Software Engineering

DECAP509

Edited by
Balraj Kumar

LOVELY PROFESSIONAL UNIVERSITY

Software Engineering

Edited By:
Balraj Kumar

Title: SOFTWARE ENGINEERING

Author's Name: Ashwani Kumar

Published By : Lovely Professional University

Publisher Address: Lovely Professional University, Jalandhar Delhi GT road, Phagwara - 144411

Printer Detail: Lovely Professional University

Edition Detail: (I)

ISBN: 978-81-19929-99-3

Content

Unit 01: Introduction to Software Engineering

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss various concepts of software engineering.
- Characteristics and types of software
- software process and software engineering practices
- Explain the software engineering challenges and approach

Introduction

User needs and constraints must be determined and explicitly stated before a software product can be developed; the product must be designed to accommodate implementers, users, and maintainers; the source code must be carefully implemented and thoroughly tested; and supporting documents must be prepared. Analysis of change requests, redesign and modification of source code, rigorous testing of the updated code, update of documents to reflect the changes, and dissemination of modified work products to the appropriate user are all examples of software maintenance duties.

In the 1960s, the necessity for methodical approaches to software creation and maintenance became obvious. Cost overruns, timetable slippage, lack of reliability, inefficiency, and lack of user approval plagued many software projects at the time. As computer systems grew larger and more complex, it became clear that demand for software was outpacing our ability to create and maintain it. As a result, the field of software engineering has grown into a significant technological discipline.

In the last four decades, the complexity and nature of software have changed dramatically. Applications of the 1970s used a single processor, took single line inputs, and returned alphanumeric results. Today's programmes, on the other hand, are significantly more complicated,

run-on client-server technology, and have a user-friendly interface. They run on a variety of CPUs, operating systems, and even machines from various parts of the world.

The software groups do everything they can to stay on top of fast evolving new technologies while also dealing with development challenges and backlogs. Even the Software Engineering Institute (SEI) recommends that the development process be improved. Change is an unavoidable requirement of the hour. However, it frequently leads to tensions between those who embrace change and others who staunchly adhere to established working methods. As a result, there is a pressing need to adopt software engineering principles, techniques, and strategies in order to avoid conflicts and improve software development in order to provide high-quality software on time and on budget.

Software is a set of instructions to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software.It also includes a set of documents, such as the software manual, meant for users to understand the software system

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

1.1 Concepts of Software Engineering

Evolving Role of Software

In the previous few decades, software's position has shifted dramatically. Hardware, processing architecture, memory, storage capacity, and a wide range of unexpected input and output situations are all improved. As a result of all of these tremendous advancements, increasingly complex and sophisticated computer-based systems have emerged. Although sophistication leads to better results, it can also present issues for those who design these systems.

A team of software experts has taken the place of the lone coder. In order to deliver a complex application, these professionals concentrate on different pieces of technology. However, the specialists are still confronted with the same issues as a lone coder:

Why does it take long to finish software?

Why are the costs of development so high?

Why aren't all the errors discovered before delivering the software to customers?

Why is it difficult to measure the progress of developing software?

All these questions and many more have led to the manifestation of the concern about software and the manner in which it is developed – a concern which lead to the evolution of the software engineering practices.

Software now serves a dual purpose. It is both a product and a vehicle for the delivery of a product. It supplies the computational power embodied by computer hardware or, more broadly, a network of computers accessible by local hardware as a product.

Software is an information transformer, whether it's in a cell phone or a mainframe computer, creating, organizing, obtaining, changing, displaying, or sending data that might be as simple as a single bit or as complicated as a multimedia presentation. Software serves as the vehicle for delivering the product, serving as the foundation for computer control (operating systems), information exchange (networks), and the creation and control of additional programmes (software tools and environments). The most essential product of our day is information, which is delivered by software.

Software transforms personal data (e.g., an individual's financial transactions) to make it more useful in a local context; it manages business information to improve competitiveness; and it serves as a gateway to global information networks (e.g., the Internet) and a means of acquiring information in all of its forms.

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

The same questions asked of the lone programmer are being asked when modern computer- Notes based systems are built. Give answers to questions:

1. Why does it take so long to get software finished?

2. Why are development costs so high?

3. Why can't we find all the errors before we give the software to customers?

4. Why do we continue to have difficulty in measuring progress as software is being developed?

1.2 Types of Software

System Software- A collection of programs written to service other programs at system level, For example, compiler, operating systems.

Application software- Developed as per user requirement.

Real-time Software- Programs those monitor/analyze/control real world events as they occur.

Business Software- Programs that access, analyze and process business information

Engineering and Scientific Software-:Software using "numbercrunching" algorithms fordifferent science and applications, Systemsimulation, computer-aided design.

Embedded Software-: Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. It has very limited and esoteric functions and control capability.

Artificial Intelligence (AI) Software: Programs make use of AI techniques and methods to solve complex problems. Active areas are expert systems, pattern recognition, games

Internet Software: Programs that support internet accesses and applications. For example, search engine, browser, e-commerce software, authoring tools.

Software Tools and CASE environment: Tools and programs that help the construction of application software and systems. For example, test tools, version control tools.

1.3 Software Characteristics

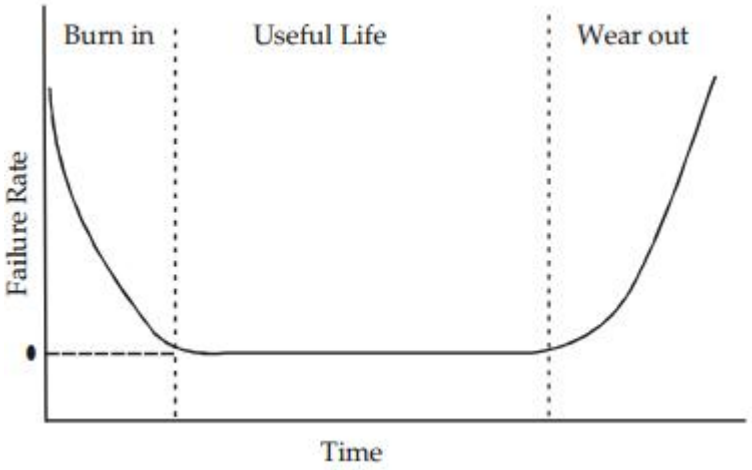
A software product can be judged by what it offers and how well it can be used. In the first NATO conference on software engineering in 1968, Fritz Bauer defined Software engineering as "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines". Stephen Schach defined the same as "A discipline whose aim is the production of quality software, software that is delivered on time, within budget and that satisfies its requirements".

Let us now explore the characteristics of software in detail:

Software is Developed or Engineered and not manufactured: Although there exists fewsimilarities between the hardware manufacturing and software development, the twoactivities differ fundamentally. Both require a good design to attain high quality. But themanufacturing phase of hardware can induce quality related problems that are either nonexistent for software or can be easily rectified. Although both activities depend on peoplebut the relationship between people and work is totally different.

Software does not Wear Out: Figure shows the failure rate of hardware as a function oftime. It is often called the "bathtub curve", indicating that a hardware shows high failureat its early stage (due to design and manufacturing defects); defects get resolved and thefailure rate reduces to a steady-state level for some time. As time progresses, the failurerate shoots up again as the hardware wears out due to dust, temperature and otherenvironmental factors.

Failure Curve for Hardware



Because the software does not undergo environmental degradation, its failure curve ideally should flatten out after the initial stage of its life. The initial failure rate is high because of undiscovered defects. Once these defects are fixed, the curve flattens at a later stage. Thus, the software does not wear out but deteriorates.

Software is Custom Built and Not Designed Component Wise: Software is designed and built so that it can be reused in different programs. Few decades ago subroutine libraries were created that re-used well defined algorithms but had a limited domain. Today this view has been extended to ensure re-usability of not only algorithms but data structures as well.

Functionality: Refers to the degree of performance of the software against its intended purpose.

Reliability: Refers to the ability of the software to provide desired functionality under the given conditions.

Usability: Refers to the extent to which the software can be used with ease.

Efficiency: Refers to the ability of the software to use system resources in the most effective and efficient manner.

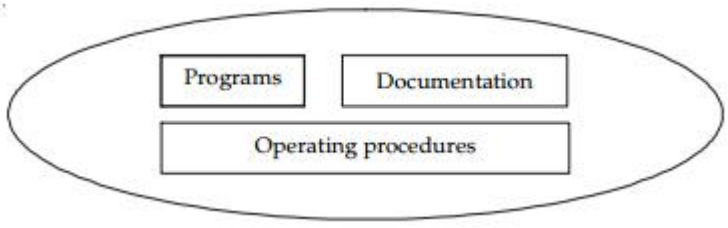
Maintainability: Refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors.

Portability: Refers to the ease with which software developers can transfer software from one platform to another, without (or with minimum) changes. In simple terms, it refers to the ability of software to function properly on different hardware and software platforms without making any changes in it.

Program vs. Software

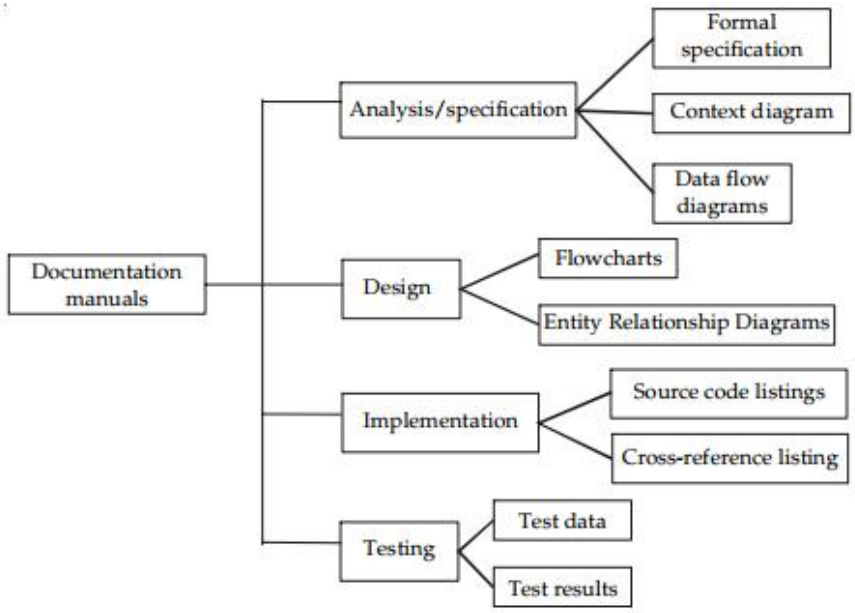
Software is more than programs. It comprises of programs, documentation to use these programs and the procedures that operate on the software systems.

Components of software



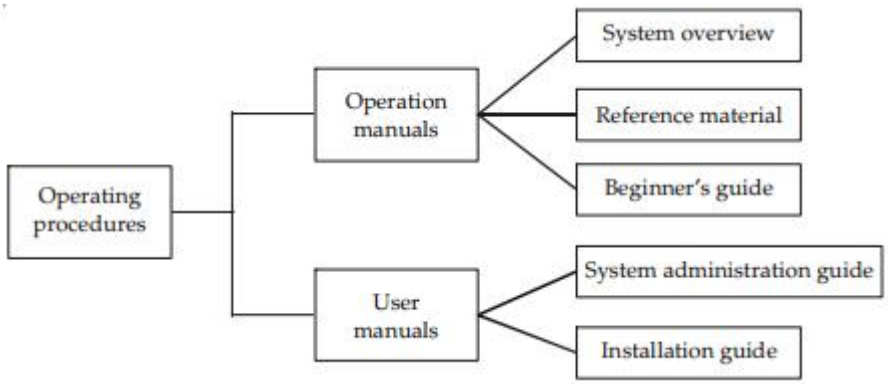
A program is a part of software and can be called software only if documentation and operating procedures are added to it. Program includes both source code and the object code.

List of Documentation Manuals



Operating procedures comprise of instructions required to setup and use the software and instructions on actions to be taken during failure. List of operating procedure manuals/ documents is given in Figure

List of Operating Procedure Manuals



1.4 Software Myths

Myth 1: Computers are more reliable than the devices they have replaced.

Considering the reusability of the software, it can undoubtedly be said that the software does not fail. However, certain areas which have been mechanized have now become prone to software errors as they were prone to human errors while they were manually performed.

Myth 2: Software is easy to change.

Yes, changes are easy to make – but hard to make without introducing errors. With every change the entire system must be re-verified.

Myth 3: If software development gets behind scheduled, adding more programmers will put the development back on track.

Software development is not a mechanistic process like manufacturing. In the words of

Brooks: "adding people to a late software project makes it later".

Myth 4: Testing software removes all the errors.

Testing ensures presence of errors and not absence. Our objective is to design test cases such that maximum number of errors is reported.

Myth 5: Software with more features is better software.

This is exactly the opposite of the truth. The best programs are those that do one kind of work.

Myth 6: Aim has now shifted to develop working programs.

The aim now is to deliver good quality and efficient programs rather than just delivering working programs. Programs delivered with high quality are maintainable.

The list is unending. These myths together with poor quality, increasing cost and delay in the software delivery have lead to the emergence of software engineering as a discipline.

1.5 Types of Myths

Software Myths: Software Myth beliefs about software and the process used to build it – can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious.

Management Myths: Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if the Belief will lessen the pressure.

Myth: We already have a book that's full of standards and procedures for building software.

Won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used?

- Are software practitioners aware of its existence?
- Does it reflect modern software engineering practice?
- Is it complete? Is it adaptable?
- Is it streamlined to improve time to delivery while still maintaining a focus on Quality?

In many cases, the answer to these entire questions is no:

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. In the words of Brooks: "Adding people to a late software project makes it later." At first, this statement may seem counter intuitive. However, 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.

Myth: If we decide to outsource the software project to a third party, I can just relaxand let that firm build it.

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

Customer Myths: A customer who requests computer software may be a person at the nextdesk, a technical group down the hall, the marketing/sales department, or an outsidecompany that has requested software under contract. In many cases, the customer believesmyths about software because software managers and practitioners do little to correctmisinformation.

Myths led to false expectations and ultimately, dissatisfaction with the developers.

Myth: A general statement of objectives is sufficient to begin writing programs we

can fill in details later.

Reality: Although a comprehensive and stable statement of requirements is not alwayspossible, an ambiguous statement of objectives is a recipe for disaster. Unambiguousrequirements are developed only through effective and continuous communicationbetween customer and developer.

Myth: Project requirements continually change, but change can be easilyaccommodated because software is flexible.

Reality: It's true that software requirement changes, but the impact of change varieswith the time at which it is introduced. When requirement changes are requestedearly, cost impact is relatively small.

However, as time passes, cost impact grows rapidly – resources have been committed, a designframework has been established, and change can cause upheaval that requires additional resourcesand major design modification.

1.6 Software Engineering Framework and Process

It is engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures.The outcome of software engineering is an efficient and reliable software product.

Need of Software Engineering

Large software

Adaptability

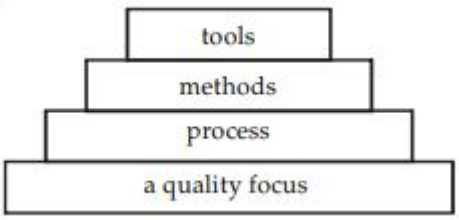
Cost

Dynamic Nature

Quality Management

Software Engineering has a three layered framework. The foundation for software engineering is the process layer. Software engineering process holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of Key Process Areas (KPA) that must be established for effective delivery of software engineering technology.

Software Engineering Framework



Software engineering methods provide the technical knowhow for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering tools provide automated or semi-automated support for the process and the methods.

Software Process

A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind.

In simpler words, when you build a product or system, it's important to go through a series of predictable steps – a road map that helps you create a timely, high quality result. The road map that you follow is called a software process.

In the last decade there has been a great deal of resources devoted to the definition, implementation, and improvement of software development processes.

ISO 9000

Software Process Improvement and Capability Determination (SPICE)

SEI Processes

Capability Maturity Model (CMM) for software

Personal Software Process (PSP)

Team Software Process (TSP)

A set of activities whose goal is the development or evolution of software

Generic activities in all software processes are:

Specification: what the system should do and its development constraints

Development: production of the software system

Validation: checking that the software is what the customer wants

Evolution: changing the software in response to changing demands.

Software Process Model

A simplified representation of a software process, presented from a specific perspective.

 *Example*: Process perspectives are:

Workflow perspective – sequence of activities

Data-flow perspective – information flow

Role/action perspective – who does what

Generic Process Models

Waterfall

Evolutionary development

Formal transformation

Integration from reusable components

Software Engineering Methods

Structured approaches to software development which include system models, notations, rules, design advice and process guidance

Rules Constraints applied to system models Notes

Recommendations – Advice on good design practice

Process guidance – What activities to follow

CASE (Computer-Aided Software Engineering)

Software systems which are intended to provide automated support for software process activities.

CASE systems are often used for method support:

Upper- CASE: Tools to support the early process activities of requirements and design.

Lower- CASE: Tools to support later activities such as programming, debugging and testing. Model descriptions – Descriptions of graphical models, which should be produced

Attributes of Good Software

The software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.

Maintainability: Software must evolve to meet changing needs.

Dependability: Software must be trustworthy.

Efficiency: Software should not make wasteful use of system resources.

Usability: Software must be usable by the users for which it was designed.

1.7 Software Engineering Practices

Software engineering practices are a set of approaches to software development.

- To overcome the software development problems.

- Understand the problem (communication and analysis)
- Plan the solution (Modeling and software design)
- Carry out the plan (code generation)
- Examine the result for accuracy (testing and quality assurance)

Understand the problem

- Who has a stake in the solution to the problem?
- What are unknowns?
- Can the problem be compartmentalized?
- Can the problem be represented graphically?

Develop Iteratively

- Critical risks are resolved before making large investments
- Initial iterations enable early user feedback
- Testing and integration are continuous
- Objective milestones provide short term focus

Manage Requirements

- Requirements are dynamic - expect them to change during software development
- User's own understanding of the requirements evolves over time
- Gain agreement with user on what the system should do and not how
- Maintain forward and backward traceability of requirements

Use Component Based Architecture

- Using components permits reuse
- Choice of thousands of commercially available components
- Improved maintainability and extensibility
- Promotes clean division of work among teams of developers

Plan the solution

- Have you seen similar problem before?
- Has a similar problem being solved?
- Can sub problems be defined?
- Can a solution be given for effective implementation?
- Can a design model be created?

Carry out the plan

- Does the solution conform the plan?
- Is the source code traceable to the design model
- Is each component solution correct?
- Have the design and code being reviewed?

Examine the result

- Is the each component part being tested?
- Does the solution produce the result?
- Has the software being evaluated against all stakeholder requirement?

Control Changes to Software

- Without explicit control parallel development degrades to chaos
- Decompose the architecture into subsystems and assign responsibility of each subsystem to a team. Establish secure workspaces for each team i.e. each team is isolated from changes made in other workspaces.
- Establish an enforceable change control mechanism where
- Change requests are prioritized
- Impact of the change request is assessed
- Plan put in place to introduce change in a particular iteration

1.8 Software Crisis

Size: Software is becoming larger and more complex with the growing complexity and expectations out of software. For example, the code in consumer products is doubling every couple of years.

Quality: Many software products have poor quality, i.e., the software produces defects after put into use due to ineffective testing techniques. For example, Software testing typically finds 25 defects per 1000 lines of code.

Cost: Software development is costly i.e., in terms of time taken to develop and the money involved. For example, Development of the FAA's Advance Automation System cost over $700 per line of code.

Delayed delivery: Serious schedule overruns are common. Very often the software takes longer than the estimated time to develop which in turn leads to cost shooting up. For example, one in four large-scale development projects are never completed.

Summary

- Software is more than programs. It comprises of programs, documentation to use these programs and the procedures that operate on the software systems.
- Any area that involves a sequence of steps (i.e. an algorithm) to generate output can make use of software engineering.
- The software myths together with poor quality, increasing cost and delay in the software delivery have lead to the emergence of software engineering as a discipline.
- Software engineering process holds the technology layers together and enables rational and timely development of computer software.
- The software engineering discipline has been faced with a number of challenges over the years, including those related to quality, management and under estimation.
- One of the challenge for the software engineering discipline is the increasing concern with respect to the design and management of software projects whose complexity is increasing exponentially.
- Legacy systems will also continue to present a considerable challenge to software engineering because the need to ensure that the old technologies are capable to co-exist with the new and vice-versa will be much greater in the near future

Keywords

Artificial Intelligence Software: AI software uses algorithms to solve problems that are not open to simple computing and analysis.

Determinate Program: A program that takes data in a predefined order, applies algorithms to it and generates results is called a determinate program.

Process: A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind.

Program: A program is a part of software and can be called software only if documentation and operating procedures are added to it.

Real-time Software: It refers to the software that controls the events and when they occur.

Software: It is defined as a discipline whose aim is the production of quality software, software that is delivered on time, within budget and that satisfies its requirements.

System Engineering: System engineering is an interdisciplinary field of engineering that focuses on the development and organization of complex artificial systems.

System Software: It is the software program that interacts with the hardware components so that other application software can run on it.

Self Assessment

1. Which one is system software?

A. MS office
B. Windows XP
C. Chrome
D. all of above

2. What are the software crises?

A. Quality
B. Size
C. Cost
D. All of above

3. Which one is application software _____

A. Sound drivers
B. Operating system
C. LAN drivers
D. Internet explorer

4. Software is _____

A. Documentation and configuration of data
B. Set of programs
C. Set of programs, documentation & configuration of data
D. None of the mentioned

5. Characteristics of a software is _____

A. Operational
B. Transitional
C. Maintenance
D. All of above

6. Software processes resources are_____

A. SPICE
B. CMM
C. PSP
D. all of above

7. Software process activities include

A. Quality
B. Size
C. Cost
D. All of above

8. The process of developing a software product using software engineering principles and methods is referred to as _____

A. Software Engineering
B. Software Evolution
C. System Models
D. Software Models

9. Software engineering practices are_____

A. Understand the problem
B. Plan the solution and Carry out the plan
C. Examine the result for accuracy
D. all of above

10. SDLC stands for

A. System Development Life cycle
B. Software Design Life Cycle
C. Software Development Life Cycle
D. System Design Life Cycle

Answer for Self Assessment

1.	B	2.	D	3.	C	4.	C	5.	A
6.	D	7.	D	8.	B	9.		10.	C

Review Question

1. "Software is designed and built so that it can be reused in different programs." Substantiate with suitable examples.
2. Suppose you are the software engineer of a modern and technically equipped company then explain how software delivers the most important product of our time—information.
3. Critically analyze the role of computer software. "Software has undergone significant change over a time span of little more than 50 years." Comment.
4. "The software differs from hardware as it is more logical in nature and hence, the difference in characteristics." Discuss.
5. Software is easy to change. It is myth? Explain why or why not? Explain with example.
6. Process defines a framework for a set of Key Process Areas (KPA) that must be established for effective delivery of software engineering technology. Analyze this statement.
7. Discuss the concept of software engineering framework.

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Links

- ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/softeng.pdf
- http://www0.cs.ucl.ac.uk/staff/A.Finkelstein/talks/10openchall.pdf
- http://thepiratebay.sx/torrent/8192581/Software_Engineering__A_
- Practitioner_s_Approach__7th_Ed._[PDF]
- http://www.etsf.eu/system/files/users/SottileF/XG_Basics_v2.pdf

Unit 02: Software Process Models

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of Processes and Models
- Explain the Software Process Model
- Characteristics of a Software Model
- Describe various process models
- Agile method of software development

Introduction

The concept of process is central to the software engineering method. "A particular technique of doing something, usually requiring a number of steps or operations," defines a process. The term "software process" refers to the way of developing software in software engineering. A software process is a set of operations that must be completed in order to create high-quality software. Is it possible to refer to the process as software engineering? Yes and no are the answers. The term "process" refers to the methods used in the development of software. Also included in the process are the technical procedures and tools used in software engineering.Software must be developed keeping in mind the demands of the end-user using a defined software process. In this unit, we will discuss the concept of software process models. We will also discuss different types of process models such as waterfall model, iterative model, etc.

2.1 Processes and Models

The software process describes how to manage and organise a software development project while taking limits and restrictions into account. A software process is a collection of actions linked by ordering constraints, with the goal of producing the intended output if the activities are completed correctly and in compliance with the ordering constraints. The goal is to produce high-quality software at a reasonable cost. Clearly, a method is not acceptable if it does not scale up and cannot handle massive software projects or produce high-quality software.

Many processes are usually running at the same time in large software development organisations. Many of these have little to do with software engineering, yet they do have an impact on software development. These process models could be classified as non-software engineering. Processes that fall under this include business process models, social process models, and training models. These processes have an impact on software development, although they are outside the scope of software engineering.

The process that deals with the technical and management issues of software development is called a software process. Clearly, many different types of activities need to be performed to develop software

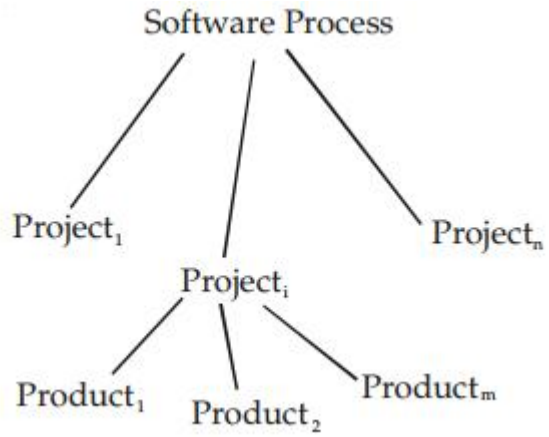
Because different types of activities are usually performed by different persons, it is better to think of the software process as a collection of component processes, each of which has a specific type of activity. Each of these component processes usually has a different goal in mind, while they obviously work together to achieve the overall software engineering goal.

Software Process framework is a set of guidelines, concepts and best practices that describes high level processes in software engineering. It does not talk about how these processes are carried out and in what order.

