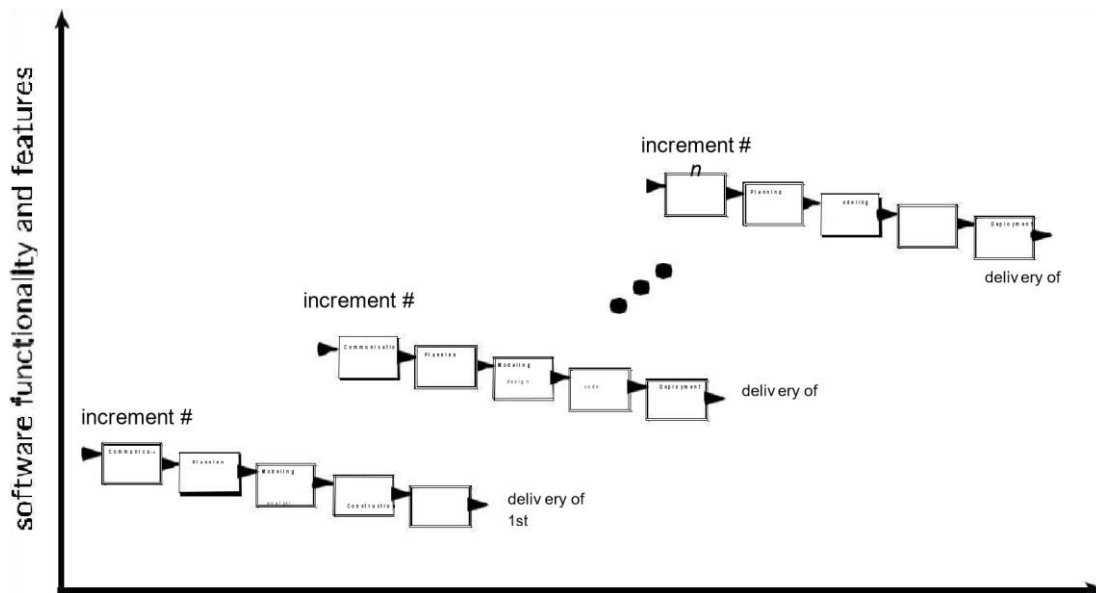
INCREMENTAL PROCESS MODELS:

The incremental model

The RAD model

THE INCREMENTAL MODEL:

Context: Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks.



The incremental model combines elements of the waterfall model applied in an iterative fashion.

The incremental model delivers a series of releases called increments that provide progressively more functionality for the customer as each increment is delivered.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed. The core product is used by the customer. As a result, a plan is developed for the next increment.

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.

This process is repeated following the delivery of each increment, until the complete product is produced.

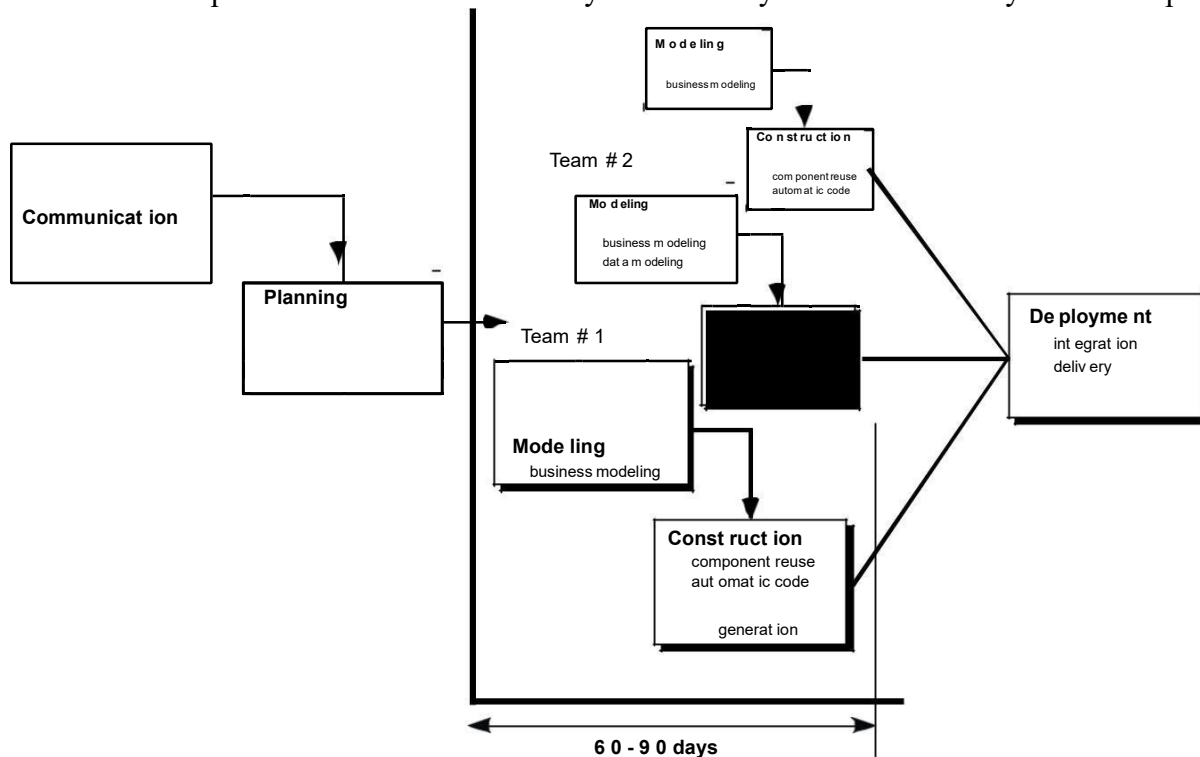
For *example*, word-processing software developed using the incremental paradigm might deliver basic file management editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

Difference: The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on delivery of an operational product with each increment.

THE RAD MODEL:

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a —high-speed‖ adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.

Context: If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a —fully functional system‖ within a very short time period.



The RAD approach maps into the generic framework activities.

Communication works to understand the business problem and the information characteristics that the software must accommodate.

Planning is essential because multiple software teams work in parallel on different system functions.

Modeling encompasses three major phases- business modeling, data modeling and process modeling- and establishes design representation that serve existing software components and the application of automatic code generation.

Deployment establishes a basis for subsequent iterations. The RAD approach has **drawbacks**:

For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail

If a system cannot be properly modularized, building the components necessary for RAD will be problematic

If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and RAD may not be appropriate when technical risks are high.

➤ Evolutionary process models:

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

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

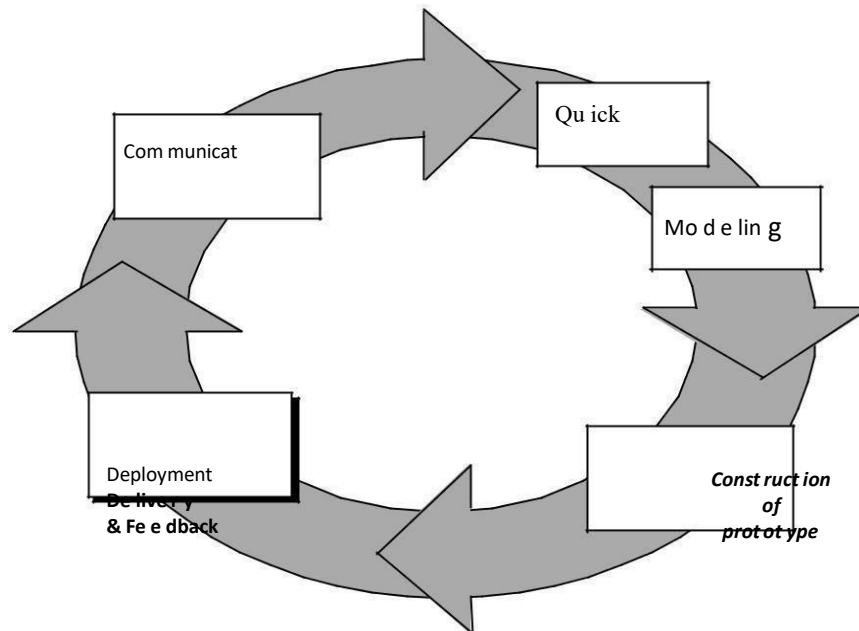
PROTOTYPING:

Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.



Context:

If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.

If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

Advantages:

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.

Prototyping can be **problematic** for the following reasons:

The customer sees what appears to be a working version of the software, unaware that the prototype is held together —with chewing gum and baling wire, unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be

rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that —a few fixes— be applied to make the prototype a working product. Too often, software development relents.

The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

THE SPIRAL MODEL

The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are

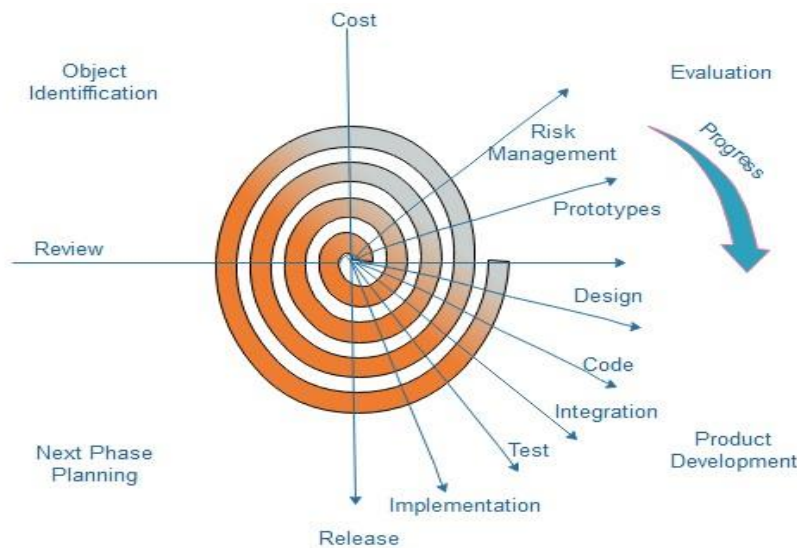


Fig. Spiral Model

Context: The spiral model can be adopted to apply throughout the entire life cycle of an application, from concept development to maintenance.

Advantages:

It provides the potential for rapid development of increasingly more complete versions of the software.

The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model uses prototyping as a risk reduction mechanism but, more importantly enables the developer to apply the prototyping approach at any stage in the evolution of the product.

Draw Backs:

The spiral model is not a panacea. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

➤ Component-Based Development (CBD)

Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing

Component-based development techniques involve procedures for developing software systems by choosing ideal off-the-shelf components and then assembling them using a well-defined software architecture. With the systematic reuse of coarse-grained components, CBD intends to deliver better quality and output

Component-based development is also known as component-based software engineering (CBSE).

The key goals of CBD are as follows

Save time and money when building large and complex systems: Developing complex software systems with the help of off-the-shelf components helps reduce software development time substantially. Function points or similar techniques can be used to verify the affordability of the existing method.

- Enhance the software quality: The component quality is the key factor behind the enhancement of software quality.
- Detect defects within the systems: The CBD strategy supports fault detection by testing the components; however, finding the source of defects is challenging in CBD. Some advantages of CBD include Minimized delivery:

- Search in component catalogs ◦ Recycling

of pre-fabricated components □ Improved

efficiency:

- Developers concentrate on application

development □ Improved quality: ◦ Component

developers can permit additional time to ensure quality

- Minimized expenditures

The specific routines of CBD are

Component development

- Component publishing
- Component lookup as well as retrieval
- Component analysis
- Component assembly

➤ Formal Methods Model

Definition

The formal methods model is an approach to software engineering that applies mathematical methods or techniques to the process of developing complex software systems. The approach uses a formal specification language to define each characteristic of the system. The language is very particular and employs a unique syntax whose components include objects, relations, and rules.

When used together, these components can validate the correctness of each characteristic. Think of the process like balancing a series of equations. At each step in the series, if the right-side of the equation doesn't equal the left, there's a problem that must be addressed.

Formal Methods Model: Steps

There are two steps that comprise the formal methods model. Those steps are the property-based specification and the model-based specification.

Property-Based Specification

Property-based specification describes two main elements in the system; those elements are the operations that can be performed on the system and the relationships between the operations. For example, consider a simple instant messaging application for your cell phone. Then some operations might be:

- Start up
- Send message
- Receive message
- Display message, and □ Shut down

The relationships between these operations might include:

- Startup must come before any other operation.
- Shut down must be the last operation performed.
- Display message comes during each send message and after each receive message.

Model-Based Specification

Model-based specification describes the states the system can be in and how the operations can transition the system from state to state. Consider the instant messaging application example mentioned earlier. States the system may be in might include the very similar-sounding states:

- Starting up
- Sending message
- Receiving message
- Displaying message, and □ Shutting down

As for transitions, they might include:

- Clicking the application icon to enter starting up
- Or, pressing the send button to leave the sending message state

Please note that these steps were described in familiar language rather than through the specification syntax for clarity's sake. With a formal specification, a non-mathematical, natural language description is transcribed into a formal language, which allows it to be rigorously checked for validity. As an example, the actual syntax to describe an instant message connection might look something like:

- message_connection: M System Connections
- assigned to: Instant Connections: System Connections
- available, sending, receiving: M System Connections

➤ Software Engineering- Fourth Generation Techniques

Implementation using a **4GL** (4th Generation Techniques) enables the software developer to represent desired results in a manner that leads to automatic generation of code to create those results. Obviously, a data structure with relevant information must exist and be readily accessible by the 4GL. To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

Software development environment that supports the 4GT paradigm includes some or all of the following tools

- 1) Non-procedural languages for database query
- 2) Report generation
- 3) Data manipulation
- 4) Screen interaction and definition
- 5) Code generation and High-level graphics capability
- 6) Spreadsheet capability
- 7) Automated generation of HTML and similar languages used for Web-site creation using advanced software tools

Pros and Cons Proponents claim dramatic reduction in software development time and greatly improved productivity for people who build software. Opponents claim that current 4GT tools are not all that much

easier to use than programming languages, that the resultant source code produced by such tools is "inefficient" and that the maintainability of large software systems developed using 4GT is open to question.

Advantages:

Simplified the programming process.

Use non-procedural languages that encourage users and programmers to specify the results they want, while the computers determines the sequence of instruction that will accomplish those results.

Use natural languages that impose no rigid grammatical rules.

Disadvantages:

Less flexible than other languages

Programs written in 4GLs are generally far less efficient during program execution than programs in highlevel languages.

Therefore, their use is limited to projects that do not call for such efficiency.

➤ **An Agile view of processes and Development:**

The meaning of Agile is swift or versatile. "**Agile process model**" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.



Fig. Agile Model

Phases of Agile Model:

Following are the phases in the Agile model are as follows:

1. Requirements gathering
2. Design the requirements
3. Construction/ iteration
4. Testing/ Quality assurance
5. Deployment
6. Feedback

- 1. Requirements gathering:** In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.
- 2. Design the requirements:** When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.
- 3. Construction/ iteration:** When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.
- 4. Testing:** In this phase, the Quality Assurance team examines the product's performance and looks for the bug.
- 5. Deployment:** In this phase, the team issues a product for the user's work environment.
- 6. Feedback:** After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Agile Testing Methods:

- Scrum ○ Crystal
 - Dynamic Software Development Method(DSDM) ○ Feature Driven Development(FDD) ○ Lean Software Development ○ extreme Programming(XP) Scrum
- SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.
- There are three roles in it, and their responsibilities are:
- **Scrum Master:** The scrum can set up the master team, arrange the meeting and remove obstacles for the process
 - **Product owner:** The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.
 - **Scrum Team:** The team manages its work and organizes the work to complete the sprint or cycle.

Extreme Programming(XP)

This type of methodology is used when customers are constantly changing demands or requirements, or when they are not sure about the system's performance.

Crystal:

There are three concepts of this method-

1. Chartering: Multi activities are involved in this phase such as making a development team, performing feasibility analysis, developing plans, etc.
2. Cyclic delivery: under this, two more cycles consist, these are: A. Team updates the release plan.
B. Integrated product delivers to the users.
3. Wrap up: According to the user environment, this phase performs deployment, post-deployment. Dynamic Software Development Method(DSDM):
DSDM is a rapid application development strategy for software development and gives an agile project distribution structure. The essential features of DSDM are that users must be actively connected, and teams have been given the right to make decisions. The techniques used in DSDM are:
 1. Time Boxing
 2. MoSCoW Rules
 3. Prototyping

The DSDM project contains seven stages:

1. Pre-project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design and build Iteration
6. Implementation
7. Post-project

Feature Driven Development (FDD):

This method focuses on "Designing and Building" features. In contrast to other smart methods, FDD describes the small steps of the work that should be obtained separately per function.

Lean Software Development:

Lean software development methodology follows the principle "just in time production." The lean method indicates the increasing speed of software development and reducing costs. Lean development can be summarized in seven phases.

1. Eliminating Waste
2. Amplifying learning
3. Defer commitment (deciding as late as possible)
4. Early delivery
5. Empowering the team
6. Building Integrity
7. Optimize the whole

When to use the Agile Model?

- When frequent changes are required.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with a software team all the time.
- When project size is small.

Advantage (Pros) of Agile Method:

1. Frequent Delivery
2. Face-to-Face Communication with clients.
3. Efficient design and fulfils the business requirement.
4. Anytime changes are acceptable.
5. It reduces total development time.

Disadvantages (Cons) of Agile Model:

1. Due to the shortage of formal documents, it creates confusion and crucial decisions taken throughout various phases can be misinterpreted at any time by different team members.
2. Due to the lack of proper documentation, once the project completes and the developers allotted to another project, maintenance of the finished project can become a difficulty.

➤ **Software Engineering Practices in Software Development**

A software engineer applies engineering practices to software development, and typically handles the overall system design of the software application. While some software engineers also handle programming, these engineers typically hand off the engineered designs to software programmers for coding. Once the coding is complete, they test the software and make sure it performs within the engineered requirements.

Software engineering is divided into several specializations, which focus on a particular software development area. A few of the most common software engineering specialties include:

- Requirements engineering looks at the overall requirements of software, how it fits into the current system, and whether the code from software programmers meets the necessary requirements. Software quality management fulfills a similar set of duties, although these software engineers may only be confirming the software meets someone else's set of requirements.

Software engineering process looks at the software development lifecycle and identifies areas of improvement and optimization. By engineering the development lifecycle itself, this software engineer ensures the entire development process runs more efficiently.

Software design covers one of the most common software engineering specialties. These engineers define everything from the software's infrastructure to its interface before programmers start on the code, in order to create software effective for the environment it's getting deployed in.

The typical software engineer has many duties and plays an integral part in the software design and development process:

- Look at proposed software development projects and determines whether the guidelines follow sound engineering practices. If they don't, the engineer addresses any problems and provides an updated project requirements list.

Creates documentation for the software design and development of the project, which allows the programmers to work effectively and any other engineers working on the project to understand what's going on with the design.

- Examines business needs to determine whether software can solve existing problems, fix operational inefficiencies, or otherwise optimize work processes. When these areas are identified, the software engineer begins working on a software design document to see whether the solution is feasible with the resources on hand.
- Software quality assurance and testing sometimes fall in the software engineering realm, especially if the engineer created the original design document.

Software engineers create needed efficiency in software development projects, which ensures the projects fit the needs of the company, stay within the project scope, and tests developed software to confirm it follows the engineering practices laid out in the initial design.

Software Engineering:

Software engineering is defined as a process of analyzing user requirements and then designing, building, and testing software application which will satisfy those requirements. Let's look at the various definitions of software engineering:

- IEEE, in its standard 610.12-1990, defines software engineering as the application of a systematic, disciplined, which is a computable approach for the development, operation, and maintenance of software.
- Fritz Bauer defined it as 'the establishment and used standard engineering principles. It helps you to obtain, economically, software which is reliable and works efficiently on the real machines'.
- Boehm defines software engineering, which involves, 'the practical application of scientific knowledge to the creative design and building of computer programs. It also includes associated documentation needed for developing, operating, and maintaining them. ➤ **Software Engineering Practice**

Practice is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed.

It represents the details—the technical considerations and how to's—that are below the surface of the software process—the things that you'll need to actually build high-quality computer software.

The Essence of Practice

This section lists the generic framework (communication, planning, modeling, construction, and deployment) and umbrella (tracking, risk management, reviews, measurement, configuration management,

reusability management, work product creation, and product) activities found in all software process models.

1. **Understand the problem** (communication and analysis). Who are the stakeholders?

What are the unknowns? “Data, functions, features to solve the problem?”

Can the problem be compartmentalized? “Smaller that may be easier to understand? Can the problem be represented graphically? Can an analysis model be created?

2. **Plan a solution** (modeling and software design).

Have you seen a similar problem before?

Has a similar problem been solved? If so, is the solution reusable?

Can sub-problems be defined?

Can you represent a solution in a manner that leads to effective implementation?

3. Carry out the plan (code generation). Does the solution conform to the plan?

Is each component part of the solution probably correct?

4. Examine the result for accuracy (testing and quality assurance).

Is it possible to test each component part of the solution?

Does the solution produce results that conform to the data, functions, features, and behavior that are required?

1. Core Principles

The Reason It All Exists: Provide value to the customer and the user. If you can’t provide value, then don’t do it.

KISS—Keep It Simple, Stupid! *All design should be as simple as possible, but no simpler.* This facilitates having a more easily understood and easily maintained system.

Maintain the product and project “vision.” *A clear vision is essential to the success of a S/W project.*

What you produce, others will consume. *Always specify, design, and implement knowing someone else have to understand what you are doing.*

Be open to the Future. *Never design yourself into a corner.* Always ask “what if,” and prepare yourself for all possible answers by creating systems that solve the general problem, not just the specific one.

Plan Ahead for Reuse. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

Think! Placing clear, complete thought before action almost always produces better results.

➤ **Communication Practices**

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a *communication* (also called *requirement elicitation*) activity.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront a S/W engineer.

In this context, the following are communication **principles** and concepts that apply to customer communication:

Listen: focus on the speaker’s words, rather than formulating your response to those words. Be a polite listener.

Prepare before you communicate: Spend the time to understand the problem before you meet with others “research”.

Someone should facilitate the communication activity. Have a leader “moderator” to keep the conversation moving in a productive direction.

Face-to-face communication is best.

Take notes and document decisions.

Collaborate with the customer. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Stay focused, modularize your discussion. The facilitator should keep the conversation modular; leaving one topic only after it has been resolved. Draw pictures when things are unclear.

(a) Once you agree to something, move on; (b) if you can't agree to something, move on; (c) if a feature or function is unclear and can't be clarified at the moment, move on.

➤ Planning Practices

The planning activity encompasses a set of management and technical practices that enable the S/W team to define a road map as it travels toward its strategic goal and tactical objectives.

Regardless of the rigor with which planning is conducted, the following principles always apply:

Understand the project scope. Scope provides the S/W team with a destination.

Involve the customer (and other stakeholders) in the planning activity. The customer defines priorities and establishes project constraints. S/W engineers must often negotiate order of delivery, timelines, and other related issues.

Recognize that planning is iterative. A plan must be adjusted to accommodate changes.

Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. **Consider risk as you define the plan.**

Be realistic. Even the best S/W engineers make mistakes.

Adjust granularity as you plan. A fine granularity plan provides significant work task detail that is planned over relatively short time increments. A coarse granularity plan provides broader work tasks that are planned over longer time periods. **Define how quality will be achieved.**

Define how you'll accommodate changes. "Can the customer request a change at any time?"

Track what you've planned and make adjustments as required.

Barry Boehm states: "You need an organizing principle that scales down to provide simple plans for simple projects."

Boehm suggests an approach that addresses project objectives, milestones, and schedules, responsibilities, management and technical approaches, and required resources.

Boehm calls it *W⁵HH principle*, after a series of questions that lead to a definition of key project characteristics and the resultant project plan.

Why is the system being developed? Does the business purpose justify the expenditure of people, time and money?

What will be done? Identify the functionality to be built.

When will it be accomplished? Establish a workflow and timeline for key project tasks and identify milestones required by the customer.

Who is responsible for a function? Define members' roles and responsibilities.

Where are they located (organizationally)? Customers also have responsibilities.

How will the job be done technically and managerially? Once a scope is defined, a technical strategy must be defined.

How much of each resource is needed? The answer is derived by developing estimates based on answers to earlier questions.

➤ Modeling Practices

The process of developing analysis and design models is described in this section. The emphasis is on describing how to gather the information needed to build reasonable models, but no specific modeling notations are presented in this chapter. UML and other modeling notations are described in detail later in the text.

In S/W Eng. work, two models are created: analysis models and design models.

Analysis models represent the customer requirements by depicting the S/W in three different domains: the information domain, the functional domain, and the behavioral domain.

Design models represent characteristics of the S/W that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Analysis Modeling Principles

The information domain of a problem must be represented and understood. The *information domain* encompasses the data that flow into the system (end-users, other systems, or external devices), the data that flow out of the system and the data stores that collect and organize persistent data objects.

Represent software functions. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Represent software behavior. The behavior of the S/W is driven by the interaction with the external environment.

The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion (or hierarchical).

The analysis task should move from essential information toward implementation detail. Analysis begins by describing the problem from the end-user perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented.

Design Modeling Principles
The software design model is the equivalent of an architect's plans for a house. Set of principles used:

Design must be traceable to the analysis model. The analysis model describes the information domain of the problem, user visible functions, system behavior, and a set of analysis classes that package business objects with the methods that service them.

The design model translates this information into an architecture: a set of subsystems that implement major functions, and a set of component-level designs that are the realization of analysis class.

Always consider architecture. S/W architecture is the skeleton of the system to be built. Only after the architecture is built should the component-level issues should be considered.

Focus on the design of data as it is as important as a design. Data design is an essential element of architectural design.

Interfaces (both user and internal) must be designed. A well designed interface makes integration easier and assists the tester in validating component functions.

User interface design should be tuned to the needs of the end-user. "Ease of use."

Component-level design should exhibit functional independence. The functionality that is delivered by a component should be *cohesive*- that is, it should focus on one and only one function.

Components should be loosely coupled to one another and to the external environment. *Coupling* is achieved in many ways – via a component interface, by messaging through global data. Coupling should be kept as low as is reasonable. As the level of coupling increases, error propagation also increases and the overall maintainability of the system decreases.

Design representation (models) should be easily understood.

The design model should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

➤ Construction Practices

In this text "construction" is defined as being composed of both coding and testing. The purpose of testing is to uncover defects. Exhaustive testing is not possible so processing a few test cases successfully

does not guarantee that you have bug free program. Unit testing of components and integration testing will be discussed in greater later in the text along with software quality assurance activities.

Although testing has received increased attention over the past decade, it is the weakest part of software engineering practice for most organizations.

Coding Principles and Concepts

Preparation Principles: Before writing one line of code, be sure of:

1. Understand the problem you are trying to solve.
2. Understand the basic design principles.
3. Pick a programming language that meets the needs of the S/W to be built and the environment in which it will operate.
4. Select a programming environment that provides tool that will make your work easier.
5. Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles: As you begin writing code, be sure you

1. Constrain your algorithm by following structured programming practice.
 2. Select the proper data structure.
 3. Understand the software architecture.
 4. Keep conditional logic as simple as possible.
 5. Create easily tested nested loops.
6. Write code that is self-documenting.
 7. Create a visual layout.

Validation Principles: After you've completed your first coding pass, be sure you 1.

Conduct a code walkthrough.

2. Perform unit test and correct errors.
3. Refactor the code.

Testing Principles

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

➤ Deployment Practices

Software development involves the process of creating applications and software programs by writing and maintaining the source code. It is about the complete process and the stages involved throughout the software development life cycle (SDLC).

The software development is a step by step process of inventing, specifying, coding, recording, testing and fixing bugs that is done to create and manage frameworks, software components or even to develop a complete application.

Customer Expectations for the software must be managed. "Don't promise more than you can deliver." A complete delivery package should be assembled and tested.

A support regime must be established before the software is delivered.

Appropriate instructional materials must be provided to end-users.

Buggy software should be fixed first, delivered later.

Best Deployment in Software Practices

Implement a deployment checklist

Set up a process while you deploy a new software. A checklist helps you to follow what must be done next so that you will not miss out any of the crucial steps **Choose the right Deployment Method**

Implement the software that is easy to integrate with the existing local applications and other tools.

Automated Software Deployment Process

Deployment of new versions of software manually is a daunting task that brings in a lot of possibilities of human errors. Automating the deployment process, mitigates the possibilities of errors, increases the deployment speed and streamlines the process

Adopt continuous delivery

Adopting Continuous Delivery ensures to enable the code for the required deployment. This is done by implementing the application in a proto-type environment to ensure if the application is good to function and meet the demands once deployed.

Use a continuous integration server

Continuous Server Integration is crucial for any successful agile deployment. This ensures that the developed program works on a developer's machine while the it helps you deny "integration hell".

Software engineers use product metrics to help them assess the quality of the design and construction of the software product being built. Product metrics provide software engineers with a basis to conduct analysis, design, coding, and testing more objectively. Qualitative criteria for assessing software quality are not always sufficient by themselves. The process of using product metrics begins by deriving the software measures and metrics that are appropriate for the software representation under consideration. Then data are collected and metrics are computed. The metrics are computed and compared to pre-established guidelines and historical data. The results of these comparisons are used to guide modifications made to work products arising from analysis, design, coding, or testing.

Software engineers use product metrics to help them assess the quality of the design and construction of the software product being built. Product metrics provide software engineers with a basis to conduct analysis, design, coding, and testing more objectively. Qualitative criteria for assessing software quality are not always sufficient by themselves. The process of using product metrics begins by deriving the software measures and metrics that are appropriate for the software representation under consideration.

➤ System Engineering

Software engineering occurs as a consequence of a process called system engineering. Instead of concentrating solely on software, system engineering focuses on a variety of elements, analyzing, designing, and organizing those elements into a system that can be a product, a service, or a technology for the transformation of information or control.

The system engineering process takes on different forms depending on the application domain in which it is applied. Business process engineering is conducted when the context of the work focuses on a business enterprise. When a product is to be built, the process is called product Engineering.

Both business process engineering and product engineering attempt to bring order to the development of computer-based systems, work to allocate a role for computer software and, at the same time, to establish the links that tie software to other elements of a computer-based system.

➤ Computer-Based Systems

1.Set or arrangement of things so related as to form a unity or organic whole;2.a set of facts,principles,rules,etc.classified and arranged in an orderly form so as to show a logical plan linking the various parts3.a method or plan of classification or arrangement;4.an established way of doing something;method;procedure.....

We define a computer-based system as

A set or arrangement of elements that are organized to accomplish some predefined goal by processing information.

The goal may be to support some business function or to develop a product that can be sold to generate business revenue.To accomplish the goal,a computer-based system makes use of a variety of system elements:

Software.Computer programs, data structures, and related work products that serve to effect the logical method, procedure, or control that is required.

Hardware:Electronic devices that provide computing capability ,the interconnectivity devices(e.g. , network switches, telecommunications devices)that enable the flow of data, and electromechanical devices(e.g., sensors,motors,pumps)that provide external world function.

People: users and operators of hardware and software.

Database: A Large, organized collection information (e.g., models, specifications, hard-copy manuals, on-line help files, web sites) that portrays the use and/or operation of the system.

Procedures: The steps that define the specific use of each system element or the procedural context in which the system resides.

These elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and controls software to support the robot both require system engineering.

One complicated characteristic of computer-based systems is that the elements constituting one system may also represent one macro element of a still larger system. The macro element is a computer-based system that is one part of a larger computer-based system. As an example, we consider a factory automation system that is essentially a hierarchy of systems. At the lowest level of the hierarchy we have a numerical control machine, robots, and data entry devices.

➤ THE SYSTEM ENGINEERING HIERARCHY

System Engineering encompasses a collection of top-down and methods to navigate the hierarchy illustrated in figure 3.1. The system engineering process usually begins with a “world view”. That is, the entire business or product domain is examined to ensure that the proper business or technology context can be established. The world view is refined to focus more fully on a specific domain of interest. Within a specific domain, the need for targeted system elements is analysed. Finally, the analysis, design, and construction of a targeted system element are initiated. At the top of the hierarchy, a very broad context is established and, at the bottom, detailed activities, performed by the relevant engineering disciplines are conducted.

The world view (WV) is composed of a set of domains (D₁), which can each be a system or system of systems in its own right

$$WV = \{D_1, D_2, D_3, \dots, D_n\}$$

Each domain is composed of specific elements (E_j) each of which serves some role in accomplishing the objective and goals of the domain and component

$$D_i = \{E_1, E_2, E_3, \dots, E_m\}$$

Finally, each element is implemented by specifying the technical components (C_k) that achieve the necessary function for an element

$$E_j = \{C_1, C_2, C_3, \dots, C_k\}$$

In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

System Modeling System modeling is an important element of the system engineering process. Whether the focus is on the world view or the detailed view, the engineer creates models that:

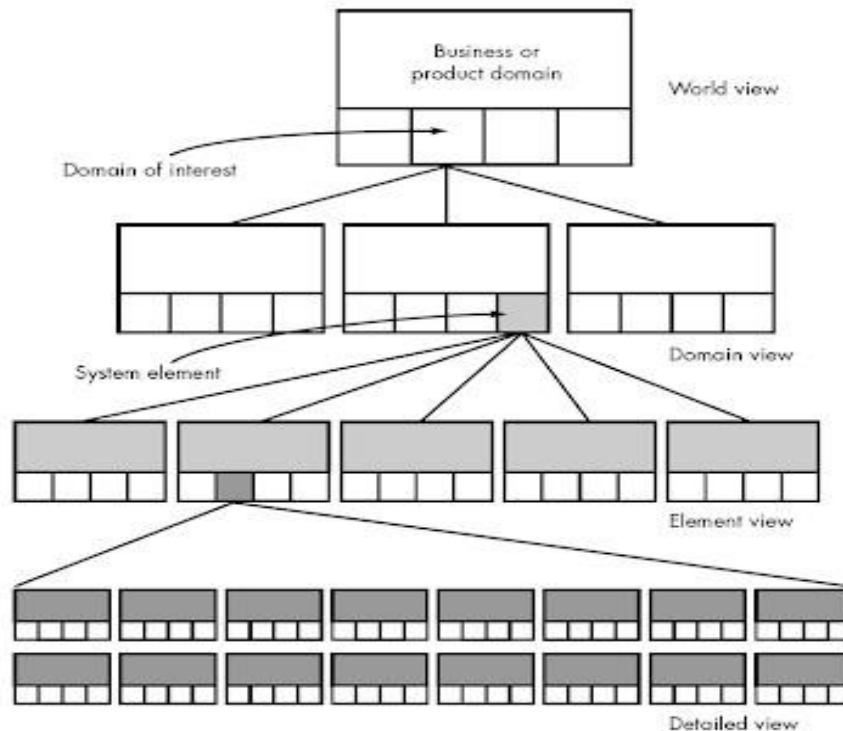
- Define the process that serves the needs of the view under consideration.
- Represent the behavior of the processes and the assumption on which the behavior is based. To construct a system model, the engineer should consider a number of restraining factors:
 1. Assumption that reduce the number of possible permutations and variations, thus enabling a model to reflect the problem in a reasonable manner
 2. Simplifications that the model has to create in a timely manner
 3. Limitations that help to bound the system
 4. Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented.

