MCA 301: Software Engineering

UNIT I

Software and Software Engineering: Introduction, Software, Software Myths, Software Engineering-Process:- Software Engineering, Software Processes models; Evolutionary process Models, Component based development; Formal Methods Model, Fourth generation Techniques. An Agile view of processes and Development: Software Engineering practice – Software Engineering, communication, planning, modeling, construction practices and deployment.

UNIT II

System Engineering: Computer-based systems, the system engineering Hierarchy, business process engineering, product engineering and system modeling. Building the analysis model, Requirement analysis, modeling approaches, data modeling. Behavioral model.

UNIT –III

Design Engineering: Design process and quality, design concepts the design model, and pattern-used software design. Architectural design: Software architecture, data design, architectural styles and patterns, architectural design mapping data flow into software architecture. Component-based software engineering, Critical systems development, Software reuse, User interface design, web apps design issues and architecture design.

UNIT –IV

Testing strategies: Strategies and issues, testing strategies for and object-oriented software. Validation testing and system testing. Software testing tactics: Fundamentals, black-box and white-box testing white-box testing basis path testing. Control structure testing, Black-box testing, Object-oriented testing methods. Testing methods applicable at the class level inter class testing case design.

UNIT V

Product Metrics: Software quality, framework, metrics for analysis model design model, source case and testing. Managing Software Projects: The management spectrum, the W^5 HH principle, metrics in process, software measurement,Estimation: Observations, Decomposition Techniques, Empirical Models, Estimation For Object-Oriented Projects Other Estimation Techniques, Project Scheduling, Risk Management, Reengineering.

Text Books:

Roger, S, Pressman, Software Engineering, A Practitioner's Approach, Six Edition, McGraw-Hill,

International Edition, 2005.

Ian Sommerville, Software Engineering, Pearson Education, 8th Edition.

Reference Books:

James F Peters, Software Engineering, John Wiley

Waruan S Jawadekar, Software Engineering, Tata McGraw Hill, 2004.

Carlo Ghezzi, Mehdi Jazayeri, Dino Manrioli, Fundamentals of Software Engineering, PHI, 2001
Pankaj Jalote, An Integrated approach to Software Engineering Narosa

Lecture notes

Unit-1

¬ 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 o For more Scalability o Cost Management o To manage the dynamic nature of software o 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. o 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. o 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. o 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. o 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. o 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.

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.

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.

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:

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 procedure

¬ Software Myths Software myths propagate misinformation and confusion. There are three kinds of software myths:

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? RCR Institute of Management and Technology

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

Customer myth:

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

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 time

Software Engineering Processes

s. 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 dia- gram 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 RequirementsSolve the Problem Design and Implement Software Product Verify the Solution Test that Product Meets Requirements 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. Analyze and Specify Software Requirements Design and ImplementSoftware Product Test that Product Meets Requirements Deploy, Maintain, andSoftware Engineer

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.

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,

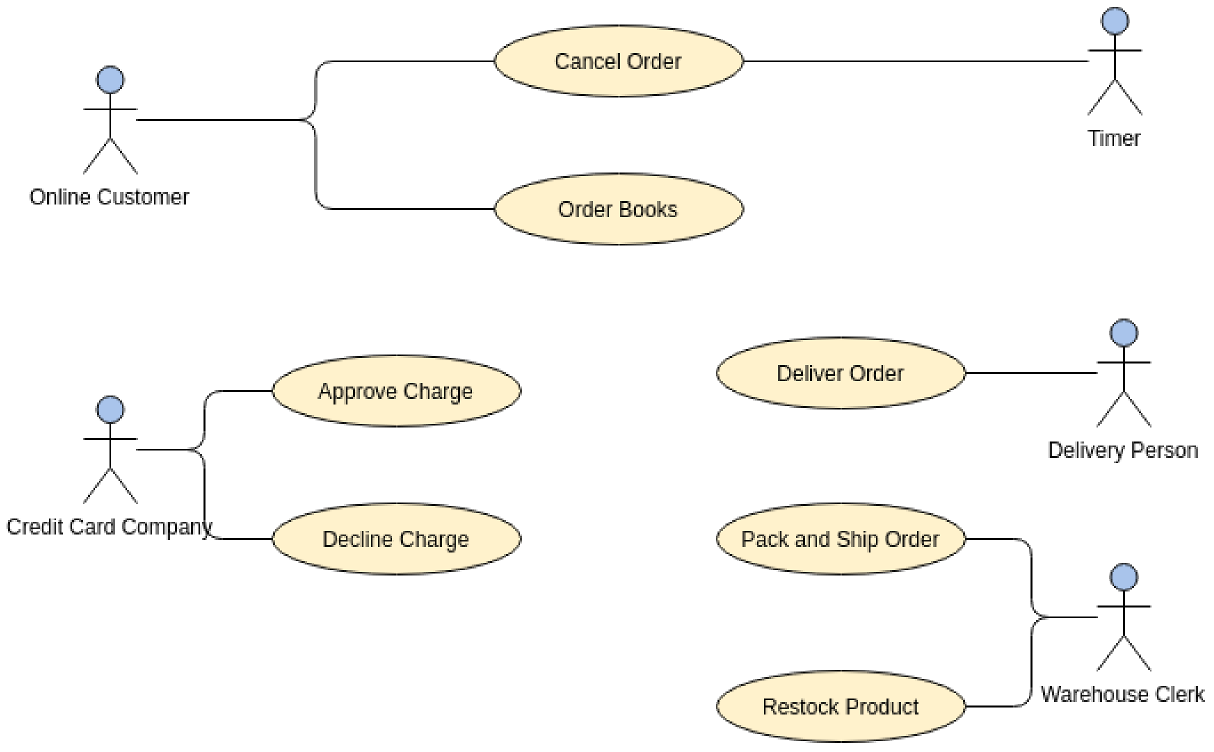
steps is depicted with the arrowed lines. Process sub-steps are typically shown with a box expansion notation. For example, Figure 2 shows the expansion of the "Analyze and Specify" step. The activities of the first sub-step include general aspects of requirements analysis, such as defining the overall 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. DownFigure 2: Expansion of the ‘‘Analyze and Specify’’ Step.

Analyze and Specify Software Requirements

Perform General Requirements Analysis State Problem to be SolvedIdentify People Involved Analyze Operational SettingAnalyze Impacts Identify Positive Impacts Identify Negative Impacts Analyze Related SystemsAnalyze 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-



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.

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 Beginner

s. 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

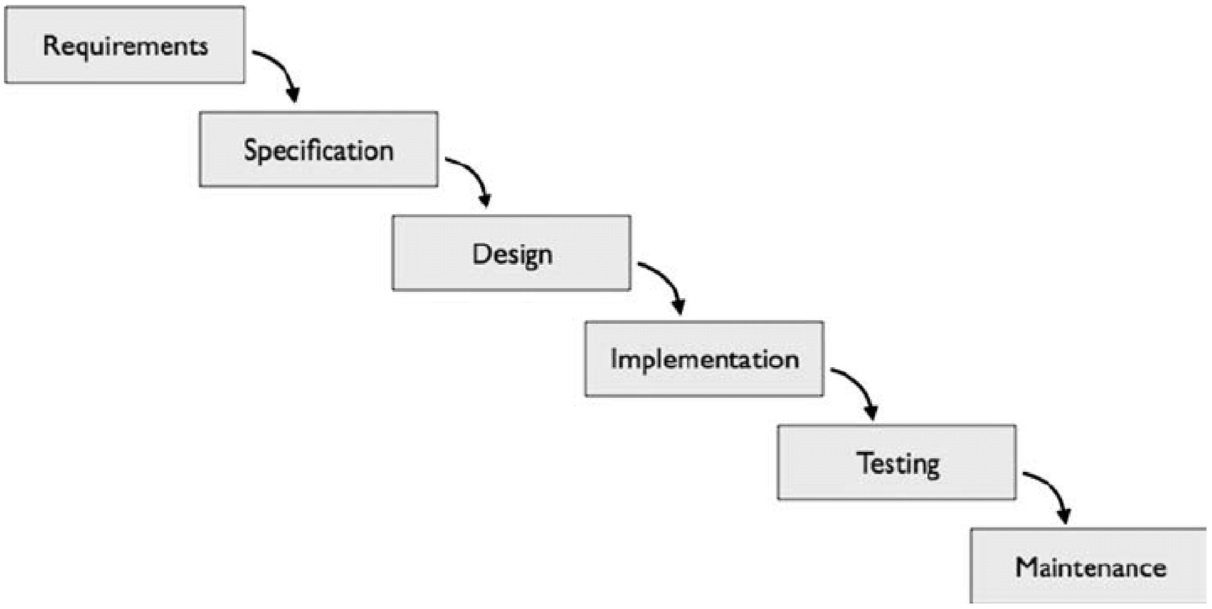
mathematical system specification and transforming this specification, using mathematical methods to a program. These transformations are 'correctness preserving.' This means that you can be sure that the developed programs meet its specification. 4. System assembly from reusable components: This method assumes the parts of the system already exist. The system development process targ

¬ 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.