As previously stated, a software process defines a method for building software. A software project, on the other hand, is a development project that involves the usage of a software process. A software project's outputs are known as software products. Each software development project begins with a set of requirements and is expected to end with software that meets those requirements. A software process is an abstract set of operations that must be carried out in order to get from user requirements to the end result.

One can view the software process as an abstract type, and each project is done using that process as an instance of this type. In other words, there can be many projects for a process and there can be many products produced in a project. This relationship is shown in Figure

Relation between Processes, Projects and Products



The sequence of activities specified by the process is typically at an abstract level because they have to be usable for a wide range of projects. Hence, "implementing" them in a project is not straightforward.

Example: Let us take the example of book writing. A process for book writing on a subject will be something like this:

1. Set objectives of the book – audience, marketing etc.

2. Outline the contents.

3. Research the topics covered.

4. Do literature survey.

5. Write and/or compile the content.

6. Get the content evaluated by a team of experts.

7. Proof read the book.

8. Make corrections, if any.

9. Get the book typeset.

10. Print the book.

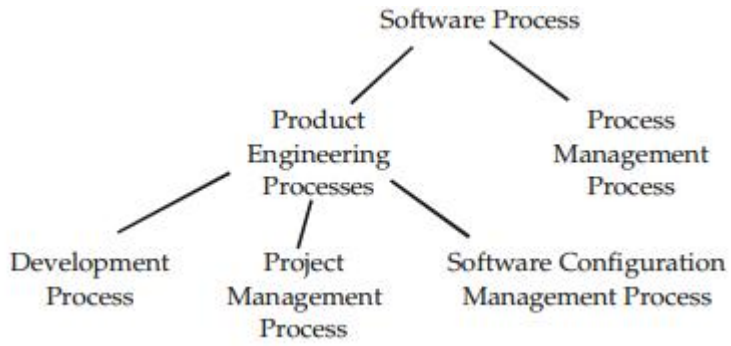
11. Bind the book.

Overall, the process specifies actions that are not project-specific at an abstract level. It's a generalised list of tasks that doesn't provide a precise roadmap for a specific project. The detailed roadmap for a project is the project plan that outlines what precise actions to undertake for this project, when to do them, and how to keep the project on track. The project plan for publishing a book on Software Engineering, in our example, will be a comprehensive marked map outlining the activities, as well as additional specifics such as plans for getting illustrations, photographs, and so on.

It should be obvious that a project is driven by its process. A method restricts a project's degrees of freedom by stating which activities must be completed and in what order. Furthermore, the project plan, which is itself within the process's bounds, specifies restrictions on the degrees of freedom for a specific project. To put it another way, a project plan cannot include an action that is not part of the process.

Because each project is an instance of the process it follows, it is fundamentally the process that determines the expected outcomes of a project. As a result, software engineering places a strong emphasis on the process.

Software Process



A software process is the set of activities and associated results that produce a software product. Software engineers mostly carry out these activities. There are four fundamental process activities, which are common to all software processes. These activities are:

1. Software Specification: The functionality of the software and constraints on its operation must be defined.

2. Software Development: The software to meet the specification must be produced.

3. Software Validation: The software must be validated to ensure that it does what the customer wants.

4. Software Evolution: The software must evolve to meet changing customer needs. Different software processes organize these activities in different ways and are described at different levels of detail. The timing of the activities varies as does the results of each activity.

Different organizations may use different processes to produce the same type of product. However, some processes are more suitable than others for some types of application. If an inappropriate

process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

Different software processes organize these activities in different ways and are described at different levels of detail. The timing of the activities varies as does the results of each activity. Different organizations may use different processes to produce the same type of product. However, some processes are more suitable than others for some types of application. If an inappropriate process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

2.2 The Software Process Model

A software process model is a simplified representation of a software process provided from a certain viewpoint. Because models are by definition simplifications, a software process model is an abstraction of the process being represented. Activities that are part of the software process, software products, and the responsibilities of persons involved in software engineering can all be included in process models.

 **Example**: Some examples of the types of software process model that may be produced are:

A Workflow Model: This shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in this model represent human actions.

A Dataflow or Activity Model: This represents the process as a set of activities each of which carries out some data transformation. It shows how the input to the process such as a specification is transformed to an output such as a design. The activities here may be at a lower level than activities in a workflow model. They may represent transformations carried out by people or by computers.

A Role/action Model: This represents the roles of the people involved in the software process and the activities for which they are responsible.

There are a number of different general models or paradigms of software development

The Waterfall Approach: This takes the above activities and represents 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 on to the following stage.

Evolutionary Development: This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system which satisfies the customer's needs. The system may then be delivered. Alternatively, it may be reimplemented using a more structured approach to produce a more robust and maintainable system.

Formal Transformation: This approach 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 program meets its specification.

System Assembly from Reusable Components: This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch.

2.3 Characteristics of a Software Model

When creating any form of software, the first question that comes to mind for each developer is, "What are the qualities that good software should have?" We'd want to state the basic expectations one has from any software before diving into technical qualities. First and foremost, a software solution must meet all of the customer's or end-needs. Users In addition, the software's development and maintenance costs should be modest. Software development should be finished within the stated time range.

Well these were the obvious things which are expected from any project (and software development is a project in itself). Now let's take a look at Software Quality factors. These set of

factors can be easily explained by Software Quality Triangle. The three characteristics of good application software are:

1. Operational Characteristics

2. Transition Characteristic

3. Revision Characteristics

Operational Characteristics of a Software

These are functionality-based factors and related to 'exterior quality' of software. Various Operational Characteristics of software are:

Correctness: The software which we are making should meet all the specifications stated by the customer.

Usability/Learnability: The amount of efforts or time required to learn how to use the software should be less. This makes the software user-friendly even for IT-illiterate people.

Integrity: Just like medicines have side-effects, in the same way software may have a side-effect i.e. it may affect the working of another application. But quality software should not have side effects.

Reliability: The software product should not have any defects. Not only this, it shouldn't fail while execution.

Efficiency: This characteristic relates to the way software uses the available resources. The software should make effective use of the storage space and execute command as per desired timing requirements.

Security: With the increase in security threats nowadays, this factor is gaining importance. The software shouldn't have ill effects on data/hardware. Proper measures should be taken to keep data secure from external threats.

Safety: The software should not be hazardous to the environment/life.

Revision Characteristics of Software

These engineering based factors of the relate to 'interior quality' of the software like efficiency, documentation and structure. These factors should be in-build in any good software. Various Revision Characteristics of software are:

Maintainability: Maintenance of the software should be easy for any kind of user.

Flexibility: Changes in the software should be easy to make.

Extensibility: It should be easy to increase the functions performed by it.

Scalability: It should be very easy to upgrade it for more work (or for more number of users).

Testability: Testing the software should be easy.

Modularity: Any software is said to made of units and modules which are independent of each other. These modules are then integrated to make the final software. If the software is divided into separate independent parts that can be modified, tested separately, it has high modularity.

Transition Characteristics of the Software

Interoperability: Interoperability is the ability of software to exchange information with other applications and make use of information transparently.

Reusability: If we are able to use the software code with some modifications for different purpose then we call software to be reusable.

Portability: The ability of software to perform same functions across all environments and platforms, demonstrate its portability.

Importance of any of these factors varies from application-to-application.

2.4 Software Development Life Cycle

The Software Development Life Cycle is process or method which is used to design, develop quality software. SDLC is a systematic process for building software that ensures the quality and correctness of the software built. SDLC process aims to produce high-quality software that meets customer expectations. The system development should be complete in the pre-defined time frame and cost. SDLC consists of a detailed plan which explains how to plan, build, and maintain specific software. Every phase of the SDLC life Cycle has its own process and deliverables that feed into the next phase.

ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

SDLC Phase

SDLC consist different phases every phase of the SDLC life Cycle has its own process/ activities and it is linked with next phase or it provides the input to the next phase.

Requirement analysis

Feasibility study

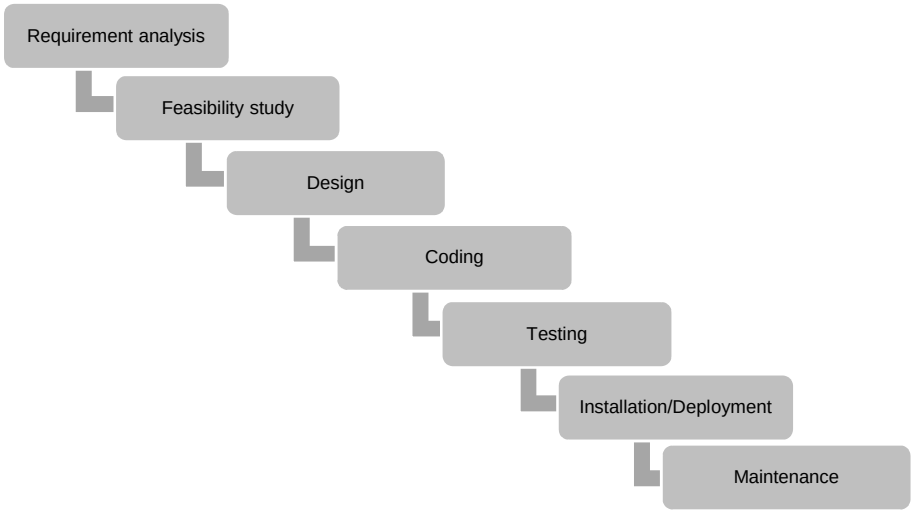
Design

Coding

Testing

Installation/Deployment

Maintenance



Requirement Analysis

Requirement Analysis is use to gather information from different stake holders e.g. sales department, market surveys and domain experts in the industry.This information is then used to plan the basic project approach and to conduct product feasibility study

Feasibility study

A feasibility study is simply an assessment of the practicality of a proposed plan or method.

Economic

Time

Operation feasibility

Technical

Legal

Design

The design phase helps define overall system architecture.In design phase documents are prepared as per the requirement specification document and feasibility study.

Coding

In this phase developers start build the entire system by writing code using the chosen programming language/ platform.In this phase tools use programming tools like compiler, interpreters, debugger to generate and implement the code

Testing

This phase is used to verify that the entire application works according to the customer requirement by using different testing techniques.

Unit Testing

Integration Testing

Regression Testing

Black box testing

White box testing

Beta Testing

Installation/Deployment

In this phase software is deploy to the production environment so users can start using the product.Based on the feedback given by the user's manager, the final software is released and checked for deployment issues.This phase allows users to safely play with the product before releasing it to the market.

Maintenance

This phase deal with the real world changing conditions, we need to update and advance the software to match.

Bug fixing

Upgrade

Enhancement

2.5 Prototype Model

It is software development model in which prototype model is built, tested, and reworked until an acceptable prototype is achieved.It is a demo implementation of the actual product or system.

Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. A prototype model usually exhibits limited functional capabilities, low reliability, and inefficient performance as compared to the actual software.

It is suitable where the project's requirements are not known in detail to customer. It is an iterative, trial and error method which takes place between developer and client.

Prototype model phase

Requirements gathering and analysis

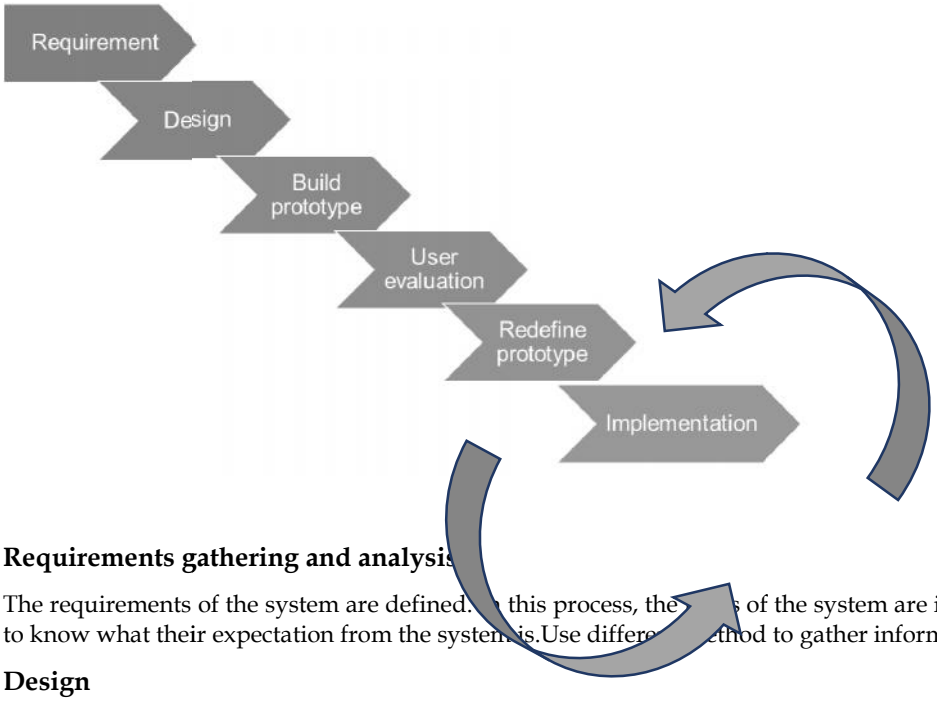
Design

Build prototype

User evaluation

Refining prototype

Implementation and Maintenance



Requirements gathering and analysis

The requirements of the system are defined. In this process, the users of the system are interviewed to know what their expectation from the system is.Use different method to gather information.

Design

In this phase, a simple design of the system is built.It gives a brief idea of the system to the user

Build prototype

In this phase the initial Prototype is developed, where the basic requirements are showcased and user interfaces are provided. The actual prototype is designed based on the information gathered from design phase

User evaluation

In this phase prototype developed is presented to the customer and the other important stakeholders in the project for an initial evaluation. The feedback is collected and used for further changes in the product under development.This phase help to find out the strength and weakness of the working model

Refining prototype

In this phase, Based upon customer feedback negotiations happen on factors like – time and budget constraints and technical feasibility of the actual implementation.Changes are accepted and incorporated in the new Prototype model and the cycle repeats until the customer expectations are met.

Implementation and Maintenance

After development of final system based on the final prototype, it is tested and deployed to production. The system undergoes routine maintenance and updates based upon real time environment changes for minimizing downtime and prevent large-scale failures and for.

Types of Prototyping Models

Rapid Throwaway prototypes

This technique offers a useful method of exploring ideas and getting customer feedback for each of them. In this method, a developed prototype need not necessarily be a part of the ultimately accepted prototype. Customer feedback helps in preventing unnecessary design faults and hence, the final prototype developed is of better quality.

Evolutionary prototype

The prototype developed is incrementally refined based on customer's feedback until it is finally accepted. It helps you to save time as well as effort. That's because developing a prototype from scratch for every interaction of the process can sometimes be very frustrating.

Incremental Prototype

In incremental Prototyping, the final product is decimated into different small prototypes and developed individually. Eventually, the different prototypes are merged into a single product. This method is helpful to reduce the feedback time between the user and the application development team.

Extreme Prototype

Extreme prototyping method is mostly used for web development. It is consists of three sequential phases.

- Basic prototype with all the existing page is present in the HTML format.
- You can simulate data process using a prototype services layer.
- The services are implemented and integrated into the final prototype.

Advantages of prototype model

- The Users involved in development. Errors can be detected in the initial stage of the software development process.
- Suitable where requirement are changing.
- Reduce Maintenance cost

Disadvantages of prototype model

- It is a slow and time taking process
- Cost increases with respect to time.
- Customer involvement is higher so it requirement changes are very dynamic and it impact overall product development

2.6 Water Fall Model

The waterfall model derives its name due to the cascading effect from one phase to the other as is illustrated in the figure below. In this model each phase is well defined, has a starting and ending point, with identifiable deliveries to the next phase.

- It is Linear sequential model
- Introduced by Winston W. Royce in 1970.
- The waterfall model is the classic lifecycle model.
- Whole system is developed in a sequential approach.
- Each phase is completed fully before the next begin.

Stages of a prototype model are:

Requirement analysis

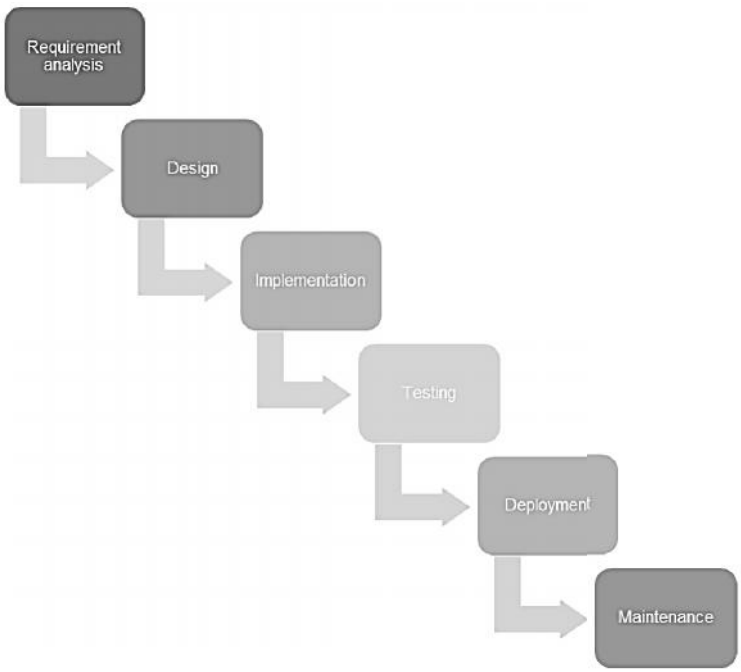
Design

Implementation

Testing

Deployment

Maintenance



Requirement analysis

- In this stage to understand the requirements of the system to be developed
- Software definition to be detailed and accurate with no ambiguities
- Documented requirements and reviewed.
- Software Requirement Specification (SRS) document is created

Design

In this phase the requirements gathered in the SRS transform into a suitable form for coding in a programming language or selection of platform.It helps in defining the overall system architecture.

Implementation

In this phase software design is translated into source code using any suitable programming language.

Testing

In this phase code is thoroughly examined based upon different testing methods.

- Unit testing
- Integration testing
- Alpha testing
- Beta testing

Deployment

After testing is done, the product is deployed in the customer environment or released into the market.

Maintenance

- It is performed by every user once the software has been delivered to the customer, installed, and operational as per customer request
- Meet the changing customer needs
- Adapted to accommodate changes in the external environment

Advantages

- Simple and easy to understand and use
- Phases in this model are processed one at a time
- Suited for smaller projects where requirements are well defined
- Clearly defined phase.

- Process and results are well documented

Disadvantages

- All requirements must be known upfront
- Not suitable where requirements are changing frequently
- Not a good model for complex and object-oriented projects.
- High amounts of risk and uncertainty
- It is difficult to measure progress within stages.

2.7 V model

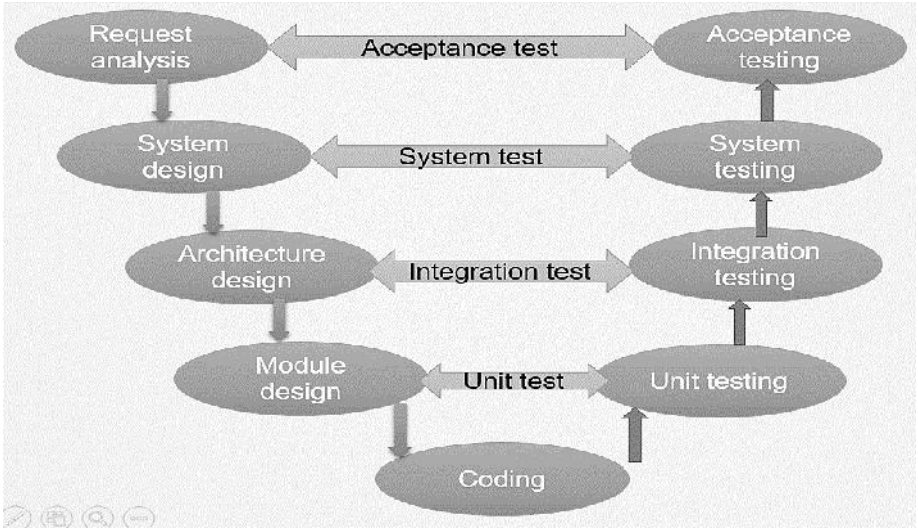
The V model is type of SDLC model. Testing phase parallel to each development phase, Process executes in a sequential manner in V-shape. It is also known as Verification and Validation model

The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.

Verification and validation

Verification: Verification is a static analysis technique. In this technique, testing is done without executing the code. Examples include – Reviews, InspectionIt is the process of evaluation of the product development phase to find whether specified requirements meet.

Validation: Validation is a dynamic analysis technique where testing is done by executing the code. Examples include functional and non-functional testing techniques.



Design phase

Requirement Analysis

System design

Architectural Design

Module design

Testing phase

Unit testing

Integration testing

System testing

User acceptance testing

Requirement Analysis

In this stage to understand the requirements of the system to be developed, this phase involves detailed communication with the customer and domain experts.Different method are used to gather information

System design

In this phase, a simple design of the system is built as per requirement gathered in previous phase.The system design will have the understanding and detailing the complete hardware and communication setup for the product under development.

Architectural Design

In architecture design it should understand all modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, and technology detail

The system design is broken down further into modules taking up different functionality. This is also referred to as High Level Design (HLD).

Module design

In this phase, the system breaks down into small modules. The detailed internal design for all the system modules is specified, referred to as Low Level Design (LLD).In this phase design is checked for its compatibility with the other modules in the system architecture and the other external systems.

Coding

In this phase perform task based upon design phase inputs and follow guidelines for coding, Selection of programming language or platform.

Unit testing

In this phase Unit Test Plans (UTPs) are developed during the module design phase. These UTPs are executed to eliminate errors at code level or unit level. A unit is the smallest entity which can independently exist

Integration testing

In this phase modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. It verifies the communication of modules among themselves.

System testing

In this phase complete application is checked with its functionality, inter dependency, and communication. It tests the functional and non-functional requirements of the developed application.It is associated with the system design phase.

Acceptance testing

It is associated with the requirement analysis phase and involves testing the product in user environment.It uncovers the compatibility issues with the other systems available in the user environment.

Advantages

- This model focuses on verification and validation activities early in the life cycle
- Avoids the downward flow of the defects.
- This is a disciplined model and Phases are completed one at a time.
- Useful in small projects where requirements are clear

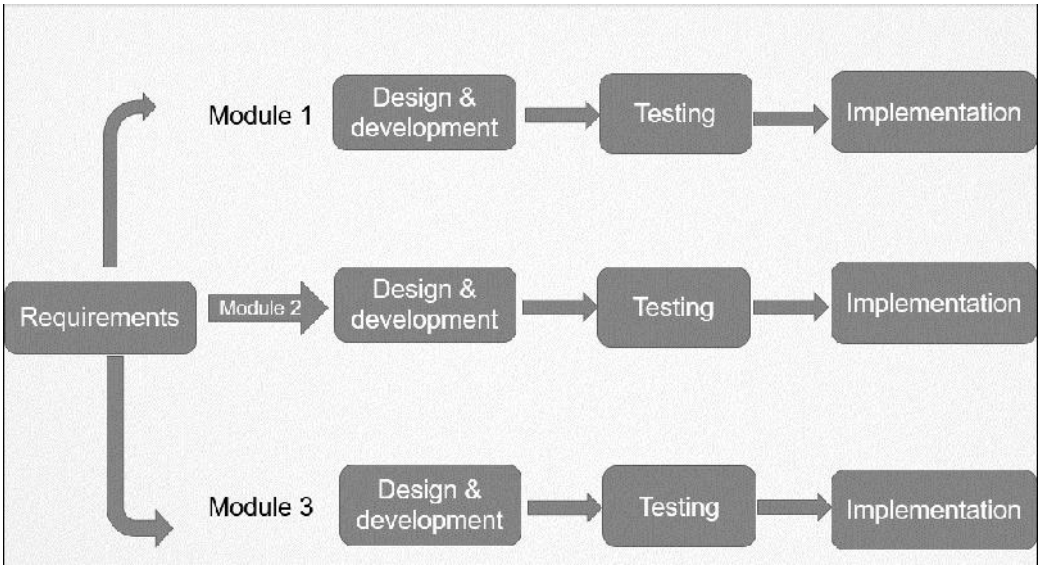
Disadvantages

- It is not suitable for projects where requirements are more dynamic and contains high risk of changing
- Not suitable for complex projects.

2.8 Incremental Model

In Incremental Model requirements are broken down into multiple standalone modules of software development cycle.Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle.

Each iteration passes through the requirements, design, coding and testing phases. After completion of one module the system adds function to the previous release until all designed functionality has been implemented.



Incremental Model Phase

- Requirement analysis
- Design & Development
- Testing
- Implementation

Requirement analysis

In this phase requirements are gathered and analyzed using different techniques and tools.This phase involves detailed communication with the customer and domain experts.

Design and Development

In this phase the design of the system functionality and the development method are finished with success. When software develops new practicality, the incremental model uses style and development phase.

Testing

In this phase the testing phase checks the performance of each existing function as well as additional functionality. Different methods and tools are used to test the behavior of each task.

Implementation

In this phase final coding is done based upon designing and development phase and tests the functionality in the testing phase. After completion of this phase, the number of the product working is enhanced and upgraded up to the final system product.

Characteristics of incremental model

- System development is broken down into many mini development projects
- Partial systems are successively built to produce a final total system
- Highest priority requirement is tackled first
- Once the incremented portion id developed, requirements for that increment are frozen

Advantages

- Easy to Identify error
- Flexibility is high, less expensive to change requirements and scope
- Easier to test and debug
- Design and development is easy as modules are handled individually.

Disadvantages

- Good planning is required
- Problems might cause due to all requirements are handled in different modules.
- Rectifying a problem in one unit requires correction in all the units and consumes a lot of time

2.9 Agile Method of Software Development

Agile Methodology

Agility is the ability to both create and respond to change in order to profit in a turbulent business environment. It is a practice that promotes continuous iteration of development and testing throughout the software development lifecycle of the project.

Agility is the ability to both create and respond to change in order to profit in a turbulent business environment. It is a practice that promotes continuous iteration of development and testing throughout the software development lifecycle of the project.