5. Preference that indicate the preferred architecture for all data, functions, and technology. **System Simulations**

Many computer-based systems interact with the real world in a reactive fashion. That is, real-world events are monitored by the hardware and software that form the computer-based system, and based on these events; the system imposes control on the machine, process, and even people who cause the events to occur. Real-time and embedded systems often fall into the reactive systems category

There are many systems in the reactive category- control machine and/or processes (e.g., commercial aircraft or petroleum refineries) that must operate with an extremely high degree of reliability. If the system fails, significant economic or human loss could occur. For this reason system modeling and simulation tools are used to help eliminate surprises when reactive computer-based systems are built. These tools are applied during the system engineering process, while the role of hardware and software, databases, and people is being specified. Modeling and simulation tools enable a system engineer to “test drive” a specification of the system **System Simulation Tools:**

Objective: System simulation tools provide the software engineer with the ability to predict the behavior



of a real-time system prior to the time that it is built. In addition, these tools enable the software engineer to develop mock-ups of the real-time system, allowing the customer to gain insight into the function, operation, and response prior to actual implementation

Mechanics: Tools in this category allow a team to define the elements of a computer-based system and then execute a variety of simulations to better understand the operating characteristics and overall performance of the system. Two broad categories of system simulation tools exist: (1) general purpose tools that can model virtually any computer-based system, and (2) special purpose tools that are designed to address a

specific application domain (e.g., aircraft avionics systems, manufacturing systems, electronic- systems)

Representative Tools: CSIM, developed by Lockheed Martin Advanced Technology Labs (www.atl.external.lmco.com), is a general purpose discrete- event simulation for block diagram- oriented systems.

➤ BUSINESS PROCESS ENGINEERING

The goal of business process engineering is to define architectures that will enable a business to use information effectively. When taking a world view of a company’s information technology needs, there is little doubt that system engineering is required. Not only is the specification of the appropriate computing architecture required. But the software architecture that populates the organization’s unique configuration of computing resources must be developed. Business process engineering is one approach for creating an overall plan for implementing the computing architecture.

The different architectures must be analyzed and designed within the context of business objectives and goals

- Data architecture

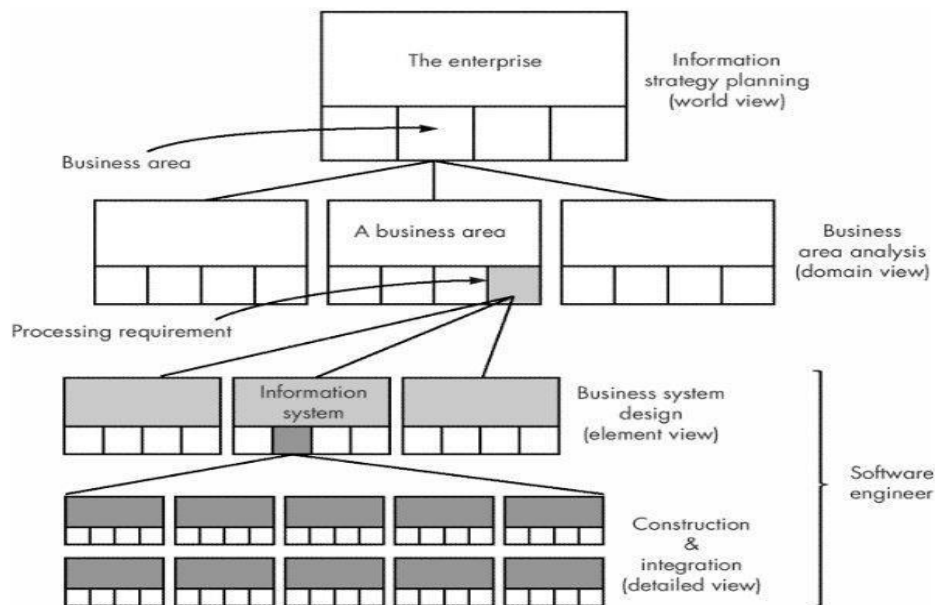
- Applications architecture
- Technology infrastructure

The data architecture provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used the business. A data object contains a set of attributes that define some aspect. Quality, characteristic, or descriptor of the data that are being described

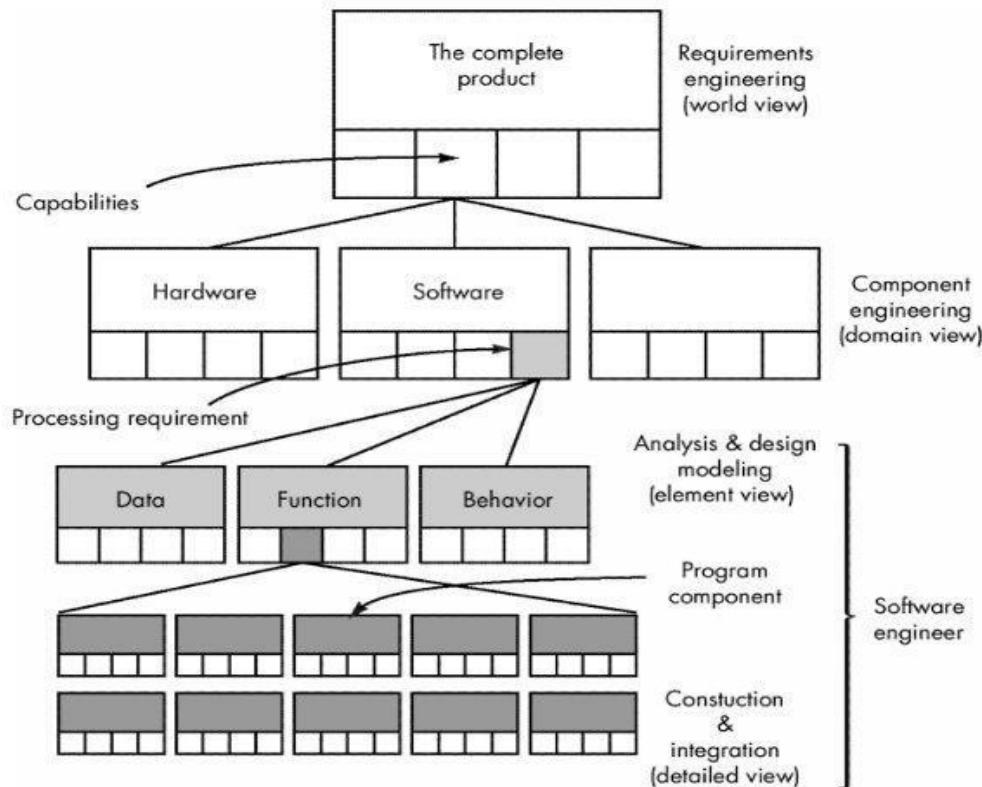
Once set of data objects is defined, their relationships are identified. A relationship indicates how objects are connected to one another. As an example, consider the objects: customer and product A. The two objects can be connected by the relationship purchases; that is, a customer purchases product A or product is purchased by customer. The data objects (there may be hundreds or even thousands for a major business activity) flow between business functions, are organized within a database, and are transformed to provide information that serves the needs of the business

The application architecture encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we consider the application architecture to be the system of programs (software) that performs this transformation. However, in a broader context, the application architecture might incorporate the role of people (who are information transformers and users) and business procedures that have not been automated

The technology infrastructure provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support the applications and data. This includes computers, operating systems, networks, telecommunication links, storage technologies, and the architecture (e.g., client/ server) that has been designed to important these technologies.



Product Engineering:



The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering—like business process engineering—must derive architecture and infrastructure. The architecture encompasses four distinct system components: software, hardware, data (and databases), and people. A support infrastructure is established and includes the technology required to tie the components together and the information (e.g., documents, CD-ROM, video) that is used to support the components.

The world view is achieved through requirements engineering. The overall requirements of the product are elicited from the customer. These requirements encompass information and control needs, product function and behavior, overall product performance, design and interfacing constraints, and other special needs. Once these requirements are known, the job of requirements engineering is to allocate function and behavior to each of the four components noted earlier.

Once allocation has occurred, system component engineering commences. System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering. Each of these engineering disciplines takes a domain-specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of the role of requirements engineering is to establish the interfacing mechanisms that will enable this to happen.

The element view for product engineering is the engineering discipline itself applied to the allocated component. For software engineering, this means analysis and design modeling activities and construction and integration activities that encompass code generation, testing, and support steps. The analysis step models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

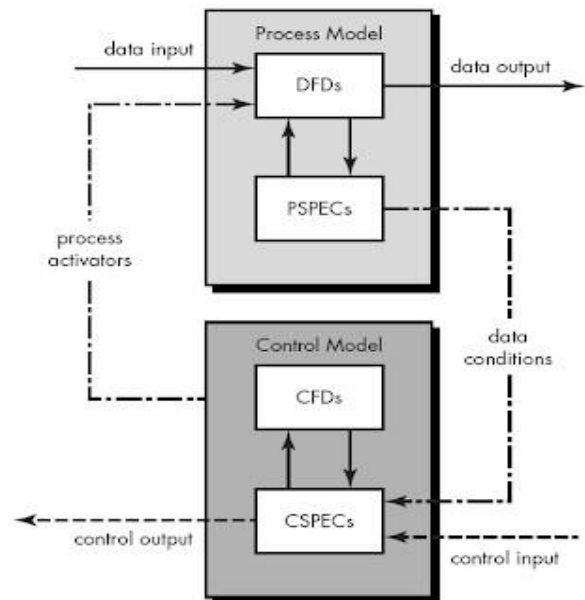
➤ System Modeling

System models are hierarchical or layered as a system is represented at different levels of abstraction. The top level of hierarchy presents the complete system. The data objects, functions, behaviors are represented. As the hierarchy is refined or layered, component level detail is modeled and finally system models evolve into engineering models.

Hatley-Pirbhai modeling is an extension of the concept that every computer system can be modeled through the usage of an input-processing-output model by including the two additional features of user interface process and maintenance/self testing. These five components are added to a system model template to allow for modeling of the system which allows for proper assignment to the processing regions.

The Hatley-Pirbhai Modeling

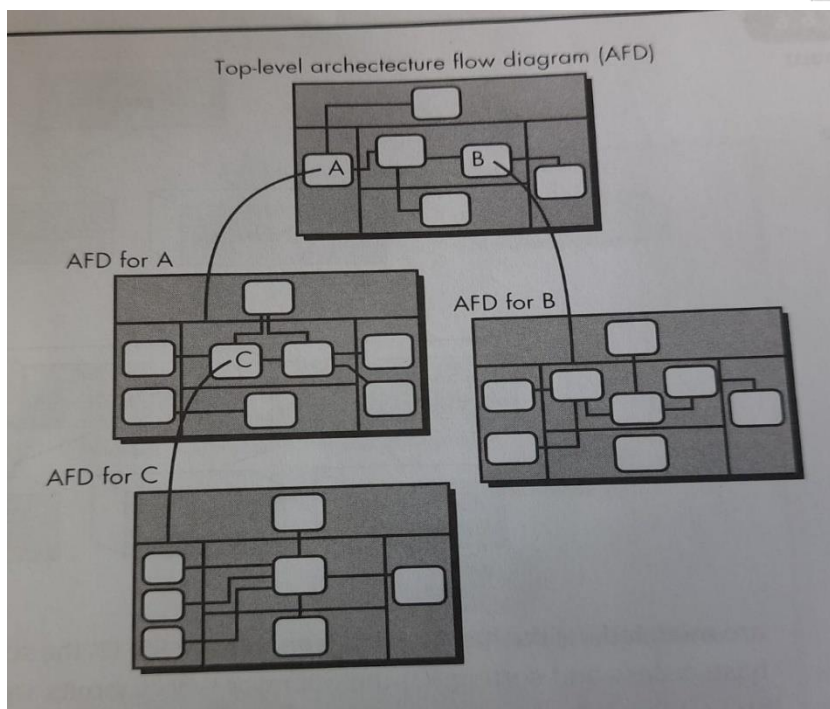
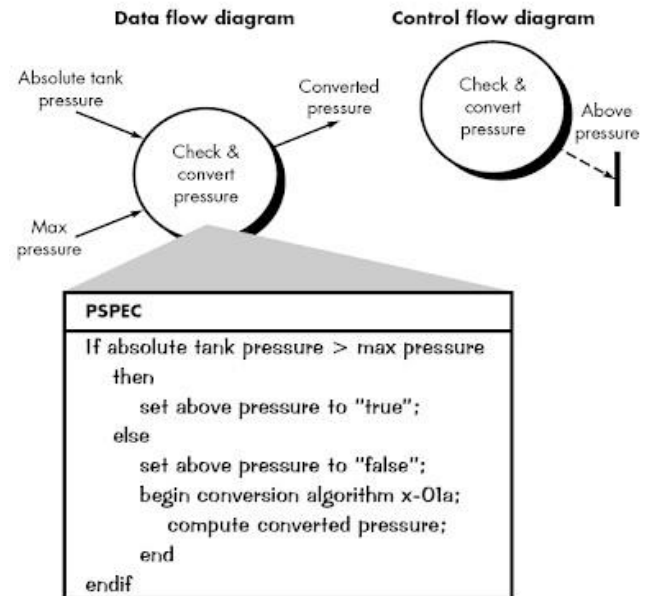
The Hatley and Pirbhai extensions to basic structured analysis notation focus less on the creation of additional graphical symbols and more on the representation and specification of the control-oriented aspects of the software. The dashed arrow is once again used to represent control or event flow. Unlike Ward and Mellor, Hatley and Pirbhai suggest that dashed and solid notation be represented separately. Therefore, a control flow diagram is defined. The CFD contains the same processes as the DFD, but shows control flow, rather than data flow. Instead of representing control processes directly within the flow model, a notational reference (a solid bar) to a control specification (CSPEC) is used. In essence, the solid bar can be viewed as a "window" into an "executive" (the CSPEC) that controls the processes (functions) represented in the DFD based on the event that is passed through the window. The CSPEC is used to indicate (1) how the software behaves when an event or control signal is sensed and (2) which processes are invoked as a consequence of the occurrence of the event. A process specification is used to describe the inner workings of a process represented in a flow diagram.



Using the notation described in earlier figures, along with additional information contained in PSPECs and CSPECs, Hatley and Pirbhai create a model of a real-time system. Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various processes to be activated. The interrelationship between the process and control models is shown schematically in the figure below. The process model is "connected" to the control model through data conditions. The control model is "connected" to the process model through process activation information contained in the CSPEC. A data condition occurs whenever data input to a process result in control output. This situation is illustrated in figure below, part of a flow model for an automated monitoring and control system for pressure vessels in an oil refinery. The process check and convert pressure implements the algorithm described in the PSPEC pseudocode shown. When the absolute tank pressure is greater than an allowable maximum, an above pressure event is generated. Note that when Hatley and Pirbhai notation is used, the data flow is shown as part of a DFD, while the control flow is noted separately as part of a control flow diagram. As we noted earlier, the vertical solid bar into which the above pressure event flows is a pointer to the CSPEC. Therefore, to determine what

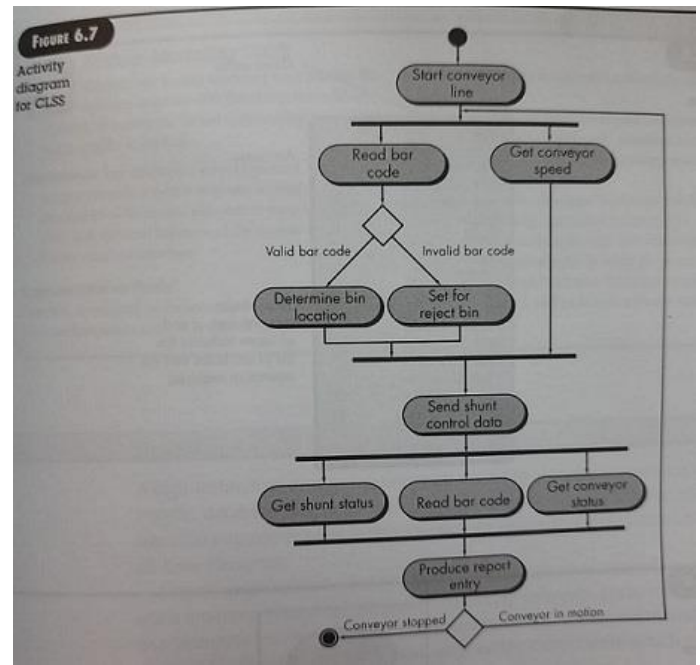
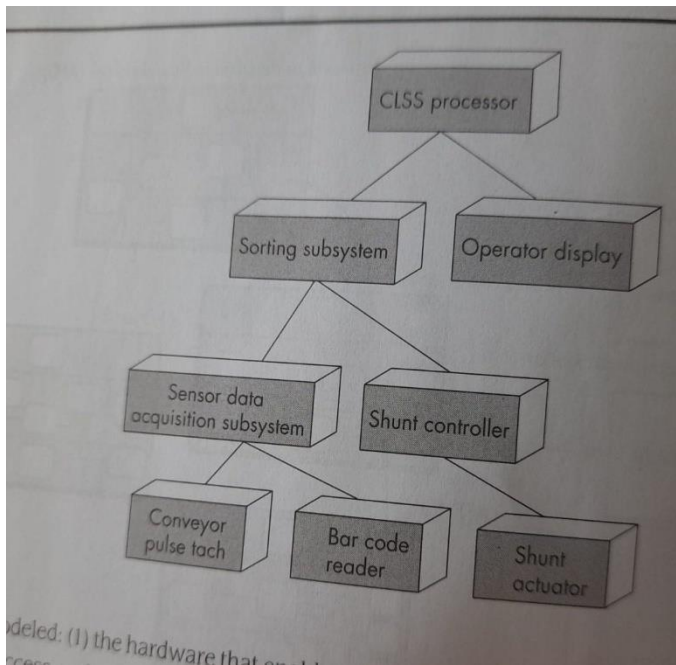
CSPEC.

The control specification (CSPEC) contains a number of important modeling tools. A process activation table is used to indicate which processes are activated by a given event. For example, a process activation table (PAT) for the above figure might indicate that the above pressure event would cause a process reduce tank pressure (not shown) to be invoked. In addition to the PAT, the CSPEC may contain a state transition diagram. The STD is a behavioral model that relies on the definition of a set of system states and is described in the following section.

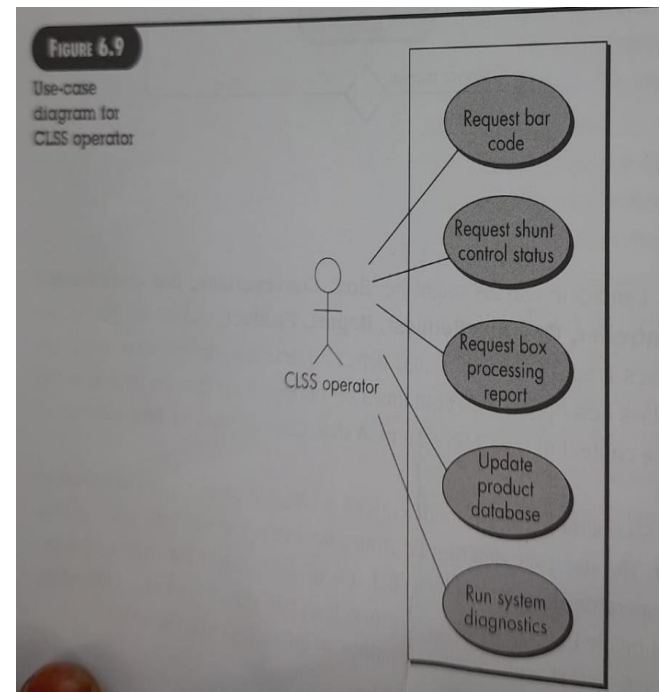
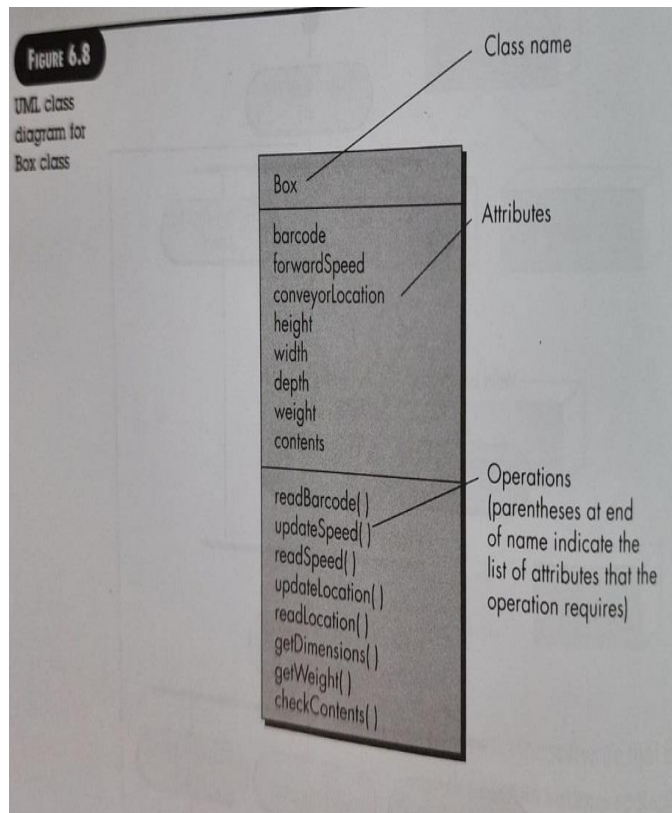


happens when this event occurs, we must check the

System Modeling with UML: UML provides a wide array of diagrams that can be used for analysis and design at both the system and the software level. for the CLSS system, four important system elements are modeled: 1) the hardware that enables CLSS; 2) the software that implements data-base access and sorting; 3) the operator who submits various requests to the system; and 4) the database that contains relevant bar code and destination information.



CLSS hardware can be modeled at the system level using a UML deployment diagram in the figure



- **Activity Diagram:** It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur.

- **Use Case Diagram:** It represents the functionality of a system by utilizing actors and use cases. It encapsulates the functional requirement of a system and its association with actors. It portrays the use case view of a system
- **Deployment Diagram:** It presents the system's software and its hardware by telling what the existing physical components are and what software components are running on them. It produces information about system software. It is incorporated whenever software is used, distributed, or deployed across multiple machines with dissimilar configurations.
- **Class Diagram:** Class diagrams are one of the most widely used diagrams. It is the backbone of all the object-oriented software systems. It depicts the static structure of the system. It displays the system's class, attributes, and methods. It is helpful in recognizing the relation between different objects as well as classes.

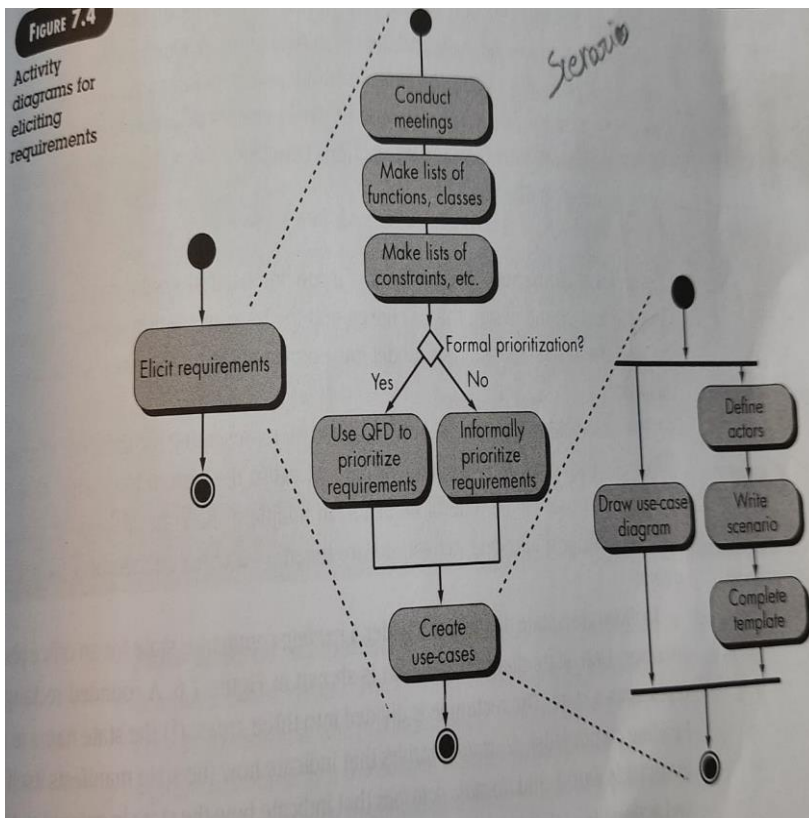
➤ Building the analysis model

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as software engineers learn more about the system to be built, and stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time.

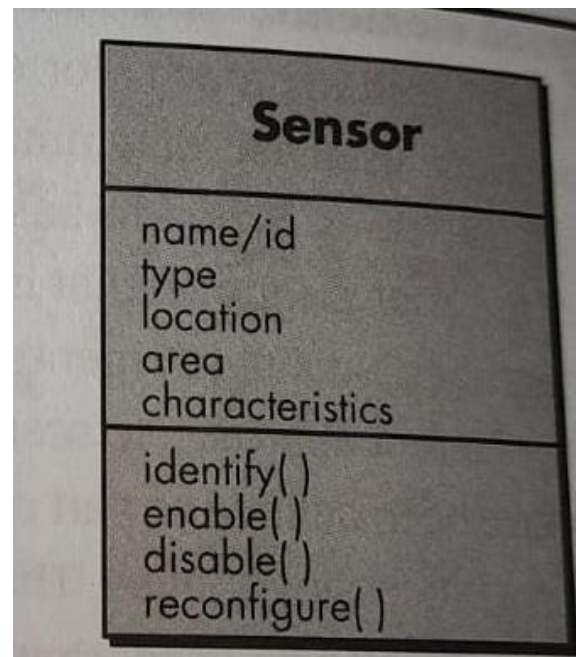
- Analysis model operates as a link between the 'system description' and the 'design model'.
- In the analysis model, information, functions and the behaviour of the system is defined and these are translated into the architecture, interface and component level design in the 'design modeling'. **Elements of the analysis model**

1. Scenario based element

- This type of element represents the system user point of view.
- Scenario based elements are use case



diagram, user stories.

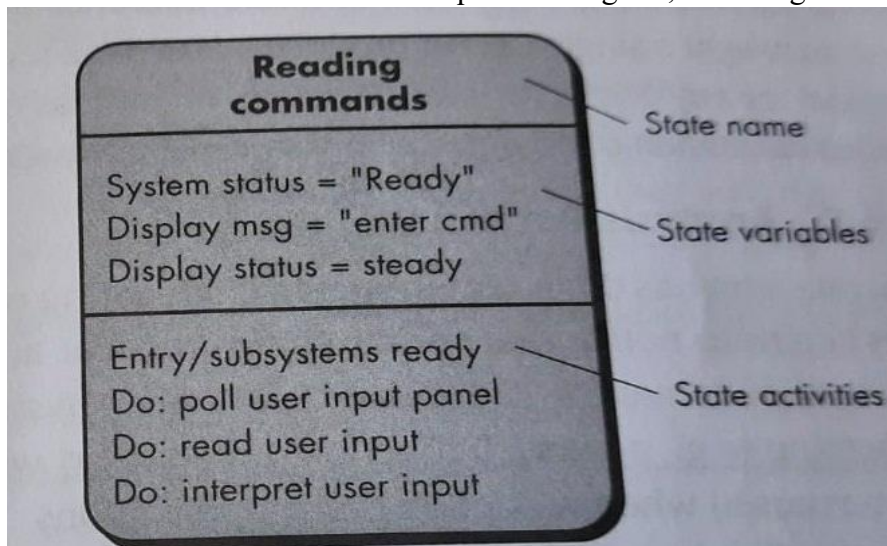


2. Class based elements

- The object of this type of element manipulated by the system.
- It defines the object, attributes and relationship.
- The collaboration is occurring between the classes.
- Class based elements are the class diagram, collaboration diagram.

3. Behavioral elements

- Behavioral elements represent state of the system and how it is changed by the external events.
- The behavioral elements are sequenced diagram, state diagram.



4. Flow oriented elements

- An information flows through a computer-based system it gets transformed.
- It shows how the data objects are transformed while they flow between the various system functions.
- The flow elements are data flow diagram, control flow diagram.

Analysis Rules of Thumb

The rules of thumb that must be followed while creating the analysis model **The rules are as follows:**

- The model focuses on the requirements in the business domain. The level of abstraction must be high i.e there is no need to give details.
- Every element in the model helps in understanding the software requirement and focus on the information, function and behaviour of the system.
- The consideration of infrastructure and nonfunctional model delayed in the design. **For example**, the database is required for a system, but the classes, functions and behavior of the database are not initially required. If these are initially considered then there is a delay in the designing.
- Throughout the system minimum coupling is required. The interconnection between the modules is known as 'coupling'.
- The analysis model gives value to all the people related to model.
- The model should be simple as possible. Because simple model always helps in easy understanding of the requirement.

Analysis Patterns It has been observed that the software engineer 'reuses' certain functions, classes, and/or behavior across all the projects, which may or may not form a part of the specific application domain.

For example, features and functions described by a user interface are almost common, regardless of the application domain chosen. Analysis patterns refer to classes, functions, and other features that can be reused when developing applications within a specific application domain. The objectives of analysis patterns are listed below.

- To quicken the requirements analysis phase by providing reusable analysis models with the description of both advantages and limitations.
- To suggest several design patterns and feasible solutions to common problems in order to help the software designer translate an analysis model into a design model.

Analysis patterns have a unique pattern name, which allows the development team to refer to them with their pattern names. These patterns are stored in a repository so that the software engineer can refer to these patterns and reuse them while developing new software.

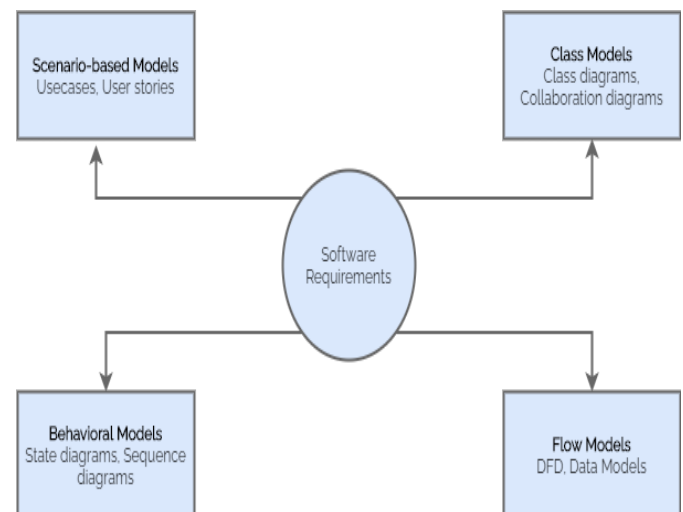
Information is stored in the analysis pattern, which can be viewed in the form of a template. The analysis pattern template, which is comprises the following sections.

- **Pattern name:** Consists of important information (in the form of descriptor) about the analysis pattern. This descriptor is used within the analysis model when reference is made to a pattern.
- **Intent:** Describes the problem that is addressed in an analysis pattern, which facilitates the software engineer to use analysis patterns in the specified application domain.
- **Motivation:** Represents how an analysis pattern can be used to address a particular problem in the application domain with the help of a scenario.
- **External issues and contexts:** Lists the external issues that affect the manner in which an analysis pattern works. The external issues include business related subjects, external technical constraints, and so on. In addition, external issues and contexts specify the external issues that are to be resolved by an analysis pattern.
- **Solution:** Describes the procedure by which an analysis pattern is implemented in the analysis model.
- **Consequences:** Describes the implementation effects of an analysis pattern into the application domain. It also provides a description of limitations that arises during its implementation.
- **Design:** Describes the procedure of achieving analysis patterns by using the known design patterns.
- **Known uses:** Provides information about the uses of an analysis pattern within the system.
- **Related patterns:** Lists the analysis patterns that are related to the 'chosen' pattern.

➤ Requirements Analysis:

Requirements analysis leads to the specification of software's operating characteristics, the identification of software's interface with other system elements, and the establishment of constraints that software must meet. Requirements analysis helps you elaborate on basic requirements generated through inception, elicitation, and negotiation requirements engineering tasks.

The action of requirements modeling produces one or more of the following types of models:



- Scenario-based models of requirements from the perspective of various system “actors” Scenario-based models of requirements from the perspective of various system “actors” .
- Data models that depict the problem’s information domain.
- Class-oriented models that represent object-oriented classes (attributes and operations) and how classes collaborate to achieve system requirements.
- Flow-oriented models that represent the system’s functional elements and how they transform data as it moves through the system.

Behavioral models that describe how software behaves in response to external “events” that describe how software behaves in response to external “events”

These models provide information to a software designer that can be turned into architectural, interface, and component-level designs.

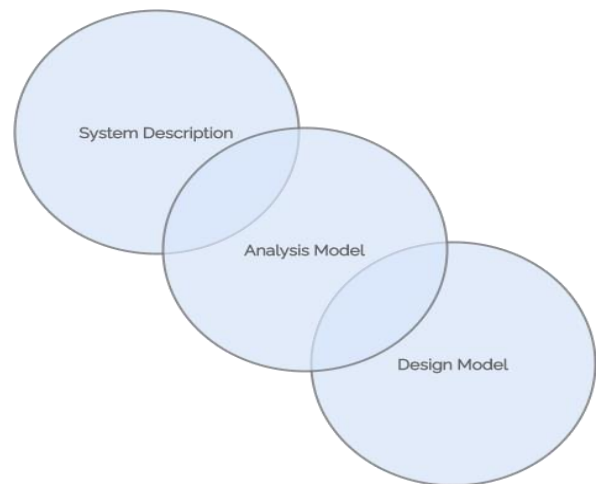
Overall Objectives and Philosophy

Throughout the requirements modeling process, your primary focus should be on what, not how.

The requirements model must accomplish three basic goals:

1. Describing what the client requires.
2. Establishing a foundation for the production of a software design.
3. Defining a set of criteria that can be validated once the software is constructed.

The analysis model bridges the gap between a system-level description of the overall system or business functioning as it is delivered through the use of software, hardware, data, human, and other system aspects and a software design. This relationship is shown in the figure below



Analysis Rules of Thumb

Arlow and Neustadt suggest a few useful rules of thumb to follow when developing the analytical model.

- **The model should concentrate on requirements that are obvious within the problem or business domain.** The level of abstraction should be moderate. “Don’t get bogged down in details” that attempt to explain how the system will function.
- Each element of the requirements **model should contribute to a broader understanding of software needs and provide insight into the system’s information domain, function, and behaviour.**
- **Put off thinking about infrastructure and other nonfunctional models until design.** That is, while a database may be required, the classes required to implement it, the functions required to access it, and the behaviour that will be displayed as it is used should be considered only after the problem domain analysis is complete.
- **Keep coupling to a minimum** across the system. Relationships between classes and functions must be represented. If the level of “interconnectedness” is excessively high, however, efforts should be made to reduce it.
- **Check to see if the requirements model gives value to all stakeholders.** Each constituency has a different application for the model. Business stakeholders, for example, should use the model to validate

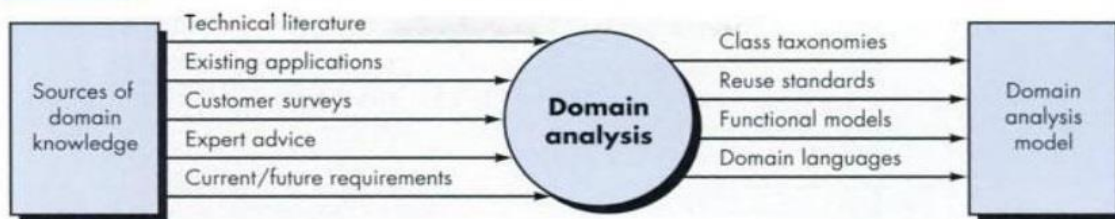
requirements; designers should use the model as a foundation for design; and QA personnel should use the model to help prepare acceptance tests.

- **Keep the model as simple as possible.** Don't make any new diagrams if they don't contribute any new information. When a basic list will enough, avoid using complex notational forms.

Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain...the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

FIGURE 8.2 Input and output for domain analysis



Modeling Approaches

Structured Analysis

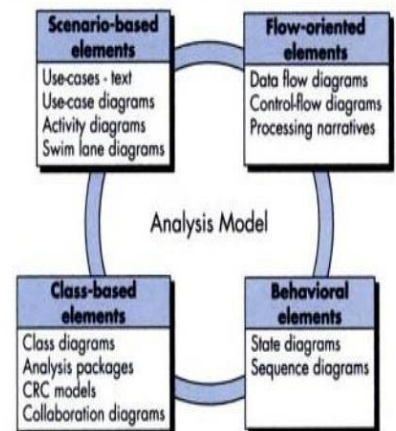
Structured analysis is one approach to requirements modeling that treats data and the processes that transform it as separate entities. Data objects are modeled in such a way that their attributes and relationships are defined. Processes that change data items are designed in such a way that it is clear how they transform data as it flows through the system.

Object-Oriented Analysis

The object-oriented analysis focuses on the definition of classes and how they interact with one another. The Unified Process and UML are both mostly object-oriented.

FIGURE 8.3

Elements of the analysis model



➤ Data Modeling Concepts

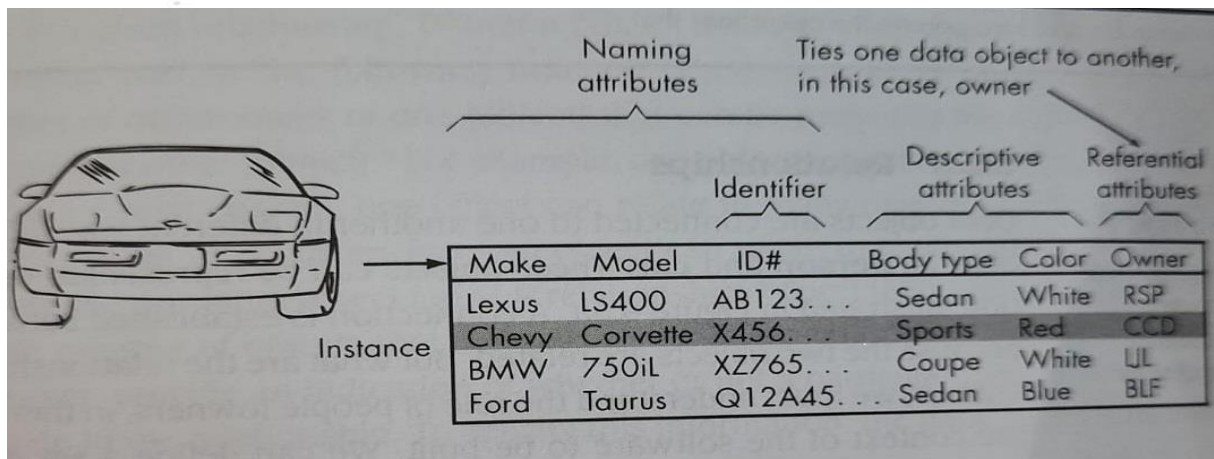
Analysis modeling often begins with *data modeling*. The software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.

8.3.1 Data Objects

A *data object* is a representation of almost any composite information that must be understood by software. By *composite information*, we mean something that has a number of different properties or attributes. Therefore, “width” (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.⁵ Therefore, the data object can be represented as a table as shown in Figure 8.4. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of **make**, **model**, **ID number**, **body type**, **color** and **owner**. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.



Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to 1) name an instance of the data object, 2) describe the instance, or 3) make reference to another instance in another table.

One or more of the attributes must be defined as an identifier—that is the identifier attribute becomes a “key” when we want to find an instance of the data object. In some cases; values for the identifier(s) are unique, although this is not a requirement.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a

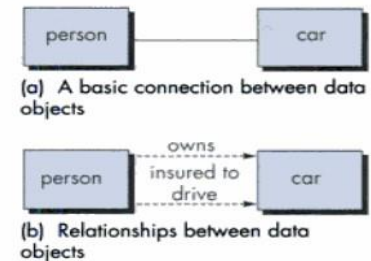
Department of Motor vehicles, but these attributes would be used by a Department of Motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software.

Relationships

Data objects are connected to one another in different ways. Two data objects, person and car. These objects can be represented using the simple notation in figure. A connection is established between person and car because the two objects are related. We can define a set of object/relationship pairs that define the relevant relationships. For example, □ A person owns a car.

□ A person is insured to drive a car.

The relationship owns and insured to drive define the relevant connections between person and car. The above fig illustrates these object/relationship pairs graphically. The arrows noted in diagram these provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.



Cardinality and Modality

The elements of data modeling—data objects, attributes, and relationships—provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states that **objectX** relates to **objectY** does not provide enough information for software engineering purposes. We must understand how many occurrences of **objectX** are related to how many occurrences of **objectY**. This leads to a data modeling concept called *cardinality*.

The data model must be capable of representing the number of occurrences of objects in a given relationship. Tillmann [TIL93] defines the cardinality of an object/relationship pair in the following manner: "Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]." For example, one object can relate to only one other object (a 1:1 relationship); one object can relate to many objects (a 1:N relationship); some number of occurrences of an object can relate to some other number of occurrences of another object (an M:N relationship).⁶ Cardinality also defines "the maximum number of objects that can participate in a relationship" [TIL93]. It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

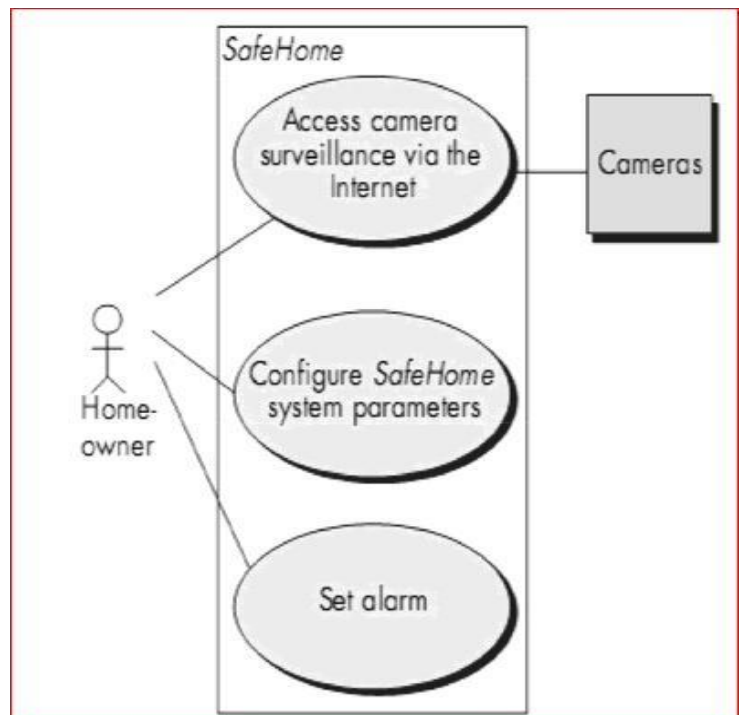
➤ Behavioral model

The system's behaviour can be represented as a function of certain events and time. The behavioural model describes how the software will respond to outside events or stimuli. Go through the following steps, to make the model.

1. Evaluate all use cases to properly understand the system's interaction sequence.
2. Recognize the events that drive the interaction sequence and understand how these events are related to specific objects.
3. For each use case, create a sequence.
4. Create a system state diagram.
5. Go over the behavioural model again to ensure its accuracy and consistency.

Identifying Events with the Use Case

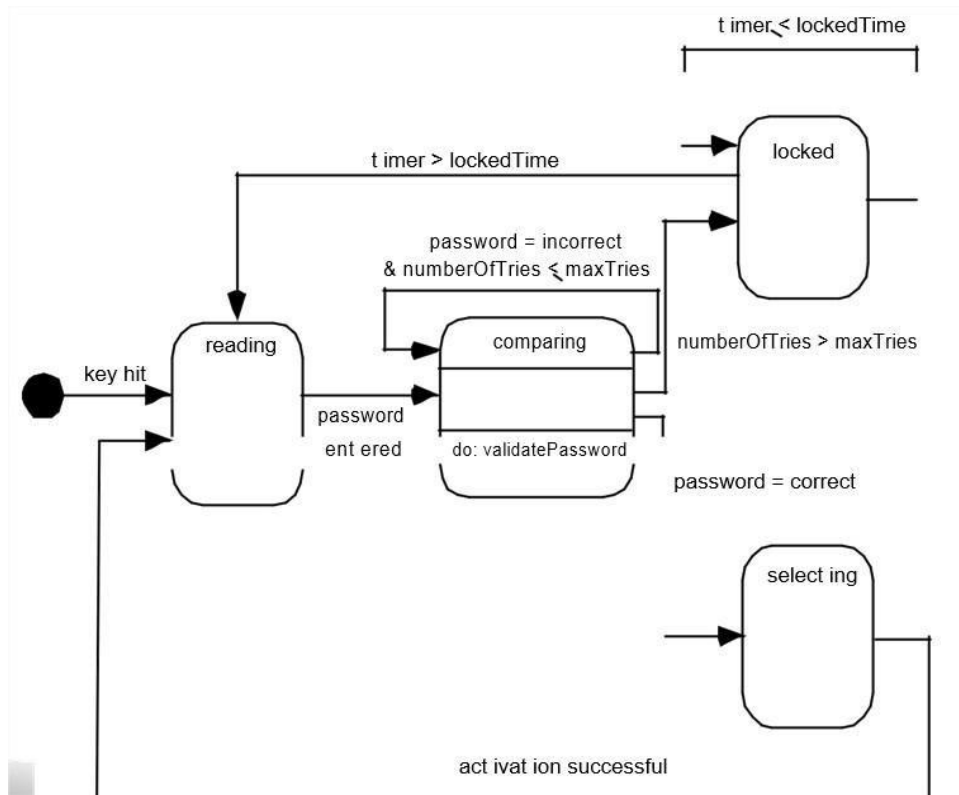
- The use case represents a sequence of activities that involves actors and the system.
- In general, an event occurs whenever the system and an actor exchange information.
- It has been indicated that an event is not the information that has been exchanged, but rather the fact that information has been exchanged.
- For example: The use case for a portion of the Safe Home security function.
- The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further act.
- The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted,
- Here the object, Homeowner, transmits an event to the object Control Panel. The event might be called password entered.



State Representations

- In the context of behavioral modeling, two different characterizations of states must be considered:
- The state of each class as the system performs its function and
- The state of the system as observed from the outside as the system performs its function.
- The state of a class takes on both passive and active characteristics.

- A passive state is simply the current status of all of an object's attributes.
- The active state of an object indicates the current status of the object as it undergoes a continuing processing.
- For example, the passive state of the class Player (in the video game application) would include the current position and orientation attributes of Player as well as other features of Player that are relevant to the game .
- The class Player might have the following active states:
- Moving, at rest, injured, being cured;, lost etc.
- An event (sometimes called a trigger) must occur to force an object to make a transition from one active state to another.



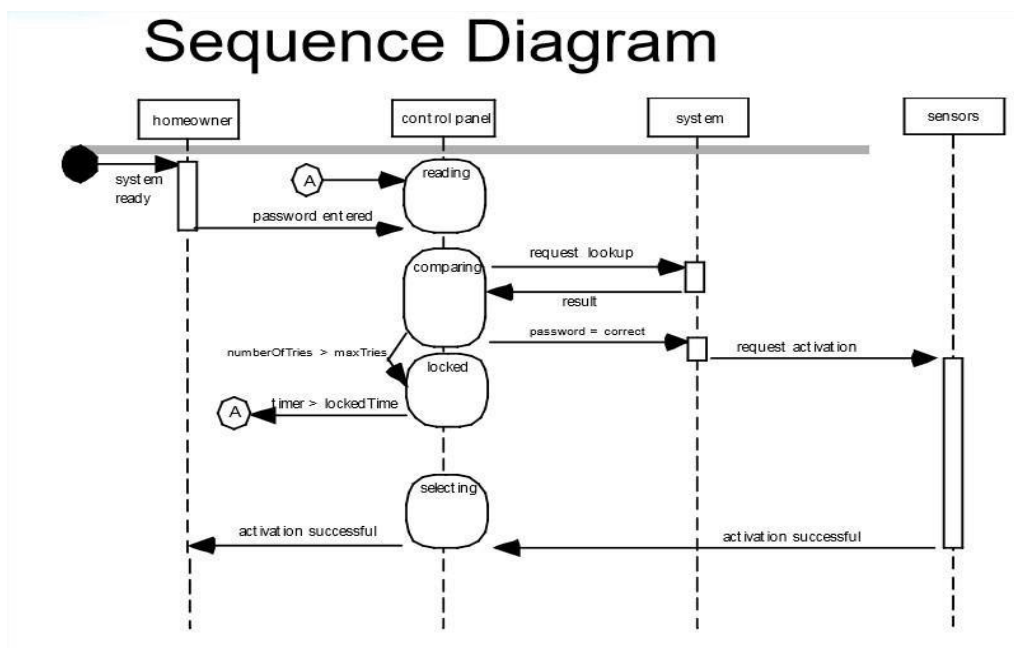
State diagram

- One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states.
- Figure (In Previous Slide) illustrates a state diagram for the Control Panel object in the Safe Home security function.
- Each arrow shown in Figure represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition.
- An action occurs concurrently with the state transition, generally involves one or more operations (responsibilities) of the object.
- For example, the action connected to the password entered event (ing Figure) is an operation named validate Password () that accesses a password object and performs a digit-by-digit comparison to validate the entered password. **The States of a System**

- state—a set of observable circumstances that characterizes the behavior of a system at a given time.
- state transition—the movement from one state to another.
- event—an occurrence that causes the system to exhibit some predictable form of behavior.
- action—process that occurs as a consequence of making a transition.
- state—a set of observable circumstances that characterizes the behavior of a system at a given time.
- state transition—the movement from one state to another.

Sequence diagram

- The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object.
- Once events have been identified by examining a use case, the modeler creates a sequence diagram.
- It is a representation of how events cause flow from one object to another as a function of time.
- The sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.
- For example: Figure (In Next Slide) illustrates a partial sequence diagram for the Safe Home security function.
- Each of the arrows represents an event (derived from a use case)
- It indicates how the event channels behavior between Safe Home objects.
- Time is measured vertically (downward),
- Narrow vertical rectangles represent time spent in processing an activity.



Sequence diagram

- Explanation of Figure: The first event, system ready, is derived from the external environment and channels behavior to the Homeowner object.
- The homeowner enters a password. A request lookup event is passed to System, which looks up the password in a simple database and returns a result (found or not found) to ControlPanel (now in the comparing state).
- A valid password results in a password=correct event to System, which activates Sensors with a request activation event. Ultimately, control is passed back to the homeowner with the activation successful event.

➤ Design Engineering

Encompasses the set of principles, concepts and practices that lead to the development of high quality system or product. Design creates a representation or model of the software. Design model provides details about S/W architecture, interfaces and components that are necessary to implement the system. Quality is established during Design. Design should exhibit firmness, commodity and design. Design sits at the kernel of S/W Engineering. Design sets the stage for construction.

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed.

What is the work product?

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

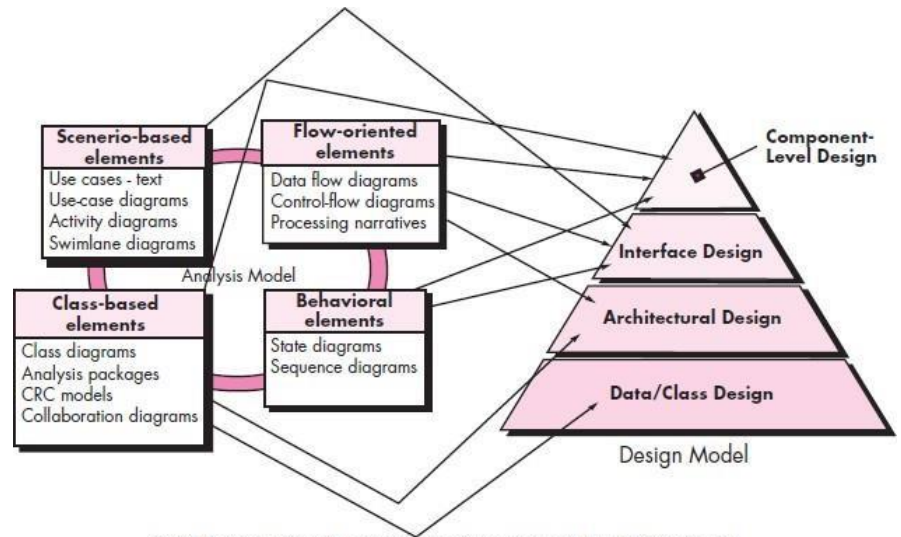


Fig 8.1: Translating the requirements model into the design model

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 8.1. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task.

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information

(e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

➤ THE DESIGN PROCESS AND QUALITY

Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Software Quality Guidelines and Attributes: Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. The three characteristics that serve as a guide for the evaluation of a good design:

The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders. The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

Quality Guidelines. In order to evaluate the quality of a design representation, the software team must establish technical criteria for good design. Guidelines are as follows. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

A design should contain distinct representations of data, architecture, interfaces, and components.

A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

A design should lead to components that exhibit independent functional characteristics.

A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

A design should be represented using a notation that effectively communicates its meaning.

FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation.

Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

Performance is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

Supportability combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design: The evolution of software design is a continuing process. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural

aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

➤ DESIGN CONCEPTS

Design concepts have evolved over the history of software engineering. Each concept provides the software designer with a foundation from which more sophisticated design methods can be applied. A brief overview of important software design concepts that span both traditional and object-oriented software development is given below.

Abstraction: When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. A data abstraction is a named collection of data that describes a data object.

Architecture: Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”.

In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

One goal of software design is to derive an architectural rendering of a system. A set of architectural patterns enables a software engineer to solve common design problems.

Shaw and Garlan describe a set of properties as part of an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns: A pattern is a named suggest of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns.

Stated

A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine whether the pattern is applicable to the current work whether the pattern can be reused (hence, saving design time) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns: Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than

the effort required to solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem. It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

Modularity: Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements. It has been stated that “modularity is the single attribute of software that allows a program to be intellectually manageable”. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

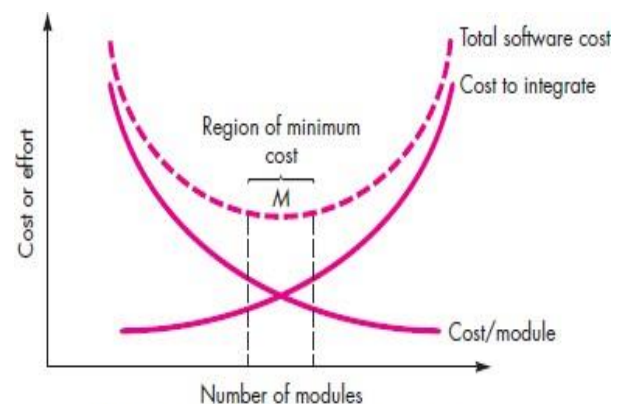


Fig 8.2: Modularity and software cost

if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.

Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

Information Hiding: The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.” In other words,

modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence: The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with “single minded” function and an “aversion” to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling.

Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-

mindedness), it is often necessary and advisable to have a software component perform multiple functions.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect”, caused when errors occur at one location and propagates throughout a system.

Refinement: Stepwise refinement is a top-down design strategy. . A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Aspects: As requirements analysis occurs, a set of “concerns” is uncovered. These concerns “include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts”. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B “if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account”.

Refactoring: An important design activity for many agile methods is refactoring a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or

inappropriate data structures, or any other design failure that can be corrected to yield a better design. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Object-Oriented Design Concepts: The object-oriented (OO) paradigm is widely used in modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

Design Classes: As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be

implemented, and implement a software infrastructure that supports the business solution.

Five different types of design classes, each representing a different layer of the design architecture, can be developed.

User interface classes define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.

Business domain classes are often refinements of the analysis classes. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

Process classes implement lower-level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.

System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class.

Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design

model is highly coupled, the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the Law of Demeter, suggests that a method should only send messages to methods in neighboring classes.

➤ THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 8.4. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

Referring to Figure 8.4, the dashed line indicates the boundary between the analysis and design models. The analysis model slowly blends into the design and a clear distinction is less obvious.

The elements of the design model use UML diagrams, that were used in the analysis model.

The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

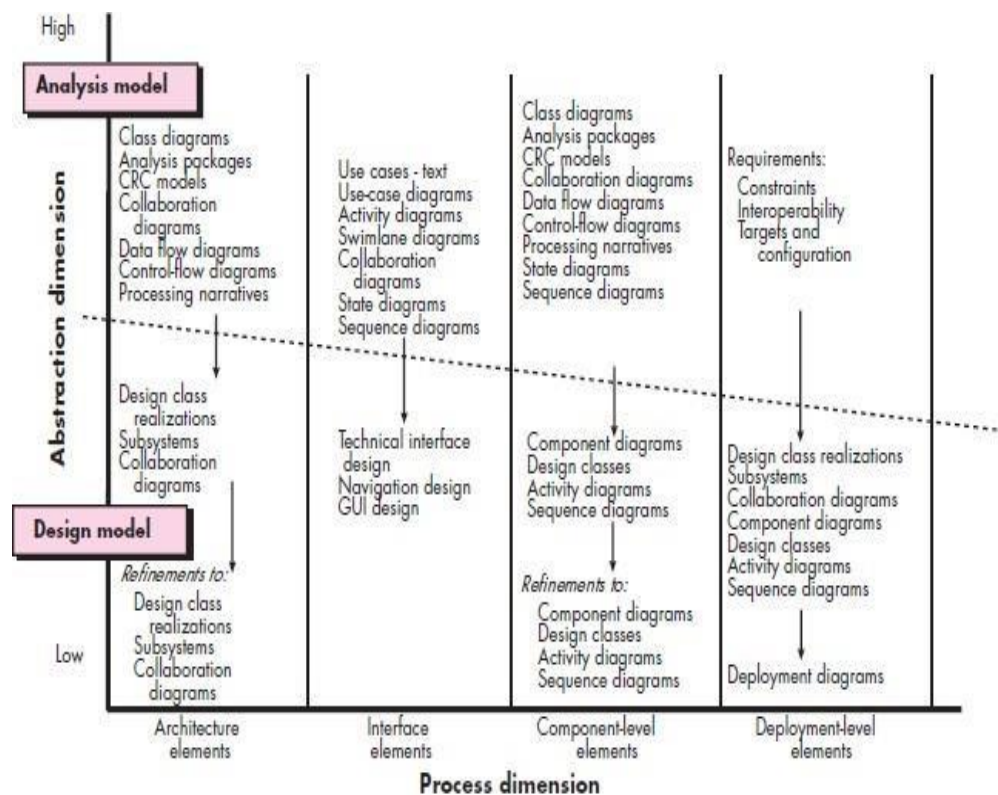


Fig 8.4: Dimensions of the design model

Data Design Elements: Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements: The architectural design for software is the equivalent to the floor plan of a house. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software. The architectural model is derived from three sources:

Information about the application domain for the software to be built; specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and the availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, Each subsystem may have its own architecture

Interface Design Elements: The interface design for software is analogous to a set of detailed drawings for the doors, windows, and external utilities of a house. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design:
the user interface (UI);

external interfaces to other systems, devices, networks, or other producers or consumers of information; and internal interfaces between various design components.

These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called usability design) is a major software engineering action. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components).

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of internal interfaces is closely aligned with component-level design.

Component-Level Design Elements: The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component



Fig 8.6: A UML component diagram

operations (behaviors). Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure

8.6.

A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are

usually modeled using pseudocode or the programming language to be used for implementation.

Deployment-Level Design Elements:

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

During design, a UML deployment diagram is developed and then refined as shown in Figures 8.7.

The diagram shown is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details.

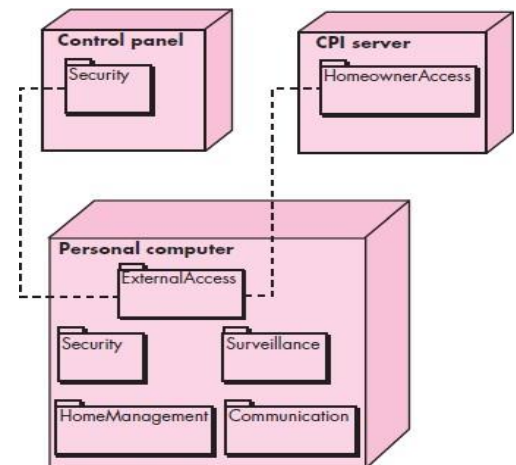


Fig 8.7: A UML deployment diagram

SOFTWARE ARCHITECTURE

What Is Architecture? :Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another. The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables you to analyze the effectiveness of the design in meeting its stated requirements architectural alternatives at a stage when making design changes is still relatively easy reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are

not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

There is a distinct difference between the terms architecture and design. A design is an instance of an architecture similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?: Three key reasons that software architecture is important:

Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system. The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

Architectural Descriptions: Different stakeholders will see architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Architectural Decisions: Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture .

ARCHITECTURAL GENRES

Although the underlying principles of architectural design apply to all types of architecture, the architectural genre will often dictate the specific architectural approach

to the structure that must be built. In the context of architectural design, genre implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.

Grady Booch suggests the following architectural genres for software-based systems:

Artificial intelligence—Systems that simulate or augment human cognition, locomotion, or other organic processes.

Commercial and nonprofit—Systems that are fundamental to the operation of a business enterprise.

Communications—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.

Content authoring—Systems that are used to create or manipulate textual or multimedia artifacts.

Devices—Systems that interact with the physical world to provide some point service for an individual.

Entertainment and sports—Systems that manage public events or that provide a large group entertainment experience.

Financial—Systems that provide the infrastructure for transferring and managing money and other securities.

- I. Games—Systems that provide an entertainment experience for individuals or groups.
- II. Government—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- III. Industrial—Systems that simulate or control physical processes.
- IV. Legal—Systems that support the legal industry.
- V. Medical—Systems that diagnose or heal or that contribute to medical research.

VI. Military—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons.

Operating systems—Systems that sit just above hardware to provide basic software services.

Platforms—Systems that sit just above operating systems to provide advanced services.

Scientific—Systems that are used for scientific research and applications.

Tools—Systems that are used to develop other systems.

Transportation—Systems that control water, ground, air, or space vehicles.

Utilities—Systems that interact with other software to provide some point service.

SAI (Software Architecture for Immersipresence) is a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. The goal of SAI is to provide a universal framework for the distributed implementation of algorithms and their easy integration into complex systems. ➤ **ARCHITECTURAL STYLES**

Architectural style describes a system category that encompasses a set of components (e.g., a database, computational modules) that perform a function required by a system; a set of connectors that enable “communication, coordination and cooperation” among components; constraints that define how components can be integrated to form the system; and semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

An architectural pattern, like an architectural style, imposes a transformation on the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;

a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

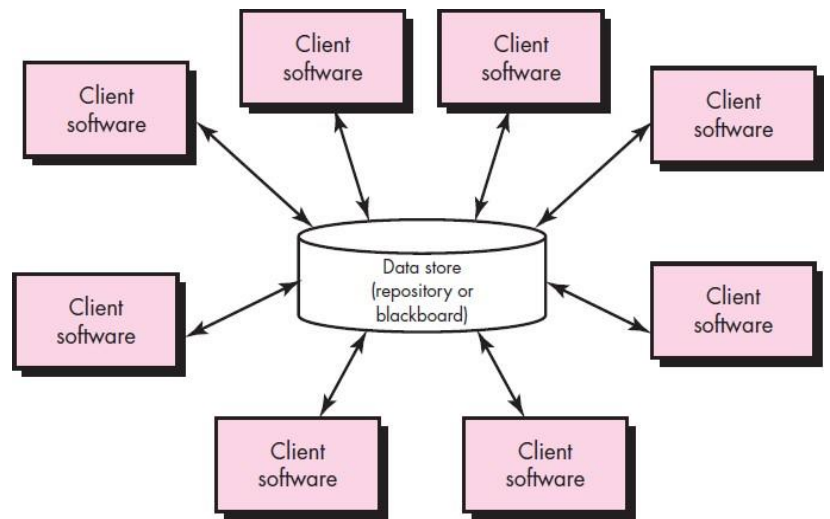


Fig 9.1: Data-centered architecture

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a

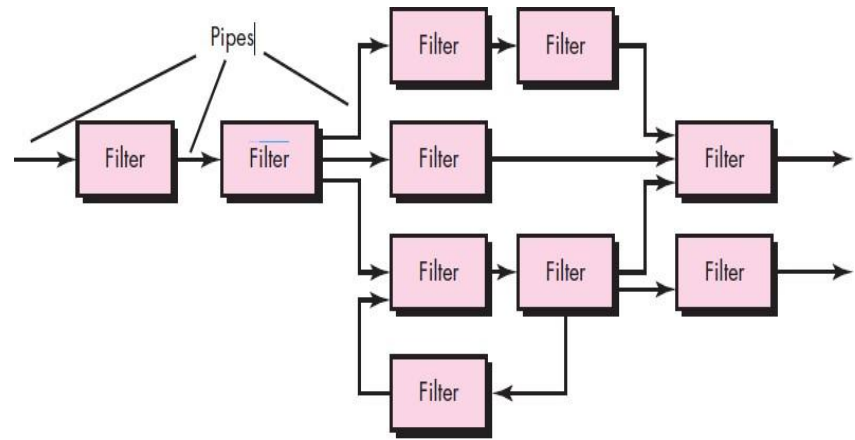
certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:

Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of

program components that in turn may invoke still other components. Figure 9.3 illustrates architecture of this type.

Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.



Pipes and filters
Fig 9.2: Data-flow architecture

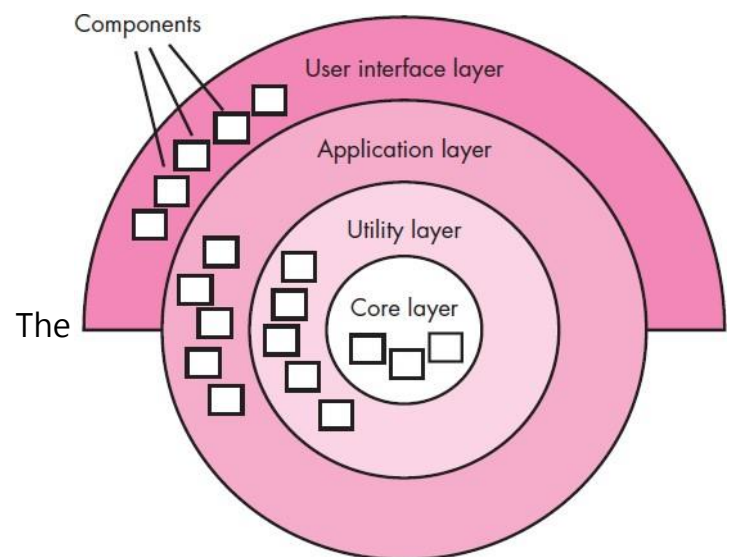


Fig 9.4: Layered architecture

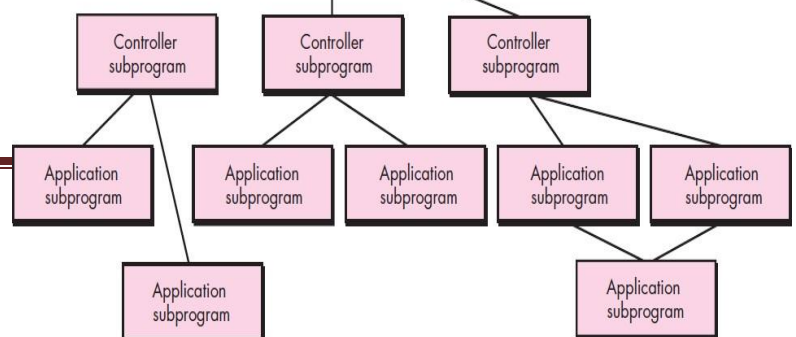


Fig 9.3: Main program / subprogram architecture

Object-oriented architectures. components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. For example, a layered style (appropriate for most systems) can be combined with a data- centered architecture in many database applications.

Architectural Patterns: Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design. For example, the overall architectural style for an application might be call-and-return or object- oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns.

Organization and Refinement: Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously?

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be

shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

➤ ARCHITECTURAL DESIGN

The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An archetype is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context: Architectural context represents how the software interacts with entities external to its boundaries.

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 9.5. Referring to

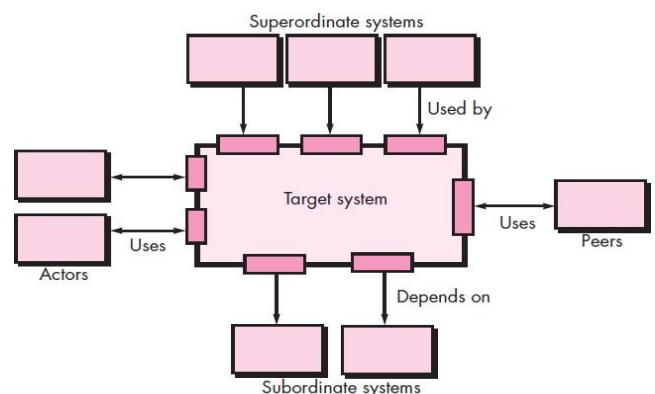


Fig 9.5: Architectural context diagram

the figure, systems that interoperate with the target system are represented as Super ordinate systems—those systems that use the target system as part of some higher-level processing scheme.

Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Peer-level systems—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system).

Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

Defining Archetypes: Archetypes are the abstract building blocks of an architectural design. An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

The following are the archetypes for safe Home: Node, Detector, Indicator., and Controller.

Refining the Architecture into Components: As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory

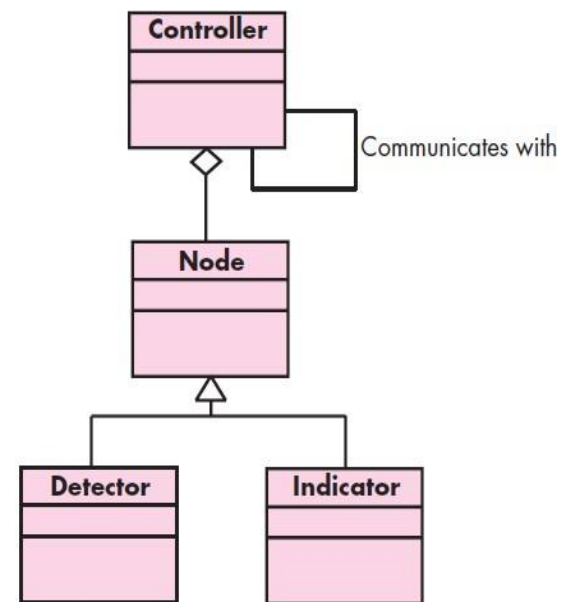


Fig 9.7: UML relationships for safehome function Archetypes

management components, communication components, database components, and task management components are often integrated into the software architecture.

As an example for SafeHome home security, the set of top-level components that address the following functionality:

External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

Control panel processing—manages all control panel functionality.

Detector management—coordinates access to all detectors attached to the system.

Alarm processing—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall Safe Home architecture.

Describing Instantiations of the System: The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement is still necessary.