DLC Models

The SDLC aims to produce high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

Some of the SDLC Models are as follows

Waterfall Model

Incremental SDLC Model

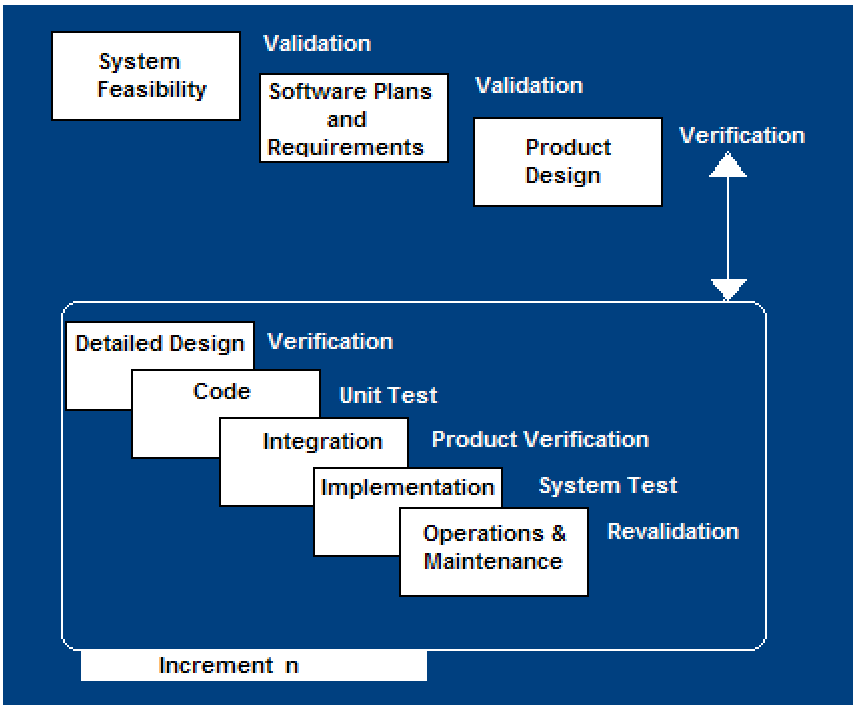
Spiral Model

Evolutionary Prototyping Model

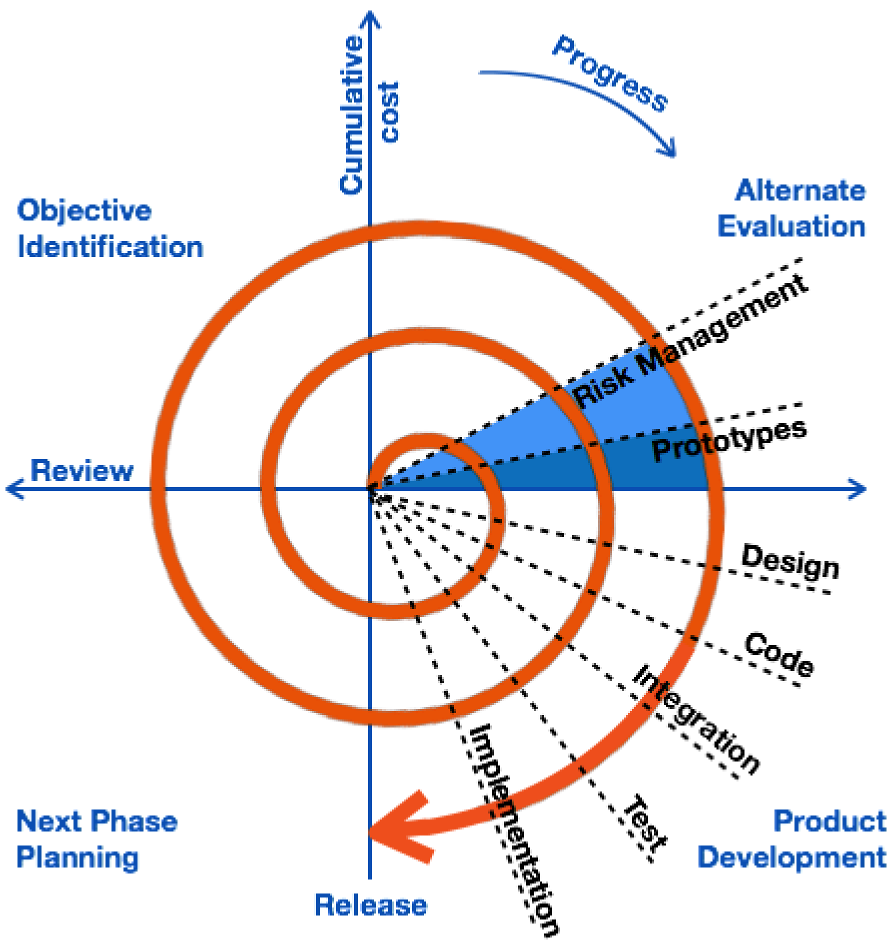
Agile Model

RAD Model

2) Incremental SDLC Model



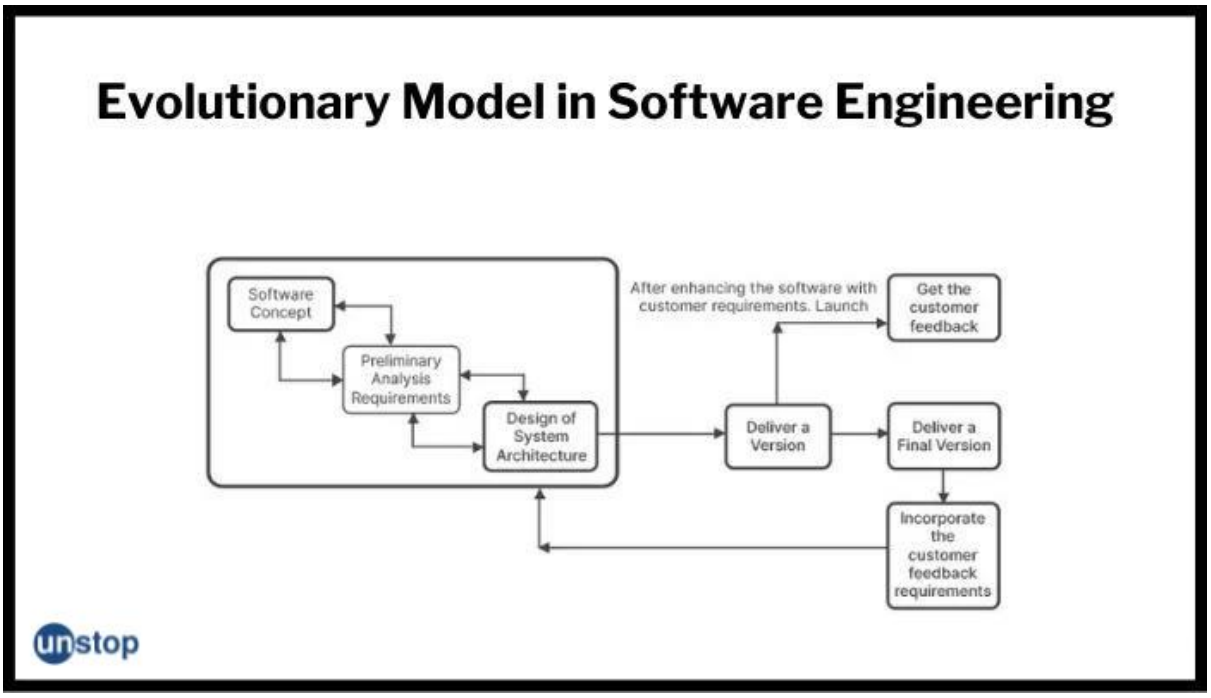
Spiral SDLC Model



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 do


Evolutionary Model in Software Engineering
Software Concept
Preliminary Analysis Requirements
Design of System Architecture
Deliver a Version
After enhancing the software with customer requirements. Launch
Get the customer feedback
Deliver a Final Version
Incorporate the customer feedback requirements
unstop

¬ 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

¬ 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 practic

uccessful 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

Short question important:

1.Difference between software and enginnering?

2.what are software myths?

3.what is meant by software engineering process?

4.list out models in software enginnering?

5.what is deployment?

Long questions

1.what are the various methods available in sdlc?

2.what is component based development?

3.what is meant by fourth generation techinuques?

4.define the factors involved in software enginnering?

5.explain construction planning and deployment?