Agility is not merely reaction, but also action. It is ability to react, to respond quickly and effectively to both anticipated and unanticipated changes in the business environment.Agility means being responsive or flexible within a defined context.

Agile vs. Traditional Models

Agile model is based on the adaptive software development methods.Traditional SDLC models like the waterfall model are based on a predictive approach.

Predictive methods entirely depend on the requirement analysis and planning done in the beginning of cycle.Agile uses an adaptive approach where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed.

Agile Process Model

It is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product.Agile Methods break the product into small incremental builds. These builds are provided in iterations.

Agile process model can be implemented with the help of various methods:

- Extreme Programming
- DSDM
- Essential Unified Process
- Agile Data Method
- Agile Modeling
- Agile Unified Process

Advantages of the Agile Models

- Promotes teamwork and cross training.
- Realistic approach to software development.
- Functionality can be developed rapidly and demonstrated.
- Resource requirements are minimum.
- Suitable for fixed or changing requirements
- Delivers early partial working solutions.

Disadvantages of the Agile Models

- Not suitable for complex problem
- More risk of sustainability, maintainability and extensibility.
- Depends heavily on customer interaction.

- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.

Summary

- A software process is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced.
- Software Process framework is a set of guidelines, concepts and best practices that describes high level processes in software engineering.
- The sequence of activities specified by the process is typically at an abstract level because they have to be usable for a wide range of projects.
- A software process model is a simplified description of a software process, which is Notes presented from a particular perspective.
- In waterfall model each phase is well defined, has a starting and ending point, with identifiable deliveries to the next phase.
- In a prototype model, a working prototype is built with the available set of requirements such that it has limited functionalities, low reliability and performance.
- An iterative lifecycle model does not attempt to start with a full specification of requirements.

Keywords

Workflow Model: This shows the sequence of activities in the process along with their inputs, outputs and dependencies.

Dataflow or Activity Model: This represents the process as a set of activities each of which carries out some data transformation.

Design Phase: A Design phase is a phase in which a software solution to meet the requirements is designed.

Prototype Model: It is a model in which a working prototype is built with the available set of requirements such that it has limited functionalities, low reliability and performance.

Requirements Phase: A Requirements phase is a phase in which the requirements for the software are gathered and analyzed.

Software Processes: Software processes are the activities involved in producing and evolving a software system.

Software Process Framework: It is a set of guidelines, concepts and best practices that describes high level processes in software engineering.

Software Process Model: A software process model is a simplified description of a software process, which is presented from a particular perspective

Self Assessment

1. Which is not a SDLC phase?

A. Design
B. Coding
C. Testing
D. Coupling

2. Which is not a part of feasibility study?

A. Time
B. Operational
C. Module
D. Cost

3. Which is not a testing type?

A. Unit
B. Beta
C. Cohesion
D. Black box

4. Prototype model is____

A. Suitable where the project's requirements are not known in detail to customer
B. It is a demo implementation of the actual product or system
C. Used to allow the users evaluate developer proposals
D. All of above

5. Which is not a part of prototype model phases?

A. Requirement
B. Design
C. Process
D. Implementation

6. Which is not a prototype model type?

A. Evolutionary
B. Beta
C. Incremental
D. Extreme

7. Waterfall model is____

A. It is linear sequential model
B. It is the classic lifecycle model
C. Introduced by Winston W. Royce in 1970
D. All of above

8. Stages in Waterfall model are_____

A. Design
B. Implementation
C. Testing and Deployment
D. All of above

9. Information is gathered in _____ phase

A. Design
B. Coding
C. Requirement analysis
D. Maintenance

10. Incremental Model is____

A. Broken down into multiple standalone modules of software development cycle.
B. Multiple development cycles take place
C. Each iteration passes through the requirements, design, coding and testing phases
D. All of above

Answer for Self Assessment

1.	D	2.	C	3.	C	4.	D	5.	C
6.	B	7.	D	8.	D	9.	C	10.	D

Review Questions

1. Explain the concept of software processes.
2. Describe the relation between Processes, Projects and Products.
3. Explain the concept of software process model with examples.
4. Discuss the number of different general models or paradigms of software development.
5. Discuss the characteristics of a Software Model.
6. Explain the working of waterfall model.
7. Describe the phases included in iterative model.
8. What is Agile Methodology?
9. Name few Agile Process Model.

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education.

Web Links

- http://www.ijcsi.org/papers/7-5-94-101.pdf
- http://people.sju.edu/~jhodgson/se/models.html
- http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SEEncyc.pdf
- https://www.tutorialspoint.com/
- https://www.guru99.com/

Unit 03: Requirement Engineering

CONTENTS

Objective

After studying this unit, you will be able to:

- Describe requirement engineering task
- Explain requirement management
- Explain requirement analysis and stakeholder analysis

Introduction

Requirements define the "what" of a system rather than the "how." Understanding what the customer wants, analyzing the need, determining feasibility, negotiating a reasonable solution, clearly specifying the solution, validating the specifications, and managing the requirements as they are transformed into a working system are all covered by requirements engineering. As a result, requirements engineering is the systematic use of well-established ideas, methodologies, tools, and notations to characterize a proposed system's expected behavior and restrictions.

In the engineering design process, requirements engineering refers to the process of identifying, documenting, and managing requirements. Understanding what the customer wants, analysing the need, determining feasibility, negotiating a reasonable solution, clearly specifying the solution, validating the specifications, and managing the requirements as they are transformed into a working system are all covered by requirement engineering. As a result, requirement engineering is the systematic use of well-established ideas, methodologies, tools, and notation to specify a proposed system's expected behaviour and restrictions.

It is process to gather, defining, documenting, and maintaining the software requirements from client, analyze and document them is known as requirement engineering.Requirement engineering provides the appropriate mechanism to understand what the customer desires, analyzing the need, and assessing feasibility, negotiating a reasonable solution.

3.1 Requirement Engineering

The requirements engineering includes the following activities:

- Identification and documentation of customer and user's needs.
- Creation of a document describing the external behavior and associated constraints that will satisfy those needs.
- Analysis and validation of the requirements document to ensure consistency, completeness and feasibility.
- Evolution of needs.

The fundamental output of requirements engineering is the specification of requirements. It is a system requirement if it describes both hardware and software, and a software requirements specification if it solely provides software needs. The system is viewed as a black box in both of these scenarios.

Requirements Specification

A text, a graphical model, a prototype, or a combination of these can be used to specify requirements. Depending on the system being created, a standard template or a flexible methodology can be used to elicit system specifications.

As a result of system and requirements engineering, a system specification is created. It offers information about the hardware, software, and database, among other things. It outlines the function as well as the constraints that must be met while creating a system. It also provides information about the system's input and output.

Requirements Validation

During the validation process, the work product created as a result of requirements engineering is evaluated for quality. Validation of requirements guarantees that all requirements have been written accurately, that inconsistencies and errors have been identified and addressed, and that the work result meets the process, project, and product standards.

Although the requirements validation can be done in any way that leads to discovery of errors, the requirements can be examined against a checklist. Some of the checklist questions can be: Are the requirements clear or they can be misinterpreted?

- Is the source of the requirements identified?
- Has the final statement of requirement been verified against the source?
- Is the requirement quantitatively bounded?
- Is the requirement testable?
- Does the requirement violate domain constraints?
- These questions and their likes ensure that validation team does everything possible for a thorough review.

Requirements Management

Throughout the life of a computer-based system, requirements may alter. The operations that assist a project team in identifying, controlling, and tracking requirements and modifications to these requirements at any point during the project are referred to as requirements management. Identification is the first step in managing requirements. Traceability tables are developed once the criteria have been identified. Each criterion is linked to one or more aspects of the system or its surroundings in this table. They may be used as a requirements database to understand how a change in one requirement affects other components of the system under construction.

A Brigade of Design & Construction

Any engineering activity's eventual purpose is to provide some form of documentation. The design documentation is supplied to the manufacturing team when a design effort is completed. This is a different group of people with different abilities than the design team. The production team can proceed to produce the product if the design documentation actually represent a complete design. In fact, they can develop a large portion of the product without the help of the designers. After

evaluating the software development life cycle today, it appears that the source code listings are the only software documentation that appears to meet the criteria of an engineering design.

Software runs on computers. It is a sequence of ones and zeros. Software is not a program listing. A program listing is a document that represents a software design. Compilers, linkers, and interpreters actually build (manufacture) software. Software is very cheap to build. You just press a button. Software is expensive to design because it is complicated and all phases of the development cycle are part of the design process. Programming is a design activity; a good software design process recognizes this and does not hesitate to code when coding makes sense.

More often than you might think, coding makes sense. Often, the process of coding the design reveals flaws and the need for additional design work. It is preferable if this occurs sooner rather than later. Testing and debugging are design tasks, and they are the engineering equivalents of design validation and refinement. They can't be taken advantage of. Because it is cheaper to design software and test it than to prove it, formal validation methods aren't very useful.

Think about it. When you are programming, you are doing detailed design. The manufacturing team for software is your compiler or interpreter. The source is the only complete specification of what the software will do.

Design is present whenever objects are developed for humans to utilize. Design can be done systematically or haphazardly, intentionally or unconsciously. However, when people create software – or any other product – they make decisions and build objects with the purpose of knowing what those objects will do and how they will be viewed and used..

The education of computer professionals has often concentrated on the understanding of computational mechanisms, and on engineering methods that are intended to ensure that the mechanisms behave as the programmer intends. The focus is on the objects being designed: the hardware and software. The primary concern is to implement a specified functionality efficiently.

When software engineers or programmers say that a piece of software works, they typically mean that it is robust, is reliable, and meets its functional specification. These concerns are indeed important. Any designer who ignores them does so at the risk of disaster.

But this inward-looking perspective, with its focus on function and construction, is one-sided.To design software that really works, we need to move from a constructor's-eye view to adesigner's-eye view, taking the system, the users, and the context all together as a starting point.

3.2 Requirements Engineering Tasks

The processes used for RE vary widely depending on the application domain, the people involved and the organization developing the requirements.

However, there are a number of generic activities common to most processes:

Inception

Requirements elicitation

Negotiation

Requirements specification

Requirements validation

Requirement Management

Requirements Reviews/Inspections

Regular reviews should be held while the requirements definition is being formulated. Both client and contractor staff should be involved in reviews. (Stakeholders) Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review Check-list

- Verifiability: Is the requirement realistically testable?
- Comprehensibility: Is the requirement properly understood?

- Traceability: Is the origin of the requirement clearly stated?
- Adaptability: Can the requirement be changed with minimum impact on other Notes requirements? (Especially when change is anticipated!)

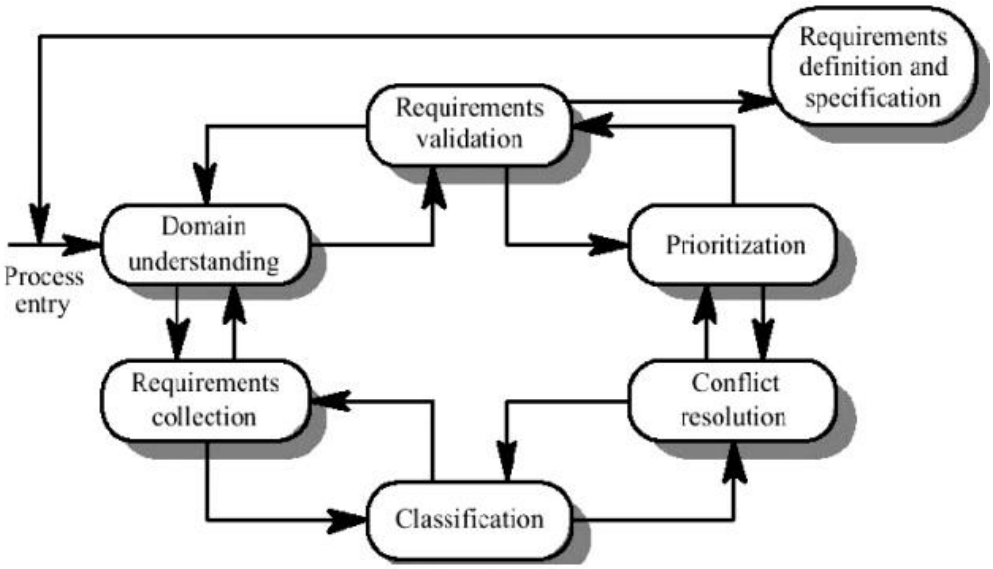
Elicitation

Involves working with customers to learn about the application domain, the services needed and the system's operational constraints.May also involve end-users, managers, maintenance personnel, domain experts, trade unions, etc. (That is, any stakeholders.)

Problems of Elicitation and Analysis

- Getting all, and only, the right people involved.
- Stakeholders often don't know what they really want ("I'll know when I see it").
- Stakeholders express requirements in their own terms.
- Stakeholders may have conflicting requirements.
- Requirements change during the analysis process.
- Organizational and political factors may influence the system requirements.

The Elicitation and Analysis Process



Viewpoint-oriented Elicitation

Stakeholders represent different ways of looking at a problem (viewpoints). A multi-perspective analysis is important, as there is no single correct way to analyze system requirements, provides a natural way to structure the elicitation process.

Types of Viewpoints: Data sources or sinks: Viewpoints are responsible for producing or consuming data.Analysis involves checking that assumptions about sources and sinks are valid.

Representation frameworks: Viewpoints represented by different system models (i.e., dataflow, ER, finite state machine, etc.). Each model yields different insights into the system.

Receivers of services: Viewpoints are external to the system and receive services from it, Natural to think of end-users as external service receivers.

Method-based RE

"Structured methods" to elicit, analyse, and document requirements.

Examples include:

Ross' Structured Analysis (SA),

Volere Requirements Process (www.volere.co.uk).

Knowledge Acquisition and Sharing for Requirement Engineering (KARE)(www.kare.org),

Somerville's Viewpoint-Oriented Requirements Definition (VORD), andThe baut's Scenario-Based Requirements Engineering (SBRE)˜

Scenarios

Depict examples or scripts of possible system behavior.People often relate to these more readily than to abstract statements of requirements, particularly useful in dealing with fragmentary, incomplete, or conflicting requirements.

Scenario Descriptions

System state at the beginning of the scenario, Sequence of events for a specific case of some generic task the system is required to accomplish.Any relevant concurrent activitiesSystem state at the completion of the scenario.

Scenario-Based Requirements Engineering (SBRE)

Marcel support environment allows rapid construction of an operational specification of the desired system and its environment.Based on a forward chaining rule-based language, an interpreter executes the specification to produce natural language based scenarios of system behavior.

UML Use-cases and Sequence Diagrams

Use-cases are a graphical notation for representing abstract scenarios in the UML.They identify the actors in an interaction and describe the interaction itself.A set of use-cases should describe all types of interactions with the system.Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing.

Social and Organizational Factors

All software systems are used in a social and organizational context. This can influence or even dominate the system requirements.Good analysts must be sensitive to these factors, but there is currently no systematic way to tackle their analysis.

Ethnography

Social scientist spends considerable time observing and analyzing how people actually work.People do not have to explain or articulate what they do.Social and organizational factors of importance may be observed.Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Focused Ethnography

Developed during a project studying the air traffic control process Combines ethnography with prototyping.Prototype development raises issues, which focus the ethnographic analysis.Problem with ethnography alone: it studies existing practices, which may not be relevant when a new system is put into place.

3.3 Types of Requirement

User requirements

Statements in natural language plus diagrams of the services the system provides and its operational constraints Written for customers.

System requirements

A structured document setting out detailed descriptions of the system's functions, services and operational constraints, Defines what should be implemented so may be part of a contract between client and contractor.

Functional requirements

Statements of services the system should provide how the system should react to particular inputs and how the system should behave in particular situations.May state what the system should not

do.Describe functionality or system services.Depend on the type of software, expected users and the type of system where the software is used.Functional user requirements may be high level statements of what the system should do.Functional system requirements should describe the system services in detail.

Non-functional requirement

Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.Often apply to the system as a whole rather than individual features or services.

These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.Process requirements may also be specified mandating a particular IDE, programming language or development method.

Types of non-functional requirement

Product requirement

Efficiency	Dependability
Security	Usability

Organizational requirement

Operational	Development

External requirements

Regulatory	Ethical

Non-functional classifications

Product requirements

Requirements which specify that the delivered product must behave in particular way e.g. execution speed reliability, etc.

Organizational requirements

Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.

External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

3.4 Requirement Negotiation

After they have been collected, requirements must be analyzed to obtain a satisfactory understanding of the customer's need and negotiated to establish an agreed set of consistent (unambiguous, correct, complete, etc.) requirements.

The requirements negotiation process is expensive and time-consuming, requiring very experienced personnel exercising considerable judgment. This is exacerbated by the fact that the requirements negotiation process is not sufficiently structured that an automated requirements negotiation methodology would be appropriate.

Requirement Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. Typical physical systems have a relatively small number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required.

There are some approaches to requirements specification:

- Natural language
- Structured natural language
- Design description language
- Requirements specification language
- Graphical notation
- Formal specification

3.5 Requirements Validation

Concerned with whether or not the requirements define a system that the customer really wants.

Requirements error costs are high, so validation is very important. (Fixing a requirements error after delivery may cost up to 100 times that of fixing an error during implementation.)

Requirements Checking

Validity: Does the system provide the functions which best support the customer's needs?

Consistency: Are there any requirements conflicts?

Completeness: Are all functions required by the customer included?

Realism: Can the requirements be implemented given available budget and technology?

Verifiability: Can the requirements be tested?

Requirements Validation Techniques

Requirements reviews/inspections: systematic manual analysis of the requirements.

Prototyping: using an executable model of the system to check requirements.

Test-case generation: developing tests for requirements to check testability.

Automated consistency analysis: checking the consistency of a structured requirements description.

3.6 Requirements Management

Requirements management is the process of managing changing requirements during the requirements engineering process and system development.New requirements emerge during the process as business needs change and a better understanding of the system is developed.The priority of requirements from different viewpoints changes during the development process, the business and technical environment of the system changes during its development.

Enduring and Volatile Requirements

Enduring requirements: Stable requirements derived from the core activity of the customer organization. For example, a hospital will always have doctors, nurses, etc. May be derived from domain models, Volatile requirements: Requirements which change during development or when the system is in use. E.g., requirements derived from the latest health-care policy.

Classification of Requirements

Mutable requirements: those that change due to changes in the system's environment.Emergent requirements: those that emerge as understanding of the system develop.Consequential requirements: those that result from the introduction of the system.Compatibility requirements: those that depend on other systems or organizational processes.

Requirements Management Planning

During requirements management planning, you must decide on:Requirements identification: How requirements will be individually identified.A change management process: A process to be followed when analyzing the impact and costs of a requirements change.Traceability policies: The amount of information about requirements relationships that is maintained.CASE tool support: The tool support required to help manage requirements change.

Traceability

Traceability is concerned with the relationships between requirements, their sources, and the system design.

Source traceability – links from requirements to stakeholders who proposed these requirements

Requirements traceability – links between dependent requirements.

Design traceability – links from the requirements to the design.

CASE Tool Support

Requirements storage – requirements should be managed in a secure, managed data store.

Change management – the process of change management is a workflow process whose stages can be defined and information flow between the stages partially automated.

Traceability management – automated discovery and documentation of relationships between requirements

Requirements change management should apply to all proposed changes to the requirements.

Principal Stages

Problem analysis – discuss identified requirements problem and propose specific changes.

Change analysis and costing – assess effects of change on other requirements.

Change implementation – modify requirements document and others to reflect change.

Summary

The requirements engineering process includes feasibility study, elicitation and analysis, specification, and validation.

Requirements analysis is an iterative process involving domain understanding, requirements collection, classification, structuring, prioritization and validation.

Systems have multiple stakeholders with different viewpoints and requirements. Social and organization factors influence system requirements.

Requirements validation is concerned with checks for validity, consistency, completeness, realism, and verifiability.

Business, organizational, and technical changes inevitably lead to changing requirements.

Requirements management involves careful planning and a change management process.

Keywords

KARE: Knowledge Acquisition and Sharing for Requirement Engineering

SA: Structured Analysis

SBRE: Scenario-Based Requirements Engineering

VORD: Viewpoint-Oriented Requirements Definition

Self Assessment

1. Software requirement gathering from client, analyze and document is known as

A. Requirement Gathering
B. Requirement Engineering
C. Feasibility Study
D. System Requirements Specification

2. The objective of requirement engineering is to develop and maintain_____ document.

A. Feasibility Study
B. System Requirements Specification
C. Requirement Gathering
D. Software Requirement Validation

3. Which one is a correct requirement type?

A. Reliability
B. Availability
C. Usability
D. All of the above

4. Which is not step of requirement engineering?

A. Elicitation
B. Documentation
C. Design
D. Analysis

5. What are the activities involve in requirement gathering?

A. Interview
B. Survey
C. Group discussion
D. All of above

6. Product requirement include_____

A. Efficiency
B. Operational
C. Development
D. All of above

7. What are the types of requirements?

A. Functional
B. Non –functional
C. System

D. All of above

8. Requirement Engineering process activities are_

A. Feasibility Study
B. Requirement Elicitation and Analysis
C. Software Requirement Validation
D. All of above

9. Economic Feasibility deals with.

A. Technical activity
B. Operational activity
C. Cost activity
D. None of above

10. Requirement Elicitation activities are____

A. Organizing Requirements
B. Negotiation & discussion
C. Documentation
D. All of the above

11. Which is not Requirement Elicitation Techniques?

A. Surveys
B. Questionnaires
C. Design
D. Task analysis

12. Requirements Validation Techniques are____

A. Requirements reviews/inspections
B. Prototyping
C. Test-case generation
D. All of above

13. How the interviews held between two persons across the table is___

A. Non-structured
B. One-to-one
C. Written
D. Group

14. Which is not a feasibility study?

A. Operational
B. Finance

C. Modular

D. Social

Answer for Self Assessment

1.	B	2.	B	3.	D	4.	C	5.	D
6.	A	7.	D	8.	D	9.	C	10.	D
11.	C	12.	D	13.	B	14.	C		

Review Questions

1. What are the current process problems and how would the system help with these?
2. Is new technology needed new skills?
3. What must be supported by the system and what had not be supported?
4. Requirements may change throughout the life of a computer based system. Explain.
5. Requirements management is the process of managing changing requirements during the requirements engineering process and system development. Discuss.
6. What is significance of requirement engineering?

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, Second Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering a Practioner's Approach, 5th edition, Tata McGraw
- Hill Higher education.
- Sommerville, Software Engineering, Sixth edition, Pearson Education.

Web Links

- http://ksi.cpsc.ucalgary.ca/KAW/KAW96/herlea/FINAL.html
- http://en.wikipedia.org/wiki/Requirements_analysis
- https://www.javatpoint.com/

Unit 04: Requirement Specification

CONTENTS

Objective

After studying this unit, you will be able to:

- Discuss problem analysis
- Explain software requirement specification
- Describe the characteristics and components of SRS document.
- Explain software requirement specification language
- Discuss the structure of document

Introduction

As a result of the analysis, the Software Requirements Specification is created. Requirements analysis is a software engineering task that connects the dots between system and software engineering. Software engineers can utilize requirements analysis to fine-tune software allocation and construct models of data, function, and behavioral domains that will be used by the software. It gives the designer the knowledge he or she needs to create data, architectural, interface, and component-level designs. Finally, it allows both the developer and the customer to evaluate the quality of the software once it has been produced. Software requirements analysis can be divided into five areas of effort: Problem recognition, Evaluation and synthesis, Modeling, Specification, and Review. Initially, the analyst studies the system specification and the software project plan to understand the system as a whole in relation to the software and to review the scope of the software as per the planning estimates. After this the communication for analysis is established to ensure problem recognition.

The SRS (Software Requirements Specification) is a document that specifies specifications for a specific software product, programme, or collection of programmes that execute a specified function in a specified environment. A Software Requirements Specification (SRS) is a detailed description of a software system to be developed, including both functional and non-functional requirements. The customer could write the SRS. It must cater to the users' requirements and

expectations. SRS could be written by the system's creator. It has a separate goal and emphasises the difference between the developer and the customer.

The analyst begins the requirements gathering and analysis process by obtaining all relevant information from the customer that could be utilised to establish the system's requirements. He then examines the data gathered to gain a clear and full understanding of the product that will be built, in order to eliminate any ambiguities or inconsistencies in the customer's initial impression of the problem. In order to have a good understanding of the problem, the analyst should be able to answer the following basic questions about the project.

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has a thorough understanding of the customer's actual requirements, he moves on to identifying and resolving the numerous requirements issues. Anomalies, inconsistencies, and incompleteness are the most crucial requirements problems for the analyst to uncover and eliminate. When the analyst discovers any contradictions, anomalies, or gaps in the requirements, he remedies them by having additional discussions with the end users and customers.

4.1 Problem Analysis

The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and what the constraints on the solution are. Frequently the client and the users do not understand or know all their needs, because the potential of the new system is often not fully appreciated. The analysts have to ensure that the real needs of the clients and the users are uncovered, even if they don't know them clearly. That is, the analysts are not just collecting and organizing information about the client's organization and its processes, but they also act as consultants who play an active role of helping the clients and users identify their needs.

The basic principle used in analysis is the same as in any complex task: divide and conquer. That is, partition the problem into sub problems and then try to understand each sub problem and its relationship to other sub problems in an effort to understand the total problem. The concepts of state and projection can sometimes also be used effectively in the partitioning process. A state of a system represents some conditions about the system. Frequently, when using state, a system is first viewed as operating in one of the several possible states, and then a detailed analysis is performed for each state.