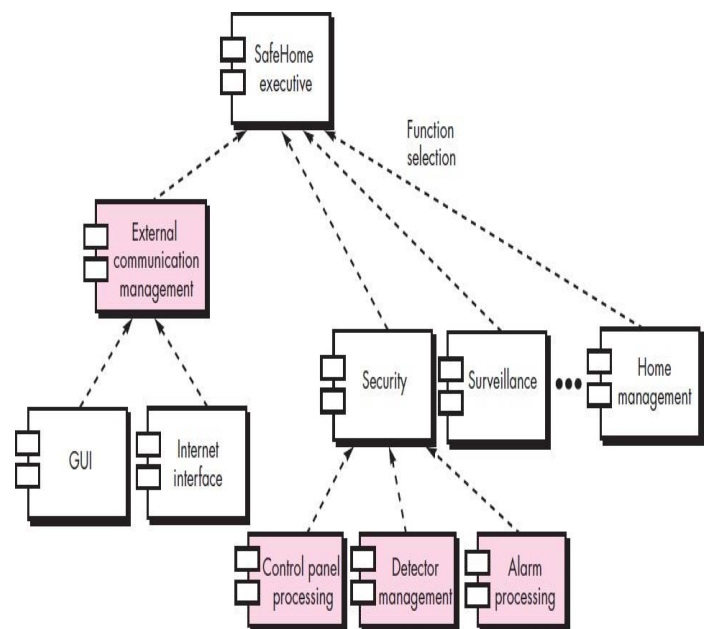


Fig 9.8: Overall architectural structure for SafeHome with top-level components

➤ ARCHITECTURAL MAPPING USING DATA FLOW

To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems. The call and return architecture can reside within other more sophisticated architectures. For example, the

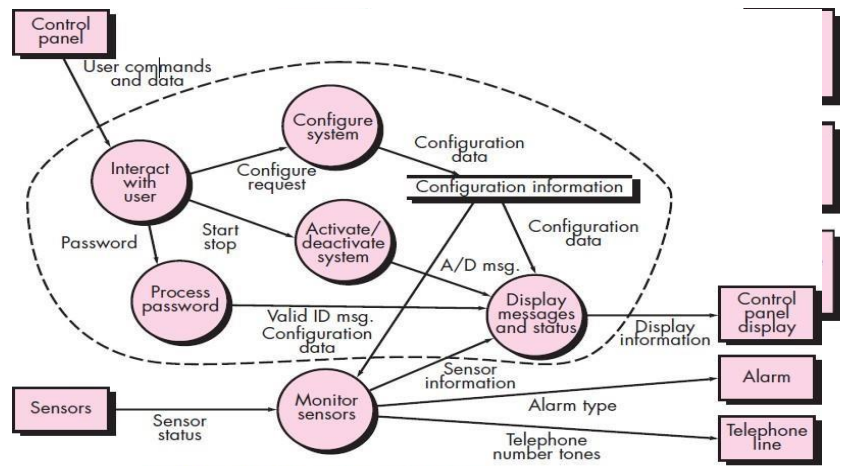


Fig 9.11: Level 1 DFD for safeHome security function

architecture of one or more components of a client-server architecture might be call and return. A mapping technique, called structured design, is often characterized as a data flow oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six step process: (1) the type of information flow is established, (2) flow boundaries are indicated, (3) the DFD is mapped into the program structure, (4) control hierarchy is defined, (5) the resultant structure is refined using design measures and heuristics, and (6) the architectural description is refined and elaborated.

Transform Mapping: Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

Step 1. Review the fundamental system model. The fundamental

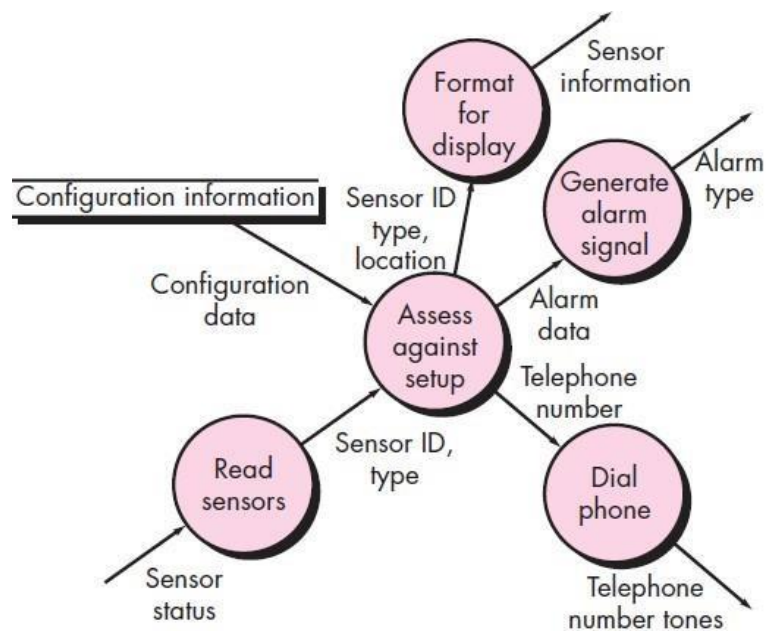


Fig 9.12: Level 2 DFD that refines the monitor sensors transform

system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors (Figure 9.12) is examined, and a level 3 data flow diagram is derived as shown in Figure 9.13. The data flow diagram exhibits relatively high cohesion. Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD (Figure 9.13), we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles)

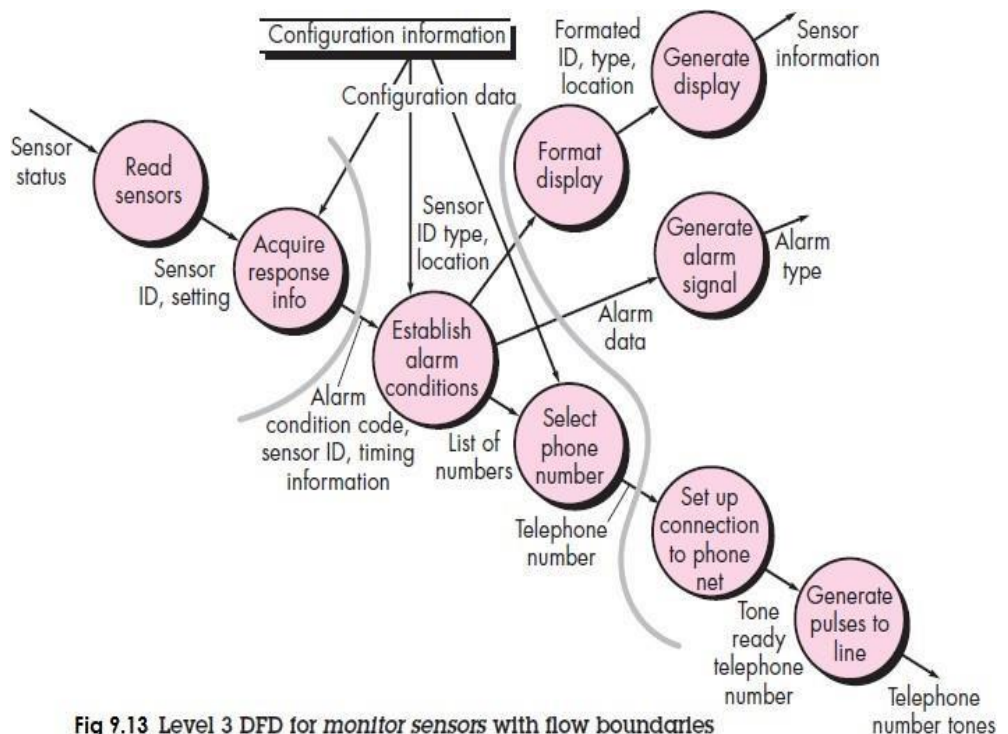


Fig 9.13 Level 3 DFD for monitor sensors with flow boundaries

that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary. The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

Step 5. Perform “first-level factoring.” The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and low level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

Step 6. Perform “second-level factoring.” Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refining the Architectural Design: Refinement of software architecture during early stages of design is to be encouraged. Alternative architectural styles may be derived, refined, and evaluated for the “best” approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

➤ COMPONENT-BASED SOFTWARE ENGINEERING

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software

“components.”

Domain Engineering: The intent of domain engineering is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

The overall approach to domain analysis is often characterized within the context of object-oriented software engineering. The steps in the process are defined as:

Define the domain to be investigated.

Categorize the items extracted from the domain.

Collect a representative sample of applications in the domain.

Analyze each application in the sample and define analysis classes.

Develop a requirements model for the classes.

It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development.

Component Qualification, Adaptation, and Composition: Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in-house, others can be extracted from existing applications, and still others may be acquired from third parties. Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

Component Qualification. Component qualification ensures that a candidate component will perform the function required, will properly “fit” into the architectural style specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.

Application programming interface (API).

Development and integration tools required by the component.

Run-time requirements, including resource usage (e.g., memory or storage), timing or speed, and network protocol. Service requirements, including operating system interfaces and support from other components.

Security features, including access controls and authentication protocol.

Embedded design assumptions, including the use of specific numerical or non numerical algorithms.

Exception handling

Component Adaptation. In an ideal setting, domain engineering creates a library of components that can be easily integrated into application architecture. The implication of “easy integration” is that (1) consistent methods of resource management have been implemented for all components in the library, (2) common activities such as data management exist for all components, and (3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Conflicts may occur in one or more of the areas in selection of components. To avoid these conflicts, an adaptation technique called component wrapping is sometimes used. When a software team has full access to the internal design and code for a component white-box wrapping is applied. Like its counterpart in software testing white-box wrapping examines the internal processing details of the component and makes codelevel modifications to remove any conflict. Gray-box wrapping is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. Black-box wrapping requires the introduction of pre- and post processing at the component interface to remove or mask conflicts.

Component Composition. The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure and coordination must be established to bind the components into an operational system.

OMG/CORBA. The Object Management Group has published a common object request broker architecture (OMG/CORBA). An object request broker (ORB) provides a variety of

services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.

Microsoft COM and .NET. Microsoft has developed a component object model (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. From the point of view of the application, "the focus is not on how implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects". The Microsoft .NET framework encompasses COM and provides a reusable class library that covers a wide array of application domains.

Sun JavaBeans Components. The JavaBeans component system is a portable, platform-independent CBSE infrastructure developed using the Java programming language. The JavaBeans component system encompasses a set of tools, called the Bean Development Kit (BDK), that allows developers to (1) analyze how existing Beans (components) work, (2) customize their behavior and appearance, (3) establish mechanisms for coordination and communication (4) develop custom Beans for use in a specific application, and (5) test and evaluate Bean behavior.

Analysis and Design for Reuse: Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with objectoriented methods, testing, software quality assurance (SQA), and correctness verification methods all contribute to the creation of software components that are reusable.

The requirements model is analyzed to determine those elements that point to existing reusable components. Elements of the requirements model are compared to WebRef descriptions of reusable components in a process that is sometimes referred to as "specification matching". If specification matching points to an existing component that fits the needs of the current application, you can extract the component from a reuse library (repository) and use it in the design of a new system. If components cannot be found (i.e., there is no match), a new component is created i.e design for reuse (DFR) should be considered.

Standard data. The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures. Standard interface protocols. Three levels of interface protocol should be established: the nature of intramodular interfaces, the design of external technical (nonhuman) interfaces, and the human-computer interface.

Program templates. An architectural style is chosen and can serve as a template for the architectural design of a new software. Once standard data, interfaces, and program templates have been established, you have a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

Classifying and Retrieving Components: Consider a large component repository. Tens of thousands of reusable software components reside in it.

A reusable software component can be described in many ways, but an ideal description encompasses the 3C model—concept, content, and context. The concept of a software component is “a description of what the component does”. The interface to the component is fully described and the semantics—represented within the context of pre- and post conditions— is identified. The content of a component describes how the concept is realized. The context places a reusable software component within its domain of applicability.

A reuse environment exhibits the following characteristics:

- i. A component database capable of storing software components and the classification
- ii. information necessary to retrieve them.
- iii. A library management system that provides access to the database.
- iv. A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- v. CBSE tools that support the integration of reused components into a new design or implementation.

➤ Critical systems development

A critical system is any system whose 'failure' could threaten human life, the system's environment or the existence of the organisation which operates the system. 'Failure' in this context does NOT mean failure to conform to a specification but means any potentially threatening system behaviour.

Examples of critical systems

Communication systems such as telephone switching systems, aircraft radio systems, etc.

Embedded control systems for process plants, medical devices, etc.

Command and control systems such as air-traffic control systems, disaster management systems, etc.

Financial systems such as foreign exchange transaction systems, account management systems, etc.

Critical systems usage

Most critical systems are now computer-based systems

Critical systems are becoming more widespread as society becomes more complex and more complex activities are automated

People and operational processes are very important elements of critical systems - they cannot simply be considered in terms of hardware and software

Critical systems failure

The cost of failure in a critical system is likely to exceed the cost of the system itself As well as direct failure costs, there are indirect costs from a critical systems failure. These may be significantly greater than the direct costs

Society's views of critical systems are not static - they are modified by each highprofile system failure

Critical systems development

Critical systems attributes are NOT independent - the systems development process must be organised so that all of them are satisfied at least to some minimum level More rigorous (and expensive) development techniques have to be used for critical systems development because of the potential cost of failure

Developing reliable systems

Reliable systems should be 'fault-free' systems where 'fault-free' means that the system's behaviour always conforms to its specification

Systems which are 'fault-free' may still fail because of specification or operational errors

The cost of producing reliable systems grows exponentially as reliability requirements are increased. In reality, we can never be sure that we have produced a 'fault-free' system.

Reliability achievement

Achieving systems reliability is generally based on the notion that system failures may be reduced by reducing the number of system faults

Fault reduction techniques

Fault avoidance

Fault detection

Alternatively, reliability can be achieved by ensuring faults do not result in failures Fault tolerance

➤ What is software reuse?

Software reuse is a term used for developing the software by using the existing software components. Some of the components that can be reuse are as follows;

Source code

Design and interfaces

User manuals

Software Documentation

Software requirement specifications and many more.

What are the advantages of software reuse?

Less effort: Software reuse requires less effort because many components use in the system are ready made components.

Time-saving: Re-using the ready made components is time saving for the software team.

Reduce cost: Less effort, and time saving leads to the overall cost reduction.

Increase software productivity: when you are provided with ready made components, then you can focus on the new components that are not available just like ready made components.

Utilize fewer resources: Software reuse save many sources just like effort, time, money etc.

Leads to a better quality software: Software reuse save our time and we can consume our more time on maintaining software quality and assurance.

What are Commercial-off-the-shelf and Commercial-off-the-shelf components? Commercial-off-the-shelf is ready-made software. Commercial-off-the-shelf software components are ready-made components that can be reused for a new software.

What is reuse software engineering?

Reuse software engineering is based on guidelines and principles for reusing the existing software.

What are stages of reuse-oriented software engineering?

Requirement specification:

First of all, specify the requirements. This will help to decide that we have some existing software components for the development of software or not.

Component analysis

Helps to decide that which component can be reused where.

Reuse System design

If the requirements are changed by the customer, then still existing system designs are helpful for reuse or not.

Development

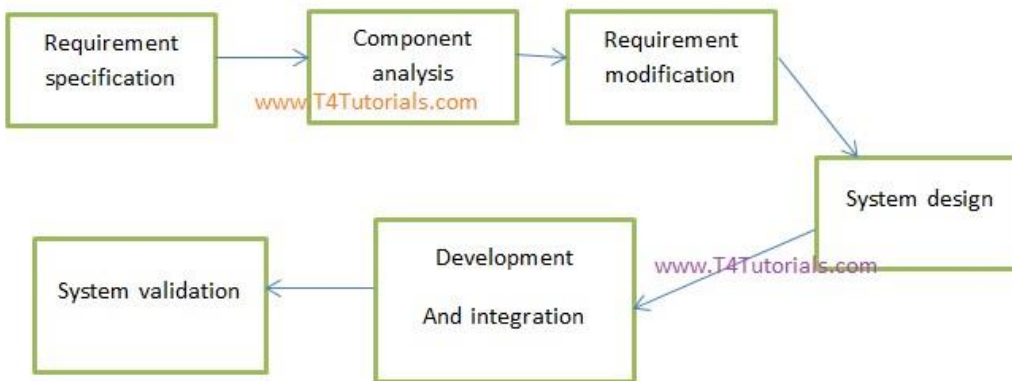
Existing components are matching with new software or not.

Integration

Can we integrate the new systems with existing components?

System validation

To validate the system that it can be accepted by the customer or not.



software reuse success factors

Capturing Domain Variations

Easing Integration

Understanding Design Context

Effective Teamwork

Managing Domain Complexity

How does inheritance promote software re-usability?

Inheritance helps in the software re-usability by using the existing components of the software to create new component.

In object oriented programming protected data members are accessible in the child and so we can say that yes inheritance promote software re-usability.

➤ USER INTERFACE DESIGN

What is it? User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the

computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

How do I ensure that I've done it right? An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

THE GOLDEN RULES

The three golden rules on interface design are

- i. Place the user in control.
- ii. Reduce the user's memory load.
- iii. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

Place the User in Control: During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. User wanted to control the computer, not have the computer control her. Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. The result may be an interface that is easy to build, but frustrating to use. Mandel defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to “undo” any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a “macro” mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is “inside” the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to “stretch” an object is an implementation of direct manipulation.

Reduce the User's Memory Load: The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory.

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember.

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

Make the Interface Consistent: The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to

determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction. If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard the user expects this in every application he encounters. A change will cause confusion.

USER INTERFACE ANALYSIS AND DESIGN

Interface Analysis and Design Models: Four different models come into play when a user interface is to be analyzed and designed. To build an effective user interface, “all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality . In addition, users can be categorized as:

Novices. No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, frequent users. Good semantic and syntactic knowledge that often leads to the “power-user syndrome”; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user’s **mental model** (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response.

The implementation model combines the outward manifestation of the computer based system (look and feel interface), coupled with all supporting information (books, manuals, videotapes, help) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks." **The Process:** The analysis and design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure 11.1, the

four distinct framework activities:

(1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation.

The spiral shown in Figure 11.1 implies that each of these tasks will

occur more than once, with each pass

around the spiral representing additional elaboration of requirements and the resultant design.

In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed. Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated.

Finally, analysis of the user environment focuses on the physical work environment.

Among the questions to be asked are

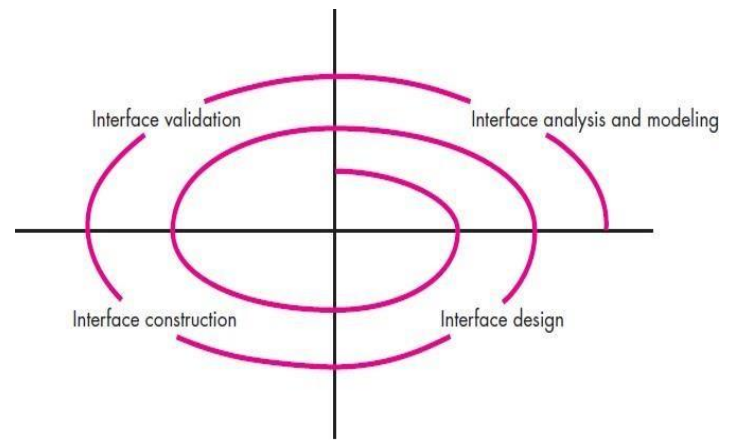


Fig 11.1: The user interface design process

Where will the interface be located physically?

Will the user be sitting, standing, or performing other tasks unrelated to the interface?

Does the interface hardware accommodate space, light, or noise constraints?

Are there special human factors considerations driven by environmental factors?

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.

Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

INTERFACE ANALYSIS

A key tenet of all software engineering process models is: understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. We examine these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

User Analysis: The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

User Interviews. The most direct approach, members of the software team meet with end

users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions will help you to better understand the users of a system:

Are users trained professionals, technicians, clerical, or manufacturing workers?

What level of formal education does the average user have?

Are the users capable of learning from written materials or have they expressed a desire for classroom training?

Are users expert typists or keyboard phobic?

What is the age range of the user community?

Will the users be represented predominately by one gender?

How are users compensated for the work they perform?

Do users work normal office hours or do they work until the job is done?

Is the software to be an integral part of the work users do or will it be used only occasionally?

What is the primary spoken language among users?

What are the consequences if a user makes a mistake using the system?

Are users experts in the subject matter that is addressed by the system?

Do users want to know about the technology that sits behind the interface?

completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire

process will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swim lane diagram (a variation on the activity diagram). See Figure 11.2

Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.

The interface design

for pharmacists and

physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).

Many of the activities noted in the swim lane diagram can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the following user task and subtask hierarchy.

User task: Requests that a prescription be refilled
Provide identifying information.
Specify name.

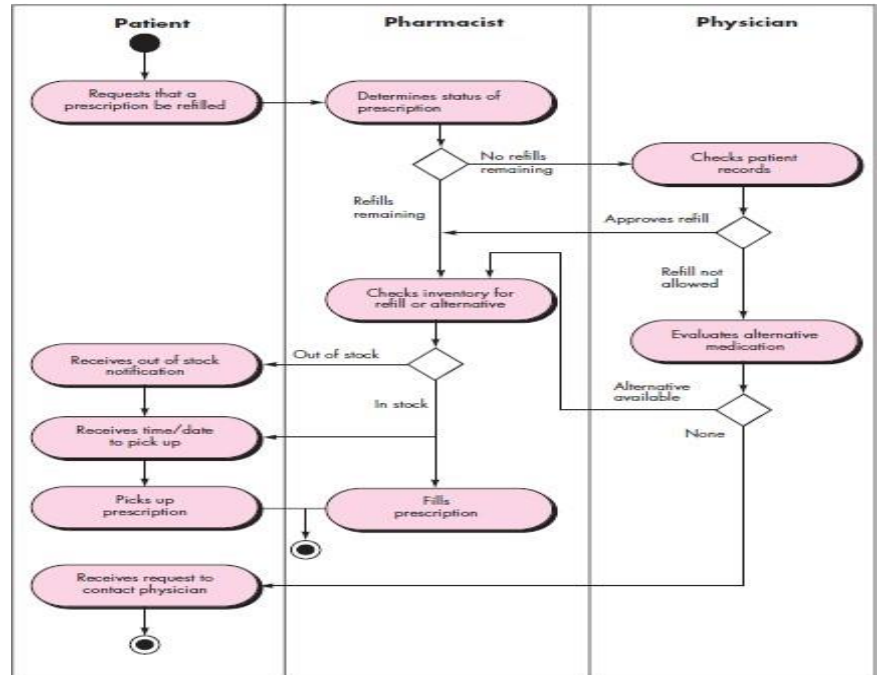


Fig 11.2: Swimlane diagram for prescription refill function

Specify userid.

Specify PIN and password.

Specify prescription number.

Specify date refill is required.

To complete the task, three subtasks are defined. One of these subtasks, provide identifying information, is further elaborated in three additional sub-subtasks.

Analysis of Display Content: The user tasks lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files).

These data objects may be

(1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?

Can the user customize the screen location for content?

Is proper on-screen identification assigned to all content?

If a large report is to be presented, how should it be partitioned for ease of understanding?

Will mechanisms be available for moving directly to summary information for large collections of data?

Will graphical output be scaled to fit within the bounds of the display device that is used?

How will color be used to enhance understanding?

How will error messages and warnings be presented to the user?

The answers to these (and other) questions will help you to establish requirements for content presentation.

INTERFACE DESIGN STEPS

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design is an iterative process. Although many different user interface design models (e.g., have been proposed, all suggest some combination of the following steps: Using information developed during interface analysis define interface objects and actions (operations).

Define events (user actions) that will cause the state of the user interface to change.

Model this behavior.

Depict each interface state as it will actually look to the end user.

Indicate how the user interprets the state of the system from information provided through the interface.

Applying Interface Design Steps: The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.

User Interface Design Patterns: Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. A vast array of interface design patterns has been proposed over the past decade.

Design Issues: As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling

Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time.

Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues must be addressed when a help facility is considered: Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions. How will the user request help? Options include a help menu, a special function key, or a HELP command.

How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.

Error handling. Error messages and warnings are “bad news” delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.

In general, every error message or warning produced by an interactive system should have the following characteristics:

The message should describe the problem in jargon that the user can understand.

The message should provide constructive advice for recovering from the error.

The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the “error color.”

The message should be “nonjudgmental.” That is, the wording should never place blame on the user.

Menu and command labeling. The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

Application accessibility. As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines - many designed for Web applications but often applicable to all types of software— provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others provide specific guidelines for “assistive technology” that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments. ➤ **Web Apps Design Issues**

In plain terms, an application is a software program that runs on a device. A web app is one that’s stored on the Internet and can be accessed from any browser. They aren’t constrained to any one operating system, don’t take up hard drive space on your device, and can be used by anyone with an Internet connection.

Web applications can include everything from productivity-enhancing collaboration tools (e.g. Google Drive and Slack) to less productive (but often fun) games (e.g. Candy Crush and Words with Friends). Popular web apps are also often available as mobile apps too.

Web application development challenges during planning

1. Clearly defining your goals.

Clearly defining your goals and requirements will make or break your web app. Many of the other challenges that we'll talk about later, such as application performance and speed, can be almost entirely addressed by making thoughtful choices during the planning stage.

It all starts with your vision for your app. That affects everything that follows. The key requirements for an enterprise-level collaborative app are different than the requirements for a fitness app or a game.

There are a few key considerations in any development process:

- Who are your intended users?
- What experience do you want to give them?
- What are your must-have design features?
- What are your technical requirements?

Once you have defined all of these things, you can start to address them

2. Choosing the right tech stack.

Throughout this section we're going to talk about tech stacks. They're a combination of programming languages, frameworks, servers, software and other tools used by developers. In essence, they're the set of technologies a web development and design agency uses to build a web or mobile app development.

You should always choose a tech stack that aligns with the problems you are trying to solve. For example, you probably won't need a complex tech stack for a simple web application, or a tech stack that helps you optimize for scalability when your user base will be a consistent size.

You should consider whether your tech stack is widely used in the industry. With industrystandard tech stacks, you will have a large pool of skilled developers to draw from for your initial build and future requirements.

Make sure to choose a tech stack that has thorough documentation. It's almost inevitable that you'll run into a troubleshooting problem at some point during development. Great documentation and support can save you time, money and hassle.

3. UX

User experience (UX) encapsulates the reactions, perceptions, and feelings your users experience while engaged in your application. It's the feeling of ease and simplicity that you get from great design. It's also the frustration that you feel when interacting with poor design.

That's why it's important to think about the overall impression you want to leave on your users before you start making detailed decisions about how to build it.

If you deliver effectively to the deeper needs of all your users – providing pleasure, engagement, and an overall emotional appeal – your application can become a central part of your users' day-to-day lives and deliver to your business goals.

4. UI & simplistic design

User Interface (UI) design includes all the visual elements your users interact with on your web application. It is everything that your users see on their screens and everything they click on to guide them through the experience.

Great UI design certainly makes your application visually appealing, but it goes beyond just simple aesthetics. The goal is to make the actual user experience simple and accessible – and usable. This means using only a targeted, purposeful selection of copy and content, making clear the options your users have throughout the experience and ensuring information is readily available at each step.

Intuitive UI typically involves:

- Clear navigation
- Engaging visuals
- Easy-to-read typography

Here's a succinct way to think about the relationship between [UX design and UI design](#): great UI makes it easy for your users while great UX makes it meaningful.

5. Performance and speed

No user likes slow load times. And they can have real consequences for your business. If your application is slow, users won't wait. They'll leave. This is the reality of web application

development issues today. You may only get a single chance to hook a user on your product.

So, if you know that you're building an application with a lot of content (e.g. videos), you should outline that upfront so your developers can build a more robust application to ensure performance. Likewise, be clear about any intentions you have to scale your application rapidly. You don't want your application to slow down if it makes a splash in the market or you see periodic surges in traffic.

This kind of planning for the future will ensure your initial app launch can deliver the kind of speed and performance you want for your early, often-critical, users. It will also mean that your initial build will provide the foundation you need for future business growth.

6. Scalability The challenge of scalability relates to how you want your application to develop over time. If you want your application built right today, you'll need to know as much as possible about what you need it to do in the future.

Your application might include fairly lean content at launch but, a year or two in, you could be planning a detailed, expansive content-rich experience. This is where extensibility comes in. Extensibility is when an application is initially designed to incorporate new capabilities and functionality in the future. This can be integrated right from the start of the development process. If you can articulate a long-term vision for your app, it can be built to grow and evolve over time.

7. Web security threats

There are a number of things to prioritize so your application and your users are secure. Choosing the right development infrastructure is one. Make sure that the infrastructure you build on has enough security services and options so your developers can implement the proper security measures for your application.

SSL certificates are a global standard security technology that enables encrypted communication between your web browser and server. When integrated in your app, they enhance its security and eliminate the chance it is flagged as unsecure by web browsers. SSL certificates also help protect credit-card numbers in ecommerce transactions and other sensitive user information like usernames, passwords and email addresses. In essence, SSL

allows for a private “conversation” just between the two intended parties and hides sensitive information from hackers and identity thieves.

Creating robust password requirements and multi-factor authentication for your users are also effective security measures. More complex passwords are less likely to be hacked. Multi-factor authentication, where your users take multiple steps to confirm their identities, also gives your application an added layer of protection. **Conclusion**

- Planning strategically, and in detail, will mitigate your toughest development challenges. This is where you should always start. Set business goals. Then ensure everything that follows is in service to them.
- Always remember that your web application will feature a blend of design elements and build (technical) elements. It needs to deliver to both if you want to meet your business goals. You can’t overcome bad design with a good build, or the reverse.
- Your application is for your users. Remember you are serving your own business goals by serving your users first, last and always.

➤ What Is Web Application Architecture?

Web app architecture is a kind of software architecture that describes the processes associated with programs running in a browser. That’s why the architecture of mobile apps or programs for IoT uses another type of digital architecture.

Web app architecture contains a set of components and a description of their logic interaction. Within the web development process in general, it determines the future design of your product, its IT infrastructure, user experience, software modules (web architecture design), as well as the promotion and monetization of your future web application. Working on it is the first step when creating your product.

Based on decisions related to the high-level architecture of an app (for example, whether it will be monolithic or based on the micro service pattern), the team forms more specific technical requirements for the future software and its tiers and plans further work.

Here are the following parameters by which the quality of web app architecture can be described:

Safety level and stability

- Request processing speed
- Component reusability
- Clarity towards program code
- Ability to collect analytics and test different components independently
- Scalability of the product and its components
- System automation level

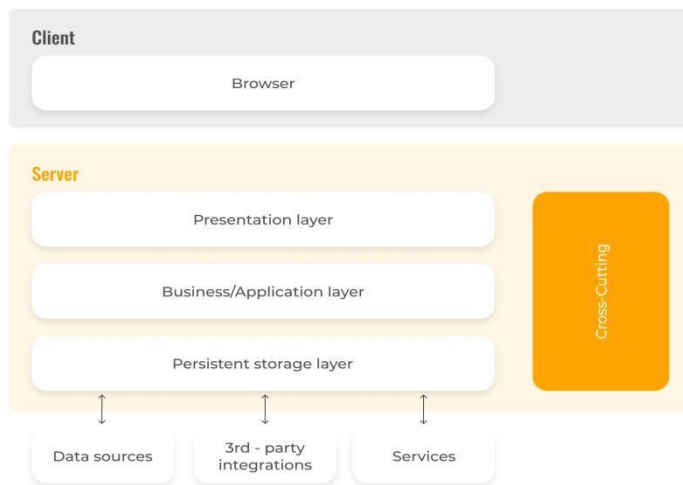
Why Does Web Architecture Matter

As we already know, web application architecture design is the first stage of software development, and web architecture, in turn, is the backbone of your web app. So what makes digital product architecture so important?

First, the web architecture is literally the foundation onto which you put all the other product components. If this foundation is solid and stable, further work on the product will be time- and cost-effective. If you make mistakes at this critical stage, all the other stages of software development, including scale changes or writing components using a particular programming language, will be slowed down.

Second, high-level web architecture is difficult to amend or modify in the later stages of product development. Typically, major changes of this kind literally mean you have to rebuild the product you want from scratch. Such a

Layers of Web App Architecture



LANARS

decision means postponing the release date, as well as losing financial resources associated with your inability to use components from the previous version of your Node.js, Python, or Java web application architecture.

Third, web app maintenance can become more expensive if you made bad

decisions at the overall architecture development stage.

Web Application Architecture Diagram

A person uses a browser to enter their request (this can be an address of the website they want to go to (URL), or a command they send via a web page interface).

The browser translates the user's request into digital language, and then determines the path to the site that the user needs and requests access to it.

1. When the server is found, it receives the browser request, processes it, and, in response to it, sends the required data.
2. The user's browser processes this data and displays the result on the user's screen: a new or updated web page.

Modern web architecture is based on the principles described in this diagram. While this scheme of user-server interaction may seem simple and straightforward, there are many

approaches and architectural patterns that are suitable for specific types of web applications.

Components of Web Application Architecture

App components are the most important part of web application architecture, but not the only part. As well as this, you will be dealing with data for your database and middleware system design. Nevertheless, it is the components of your system that determine the functions of your future product and the quality of the user experience that your web application will offer to Internet users.

Modern web application architecture operates in two categories of components: structural and UI & UX components. Components of the first group are created using programming languages such as Java, .Net, NodeJS, or Python. UI and UX components are created by designers. After being created, their design layouts are commonly sent for further implementation by programmers in the form of a working web application interface.

Structural (server) components: your server and database

UI and UX (client) components: includes interface notification elements and input controls, navigational components, admin and other dashboards, design layouts, activity tracking tools, informational elements, and many others

The goal of the UI and UX web application architecture components is to design a perfect user experience. At the same time, structural components are more about the smooth running of the web app and its features. You can use a web application model that consists of one server and one database, or you can design an application that operates on multiple servers and databases. The second approach is considered to be more sustainable and reliable.

➤ Strategic Approach to Software Testing

Testing begins at the component level and works outward toward the integration of the entire Computer-based system.

Different testing techniques are appropriate at different points in time

The developer of the software conducts testing and may be assisted by independent test groups for large projects.

The role of the independent tester is to remove the conflict of interest inherent when the builder is testing his or her own product.

Testing and debugging are different activities.

Debugging must be accommodated in any testing strategy.

Make a distinction between verification (are we building the product right?) and validation (are we building the right product?)

➤ Strategic Testing Issues

Specify product requirements in a quantifiable manner before testing starts

Specify testing objectives explicitly

Identify the user classes of the software and develop a profile for each.

Develop a test plan that emphasizes rapid cycle testing

Build robust software that is designed to test itself (e.g. uses anitbugging)

Use effective formal reviews as a filter prior to testing

Conduct formal technical reviews to assess the test strategy and test cases

Characteristics of Testing Strategies

- The process of investigating and checking a program to find whether there is an error or not and does it fulfill the requirements or not is called testing.
- When the number of errors found during the testing is high, it indicates that the testing was good and is a sign of good test case.
- Finding an unknown error that's discovered yet is a sign of a successful and a good test case. The main objective of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software.

Software Testing Myths and Facts: Just as every field has its myths, so does the field of Software Testing. Software testing myths have arisen primarily due to the following:

- i. Lack of authoritative facts.
- ii. Evolving nature of the industry.
- iii. General flaws in human logic.

Some of the myths are explained below, along with their related facts:

1. MYTH: Quality Control = Testing.

FACT: Testing is just one component of software quality control. Quality Control includes other activities such as Reviews.

2.MYTH: The objective of Testing is to ensure a 100% defect- free product.

FACT: The objective of testing is to uncover as many defects as possible. Identifying all defects and getting rid of them is impossible.

3.MYTH: Testing is easy.