UNIT-2

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

¬ 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 system:

1.Set or arrangement of things so related as to form a unity or organic whole;2.a se

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 portays 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

¬ 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 = {D1 D2 D3………, Dn} Each domain is composed of specific elements

(Ej) each of which serves some role in accomplishing the objective and goals of the domain and component Di = {E1 E2 E3 …… Em} Finally, each element is implemented by specifying the technical components (Ck) that achieve the necessary function for an element Ej = {C1, C2 C3 …….. C k} In the software context, a component could be a computer program, a reusable program component, a module, a class or object

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:

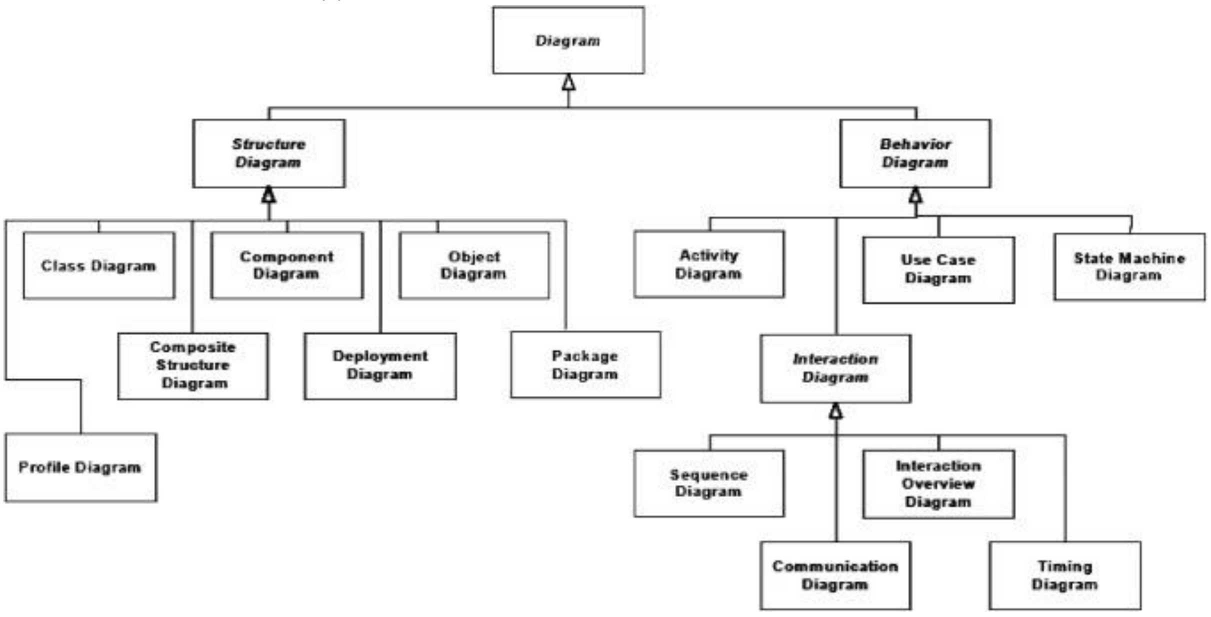
• Define the process that serves the needs of the view under consideration.

• 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 theh



System Simulations

Many computer- based systems interact with the real world in a reactive fashion. That is, realworld events are monitored by the hardware and software that from 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

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.

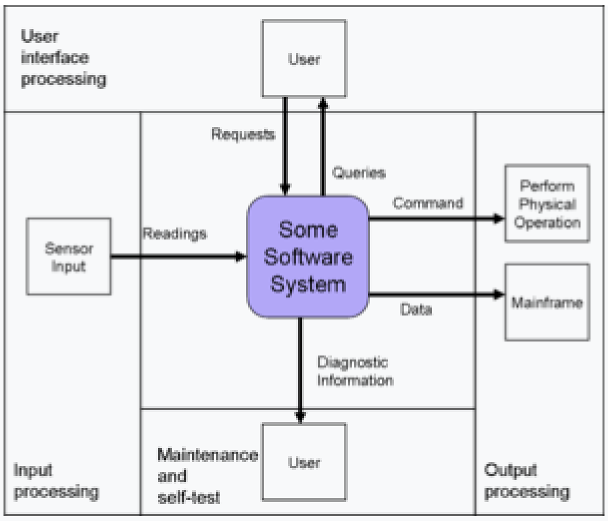
The two objects can be connected by the relationship purchases; that is, a customer purchases product A or product is purchased by customer. The data objects (there may be hundreds or even thousands for a major business activity) flow between business functions, are organized within a database, and are 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 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



A sample system context diagram using Hatley–Pirbhai modeling

Hatley–Pirbhai modeling is a system modeling technique based on the input–process–output model (IPO model), which extends the IPO model by adding user interface processing and maintenance and self-testing processing.[1]

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

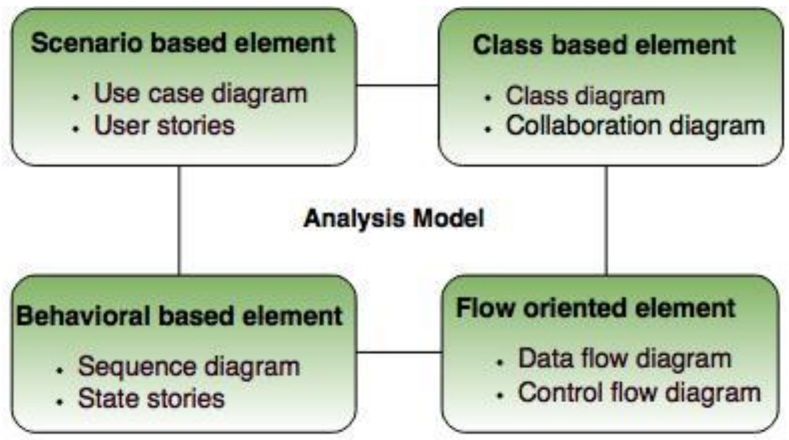


Fig. - Elements of analysis model

2. Class based element

s • 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

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.

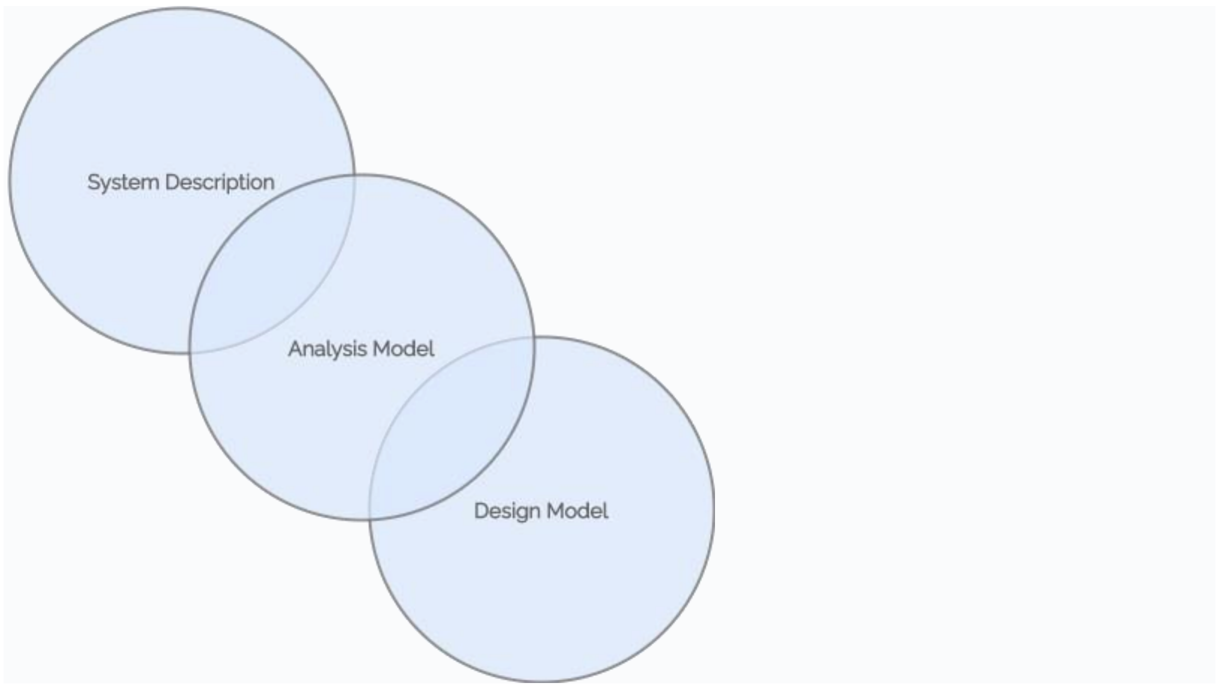
3. Defining a set of criteria that can be validated once the software is constructed. The analysis model bridges the gap between a systemlevel description of the overall system or business functioning as it is delivered through the use of software, hardware, data, human, and other system aspects and 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 be

ata, human, and other system aspects and a software design. This relationship is shown in the figure below.



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. ¬

Data Modeling Concept

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,or3

)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.