Problem Analysis consists of the following approaches:

1. ***Informal Approach***: An informal approach is one where no defined methodology is used to analyse. The information about the system is attained by interaction with the questionnaires, client, end users, study of existing documents, brainstorming, etc. The informal approach is used extensively to analysis and can be relatively suitable as conceptual modeling-based approaches repeatedly do not model all facets of the problem and are not always suitable for all the problems. As the Software Requirements Specification (SRS) is to be authorised and the feedback from the validation activity may necessitate for further analysis or requirement. Selecting an informal approach to analysis is not very uncertain—the errors that could be presented are not essentially heading for slip by the prerequisites phase. Therefore such approaches might be the utmost useful approach to analysis in some conditions.
2. ***Shadowing***: Shadowing provides an effective means to discover what is currently being done in a business. Shadowing is a good way to get an idea of what a person does on a day-to-day basis. However, you might not be able to observe all the tasks during shadowing because more than likely the person will not, during that session, perform all the tasks he or she is assigned. For example, accounting people might create reports at the

end of the month, developers might create status reports on a weekly basis, and management might schedule status meetings only biweekly. In addition, you can also learn the purpose of performing a specific task. To gather as much information as possible, you need to encourage the user to explain the reasons for performing a task in as much detail as possible. Shadowing can be both passive and active. When performing passive shadowing, you observe the user and listen to any explanations that the user might provide. When performing active shadowing, you ask questions as the user explains events and activities. You might also be given the opportunity to perform some of the tasks, at the user's discretion.

3. ***Interviews***: In interview is a one-on-one meeting between a member of the project team and a user. The quality of the information a team gathers depends on the skills of both the interviewer and the interviewee. An interviewer can learn a great deal about the difficulties and limitations of the current solution. Interviews provide the opportunity to ask a wide range of questions about topics that you cannot observe by means of shadowing. Shadowing is not the best option for gathering information about tasks such as management-level activities; long-term activities those span weeks, months, or years; or processes that require little or no human intervention. An example of a process that requires no human intervention is the automatic bill paying service provided by financial institutions. For gathering information about such activities and processes, you need to conduct interviews.

4.2 Characteristics of a Good SRS

An SRS should have the following characteristics:

Correct: An SRS is said to be correct if it contains the requirements that the software should meet. The correctness cannot be measured through a tool or a procedure.

Unambiguous: An SRS is unambiguous only if the requirements contained in it have only one interpretation. It must be unambiguous to both who create and who use it. However, the two groups may use different languages for description. But the language should be a natural language that can be easily understood by both the groups.

Complete: An SRS is complete if it includes the following:

All significant requirements related to functionality, performance, design, attributes or external interfaces, Definition of the software responses to all possible input classes in all possible situations, Full references and labels to figures, tables, diagrams and the definition of all terms and units of measure.

Consistent: An SRS is consistent if no subset of a requirement defines it in entirety. The conflicts can be of three types:

- Characteristics of real-world objects.
- Logical or temporal conflicts.
- The same real world object defined in more than one places in different terminologies.

Ranked for Importance and Stability: An SRS is ranked for importance and stability if each requirement contained in it has an identifier indicating, its importance or stability of that requirement,another way to implement the importance of requirements is to classify them as essential conditional and optional.

Verifiable: An SRS is verifiable if every requirement contained in it is verifiable i.e. there exists some defined method to judge if a software meets all the requirements. Non verifiable requirements should be removed or revised.

- Modifiable: An SRS is modifiable if its styles and structures can be retained easily, completely and consistently while changing a requirement.
- Traceable: An SRS is traceable if each of its requirements is clear and can be referenced in future development or enhancement documentation. Two types of traceability are recommended:
    - Backward Traceability: This depends upon each requirement explicitly referencing its source in earlier documents.
    - Forward Traceability: This depends upon each requirement in SRS having a unique name or reference number.

- Backward Traceability: This depends upon each requirement explicitly referencing its source in earlier documents.
- Forward Traceability: This depends upon each requirement in SRS having a unique name or reference number.

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

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

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

Right level of abstraction: If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used hence, the level of abstraction varies according to the purpose of the SRS.

4.3 Components of SRS

Completeness of specifications is difficult to achieve and even more difficult to verify. Having guidelines about what different things an SRS should specify will help in completely specifying the requirements. The basic issues an SRS must address are:

Functionality

Performance

Design constraints imposed on an implementation

External interfaces

Functionality

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

Performance requirements

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

Design constraints

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

It imposes a restriction sometimes on the use of some commands, control access to data; require the use of passwords and cryptography techniques etc

External Interface requirements

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

Notes: If there is a formula for computing the output, it should be specified.

An important part of the specification is the system behavior in abnormal situations, like invalid input (which can occur in many ways) or error during computation. The functional requirement

must clearly state what the system should do if such situations occur. Specifically, it should specify the behavior of the system for invalid inputs and invalid outputs. Furthermore, behavior for situations where the input is valid but the normal operation cannot be performed should also be specified.

Example: An example of this situation is a reservation system, where a reservation cannot be made even for a valid request if there is no availability.

In short, the system behavior for all foreseen inputs and all foreseen system states should be specified.

The performance requirements part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic. Static requirements are those that do not impose constraint on the execution characteristics of the system. These include requirements like the number of terminals to be supported, the number of simultaneous users to be supported, and the number of files that the system has to process and their sizes. These are also called capacity requirements of the system.

Dynamic requirements specify constraints on the execution behavior of the system. These typically include response time and throughput constraints on the system. Response time is the expected time for the completion of an operation under specified circumstances. Throughput is the expected number of operations that can be performed in a unit time.

Example: The SRS may specify the number of transactions that must be processed per unit time, or what the response time for a particular command should be. Acceptable ranges of the different performance parameters should be specified, as well as acceptable performance for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms. Requirements such as "response time should be good" or the system must be able to "process all the transactions quickly" are not desirable because they are imprecise and not verifiable. Instead, statements like "the response time of command x should be less than one second 90% of the times" or "a transaction should be processed in less than one second 98% of the times" should be used to declare performance specifications.

There are a number of factors in the client's environment that may restrict the choices of a designer leading to design constraints. Such factors include standards that must be followed, resource limits, operating environment, reliability and security requirements, and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Example: Some examples of these are:

Standards Compliance: This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit requirements which may require logging of operations.

Hardware Limitations: The software may have to operate on some existing or Notes predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage. Reliability and Fault

Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed, as they make the system more complex and expensive. Recovery requirements are often an integral part here, detailing what the system should do if some failure occurs to ensure certain properties.

Security: Security requirements are becoming increasingly important. These requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system. They may also require proper assessment of security threats, proper programming techniques, and use of tools to detect flaws like buffer overflow.

In the external interface specification part, all the interactions of the software with people, hardware, and other software should be clearly specified. For the user interface, the characteristics of each user interface of the software product should be specified. User interface is becoming increasingly important and must be given proper attention. A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages. Like other specifications, these requirements should be precise and verifiable. So, a statement like "the system should be user friendly" should be avoided and statements like "commands should be no longer than six characters" or "command names should reflect the function they perform" used.

For hardware interface requirements, the SRS should specify the logical characteristics of each interface between the software product and the hardware components. If the software is to execute on existing hardware or on predetermined hardware, all the characteristics of the hardware, including memory restrictions, should be specified. In addition, the current use and load characteristics of the hardware should be given.

The interface requirement should specify the interface with other software the system will use or that will use the system. This includes the interface with the operating system and other applications.

4.4 Structure of Document

The Institute of Electrical and Electronics Engineers (IEEE) has published guidelines and standards to organize an SRS document. There is no single method that is suitable for all projects.

The different ways are proposed to structure the SRS. Although the first two sections are the same, the third section containing the "specific requirements" can differ.

Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms and Abbreviations

1.4 References

1.5 Overview

2. **Overall Description**

2.1 Product Perspective

2.2 Product Functions

2.3 User Characteristics

2.4 Constraints

2.5 Assumptions and Dependencies

3. Specific Requirements

4.5 Properties of a Good SRS Document

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

Concise: The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

Structured: It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

Black-box view: It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

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

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

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

Problems without a SRS document

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

Without developing the SRS document, the system would not be implemented according to customer needs.

Software developers would not know whether what they are developing is what exactly required by the customer.

Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.

It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Problems with an unstructured specification

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

4.6 Functional and Non-Functional Requirement

Functional requirement

A functional requirement defines a system or its component. It describes the functions software must perform.

A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

A functional requirement will describe a particular behaviour of function of the system when certain conditions are met, for example: Send email when input is not given on time, any Requirement Which Specifies "What the System Should Do".

Depend on the type of software, expected users and the type of system where the software is used. Functional system requirements should describe the system services in detail.

Functional requirements include:

- Business Rules
- Transaction corrections, adjustments and cancellations
- Administrative functions
- Authentication
- Authorization levels
- Audit Tracking

- External Interfaces
- Certification Requirements
- Reporting Requirements
- Historical Data
- Legal or Regulatory Requirements

Business requirements: They contain the ultimate goal, such as an order system, an online catalogue, or a physical product. It can also include things like approval workflows and authorization levels.

Administrative functions. They are the routine things the system will do, such as reporting.

User requirements: They are what the user of the system can do, such as place an order or browse the online catalogue.

System requirements: These are things like software and hardware specifications, system responses, or system actions.

Functional Requirements Examples:

Only Managerial level employees have the right to view revenue data.

The software system should be integrated with banking API

The Sales system should allow users to record customer's sales

Authentication of a user when he/she tries to log into the system

System shutdown in the case of a cyber-attack

Verification email is sent to user whenever he/she registers for the first time on some software system.

Advantages of Functional Requirement

- A functional requirement document helps you to define the functionality of a system or one of its subsystems.
- Functional requirements along with requirement analysis help identify missing requirements. They help clearly define the expected system service and behavior.
- Errors caught in the Functional requirement gathering stage are the cheapest to fix.
- Support user goals, tasks, or activities for easy project management.
- Functional requirement can be expressed in Use Case form or user story as they exhibit externally visible functional behavior.
- Helps you to check whether the application is providing all the functionalities that were mentioned in the functional requirement of that application.

Non Functional Requirement

A non-functional requirement defines the quality attribute of a software system. They represent a set of standards used to judge the specific operation of a system.

Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards.A non-functional requirement is essential to ensure the usability and effectiveness of the entire software system.A non-functional requirement often applies to the system as a whole rather than individual features or services.

Non-functional Requirements allows you to impose constraints or restrictions on the design of the system across the various agile backlogs. Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Non Functional Requirement includes:

Portability

Security

Maintainability

Reliability

Scalability

Performance

Reusability

Flexibility

Reliability/Availability: What are the uptime requirements? Does it need to function 24/7/365?

Performance: How fast does it need to operate?

Usability: This focuses on the appearance of the user interface and how people interact with it. What colour are the screens? How big are the buttons?

Scalability: As needs grow, can the system handle it? For physical installations, this includes spare hardware or space to install it in the future.

Supportability: Is support provided in-house or is remote accessibility for external resources required?

Security: What are the security requirements, both for the physical installation and from a cyber-perspective?

Example:Non-functional requirements

- A website should be capable enough to handle 20 million users with affecting its performance.
- The software should be portable. So moving from one OS to other OS does not create any problem.
- Privacy of information, the export of restricted technologies, intellectual property rights, etc. should be audited.
- Emails should be sent with a latency of no greater than 10 hours.
- Each request should be processed within predefined time
- The site should load in 5 seconds when the number of simultaneous users are > 15000

Advantages of Non-functional requirements

They ensure the reliability, availability, and performance of the software system

They ensure good user experience and ease of operating the software.

The nonfunctional requirements ensure the software system follow legal and compliance rules.

They help in formulating security policy of the software system.

Identifying functional requirements from a problem description

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

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

F1: Search Book function

Input: an author's name

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

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

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

Documenting functional requirements

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

Example: Withdraw Cash from ATM

R1: withdraw cash

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

R1.1 select withdraw amount option

Input: "withdraw amount" option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: get required amount

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

Output: The requested cash and printed transaction statement.

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

4.7 Modeling Techniques

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

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

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

Entity Life Histories (ELHs): Entity Life Histories (ELHs) describe the changes which happen to 'things' (entities) within the system

These three techniques are common to many methodologies and are widely used in system analysis. Notation and graphics style may vary across methodologies, but the underlying principles are generally the same. In SSADM (Structured Systems Analysis and Design Methodology - which has for a number of years been widely used in the UK) systems analysts and modelers use the above techniques to build up three, inter-related, views of the target system, which are cross-checked for consistency.

Summary

- The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and what the constraints on the solution are.
- Data flow diagrams (also called data flow graphs) are commonly used during problem analysis.
- The SRS (Software Requirements Specification) contains specifications for a particular software product, program or a set of programs that perform a particular function in a specific environment.
- The performance requirements part of an SRS specifies the performance constraints on the software system.
- Unlike formal language that allows developers and designers some latitude, the natural language of software requirements specifications must be exact, without ambiguity, and precise.

Keywords

DFD: A DFD shows the flow of data through a system.

Evolutionary Prototyping: It uses the prototype as the first part of an analysis activity that will be continued into design and construction.

Object-Oriented Software Engineering (OOSE): It is a software design technique that is used in software design in object-oriented programming.

Problem Analysis: The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and what the constraints on the solution are.

Requirements Analysis: It is a software engineering task that bridges the gap between system level engineering and software design.

SRS: Software Requirements Specification contains specifications for a particular software product, program or a set of programs that perform a particular function in a specific environment.

Self Assessment

1. Functional and non-functional requirement are the part of ____

A. Design phase
B. SRS
C. Testing
D. Maintenance

2. Functional requirements are_____

A. Defines a system or its component
B. Describe a particular behavior of function of the system when certain conditions are met
C. "What the System Should Do"
D. All of above

3. Functional requirements not include _____

A. Business Rules
B. Administrative functions
C. Reusability
D. Authentication

4. Audit Tracking, External Interfaces and historical data is part of ____

A. Nonfunctional requirement
B. Functional requirement
C. All of the above

5. Nonfunctional requirements are _____

A. Defines the quality attribute of a software system
B. Represent a set of standards used to judge the specific operation of a system
C. Essential to ensure the usability and effectiveness of the entire software system
D. All of above

6. Non Functional Requirement include

A. Scalability
B. Performance
C. Reusability
D. All of above

7. Each request should be processed within predefined time is _____

A. Nonfunctional requirement
B. Functional requirement
C. All of the above

8. SRS stands for ____

A. System Requirements Specification
B. Software Requirements Specification
C. Software Requirements support
D. None of above

9. Software Requirements Specification is _____

A. Developed with its functional and non-functional requirements
B. Must address the needs and expectations of the users
C. It serves a different purpose and mentions the contrast between the customer and the developer.
D. All of above

10. Characteristics of a SRS_____

A. Correct
B. Unambiguous
C. Complete
D. All of above

11. As per IEEE which is not a part of Structure of Document

A. Purpose
B. Scope
C. Testing
D. Product Functions

12. The SRS document is also known as _____ specification

A. Black-box
B. White-box
C. Grey-box
D. None of above

13. Which of the following is not included in SRS?

A. Functionality
B. Performance
C. Design solutions
D. External Interfaces

14. What is the first step of requirement elicitation?

A. Identifying Stakeholder
B. Listing out Requirements
C. Requirements Gathering
D. All of above

Answer for Self Assessment

1.	B	2.	D	3.	C	4.	B	5.	D
6.	D	7.	A	8.	B	9.	D	10.	D
11.	C	12.	A	13.	C	14.	D		

Review Questions

1. What is the aim of problem analysis? Discuss.
2. Describe the different approaches used in problem analysis.
3. Explain the concept of data flow in software requirements.
4. Discuss the concept of object orientedmodeling.
5. Describe the techniques used for conducting rapid prototyping.
6. Explain Software Requirements Specification (SRS). Also discuss the characteristics of SRS.
7. Discuss the different components of SRS.
8. Discuss the language quality characteristics of an SRS.
9. Discuss how to represent the Structure of SRS document?

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Links

- http://softwareengineeringhub.blogspot.in/2010/03/system-requirementsspecification.html
- http://computersciencesource.wordpress.com/2010/03/14/softwareengineering-object-oriented-modelling/
- http://www.software-requirements.tuffley.com/sample.pdf
- https://www.geeksforgeeks.org/
- https://educatech.in/

Unit 05: Design

CONTENTS

Objectives

- after studying this unit, you will be able to:
- design process and design concepts
- data flow diagram (DFD) and flow chart, architectural design
- use case diagram, class diagram, activity diagram

Introduction

Software design is a problem-solving and planning method for creating a software solution. After determining the objective and parameters of software, software developers will design or hire designers to create a solution plan. It covers both the architectural view and low-level component and algorithm implementation difficulties.

Design is defined as "the process of using diverse approaches and principles to define a device, a process, or a system in sufficient detail to allow physical realization."

The design process converts the ''what'' of the requirements to the ''how'' of the design. Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain.

5.1 What is a Software Design?

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

Software design can be defined as the process of employing available capabilities to create a solution to the problem(s) at hand. As a result, the fundamental distinction between software analysis and design is that the outcome of a software problem analysis will be fewer problems to solve, and it should not change too much even if it is carried out by various team members or even whole different groups. However, because design is based on capabilities, several designs for the same problem can exist depending on the capabilities of the environment that will host the solution (whether it is some OS, web , mobile or even the new cloud computing paradigm).The solution will depend also on the used development environment (Whether you build a solution from scratch or using reliable frameworks or at least implement some suitable design patterns)

Software design is the process of transforming user requirements into a format that the programmer can use to code and implement software. An SRS (Software Requirement Specification) document is prepared for assessing user requirements; however more explicit and precise software requirements are required for coding and implementation. The output of this procedure can be directly implemented in computer languages.

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

5.2 Design Process

The phrase software design is often used to characterize the discipline that is also called software engineering—the discipline concerned with the construction of software that is efficient, reliable, robust, and easy to maintain.

Interface Design

Bringing Design to Software suggests that the design object is software, ignoring the interface devices that are the user's inextricable embodiment of software. Design cannot be easily split into software and device compartments: Physical interfaces both create and constrain software options. The great majority of computer applications currently offer themselves to users in a typical manner—a visual display with a keyboard and mouse. However, computers in the future will provide more resources for physical human–computer interactions. On a small scale, certain innovative devices are already in use.

Example: Pen-based personal digital assistants (PDAs), virtual-reality goggles and gloves, and computers embedded in electromechanical devices of all kinds. Researchers are exploring further possibilities, including tactile input and output devices, immersive environments, audio spaces, wearable computers, and a host of gadgets that bear little resemblance to today's personal computer or workstation. As experience with a wider variety of devices accumulates, the design of interaction based on new combinations of devices and software will be an important emerging topic in what we have—for the moment—called software design.

Human-computer Interaction

Whenever someone designs software that interacts with people, the effects of the design extend beyond the software itself to include the experiences that people will have in encountering and using that software. A person encountering any artifact applies knowledge and understanding, based on wide variety of cognitive mechanisms grounded in human capacities for perception, memory, and action. Researchers in human-computer interaction have studied the mental worlds of computer users, developing approaches and methods for predicting properties of the interactions and for supporting the design of interfaces. Although it would be overstating the case to say that the cognitive analysis of human-computer interaction has led to commonly accepted and widely applied methods, there is a substantial literature that can be of value to anyone designing interactive software.

Art

The experience of a person who is interacting with a computer system is not limited to the cognitive aspects that have been explored in the mainstream literature on human-computer interaction. As

humans, we experience the world in aesthetic, affective, and emotional terms as well. Because computing evolved initially for use in the laboratory and the office, non-cognitive aspects have been largely ignored, except by creators of computer games. Yet, whenever people experience a piece of software—whether it be a spreadsheet or a physics simulation—they have natural human responses. They experience beauty, satisfaction, and fun, or the corresponding opposites.

As computing becomes integrated into technologies for entertainment, and as the typical user moves from the well-regimented office to the home recreation room, software designers will need to focus more on the affective dimensions of human response. We can learn from the history of other human communication media, and can adapt the principles and techniques of novelists, film makers, composers, visual artists, and many other designers in what are loosely called the arts. Designing for the full range of human experience may well be the theme for the next generation of discourse about software design.

Perhaps even more difficult than the task of defining software is the task of defining design. A dictionary provides several loosely overlapping meanings, and a glance at the design section in a library or bookstore confuses the issue even more. Although we label it with a noun, design is not a thing. The questions that we can ask fruitfully are about the activity of designing.

Design is Conscious

People may refer to an object as being well designed whenever it is well suited to its environment, even if this suitability resulted from a process of unintentional evolution. In this book, we concentrate on what happens when a designer reflects on and brings focus to the design activity.

Consciousness about designing does not imply the application of a formal, consistent, or comprehensive theory of design or of a universal methodology. Systematic principles andmethods at times may be applicable to the process of design, but there is no effective equivalentto the rationalized generative theories applied in mathematics and traditional engineering.Design consciousness is still pervaded by intuition, tacit knowledge, and gut reaction.

Design keeps Human Concerns in the Center

All engineering and design activities call for the management of tradeoffs. Real-world problemsrarely have a correct solution of the kind that would be suitable for a mathematics problem orfor a textbook exercise. The designer looks for creative solutions in a space of alternatives thatis shaped by competing values and resource needs. In classical engineering disciplines, thetradeoffs can often be quantified: material strength, construction costs, rate of wear, and the like.In design disciplines, the tradeoffs are more difficult to identify and to measure. The designer stands with one foot in the technology and one foot in the domain of human concerns, and these two worlds are not easily commensurable.

As an example, it is easy for software designers to fall into a single-minded quest, in which easeof use (especially for beginning users) becomes a holy grail. But what is ease of use? How muchdoes it matter to whom? A violin is extremely difficult for novices to play, but it would be foolhardy to argue that it should therefore be replaced by the autoharp. The value of an artifact may lie in high-performance use by virtuosos, or in ease of use for some special class of users,such as children or people with disabilities. There is room for the software equivalents of high-strung racing cars alongside automatic-transmission minivans.

Design is a Dialog with Materials

The ongoing process of designing is iterative at two levels: iteration by the designer as a pieceof current work develops and iteration by the community as successive generations reveal new possibilities for the medium. Designing is not a process of careful planning and execution, but rather it is like a dialog, in which the dialog partner—the designed object itself—can generate unexpected interruptions and contributions. The designer listens to the emerging design, as well as shapes it.

Design always implies a medium of construction, and new technologies bring with them new domains for design. Architecture as we know it appeared with the technology for building with stone. Graphic design emerged with the technologies of printing, and modern product design flourished with the development of plastics that expanded the possible variety of forms for everyday products.

Design is Creative

It is one thing to lay out a list of criteria for good design. It is quite another thing to do design well. Designer lacks the comforting restraints of a well-organized engineering discipline, because designing is inherently messy; it includes, but goes beyond, the ability to be creative in solving problems. It begins with creativity in finding the problems—envisioning the needs that people have but do not recognize.

Design is more an art than a science—it is spontaneous, unpredictable, and hard to define. The skill of the artist–designer is not reducible to a set of methods, and is not learned through the kind of structured curriculum that serves in science and engineering. On the other hand, it is not a mysterious gift. There is a long and rich tradition of education in the design fields; it draws on the interaction between learner and teacher, designer and critic.

Design is Communication

Previous sections have described the interaction between the user and his world, and the interaction between the designer and her materials. What matters most is the interaction between these two interactions. Virtuality is neither just what the designer thinks it is, nor what any particular user thinks it is. It exists in the ongoing dialog between designers and users. To succeed in designing, we need to understand how designers convey meaning.

At a surface level, designed artifacts communicate content. Skilled designers in every discipline know how to manage these layers of meaning. In addition, an active artifact—whether it be a computer program or a coffee maker—communicates to users about its use. A device as apparently simple as a door communicates to its users through convention. A door with a flat plate near shoulder level says "Push me!" One with a round knob says "Twist here." Although these messages are constrained by the physical mechanisms, they are also a matter of convention and learning, as every tourist finds out in trying to deal with everyday objects in an unfamiliar culture.

As is true of any human language, what is visible in a statement is only a small part of the full picture. A hearer of the sentence "I'm going to kill you!" cannot interpret the intended meaning without knowing the situational context—was it said to a friend walking onto the tennis court, in an anonymous note mailed to the President, or by a parent to a child who has just left dirty clothing on the bathroom floor again? The literal meaning is but the shadow of the meaning in context.

The same is true for the artifacts that we build, including software: People do not approach them in isolation. Every object appears in a context of expectations that is generated by the history of previous objects and experiences, and by the surroundings in the periphery—the physical, social, and historical context in which the object is encountered.

Design has Social Consequences

Much of the discussion on software design uses examples from generic mass-distribution software, such as word processors, operating systems, spreadsheets, graphics programs, and games. Although many key concerns of software design are addressed in these applications, others are not. These highly visible applications are part of a larger picture that includes a vast number of vertical applications (for example, a medical-office billing application) and systems designed for a specific workplace setting. In these more targeted applications, the designer can see and take into account the specific effects of the design on the people who will inhabit it. Organizational aspects of software design in which we build integrated computer systems for specific organizations and workplaces. The needs and concerns of the people in those workplaces

can lead to complex—and, at times, controversial—design considerations, which we can address only by making the social and political dimensions an explicit part of the analysis and of the design dialog. The designer takes on a role that includes organizational as well as technical design, and can enlist workers directly in this process through techniques such as participatory design.

Design is a Social Activity

In concentrating on the activity of an individual designer, we can fall into the error of assuming that the overall quality of a design is primarily a result of the qualities and activities of the creative individual. As Kelley points out, the designer operates in a larger setting, which is both facilitated and constrained by interactions with other people.

5.3 Objectives of Software Design

Correctness: A good design should be correct i.e. it should correctly implement all the functionalities of the system.

Efficiency: A good software design should address the resources, time and cost optimization issues.

Understandability: A good design should be easily understandable, for which it should be modular and all the modules are arranged in layers.

Completeness: The design should have all the components like data structures, modules, and external interfaces, etc.

Maintainability: A good software design should be easily amenable to change whenever a change request is made from the customer side.

Flexibility: Able to modify on changing needs.

5.4 Software Design Levels

Interface Design: Interface design is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system

Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts.

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.Specification on the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

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

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between components.

High-level Design- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other.

High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

Detailed Design- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs.