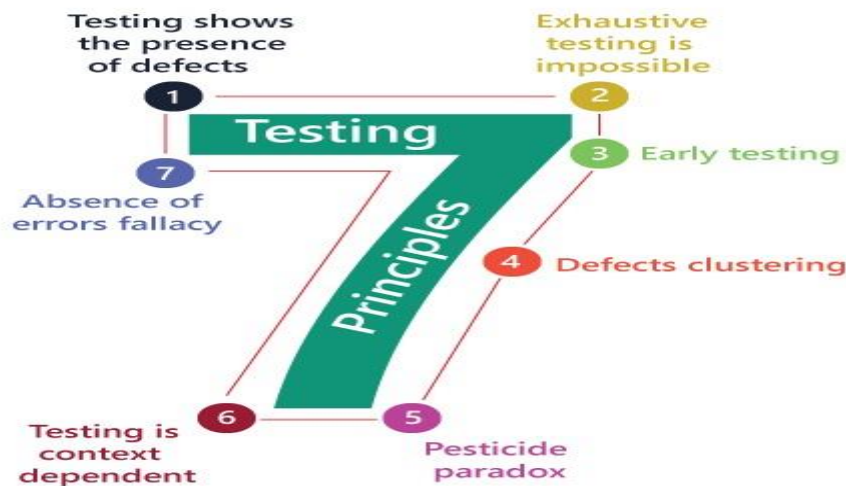
FACT: Testing can be difficult and challenging (sometimes, even more so than coding).

4. MYTH: Anyone can test.
FACT: Testing is a rigorous discipline and requires many kinds of skills.
5. MYTH: There is no creativity in testing.
FACT: Creativity can be applied when formulating test approaches, when designing tests, and even when executing tests.
6. MYTH: Automated testing eliminates the need for manual testing.
FACT: 100% test automation cannot be achieved. Manual Testing, to some level, is always necessary.
7. MYTH: When a defect slips, it is the fault of the Testers.
FACT: Quality is the responsibility of all members/stakeholders, including developers, of a project.
8. MYTH: Software Testing does not offer opportunities for career growth.
FACT: Gone are the days when users had to accept whatever product was dished to them; no matter what the quality.
With the abundance of competing software and increasingly demanding users, the need for software testers to ensure high quality will continue to grow.

Software Testing Principles

Software testing is a procedure of implementing software or the application to identify the defects or bugs. For testing an application or software, we need to follow some principles to make our product defects free, and that also helps the test engineers to test the software with their effort and time. Here, in this section, we are going to learn about the seven essential principles of software testing. Let us see the seven different testing principles, one by one:

- Testing shows the presence of defects
- Exhaustive Testing is not possible
- Early Testing
- Defect Clustering
- Pesticide Paradox
- Testing is context-dependent
- Absence of errors fallacy



Testing shows the presence of defects

The test engineer will test the application to make sure that the application is bug or defects free. While doing testing, we can only identify that the application or software has any errors. The primary purpose of doing testing is to identify the numbers of unknown bugs with the help of various methods and testing techniques because the entire test should be traceable to the customer requirement, which means that to find any defects that might cause the product failure to meet the client's needs.

By doing testing on any application, we can decrease the number of bugs, which does not mean that the application is defect-free because sometimes the software seems to be bug-free while performing multiple types of testing on it. But at the time of deployment in the production server, if the end-user encounters those bugs which are not found in the testing process.

Exhaustive Testing is not possible

Sometimes it seems to be very hard to test all the modules and their features with effective and non-effective combinations of the inputs data throughout the actual testing process.

Hence, instead of performing the exhaustive testing as it takes boundless determinations and most of the hard work is unsuccessful. So we can complete this type of variations according to the importance of the modules because the product timelines will not permit us to perform such type of testing scenarios. **Early Testing**

Here early testing means that all the testing activities should start in the early stages of the software development life cycle's **requirement analysis stage** to identify the defects because if we find the bugs at an early stage, it will be fixed in the initial stage itself, which may cost us very less as compared to those which are identified in the future phase of the testing process.

To perform testing, we will require the requirement specification documents; therefore, if the requirements are defined incorrectly, then it can be fixed directly rather than fixing them in another stage, which could be the development phase.

Defect clustering

The defect clustering defined that throughout the testing process, we can detect the numbers of bugs which are correlated to a small number of modules. We have various reasons for this, such as the modules could be complicated; the coding part may be complex, and so on.

These types of software or the application will follow the **Pareto Principle**, which states that we can identify that approx. Eighty percent of the complication is present in 20 percent of the modules. With the help of this, we can find the uncertain modules, but this method has its difficulties if the same tests are performing regularly, hence the same test will not able to identify the new defects. **Pesticide paradox**

This principle defined that if we are executing the same set of test cases again and again over a particular time, then these kinds of the test will not be able to find the new bugs in the software or the application. To get over these pesticide paradoxes, it is very significant to review all the test cases frequently. And the new and different tests are necessary to be written for the implementation of multiple parts of the application or the software, which helps us to find more bugs. **Testing is context-dependent**

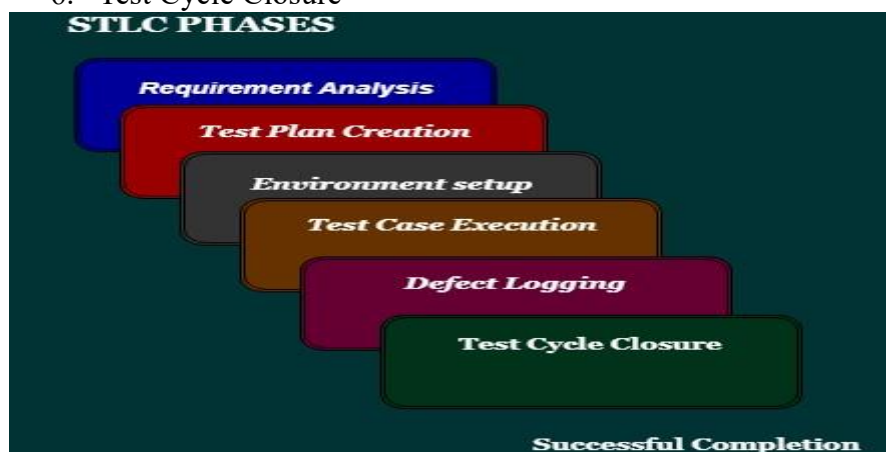
Testing is a context-dependent principle states that we have multiple fields such as e-commerce websites, commercial websites, and so on are available in the market. There is a definite way to test the commercial site as well as the e-commerce websites because every application has its own needs, features, and functionality. To check this type of application, we will take the help of various kinds of testing, different technique, approaches, and multiple methods. Therefore, the testing depends on the context of the application.

Absence of errors fallacy

Once the application is completely tested and there are no bugs identified before the release, so we can say that the application is 99 percent bug-free. But there is the chance when the application is tested beside the incorrect requirements, identified the flaws, and fixed them on a given period would not help as testing is done on the wrong specification, which does not apply to the client's requirements. The absence of error fallacy means identifying and fixing the bugs would not help if the application is impractical and not able to accomplish the client's requirements and needs **Software testing life cycle**

contains the following steps:

1. Requirement Analysis
2. Test Plan Creation
3. Environment setup
4. Test case Execution
5. Defect Logging
6. Test Cycle Closure



➤ **Test strategies for Object-Oriented Software** Introduction:

- i. Introduction: The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span.
- ii. Although this fundamental objective remains unchanged for object oriented software.
- iii. The nature of object-oriented software changes both testing strategy and testing tactics (Plan).

Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects.

This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data.

An encapsulated class is usually the focus of unit testing.

However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes.

Class testing for OO software is the equivalent of unit testing for conventional software.

Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface,

Class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

Integration Testing in the OO Context

Different strategies for integration testing of OO systems.

- i. Thread-based testing
- ii. use-based testing
- iii. Cluster testing
 - The first, thread-based testing, integrates the set of classes required to respond to one input or event for the system.
 - Each thread is integrated and tested individually.
 - Regression testing is applied to ensure that no side effects occur.
 - The second integration approach, use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) server classes.
 - After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested.
 - This sequence of testing layers of dependent classes continues until the entire system is constructed.

Cluster testing

Cluster testing is one step in the integration testing of OO software.

Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations. Validation testing

Validation testing is testing where tester performed functional and non-functional testing. Here **functional testing** includes Unit Testing (UT), Integration Testing (IT) and

System Testing (ST), and **non-functional** testing includes User acceptance testing (UAT).

Validation testing is also known as dynamic testing, where we are ensuring that "**we have developed the product right.**" And it also checks that the software meets the business needs of the client.

Unit Testing

Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

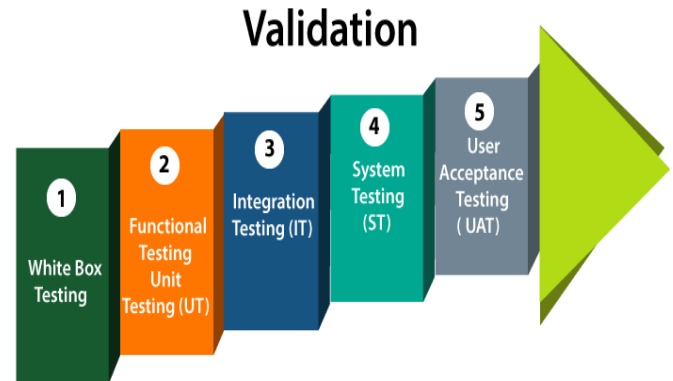
A unit is a single testable part of a software system and tested during the development phase of the application software.

The purpose of unit testing is to test the correctness of isolated code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the developers.

Whenever the application is ready and given to the Test engineer, he/she will start checking every component of the module or module of the application independently or one by one, and this process is known as **Unit testing** or **components testing**.t tests for services, in minutes Why

Unit Testing?

In a testing level hierarchy, unit testing is the first level of testing done before integration and other remaining levels of the testing. It uses modules for the testing process which reduces the dependency of waiting for Unit testing frameworks, stubs, drivers and mock objects are used for assistance in unit testing.



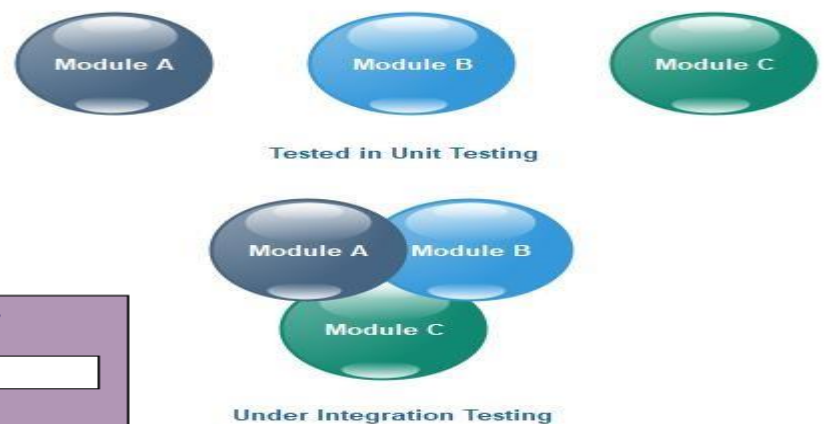
Generally, **the** software goes under four level of testing: Unit Testing, Integration Testing, System Testing, and Acceptance Testing but sometimes due to time consumption software testers does minimal unit testing but skipping of unit testing may lead to higher defects during Integration Testing, System Testing, and Acceptance Testing or even during Beta Testing which takes place after the completion of software application. **Integration testing**

Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.

modules are working independently, then we need to check the data flow between the dependent modules is known as **integration testing**.

Let us see one sample example



of a banking application, as we can see in the below image of amount transfer.

- First, we will login as a user **P** to amount transfer and send Rs200 amount, the confirmation message should be displayed on the screen as **amount transfer successfully** . Now logout as

Once all the components or P and login as user **Q** and go to amount balance page and check for a balance in that account = Present balance + Received Balance. Therefore, the integration test is successful.

- Also, we check if the amount of balance has reduced by Rs200 in P user account.
- Click on the transaction, in P and Q, the message should be displayed regarding the data and time of the amount transfer.

Guidelines for Integration Testing

- We go for the integration testing only after the functional testing is completed on each module of the application.
- We always do integration testing by picking module by module so that a proper sequence is followed, and also we don't miss out on any integration scenarios.

- First, determine the test case strategy through which executable test cases can be prepared according to test data.
- Examine the structure and architecture of the application and identify the crucial modules to test them first and also identify all possible scenarios.

- Design test cases to verify each interface in detail. ○ Choose input data for test case execution. Input data plays a significant role in testing. ○ If we find any bugs then communicate the bug reports to developers and fix defects and retest.
- Perform **positive and negative integration testing**.

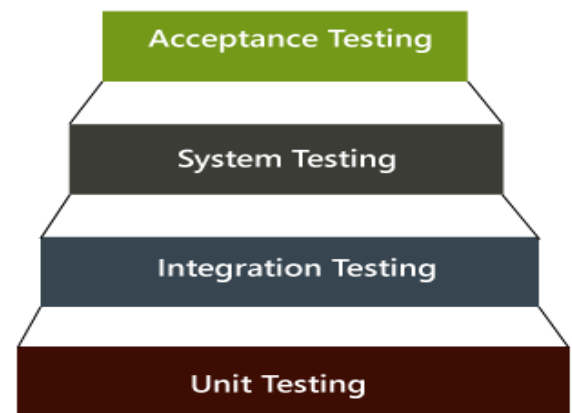
➤ System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system). The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware. System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.



To check the end-to-end flow of an application or the software as a user is known as **System testing**. In this, we navigate (go through) all the necessary modules of an application and check if the end features or the end business works fine, and test the product as a whole system. It is **end-to-end testing** where the testing environment is similar to the production environment. here are four levels of software testing

: unit testing
 , integration testing
 , system testing and acceptance testing
 , all are used for the testing purpose. Unit Testing used to test a single software; Integration Testing used to test a group of units of software, System Testing used to test a whole system and Acceptance Testing used to test the acceptability of business requirements. Here we are discussing system testing which is the third level of testing levels of India | List of Prime Minister



Hierarchy of Testing Levels

There are mainly two widely used methods for software testing, one is **White box testing** which uses internal coding to design test cases and another is black box testing which uses GUI or user perspective to develop test cases?

- White box testing ○ Black box testing

System testing falls under Black box testing as it includes testing of the external working of the software. Testing follows user's perspective to identify minor defects.

System Testing includes the following steps.

- Verification of input functions of the application to test whether it is producing the expected output or not.
- Testing of integrated software by including external peripherals to check the interaction of various components with each other.
- Testing of the whole system for End to End testing.
- Behavior testing of the application via a user's experience

Example of System testing

Suppose we open an application, let say **www.rediff.com**, and there we can see that an advertisement is displayed on the top of the homepage, and it remains there for a few seconds before it disappears. These types of Ads are done by the Advertisement Management System (AMS). Now, we will perform system testing for this type of field.

The below application works in the following manner:

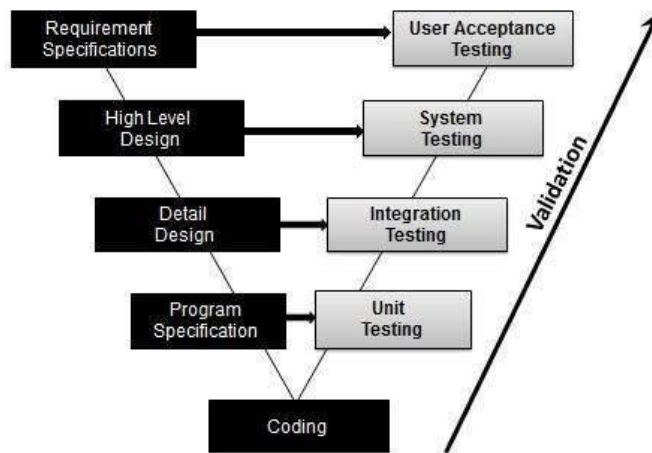
- Let's say that Amazon wants to display a promotion ad on January 26 at precisely 10:00 AM on the Rediff's home page for the country India.

- Then, the sales manager logs into the website and creates a request for an advertisement dated for the above day.
- He/she attaches a file that likely an image files or the video file of the AD and applies.
- The next day, the AMS manager of Rediffmail login into the application and verifies the awaiting Ad request.
- The AMS manager will check those Amazons ad requests are pending, and then he/she will check if the space is available for the particular date and time.
- If space is there, then he/she evaluate the cost of putting up the Ad at 15\$ per second, and the overall Ad cost for 10 seconds is approximate 150\$.
- The AMS manager clicks on the payment request and sends the estimated value along with the request for payment to the Amazon manager.
- Then the amazon manager login into the Ad status and confirms the payment request, and he/she makes the payment as per all the details and clicks on the **Submit** and **Pay**
- As soon as Rediff's AMs manager gets the amount, he/she will set up the Advertisement for the specific date and time on the Rediffmail's home page.

➤ Software Testing - Validation Testing

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment. It answers to the question, Are we building the right product?



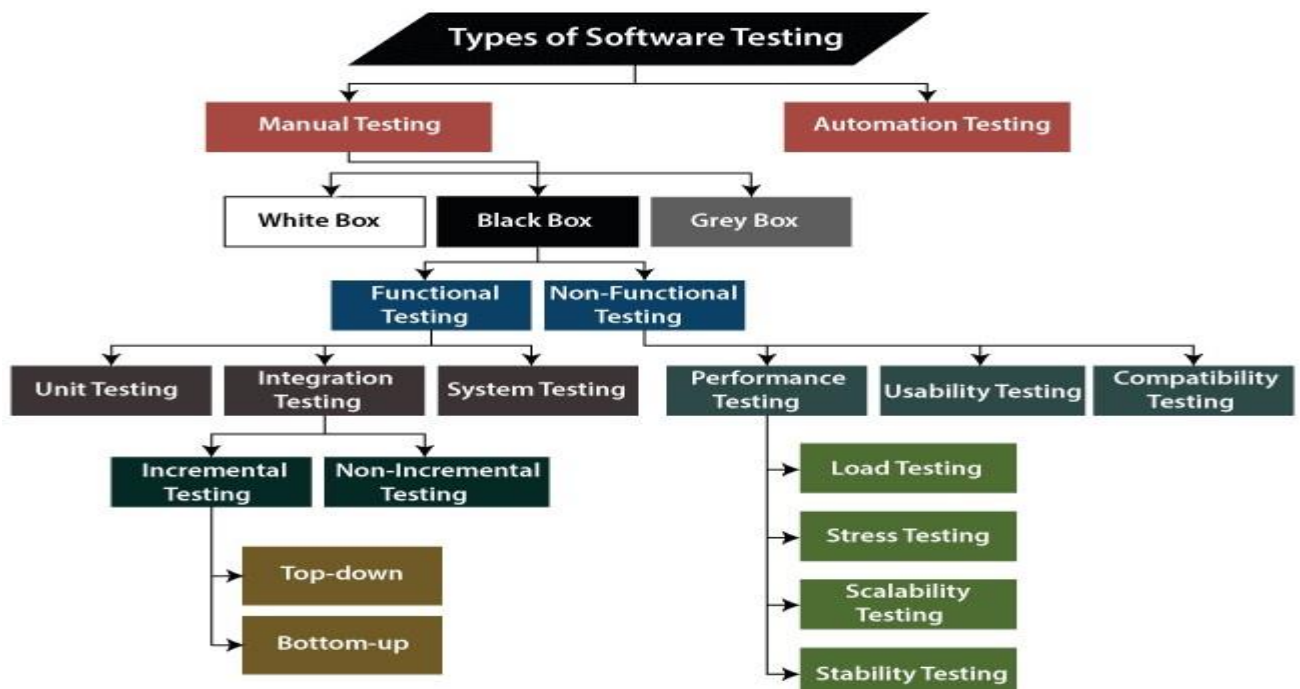
Validation Testing

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.

Activities:

- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing ➤ **Software testing tactics:**

If we want to ensure that our software is bug-free or stable, we must perform the various types of software testing because testing is the only method that makes our application bug-free.



The different types of Software Testing

The categorization of software testing is a part of diverse testing activities, such as **test strategy, test deliverables, a defined test objective, etc.** And software testing is the execution of the software to find defects.

The purpose of having a testing type is to confirm the **AUT** (Application Under Test).

To start testing, we should have a **requirement, application-ready, necessary resources available**. To maintain accountability, we should assign a respective module to different test engineers.

into two parts, which are as follows:

- **Manual Testing**
- **Automation Testing**

What is Manual Testing?

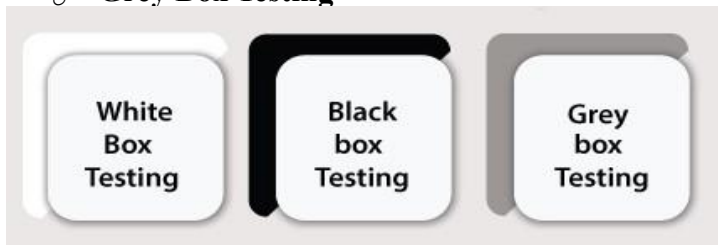
Testing any software or an application according to the client's needs without using any automation tool is known as **manual testing**.

In other words, we can say that it is a procedure of **verification and validation**. Manual testing is used to verify the behavior of an application or software in contradiction of requirements specification.

Classification of Manual Testing

In software testing, manual testing can be further classified into **three different types of testing**, which are as follows:

- **White Box Testing**
- **Black Box Testing**
- **Grey Box Testing**



For our better understanding let's see them one by one:
White Box Testing

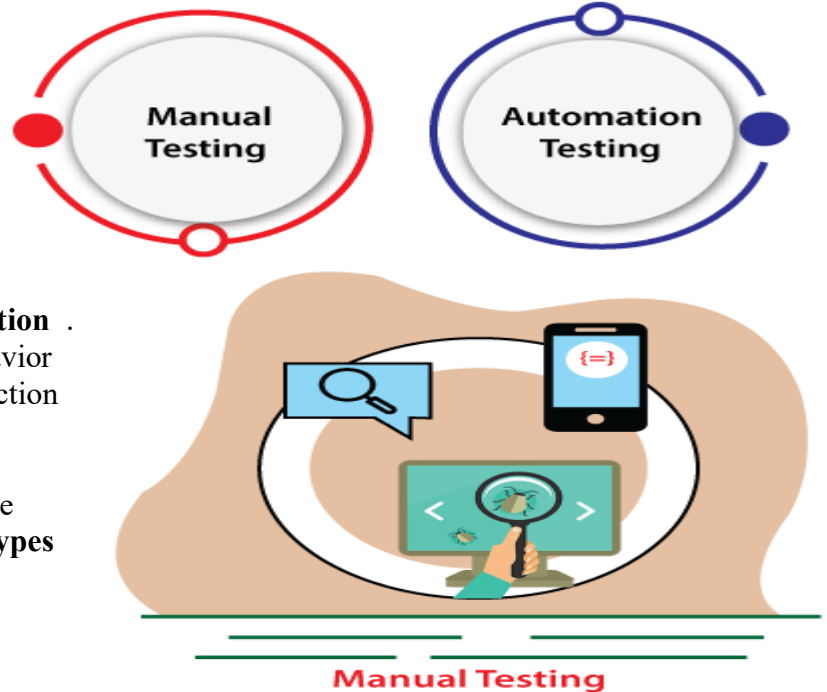
In white-box testing, the developer will inspect every line of code before handing it over to the testing team or the concerned test engineers.

Subsequently, the code is noticeable for developers throughout testing; that's why this process is known as **WBT (White Box Testing)**.

In other words, we can say that the **developer** will execute the complete white-box testing for the particular software and send the specific application to the testing team.

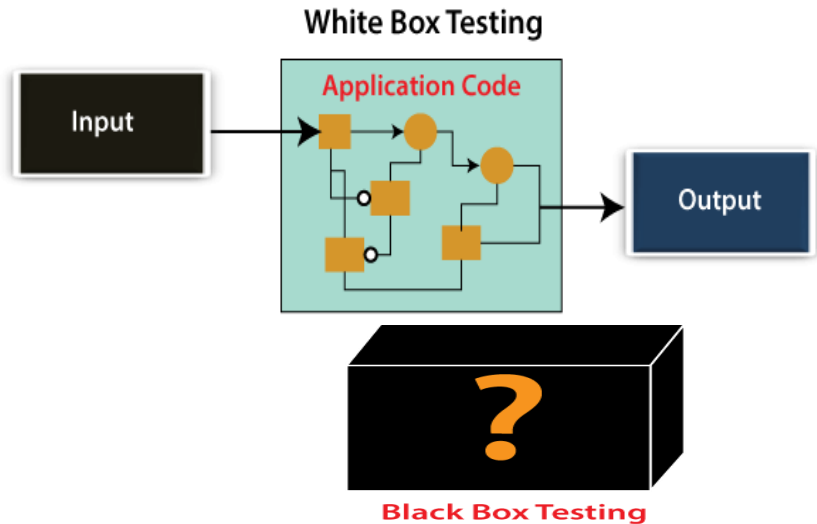
The software testing mainly divided

Types of Software Testing



White Box Testing

The purpose of implementing the white box testing is to emphasize the flow of inputs and outputs over the software and enhance the security of an application. White box testing is also known as **open box testing, glass box testing, structural testing, clear box testing, and transparent box testing.**

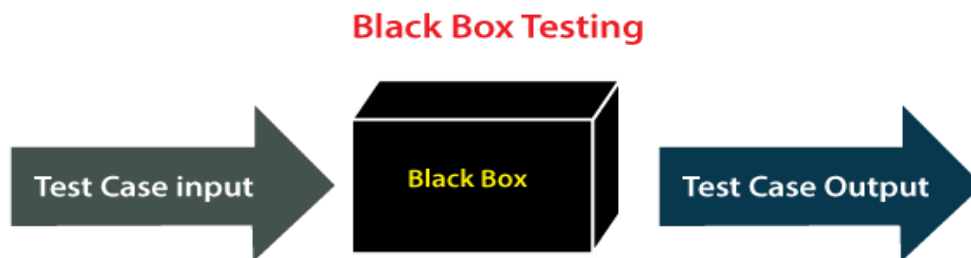


Black Box Testing

Another type of manual testing is **black-box testing**. In this testing, the test engineer will analyze the software against requirements, identify the defects or bug, and sends it back to the development team. Then, the developers will fix those defects, do one round of White box testing, and send it to the testing team. Here, fixing the bugs means the defect is resolved, and the particular feature is working according to the given requirement.

The main objective of implementing the black box testing is to specify the business needs or the customer's requirements.

In other words, we can say that black box testing is a process of checking the functionality of an application as per the customer requirement. The source code is not visible in this testing; that's why it is known as **black-box testing**.



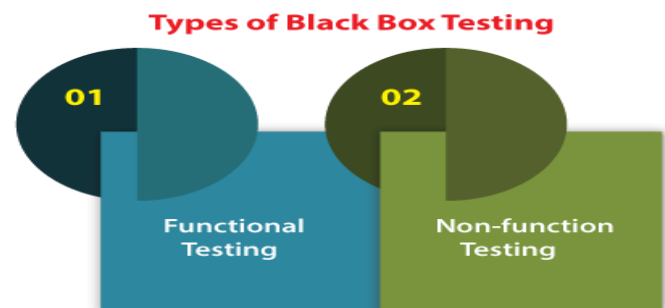
Types of Black Box Testing

Black box testing further categorizes into two parts, which are as discussed below:

- **Functional Testing** ○ **Non-function Testing**

Functional Testing

The test engineer will check all the components systematically against requirement specifications is known as **functional testing**. Functional testing is also known as **Component testing**.



In functional testing, all the components are tested by giving the value, defining the output, and validating the actual output with the expected value.

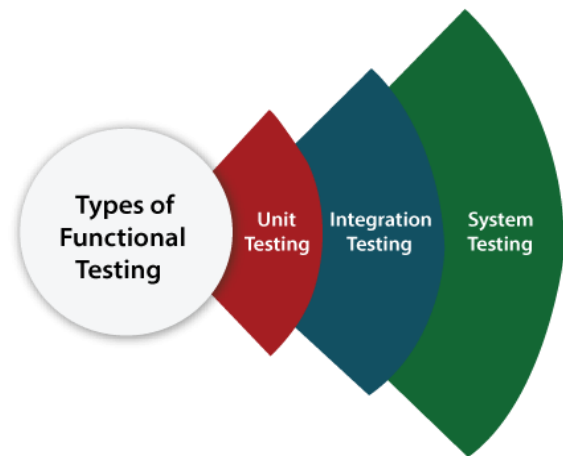
Functional testing is a part of black-box testing as emphasizes on application requirement rather than actual code. The test engineer has to test only the program instead of the system.

Types of Functional Testing

Just like another type of testing is divided into several parts, functional testing is also classified into various categories.

The diverse **types of Functional Testing** contain the following:

- **Unit Testing** ○ **Integration Testing**
- **System Testing**



1. Unit Testing

Unit testing is the first level of functional testing in order to test any software. In this, the test engineer will test the module of an application independently or test all the module functionality is called **unit testing**.

The primary objective of executing the unit testing is to confirm the unit components with their performance. Here, a unit is defined as a single testable function of a software or an application. And it is verified throughout the specified application development phase.

2. Integration Testing

Once we are successfully implementing the unit testing, we will go integration testing

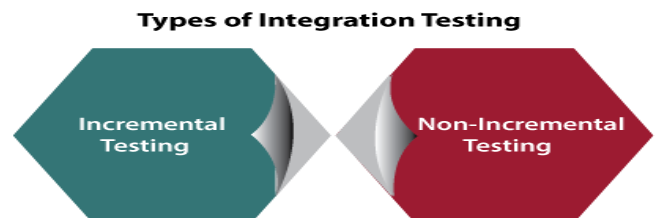
. It is the second level of functional testing, where we test the data flow between dependent modules or interface between two features is called **integration testing**.

The purpose of executing the integration testing is to test the statement's accuracy between each module.

Types of Integration Testing

Integration testing is also further divided into the following parts:

- **Incremental Testing** ○ **Non-Incremental Testing**



Whenever there is a clear relationship between modules, we go for incremental integration testing. Suppose, we take two modules and analysis the data flow between them if they are working fine or not.

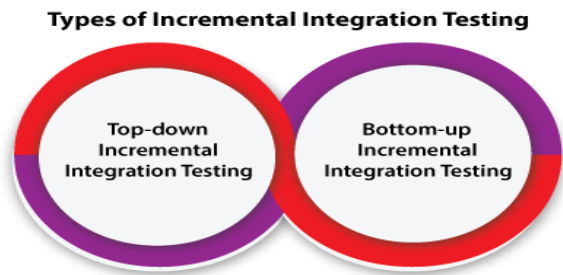
If these modules are working fine, then we can add one more module and test again. And we can continue with the same process to get better results.

In other words, we can say that incrementally adding up the modules and test the data flow between the modules is known as **Incremental integration testing**.

Types of Incremental Integration Testing

Incremental integration testing can further classify into two parts, which are as follows:

1. **Top-down Incremental Integration Testing**
2. **Bottom-up Incremental Integration Testing**



Let's see a brief introduction of these types of integration testing:

1. Top-down Incremental Integration Testing

In this approach, we will add the modules step by step or incrementally and test the data flow between them. We have to ensure that the modules we are adding are the **child of the earlier ones**.

2. Bottom-up Incremental Integration Testing

In the bottom-up approach, we will add the modules incrementally and check the data flow between modules. And also, ensure that the module we are adding is the **parent of the earlier ones**.

Non-Incremental Integration Testing/ Big Bang Method

Whenever the data flow is complex and very difficult to classify a parent and a child, we will go for the non-incremental integration approach. The non-incremental method is also known as **the Big Bang method**.

3. System Testing

Whenever we are done with the unit and integration testing, we can proceed with the system testing. In system testing, the test environment is parallel to the production environment. It is also known as **end-to-end** testing. Non-function Testing

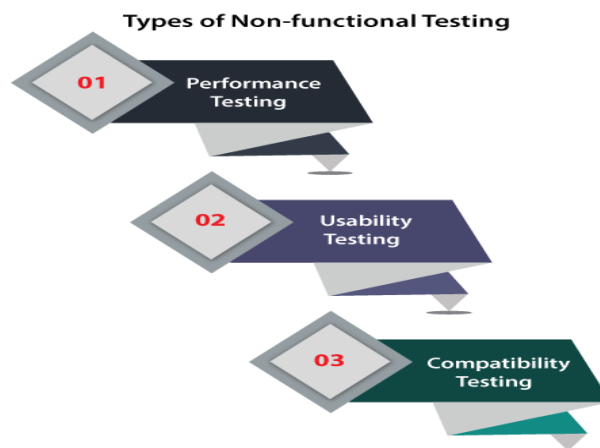
The next part of black-box testing is **non-functional testing**. It provides detailed information on software product performance and used technologies.

Non-functional testing will help us minimize the risk of production and related costs of the software.

Non-functional testing is a combination of **performance, load, stress, usability and, compatibility testing**.

Types of Non-functional Testing Non-functional testing categorized into different parts of testing, which we are going to discuss further: ○

- **Performance Testing**
- **Usability Testing**
- **Compatibility Testing**



1. Performance Testing

In performance testing, the test engineer will test the working of an application by applying some load. In this type of non-functional testing, the test engineer will only focus on several aspects, such as **Response time, Load, scalability, and Stability** of the software or an application.

Classification of Performance Testing

Performance testing includes the various types of testing, which are as follows:



- **Load Testing** ○

- **Stress Testing** ○ **Scalability Testing** ○ **Stability Testing** ○ **Load Testing**

While executing the performance testing, we will apply some load on the particular application to check the application's performance, known as **load testing**. Here, the load could be less than or equal to the desired load.

It will help us to detect the highest operating volume of the software and bottlenecks.

- **Stress Testing**

It is used to analyze the user-friendliness and robustness of the software beyond the common functional limits.

Primarily, stress testing is used for critical software, but it can also be used for all types of software applications. **Scalability Testing**

To analysis, the application's performance by enhancing or reducing the load in particular balances is known as **scalability testing**.

In scalability testing, we can also check the **system, processes, or database's ability** to meet an upward need. And in this, the **Test Cases** are designed and implemented efficiently. ○ **Stability Testing**

Stability testing is a procedure where we evaluate the application's performance by applying the load for a precise time.

It mainly checks the constancy problems of the application and the efficiency of a developed product. In this type of testing, we can rapidly find the system's defect even in a stressful situation.

2. Usability Testing

Another type of **non-functional testing** is **usability testing**. In usability testing, we will analyze the user-friendliness of an application and detect the bugs in the software's end-user interface. Here, the term **user-friendliness** defines the following aspects of an application:

- The application should be easy to understand, which means that all the features must be visible to end-users.
- The application's look and feel should be good that means the application should be pleasant looking and make a feel to the end-user to use it.

3. Compatibility Testing

In compatibility testing, we will check the functionality of an application in specific hardware and software environments. Once the application is functionally stable then only, we go for **compatibility testing**.

Here, **software** means we can test the application on the different operating systems and other browsers, and **hardware** means we can test the application on different sizes.

Grey Box Testing

Another part of **manual testing** is **Grey box testing**. It is a **collaboration of black box and white box testing**.

Since, the grey box testing includes access to internal coding for designing test cases. Grey box testing is performed by a person who knows coding as well as testing.



In other words, we can say that if a single person team done both **white box and blackbox testing**, it is considered **grey box testing**. **Automation Testing**

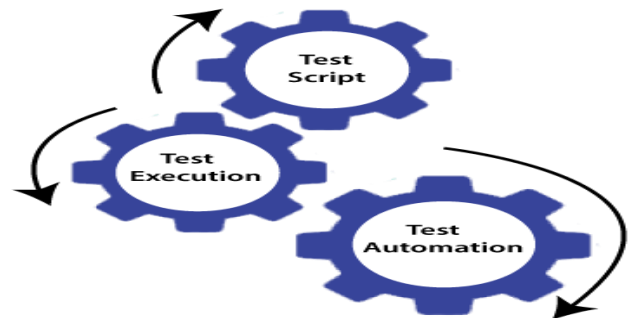
The most significant part of Software testing is Automation testing. It uses specific tools to automate manual design test cases without any human interference.