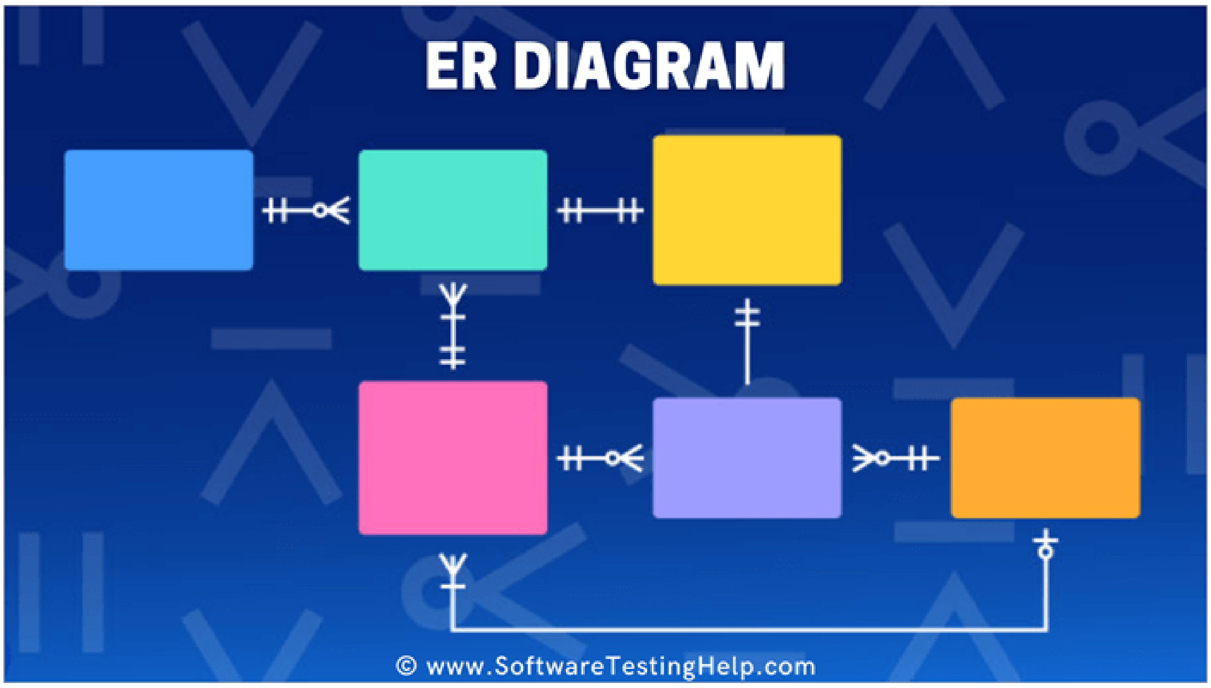
The attributes for car might serve well for an application that would be used by a Department of Motor vehicles, but these attribute.

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 o



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)

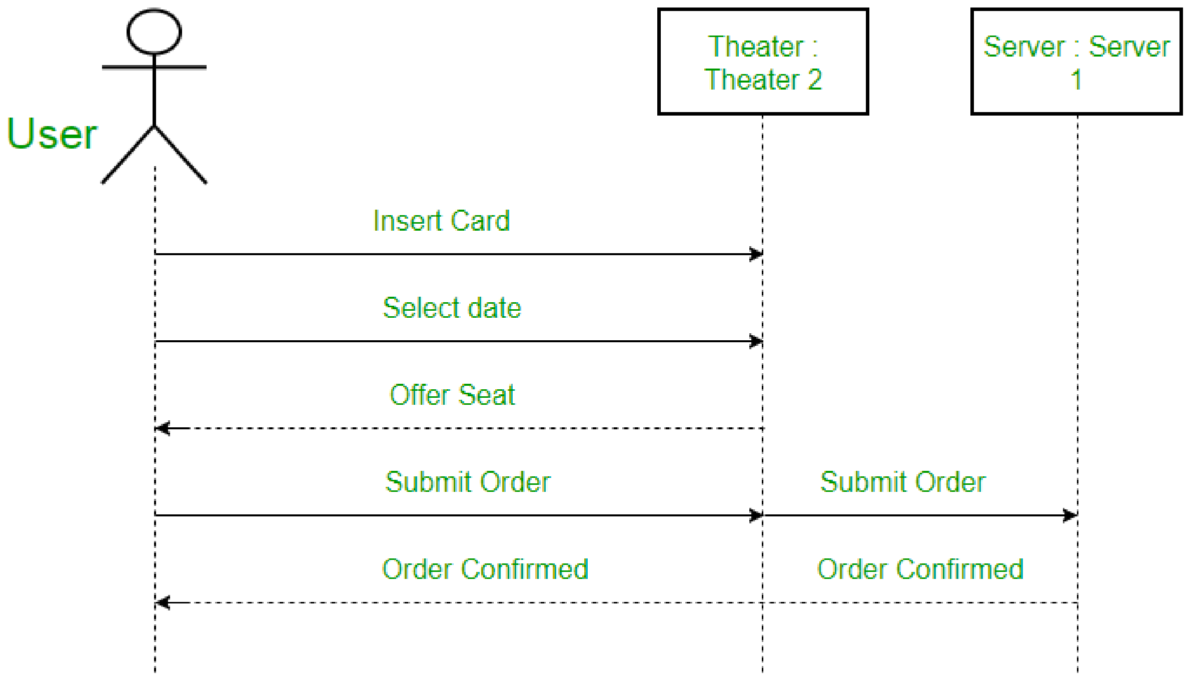
• It indicates how the event channels behavior between Safe Home objects.

• Time is measured vertically (downward),

• Narrow vertical rectangles represent time spent in processing an activit

symbol for actorWe use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

For example – Here the user in seat reservation system is shown as an actor sswhere it exists outside the system and is not a part of the system.



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.

Important questions

Shorts:

1.what is system enginnering?

2.explain computer based systems?

3.what is business process?

4.what is meant by product engineering?

5.what is analysis?

Long:

1.what is meant system enginnering herarchy?

2.what is meant by business process engineering?

3.explain briefly approaches involved in software enginnering?

4.explain datail on datamodelling?

5.write short on behavioral model?

UNIT-III

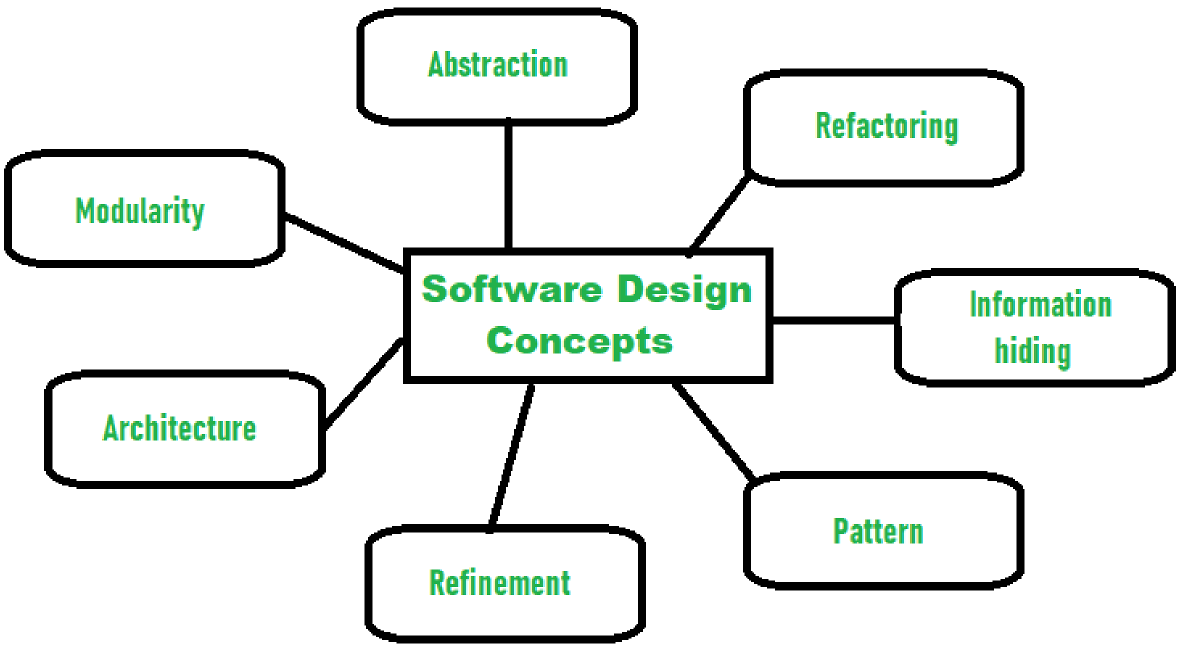
Design Engineering Encompasses

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, thedesign model provides detail about software architecture, data structures, interfaces, andcomponents 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



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 architecturecan be implemented.

The interface design describes how the software communicates with systems

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

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 a

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 th

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. 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 softwarModularity: 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.

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 i

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.