It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units

- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

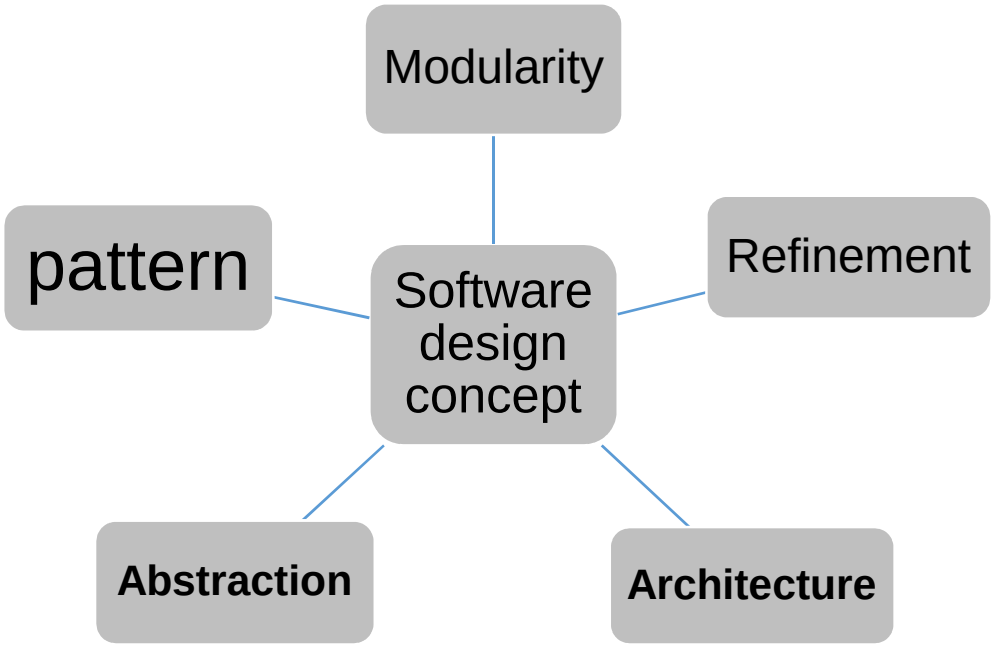
5.5 Software Design Concepts

In order to deal with the complexity, software is designed in layers. When a programmer is worrying about the detailed design of one module, there are probably hundreds of other modules and thousands of other details that he cannot possibly worry about at the same time. For example, there are important aspects of software design that do not fall cleanly into the categories of data structures and algorithms. Ideally, programmers should not have to worry about these other aspects of a design when designing code.

The software design concept is the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software

The software design concept provides a supporting and essential structure or model for developing the right software.

Software Design Concepts



Modularization

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

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

Advantage of Modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction ca n be brought in the program

- Components with high cohesion can be re-used again.
- Concurrent execution can be made possible
- Desired from security aspect

Refinement

The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software.

Concurrency

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

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.

It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

On any software project of typical size, problems like these are guaranteed to come up. Despite all attempts to prevent it, important details will be overlooked. This is the difference between craft and engineering. Experience can lead us in the right direction. This is craft. Experience will only take us so far into uncharted territory. Then we must take what we started with and make it better through a controlled process of refinement. This is engineering.

As just a small point, all programmers know that writing the software design documents after the code instead of before, produces much more accurate documents. The reason is now obvious. Only the final design, as reflected in code, is the only one refined during the build/test cycle. The probability of the initial design being unchanged during this cycle is inversely related to the number of modules and number of programmers on a project.

In software engineering, we desperately need good design at all levels. In particular, we need good top level design. The better the early design, the easier detailed design will be. Designers should use anything that helps. Structure charts, Brooch diagrams, state tables, PDL, etc. — if it helps, then use it. We must keep in mind, however, that these tools and notations are not a software design. Eventually, we have to create the real software design, and it will be in some programming language. Therefore, we should not be afraid to code our designs as we derive them. We simply must be willing to refine them as necessary.

There is as yet no design notation equally suited for use in both top level design and detailed design. Ultimately, the design will end up coded in some programming language. This means that top level design notations have to be translated into the target programming language before detailed design can begin. This translation step takes time and introduces errors. Rather than translate from a notation that may not map cleanly into the programming language of choice, programmers often go back to the requirements and redo the top level design, coding it as they go. This, too, is part of the reality of software development.

5.6 Coupling and Cohesion

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

In modular approach tasks are divided into several modules based on some characteristics. Modules are set of instructions put together in order to achieve some tasks. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve customer's problem.

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

A module having low coupling and high cohesion is said to be functionally independent of other modules

Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability.

Low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand.

Types of Cohesion

Coincidental Cohesion (Worst)

Logical Cohesion

Temporal Cohesion

Procedural Cohesion

Communicational Cohesion

Sequential Cohesion

Functional Cohesion (Best)

- Co-incidental cohesion - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- Logical cohesion - When logically categorized elements are put together into a module, it is called logical cohesion.
- Temporal Cohesion - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- Procedural cohesion - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- Communicational cohesion - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- Sequential cohesion - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- Functional cohesion - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program the use of global variables can result in an enormous number of connections between the modules of a program

The degree of coupling between two modules is a function of several factors

How complicated the connection is

Whether the connection refers to the module itself or something inside it

What is being sent or received

Coupling can be "low" ("loose" and "weak") or "high" ("tight" and "strong").

Low coupling means that one module does not have to be concerned with the internal implementation of another module, and interacts with another module with a stable interface

Low coupling is a sign of a well-structured computer system.

Types of coupling

Data coupling

Content coupling

Common coupling

Control coupling

Stamp coupling

External Coupling

- ***Content coupling*** - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- ***Common coupling***- When multiple modules have read and write access to some global data, it is called common or global coupling.
- ***Control coupling***- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- ***Stamp coupling***- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.Stamp coupling is when modules share a composite data structure, each module not knowing which part of the data structure will be used by the other

  Example: passing a student record to a function which calculates the student's GPA
- ***Data coupling***- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.
- ***External coupling***- External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Ideally, no coupling is considered to be the best.

5.7 Data Flow Diagram and Flow Chart

"A picture is worth a thousand words"

Data flow diagram is graphical representation of flow of data in an information system.

It includes data inputs and outputs, data stores, and the various sub processes the data moves through.

Diagrams are used to map out an existing system and make it better or to plan out a new system for implementation. Visualizing each element makes it easy to identify inefficiencies and produce the best possible system.

DFDs are built using standardized symbols and notation to describe various entities and their relationships.

Types of DFD

Logical

Physical

Logical

Logical data flow diagrams focus on what happens in a particular information flow.What information is being transmitted, what entities are receiving that info, what general processes occur, etc.A logical DFD doesn't deal with the technical aspects of a process or system.

Physical

Physical data flow diagrams focus on how things happen in an information flow. Physical DFD specify the software, hardware, files, and people involved in an information flow. A detailed physical data flow diagram can facilitate the development of the code needed to implement a data system.

Levels of Data Flow Diagram

Level 0

Level 1

Level 2

Level 0 DFD

It is also known as a context diagram. It's designed to be an abstraction view, showing the system as a single process with its relationship to external entities. It represents the entire system as a single bubble with input and output data indicated by incoming/outgoing arrows.

Level 1 DFD

In level 1 DFD, the context diagram is decomposed into multiple modules/processes. In this level, we highlight the main functions of the system and breakdown the high-level process of 0-level DFD into sub processes.

Level 2 DFD

At this level, DFD shows how data flows inside the modules mentioned in Level 1.It can be used to plan or record the specific necessary detail about the system functioning.

DFD symbols

There are four basic elements of a data flow diagram:

Processes

Data stores

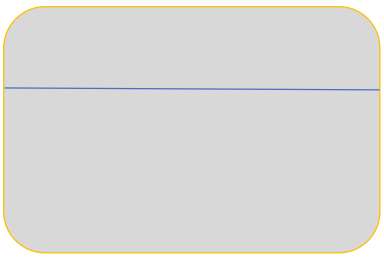
External entities

Data flows

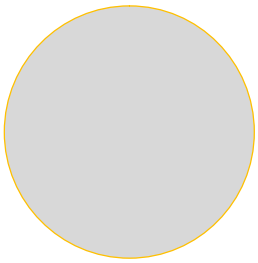
Depending on the methodology, DFD symbols vary slightly.

Process

A process receives input data and produces output with a different content or form. Processes can be as simple as collecting input data and saving in the database



Gane and Sarson



Yourdon and Coad

Data store

Data stores are repositories of data in the system. They are sometimes also referred to as files.

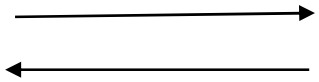
Gane and Sarson Yourdon and Coad

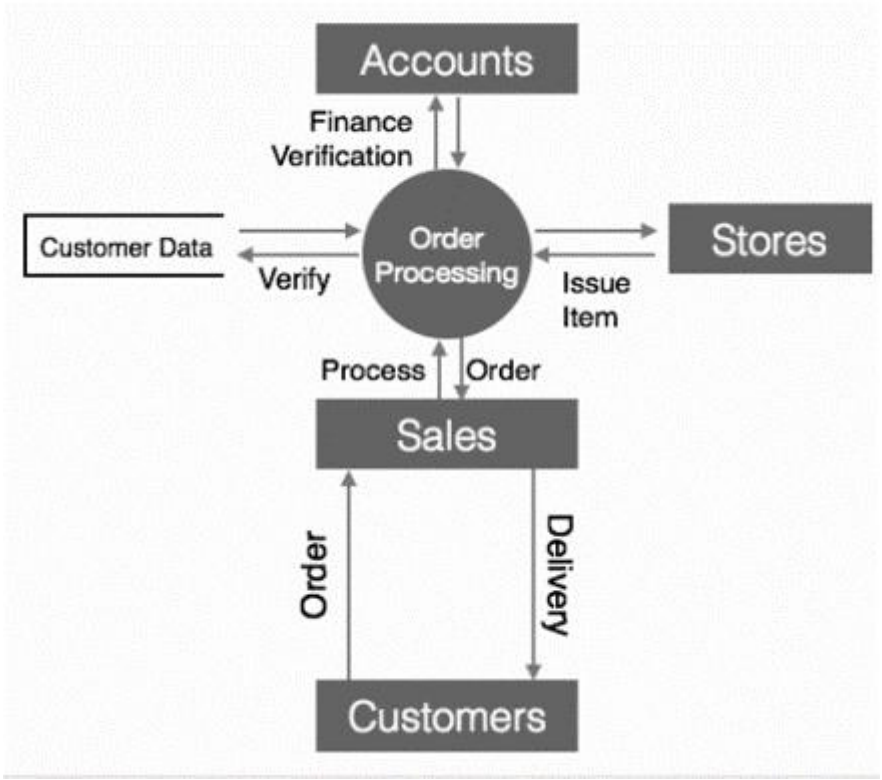
External entities

External entities are objects outside the system, with which the system communicates. External entities are sources and destinations of the system's inputs and outputs

Data flow

A data-flow is a path for data to move from one part of the information system to another





Guideline to create a DFD

- Identify major inputs and outputs in your system
- Build a context diagram
- Expand the context diagram into a level 1 DFD
- Expand to a level 2+ DFD
- Confirm the accuracy of your final diagram

- The name of the entity should be easy and understandable without any extra assistance
- The processes should be numbered or put in ordered list to be referred easily.
- The DFD should maintain consistency across all the DFD levels.

Advantages of DFD

- It helps us to understand the functioning and the limits of a system.
- It is a graphical representation which is very easy to understand as it helps visualize contents.
- Data Flow Diagram represent detailed and well explained diagram of system components.
- It is used as the part of system documentation file.

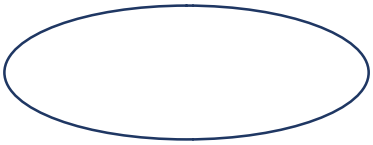
5.8 Flow Chart

A flow chart is a graphical or symbolic representation of a process. Each step in the process is represented by a different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction.

Flow chart symbols

Terminator

The terminator symbol represents the starting or ending point of the system.

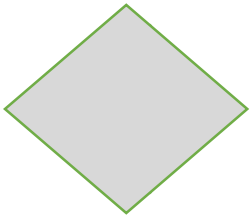


Process

A rectangular flow chart shape indicating a normal process flow step

Decision

A diamond flow chart shape indication a branch in the process flow. Lines' coming out from the diamond indicates different possible situations, leading to different sub-processes.

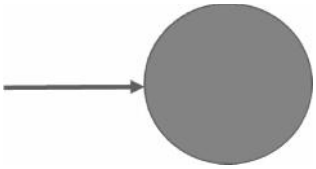


Flow lines

Flow lines are used to show flow of data

Connector

A small, labeled, circular flow chart shape used to indicate a jump in the process flow.



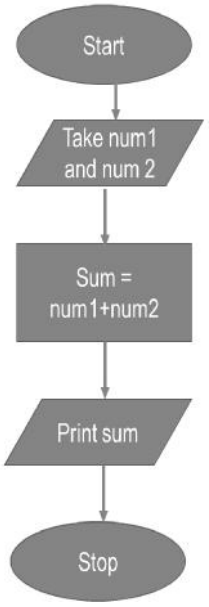
Data

It represents information entering or leaving the system. An input might be an order from a customer. Output can be a product to be delivered.

Document

Used to indicate a document or report

Addition of two numbers.



5.9 Architectural Design

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

The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called architectural design

In architectural design process the activities carried out are system structuring (system is decomposed into sub-systems and communications between them are identified), control modeling, modular decomposition

IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system."

Architectural Design Representation

Structural model: Illustrates architecture as an ordered collection of program components

Dynamic model: Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment

Process model: Focuses on the design of the business or technical process, which must be implemented in the system

Functional model: Represents the functional hierarchy of a system

Framework model: Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction

Architectural styles

An architecture style involves its components, connectors, and constraints on combining components.

Pipes and Filters

In Pipes and Filters, each component (filter) reads streams of data on its inputs and produces streams of data on its output. Pipes are the connectors that transmit output from one filter to another.Functional transformations process their inputs to produce outputs

Call-and-return systems

The program structure decomposes function into a control hierarchy where a "main" program invokes (via procedure calls) a number of program components, which in turn may invoke still other components.

Example: Structure Chart is a hierarchical representation of main program and subprograms

Object-oriented systems

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

Layered systems

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

Data - centered systems

A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.

Data centered systems use repositories. Repository includes a central data structure representing current state, and a collection of independent components that operate on the central data store

Distributed systems

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

Set of stand-alone servers which provide specific services such as printing, data management, etc.

5.10 Component Based Design

In component-based software engineering, a software system is considered as a set of software components assembled together instead of as a set of functions from the traditional perspective. This architecture focuses on the decomposition of the design into logical components which contain events, methods and properties.

Component-Based architecture divides the problem into sub-problems and each problem associated with component partitions.It provides a higher level of abstraction than object-oriented design principles.It does not focus on issues such as communication protocols and shared state.

The objective of component-based architecture is to ensure component reusability. A component encapsulates functionality and behavior of a software element into a reusable and self-deployable binary unit.

Key Issues of Component based design

Component planning

Component building

Component assembling

Component representation

Component retrieval

A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.A software component can be deployed independently and is subject to third-party composition.

Characteristics of Components

Reusability − Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.

Replaceable − Components may be freely substituted with other similar components.

Not context specific − Components are designed to operate in different environments and contexts.

Extensible − A component can be extended from existing components to provide new behavior.

Encapsulated − A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.

Independent − Components are designed to have minimal dependencies on other components.

Component Based design activities

Component Qualification: This activity ensures that the system architecture defines the requirements of the components for becoming a reusable component.Reusable components are generally identified through the traits in their interfaces.

Component Adaptation: This activity ensures that the architecture defines the design conditions for all component and identifying their modes of connection.

Component Composition: This activity ensures that the Architectural styles of the system integrates the software components and form a working system. By identifying connection and coordination mechanisms of the system, the architecture describes the composition of the end product.

Advantages of Component Based design

Reusable − The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.

Reduced cost − the use of third-party components allows you to spread the cost of development and maintenance

Ease of development − Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.

Ease of deployment − as new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.

Modification of technical complexity − A component modifies the complexity through the use of a component container and its services.

Reliability − The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse.

Independent − Independency and flexible connectivity of components Independent development of components by different group in parallel Productivity for the software development and future software development

5.11 Object-Oriented Design

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

Object oriented design represent the problem domain, makes it easy to understand and produce design. In the object-oriented design, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data.

It is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during the Object Oriented Analysis (OOA). An analysis model created using object oriented analysis is transformed by object oriented design into a design model that works as a plan for software development.

Object Oriented Design concepts

Objects - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

Classes - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

Encapsulation - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

Inheritance - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

Messages: Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

Abstraction In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

Polymorphism - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Characteristics of Object Oriented Design

- Objects are abstractions of the real-world or system entities and manage themselves.
- The objects are independent and in an encapsulated state and representation information.

- Shared data areas are eliminated.
- System functionality is expressed in terms of object services.
- Communication between objects is through message passing.
- The objects may be distributed and may execute sequentially or in parallel.

Design Process

A solution design is created from requirement or previous used system and/or system sequence diagram.Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.Class hierarchy and relation among them are defined.Application framework is defined.

Advantages of Object Oriented Design

- Easier to maintain objects.
- Objects may be understood as stand-alone entities.
- Objects are appropriate reusable components.
- For some systems, there may be an obvious mapping from real entities to system objects.

Top down Design approach

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics.

Each subsystem or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Bottom-up Design

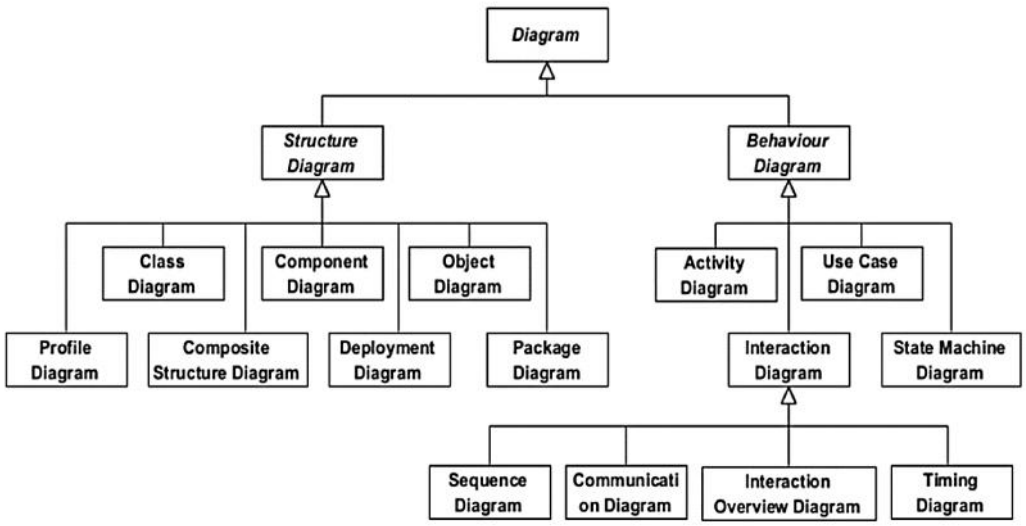
The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components.

It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

5.12 Use Case Diagram, Class Diagram and Activity Diagram

Unified Modeling Language is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

UML diagram



A structure diagram represents the static aspects of the system. It emphasize the things that must be present in the system being modeled

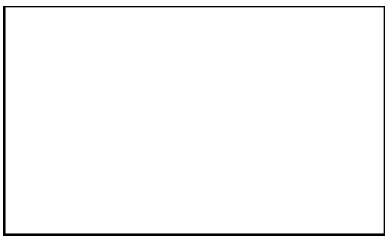
Behaviour diagrams represent the dynamic aspect of the system. It emphasizes what must happen in the system being modeled.

A Unified Modeling Language (UML) use case diagram is the primary form of system/software requirements for a new software program underdeveloped. A use case diagram is a dynamic or behaviour diagram in UML. Use case diagrams model the functionality of a system using actors and use cases. Use cases are a set of actions, services, and functions that the system needs to performUse case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements.

Symbols and Notations

System

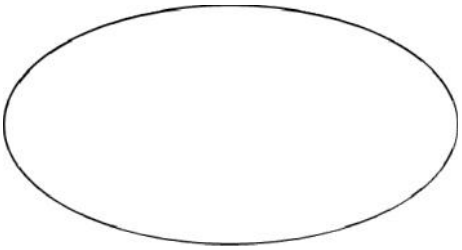
Draw your system's boundaries using a rectangle that contains use cases. Place actors outside the system's boundaries.



System

Use case

A description of a set of sequences of actions, including variants, that system performs that yields an observable value to an actor.

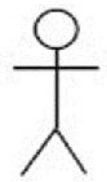


Use case

Actor

The people or systems that provide or receive information from the system; they are amongthe stakeholders of a system.

It could be human beings, other systems, timers and clocks or hardware devices. Actors that stimulate the system and are the initiators of events are called primary actors (active) Actors that only receive stimuli from the system are called secondary actors (passive).

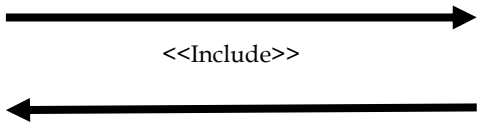


Actor

Relationship

Include: Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source.

Extend: Specifies that the target use case extends the behavior of the source.



Guidelines for use case diagram

- Use cases should ideally begin with a verb – i.egenerates report. Use cases should NOT be open ended – i.e Register (instead should be named as Register New User)
- Avoid showing communication between actors.
- Actors should be named as singular. i.e student and NOT students. NO names should be used – i.e John, Sam, etc.
- Do NOT show behaviour in a use case diagram; instead only depict only system functionality.
- Use case diagram does not show sequence – unlike DFDs.

Class Diagram

Class diagram is a static diagram. It represents the static view of an application

It describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

The class diagrams are widely used in the modeling of object oriented systems

Class

A class represents a concept which encapsulates state (attributes) and behaviour (operations).

A Class is a blueprint for an object. Objects and classes go hand in hand

Class describes what an object will be, type of objects, while objects are usable instances of classes

Class Notation

Class Name: The name of the class appears in the first partition.

Class Attributes

Class Operations (Methods)

Example student (class)

Attributes (name, roll no, height)

Methods (playing, reading)

Class Attributes:

Attributes are shown in the second partition.

The attribute type is shown after the colon.

Attributes map onto member variables (data members) in code.

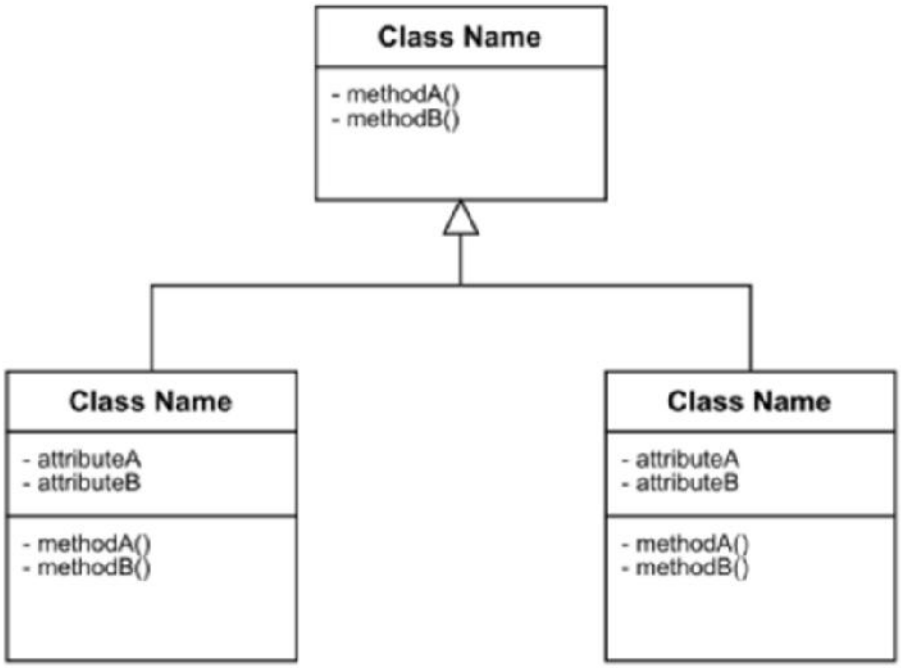
Class Operations (Methods):

Operations are shown in the third partition. They are services the class provides.

The return type of a method is shown after the colon at the end of the method signature.

The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code.

Class diagram



Advantages of class diagram

- It can represent the object model for complex systems.
- It provides a general schematic of an application for better understanding.
- It represents a detailed chart by highlighting the desired code, which is to be programmed.
- It reduces the maintenance time by providing an overview of how an application is structured before coding.
- It is helpful for the stakeholders and the developers.

Activity diagram

In UML activity diagram to describe dynamic aspects of the system

Activity diagram represent the flow from one activity to another activity. The activity can be described as an operation of the system.

An activity diagram is a behavioral diagram i.e. it depicts the behavior of a system.

Objective of Activity Diagram

- Modeling work flow by using activities.
- Modeling business requirements.
- Model workflows between/within use cases
- High level understanding of the system's functionalities.
- Investigating business requirements at a later stage

Activity Diagram Notations

Initial State or Start Point: A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram.

Start/ initial point

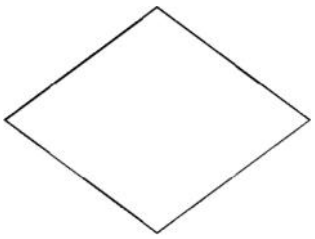
Activity or Action State: An action state represents the non-interruptible action of objects.

Action/ activity

Action Flow: Action flows, also called edges and paths, illustrate the transitions from one action state to another.

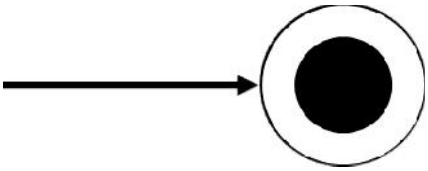
Action flow

Decisions and Branching: A diamond represents a decision with alternate paths. When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.