Automation testing is the best way to enhance the efficiency, productivity, and coverage of Software testing.

It is used to re-run the test scenarios, which were executed manually, quickly, and repeatedly.

In other words, we can say that whenever we are testing an application by using some tools is known as **automation testing**.

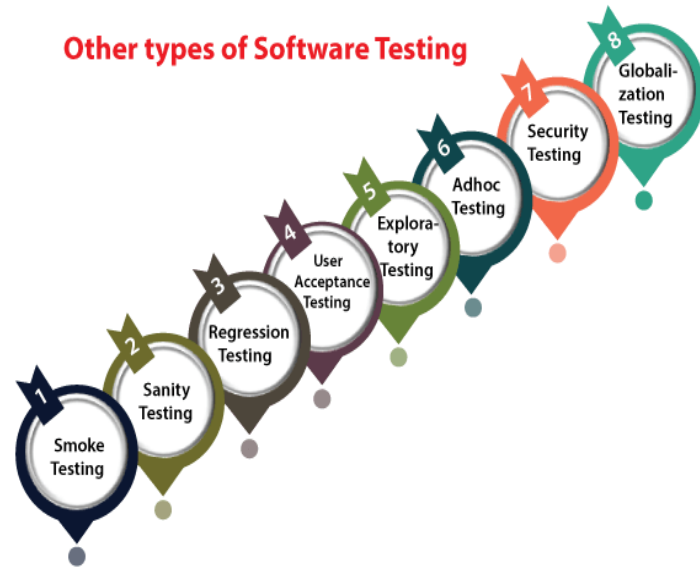
We will go for automation testing when various releases or several regression cycles goes on the application or software. We cannot write the test script or perform the automation testing without understanding the programming language.



Some other types of Software Testing

In software testing, we also have some other types of testing that are not part of any above discussed testing, but those testing are required while testing any software or an application.

- Smoke Testing
- Sanity Testing
- Regression Testing
- User Acceptance Testing
- Exploratory Testing
- Adhoc Testing
- Security Testing
- Globalization Testing



Let's understand those types of testing one by one:

In smoke testing, we will test an application's basic and critical features before doing one round of deep and rigorous testing.

Or before checking all possible positive and negative values is known as **smoke testing**. Analyzing the workflow of the application's core and main functions is the main objective of performing the smoke testing. Sanity Testing

It is used to ensure that all the bugs have been fixed and no added issues come into existence due to these changes. Sanity testing is unscripted, which means we cannot document it. It checks the correctness of the newly added features and components.

Regression Testing

Regression testing is the most commonly used type of software testing. Here, the term **regression** implies that we have to re-test those parts of an unaffected application.

Regression testing is the most suitable testing for automation tools. As per the project type and accessibility of resources, regression testing can be similar to **Retesting**.

Whenever a bug is fixed by the developers and then testing the other features of the applications that might be simulated because of the bug fixing is known as **regression testing**.

In other words, we can say that whenever there is a new release for some project, then we can perform Regression Testing, and due to a new feature may affect the old features in the earlier releases. **User Acceptance Testing**

The User acceptance testing (UAT) is done by the individual team known as domain expert/customer or the client. And knowing the application before accepting the final product is called as **user acceptance testing**.

In user acceptance testing, we analyze the business scenarios, and real-time scenarios on the distinct environment called the **UAT environment**. In this testing, we will test the application before UAT for customer approval. exploratory Testing

Whenever the requirement is missing, early iteration is required, and the testing team has experienced testers when we have a critical application. New test engineer entered into the team then we go for the **exploratory testing**.

To execute the exploratory testing, we will first go through the application in all possible ways, make a test document, understand the flow of the application, and then test the application. Adhoc Testing Testing the application randomly as soon as the build is in the checked sequence is known as **Adhoc testing**.

It is also called **Monkey testing and Gorilla testing**. In Adhoc testing, we will check the application in contradiction of the client's requirements; that's why it is also known as **negative testing**.

When the end-user using the application casually, and he/she may detect a bug. Still, the specialized test engineer uses the software thoroughly, so he/she may not identify a similar detection.

Security Testing

It is an essential part of software testing, used to determine the weakness, risks, or threats in the software application.

The execution of security testing will help us to avoid the nasty attack from outsiders and ensure our software applications' security. **Globalization Testing**

Another type of software testing is **Globalization testing**. Globalization testing is used to check the developed software for multiple languages or not. Here, the words **globalization** means enlightening the application or software for various languages.

Globalization testing is used to make sure that the application will support multiple languages and multiple features.

In present scenarios, we can see the enhancement in several technologies as the applications are prepared to be used globally.

Levels of Testing

What are the levels of Software Testing?

Testing levels are the procedure for finding the missing areas and avoiding overlapping and repetition between the development life cycle stages. We have already seen the various phases such as **Requirement collection, designing, coding testing, deployment, and maintenance** of SDLC (Software Development Life Cycle)

In order to test any application, we need to go through all the above phases of SDLC. Like SDLC, we have multiple levels of testing, which help us maintain the quality of the software.

Different Levels of Testing

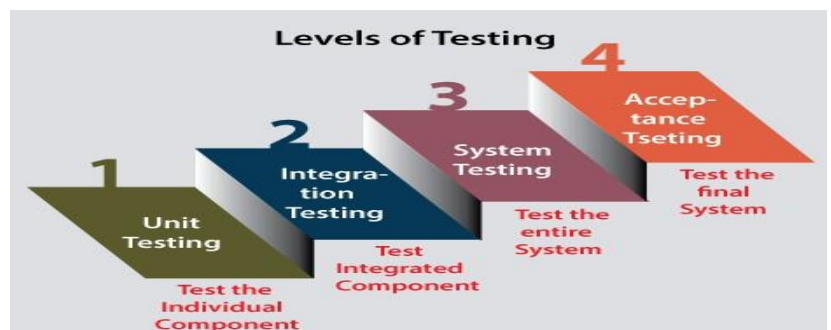
The levels of software testing involve the different methodologies, which can be used while we are performing the software testing. In software testing we have four different levels of testing, which are as discussed below:

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**
4. **Acceptance Testing**

As we can see in the above image that all of these testing levels have a specific objective which specifies the value to the software development lifecycle.

For our better understanding, let's see them one by one:

Level1: Unit Testing



Unit testing is the first level of software testing, which is used to test if software modules are satisfying the given requirement or not.

The first level of testing involves **analyzing each unit or an individual component** of the software application.

Unit testing is also the first level of **functional testing**

The primary purpose of executing unit testing is to validate unit components with their performance. A unit component is an individual function or regulation of the application, or we can say that it is the smallest testable part of the software. The reason of performing the unit testing is to test the correctness of inaccessible code.

Unit testing will help the test engineer and developers in order to understand the base of code that makes them able to change defect causing code quickly. The developers implement the unit.

Level2: Integration Testing

The second level of software testing is the **integration testing**. The integration testing process comes after **unit testing**.

It is mainly used to test the **data flow from one module or component to other modules**.

In integration testing, the **test engineer** tests the units or separate components or modules of the software in a group.

The primary purpose of executing the integration testing is to identify the defects at the interaction between integrated components or units.

When each component or module works separately, we need to check the data flow between the dependent modules, and this process is known as **integration testing**.

We only go for the integration testing when the functional testing has been completed successfully on each application module.

In simple words, we can say that **integration testing** aims to evaluate the accuracy of communication among all the modules.

Level3: System Testing

The third level of software testing is **system testing**, which is used to test the software's functional and non-functional requirements.

It is **end-to-end testing** where the testing environment is parallel to the production environment. In the third level of software testing, **we will test the application as a whole system**.

To check the end-to-end flow of an application or the software as a user is known as **System testing**. In system testing, we will go through all the necessary modules of an application and test if the end features or the end business works fine, and test the product as a complete system.

In simple words, we can say that System testing is a sequence of different types of tests to implement and examine the entire working of an integrated software computer system against requirements.

Level4: Acceptance Testing

The **last and fourth level** of software testing is **acceptance testing**, which is used to evaluate whether a specification or the requirements are met as per its delivery.

The software has passed through three testing levels (**Unit Testing, Integration Testing, System Testing**). Some minor errors can still be identified when the end-user uses the system in the actual scenario.

In simple words, we can say that Acceptance testing is the **squeezing of all the testing processes that are previously done**.

The acceptance testing is also known as **User acceptance testing (UAT)** and is done by the customer before accepting the final product.

Usually, UAT is done by the domain expert (customer) for their satisfaction and checks whether the application is working according to given business scenarios and real-time scenarios.

➤ White Box Testing

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**. It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing. In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software. The term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Developers do white box testing. In this, the developer will test every line of the code of the program. The developers perform the White-box testing and then send the application or the software to the testing team, where they will perform the black box testing and verify the application along with the requirements and identify the bugs and sends it to the developer.

The developer fixes the bugs and does one round of white box testing and sends it to the testing team. Here, fixing the bugs implies that the bug is deleted, and the particular feature is working fine on the application.

Here, the test engineers will not include in fixing the defects for the following reasons:

- Fixing the bug might interrupt the other features. Therefore, the test engineer should always find the bugs, and developers should still be doing the bug fixes.
- If the test engineers spend most of the time fixing the defects, then they may be unable to find the other bugs in the application.

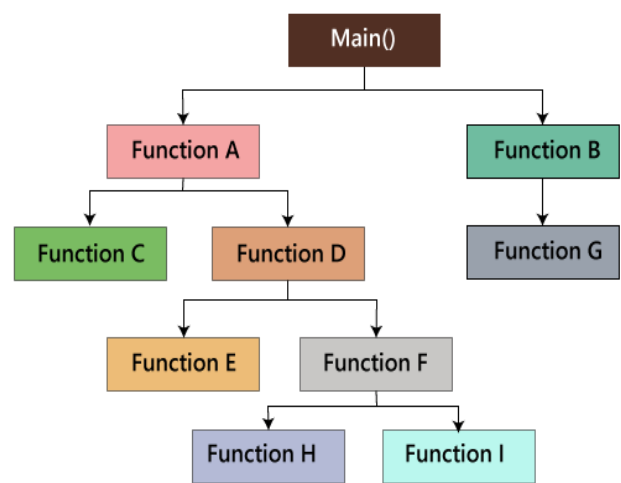
The white box testing contains various tests, which are as follows:

- Path testing
- Loop testing
- Condition testing
- Testing based on the memory perspective
- Test performance of the program

Path testing

In the path testing, we will write the flow graphs and test all independent paths. Here writing the flow graph implies that flow graphs are representing the flow of the program and also show how every program is added with one another as we can see in the below image:

And test all the independent paths implies that suppose a path from main() to function G, first set the parameters and test if the program is correct in



that particular path, and in the same way test all other paths and fix the bugs. Loop testing
In the loop testing, we will test the loops such as while, for, and do-while, etc. and also check for ending condition if working correctly and if the size of the conditions is enough.

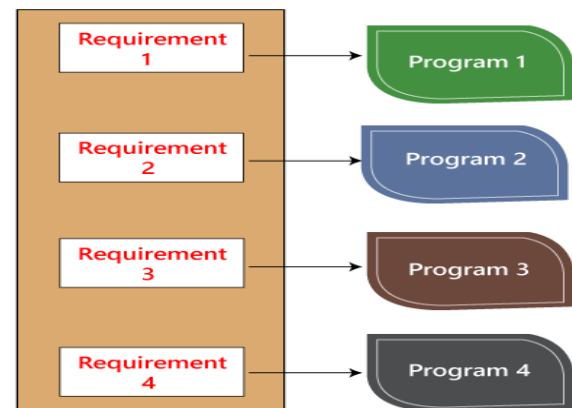
For example: we have one program where the developers have given about 50,000 loops.

```
1.  {
2.  while(50,000)
3.  .....
4.  .....
5.  }
```

We cannot test this program manually for all the 50,000 loops cycle. So we write a small program that helps for all 50,000 cycles, as we can see in the below program, that test P is written in the similar language as the source code program, and this is known as a Unit test. And it is written by the developers only.

```
1.  Test P
2.  {
3.  .....
4.  ..... }
```

As we can see in the below image that, we have various requirements such as 1, 2, 3, 4. And then, the developer writes the programs such as program 1,2,3,4 for the parallel conditions. Here the application contains the 100s line of codes.

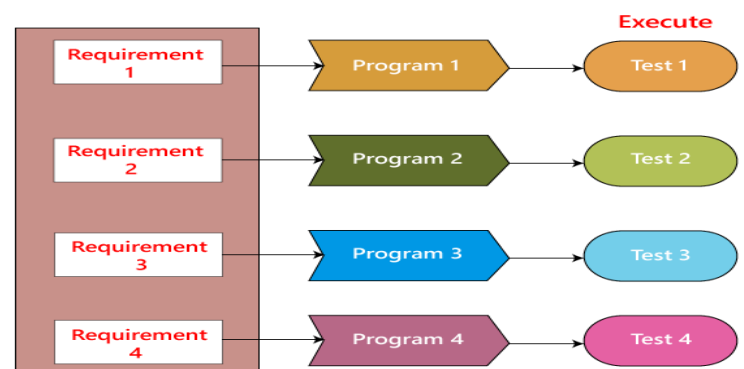


The developer will do the white box testing, and they will test all the five programs line by line of code to find the bug. If they found any bug in any of the programs, they will correct it. And they again have to test the system then this process contains lots of time and effort and slows down the product release time.

Now, suppose we have another case, where the clients want to modify the requirements, then the developer will do the required changes and test all four program again, which take lots of time and efforts.

These issues can be resolved in the following ways:

In this, we will write test for a similar program where the developer writes these test code in the related language as the source code. Then they execute these test code, which is also known as **unit test programs**. These test programs linked to the main program and implemented as programs.



Therefore, if there is any requirement of modification or bug in the code, then the developer makes the adjustment both in the main program and the test program and then executes the test program.

Condition testing

In this, we will test all logical conditions for both **true** and **false** values; that is, we will verify for both **if** and **else** condition. **For example:**

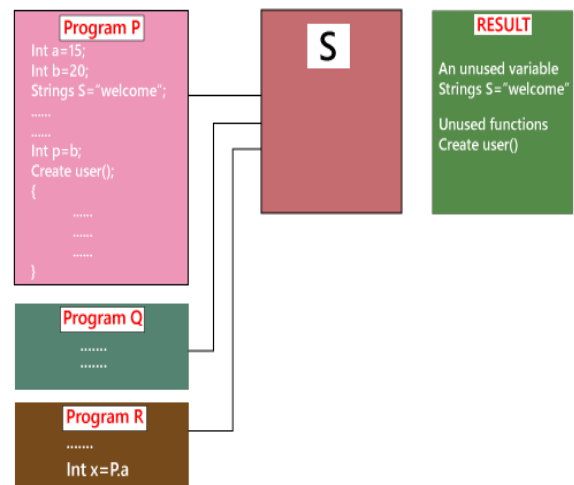
1. if(condition) - true
2. {
3.
4.
5.
6. }
7. else - false
8. {
9.
10.
11.
12. }

The above program will work fine for both the conditions, which means that if the condition is accurate, and then else should be false and conversely.

Testing based on the memory (size) perspective The size of the code is increasing for the following reasons:

- **The reuse of code is not there:** let us take one example, where we have four programs of the same application, and the first ten lines of the program are similar. We can write these ten lines as a discrete function, and it should be accessible by the above four programs as well. And also, if any bug is there, we can modify the line of code in the function rather than the entire code.
- The **developers use the logic** that might be modified. If one programmer writes code and the file size is up to 250kb, then another programmer could write a similar code using the different logic, and the file size is up to 100kb.
- The **developer declares so many functions and variables** that might never be used in any portion of the code. Therefore, the size of the program will increase. **For example,**

- 1.Int a=15;
- 2.Int b=20;
- 3.String S= "Welcome";
- 4.....
- 5.....
- 6.....
- 7.....
- 8.....
- 9.Int p=b;



```

10. Create user()
11. {
12. ....
13. ....
14. .... 200's line of code
15. }

```

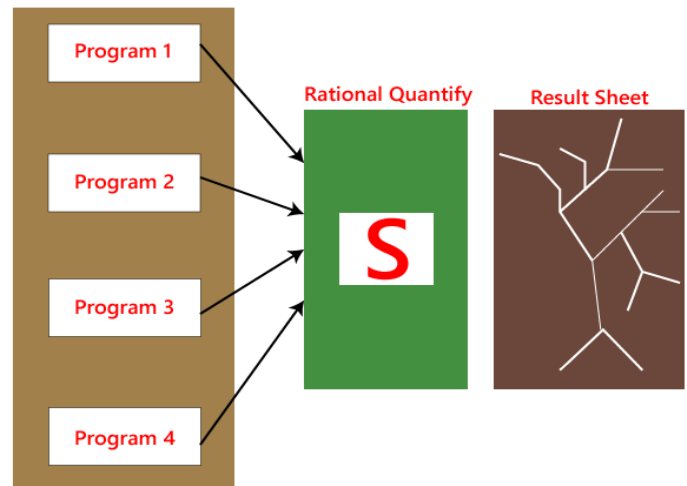
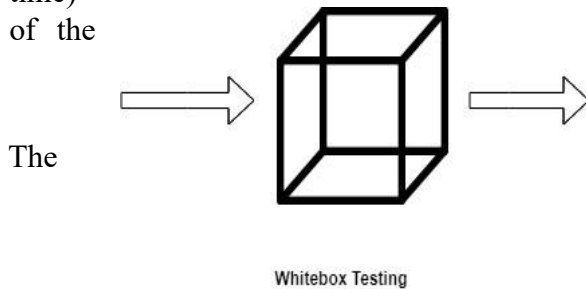
In the above code, we can see that the **integer a** has never been called anywhere in the program, and also the function **Create user** has never been called anywhere in the code. Therefore, it leads us to memory consumption.

We cannot remember this type of mistake manually by verifying the code because of the large code. So, we have a built-in tool, which helps us to test the needless variables and functions. And, here we have the tool called **Rational purify**.

Suppose we have three programs such as Program P, Q, and R, which provides the input to S. And S goes into the programs and verifies the unused variables and then gives the outcome. After that, the developers will click on several results and call or remove the unnecessary function and the variables.

- The developer does not use the available in-built functions; instead they write the full features using their logic. Therefore, it leads us to waste of time and also postpone the product releases.

Test the performance (Speed, response time) of the



program

application could be slow for the following reasons:

- When logic is used.
- For the conditional cases, we will use **or & and** adequately.
- Switch case, which means we cannot use **nested if**, instead of using a switch case.

White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Advantages of White box testing

- White box testing optimizes code so hidden errors can be identified.
- Test cases of white box testing can be easily automated.

- This testing is more thorough than other testing approaches as it covers all code paths.
- It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

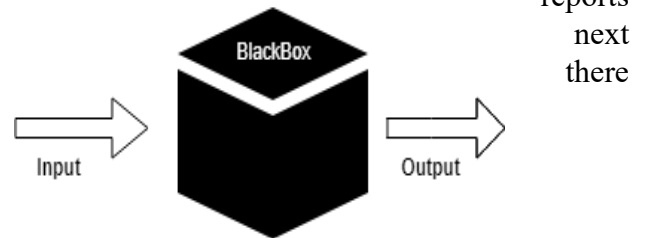
- White box testing is too much time consuming when it comes to large-scale programming applications.

- White box testing is much expensive and complex. ○ It can lead to production error because it is not detailed by the developers.
- White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation

➤ **Black box testing**

Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer.

In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.



Generic steps of black box testing

- The black box test is based on the specification of requirements, so it is examined in the beginning.
- In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.
- In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.
- The fourth phase includes the execution of all test cases. ○ In the fifth step, the tester compares the expected output against the actual output. ○ In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Test procedure

The test procedure of black box testing is a kind of process in which the tester has specific knowledge about the software's work, and it develops test cases to check the accuracy of the software's functionality.

Test cases

Test cases are created considering the specification of the requirements. These test cases are generally created from working descriptions of the software including requirements, design parameters, and other specifications. For the testing, the test designer selects both positive test scenario by taking valid input values and adverse test scenario by taking invalid input values to determine the correct output. Test cases are mainly designed for functional testing but can also be used for non-functional testing. Test cases are designed by the testing team, there is not any involvement of the development team of software.

Techniques Used in Black Box Testing

Decision Table Decision Table Technique is a systematic approach where various Technique combinations and their respective system behavior are captured in a tabula

It is appropriate for the functions that have a logical relationship between two or more than two inputs.

Boundary Value Technique is used to test boundary values, boundary values of those that contain the upper and lower limit of a variable. It tests, while e

State Transition	boundary value whether the software is producing correct output or not.
<u>Technique</u>	State Transition Technique is used to capture the behavior of the software application when different input values are given to the same function
All-pair testing	applies to those types of applications that provide the specific number of attempts to access the application.
Cause-Effect	All-pair testing Technique is used to test all the possible discrete combinations of input values. This combinational method is used for testing the application through checkbox input, radio button input, list box, text box, etc.
<u>Technique</u>	Cause-Effect Technique underlines the relationship between a given result and the factors affecting the result. It is based on a collection of requirements.
Equivalence	Equivalence partitioning is a technique of software testing in which input data is divided into partitions of valid and invalid values, and it is mandatory that each partition must exhibit the same behavior.
<u>Technique</u>	Error Guessing Error guessing is a technique in which there is no specific method for identifying the error. It is based on the experience of the test analyst, where the tester uses
<u>Technique</u>	experience to guess the problematic areas of the software.

Use Case Technique used to identify the test cases from the beginning to the end of the system as per the usage of the system. By using this

technique, the test team creates a test scenario that can exercise the entire software based on the functionality of each function from start to end.

Decision table technique in Black box testing

Decision table technique is one of the widely used case design techniques for black box testing. This is a systematic approach where various input combinations and their respective system behavior is captured in a tabular form.

That's why it is also known as a cause-effect table. This technique is used to pick the test cases in a systematic manner; it saves the testing time and gives good coverage to the testing area of the software application.

Decision table technique is appropriate for the functions that have a logical relationship between two or more than two inputs.

This technique is related to the correct combination of inputs and determines the result of various combinations of input. To design the test cases by decision table technique, we need to consider conditions as input and actions as output. **All-pairs testing**

All-pairs testing technique is also known as pair wise testing. It is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input (radio button is used when you have to select only one option for example when you select gender male or female, you can select only one option), list box, text box, etc.

Cause and Effect Graph in Black box Testing

Cause-effect graph comes under the black box testing technique which underlines the relationship between a given result and all the factors affecting the result. It is used to write dynamic test cases.

The dynamic test cases are used when code works dynamically based on user input. For example, while using email account, on entering valid email, the system accepts it but, when you enter invalid email, it throws an error message. In this technique, the input conditions are assigned with causes and the result of these input conditions with effects.

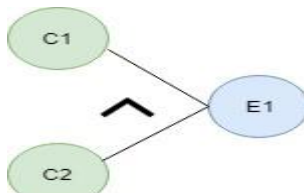
Cause-Effect graph technique is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of the software.

The main advantage of cause-effect graph testing is, it reduces the time of test execution and cost. This technique aims to reduce the number of test cases but still covers all necessary test cases with maximum coverage to achieve the desired application quality.

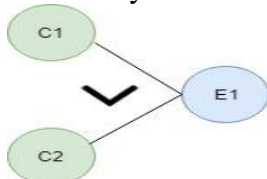
Cause-Effect graph technique converts the requirements specification into a logical relationship between the input and output conditions by using logical operators like AND, OR and NOT.

Notations used in the Cause-Effect Graph

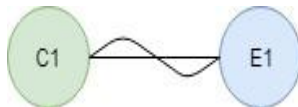
AND - E1 is an effect and C1 and C2 are the causes. If both C1 and C2 are true, then effect E1 will be true.



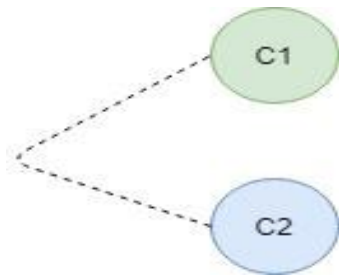
OR - If any cause from C1 and C2 is true, then effect E1 will be true.



NOT - If cause C1 is false, then effect E1 will be true.



Mutually Exclusive - When only one cause is true.



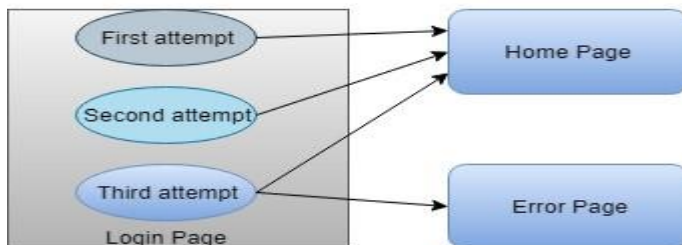
State Transition Technique

The general meaning of state transition is, different forms of the same situation, and according to the meaning, the state transition method does the same. It is used to capture the behavior of the software application when different input values are given to the same function.

We all use the ATMs, when we withdraw money from it, it displays account details at last. Now we again do another transaction, then it again displays account details, but the details displayed after the second transaction are different from the first transaction, but both details are displayed by using the same function of the ATM. So the same function was used here but each time the output was different, this is called state transition. In the case of testing of a software application, this method tests whether the function is following state transition specifications on entering different inputs.

This applies to those types of applications that provide the specific number of attempts to access the application such as the login function of an application which gets locked after the specified number of incorrect attempts. Let's see in detail, in the login function we use email and password, it gives a specific number of attempts to access the application, after crossing the maximum number of attempts it gets locked with an error message.

There is a login function of an application which provides a maximum three number of attempts, and after exceeding three attempts, it will be directed to an error page.



State transition table

STATE	LOGIN	VALIDATION	REDIRECTED
S1	First Attempt	Invalid	S2
S2	Second Attempt	Invalid	S3
S3	Third Attempt	Invalid	S5
S4	Home Page		
S5	Error Page		

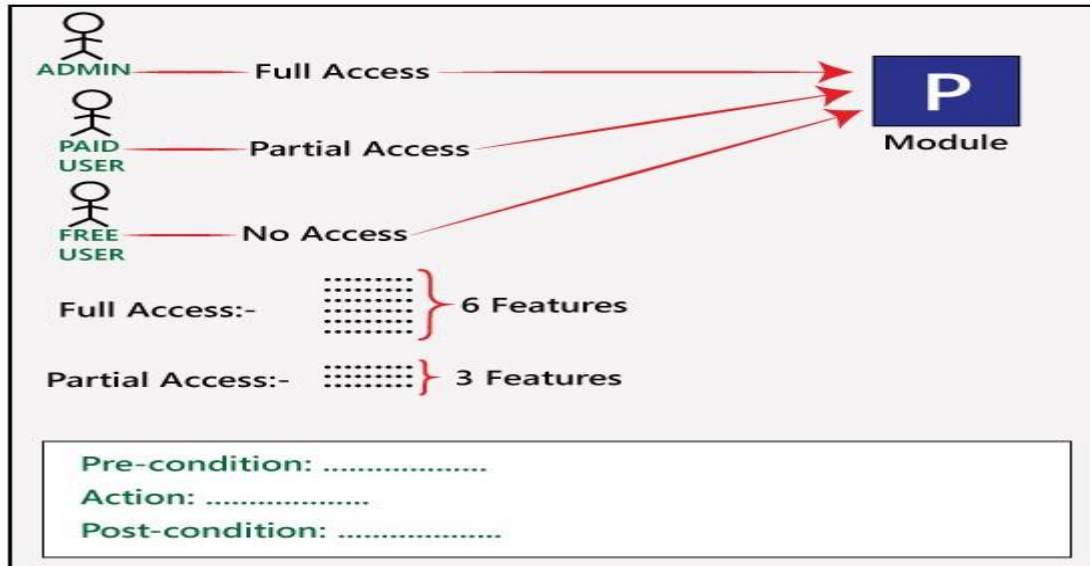
In the above state transition table, we see that state S1 denotes first login attempt. When the first attempt is invalid, the user will be directed to the second attempt (state S2). If the second attempt is also invalid, then the user will be directed to the third attempt (state S3). Now if the third and last attempt is invalid, then the user will be directed to the error page (state S5). But if the third attempt is valid, then it will be directed to the homepage (state S4).

Use Case Technique

The use case is functional testing of the black box testing used to identify the test cases from the beginning to the end of the system as per the usage of the system. By using this technique, the test team

creates a test scenario that can exercise the entire software based on the functionality of each function from start to end.

It is a graphic demonstration of business needs, which describe how the end-user will cooperate with the software or the application. The use cases provide us all the possible techniques of how the enduser uses the application as we can see in the below image, that how the **use case** will look like:



In the above image, we can see that a sample of a use case where we have a requirement related to the customer requirement specification (CRS).

For **module P** of the software, we have six different features a Tricky Program 23 - Main method

And here, **Admin** has access to all the **six features**, the **Paid user** has access to the **three features** and for the **Free user**, there is **no access** provided to any of the features.

Like for **Admin**, the different conditions would be as below:

Pre-condition→ Admin must be generated **Action**→

Login as Paid user

Post-condition→ 3 features must be present

And for **Free user**, the different condition would be as below:

Pre-condition→ free user must be generated

Action→ Login as a free user

Post-condition→ no features

➤ Control Structure Testing

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

The login form titled 'Login Here' has two input fields: 'Email' with a placeholder 'Enter Your Email' and 'Password' with a placeholder '10 characters'. Below the fields are two buttons: 'Back' and 'Login'.

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

1. Condition Testing

Condition testing is a test case design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect boolean operators, missing parenthesis in a boolean expression, error in relational operators, arithmetic expressions, and so on.

The common types of logical conditions that are tested using condition testing are-

1. A relation expression, like $E1 \text{ op } E2$ where 'E1' and 'E2' are arithmetic expressions and 'OP' is an operator.
2. A simple condition like any relational expression preceded by a NOT ($\sim$) operator. For example, $(\sim E1)$ where 'E1' is an arithmetic expression and 'a' denotes NOT operator.
3. A compound condition consists of two or more simple conditions, Boolean operator, and parenthesis. For example, $(E1 \ \& \ E2) \mid (E2 \ \& \ E3)$ where E1, E2, E3 denote arithmetic expression and '&' and '|' denote AND or OR operators.
4. A Boolean expression consists of operands and a Boolean operator like 'AND', OR, NOT. For example, 'A|B' is a Boolean expression where 'A' and 'B' denote operands and '|' denotes OR operator.

2. Data Flow Testing

The data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program.

The data flow test approach is depicted as follows suppose each statement in a program is assigned a unique statement number and that theme function cannot modify its parameters or global variables. For example, with S as its statement number.

$DEF(S) = \{X \mid \text{Statement } S \text{ has a definition of } X\}$

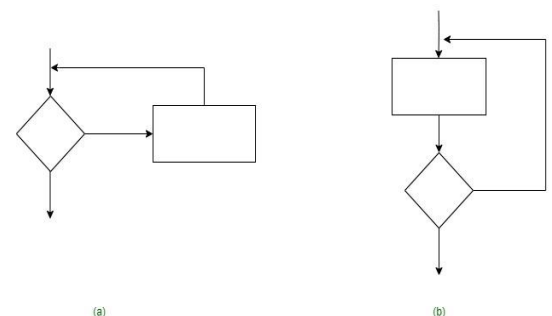
$USE(S) = \{X \mid \text{Statement } S \text{ has a use of } X\}$

If statement S is an if loop statement, then its DEF set is empty and its USE set depends on the state of statement S. The definition of the variable X at statement S is called the line of statement S' if the statement is any way from S to statement S' then there is no other definition of X.

A definition use (DU) chain of variable X has the form $[X, S, S']$, where S and S' denote statement numbers, X is in $DEF(S)$ and $USE(S')$, and the definition of X in statement S is line at statement S'.

A simple data flow test approach requires that each DU chain be covered at least once. This approach is known as the DU test approach. The DU testing does not ensure coverage of all branches of a program.

However, a branch is not guaranteed to be covered by DU testing only in rare cases such as then in which the other construct does not have any certainty of any variable in its later part and the other part is not



SIMPLE LOOPS

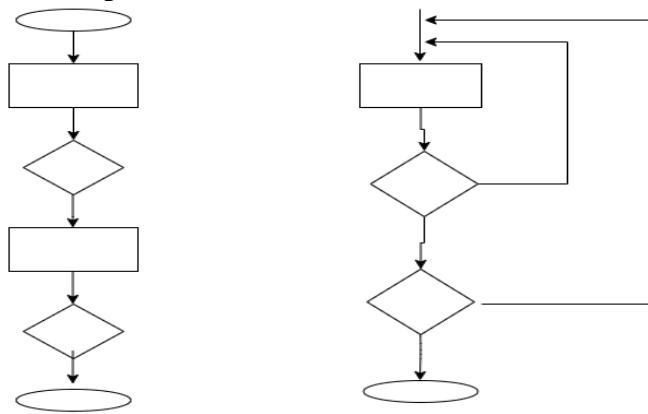
present. Data flow testing strategies are appropriate for choosing test paths of a program containing nested if and loop statements.

3. Loop Testing:

Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction.

Following are the types of loops.

1. **Simple Loop** – The following set of test can be applied to simple loops, where the maximum allowable number through the loop is n .
 1. Skip the entire loop.
 2. Traverse the loop only once.
 3. Traverse the loop two times.
 4. Make p passes through the loop where $p < n$.
 5. Traverse the loop $n-1$, n , $n+1$ time.
- 2 **Concatenated Loops** – **If loops are not dependent on each other, contact loops can be tested using the approach used in simple loops. if the loops are interdependent, the steps are followed in nested loops.**



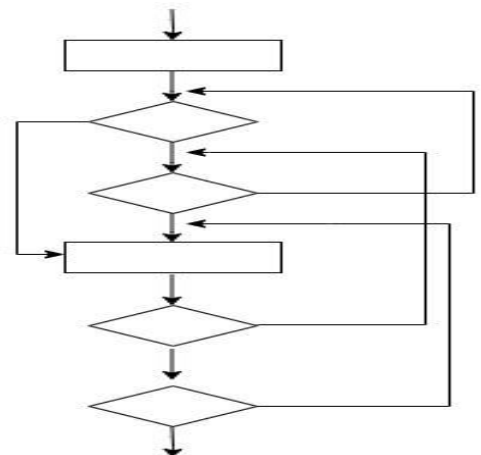
Concatenated Loops

3 Nested Loops – Loops within loops are called as nested loops. when testing nested loops, the number of tested increases as level nesting increases. The following steps for testing nested loops are as follows-

1. Start with inner loop. set all other loops to minimum values.
2. Conduct simple loop testing on inner loop.
3. Work outwards.
4. Continue until all loops tested.

4 Unstructured loops – This type of loops should be

redesigned, whenever possible, to reflect the use of unstructured the structured programming constructs.



Unstructured Loops

➤ Basis Path Testing in Software Testing

Basis Path Testing is a white-box testing technique based on the control structure of a program or a module. Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing. Therefore, by definition,

Basis path testing is a technique of selecting the paths in the control flow graph, that provide a basis set of execution paths through the program or module.

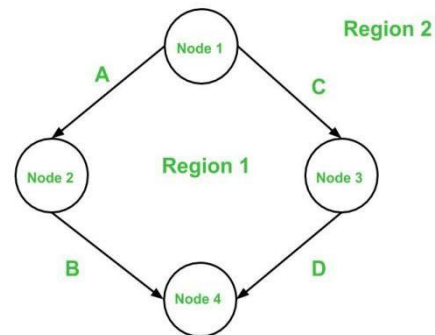
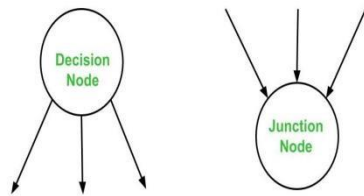
Since this testing is based on the control structure of the program, it requires complete knowledge of the program's structure. To design test cases using this technique, four steps are followed:

1. Construct the Control Flow Graph
2. Compute the Cyclomatic Complexity of the Graph
3. Identify the Independent Paths
4. Design Test cases from Independent Paths Let's understand each step one by one.