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

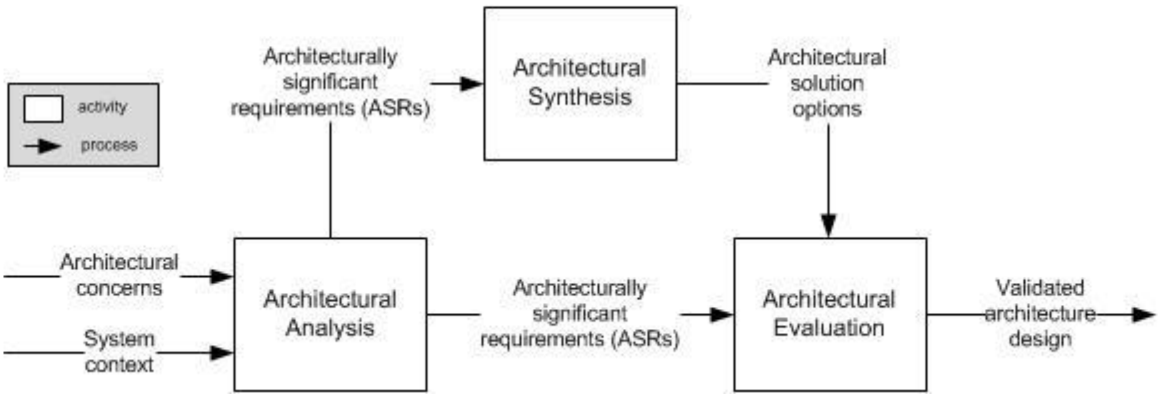
A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e.,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

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.

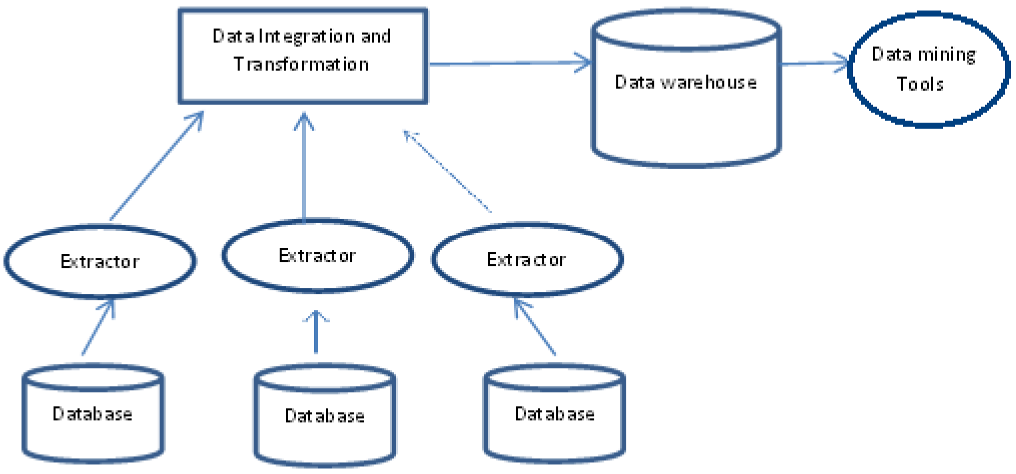
SOFTWARE ARCHITECTURE What Is Architecture?

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.


activity
process
Architecturally significant requirements (ASRs)
Architectural Synthesis
Architectural solution options
Architectural concerns
System context
Architectural Analysis
Architecturally significant requirements (ASRs)
Architectural Evaluation
Validated architecture design

a pattern imposes a rule on the architecture, describing how the software will handle someaspect 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). Patterns can be used in conjunction with an architectural style to shape the overall structure of asystem.

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



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. Object-oriented architectures. The 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 architechture:

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. Object-oriented architecture

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

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

. THE GOLDEN RULE

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 guidethis important aspect of software design.

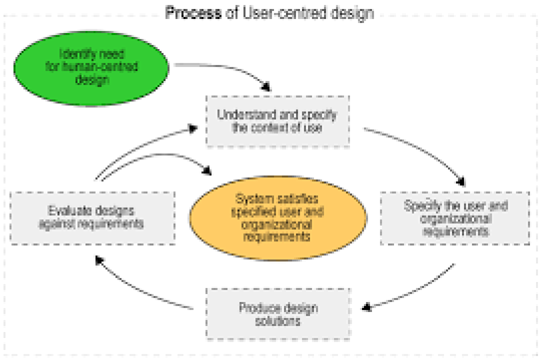
Place the User in Control: During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.

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 move

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 intheir 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

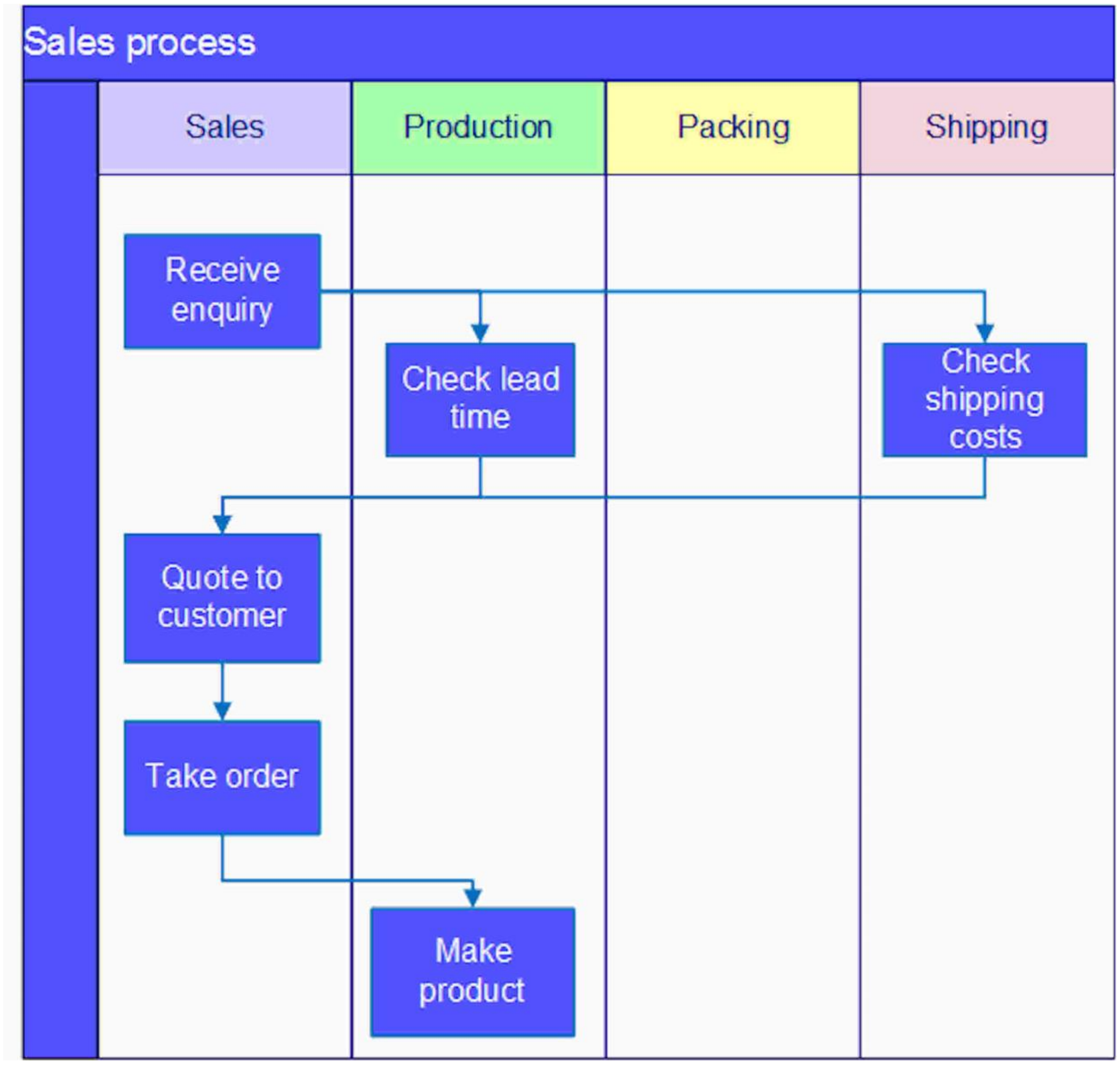

Process of User-centred design
Identify need for human-centred design
Understand and specify the context of use
Specify the user and organizational requirements
Produce design solutions
Evaluate designs against requirements
System satisfies specified user and organizational requirements

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



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

Important question short:

Short:

1.what is design enginnering?

2.explain design concepts?

3.what is patterns invoved in designs?

4.what is architectural design?

5.explain software architecture?

Long:

1.what is software architecture,data design and archictural styles?

2.what are patterns and mapping designs?

3.what is component based software enginnering?

4.what is meant by critical system development?

5.what is reuse and user interface?

UNIT-4

STRATEGIES OF TESTING:

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 systematic

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 thefollowing: 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).

MYTH: Anyone can test.

FACT: Testing is a rigorous discipline and requires many kinds of skills.

MYTH: There is no creativity in testing.

FACT: Creativity can be applied when formulating test approaches, when designing tests, and even when executing tests.