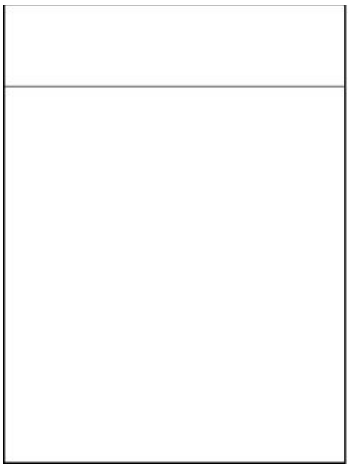
Decision box

Final State or End Point: An arrow pointing to a filled circle nested inside another circle represents the final action state



End point

Swim lanes – We use Swim lanes for grouping related activities in one column. Swim lanes group related activities into one column or one row.



Swim lanes

Guidelines for activity diagram

- All activities in the system should be named.
- Activity names should be meaningful.
- Constraints must be identified.
- Activity associations must be known.

Use case diagram Vs. Activity diagram

- Activity diagram is used to model the workflow depicting conditions, constraints, sequential and concurrent activities
- Use Case is to just depict the functionality i.e. what the system does and not how it is done
- Activity diagram shows 'How' while a Use case shows 'What' for a particular system

Summary

- Whenever objects are created for people to use, design is pervasive. Design may be done methodically or offhandedly, consciously or accidentally.
- The phrase software design is often used to characterize the discipline that is also called software engineering—the discipline concerned with the construction of software that is efficient, reliable, robust, and easy to maintain.
- Software design principles represent a set of guidelines that helps us to avoid having a bad design.
- The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices".
- In order to deal with the complexity, software is designed in layers. When a programmer is worrying about the detailed design of one module, there are probably hundreds of other modules and thousands of other details that he cannot possibly worry about at the same time.
- A modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of different parts of a system.

Keywords

Coupling: Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system.

Dependency Inversion Principle: Dependency Inversion Principle states that we should decouple high level modules from low level modules.

Design Patterns: A software design pattern describes a family of solutions to a software design problem.

Modular Design: A modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of different parts of a system.

Self Assessment

1. Software design process is ____

A. Converts the ''what'' of the requirements to the ''how'' of the design

B. The first step in SDLC (Software Design Life Cycle)

C. Process of problem solving and planning for a software solution

D. All of above

2. Which is not Software Design Level?

a. Interface Design
b. Architectural Design
c. Use case design.
d. High-level Design

3. Software Design Concept include____
A. Modularity
B. Refinement
C. Abstraction
D. All of above

4. How to measure quality and interaction between modules
A. Coupling
B. Cohesion
C. Both coupling and cohesion
D. None of above

5. Which is not a type of coupling?
A. Content
B. Common
C. Logical
D. Control

6. Which is not a type of cohesion?
A. Temporal
B. Procedural
C. Communicational
D. Stamp

7. What are the types of Data Flow Diagram?
A. Physical
B. Logical
C. Both physical and logical
D. None of above

8. Which is not a part of DFD symbol?
A. Processes
B. Data stores
C. Decision table
D. External entities

9. Use case and activity diagram is part of _____
A. Structure diagram

B. Behavior diagram
C. Both Behavior diagram and Structure diagram
D. None of above

10. Which symbol is not a part of use case diagram?
A. Actor
B. Decision making
C. System
D. Use case

Answer for Self Assessment

1.	D	2.	C	3.	D	4.	C	5.	C
6.	D	7.	C	8.	C	9.	B	10.	B

Review Questions

1. What is significance of good design?
2. Explain with example that how the modular design reduces the design complexity?
3. "Software Design is more an art than a science." Analyze.
4. Clients should not be forced to depend upon interfaces that they don't use. Explain.
5. Discuss the set of guidelines provided by the software design principles that helps to avoid having a bad design.
6. Dependency Inversion and Inversion of Control have the same meaning? Why or why not? Explain briefly.
7. A good design is the one that has low coupling. Explain.
8. Engineering is more about how you do the process than it is about what the final product looks like. Discuss.
9. The design of software is often depicted by graphs that show components and their relationships. Comment.
10. "Two modules that are tightly coupled are strongly dependent on each other." Explain why?

Further Reading

- Rajib Mall, Fundamentals of Software Engineering, Second Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering a Practioner's Approach, 5th edition, Tata McGraw
- Hill Higher education.
- Sommerville, Software Engineering, Sixth edition, Pearson Education.

Web Links

- http://en.wikipedia.org/wiki/Software_design
- http://www.developerdotstar.com/mag/articles/reeves_design.html

- https://www.geeksforgeeks.org/
- https://www.tutorialspoint.com/

Unit 06: User Interface Design

CONTENTS

Objective

After studying this unit, you will be able to:

- user interface design
- golden rules
- interface design process
- interface design activities

Introduction

The User Interface (UI) design principles are the methods/processes for designing the front end view of a software application so that the client or user may easily interact/use it without risk. If the user interface (UI) of a software application is appealing, clear, intelligible, and responsive, the software application will be very useful. Simply said, UI design principles give a user interface/view for a software product with a focus on look and feel and style, i.e. it is an evolutionary approach that provides maximum usability and the best user experience.

The process of developing or constructing the interfaces through which a user can communicate with a computer is known as user interface design. It is the software engineer who is in charge of creating the user interface. The software engineer ensures that the user interface is simple to comprehend, achieves goals, is enjoyable to use, and is encouraging and forgiving.

The user interface is a device or a programme (application programme) that allows a user to communicate with a computer and tell it what they want to accomplish or get from it. When consumers use a computer, the user interface is what they see. Whatever you see on the computer screen, as seen in the image below, is a user interface.

6.1 Different User Interface Design Principles

Because the popularity of a generated software application is based on the software product's user interface design, user interface design principles can help to improve the quality of the software application's user interface. The Graphical User Interface (GUI) or command line interface of the software application is the focus of these principles. The user interface should be simple, meaningful, and easy to grasp. There are several UI design principles to consider.

Given below are user interface design principles:

The Concept of Structural

The Concept of Simplicity

The Concept of Visibility

The Concept of Feedback

The Concept of Tolerance

The Concept of Reusability

1. The Concept of Structural

This is the first stage in designing the user interface for a software application, and it deals with the overall UI architecture. It offers the software application's structural data, as well as combining application-related data and separating undesirable items. This notion ensures that the software application is well-organized (meaningful), innovative (technology advancement), intelligible (software application purpose), and beneficial (satisfy certain criteria).

2. The Concept of Simplicity

This technique implies that the user interface of a software application should be simple, long-lasting, and user-friendly, with user control over the interface, sustainability, complexity clarification, accurate throughput response, maximum usability, and user language. We anticipate our users to make UI mistakes, justify our data, avoid developing a busy interface, and explain the rules for designing a user interface for a software programme. It will always assist in reducing the amount of activities required to perform all of the actions on the UI, as well as safeguarding the users' data in the software application.

3. The Concept of Visibility

This concept of UI design technique involves the visibility of user interface for a software application. It mainly deals with graphical user interface i.e. the alignment is correct or not, spelling checking, position of logo or banners, consistency, easy to navigate with status, colour& brightness of UI and without hesitation to the user. So that the look & feel of the interface should be perfect, clarity, progressive disclose, transparency and error preventive. We should emphasis on the performance issue of the UI.

4. The Concept of Feedback

This principle indicates the enhancement of user interface view depends of feedback. It always welcomes to change or modify the UI depends upon the user or client reviews. So the designer always analyse the process, view, any ambiguity issues, bugs, technological advancement, change of conditions and to make more user friendly of user interface of the software product. For users every action on the UI returns a meaningful and clear reaction or feedback.

5. The Concept of Tolerance

This is one of the concepts of user interface design for the software application which deals or affects the budgeting of the application. The UI designer will design the interface that will be always flexible and tolerant, so it reducing the cost of rework of user interfaces change. For any stage of the application the UI should be tolerable in sense.

6. The Concept of Reusability

This is the last user interface design technique. The user interface view should be reusable within the software application, according to the designer. Some internal or external components, component behaviour, and application consistency should be reused within it. So that the user does not have to remember the application's interconnected process flow on the user interface. It gives the user several shortcut keys to make working on that easier.

The user interface design mainly deals with graphical user interface (GUI) design, which involves multiple characteristics like graphics of the UI, Icons, Windows, Menus, font size, color, alignment, space between lines, etc. here we choose some specific techniques for the design of UI for the Software application.

The main idea behind user interface design to make it so comfortable for a user then, the user easily interacts with the software product. The tag line for the design of user interface process will be "The best User Interface design can attracts more customers".

6.2 Categories of User Interface

- Command-line Interface
- Menu-Based Interface
- Graphical User Interface

1. Command Line Interface

The command-line interface was the only mode of interaction with the computer in the starting days of computing. With the command-line interface, the user has to communicate with the computer using the textual instructions or we say commands.For example, if one needs to delete a file from the system. The command would be: del c:\file.txt

But, this kind of user interface is not user-friendly as it is hard to learn such commands, even they can be error-prone and unforgiving if an error occurs. Such an interface may be irritating. So, the command line interface was not meant for the casual user but the experienced users generally prefer command-line interface. Example: Unix operating system

Advantages

Many and easier to customizations options

Typically capable of more important tasks

Disadvantages

Relies heavily on recall rather than recognition

Navigation is often more difficult.

A text-based command line interface can have the following elements:

Command Prompt - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.

Cursor - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.

Command - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

2. Menu Based Interface

It is somewhat easy than the command line interface as users interacting with menu-based interface 'do not have to learn the command name', nor it had to put efforts in writing the commands. As here, the user does not have to type the commands the syntax error is automatically avoided.

A very popular example of a menu-based interface is ATM. A user uses menu buttons to select the options from the menu.

3. Graphical User Interface (GUI)

Over a period of time, there has been a tremendous evolution in the computer's user interface. As today's computer have a high-resolution screen with pointing devices (mouse). The modern-day computer's interface has windows, scroll bars, text boxes, buttons, icons, menus and to pick and point them we have a mouse.

Graphical user interfaces are easy to learn as compared to the command-line interface. GUI provides multiple windows to the user at the same time, to interact with the system and even the user is allowed to switch between the windows.

GUI Characteristics

Characteristics	Descriptions

GUI Elements

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:

Window - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.

Tabs - If an application allows executing multiple instances of itselfthey appear on the screen as separate windows. Tabbed Document Interface has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.

Menu - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.

Icon - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.

Cursor - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

Application specific GUI components

Application Window - Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.

Dialogue Box - It is a child window that contains message for the user and request for some action to be taken. For Example: Application generates a dialogue to get confirmation from user to delete a file.

Text-Box - Provides an area for user to type and enter text-based data.

Buttons - They imitate real life buttons and are used to submit inputs to the software.

Radio-button - Displays available options for selection. Only one can be selected among all offered

Check-box - Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.

List-box - Provides list of available items for selection. More than one item can be selected.

Advantages

Less expert knowledge is required to use it.

Easier to Navigate and can look through folders quickly in a guess and check manner.

The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages

Typically decreased options

Usually less customizable Not easy to use one button for tons of different variations.

6.3 Golden Rule

The user interface (UI) is a critical part of any software product. When it's done well, users don't even notice it. When it's done poorly, users can't get past it to efficiently use a product. To increase the chances of success when creating user interfaces, most designers follow interface design principles. Interface design principles represent high-level concepts that are used to guide software design. In this article, I'll share a few fundamental principles. These are based on Jakob Nielsen's 10 Usability Heuristics for UI Design, Ben Shneiderman's The Eight Golden Rules of Interface Design, and Bruce Tognazzini's Principles of Interaction Design. Most of the principles are applicable to any interactive systems — traditional GUI environments (such as desktop and mobile apps, websites) and non-GUI interfaces (such as voice-based interaction systems).

- Place users in control of the interface
- Make it comfortable to interact with a product
- Reduce cognitive load
- Make user interfaces consistent

1. Place users in control of the interface

Good UIs instill a sense of control in their users. Keeping users in control makes them comfortable; they will learn quickly and gain a fast sense of mastery.

Make actions reversible – be forgiving

This rule states that the user should always be able to rapidly return to their previous state. This lets customers to experiment with the product without fear of failure – knowing that mistakes can be readily undone encourages users to try out new things. On the other hand, requiring a user to be extremely cautious with every step they take results in a slower exploration and a nerve-wracking experience that no one wants.

Perhaps the most common GUIs where users have the 'Undo/Redo' option are text and graphics editors. While writing text or creating graphics, 'Undo' lets users make changes and go back step-by-step through changes that were made. 'Redo' lets users undo the undo, which means that once they go back a few steps, they are able to move forward through their changes again.

'Undo' can be extremely helpful when users choose system function by mistake. In this case, the undo function serves as an 'emergency exit,' allowing users to leave the unwanted state. One good example of such emergency exits is Gmail's notification message with an undo option when users accidentally delete an email.

Create an easy-to-navigate interface

Navigation should always be clear and self-evident. Users should be able to enjoy exploring the interface of any software product. Even complex B2B products full of features shouldn't intimidate users so that they are afraid to press a button. Good UI puts users in their comfort zone by providing some context of where they are, where they've been, and where they can go next:

Provide visual cues. Visual cues serve as reminders for users. Allow users to navigate easily through the interface by providing points of reference as they move through a product interface. Page titles, highlights for currently selected navigation options, and other visual aids give users an immediate view of where they are in the interface. A user should never be wondering, "Where am I?" or "How did I get to this screen?"

Predictability: Users should be provided with cues that help them predict the result of an action. A user should never be wondering, "What do I need to press in order to do my task?" or "What is this button for?"

Provide informative feedback – be acknowledging

Feedback is typically associated with points of action — for every user action, the system should show a meaningful, clear reaction. A system with feedback for every action helps users achieve their goals without friction.

UI design should consider the nature of interaction. For frequent actions, the response can be modest. For example, when users interact with an interactive object (such as a button), it's essential to provide some indication that an action has been acknowledged. This might be something as simple as a button changing color when pressed (the change notifies the user of the interaction). The lack of such feedback forces users to double-check to see if their intended actions have been performed.

For infrequent and significant actions, the response should be more substantial. For example, when filling out a password field in the signup form, good UI might inform users of the requirements for their password.

Show the visibility of system status

Users are much more forgiving when they have information about what is going on and are given periodic feedback about the status of the process. Visibility of system status is essential when users initiate an action that takes some time for a computer to complete. Users don't like to be left seeing nothing on the device screen while the app is supposed to be doing something. The use of progress indicators is one of the subtle aspects of UI design that has a tremendous impact on the comfort and enjoyment of users.

Good UI can comfort users by showing progress while the system is completing a task. Drop box is indicating the status of a document upload: the current progress and the amount of time left.

Accommodate users with different skill levels

Users of different skill levels should be able to interact with a product at different levels. Don't sacrifice expert users for an easy-to-use interface for novice or casual users. Instead, try to design for the needs of a diverse set of users, so it doesn't matter if your user is an expert or a newbie.

Adding features like tutorials and explanations is extremely helpful for novice users (just make sure that experienced users are able to skip this part).

Once users are familiar with a product, they will look for shortcuts to speed up commonly-used actions. You should provide fast paths for experienced users by enabling them to use shortcuts.

2. Make it comfortable for a user to interact with a product

Eliminate all elements that are not helping your users

Interfaces shouldn't contain information that is irrelevant or rarely needed. Irrelevant information introduces noise in UI —it competes with the relevant information and diminishes its relative visibility. Simplify interfaces by removing unnecessary elements or content that does not directly support user tasks. Strive to design UI in a way that all information presented on the screen will be valuable and relevant. Examine every element and evaluate it based on the value it delivers to users.

A good example of an app that follows the 'less is more' approach by avoiding overloading the interface with content or features is iA Writer.

The interface of iA Writer app is a clean typing sheet with no distractions. It allows users to focus on what they're writing and hides everything else.

Don't ask users for data they've already entered

Don't force users to have to repeat data they've previously entered. Users are easily annoyed by tedious data-entry sequences, especially when they have provided all the required information before. Good UI performs a maximum of work while requiring a minimum amount of information from users.

Avoid jargon and system-oriented terms

When designing a product, it's important to use language that is easy to read and understand. The system should speak the user's language, with words, phrases, and concepts familiar to the user, rather than jargon or system-oriented terms.

Apply Fitts's Law to interactive elements

Fitts Law states that the time to acquire a target is a function of the distance to and size of the target. This means that it's better to design large targets for important functions (big buttons are easier to interact with).

It's also important to remember that the time required to acquire multiple targets is the sum of the time to acquire each. Thus, when working on UI design, to increase the efficiency of an interaction, try to not only reduce distances and increase target sizes, but also reduce the total number of targets that users must interact with to complete a given task.

Design accessible interfaces

When we design products it's important to remember that a well-designed product is accessible to users of all abilities, including those with low vision, blindness, hearing impairments, cognitive impairments, or motor impairments. Good UI is accessible UI because improving your product's accessibility enhances the usability for all groups of users.

Color is one of the elements of an interface that has a strong impact on accessibility. People perceive color differently — some users can see a full range of colors, but many people can only make out a limited range of colors. Approximately 10 percent of men and one percent of women have some form of color blindness. When designing interfaces, it's better to avoid using color as the only way to convey information. Anytime you want color to convey information in the interface, you should use other cues to convey the information to those who cannot see the colors.

Use real-world metaphors

Using metaphors in UI design allows users to create a connection between the real world and digital experiences. Real-world metaphors empower users by allowing them to transfer existing knowledge about how things should look and work. Metaphors are often used to make the unfamiliar familiar. Take the recycle bin on your desktop, which holds deleted files, as an example – it's not a real trash bin, but it's visually represented in a way that helps you understand the concept more easily.

Good metaphors generate a strong connection to past experiences from the real world in users' minds. The recycle bin icon on Macs is similar to an actual bin, and it shows whether it has files in it.

When choosing a metaphor for UI, select the one that will enable users to grasp the finest details of the conceptual model. For example, when asking for credit card details for payment processing, you can reference a real-world physical card as an example.

Engineer for errors

Errors are inadvertent in the user journey. Bad error handling paired with useless error messages can fill users with frustration and lead them to abandon your app. A well-crafted error message, on the other hand, can turn a moment of frustration into a moment of conversion. An effective error message is a combination of explicit error notification together with hints for solving the problem.

Even better than writing good error messages is having UI design that prevents a problem from occurring in the first place. Try to either eliminate error-prone conditions or check for them and present users with a confirmation dialog before they commit to the action. For example, Gmail prompts you when you forget to insert an attachment.

The best designs have excellent error recovery while trying to prevent users from making those errors in the first place. Error prevention in Gmail shows a pop-up if users forget to insert an attachment after referencing one.

Protect a user's work

Ensure that users never lose their work. Users should not lose their work as a result of an error on their side (i.e. accidentally refresh a web page with a form that has user input), a system error, problems with an internet connection, or any other reason other than those that are completely unavoidable, like an unexpected power loss.

3. Reduce cognitive load

Cognitive load is the amount of mental processing power required to use a product. It's better to avoid making users think/work too hard to use your product.

Chunking for sequences of information or actions

In 1956, psychologist George Miller introduced the world to the theory of chunking. In his works, Miller says the human working memory can handle seven-plus-or-minus two "chunks" of information while we're processing information.

This rule can be used when organizing and grouping items together. For example, if you're UI forces users to enter telephone numbers without normal spacing it can result in a lot of incorrectly-captured phone numbers. People cannot typically scan clusters of ten or more digits to discover errors. That's exaReduce the number of actions required to complete a task

When designing a user interface, strive to reduce the total number of actions required from a user to achieve the goal. It's worth remembering the three-click rule, which suggests the user of a product should be able to find any information with no more than three mouse clicks.

Recognition over recall

One of the Jakob Nielsen's 10 usability heuristics advises promoting recognition over recall in UI design. Recognizing something is much easier than recalling it because recognition involves more cues in our brain (cues spread activation to related information in memory, and those cues help us remember information).

Designers can promote recognition in user interfaces by making information and functionality visible and easily accessible. Visual aids, such as tooltips and context-sensitive details, also help support users in recognizing information.Ctly why phone numbers are broken up into smaller pieces

Promote visual clarity

Good visual organization improves usability and legibility, allowing users to quickly find the information they are looking for and use the interface more efficiently.

When designing layouts:

- Avoid presenting too much information at one time on the screen. Construct a grid system design to avoid visual clutter.
- Remember the principle 'form follows function.' Make things look like they work.
- Apply the general principles of content organization such as grouping similar items together, numbering items, and using headings and prompt text.

4. Make user interfaces consistent

Consistency is an essential property of good UI—consistent design is intuitive design. Consistency is one of the strongest contributors to usability and Learnability. The main idea of consistency is the idea of transferable knowledge — let users transfer their knowledge and skills from one part of an app's UI to another and from one app to another app.

Visual consistency (style)

Users should never question the integrity of a product. The same colors, fonts, and icons should be present throughout the product. Be sure to always reference your design system manager to ensure you don't change visual styles within your product for no apparent reason. For example, a Submit but Avoid using different styles for elements on different pages of the site. Users should not have to wonder whether a transformed button like this example means the same thing.

Functional consistency (behavior)

Consistency of behavior means the object should work in the same way throughout the interface. The behavior of interface controls, such as buttons and menu items, should not change within a product. Users don't want surprises or changes in familiar behavior — they become easily frustrated when things don't work. This can inhibit learning and stop users from feeling confident about consistency in the interface. Do not confuse your user — keep actions consistent by following "The principle of least surprise," to have the interface behave the way users expect it to.

Consistent with user expectations

People have certain expectations about the apps/websites they use. Designing your product in a way that contradicts a user's expectations is one of the worst things you can do to a user. It doesn't matter what logical argument you provide for how something should work or look. If users expect it to work/look a different way, you will face a hard time changing those expectations. If your approach offers no clear advantage, go with what your users expect.

Follow platform conventions. Your product should be consistent with standards dictated by platform guidelines. Guidelines ensure that your users can understand individual interface elements in your design.

Don't reinvent patterns. For most design problems, proper solutions already exist. These solutions are called patterns. Popular patterns become conventions and the majority of users are familiar with them. Not taking this solution into account and continuing to design your own solution can lead to challenges for users. In most cases, breaking design conventions results in a frustrating user experience — you'll face usability problems not necessarily because your solution will be wrong, but because users won't be familiar with it.

Don't try to reinvent terminology. Avoid using new terms when there are words available that users already know. Users spend most of their time in other apps and on other sites, so they have certain expectations about naming. Using different words might confuse them.Tton on one page of your site should look the same on any other page.

6.4 Interface Design Models

Four different models come into play when a user interface is to be analyzed and designed

User model: a profile of all end users of the system Users can be categorized as:

Novices: No syntactic and little semantic knowledge of the system.

Knowledgeable, intermittent users: reasonable knowledge of the system.

Knowledgeable, frequent users: good syntactic and semantic knowledge of the system

Design model: a design realization of the user model that incorporates data, architectural, interface, and procedural representations of the software.

Mental model (system perception): the user's mental image of what the interface is. The user's mental model shapes how the user perceives the interface and whether the UI meets the user's needs.

Implementation model: the interface "look and feel of the interface" coupled with all supporting information (documentation) that describes interface syntax and semantics.

6.5 Interface Design Process

The analysis and design process for UIs is iterative and can be represented using a spiral model.

The user interface analysis and design process encompasses four framework activities:

1. User, task and environment analysis and modeling.

2. Interface design

3. Interface construction (implementation)

4. Interface validation

User, task and environment analysis and modeling

In this phase focus is based on the profile of users who will interact with the system, i.e. understanding, skill and knowledge, type of user, etc,based on the user's profile users are made into categories. From each category requirements are gathered.

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked

- Where will the interface be located physically?

- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

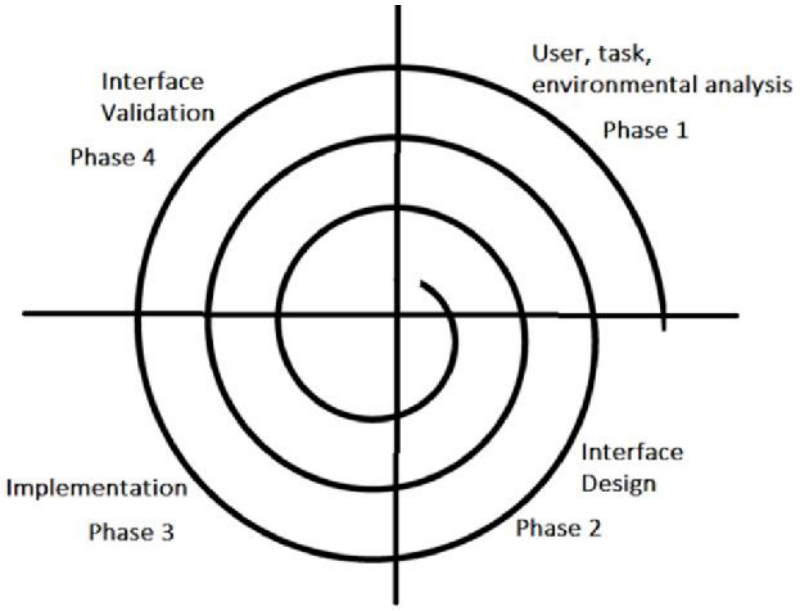
Interface design analysis: The people (end-users) who will interact with the system through the interface;

The tasks that end-users must perform to do their work,

The content that is presented as part of the interface,

The environment in which these tasks will be conducted

Interface design process



User Analysis

- User Interviews
- Sales input
- Marketing input
- Support input

Task Analysis and Modeling: The goal of task analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

Interface Design

Once interface analysis has been completed, all tasks required by the end-user have been identified in detail.Using information developed during interface analysis (Section 12.3), define interface objects and actions (operations).

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

Depict each interface state as it will actually look to the end-user.Indicate how the user interprets the state of the system from information provided through the interface.

Interface Design Patterns

The complete UI	Page layout
Forms and input	Tables
Direct data manipulation	Navigation
Searching	Page elements
E-Commerce	

Design Issues:

Response time	Help facilities
Error handling	Application accessibility
Internationalization	
Menu and command labeling	

Interface construction and implementation

The implementation activity starts with the development of prototype (model) that enables usage scenarios to be evaluated.