1. Control Flow Graph

A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module. A control flow graph (V, E) has V number of nodes/vertices and E number of edges in it. A control graph can also have :

- **Junction Node** – a node with more than one arrow entering it.
- **Decision Node** – a node with more than one arrow leaving it.
- **Region** – area bounded by edges and nodes (area outside the graph is also counted as a region.).

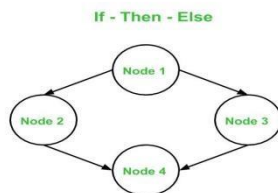


Below are the **notations** used while constructing a flow graph :

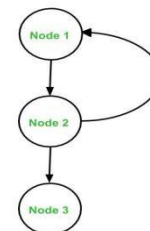
□ Sequential Statements

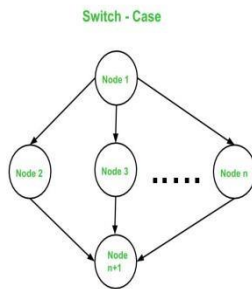


If – Then – Else –



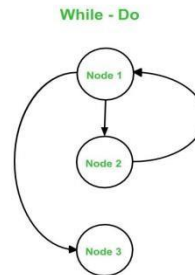
Do - While





Do – While

While – Do
Switch – Case



Cyclomatic

Complexity

The cyclomatic complexity $V(G)$ is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae:

1. Formula based on edges and nodes :

$$V(G) = e - n + 2 * P$$

Where, e is number of edges
 n is number of vertices

P is number of connected components. For example, consider first graph given above, where, $e = 4$, $n = 4$ and $p = 1$

So,

Cyclomatic complexity $V(G)$

$$= 4 - 4 + 2 * 1 =$$

2

2. Formula based on Decision Nodes :

$$V(G) = d + P \text{ where,}$$

d is number of decision nodes

P is number of connected nodes.

For example, consider first graph given above, where,

$d = 1$ and $p = 1$

So,

Cyclomatic Complexity $V(G)$

$$= 1 + 1$$

= 2

1. Formula based on Regions : $V(G) =$

number of regions in the graph

For example, consider first graph given above,

Cyclomatic complexity $V(G)$

$$= 1 \text{ (for Region 1)} + 1 \text{ (for Region 2)} =$$

2

Hence, using all the three above formulae, the cyclomatic complexity obtained remains same. All these three formulae can be used to compute and verify the cyclomatic complexity of the flow graph. **Note –**

1. For one function [e.g. Main () or Factorial ()], only one flow graph is constructed. If in a program, there are multiple functions, then a separate flow graph is constructed for each one of them. Also, in the cyclomatic complexity formula, the value of 'p' is set depending of the number of graphs present in total.
2. If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula $d = k - 1$
Here, k is number of arrows leaving the decision node.

Independent Paths

An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined. The cyclomatic complexity gives the number of independent paths present in a flow graph. This is because the cyclomatic complexity is used as an upper-bound for the number of tests that should be executed in order to make sure that all the statements in the program have been executed at least once.

Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity.

So, the independent paths in above first given graph :

- **Path 1:**
A -> B
- **Path 2:** C -> D

Note

Independent paths are not unique. In other words, if for a graph the cyclomatic complexity comes out be N, then there is a possibility of obtaining two different sets of paths which are independent in nature.

Design Test Cases:

Finally, after obtaining the independent paths, test cases can be designed where each test case represents one or more independent paths.

Advantages:

Basis Path Testing can be applicable in the following cases:

More Coverage

Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.

Maintenance Testing

When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing. **Unit Testing**

When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.

Integration Testing

When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.

Testing Effort

Since the basis path testing technique takes into account the complexity of the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

Object –Oriented Testing Methods

Test case design methods for OO software are still evolving. However, an overall approach to OO test case design has been defined by Berard :

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain:
 - a. A list of specified states for the object that is to be tested.
 - b. A list of messages and operations that will be exercised as a consequence of the test.
 - c. A list of exceptions that may occur as the object is tested.
 - d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
 - e. Supplementary information that will aid in understanding or implementing the test.

➤ The Test Case Design Implications of OO Concepts

The OO class is the target for test case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder notes, “Testing requires reporting on the concrete and abstract state of an object.” Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance also leads to additional challenges for the test case designer. We have already noted that each new context of usage requires retesting, even though reuse has been achieved. In addition, multiple inheritance complicates testing further by increasing the number of contexts for which testing is required. If subclasses instantiated from a super class are used within the same problem domain, it is likely that the set of test cases derived for the super class can be used when testing the subclass. However, if the subclass is used in an entirely different context, the super class test cases will have little applicability and a new set of tests must be designed.

Applicability of Conventional Test Case Design Methods

The white-box testing methods can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. Use-cases can provide useful input in the design of black-box and state-based tests.

Fault-Based Testing

The object of fault-based testing within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, the

preliminary planning required to perform fault based testing begins with the analysis model. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code. Consider a simple example.⁵ Software engineers often make errors at the boundaries of a problem. For example, when testing a SQRT operation that returns errors for negative numbers, we know to try the boundaries: a negative number close to zero and zero itself.

"Zero itself" checks whether the programmer made a mistake like

```
if (x > 0) calculate_the_square_root(); instead of the correct
if (x >= 0) calculate_the_square_root();
```

As another example, consider a Boolean expression:

```
if (a && !b || c)
```

Multicondition testing and related techniques probe for certain plausible faults in this expression, such as

&& should be || was left out where it was needed There should be parentheses around !b || c

For each plausible fault, we design test cases that will force the incorrect expression to fail. In the previous expression, (a=0, b=0, c=0) will make the expression as given evaluate false. If the && should have been ||, the code has done the wrong thing and might branch to the wrong path.

Of course, the effectiveness of these techniques depends on how testers perceive a "plausible fault." If real faults in an OO system are perceived to be "implausible," then this approach is really no better than any random testing technique. However, if the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find significant numbers of errors with relatively low expenditures of effort.

Integration testing looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.

Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned. Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior. It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is to determine whether errors exist in the calling code, not the called code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

The Impact of OO Programming on Testing

There are several ways object-oriented programming can have an impact on testing.

Depending on the approach to OOP,

- Some types of faults become less plausible (not worth testing for)
- Some types of faults become more plausible (worth testing now)
- Some new types of faults appear.

When an operation is invoked, it may be hard to tell exactly what code gets exercised. That is, the operation may belong to one of many classes. Also, it can be hard to determine the exact type or class of a parameter. When the code accesses it, it may get an unexpected value. The difference can be understood by considering a conventional function call:

x = func (y);

For conventional software, the tester need consider all behaviors attributed to func and nothing more. In an OO context, the tester must consider the behaviors of base::func(), of derived::func(), and so on. Each time func is invoked, the tester must consider the union of all distinct behaviors. This is easier if good OO design practices are followed and the difference between superclasses and subclasses (in C++ jargon, these are called base classes and derived classes) are limited. The testing approach for base and derived classes is essentially the same. Testing OO class operations is analogous to testing code that takes a function parameter and then invokes it. Inheritance is a convenient way of producing polymorphic operations. At the call site, what matters is not the inheritance, but the polymorphism. Inheritance does make the search for test requirements more straightforward.

By virtue of OO software architecture and construction, are some types of faults more plausible for an OO system and others less plausible? The answer is, "Yes." For example, because OO operations are generally smaller, more time tends to be spent on integration because there are more opportunities for integration faults. Therefore, integration faults become more plausible.

Test Cases and the Class Hierarchy

Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.

Consider the following situation. A class base contains operations inherited and redefined. A class derived redefines redefined to serve in a local context. There is little doubt the derived::redefined () has to be tested because it represents a new design and new code. But does derived::inherited() have to be retested?

If derived::inherited() calls redefined and the behavior of redefined has changed, derived::inherited () may mishandle the new behavior. Therefore, it needs new tests even though the design and code have not changed. It is important to note, however, that only a subset of all tests for derived::inherited() may have to be conducted. If part of the design and code for inherited does not depend on redefined (i.e., that does not call it nor call any code that indirectly calls it), that code need not be retested in the derived class.

Base::redefined () and **derived::redefined ()** are two different operations with different specifications and implementations. Each would have a set of test requirements derived from the specification and implementation. Those test requirements probe for plausible faults: integration faults, condition faults, boundary faults, and so forth. But the operations are likely to be similar. Their sets of test requirements will overlap. The better the OO design, the greater is the overlap. New tests need to be derived only for those derived::redefined () requirements that are not satisfied by the base::redefined () tests. To summarize, the base::redefined () tests are applied to objects of class derived. Test inputs may be appropriate for both base and derived classes, but the expected results may differ in the derived class.

Scenario-Based Test Design

Fault-based testing misses two main types of errors: **(1)** incorrect specifications and **(2)** interactions among subsystems. When errors associated with incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong thing or it might omit important functionality. But in either circumstance, quality (conformance to requirements)

suffers. Errors associated with subsystem interaction occur when the behavior of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.

Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests .

Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time). As an example, consider the design of scenario-based tests for a text editor. Use cases follow:

Use-Case: Fix the Final Draft

Background: It's not unusual to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens

1. Print the entire document
2. Move around in the document, changing certain pages
3. As each page is changed, it's printed
4. Sometimes a series of pages is printed.

This scenario describes two things: a test and specific user needs. The user needs are obvious: (1) a method for printing single pages and (2) a method for printing a range of pages. As far as testing goes, there is a need to test editing after printing (as well as the reverse). The tester hopes to discover that the printing function causes errors in the editing function; that is, that the two software functions are not properly independent.

Use-Case: Print a New Copy

Background: Someone asks the user for a fresh copy of the document. It must be printed. 1. Open the document

2. Print it .
3. Close the document.

Again, the testing approach is relatively obvious. Except that this document didn't appear out of nowhere. It was created in an earlier task. Does that task affect this one?

In many modern editors, documents remember how they were last printed. By default, they print the same way next time. After the Fix the Final Draft scenario, just selecting "Print" in the menu and clicking the "Print" button in the dialog box will cause the last corrected page to print again. So, according to the editor, the correct scenario should look like this:

Use-Case: Print a New Copy

1. Open the document
2. Select "Print" in the menu
3. Check if you're printing a page range; if so, click to print the entire document.
4. Click on the Print button
5. Close the document.

A test case designer might miss this dependency in test design, but it is likely that the problem would surface during testing. The tester would then have to contend with the probable response, "That's the way it's supposed to work!"

Testing Surface Structure and Deep Structure

Surface structure refers to the externally observable structure of an OO program. That is, the structure that is immediately obvious to an end-user. Rather than performing functions, the users

of many OO systems may be given objects to manipulate in some way. But whatever the interface, tests are still based on user tasks. Capturing these tasks involves understanding, watching, and talking with representative users (and as many non representative users as are worth considering).

There will surely be some difference in detail. For example, in a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. If no test scenarios existed to exercise a command, testing has likely overlooked some user tasks (or the interface has useless commands). In a object based interface, the tester might use the list of all objects as a testing checklist.

The best tests are derived when the designer looks at the system in a new or unconventional way. For example, if the system or product has a command-based interface, more thorough tests will be derived if the test case designer pretends that operations are independent of objects. Ask questions like, "Might the user want to use this operation—which applies only to the Scanner object—while working with the printer?" Whatever the interface style, test case design that exercises the surface structure should use both objects and operations as clues leading to overlooked tasks.

Deep structure refers to the internal technical details of an OO program. That is, the structure that is understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the system and object design of OO software.

➤ Testing Methods applicable at the Class level

We noted that software testing begins “in the small” and slowly progresses toward testing “in the large.” Testing in the small focuses on a single class and the methods that are encapsulated by the class. Random testing and partitioning are methods that can be used to exercise a class during OO testing.

Random Testing for OO Classes

To provide brief illustrations of these methods, consider a banking application in which an account class has the following operations: open, setup, deposit, withdraw, balance, summarize, creditLimit, and close. Each of these operations may be applied for account, but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of account includes the following operations:

open•setup•deposit•withdraw•close

This represents the minimum test sequence for account. However, a wide variety of other behaviors may occur within this sequence:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]n•withdraw•close A variety of different operation sequences can be generated randomly. For example:

Test case r1: **open•setup•deposit•deposit•balance•summarize•withdraw•close**

Test case r2: **open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close**

These and other random order tests are conducted to exercise different class instance life histories.

Partition Testing at the Class Level

Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for conventional software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?

State-based partitioning categorizes class operations based on their ability to change the state of the class. Again considering the account class, state operations include deposit and withdraw, whereas no state operations include balance, summarize, and creditLimit. Tests are designed in a way that exercises operations that change state and those that do not change state separately. Therefore,

Test case p1: **open•setup•deposit•deposit•withdraw•withdraw•close**

Test case p2: **open•setup•deposit•summarize•creditLimit•withdraw•close**

Test case p1 changes state, while test case p2 exercises operations that do not change state (other than those in the minimum test sequence).

Attribute-based partitioning categorizes class operations based on the attributes that they use. For the account class, the attributes balance and creditLimit can be used to define partitions. Operations are divided into three partitions: (1) operations that use creditLimit, (2) operations that modify creditLimit, and (3) operations that do not use or modify creditLimit. Test sequences are then designed for each partition.

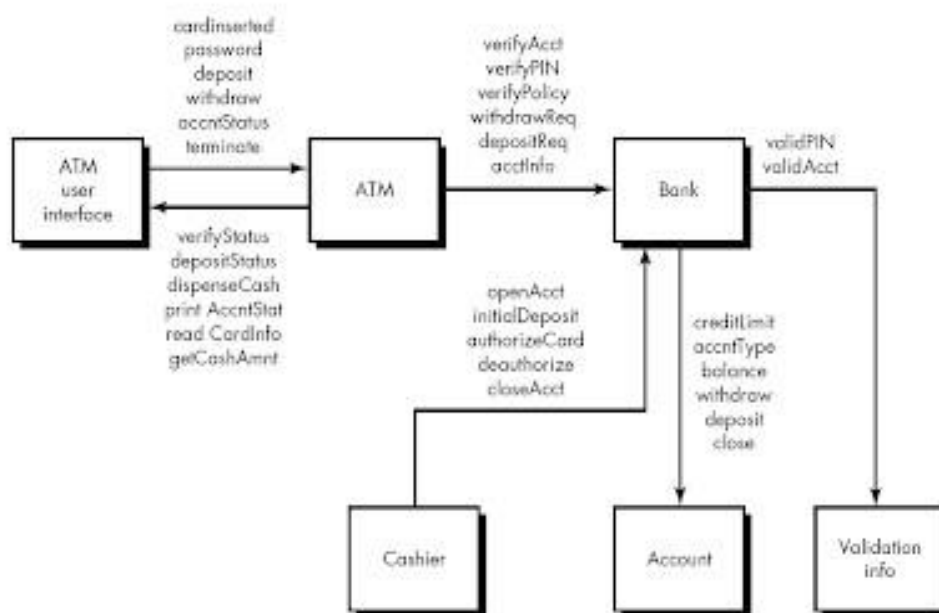
Category-based partitioning categorizes class operations based on the generic function that each performs. For example, operations in the account class can be categorized in initialization operations (open, setup), computational operations (deposit, withdraw), queries (balance, summarize, creditLimit) and termination operations (close).

➤ Interclass Test case Design

Test case design becomes more complicated as integration of the OO system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate “interclass test case generation”, we expand the banking example to include the classes and collaborations noted in Figure below. The direction of the arrows in the figure indicates the direction of messages and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Like the testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

Multiple Class Testing



Kirani and Tsai

suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

To illustrate, consider a sequence of operations for the bank class relative to an ATM class :

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]]depositReq|acctInfoREQ]n

A random test case for the bank class might be test case r3 =

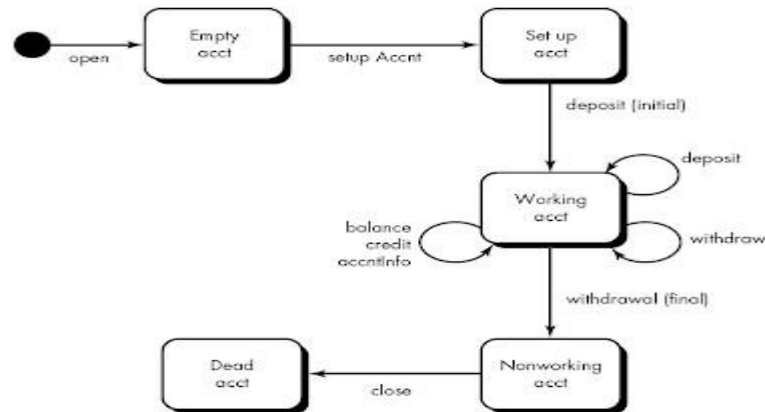
verifyAcct•verifyPIN•depositReq

In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case r3 is considered. Bank must collaborate with ValidationInfo to execute the verifyAcct and verifyPIN. Bank must collaborate with account to execute depositReq. Hence, a new test case that exercises these collaborations is The approach for multiple class partition testing is similar to the approach used for partition testing of individual classes. However, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes. An alternative approach partitions tests based on the interfaces to a particular class. Referring to figure, the bank class receives messages from the ATM and cashier classes. The methods within bank can therefore be tested by partitioning them into those that serve ATM and those that serve cashier. State-based partitioning can be used to refine the partitions further.

Tests Derived from Behavior Models

We already discussed the use of the state transition diagram as a model that represents the dynamic behavior of a class. The STD for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it). Figure below illustrates an STD for the account class. Referring to the figure, initial transitions move through the empty acct and setup acct states. The majority of all behavior for instances of the class occurs while in the working acct state. A final withdrawal and close cause the account class to make transitions to the nonworking acct and dead acct states, respectively.

test case r4 = **verifyAcctBank[validAcctValidationInfo]•verifyPINBank•[validPinValidationInfo]•depositReq• [depositaccount]**



The tests to be designed should achieve all state coverage. That is, the operation sequences should cause the account class to make transition through all allowable states:

test case s1: **open•setupAcct•deposit (initial)•withdraw (final)•close**

It should be noted that this sequence is identical to the minimum test sequence. Adding additional test sequences to the minimum sequence,

test case s2: **open•setupAcct•deposit(initial)•deposit•balance•credit•withdraw (final)•close**

test case s3: **open•setupAcct•deposit(initial)•deposit•withdraw•acctInfo•withdraw (final)•close**

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple STDs are used to track the behavioral flow of the system. The state model can be traversed in a “breadth-first” manner. In this context, breadth first implies that a test case exercise a single transition and that when a new transition is to be tested only previously tested transitions are used.

Consider the credit card object discussed earlier. The initial state of credit card is undefined (i.e., no credit card number has been provided). Upon reading the credit card during a sale, the object takes on a defined state; that is, the attributes card number and expiration date, along with bank specific identifiers are defined. The credit card is submitted when it is sent for authorization and it is approved when authorization is received. The transition of credit card from one state to another can be tested by deriving test cases that cause the transition to occur.

➤ Overview of Product Metrics

Software engineers use product metrics to help them assess the quality of the design and construction of the software product being built. Product metrics provide software engineers with a basis to conduct analysis, design, coding, and testing more objectively. Qualitative criteria for assessing software quality are not always sufficient by themselves. The process of using product metrics begins by deriving the software measures and metrics that are appropriate for the software representation under consideration. Then data are collected and metrics are computed. The metrics are computed and compared to pre-established guidelines and historical data. The results of these comparisons are used to guide modifications made to work products arising from analysis, design, coding, or testing.

➤ Software Quality

Software Quality Principles - a Qualitative View

Conformance to software requirements is the foundation from which quality is measured. Specified standards define a set of development criteria that guide the manner in which software is engineered. Software quality is suspect when a software product conforms to its explicitly stated requirements and fails to conform to the customer's implicit requirements (e.g., ease of use).

McCall's Quality Factors

The factors that affect software quality can be categorized in two broad groups.

Factors that can be directly measured (e.g., defects uncovered during testing)

Factors that can be measured only indirectly (e.g., usability or maintainability) The factors that affect software quality are shown below.

- Correctness: The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
 - Reliability: The extent to which a program can be expected to perform its intended function with required precision.
 - Efficiency: The amount of computing resources and code required by a program to perform its function.
 - Integrity: The extent to which access to software or data by unauthorized persons can be controlled.
 - Usability: The effort required to learn, operate, prepare input for, and interpret output of a program.
 - Maintainability: The effort required to locate and fix an error in a program.
 - Flexibility: The effort required to modify an operational program.
 - Testability: The effort required to test a program to ensure that it performs its intended function.
 - Portability: The effort required to transfer the program from one hardware and/or software system environment to another.
 - Reusability: The extent to which a program can be reused in other applications-related to the packaging and scope of the functions that the program performs.
 - Interoperability: The effort required to couple one system to another.
- ISO 9126 Quality Factors**
- Functionality
 - Reliability
 - Usability
 - Efficiency
 - Maintainability
 - Portability

➤ A Framework for Product Metrics

Benefits of Product Metrics

1. Assist in the evaluation of the analysis and evaluation model

2. Provide indication of procedural design complexity and source code complexity 3. Facilitate design of more effective testing

Measurement Principles

Measurement Process Activities

- Formulation - derivation of software measures and metrics appropriate for software representation being considered
- Collection - mechanism used to accumulate the data used to derive the software metrics
- Analysis - computation of metrics
- Interpretation - evaluation of metrics that results in gaining insight into quality of the work product
- Feedback - recommendations derived from interpretation of the metrics is transmitted to the software development team

Measurement Principles

- The objectives of measurement should be established before collecting any data.
- Each product metric should be defined in an unambiguous manner.
- Metrics should be derived based on a theory that is valid for the application domain.
- Metrics should be tailored to accommodate specific products and processes. **Metrics Characterization and Validation Principles**

- A metric should have desirable mathematical properties
 - The value of a metric should increase when positive software traits occur or decrease when undesirable software traits are encountered
 - Each metric should be validated empirically in several contexts before it is used to make decisions
- ### **Measurement Collection and Analysis Principles**

1. Automate data collection and analysis whenever possible
2. Use valid statistical techniques to establish relationships between internal product attributes and external quality characteristics
3. Establish interpretive guidelines and recommendations for each metric **Goal-Oriented Software Measurement (GQM)**

□ GQM emphasizes the need

1. establish explicit measurement goal specific to the process activity or product characteristic being assessed
2. define a set of questions that must be answered in order to achieve the goal
3. identify well-formulated metrics that help to answer these questions

• A goal definition template can be used to define each measurement goal **Attributes of Effective Software Metrics**

- Simple and computable
- Empirically and intuitively persuasive
- Consistent and objective
- Consistent in use of units and measures
- Programming language independent
- Provides an effective mechanism for quality feedback **Important Metrics Areas** □ Metrics for the Analysis Model

- Functionality delivered-provides an indirect measure of the functionality that is packaged within the software.
- System size –measures of the overall size of the system defined in terms of information available as part of the analysis model.
- Specification quality-provides an indication of the specificity and completeness of a requirements specification.

□ Metrics for the Design Model ○ Architecture metrics –provide an indication of the quality of the architectural design.

- Component-level metrics –measure the complexity of software components and other characteristics that have a bearing on quality.
- Specialized OO Design Metrics-measure characteristics of classes and their communication and collaboration characteristics.
- Metrics for Source Code
 - Halstead metrics –controversial but nonetheless fascinating, these metrics provide unique measures of a computer program.
 - Complexity metrics –measure the logical complexity of source code.
 - Length metrics-provide an indication of the size of the software.
- Metrics for Testing
 - Statement and branch coverage metrics –lead to the design of test cases that provide program coverage.
 - Defect-related metrics –focus on bugs found, rather than on the tests themselves.
 - Testing effectiveness –provide a real-time indication of the effectiveness of tests that have been conducted.
 - In-process metrics-process related metrics that can be determined as testing is conducted. ➤

Metrics for the Analysis Model

□ Function-based metrics

The function point metric (FP) can be used effectively as a means for measuring the functionality delivered by a system. Using historical data, the FP can then be used to ❖

Estimate the cost or effort required to design, code and test the software.

- ❖ Predict the number of errors that will be encountered during testing.
- ❖ Forecast the number of components and/or the number of projected source lines in the implemented system.

Functions points are derived from the following information domain values.

- Number of external inputs (EIs)
- Number of external outputs (EOs)
- Number of external inquiries (EQs)
- Number of internal logical files (ILFs)
- Number of external interface files (EIFs)

To compute function points (FP), the following relationship is used: $FP = \text{count total} * [0.65 + 0.01 * \sum (Fi)]$

Where count total is the sum of all FP entries and the Fi ($i=1$ to 14) are value adjustment factors (VAF) based on responses to the following questions.

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

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential)

□ Specification quality metrics (Davis)

Davis and his colleagues propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: specificity (lack of ambiguity),

completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability.

➤ Metrics for the Design Model

□ Architectural design metrics

- Structural complexity (based on module fanout): structural complexity of a module i is defined in the following manner $S(i)=f$
- Where f is the fan-out of module i . Fan-out is defined as the number f modules immediately subordinate to the module i .
- Data complexity (based on module interface inputs and outputs): provides an indication of the complexity in the internal interface for a module I and is defined as $D(i)=v(i)/[f(i)+1]$.
- Where $v(i)$ is the number of input and output variables that are passed to and from module i .
- System complexity (sum of structural and data complexity) $C(i)=S(i)+D(i)$
- Morphology (number of nodes and arcs in program graph): defined as $\text{size} = n+a$ where n is the number of nodes and a is the number of arcs.
- Design structure quality index (DSQI): the DSQI is computed in the following manner

$$DSQI = \sum w_i D_i$$

Where $i=1$ to 6 and W_i is the relative weighting of the importance of each of the intermediate values, and $\sum W_i=1$.

OO design metrics

- Size (defined in terms of four views: population, volume, length, functionality.

Population is measured by taking a static count of OO entities such as classes or operations)

- Complexity (how classes interrelate to one another)
- Coupling (physical connections between design elements of the OO design)
- Sufficiency (how well design components reflect all properties of the problem domain)
- Completeness (coverage of all parts of problem domain)
- Cohesion (manner in which all operations work together)
- Primitiveness (is the degree to which attributes and operations are atomic)
- Similarity (degree to which two or more classes are alike in terms of their structure, function, behavior or purpose.)
- Volatility (likelihood a design component will change)

➤ Metrics for Coding

Halstead proposed the first analytic laws for computer science by using a set of primitive measures, which can be derived once the design phase is complete and code is generated. These measures are listed below. n_1 = number of distinct operators in a program n_2 = number of distinct operands in a program N_1 = total number of operators N_2 = total number of operands.

By using these measures, Halstead developed an expression for overall program length, program volume, program difficulty, development effort, and so on.

Program length (N) can be calculated by using the following equation.

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

Program volume (V) can be calculated by using the following equation.

$$V = N \log_2 (n_1 + n_2).$$

Note that program volume depends on the programming language used and represents the volume of information (in bits) required to specify a program. Volume ratio (L) can be calculated by using the following equation.

$$L = \frac{\text{Volume of the most compact form of a program}}{\text{Volume of the actual program}}$$

Where, value of L must be less than 1. Volume ratio can also be calculated by using the following equation.

$$L = (2/n_1) * (n_2/N_2).$$

Program difficulty level (D) and effort (E) can be calculated by using the following equations.

$$D = (n_1/2) * (N_2/n_2).$$

$$E = D * V.$$

➤ Metrics for Testing

Majority of the metrics used for testing focus on testing process rather than the technical characteristics of test. Generally, testers use metrics for analysis, design, and coding to guide them in design and execution of test cases.

Function point can be effectively used to estimate testing effort. Various characteristics like errors discovered, number of test cases needed, testing effort, and so on can be determined by estimating the number of function points in the current project and comparing them with any previous project.

Metrics used for architectural design can be used to indicate how integration testing can be carried out. In addition, cyclomatic complexity can be used effectively as a metric in the basis-path testing to determine the number of test cases needed.

Halstead Metrics Applied to Testing:

Halstead measures can be used to derive metrics for testing effort. By using program volume (V) and program level (PL), Halstead effort (e) can be calculated by the following equations. $e = V / PL$

Where

$$PL = 1 / [(n_1/2) * (N_2/n_2)] \quad \dots (1)$$

For a particular module (z), the percentage of overall testing effort allocated can be calculated by the following equation.

$$\text{Percentage of testing effort (z)} = e(z) / \sum e(i)$$

Where, $e(z)$ is calculated for module z with the help of equation (1). Summation in the denominator is the sum of Halstead effort (e) in all the modules of the system.

Metrics for Object –Oriented Testing:

For developing metrics for object-oriented (OO) testing, different types of design metrics that have a direct impact on the testability of object-oriented system are considered. While developing metrics for OO testing, inheritance and encapsulation are also considered. A set of metrics proposed for OO testing is listed below.

- **Lack of cohesion in methods (LCOM):** This indicates the number of states to be tested. LCOM indicates the number of methods that access one or more same attributes. The value of LCOM is 0, if no methods access the same attributes. As the value of LCOM increases, more states need to be tested.
- **Percent public and protected (PAP):** This shows the number of class attributes, which are public or protected. Probability of adverse effects among classes increases with increase in value of PAP as public and protected attributes lead to potentially higher coupling.
- **Public access to data members (PAD):** This shows the number of classes that can access attributes of another class. Adverse effects among classes increase as the value of PAD increases.
- **Number of root classes (NOR):** This specifies the number of different class hierarchies, which are described in the design model. Testing effort increases with increase in NOR.
- **Fan-in (FIN):** This indicates multiple inheritances. If value of FIN is greater than 1, it indicates that the class inherits its attributes and operations from many root classes. Note that this situation (where $FIN > 1$) should be avoided.

➤ Overview Of Managing Software Projects

- Project management involves the planning, monitoring, and control of people, process, and events that occur during software development.
- Everyone manages, but the scope of each person's management activities varies according his or her role in the project.
- Software needs to be managed because it is a complex, long duration undertaking.
- Managers must focus on the four P's to be successful (people, product, process, and project).
- A project plan is a document that defines the four P's in such a way as to ensure a cost effective, high quality software product.
- The only way to be sure that a project plan worked correctly is by observing that a high quality product was delivered on time and under budget.

Management Spectrum

- People: The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization, work design, team/culture development.
- Product: Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered and technical and management constraints should be identified.
- Process: A software process provides the framework activities populated with tasks, milestones, work products, and quality assurance points.
- Project (planning, monitoring, controlling)

People

The software process is populated by stakeholders who can be categorized into one of five constituencies:

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

Team leadership model: Project management is a people-intensive activity.

- Motivation: The ability to encourage (by “push or pull”) technical people to produce to their best ability.
- Organization: The ability to mold existing processes 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.

Characteristics of effective project managers (problem solving, managerial identity, achievement, influence and team building)

Factors Affecting Team Organization

- Difficulty of problem to be solved
- Size of resulting program
- Team lifetime
- Degree to which problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of communication required for the project

Team Organizational Paradigms

- Closed paradigm (structures a team along a traditional hierarchy of authority. Top level problem solving and internal coordination managed by team leader, good for projects that repeat past efforts)
- Random paradigm (structures a team loosely and success depends on initiative of individual team members, paradigm excels when innovation and technical breakthroughs required)
- Open paradigm (work is performed collaboratively. Rotating task coordinators and group consensus, good for solving complex problems - not always efficient as other paradigms)
- Synchronous paradigm (relies on natural problem compartmentalization and team organized to require little active communication with each other)

Toxic Team Environment Characteristics

1. Frenzied work atmosphere where team members waste energy and lose focus on work objectives
2. High frustration and group friction caused by personal, business, or technological problems
3. Fragmented or poorly coordinated procedures or improperly chosen process model blocks accomplishments

4. Unclear role definition that results in lack of accountability or finger pointing
5. Repeated exposure to failure that leads to loss of confidence and lower morale **Agile Teams**
 - Teams have significant autonomy to make their own project management and technical decisions
 - Planning kept to minimum and is constrained only by business requirements and organizational standards
 - Team self-organizes as project proceeds to maximize contributions of each individual's talents □ May conduct daily (10 - 20 minute) meeting to synchronized and coordinate each day's work ○ What has been accomplished since the last meeting? ○ What needs to be accomplished by the next meeting? ○ How will each team member contribute to accomplishing what needs to be done?
 - What roadblocks exist that have to be overcome? **Coordination and Communication Issues**
 - Formal, impersonal approaches (e.g., documents, milestones, memos)
 - Formal interpersonal approaches (e.g., review meetings, inspections)
 - Informal interpersonal approaches (e.g., information meetings, problem solving)
 - Electronic communication (e.g., e-mail, bulletin boards, video conferencing)
 - Interpersonal networking (e.g., informal discussion with people other than project team members)

The Product

- Software scope (context, information objectives, function, and performance)
- Problem decomposition (partitioning or problem elaboration - focus is on functionality to be delivered and the process used to deliver it)

The Process

- Process model chosen must be appropriate for the: customers and developers, characteristics of the product, and project development environment
- Project planning begins with melding the product and the process
- Each function to be engineered must pass through the set of framework activities defined for a software organization
- Work tasks may vary but the common process framework (CPF) is invariant (project size does not change the CPF)
- The job of the software engineer is to estimate the resources required to move each function through the framework activities to produce each work product
- Project decomposition begins when the project manager tries to determine how to accomplish each CPF activity

The Project

Signs of Potential Project Failure

1. Developers do not understand customer's needs
2. Product scope poorly defined
3. Changes poorly managed
4. Chosen technology changes
5. Business needs change or ill-defined
6. Deadlines unrealistic
7. Users are resistant
8. Sponsorship lost or never obtained
9. Project team members lack appropriate skills
10. Managers and practitioners avoid best practices and lessons learned **Avoiding**

Project Failure

1. Start on the right foot: This is accomplished by working hard to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project.

2. Maintain momentum: Many projects get off to a good start and then slowly designate. 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.
3. Track progress: For software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved as part of a quality assurance activity.
4. Make smart decisions: In essence, the decisions of the project manager and the software team should be to “keep it simple”.
5. Conduct a postmortem analysis: 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.

W5HH Principle

- Why is the system being developed?: The answer to this question enables all parties to assess the validity of business reasons for the software work.
- What will be done?: The answer to this question establishes the task set that will be required for the project.
- When will it be accomplished?: The answer to this question helps the team to establish a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.
- Who is responsible for a function?: The role and responsibility of each member of the software team must be defined.
- Where they are organizationally located?: Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.
- How will the job be done technically and managerially?: Once the product scope is established, a management and technical strategy for the project must be defined.
- How much of each resource is needed?: The answer to this question is derived by developing estimates based on answers to earlier questions.

Critical Practices □

Formal risk management

- Empirical cost and schedule estimation
- Metric-based project management
- Earned value tracking
- Defect tracking against quality targets
- People-aware program management

➤ Overview of Metrics in process

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. There are four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

Measures and Metrics

- Measure - provides a quantitative indication of the size of some product or process attribute
- Measurement - is the act of obtaining a measure
- Metric - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute

Process Indicators

- Metrics should be collected so that process and product indicators can be ascertained
- Process indicators enable software project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality

Process Metrics

- Private process metrics (e.g., defect rates by individual or module) are only known to by the individual or team concerned.
- Public process metrics enable organizations to make strategic changes to improve the software process.
- Metrics should not be used to evaluate the performance of individuals.
- Statistical software process improvement helps and organization to discover where they are strong and where they are weak. **Statistical Process Control**

1. Errors are categorized by their origin
2. Record cost to correct each error and defect
3. Count number of errors and defects in each category
4. Overall cost of errors and defects computed for each category
5. Identify category with greatest cost to organization
6. Develop plans to eliminate the most costly class of errors and defects or at least reduce their frequency

Project Metrics

- A software team can use software project metrics to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

➤ **Software Measurement**

- Direct measures of a software engineering process include cost and effort.
- Direct measures of the product include lines of code (LOC), execution speed, memory size, defects reported over some time period.
- Indirect product measures examine the quality of the software product itself (e.g., functionality, complexity, efficiency, reliability, maintainability).