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 increasing

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

Testing shows the presence of defects

Exhaustive Testing is not possible

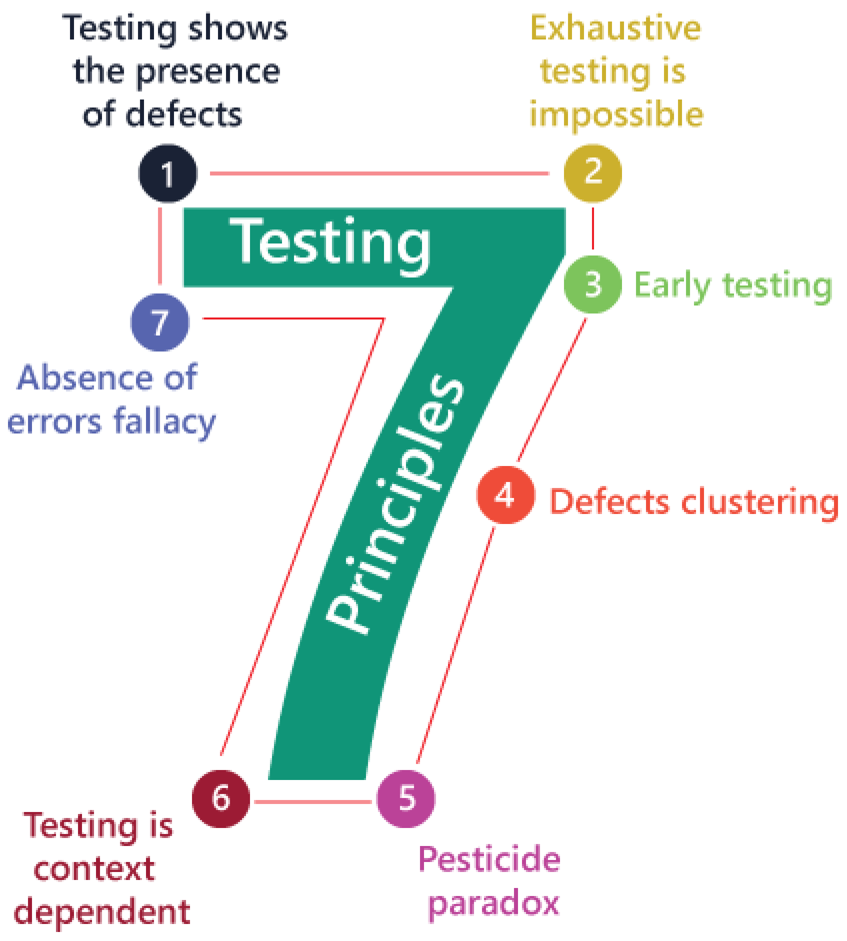
Early Testing

Defect Clustering

Pesticide Paradox

Testing is context-dependent

Absence of errors fallacy


Testing shows the presence of defects
1
Exhaustive testing is impossible
2
Testing
3
Early testing
7
Absence of errors fallacy
Principles
4
Defects clustering
6
Testing is context dependent
5
Pesticide paradox

Testing shows the presence of defects

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

Software testing life cycle contains the following steps

: 1. Requirement Analysis

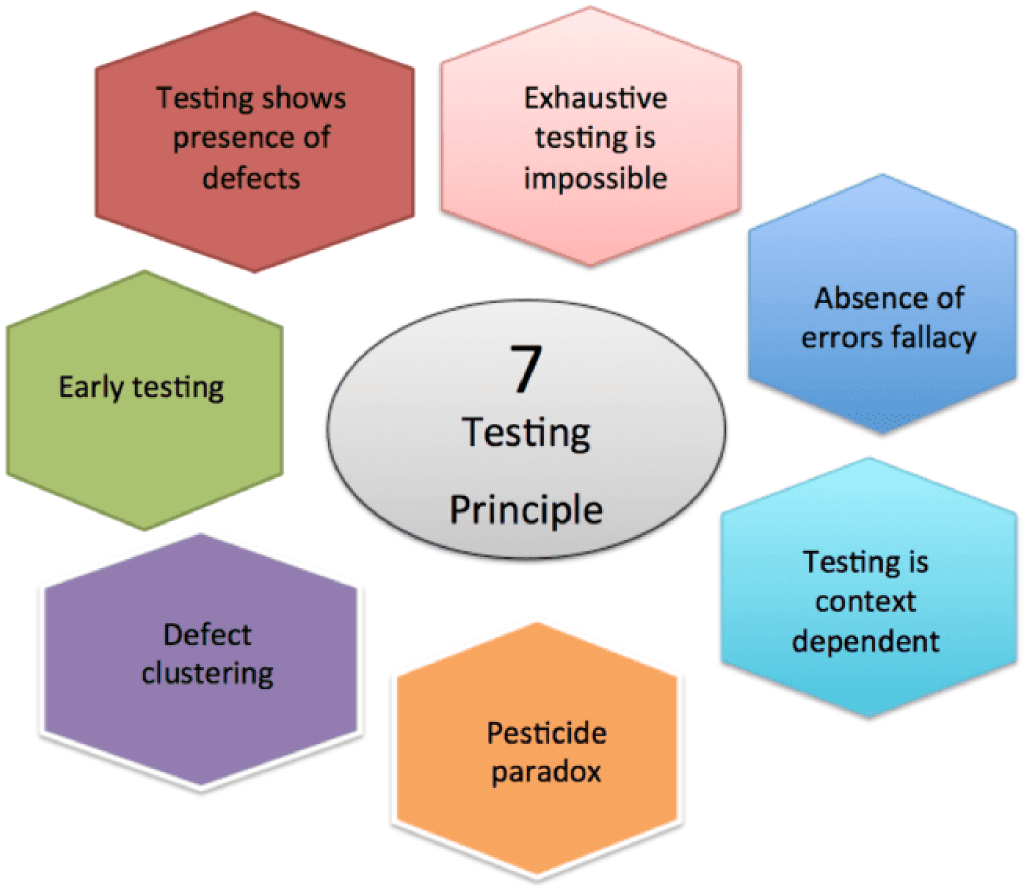
2. Test Plan Creation

3. Environment setup

4. Test case Execution

5. Defect Logging

6. Test Cycle Closur


Testing shows presence of defects
Exhaustive testing is impossible
Absence of errors fallacy
Early testing
7 Testing Principle
Testing is context dependent
Defect clustering
Pesticide paradox

Integration Testing in the OO

Context Different strategies for integration testing of OO systems.

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 nonfunctional testing includes User accept

¬ 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

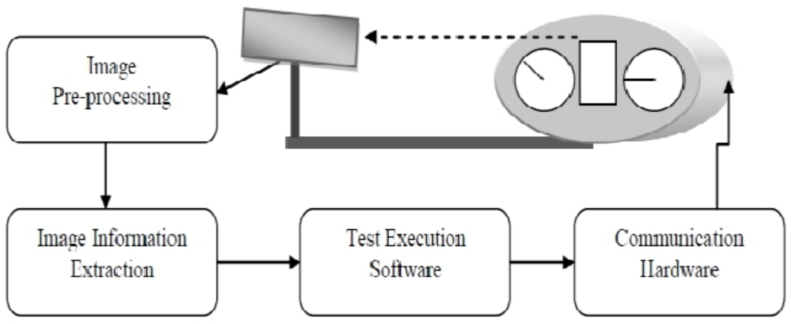
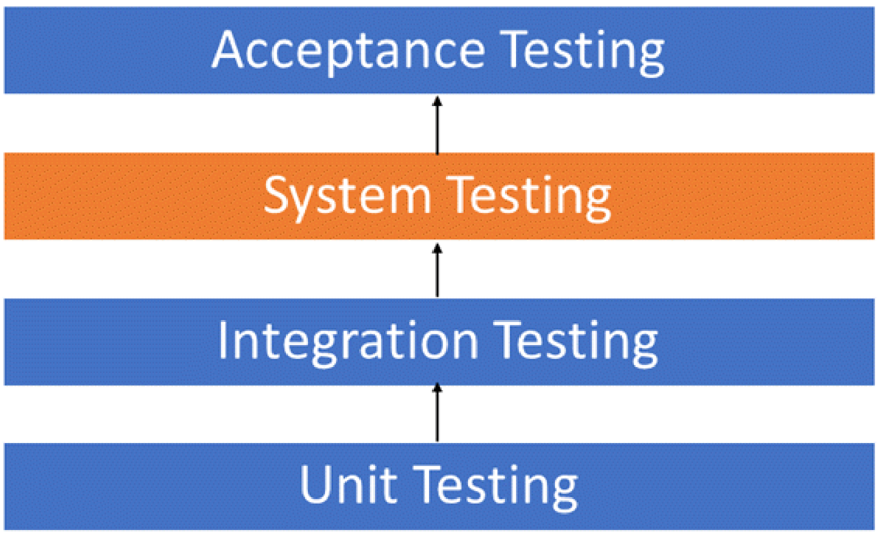


Figure 2. Automated camera based test system

SYSTEM TESTING:

That is a very basic description of what is involved in system testing. You need to build detailed test cases and test suites that test each aspect of the application as seen from the outside without looking at the actual source code.