As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

Interface Validation

This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly and it should be able to handle a variety of tasks.

It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

6.6 Interface Design Activities

There are a number of activities performed for designing user interface.

- GUI Requirement Gathering
- User analysis
- Task analysis
- GUI Design & implementation
- Testing

GUI Requirement Gathering

In this phase all functional and non-functional requirements are gathered for GUI. This can be taken from user and their existing software solution.

User analysis

The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user.

Task analysis

Designers have to analyze what task is to be done by the software solution.

Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation.

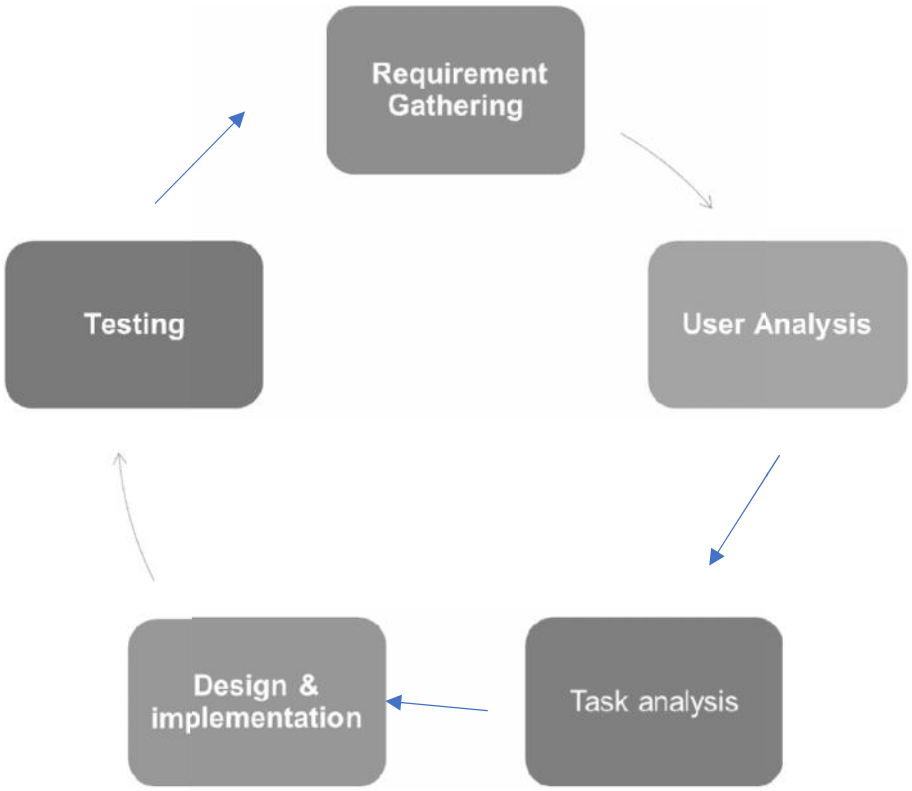
GUI Design & implementation

Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background.

Testing

Testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

Interface Design Activities



Summary

- User Interface Design is a process of designing or fabricating the interfaces through which the user can communicate with the computer.
- There are two types of User Interface

  Command Line Interface

  Graphical User Interface

- Golden rules stated by Theo Mandel that must be followed during the design of the interface.
- Place the user in control
- Reduce the users memory load
- Make the interface consistent
- User interface Design Principles
- Learnability
- Flexibility
- Robustness
- Simplicity
- Visibility

Four different models come into play when a user interface is to be analyzed and designed

- User model
- Design model

- Mental model
- Implementation model

Keywords

- Interface Design
- Design Model
- Implementation
- Visual Clarity
- Command Prompt
- Graphical Interface Design

Self Assessment

1. Which is part of user interface?
A. Command Line Interface
B. Graphical User Interface
C. Both Command Line Interface and Graphical User Interface
D. None of above

2. Which is not a part of Graphical User Interface?
A. Icon
B. Button
C. Command
D. Cursor

3. Command Line Interface is___
A. Minimal memory usage
B. Great for slow-running computers
C. This method relies primarily on the keyboard
D. All of above

4. Which is not a part of Command Line Interface?
A. UNIX
B. DOS
C. LINUX
D. None of above

5. Which is not a part of Graphical User Interface?
A. Windows XP
B. UNIX
C. LINUX
D. None of above

6. Graphical User Interface is____
A. Memorizing command lists is not necessary
B. Allows for running multiple applications, programs, and tasks simultaneously
C. Follow WYSIWYG (What You See Is What You Get)
D. All of above

7. Which is not a part of Interface Design Model?
A. User model

B. Design model
C. SDLC model`
D. Mental model

8. How many phases in Interface design process?
A. 2
B. 4
C. 3
D. 5

9. Which of the following devices are mainly responsible for the user interface?
A. Input and output devices
B. Memory devices
C. Processor
D. None of the above

10. Which is not a part of Interface design process?
A. Interface design
B. Testing
C. Interface construction
D. User, task and environment analysis

Answer for Self Assessment

1.	C	2.	C	3.	D	4.	C	5.	B
6.	D	7.	C	8.	B	9.	A	10.	B

Review Questions

1. What is significance of user interface design in software engineering?
2. Explain golden rules with suitable example.
3. What are the different elements of Graphical User Interface?
4. Discuss User interface Design Principles.
5. What are the Guidelines for user interface designing?
6. Discuss any two interface design models with example.
7. What are the four activities for interface design process?
8. Explain interface validation.

Further Reading

Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.

Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.

R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata McGraw Hill Higher education.

Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Links

http://softwareengineeringhub.blogspot.in/2010/03/system-requirementsspecification.html

http://computersciencesource.wordpress.com/2010/03/14/softwareengineering-object-oriented-modelling/

https://www.geeksforgeeks.org/

https://educatech.in/

https://www.javatpoint.com/

https://www.tutorialspoint.com/

Unit 07: Standards

CONTENTS

Objectives

After studying this unit, you will be able to:

- Good coding practices
- Coding standards and code reusability
- Documentation standards

Introduction

The process of converting a system's design into a computer language format is known as coding. The coding step of software development is responsible for converting design specifications into source code. The task of coding is to convert design specifications into source code. A coding standard ensures that code generated by various engineers has a consistent appearance. It facilitates the understanding of the code. Encourages the use of excellent programming practices.

Debugging, testing, and change should be simple with proper programming. This is accomplished by making the source code as simple and transparent as feasible. “Simple is great,” says an old adage. The trademarks of good programmes are simplicity, clarity, and elegance. Inadequate design is characterised by obscurity, cunning, and complexity. Structured coding practices, decent coding style, adequate supporting documents, good internal comments, and other factors improve source code clarity. Production of high quality software requires that the programming team should have a thorough understanding of duties and responsibilities and should be provided with a well-defined set of requirements, an architectural design specification, and a detailed design description.

Certain principles must be followed when creating and implementing software or apps to ensure that your programme satisfies the needed attributes to perform as intended. These standards also ensure that the final output meets your clients' expectations. The ability to build computer software, apps, and websites is made possible through coding. The coding step of the software development life cycle is the longest. Coding normally begins when the design documentation is complete. This phase will almost totally be devoted to the developer's efforts.The development of code that will implement the design is the focus of the coding phase of the software life-cycle. This code is written

in a programming language, which is a formal language. Programming languages have progressed over time from sequences of ones and zeros that a computer can directly parse, to symbolic machine code, assembly languages, and finally higher-level languages that humans can understand.

7.1 Common Errors

A mismatch between the programme and its specification is a common definition of a software error. To put it another way, a software error occurs when a programme fails to perform as expected by its end user. Depending on the type of programme and the specific bug involved, errors might result in a wide range of issues.

Others have the potential to cause errors in the performance of the program that result in the program behaving in unexpected ways. Sometimes a software error can even cause a program to shut down completely.

Following are the most common software errors. This helps you to identify errors systematically and increases the efficiency and productivity of software testing.

Syntax error	Logical error
Runtime error	Enumerated error
Memory leaks	Null dereferencing
Synchronization error	String handling errors
Lack of unique addresses	freeing an already freed resource

Example: We have discussed below different types of errors with examples:

Syntax error: A syntax error is an error in the source code of a program. Since computer programs must follow strict syntax to compile correctly, any aspects of the code that do not conform to the syntax of the programming language will produce a syntax error.

Syntax errors are small grammatical mistakes, sometimes limited to a single character. For example, a missing semicolon at the end of a line or an extra bracket at the end of a function may produce a syntax error.

Logical error: A logic error (or logical error) is a 'bug' or mistake in a program's source code that results in incorrect or unexpected behaviour. It is a type of runtime error that may simply produce the wrong output or may cause a program to crash while running. Many different types of programming mistakes can cause logic errors. For example, assigning a value to the wrong variable may cause a series of unexpected program errors. Multiplying two numbers instead of adding them together may also produce unwanted results.

Runtime error: If there are no syntax errors, Java may detect an error while your program is running. You will get an error message telling you the kind of error, and a stack trace that tells not only where the error occurred, but also what other method or methods you were in.

User Interface Errors: Missing/Wrong Functions, Doesn't do what the user expects, Missing information, Misleading, Confusing information, Wrong content in Help text, Inappropriate error messages. Performance issues - Poor responsiveness, Can't redirect output, inappropriate use of key board.

Error Handling: Inadequate – protection against corrupted data, tests of user input, version control; Ignores – overflow, data comparison, Error recovery – aborting errors, recovery from hardware problems.

Boundary Related Errors: Boundaries in loop, space, time, memory, mishandling of cases outside boundary.

Calculation Errors: Bad Logic, Bad Arithmetic, Outdated constants, Calculation errors, incorrect conversion from one data representation to another, Wrong formula, incorrect approximation.

Initial and Later States: Failure to – set data item to zero, to initialize a loop-control variable, or re-initialize a pointer, to clear a string or flag, Incorrect initialization.

Control Flow Errors: Wrong returning state assumed, Exception handling based exits, Stack underflow/overflow, Failure to block or unblock interrupts, Comparison sometimes yields wrong result, Missing/wrong default, Data Type errors.

Errors in Handling or Interpreting Data: Unterminated null strings, overwriting a file after an error exit or user abort.

Race Conditions: Assumption that one event or task finished before another begins,

Resource races, Tasks starts before its prerequisites are met, Messages cross or don't arrive in the order sent.

Load Conditions: Required resources are not available, No available large memory area, Low priority tasks not put off, doesn't erase old files from mass storage, and doesn't return unused memory.

Hardware: Wrong Device, Device unavailable, underutilizing device intelligence, Misunderstood status or return code, Wrong operation or instruction codes.

Source, Version and ID Control: No Title or version ID, Failure to update multiple copies of data or program files.

Testing Errors: Failure to notice/report a problem, Failure to use the most promising test case, Corrupted data files, Misinterpreted specifications or documentation, Failure to make it clear how to reproduce the problem, Failure to check for unresolved problems just before release, Failure to verify fixes, Failure to provide summary report.

7.2 Programming Practices

General coding guidelines provide the programmer with a set of best practices which can be used to make programs easier to read and maintain. Unlike the coding standards, the use of these guidelines is not mandatory. However, the programmer is encouraged to review them and attempt to incorporate them into his/her programming style where appropriate. Most of the examples use the C language syntax but the guidelines can be applied to all languages.

Line Length

It is considered good practice to keep the lengths of source code lines at or below 80 characters.

Lines longer than this may not be displayed properly on some terminals and tools. Some printers will truncate lines longer than 80 columns. FORTRAN is an exception to this standard.

Its line lengths cannot exceed 72 columns.

Spacing

The proper use of spaces within a line of code can enhance readability. Good rules of thumb are as follows:

A keyword followed by a parenthesis should be separated by a space.

A blank space should appear after each comma in an argument list.

All binary operators except "." should be separated from their operands by spaces.

Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("—") from their operands.

Casts should be made followed by a blank space.

 *Example*: Bad:

```
cost=price+ (price*sales tax);

fprintf(stdout," The total cost is %5.2f\n", cost);
```

Better:

```
cost = price + (price * sales tax);

fprintf (stdout, "The total cost is %5.2f\n", cost);
```

Wrapping Lines

When an expression will not fit on a single line, break it according to these following principles:

Break after a Comma

Example: fprintf (stdout, "\nThere are %d reasons to use standards\n", num_reasons);

Break after an Operator

Example: long int total_apples = num_my_apples + num_his_apples +

num_her_apples;

Prefer Higher-level Breaks to Lower-level Breaks

Example: Bad:

longName1 = longName2 * (longName3 + LongName4 – longName5) + 4 * longName6;

Better:

longName1 = longName2 * (longName3 + LongName4 – LongName5) + 4 * longName6;

Align the new line with the beginning of the expression at the same level on the previous line.

 ***Example***: total_windows = number_attic_windows + number_second_floor_windows + number_first_floor_windows;

Variable Declarations

Variable declarations that span multiple lines should always be preceded by a type.

 ***Example***: Acceptable:

int price, score;

Acceptable:

int price;

int score;

Not Acceptable:

int price,

score;

Program Statements

Program statements should be limited to one per line. Also, nested statements should be avoided when possible.

 ***Example***: Bad:

number_of_names = names. length; b = new JButton [number_of_names];

Better:

number_of_names = names.length;

b = new JButton [number_of_names];

Use of Parentheses

It is better to use parentheses liberally. Even in cases where operator precedence unambiguously dictates the order of evaluation of an expression, often it is beneficial from a readability point of view to include parentheses anyway.

In-line Comments

In-line comments promote program readability. They allow a person not familiar with the code to more quickly understand it. It also helps the programmer who wrote the code to remember details forgotten over time. This reduces the amount of time required to perform software maintenance tasks.

As the name suggests, in-line comments appear in the body of the source code itself. They explain the logic or parts of the algorithm which are not readily apparent from the code itself. In-line comments can also be used to describe the task being performed by a block of code.

In-line comments should be used to make the code clearer to a programmer trying to read and understand it. Writing a well-structured program lends much to its readability even without inline comments. The bottom line is to use in-line comments where they are needed to explain complicated program behavior or requirements. Use in-line comments to generalize what a block of code, conditional structure, or control structure is doing. Do not use overuse in-line comments to explain program details which are readily obvious to an intermediately skilled programmer.

Coding for Efficiency vs. Coding for Readability

There are many aspects to programming. These include writing software that runs efficiently and writing software that is easy to maintain. These two goals often collide with each other.

Creating code that runs as efficiently as possible often means writing code that uses tricky logic and complex algorithms, code that can be hard to follow and maintain even with ample in-line comments.

If a more complicated algorithm offers only small gains in the speed of a program, the programmer should consider using a simpler algorithm. Although slower, the simpler algorithm will be easier for other programmers to understand.

Meaningful Error Messages

Error handling is an important aspect of computer programming. This not only includes adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

Error messages should be meaningful. When possible, they should indicate what the problem is, where the problem occurred, and when the problem occurred. A useful Java exception handling feature is the option to show a stack trace, which shows the sequence of method calls which led up to the exception.

Code which attempts to acquire system resources such as dynamic memory or files should always be tested for failure.

Error messages should be stored in way that makes them easy to review. For non-interactive applications, such as a program which runs as part of a corn job, error messages should be logged into a log file. Interactive applications can either send error messages to a log file, standard output, or standard error. Interactive applications can also use popup windows to display error messages.

Reasonably Sized Functions and Methods

Software modules and methods should not contain an excessively large number of lines of code. They should be written to perform one specific task. If they become too long, then chances are the task being performed can be broken down into sub-tasks which can be handled by new routines or methods.

A reasonable number of lines of code for routine or a method is 200. This does not include documentation, either in the function/method header or in-line comments.

Number of Routines per File

It is much easier to sort through code you did not write and you have never seen before if there are a minimal number of routines per file. This is only applicable to procedural languages such as C and FORTRAN. It does not apply to C++ and Java where there tends to be one public class definition per file.

7.3 Coding Standards

Coding standards help to ensure safety, security, and reliability.Coding standards are collections of coding rules, guidelines, and best practices. Compliance with industry standards (e.g.,

ISO).Consistent code quality – no matter who writes the code.Software security from the start.Reduced development costs and accelerated time to market.

Coding standards is a series of procedures that can be defined for a particular programming language specifying a programming style, the methods and different procedures. A coding standard makes sure that all the developers working on the project are following certain specified guidelines.

Objective of Coding Standards

A coding standard gives a uniform appearance to the codes written by different engineers.It improves readability, and maintainability of the code and it reduces complexity also.It helps in code reuse and helps to detect error easily.It promotes sound programming practices and increases efficiency of the programmers.

- Limited use of globals
- Standard headers for different modules
- Naming conventions for local variables, global variables, constants and functions
- Indentation
- Error return values and exception handling conventions
- Avoid using a coding style that is too difficult to understand
- Avoid using an identifier for multiple purposes
- Code should be well documented
- Length of functions should not be very large
- Try not to use GOTO statement

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.

Standard headers for different modules: The header of different modules should follow some standard format and information

Name of the module

Date of module creation

Author of the module

Modification history

Different functions supported in the module along with their input output parameters

Global variables accessed or modified by the module

Naming conventions for local and global variables, constants and functions: Meaningful and understandable variables name helps anyone to understand the reason of using it

Need to follow case sensitive rule as per language

It is better to avoid the use of digits in variable names.

Indentation: There must be a space after giving a comma between two function arguments.

Each nested block should be properly indented and spaced.

Proper Indentation should be there at the beginning and at the end of each block in the program.

All braces should start from a new line and the code following the end of braces also start from a new line.

Error return values and exception handling conventions: The way error conditions are reported by different functions in a program are handled should be standard within an organization.

For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

Avoid using a coding style that is too difficult to understand: Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.

Code should be well documented: The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.

Length of functions should not be very large: Lengthy functions are very difficult to understand. That's why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.

Try not to use GOTO statement: GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.

Classes, Subroutines, Functions, and Methods: Keep subroutines, functions, and methods reasonably sized. This depends upon the languagebeing used. A good rule of thumb for module length is to constrain each module to one functionor action (i.e. each module should only do one "thing"). If a module grows too large, it is usuallybecause the programmer is trying to accomplish too many actions at one time.

The names of the classes, subroutines, functions, and methods shall have verbs in them.

That is the names shall specify an action, e.g. "get name", "compute temperature".

Source File: The name of the source file or script shall represent its function. All of the routines in a file shall have a common purpose.

Use of Braces: In some languages, braces are used to delimit the bodies of conditional statements, control constructs, and blocks of scope. Programmers shall use either of the following bracing styles:

```
for (int j = 0; j < max_iterations; ++j)
{
/* Some work is done here. */
}
```

or the Kernighan and Ritchie style:

```
for (int j = 0; j < max_iterations; ++j) {
/* Some work is done here. */
}
```

It is felt that the former brace style is more readable and leads to neater-looking code than the latter style, but either use is acceptable.

Compiler Warnings: Compilers often issue two types of messages: warnings and errors. Compiler warnings normally do not stop the compilation process. However, compiler errors do stop the compilation process, forcing the developer to fix the problem and recompile. Compiler and linker warnings shall be treated as errors and fixed. Even though the program will continue to compile in the presence of warnings, they often indicate problems which may affect the behavior, reliability and portability of the code. Some compilers have options to suppress or enhance compile-time warning messages. Developers shall study the documentation and/or man pages associated with a compiler and choose the options which fully enable the compiler's code-checking features.

Advantages of Coding Guidelines

- Coding guidelines increase the efficiency of the software and reduces the development time.
- Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.
- If coding guidelines are maintained properly, then the software code increases readability and understandability thus it reduces the complexity of the code.
- It reduces the hidden cost for developing the software.

Example: Coding standards

IEC 61508: "Functional safety of electrical/ electronic /programmable electronic safety-related systems"

ISO 26262: "Road vehicles — functional safety"

EN 50128: "Railway applications — Communication, signaling, and processing systems — Software for railway control and protection systems"

IEC 62061: "Safety of machinery: Functional safety of electrical, electronic and programmable electronic control systems"

MISRA

MISRA provides coding guidelines for C and C++.

The MISRA C coding standard was originally written for the automotive embedded software industry.

MISRA standards for C and C++ are widely used by embedded industries — including aerospace and defense, telecommunications, medical devices, and rail.

JSFC++

Joint Strike Fighter Air Vehicle C++ (JSF AV C++) is a coding standard developed by Lockheed Martin.

It helps programmers develop error-free code for safety-critical systems.

AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) was founded in order to standardize the functionality of automotive embedded software

AUTOSAR is a worldwide partnership that is comprised of over 180 automotive manufacturers, automotive suppliers, tool vendors, semiconductor vendors.

CERT

An estimated 82% of software vulnerabilities are caused by coding errors.

CERT is a secure coding standard that supports commonly used programming languages such as C, C++, and Java.

The standards are developed through a broad-based community effort.

The rules and recommendations target insecure coding practices and undefined behaviors that lead to security risks.

7.4 Code Reusability

Code reusability defines the methodology you can use to use similar code, without having to re-write it everywhere. There are many techniques to make code reusable in your applications. This technique follows the general programming philosophy of Don't Repeat Yourself (DRY).

Code can be reused when it can be

Easily extended and adapted for the new application.Ported to different hardware if needed.Shown to be free from defects or problems that affect the reliability, safety, or security of the new application.

There are many scenarios and methods

- Inheritance
- Functions
- Libraries
- Forking

Inheritance:

Inheritance has been in the programming for quite a long time and is used in Object-Oriented Programming paradigms. Inheritance lets you use the base class's functions and members in other classes.

Functions:

A return type that denotes what type of data would be returned or a void.An identifier that is used to make a call to this function.Function body that is the code block containing the code to be executed when this function is called.Zero or more parameters to provide values to the code block variables.Function calls are another way of reusing the code from various parts of your application.You create small code blocks of an algorithm or logic-based source code and provide it a name, identifier.

Libraries:

Libraries are a set of compiled programs, most often used by larger and enterprise level software, to support the execution since they can be executed upon call.

Forking:

Forking is a process of using someone else's source code and making changes to it to fit your own needs, to distribute, use, and share and so on.User can also fork someone else's source code and write some more details to it. Make some bug fixes, add some patches and share it.

This method has been widely used by various companies, to fork open source projects, make a few changes to the "Under the hood" and then distribute it with their own label and tag.

7.5 Documentation

All these documents are a vital part of good software development practice. Documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process.

Significance of Documentation

Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.Documents help the users in effectively using the system.Good documents help in effectively handling the manpower turnover problem. Production of good documents helps the manager in effectively tracking the progress of the project.

Types of documents

- User manual
- Operational manual
- Requirements Document
- Design document
- Technical document
- Testing document
- List of Known Bugs

User manual − It describes instructions and procedures for end users to use the different features of the software.

Operational manual − It lists and describes all the operations being carried out and their inter-dependencies.

Design Document − It gives an overview of the software and describes design elements in detail. It documents details like data flow diagrams, entity relationship diagrams.

Requirements Document − It has a list of all the requirements of the system as well as an analysis of viability of the requirements. It can have user cases, real life scenarios, etc.

Technical Documentation − It is a documentation of actual programming components like algorithms, flowcharts, program codes, functional modules, etc.

Testing Document − It records test plan, test cases, validation plan, verification plan, test results, etc. Testing is one phase of software development that needs intensive documentation.

Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.

External documentation - It is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

- **Guidelines for creating the documents**

- Documentation should be from the point of view of the reader
- Document should be unambiguous
- There should be no repetition
- Industry standards should be used
- Documents should always be updated
- Any outdated document should be phased out after due recording of the phase out

Advantages of Documentation

- Keeps track of all parts of a software or program
- Maintenance is easier
- Programmers other than the developer can understand all aspects of software
- Improves overall quality of the software
- Assists in user training
- Ensures knowledge de-centralization, cutting costs and effort if people leave the system abruptly

Summary

- A software error is present in a program when the program does not do what its end user expects.
- Errors can cause a wide variety of different problems depending on the kind of program and the particular kind of bug involved.
- Structured Programming is a method of planning programs that avoids the branching category of control structures.
- The single most important idea of structured programming is that the code you write should represent a clear, simple and straightforward solution to the problem at hand.
- General coding guidelines provide the programmer with a set of best practices which can be used to make programs easier to read and maintain.
- In-line comments promote program readability. They allow a person not familiar with the code to more quickly understand it.
- If a more complicated algorithm offers only small gains in the speed of a program, the programmer should consider using a simpler algorithm.
- Compiler warnings normally do not stop the compilation process. However, compiler errors do stop the compilation process, forcing the developer to fix the problem and recompile.

Keywords

Compiler Errors: Compiler errors do stop the compilation process, forcing the developer to fix the problem and recompile.

Compiler Warnings: Compiler warnings is a message which normally do not stop the compilation process.

Error Handling: Error handling includes adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

In-line Comments: In-line comments allow a person not familiar with the code to more quickly understand it.

Sequence: Sequence is a series of program statements are executed one after the other, in the order in which they appear in the source code.

Series: A series of statements is executed repeatedly until some termination condition is met.

Software Error: A software error is a mismatch between the program and its specification.

Structured Programming: Structured Programming is a method of planning programs that avoids the branching category of control structures.

Self Assessment

1. What are the common types of error in coding?

A. Memory leaks

B. Null dereferencing

C. Null dereferencing

D. All of above

2. Which is not a syntax error?

A. printf ("this is code');

B. printf ("this is code");

C. printf ('this is code');

D. none of above

3. Which is not a coding error?

A. Logical error

B. Lack of unique addresses

C. Transformation

D. None of the above

4. A good coding practice is?

A. Portability

B. Generality

C. Enhance Code Readability

D. All of above

5. Coding is___

A. To translate the design of system into a computer language format:

B. To reduce the cost of later phases
C. Making the program more readable
D. All ofabove

6. Coding standards are__
A. Help to ensure safety, security, and reliability.
B. Collections of coding rules, guidelines, and best practices
C. Compliance with industry standards
D. All of above

7. Which is not a part of coding standard?
A. ISO 2010
B. IEC 61508
C. EN 50128
D. IEC 62061

8. Which coding standard is for Railway applications?
A. ISO 26262
B. IEC 62061
C. EN 50128
D. None of the above

9. Lockheed Martin developed____
A. MISRA
B. JSFC++
C. AUTOSAR
D. All of above

10. MISRA coding standard is used for__
A. Fighter Air Vehicle
B. Automotive embedded software industry
C. Toy industry
D. None of above

11. What are the different type of documents?
A. Operational manual
B. List of Known Bugs
C. User manual
D. All of above

12. Code can be reuse in case_
A. Easily extended and adapted for the new application.
B. Ported to different hardware if needed.