Size-Oriented Metrics

- Derived by normalizing (dividing) any direct measure (e.g., defects or human effort) associated with the product or project by LOC.
- Size-oriented metrics are widely used but their validity and applicability is a matter of some debate.

Function-Oriented Metrics

- Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.
- The relationship of LOC and function points depends on the language used to implement the software.

Object-Oriented Metrics

- Number of scenario scripts (NSS): A scenario script is a detailed sequence of steps that describes the interaction between the user and the application. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases.
- Number of key classes (NKC): Key classes are the “highly independent components”. Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software.
- Number of support classes: Support classes are required to implement the system but are not immediately related to the problem domain. (e.g., UI classes, database access classes, computations classes, etc.)
- Average number of support classes per key class: In general, key classes are known early in the project. Support classes are defined throughout if the average number of support classes per key class were known for a given problem domain, estimating would be much simplified.
- Number of subsystems (NSUB): A subsystem is an aggregation of classes that support a function that is visible to the end-user of a system.

Web Engineering Project Metrics

- Number of static Web pages (N_{sp}): Web pages with static content (i.e., the end-user has no control over the content displayed on the page) are the most common of all webapp features.
- Number of dynamic Web pages (N_{dp}): Web pages with dynamic content (i.e., end-user actions result in customized content displayed on the page) are essential in all e-commerce applications, search

engines, financial applications, and many other webapp categories

- Customization index: $C = N_{sp} / (N_{dp} + N_{sp})$
- Number of internal page links within the webapp: Internal page links are pointers that provide a hyperlink to some other webpage
- Number of persistent data objects: One or more persistent data objects (e.g., a database or data file) may be accessed by a webapp. As the number of persistent data objects grows, the complexity of the webapp also grows. □ Number of external systems interfaced: webapps must often interface with “backroom” business applications. As the requirement for interacting grows, system complexity and development effort also increase.
- Number of static content objects: Static content objects encompass static text-based, graphical, video, animation, and audio information that are incorporated within the webapp.
- Number of dynamic content objects: Dynamic content objects are generated based on end-user actions and encompasses internally generated text-based, graphical, video, animation, and audio information that are incorporated within the webapp.
- Number of executable functions: An executable function (e.g., a script or applet) provides some computational service to the end-user.

Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include
 - Correctness: Correctness is the degree of which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
 - Maintainability: Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple time-oriented metric is mean time to change.
 - Integrity: This attribute measures a system’s ability to withstand attacks(both accidental and intentional) to its security. To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability that an attack of a specific type will occur within a given time. Security is the probability that the attack of a specific type will be repelled. $\text{Integrity} = \sum [1 - (\text{threat} * (1 - \text{security}))]$
 - Usability: Usability is an attempt to quantify easy to learn, easy to use, productivity increase, user attitude.
- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied throughout the process framework
$$\text{DRE} = E / (E + D)$$

E = number of errors found before delivery of work product D = number of defects found after work product delivery

Integrating Metrics with Software Process □

Many software developers do not collect measures.

- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.
- Getting this historic project data is very difficult, if the previous developers did not collect data in an ongoing manner.

Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.
- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

The following are the set of easily collected metrics:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete.
- Effort (person-hours) to perform the evaluation
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel
- Effort (person-hours) required to make the change
- Time required (hours-days) to make the change
- Errors uncovered during work to make the change
- Defects uncovered after change is released to the customer base.

Establishing a Software Metrics Program

1. Identify business goal
2. Identify what you want to know
3. Identify subgoals
4. Identify subgoal entities and attributes
5. Formalize measurement goals
6. Identify quantifiable questions and indicators related to subgoals
7. Identify data elements needed to be collected to construct the indicators
8. Define measures to be used and create operational definitions for them
9. Identify actions needed to implement the measures
10. Prepare a plan to implement the measures

➤ Overview of Estimation

Software planning involves estimating how much time, effort, money, and resources will be required to build a specific software system. After the project scope is determined and the problem is decomposed into smaller problems, software managers use historical project data (as well as personal experience and intuition) to determine estimates for each. The final estimates are typically adjusted by taking project complexity and risk into account. The resulting work product is called a project management plan.

Estimation is the process of finding an estimate, or approximation, which is a value that can be used for some purpose even if input data may be incomplete, uncertain, or unstable.

Estimation determines how much money, effort, resources, and time it will take to build a specific system or product. Estimation is based on – □ Past Data/Past Experience

- Available Documents/Knowledge
- Assumptions
- Identified Risks

The four basic steps in Software Project Estimation are – □

Estimate the size of the development product.

- Estimate the effort in person-months or person-hours.
- Estimate the schedule in calendar months.
- Estimate the project cost in agreed currency.

Observations on Estimation

- Estimation need not be a one-time task in a project. It can take place during –
 - Acquiring a Project.
 - Planning the Project.
 - Execution of the Project as the need arises.

- Project scope must be understood before the estimation process begins. It will be helpful to have historical Project Data.
- Project metrics can provide a historical perspective and valuable input for generation of quantitative estimates.
- Planning requires technical managers and the software team to make an initial commitment as it leads to responsibility and accountability.
- Past experience can aid greatly.
- Use at least two estimation techniques to arrive at the estimates and reconcile the resulting values. Refer Decomposition Techniques in the next section to learn about reconciling estimates.
- Plans should be iterative and allow adjustments as time passes and more details are known. **Project**

Planning Objectives

- To provide a framework that enables a software manager to make a reasonable estimate of resources, cost, and schedule.
- 'Best case' and 'worst case' scenarios should be used to bound project outcomes.
- Estimates should be updated as the project progresses. **Estimation Reliability Factors**
- Project complexity
- Project size
- Degree of structural uncertainty (degree to which requirements have solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information processed) □
- Availability of historical information

Project Planning Process

1. Establish project scope
2. Determine feasibility
3. Analyze risks
4. Determine required resources
 - a. Determine required human resources
 - b. Define reusable software resources
 - c. Identify environmental resources
5. Estimate cost and effort
 - a. Decompose the problem
 - b. Develop two or more estimates
 - c. Reconcile the estimates
 6. Develop project schedule
 - a. Establish a meaningful task set
 - b. Define task network
 - c. Use scheduling tools to develop timeline chart
 - d. Define schedule tracking mechanisms

Software Scope

- Describes the data to be processed and produced, control parameters, function, performance, constraints, external interfaces, and reliability.
- Often functions described in the software scope statement are refined to allow for better estimates of cost and schedule.

Customer Communication and Scope

- Determine the customer's overall goals for the proposed system and any expected benefits.
- Determine the customer's perceptions concerning the nature if a good solution to the problem. □ Evaluate the effectiveness of the customer meeting. **Feasibility**

- Technical feasibility is not a good enough reason to build a product.
- The product must meet the customer's needs and not be available as an off-the-shelf purchase.

Estimation of Resources

- Human Resources (number of people required and skills needed to complete the development project) □ Reusable Software Resources
 - Off-the-shelf components: Existing software can be acquired from a third party or has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.
 - Full-experience components: Existing specifications, design, code, or test data developed for past projects are similar to the software to be built for the current project. Therefore, modifications required for full-experience components will be relatively low-risk.
 - Partial-experience components: Existing specifications, design, code, or test data developed for past projects are related to the software to be built for the current project but will require substantial modification. Therefore, modifications required for partial-experience components have a fair degree of risk.
 - New components: Software components must be built by the software team specifically for the needs of the current project.
- Environment Resources (hardware and software required to be accessible by software team during the development process)

➤ Empirical Estimation Models

COCOMO Model

Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981. COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software. The necessary steps in this model are:

1. Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).
2. Determine a set of 15 multiplying factors from various attributes of the project.
3. Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.

The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size. To determine the initial effort E_i in person-months the equation used is of the type is shown below

$$E_i = a * (KDLOC)^b$$

The value of the constant a and b depends on the project type. In

COCOMO, projects are categorized into three types:

1. Organic
2. Semidetached
3. Embedded

1. Organic: A development project can be treated of the 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 methods of projects. Examples of this type of projects are simple business systems, simple inventory management systems, and data processing systems.

2. Semidetached: A development project can be treated with semidetached type if the development consists of a mixture of experienced and inexperienced staff. Team members may have finite experience in related systems but may be unfamiliar with some aspects of the order being developed. Example of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.

3. Embedded: A development project is treated to be of an embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational method exist. For Example: ATM, Air Traffic control.

For three product categories, Bohem provides a different set of expression to predict effort (in a unit of person month) and development time from the size of estimation in KLOC (Kilo Line of code) efforts estimation takes into account the productivity loss due to holidays, weekly off, coffee breaks, etc.

According to Boehm, software cost estimation should be done through three stages:

1. Basic Model

2. Intermediate Model

3. Detailed Model

1. Basic COCOMO Model: The basic COCOMO model provide an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model: $\text{Effort} = a_1 * (\text{KLOC})^{a_2}$ PM

$T_{dev} = b_1 * (\text{efforts})^{b_2}$ Months

Where

KLOC is the estimated size of the software product indicate in Kilo Lines of Code, a_1 ,

a_2 , b_1 , b_2 are constants for each group of software products,

T_{dev} 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).

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(KLOC)^{1.05} PM **Semi-detached: Effort = 3.0(KLOC)^{1.12} PM** **Embedded: Effort = 3.6(KLOC)^{1.20} PM**

Estimation of development time

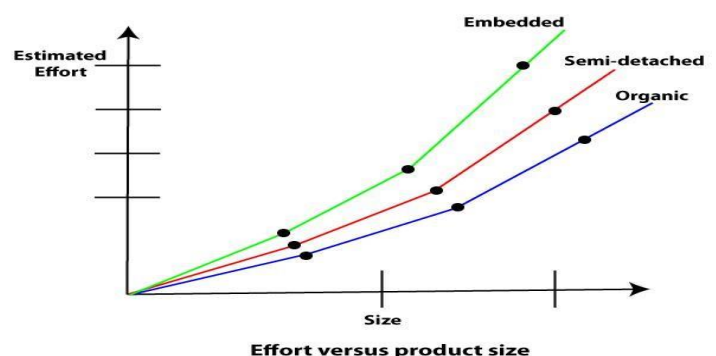
For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic: $T_{dev} = 2.5(\text{Effort})^{0.38}$ Months **Semi-detached: $T_{dev} = 2.5(\text{Effort})^{0.35}$ Months**

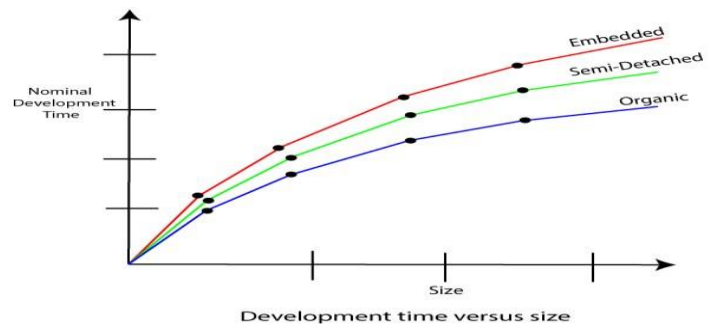
Embedded: $T_{dev} = 2.5(\text{Effort})^{0.32}$ Months Some insight into the basic COCOMO model can be obtained by plotting the estimated

characteristics for different software sizes. Fig shows a plot of estimated effort versus product size. From fig, we can observe that the effort is somewhat superlinear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

The development time versus the



product size in KLOC is plotted in fig. From fig it can be observed that the development time is a sub linear 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, it can be observed that the development time is roughly the same for all 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.

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 a 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.

Example1: Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

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

$$\text{Tdev} = b_1 * (\text{efforts})^{b_2} \text{ Months}$$

$$\text{Estimated Size of project} = 400 \text{ KLOC}$$

(i) Organic Mode

$$E = 2.4 * (400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5 * (1295.31)^{0.38} = 38.07 \text{ PM}$$

(ii) Semidetached Mode

$$E = 3.0 * (400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5 * (2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded Mode

$$E = 3.6 * (400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5 * (4772.8)^{0.32} = 38 \text{ PM}$$

Example2: A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the Effort, development time, average staff size, and productivity of the project.

Solution: The semidetached mode is the most appropriate mode, keeping in view the size, schedule and experience of development time. Hence $E = 3.0(200)^{1.12} = 1133.12 \text{ PM}$

$$D = 2.5(1133.12)^{0.35} = 29.3 \text{ PM}$$

$$\text{Average Staff Size (SS)} = \frac{E}{D} \text{ Persons}$$

$$= \frac{1133.12}{29.3} = 38.67 \text{ Persons}$$

$$\text{Productivity} = \frac{\text{KLOC}}{E} = \frac{200}{1133.12} = 0.1765 \text{ KLOC/PM}$$

$$P = 176 \text{ LOC/PM}$$

2. Intermediate Model: The basic Cocomo model considers that the effort is only a function of the number of lines of code and some constants calculated according to the various software systems. The intermediate COCOMO model recognizes these facts and refines the initial estimates obtained through the basic COCOMO model by using a set of 15 cost drivers based on various attributes of software engineering.

Classification of Cost Drivers and their attributes:

(i) Product attributes -

Required software reliability extent

Size of the application database

The complexity of the product

Hardware attributes -

Run-time performance constraints

Memory constraints

The volatility of the virtual machine environment

Required turnabout time

Personnel attributes -

Analyst capability

Software engineering capability

Applications experience Virtual machine experience

Programming language experience

Project attributes -

Use of software tools

Application of software engineering methods

Required development schedule

Intermediate COCOMO equation:

$$E = a_i (\text{KLOC})^{b_i} \text{EAF}$$

$$D = c_i (E)^{d_i}$$

Coefficients for intermediate COCOMO

Project	a_i	b_i	c_i	d_i
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

3. Detailed COCOMO Model: Detailed COCOMO incorporates all qualities of the standard version with an assessment of the cost driver's effect on each method of the software engineering process. The detailed model uses various effort multipliers for each cost driver property. In detailed cocomo, the whole software is differentiated into multiple modules, and then we apply COCOMO in various modules to estimate effort and then sum the effort. The Six phases of detailed COCOMO are:

- 1.Planning and requirements
- 2.System structure
- 3.Complete structure
- 4.Module code and test
- 5.Integration and test
- 6.Cost Constructive model

The effort is determined as a function of program estimate, and a set of cost drivers are given according to every phase of the software lifecycle.

➤ **Decomposition Techniques**

- Software sizing: The “size” of software to be built can be estimated using a direct measure, LOC (lines of code), or on indirect measure, FP (function points). Four different approaches to the sizing problem are:
 - “Fuzzy logic” sizing: To apply this approach, the planner must identify the type of application, establish its magnitude on a quantitative scale, and then refine the magnitude within the original range.
 - Function point sizing: The planner develops estimates of the information domain characteristics.
 - Standard component sizing: Software is composed of number of different “standard components” that are generic to a particular application area. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component.
 - Change sizing: This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, and deleting code) of modifications that must be accomplished.
- Problem-based estimation (using LOC decomposition focuses on software functions, using FP decomposition focuses on information domain characteristics)
- Process-based estimation (decomposition based on tasks required to complete the software process framework. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated)
- Use-case estimation (promising, but controversial due to lack of standardization of use-cases) **Causes of Estimation Reconciliation Problems**
 - Project scope is not adequately understood or misinterpreted by planner
 - Productivity data used for problem-based estimation techniques is inappropriate or obsolete for the application

➤ **Empirical Estimation Models**

- Typically derived from regression analysis of historical software project data with estimated person-months as the dependent variable and KLOC, FP, or object points as independent variables. □ Constructive COst MOdel (COCOMO) is an example of a static estimation model.
 - COCOMO II is a hierarchy of estimation models that address the following areas
 - Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation technology maturity are paramount.
 - Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.
 - Post-architecture stage model. Used during the construction of the software.
- The COCOMO II model requires sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

The COCOMO II application composition model uses object points-an indirect software measure that is computed using counts of the number of (1) screens (2) reports and (3) components likely to be required to build the application. The object point count is determined by multiplying the original number of object instances by the weighting factor. When component-based development or general

software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$\text{NOP} = (\text{object points}) * [(100 - \% \text{reuse}) / 100]$$
 Where NOP is defined as new object points.

$$\text{PROD (productivity rate)} = \text{NOP} / \text{person-month}$$
 Estimated effort = NOP / PROD

Estimation for Agile Development

1. Each user scenario is considered separately
2. The scenario is decomposed into a set of engineering tasks
3. Each task is estimated separately
 - a. May use historical data, empirical model, or experience
4. Scenario volume can be estimated (LOC, FP, use-case count, etc.) Total scenario estimate computed a. Sum estimates for each task
- b. Translate volume estimate to effort using historical data
5. The effort estimates for all scenarios in the increment are summed to get an increment estimate

➤ Estimation for Object-Oriented Projects

- Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications
- Using object-oriented analysis modeling, develop use-cases and determine a count. Recognize that the number of use-cases may change as the project progressed.
- From the analysis model, determine the number of key classes
- Categorize the type of interface for the application, and develop a multiplier for support classes:

Interface	Multiplier
No GUI	2.0
Text-based user interface	2.25
GUI	2.5
Complex GUI	3.0

Multiply the number of key classes by the multiplier to obtain an estimate for the number of support classes.

- Multiply the total number of classes by the average number of work-units per class.
- Cross-check the class-based estimate by multiplying the average number of work-units per usecase.

➤ Other estimation Techniques

Software team encounters an extremely short project duration that is likely to have a continuing stream of changes, project planning in general and estimation in particular should be abbreviated.

Estimation for Agile Development:

Estimation for agile projects uses a decomposition approach that encompasses the following steps:

- Each user scenario is considered separately for estimation purposes.
- The scenario is decomposed into the set of functions and the software engineering tasks that will be required to develop them.
- Each task is estimated separately. Note: estimation can be based on historical data, an empirical model, or “experience”.
- Alternatively the “volume” (size) of the scenario can be estimated in LOC, FP, or some other volume-oriented measure (e.g., object points).
- Estimates for each task are summed to create an estimate for the scenario.
- Alternatively, the volume estimate for the scenario is translated into effort using historical data.

- The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

This estimation approach serves two purposes:

- (1) To ensure that the number of scenarios to be included in the increment conforms to the available resources and
- (2) to establish a basis for allocating effort as the increment is developed.

Estimation for Web Engineering Projects:

The following information domain values when adapting function points for WebApp estimation:

- Inputs are each input screen or form (for example, CGI or java), each maintenance screen, and if u use a tab notebook metaphor anywhere, each tab).
- Outputs are each static Web page, each dynamic Webpage script (for example, ASP, ISAPI, or other DHTML script), and each report (whether Web based or administrative in nature).
- Tables are each logical table in the database plus, if u are using XML to store data in a file, each XML object (or collection of XML attributes).
- Interfaces retain their definition as logical files (for example, unique record formats) into our out-of-the-system boundaries.
- Queries are each externally published or use a message-oriented interface. A typical example is DCOM or COM external references.

Function points are a reasonable indicator of volume for a WebApp.

➤ Overview of Project Scheduling

The chapter describes the process of building and monitoring schedules for software development projects. To build complex software systems, many engineering tasks need to occur in parallel with one another to complete the project on time. The output from one task often determines when another may begin. It is difficult to ensure that a team is working on the most appropriate tasks without building a detailed schedule and sticking to it.

Root Causes for Late Software

- Unrealistic deadline established outside the team
- Changing customer requirements not reflected in schedule changes
- Underestimating the resources required to complete the project
- Risks that were not considered when project began
- Technical difficulties that have not been predicted in advance
- Human difficulties that have not been predicted in advance
- Miscommunication among project staff resulting in delays
- Failure by project management to recognize project falling behind schedule and failure to take corrective action to correct problems

How to Deal With Unrealistic Schedule Demands

1. Perform a detailed project estimate for project effort and duration using historical data.
2. Use an incremental process model that will deliver critical functionality imposed by deadline, but delay other requested functionality.
3. Meet with the customer and explain why the deadline is unrealistic using your estimates based on prior team performance.
4. Offer an incremental development and delivery strategy as an alternative to increasing resources or allowing the schedule to slip beyond the deadline.

Project Scheduling Perspectives

- One view is that the end-date for the software release is set externally and that the software organization is constrained to distribute effort in the prescribed time frame.
- Another view is that the rough chronological bounds have been discussed by the developers and customers, but the end-date is best set by the developer after carefully considering how to best use the resources needed to meet the customer's needs.

Software Project Scheduling Principles

- Compartmentalization - the product and process must be decomposed into a manageable number of activities and tasks
- Interdependency - tasks that can be completed in parallel must be separated from those that must be completed serially
- Time allocation - every task has start and completion dates that take the task interdependencies into account
- Effort validation - project manager must ensure that on any given day there are enough staff members assigned to complete the tasks within the time estimated in the project plan.
- Defined Responsibilities - every scheduled task needs to be assigned to a specific team member
- Defined outcomes - every task in the schedule needs to have a defined outcome (usually a work product or deliverable)
- Defined milestones - a milestone is accomplished when one or more work products from an engineering task have passed quality review

Relationship Between People and Effort

- Adding people to a project after it is behind schedule often causes the schedule to slip further
- The relationship between the number of people on a project and overall productivity is not linear (e.g., 3 people do not produce 3 times the work of 1 person, if the people have to work in cooperation with one another)
- The main reasons for using more than 1 person on a project are to get the job done more rapidly and to improve software quality.

Project Effort Distribution □ The

40-20-40 rule (a rule of thumb): ○ 40%

front-end analysis and design

○ 20% coding ○ 40%

back-end testing

□ Generally accepted guidelines are: ○

02-03 % planning ○ 10-25 %

requirements analysis

○ 20-25 % design ○ 15-20 %

coding

○ 30-40 % testing and debugging

Software Project Types

1. Concept development - initiated to explore new business concept or new application of technology
 2. New application development - new product requested by customer
 3. Application enhancement - major modifications to function, performance, or interfaces (observable to user)
 4. Application maintenance - correcting, adapting, or extending existing software (not immediately obvious to user)
 5. Reengineering - rebuilding all (or part) of a legacy system
- Factors Affecting Task Set** □ Size of project
- Number of potential users
 - Mission criticality
 - Application longevity
 - Requirement stability
 - Ease of customer/developer communication

- Maturity of applicable technology
- Performance constraints
- Embedded/non-embedded characteristics
- Project staffing

□ Reengineering factors **Concept**

Development Tasks

- Concept scoping - determine overall project scope
- Preliminary concept planning - establishes development team's ability to undertake the proposed work
- Technology risk assessment - evaluates the risk associated with the technology implied by the software scope
- Proof of concept - demonstrates the feasibility of the technology in the software context
- Concept implementation - concept represented in a form that can be used to sell it to the customer □
- Customer reaction to concept - solicits feedback on new technology from customer

Scheduling

- Task networks (activity networks) are graphic representations that can be of the task interdependencies and can help define a rough schedule for particular project
- Scheduling tools should be used to schedule any non-trivial project.
- PERT (program evaluation and review technique) and CPM (critical path method) are quantitative techniques that allow software planners to identify the chain of dependent tasks in the project work breakdown structure that determine the project duration time.
- Timeline (Gantt) charts enable software planners to determine what tasks will be need to be conducted at a given point in time (based on estimates for effort, start time, and duration for each task).
- The best indicator of progress is the completion and successful review of a defined software work product.
- Time-boxing is the practice of deciding a priori the fixed amount of time that can be spent on each task. When the task's time limit is exceeded, development moves on to the next task (with the hope that a majority of the critical work was completed before time ran out). **Tracking Project Schedules**

- Periodic status meetings
- Evaluation of results of all work product reviews
- Comparing actual milestone completion dates to scheduled dates
- Comparing actual project task start-dates to scheduled start-dates
- Informal meeting with practitioners to have them assess subjectively progress to date and future problems
- Use earned value analysis to assess progress quantitatively **Tracking Increment Progress for OO Projects** □ Technical milestone: OO analysis complete ○ All hierarchy classes defined and reviewed

- Class attributes and operations are defined and reviewed
- Class relationships defined and reviewed ○ Behavioral model defined and reviewed ○ Reusable classed identified

reviewed

- Technical milestone: OO design complete ○ Subsystems defined and

- Classes allocated to subsystems and reviewed ○ Task allocation has been established and reviewed
- Responsibilities and collaborations have been identified ○ Attributes and operations have been designed and reviewed ○ Communication model has been created and reviewed □ Technical milestone: OO programming complete ○ Each new design model class has been implemented ○ Classes extracted from the reuse library have been

implemented ○ Prototype or increment has been built □ Technical milestone: OO testing complete ○ The correctness and completeness of the OOA and OOD models has been reviewed ○ Class-responsibility-collaboration network has been developed and reviewed ○ Test cases are designed and class-level tests have been conducted for each class ○ Test cases are designed, cluster testing is completed, and classes have been integrated ○ System level tests are complete **Earned Value**

Analysis

- Earned value is a quantitative measure given to each task as a percent of project completed so far.
- 1. The total hours to complete each project task are estimated
- 2. The effort to complete the project is computed by summing the effort to complete each task 3. Each task is given an earned value based on its estimated percentage contribution to the total.

➤ Overview of Risk Management

Risks are potential problems that might affect the successful completion of a software project. Risks involve uncertainty and potential losses. Risk analysis and management is intended to help a software team understand and manage uncertainty during the development process. The important thing is to remember that things can go wrong and to make plans to minimize their impact when they do. The work product is called a Risk Mitigation, Monitoring, and Management Plan (RMMM).

Risk Strategies

- Reactive strategies - very common, also known as fire fighting, project team sets resources aside to deal with problems and does nothing until a risk becomes a problem
- Proactive strategies - risk management begins long before technical work starts, risks are identified and prioritized by importance, then team builds a plan to avoid risks if they can or minimize them if the risks turn into problems

Software Risks

- Project risks - threaten the project plan
- Technical risks - threaten product quality and the timeliness of the schedule
- Business risks - threaten the viability of the software to be built (market risks, strategic risks, management risks, budget risks). Known risks - predictable from careful evaluation of current project plan and those extrapolated from past project experience
- Unknown risks - some problems simply occur without warning

Risk Identification

- Product-specific risks - the project plan and software statement of scope are examined to identify any special characteristics of the product that may threaten the project plan
- Generic risks - are potential threats to every software product (product size, business impact, customer characteristics, process definition, development environment, technology to be built, staff size and experience)

Risk Checklist Items

- Product size
- Business impact

- Customer characteristics
- Process definition
- Development environment
- Technology to be built
- Staff size and experience

Project-Related Risk Assessment Questions

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project?
3. Are requirements fully understood by developers and customers?
4. Were customers fully involved in requirements definition?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does software team have the right skill set?
8. Are project requirements stable?
9. Does the project team have experience with technology to be implemented?
10. Is the number of people on project team adequate to do the job?
11. Do all stakeholders agree on the importance of the project the requirements for the systems being built?

Risk Impact

- Risk components - performance, cost, support, schedule
- Risk impact - negligible, marginal, critical, catastrophic
- The risk drivers affecting each risk component are classified according to their impact category and the potential consequences of each undetected software fault or unachieved project outcome are described

Risk Projection (Estimation)

1. Establish a scale that reflects the perceived likelihood of each risk
2. Delineate the consequences of the risk
3. Estimate the impact of the risk on the project and product
4. Note the overall accuracy of the risk projection to avoid misunderstandings

Risk Table Construction

- List all risks in the first column of the table
- Classify each risk and enter the category label in column two
- Determine a probability for each risk and enter it into column three
- Enter the severity of each risk (negligible, marginal, critical, catastrophic) in column four
- Sort the table by probability and impact value
- Determine the criteria for deciding where the sorted table will be divided into the first priority concerns and the second priority concerns
- First priority concerns must be managed (a fifth column can be added to contain a pointer into the RMMM)

Assessing Risk Impact

- Factors affecting risk consequences - nature (types of problems arising), scope (combines severity with extent of project affected), timing (when and how long impact is felt)
- If costs are associated with each risk table entry a risk exposure metric can be computed ($RE = \text{Probability} * \text{Cost}$) and added to the risk table.

Risk Assessment

1. Define referent levels for each project risk that can cause project termination (performance degradation, cost overrun, support difficulty, schedule slippage).
2. Attempt to develop a relationship between each risk triple (risk, probability, impact) and each of the reference levels.
3. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.

4. Try to predict how combinations of risks will affect a referent level.

Risk Refinement

- Process of restating the risks as a set of more detailed risks that will be easier to mitigate, monitor, and manage.
- CTC (condition-transition-consequence) format may be a good representation for the detailed risks (e.g., given that <condition> then there is a concern that (possibly) <consequence>).
This general condition can be refined in the following manner:
 - Sub condition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
 - Sub condition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
 - Sub condition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

Risk Mitigation, Monitoring, and Management

- Risk mitigation (proactive planning for risk avoidance)
- Risk monitoring (assessing whether predicted risks occur or not, ensuring risk aversion steps are being properly applied, collect information for future risk analysis, attempt to determine which risks caused which problems)
- Risk management and contingency planning (actions to be taken in the event that mitigation steps have failed and the risk has become a live problem)

Safety Risks and Hazards

- Risks are also associated with software failures that occur in the field after the development project has ended.
 - Computers control many mission critical applications in modern times (weapons systems, flight control, industrial processes, etc.).
 - Software safety and hazard analysis are quality assurance activities that are of particular concern for these types of applications
- ### **Risk Information Sheets**
- Alternative to RMMM (Risk Mitigation, Monitoring and Management plan) in which each risk is documented individually.
 - Often risk information sheets (RIS) are maintained using a database system.
 - RIS components - risk id, date, probability, impact, description, refinement, mitigation/monitoring, management/contingency/trigger, status, originator, assigned staff member.

➤ Overview of Reengineering

This chapter defines reengineering as the process of rebuilding legacy software products. The rebuilt software products often have increased functionality, better performance, greater reliability, and are easier to maintain than their predecessors. Business process reengineering (BPR) defines business goals, evaluates existing business processes, and creates revised business processes that better meet current goals. Software reengineering involves inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. Many reengineering work products are the same as those generated for any software engineering process (analysis models, design models, test procedures). The final product for any reengineering process is a reengineered business process and/or the reengineered software to support it. The same SQA practices are applied to software reengineering as they would to any other software development process. Testing is used to uncover errors in content, functionality, and interoperability.

Business Process Reengineering Principles

- Organize around outcomes, not tasks.
- Have those people who use the output of a process, perform the process.
- Incorporate information processing work into the real work that produces the raw information.
- Treat geographically dispersed resources as though they were centralized.
- Link parallel activities instead of integrating their results.
- Put the decision point where the work is performed and build control into the process.
- Capture the data once, at its source. **Business Process Reengineering Model**
- Business definition - business goals are identified in the context of four key drivers (cost reduction, time reduction, quality improvement, empowerment)
- Process identification - processes critical to achieving business goals are identified and prioritized
- Process evaluation - existing processes are analyzed and measured, costs and time consumed by processes are noted, quality/performance problems are isolated
- Process specification and design - use-cases are prepared for each process to be redesigned, these usecase scenarios deliver some outcome to a customer, new tasks are designed for each process
- Prototyping - used to test processes before integrating them into the business
- Refinement and instantiation - based on feedback from the prototype, business processes are refined and then instantiated within a business system **Software Reengineering**

Software Maintenance

- Corrective maintenance (fixing errors)
- Adaptive maintenance (accommodating changes in the environment or user needs)
- Perfective maintenance (reengineering the application to improve performance or make the software product easier to maintain)
- Preventative maintenance (modifying software to avoid anticipated future problems) **Software Reengineering Process Model**
- Inventory analysis - sorting active software applications by business criticality, longevity, current maintainability, and other local criteria helps to identify reengineering candidates
- Document restructuring - need to decide to live with weak documentation, update poor documents if they are used, or fully rewrite the documentation for critical systems focusing on the "essential minimum"
- Reverse engineering - process of design recovery - analyzing a program in an effort to create a representation of the program at some abstraction level higher than source code
- Code restructuring - source code is analyzed and violations of structured programming practices are noted and repaired, the revised code also needs to be reviewed and tested
- Data restructuring - usually requires full reverse engineering, current data architecture is dissected and data models are defined, existing data structures are reviewed for quality
- Forward engineering - also called reclamation or renovation, recovers design information from existing source code and uses this information to reconstitute the existing system to improve its overall quality and/or performance

Reverse Engineering Concepts

- Abstraction level - ideally want to be able to derive design information at the highest level possible
- Completeness - level of detail provided at a given abstraction level
- Interactivity - degree to which humans are integrated with automated reverse engineering tools
- Directionality - one-way means the software engineer doing the maintenance activity is given all information extracted from source code, two-way means the information is fed to a reengineering tool that attempts to regenerate the old program
- Extract abstractions - meaningful specification of processing performed is derived from old source code

Reverse Engineering Activities

- Understanding processing - source code is analyzed at varying levels of detail (system, program, component, pattern, statement) to understand procedural abstractions and overall functionality

- Understanding data
- internal data structures - program code is examined with the intention of grouping related program variables.
- database structure - often done prior to moving from one database paradigm to another (e.g., flat file to relational)

Reverse Engineering User interfaces

- What are the basic actions (e.g., key strokes or mouse operations) processed by the interface?
- What is a compact description of the system's behavioral response to these actions? □ What concept of equivalence of interfaces is relevant here?

Restructuring Benefits

- Improved program and documentation quality
- Makes programs easier to learn, improves productivity, reduces developer frustration
- Reduces effort required to maintain software
- Software is easier to test and debug
- Code restructuring
- program logic modeled using Boolean algebra and series of transformation rules are applied to yield restructured logic
- create resource exchange diagram showing data types, procedure and variables shared between modules, restructure program architecture to minimize module coupling

□ Data restructuring

- analysis of source code
- data redesign
- data record standardization
- data name rationalization
- file or database translation

Identifying Forward Engineering Candidates

1. Program will continue to be used for several more years
2. Program is currently being used successfully
3. Program is likely to undergo major modification or enhancement in the future

Forward Engineering Client/Server Architectures

- application functionality migrates to each client computer
- new GUI interfaces implemented at client sites
- database functions allocated to servers
- specialized functionality may remain at server site
- new communications, security, archiving, and control requirements must be established at both client and server sites

Forward Engineering Object-oriented Architectures

- existing software is reverse engineered so that appropriate data, functional, and behavioral models can be created
- use-cases are created if reengineered system extends functionality of application
- data models created during reverse engineering are used with CRC modeling as a basis to define classes
- create class hierarchies, object-relationship models, object-behavior models and begin object-oriented design
- a component-based process model may be used if a robust component library already exists

-
- where components must be built from scratch, it may be possible to reuse algorithms and data structures from the original application

Forward Engineering User interfaces

1. understand the original user interface and how the data moves between the user interface and the remainder of the application
 2. remodel the behavior implied by the existing user interface into a series of abstractions that have meaning in the context of a GUI
 3. introduce improvements that make the mode of interaction more efficient
 4. build and integrate the new GUI
- #### **Economics of Reengineering**
- $\text{Cost of maintenance} = \text{cost annual of operation and maintenance over application lifetime}$
 - $\text{Cost of reengineering} = \text{predicted return on investment reduced by cost of implementing changes and engineering risk factors}$
 - $\text{Cost benefit} = \text{Cost of reengineering} - \text{Cost of maintenance}$