Software Testing Hierarchy



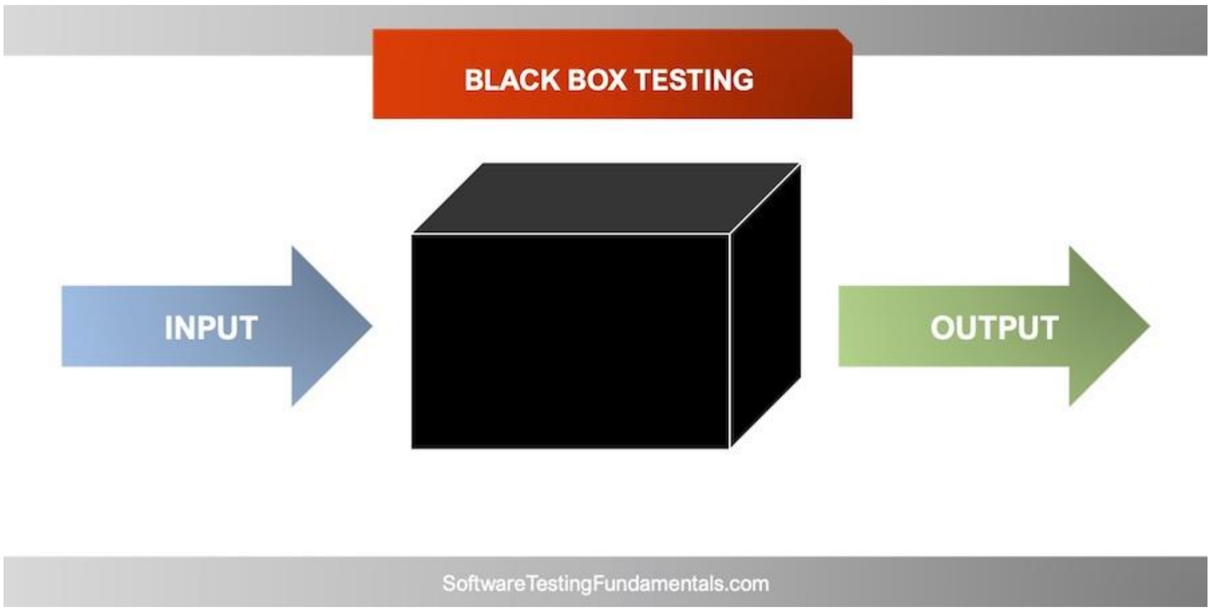
As with almost any software engineering process, software testing has a prescribed order in which things should be done. The following is a list of software testing categories arranged in chronological order. These are the steps taken to fully test new software in preparation for marketing it:

TYPES OF SOFTWARE 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.



BLACK BOX TESTING, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is **not** known to the tester. These tests can be functional or non-functional, though usually functional.

ISTQB Definition

black box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system.

black box test design technique: Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

Table of Contents

Elaboration

This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

Incorrect or missing functions

Interface errors

Errors in data structures or external database access

Behavior or performance errors

Initialization and termination errors

Example

A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.

Levels

Black Box Testing method is applicable to the following levels of software testing:

Integration Testing

System Testing

Acceptance Testing

The higher the level, and hence the bigger and more complex the box, the more black-box testing method comes into use.

Techniques

Following are some techniques that can be used for designing black box tests.

Equivalence Partitioning: It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

Boundary Value Analysis: It is a software test design technique that involves the determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.

Cause-Effect Graphing: It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Advantages

Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.

Tester need not know programming languages or how the software has been implemented.

Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.

Test cases can be designed as soon as the specifications are complete.

Disadvantages

Only a small number of possible inputs can be tested and many program paths will be left untested.

Without clear specifications, which is the situation in many projects, test cases will be difficult to design.

Tests can be redundant if the software designer/developer has already run a test case.

Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

Black Box Testing is contrasted with White Box Testing. Read Differences between Black Box Testing and White Box Testing.

Top of Form

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: o Unit Testing o Integration Testing o 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:

o Incremental Testing

o Non-Incremental Testing Incremental Integration

Testing Whenever there is a clear relationship between.

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

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 Technique Decision Table Technique is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form. It is appropriate for the functions that have a logical relationship between two and more than two inputs. Boundary Value Technique Boundary Value Technique is used to test boundary values,

boundary values are those that contain the upper and lower limit of a variable. It tests, while entering boundary value whether the software is producing correct output or not. State Transition Technique State Transition Technique is used to capture the behavior of the software application when different input values are given to the same function. This applies to those types of applications that provide the specific number of attempts to access the application. All-pair Testing Technique All-pair testing Technique 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, list box, text box, etc. Cause-Effect Technique Cause-Effect Technique underlines the relationship between a given result and all the factors affecting the result.

It is based on a collection of requirements. Equivalence Partitioning Technique Equivalence partitioning is a technique of software testing in which input data divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior. Error Guessing Technique 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 the experience to guess the problematic areas of the software. Use Case Technique 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

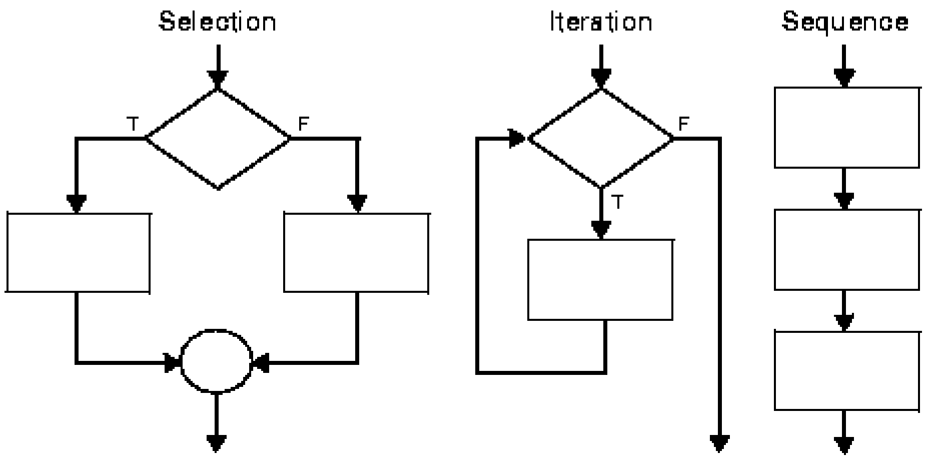
NOT.

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

Control structures:

Figure 3-1 Control Structures



Let’s understand each step one by one.

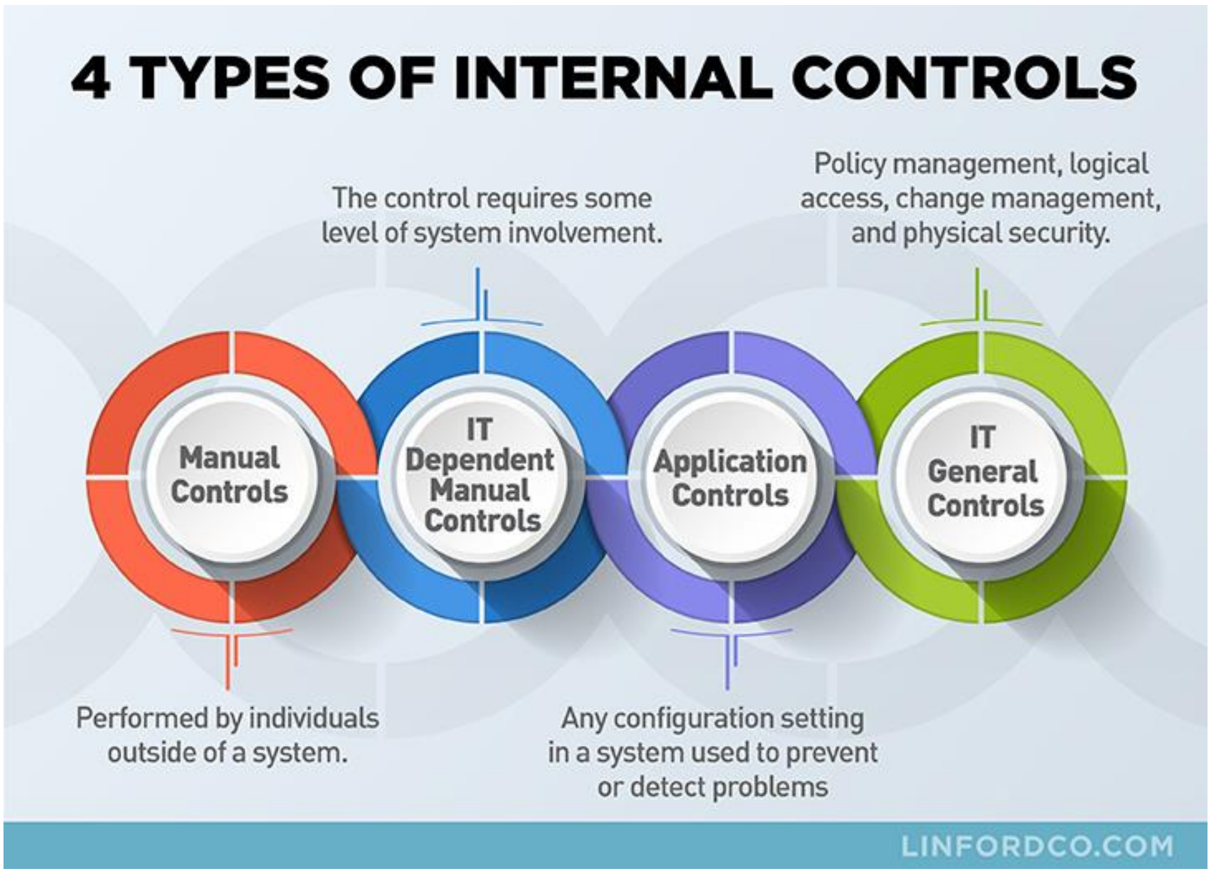
Control Flow Graph

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 then 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 constructin

ccording to the Executive Summary of the Internal Control – Integrated Framework from the Committee of Sponsoring Organizations (COSO), an *“internal control is a process, effected by an entity’s board of directors, management, or other personnel, designed to provide reasonable assurance regarding the achievement of objectives relating to operations, reporting, and compliance.”* The main goal of having internal controls is to set up key points in a process, which allows companies to track progress and sustainability of performance. In the next section, we will review control definitions and internal control examples.


4 TYPES OF INTERNAL CONTROLS
The control requires some level of system involvement.
Policy management, logical access, change management, and physical security.
Manual Controls
IT Dependent Manual Controls
Application Controls
IT General Controls
Performed by individuals outside of a system.
Any configuration setting in a system used to prevent or detect problems
LINFORDCO.COM

What Are the 4 Different Types of Controls?

When performing an audit, auditors will look to see that they can gain assurance over a process by focusing on four main types of internal controls. These types of controls consist of the following:

Manual Controls

IT Dependent Manual Controls

Application Controls

IT General Controls

The four types of internal controls mentioned above are key as they are pervasive (or at least should be) in the processes that support the systems and services provided by service organizations to their user organizations (i.e. clients and customers).

What Are Internal Control Definitions & Examples?

What Are Manual Controls?

Manual controls are performed by individuals outside of a system

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 edge

s 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 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 (for Region 1) + 1 (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 mult¬ 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 toiple functions, then a separate flow graph is constructed for each one of RCR Institute of Management and Technology

must conform to

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

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

Important question:

Short

1.what is meant by stratagies?

2.what is testing strategy?

3.explain oop software?

4.explain validation testing?

5.what is system testing?

Long:

1.what are fundamentals of testing involved in software engineering?

2.what is meant by path testing?

3.what is control structure testing?

4.what is estimation and testing methods?

5.explain class level test case design?

. UNIT-5

¬ 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 $PL = 1/ [(n1/2) * (N2/n2)]$ … (1) For a particular module (z), the percentage of overall testing effort allocated can be calculated by the following equation. 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 s

• A metric should have desirable mathematical properties

• The value of a metric should increase when positive software traits occur or decrease whenundesirable 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 andexternal 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

¬ 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 ϖ Estimate the cost or effort required to design, code and test the software.

ϖ Predict the number of errors that will be encountered during testing. ϖ 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=count total* [0.65+0.01+∑ (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

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

Management Spectrum

)

. • 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?:

¬ 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 parallelwith one another to complete the project on time. The output from one task often determines when anothermay 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 projec

 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. Thendata 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 requirementsand 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'smission objectives. • Reliability: The extent to which a program can be expected to perform its intended function with

Project Scheduling Perspectives

• One view is that the end-date for the software release is set externally and that the softwareorganization 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 usethe 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 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 staffmembers 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 workproduct or deliverable)

• Defined milestones - a milestone is accomplished when one or more work products from anengineering 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 cooperationwith one another)

• The main reasons for using more than 1 person on a project are to get the job done more rapidly andto improve software quality. Project Effort Distribution

• The 40-20-40 rule (a rule of thumb): o 40% front-end analysis and desig

• 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 proposedwork

• Technology risk assessment - evaluates the risk associated with the technology implied by thesoftware scope

• Proof of concept - demonstrates the feasibility of the technology in the software c

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 todeal 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 ifthe 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 frompast 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 Item

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

1. Basic COCOMO Model:
2. The basic COCOMO model provide an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model: Effort=a1*(KLOC) a2 PM Tdev=b1*(efforts) b2 Months Where KLOC is the estimated size of the software product indicate in Kilo Lines of Code, a1, a2, b1, b2 are constants for each group of software products, Tdev is the estimated time to develop the software, expressed in months, Effort is the total effort required to develop the software product, expressed in person months (PMs). Estimation of development effort
3. For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:
4. Organic: Effort = 2.4(KLOC) 1.05 PM
5. Semi-detached: Effort = 3.0(KLOC) 1.12
6. Basic Model
7. Intermediate Model
8. 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:

Effort=a_1*(KLOC) a_2 PM
Tdev=b_1*(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,

Cons

- COCOMO disregards the project's requirements and all of the project's relevant documentation.
- COCOMO ignores customer skills, collaboration, knowledge, and other factors.

- Sometimes COCOMO cannot clearly understand the project between the developers and the clients
- Changes in the actual amount of time spent on these phases will impact the accuracy of the COCOMO.
- Certain factors, such as hardware faults and breakdowns, are beyond the developers' or customers' control.

Important questions:

1.Define metrics?

2.what are software quality?

3.what is framework?

4.explain concept measurements?

5.what is project?

Long answers:

1.what are the formal methods of software enginnering explain briefly?

2.explain source case and testing?

3.what is software decomposition and empirical modela?

4.explain risk management and reengineering?

5.what is management spectrum and W5 HN Principle?

Question papers for Sample:

S.V.U. COLLEGE OF COMMERCE MANAGEMENT AND COMPUTER SCIENCE :: TIRUPATHI

DEPARTMENT OF COMPUTER SCIENCE

Time:**2hours** **INTERNAL EXAMINATIONS -I**

301- SOFTWARE ENGINNERING MAX MARKS:30

Section-A

Answer any five from the following 5*2=10m

1.what is software Enginnering?

2.Difference between waterfall and raid model?

3.what is software myth?

4.what is deployment in SE?

5.Explain system enginnering?

6.Explain system modelling?

7.what is heriarchy in engineering?

8.Explain behaviour model?

Section-B

Answer any one Question from each unit 2*10=20marks

Unit-1

9. what is meant by sdlclife cycle and myths that are involved in software engineering?

Or

10.what are the model in se? four generation in techinuqes in softwareenginnering?

Unit-2

11.explain computer based systems?and business process enginnering?

Or

12.what are approaches involved in se?building the analysis model?

S.V.U. COLLEGE OF COMMERCE MANAGEMENT AND COMPUTER SCIENCE :: TIRUPATHI

DEPARTMENT OF COMPUTER SCIENCE

Time:**2hours** **INTERNAL EXAMINATIONS -||**

301- SOFTWARE ENGINNERING MAX MARKS:30

Section-A

Answer any five from the following 5*2=10m

1.what is design enginnering?

2.Difference between oop object and stratagies?

3.what is critical development system?

4.what is testing?

5.Explain framework?

6.Explain management spectrum?

7.what is software reuse?

8.Explain Reengineering?

Section-B

Answer any one Question from each unit 2*10=20marks

Unit-1

9. explain component based software enginnering detail?

Or

10.what is web app design andissue in architectural desing?

Unit-2

11.what is software testing tactics?types of testing?

Or

12.explain management principle?and WHN principle?

MASTER OF COMPUTER APPLICATIONS DEGREE EXAMINATION

SECOND SEMESTER

Paper MCA 301: SOFTWARE ENGINNERING

(Under C.B.S.C Revised Regulations w.e.f.2021-2023)

(Common paper to University and all Affliated Colleges)

Time:3hrs **maxmarks70**

COMPULSARY

ANSWER ANY FIVE QUESTION: 5*2=10marks

1)a.)what is software Enginnering?

b).Difference between waterfall and raid model?

c).what is software myth?

d).what is deployment in SE?

f).Explain system enginnering?

g).what is design enginnering?

h).Difference between oop object and stratagies?

I).what is critical development system?

j).what is testing?

k).what is risk management?

SECTION-B

Answer any one question from each unit 5*10=50marks

UNIT-1

2.what is meant by sdlclife cycle and myths that are involved in software engineering?

Or

3.what are the model in se?four generation in techinuqes in softwareenginnering?

Unit-2

4.explain computer based systems?and business process enginnering?

Or

5.what are approaches involved in se?building the analysis model?

Unit-3

6.explain component based software enginnering detail?

Or

7.what is web app design and eissue in architectural desing?

Unit-4

8.what is software testing tactics?types of testing?

Or

9.explain management principle?and WHN principle?

Unit-5

10.explain product metrics detail ?analysis model design?

Or

11.explain risk management deatail?and reenginnering?