C. Shown to be free from defects
D. All of above

13. Which is not a Code reusability methods?
A. Inheritance
B. Functions
C. Classes
D. Forking

14. What are different Code reusability methods?
A. Functions
B. Libraries
C. Forking
D. All of above

15. Function call is part of __
A. Testing
B. Classes
C. Code reusability
D. None of above

Answer for Self Assessment

1.	D	2.	B	3.	C	4.	D	5.	D
6.	D	7.	A	8.	C	9.	B	10.	B
11.	D	12.	D	13.	C	14.	D	15.	C

Review Questions

1. Discuss different types of common errors with example.
2. Explain the technique used for organizing and coding computer programs.
3. Discuss the basic principles of structured programming.
4. Describe the building blocks of structured programming.
5. What are the different types of control blocks? Discuss.
6. Discuss the best programming practices which can be used to make programs easier to read and maintain.
7. Illustrate how the use of spaces within a line of code can enhance readability.
8. Discuss the general coding standards used by the developer when writing code.

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.

- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Links

- http://software-engineering142.blogspot.in/2009/04/coding-phase.html
- http://www.computer.org/portal/web/certification/resources/code_of_ethics
- http://www.indeed.com/q-Software-Process-Engineer-Coding-StandardPractice-jobs.html
- http://www.win.tue.nl/~wstomv/edu/jpid/Coding_Standard_Motivation-
- 4up.pdf

Unit 08: Software Testing

CONTENTS

Objective

After studying this unit, you will be able to:

- Discuss the fundamentals of Testing
- Test design and plan
- Test case template

Introduction

Testing is the operation of the software with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between expected and actual results.Software Testing is evaluation of the software against requirements gathered from users and system specifications.

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

Failure: This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

Test case: This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

Test suite: This is the set of all test cases with which a given software product is to be tested.

Aim of Testing

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus, testing provides a practical way of reducing defects in a system and increasing the users 'confidence in a developed system.

8.1 Software Testing

Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways.

By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear — generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects – or bugs – will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable – and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

Testing is usually performed for the following purposes:

To Improve Quality

As computers and software are used in critical applications, the outcome of a bug can be severe.Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed spaceshuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugscan cause disasters. The so-called year 2000 (Y2K) bug has given birth to a cottage industry ofconsultants and programming tools dedicated to making sure the modern world doesn't cometo a screeching halt on the first day of the next century. In a computerized embedded world, thequality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimumrequirement of quality, means performing as required under specified circumstances.Debugging, a narrow view of software testing, is performed heavily to find out design defectsby the programmer. The imperfection of human nature makes it almost impossible to

make amoderately complex program correct the first time. Finding the problems and get them fixed, isthe purpose of debugging in programming phase.

For Verification & Validation (V&V)

Just as topic Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We cannot test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors – functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. The table below illustrates some of the most frequently cited quality considerations.

8.2 Fundamentals of Testing

Testing software can be considered as the only destructive (psychologically) step in the entire life cycle of software production. Although all the initial activities aimed at building a product, the testing is done to find errors in software.

Benefits of Testing

The following are the benefits of testing:

It reveals the errors in the software.

It ensures that software is functioning as per specifications and it meets all the behavioral requirements as well.

The data obtained during testing is indicative of software reliability and quality as a whole.

It indicates presence of errors and not absence of errors.

Testing Principles

Before coming up with ways to design efficient test cases, one must understand the basic principles of testing:

- ***Test Cases must be Traceable to Requirements***: Because software testing reveals errors, so, the severe defects will be those that prevent the program from acting as per the customer's expectations and requirements.
- ***Test Planning must be done before Beginning Testing***: Test planning can begin soon after the requirements specification is complete and the detailed test cases can be developed after the design has been fixed.
- ***Pareto Principle Applies to Software Testing***: Pareto principle states that 80 percent of the uncovered errors during testing will likely be traceable to 20 percent of all the program components. Thus, the main aim is to thoroughly test these 20 percent components after identifying them.
- ***Testing should begin with Small and End in Large***: The initial tests planned and carried out are usually individual components and as testing progresses the aim shifts to find errors in integrated components of the system as whole rather than individual components.
- ***Exhaustive Testing is Impossible***: For a normal sized program the number of permutations of execution paths is very huge. Thus, it is impossible to execute all the combinations possible. Thus, while designing a test case it should be kept in minds that it must cover the maximum logic and the component-level design.

- ***Efficient Testing can be conducted only by a Third Party***: The highest probability of Notes finding errors exists when the testing is not carried by the party which develops the system.

8.3 Testing Objectives

Various testing objectives are:

- Executing a program in order to find errors.
- A good test case is the one that has a high probability of finding an undiscovered error.
- A successful test is the one that reports an as – yet undiscovered error.

Testability

A program developed should be testable i.e. it should be possible to test the program. The testability of a program can be measured in terms of few properties like: operability, observability, controllability, decomposability, simplicity, stability, understandability, etc.

These characteristics are mentioned here:

- The probability of finding error should be high. In order to achieve this, tester must understand the actual functionality of the software and think of a suitable condition that can lead to failure of the software.
- A test case must be non-redundant. Because the testing time and resources are limited, it will waste a lot of time to conduct the same test again and again. Every test must have a distinct purpose.
- A test case must be of the best quality. In a group of similar test cases, the one that covers the maximum scenarios to uncover maximum errors should only be used.
- A good test case should neither be too simple nor too complex. A complex test that includes several other tests can lead to masking of errors. Thus, each test case should be executed separately.

Error, Fault, Failure

Many people associate quality of a software system with a suggestion that some software troubles are estimated to happen (or not). And in the case if such troubles do occur, the negative influence is estimated to be minimal. Clue to the correctness perspective of software quality is in the concept of failure, fault, and error.

Error

An error is a person act that generates an erroneous result. The term error is used in two different ways. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is error refers to the difference between the actual output of software and the correct output. The term error is a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition

Fault

Fault is a condition that causes a system to fail in performing its required function. It refers to an underlying condition within software that causes failure to happen. A fault is the basic reason for software malfunction and is synonymous with the commonly used term bug. It is an incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner.

Failure

Failure is the inability of a system or component to perform a required function according to its specifications. It refers to a behavioral deviation from the user wants or the product specification. A software failure occurs if the behavior of the software is different from the specified behavior.

Problems in Testing

- Software Testing is a critical element of Software Quality Assurance
- It represents the ultimate review of the requirements, specification, the design and the code.
- It is the most widely used method to ensure Software Quality.
- For most software, exhaustive testing is not possible.
- Many organizations spend 40-50% of development time in testing.
- Testing is the most expensive way to remove defects during software development.
- Problems in testing are one of the major contributors to cost and scheduled overruns.

8.4 Test Design

Test design is the act of creating and writing test suites for testing a software. The test design revolves around tests themselves, the test conditions and ways that testing will be approached.

Test design involves creating and writing test suites for testing a software, but will require specificity and detailed input.The fundamental challenge of test design is that there are infinitely many different tests that you could run, but there is not enough time to run them all. A subset of tests must be selected; small enough to run, but well-chosen enough that the tests find bug and expose other quality-related information.

Test design is a process that describes "how" testing should be done. It includes processes for the identifying test cases by enumerating steps of the defined test conditions. The testing techniques defined in test strategy or plan is used for enumerating the steps. The test cases may be linked to the test conditions and project objectives directly or indirectly depending upon the methods used for test monitoring, control and traceability.The objectives consist of test objectives, strategic objectives and stakeholder definition of success.

When to create test design?

After the test conditions are defined and sufficient information is available to create the test cases of high or low level, test design for a specified level can be created. For lower level testing, test analysis and design are combined activity. For higher level testing, test analysis is performed first, followed by test design.

There are some activities that routinely take place when the test is implemented. These activities may also be incorporated into the design process when the tests are created in an iterative manner. An example of such a case is creation of test data. Test data will definitely be created during the test implementation. So it is better to incorporate it in the test design itself.

This approach enables optimization of test condition scope by creating low or high level test cases automatically.

Good test design supports

Defining and improving quality related processes and procedures (quality assurance). Evaluating the quality of the product with regards to customer expectations and needs (quality control). Finding defects in the product (software testing).

Prerequisites of test design

- Appropriate specification (test bases)
- Risk and complexity analysis.
- Historical data of your previous developments (if exists).

8.5 Test Plan

A test plan is a detailed document which describes software testing areas and activities. It outlines the test strategy, objectives, test schedule, required resources, test estimation and test deliverables.

A test plan can also include a test strategy, which outlines the testing approach, and gives generic details for teams to follow. Test plan gives specific responsibilities to team members, the test strategy ensures that anyone is able to execute the tests, aligning with agile practices.

Types of Test Plan

- Master Test Plan
- Phase Test Plan
- Testing Type Specific Test Plans

Master Test Plan

Master Test Plan is a type of test plan that has multiple levels of testing. It includes a complete test strategy.

Phase Test Plan

A phase test plan is a type of test plan that addresses any one phase of the testing strategy. For example, a list of tools, a list of test cases, etc.

Specific Test Plans

Specific test plan designed for major types of testing like security testing, load testing, performance testing, etc. In other words, a specific test plan designed for non-functional testing.

8.6 Test Plan as per IEEE 829 Standard

- Analyze the product
- Design the Test Strategy
- Define the Test Objectives
- Define Test Criteria
- Resource Planning
- Plan Test Environment
- Schedule & Estimation
- Determine Test Deliverables

Analyze the product

Who will use the website?

What is it used for?

How will it work?

What are software/ hardware the product uses?

Design the Test Strategy

The project's testing objectives and the means to achieve them

- Determines testing effort and costs
- Define Scope of Testing
- Identify Testing Type
- Document Risk & Issues
- Create Test Logistics

Define Test Objective

Test Objective is the overall goal and achievement of the test execution.

List all the software features (functionality, performance, GUI…) which may need to test.

Define the target or the goal of the test based on above features.

Define Test Criteria

Test Criteria is a standard or rule on which a test procedure or test judgment can be based. There're 2 types of test criteria as following

Suspension Criteria

Exit Criteria

Resource Planning

Resource plan is a detailed summary of all types of resources required to complete project task.

Resource could be human, equipment and materials needed to complete a project

Plan Test Environment

A testing environment is a setup of software and hardware on which the testing team is going to execute test cases.

The test environment consists of real business and user environment, as well as physical environments, such as server, front end running environment.

Schedule and Estimation

In the Test Estimation phase, break out the whole project into small tasks and add the estimation for each task.

Test Deliverables

Test Deliverables is a list of all the documents, tools and other components that has to be developed and maintained in support of the testing effort.

Test Plan Guidelines

- Collapse your test plan.
- Avoid overlapping and redundancy.
- If you think that you do not need a section that is already mentioned above, then delete that section and proceed ahead.
- Be specific. For example, when you specify a software system as the part of the test environment, then mention the software version instead of only name.
- Avoid lengthy paragraphs.
- Use lists and tables wherever possible.
- Update plan when needed.
- Do not use an outdated and unused document.

Importance of Test Plan

- The test plan gives direction to our thinking. This is like a rule book, which must be followed.
- The test plan helps in determining the necessary efforts to validate the quality of the software application under the test.
- The test plan helps those people to understand the test details that are related to the outside like developers, business managers, customers, etc.
- Important aspects like test schedule, test strategy, test scope etc are documented in the test plan so that the management team can review them and reuse them for other similar projects.

8.7 Test Cases

The test case a group of conditions under which a tester determines whether a software application is working as per the customer's requirements or not. A Test Case contains test steps, test data, precondition, post condition developed for specific test scenario to verify any requirement.

A test case is a set of instructions on "HOW" to validate a particular test objective/target, which when followed will tell us if the expected behavior of the system is satisfied or not.

A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly. The process of developing test cases can also help find problems in the requirements or design of an application. Having test cases that are good at revealing the presence of faults is central to successful testing. Ideally, we would like to determine a set of test cases such that successful execution of all of them implies that there are no errors in the program. This ideal goal cannot usually be achieved due to practical and theoretical constraints.

As each test case costs money, effort is needed to generate the test case, machine time is needed to execute the program for that test case, and more effort is needed to evaluate the results. Therefore, we would also like to minimize the number of test cases needed to detect errors.

These are the two fundamental goals of a practical testing activity – maximize the number of errors detected and minimize the number of test cases. With selecting test cases the primary objectives is to ensure that if there is an error or fault in the program, it is exercised by one of the test cases.

An ideal test case set is one that succeeds (meaning that its execution reveals no errors) only if there are no errors in the program. For this test selection criterion can be used. There are two aspects of test case selection – specifying a criterion for evaluating a set of test cases, and generating a set of test cases that satisfy a given criterion.

There are two fundamental properties for a testing criterion: reliability and validity. A criterion is reliable if all the sets that satisfy the criterion detect the same errors. A criterion is valid if for any error in the program there is some set satisfying the criterion that will reveal the error.

Some axioms capturing some of the desirable properties of test criteria have been proposed. The first axiom is the applicability axiom, which states that for every program there exists a test set T that satisfies the criterion.

This is clearly desirable for a general-purpose criterion: a criterion that can be satisfied only for some types of programs is of limited use in testing. The anti-extensionality axiom states that there are programs P and Q, both of which implement the same specifications, such that a test set T satisfies the criterion for P but does not satisfy the criterion for Q.

The anti-decomposition axiom states that there exists a program P and its component Q such that a test case set T satisfies the criterion for P and T1 is the set of values that variables can assume on entering Q for some test case in T and T1 does not satisfy the criterion for Q. Essentially, the axiom says that just because the criterion is satisfied for the entire program, it does not mean that the criterion has been satisfied for its components.

The anti-composition axiom states that there exist program P and Q such that T satisfies the criterion for P and the outputs of P for T satisfy the criterion for Q, but T does not satisfy the criterion for the parts P and Q does not imply that the criterion has been satisfied by the program comprising P, Q. It is very difficult to get a criterion that satisfies even these axioms. This is largely due to the fact that a program may have paths that are infeasible, and one cannot determine these infeasible paths algorithmically as the problem is undesirable.

Why test cases?

- To require consistency in the test case execution
- To make sure a better test coverage
- It depends on the process rather than on a person
- To avoid training for every new test engineer on the product

To make sure a better test coverage

We should cover all possible scenarios and document it, so that we need not remember all the scenarios again and again.

It depends on the process rather than on a person

A test engineer has tested an application during the first release, second release, and left the company at the time of third release.

If the person is not there for the third release, it becomes difficult for the new person. Hence all the derived values are documented so that it can be used in the future.

To avoid giving training for every new test engineer on the product

When the test engineer leaves, he/she leaves with a lot of knowledge and scenarios.

Those scenarios should be documented so that the new test engineer can test with the given scenarios and also can write the new scenarios.

Best Practice for writing Test Case

- Test Cases need to be simple and transparent
- Create Test Case with End User in Mind
- Avoid test case repetition
- Do not assume
- Ensure 100% Coverage
- Test Cases must be identifiable
- Implement Testing Techniques
- Self-cleaning
- Repeatable and self-standing
- Peer Review.

Test Cases need to be simple and transparent

Create test cases that are as simple as possible. They must be clear and concise as the author of the test case may not execute them.

Use assertive language like go to the home page, enter data, and click on this and so on.

Create Test Case with End User in Mind

The ultimate goal of any software project is to create test cases that meet customer requirements and is easy to use and operate.

A tester must create test cases keeping in mind the end user perspective.

Avoid test case repetition

Do not repeat test cases. If a test case is needed for executing some other test case, call the test case by its test case id in the pre-condition column.

Do not assume

Do not assume functionality and features of your software application while preparing test case. Stick to the Specification Documents.

Ensure 100% Coverage

Make sure you write test cases to check all software requirements mentioned in the specification document.

Use Traceability Matrix to ensure no functions/ conditions is left untested.

Test Cases must be identifiable

Name the test case id such that they are identified easily while tracking defects or identifying a software requirement at a later stage.

Implement Testing Techniques

Boundary Value Analysis (BVA): As the name suggests it's the technique that defines the testing of boundaries for a specified range of values.

Equivalence Partition (EP): This technique partitions the range into equal parts/groups that tend to have the same behavior.

State Transition Technique: This method is used when software behavior changes from one state to another following particular action.

Error Guessing Technique: This is guessing/anticipating the error that may arise while doing manual testing. This is not a formal method and takes advantages of a tester's experience with the application.

Self-cleaning: The test case you create must return the Test Environment to the pre-test state and should not render the test environment unusable. This is especially true for configuration testing.

Repeatable and self-standing and Peer Review: The test case should generate the same results every time no matter who tests it. After creating test cases, get them reviewed by your colleagues. Your peers can uncover defects in your test case design, which you may easily miss.

Types of test cases

- Functionality Test Cases
- User Interface Test Cases
- Performance Test Cases
- Integration Test Cases
- Usability Test Cases
- Database Test Cases
- Security Test Cases
- User Acceptance Test Cases

Functionality test cases are used to discover if an application's interface works with the rest of the system and its users. The tests identify the success or failure of functions that the software is expected to perform.

User interface test cases are used to verify that specific pieces of the Graphical User Interface (GUI) look and work as expected. These types of test cases can be used to identify cosmetic inconsistencies, grammar and spelling errors, links, and any other elements the user interacts with or sees.

Performance test cases validate response times and overall effectiveness of an application.After executing an action, how long does it take for the system to respond? Performance test cases should have a very clear set of success criteria.

Integration test cases are meant to determine how different modules interact with each other. The main purpose with integration test cases are to ensure interfaces between the different modules are working properly.

Usability test cases can often be referred to as "tasks" or "scenarios". Rather than providing detailed step-by-step instructions to execute the test, the tester is presented with a high level scenario or task to complete.

Database Test Cases: Test cases for database testing examine what's happening behind the scenes.Database tests are used to verify the developer has written the code in a way that stores and handles data in a consistent, safe manner.

Security test cases help ensure the application restricts actions and permissions wherever necessary. These test cases are written to protect data when and where it needs to be safeguarded.

User acceptance test cases, or "UAT" test cases, help the team test the user acceptance testing environment.These test cases should be broad, covering all areas of the application.

Test case Advantages

- Test cases ensure good test coverage
- Help improve the quality of software,
- Decreases the maintenance and software support costs
- Help verify that the software meets the end user requirements
- Allows the tester to think thoroughly and approach the tests from as many angles as possible

- Test cases are reusable for the future – anyone can reference them and execute the test.

8.8 Test Case Template

A test case template is a document containing an organized list of test cases for different test scenarios that check whether or not the software has the intended functionality.

It is well-designed document for developing and better understanding of the test case data for a particular test case scenario.

Test case template fields

Test case ID	Test Priority
Name of the Module	Test Designed by
Date of test designed	Test Executed by
Date of the Test Execution	Name or Test Title
Description of Test	Pre-condition
Dependencies	Test Steps
Test Data	Expected Results
Post-Condition	Actual Result
Status (Fail/Pass)	Notes

Test case template example

Test Case ID	Test Case Name	Summary	Expected Results	Remarks
<unique id to identify the test case>	<short name to identify the test case>	<brief summary of this test case>	<brief explanation of what the expected results of this test case should be>	<any additional remarks>

Test case template example

Test Case ID	TC_Functionality_01		
Description			
Module			
Prepared By	Qatutorial.com	Date Prepared	
Reviewed / Updated		Date Reviewed	
Tested By	Qatutorial.com	Date Tested	

Test Activities			
Sl. No.	Step Description	Expected Results	Actual Results

Test Data Sets			
Data Type	Data Set 1	Data Set 2	Data Set 3
Test Case Result			

Summary

A high quality software product satisfies user needs, conforms to its requirements, and design specifications and exhibits an absence of errors. Techniques for improving software quality include systematic quality assurance procedures, walkthroughs, inspections, static analysis, unit testing integration testing, acceptance testing etc. Testing plays a critical role in quality assurance for software. Testing is a dynamic method for verification and validation. In it the system is executed and the behavior of the system is observed. Due, to this testing observes the failure of the system, from which the presence of faults can be deduced. The goal of the testing is to detect the errors so there are different levels of testing. Unit testing focuses on the errors of a module while integration testing tests the system design.

A test plan outlines the strategy that will be used to test an application, the resources that will be used, the test environment in which testing will be performed, and the limitations of the testing and the schedule of the testing activities.

Keywords

Debugging: Debugging is the activity of locating and correcting errors. It starts once a failure is detected.

ET: Error Tolerance

Functional Testing: Functional testing is also known as black box testing.

Software Testing: Software Testing is the process of executing a program or system with the intent of finding errors.

Test plan

Test cases

Self Assessment

1. Good test design supports____
A. Defining and improving quality related processes and procedures
B. Evaluating the quality of the product with regards to customer expectations and needs
C. Finding defects in the product
D. All of above

2. What are the Prerequisites of test design?
A. Appropriate specification
B. Risk and complexity analysis.
C. Historical data of your previous developments
D. All of above

3. Manual Testing includes.
A. Unit testing
B. Smoke testing
C. Black box testing
D. None of above

4. What are different Types of Software Testing?
A. Functional Testing
B. Non-Functional Testing
C. Maintenance
D. All of above

5. Which is not a part of Functional testing?
A. Localization
B. Globalization
C. Black box testing
D. None of above

6. What are the Types of Test Plan?
A. Master Test Plan
B. Phase Test Plan
C. Testing Type Specific Test Plans
D. All of above

7. As per IEEE 829 standard which is not part of test plan?
A. Design the Test Strategy
B. Feasibility study
C. Define the Test Objectives
D. Define Test Criteria

8. What are the best practice for writing Test Case?
A. Create Test Case with End User in Mind
B. Avoid test case repetition
C. Do not assume
D. All of above

9. Equivalence Partition is part of____
A. Design
B. Testing Technique
C. Maintenance
D. All of above

10. What are the types of test cases?
A. User Interface Test Cases
B. Performance Test Cases
C. Integration Test Cases
D. All of above

11. Which is not part of test case?
A. Usability
B. Database
C. Design
D. Security

12. Which is test case template fields?
A. Test Priority
B. Name of the Module
C. Test Designed by
D. All of above

13. Status (Fail/Pass) is part of __
A. Design
B. Test case template fields
C. Maintenance
D. None of above

14. Test case is_____
A. Design activity
B. Feasibility study
C. Contains test steps, test data, pre-condition, post condition
D. None of above

15. Why test case required?

A. To require consistency in the test case execution

B. To make sure a better test coverage

C. To avoid training for every new test engineer on the product

D. All of above

Answer for Self Assessment

1.	D	2.	D	3.	C	4.	D	5.	C
6.	D	7.	B	8.	D	9.	B	10.	D
11.	C	12.	D	13.	B	14.	C	15.	D

Review Questions

1. Software Testing is the process of executing a program or system with the intent of finding errors. Analyze.
2. What do you know about various special system tests? Explain briefly.
3. Discuss the various debugging approaches.
4. Discuss test plan and its significance.
5. Why we need test plans?
6. What are the Best Practice for writing Test Case?
7. Discuss Error Guessing Technique.
8. What are the different Types of test cases?

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, Second Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering a Practioner's Approach, 5th edition, Tata McGraw
- Hill Higher education.
- Sommerville, Software Engineering, Sixth edition, Pearson Education.

Web Links

- http://msdn.microsoft.com/en-us/library/ms978235.aspx
- http://bazman.tripod.com/what_testing
- https://www.tutorialspoint.com/
- https://www.softwaretestinghelp.com/

Unit 09: Testing Strategies

CONTENTS

Objectives

After studying this unit, you will be able to:

- describe black-box testing
- white-box testing
- sanity testing and smoke testing

Introduction

Testing is the process of running software with real or simulated inputs to show that it meets a product's criteria and, if it doesn't, to pinpoint the exact disparities between expected and actual outcomes. Software testing is the process of comparing a piece of software to user and system standards.

Any action aimed at analysing an attribute or capability of a programme or system and deciding if it reaches its necessary results is known as software testing. Software testing, despite being critical to software quality and widely used by programmers and testers, is still considered an art due to a lack of understanding of software concepts. The difficulty in software testing derives from the software's complexity: we can't test a programme of moderate complexity entirely. Debugging is only one aspect of testing. Testing can be done for quality assurance, verification and validation, or to estimate reliability. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

White box and black box testing are terms used to describe the point of view a test engineer takes when designing test cases. Black box is an external view of the test object and white box, an internal view

9.1 Black Box Testing

Black-box testing, also known as behavioral testing, focuses on the software's functional requirements. In other words, black-box testing allows a software engineer to create sets of input conditions that fully exercise all of a program's functional requirements. Black-box testing is not a replacement for white-box testing. Rather, it's a supplement to white-box approaches, and it's more likely to find a different set of problems.

Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.

This test design method can be used at all stages of development, including unit, integration, system, and acceptance. The higher the level, and hence the larger and more complex the box, the more we must rely on black box testing to simplify. While this method can reveal parts of the specification that haven't been implemented, you can't be sure that all existing paths have been tested. It enables a software developer to create a set of inputs that can fully encompass all of a program's functional needs.

Black-box testing attempts to find errors in the following categories: Notes

1. Incorrect or missing functions

2. Interface errors

3. Errors in data structures or external data base access

4. Behavior or performance errors

5. Initialization and termination errors.

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing.

2. Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Types of Black Box Testing

- Functional testing
- Non-functional testing
- Regression testing

Functional testing: This is a type of black box testing which is related to the functional requirements of a system.Functional testing is concerned only with the functional requirements of a system and covers how well the system executes its functions.

Non-functional testing: This black box testing type is not related to testing of specific functionality, non-functional testing is concerned with the non-functional requirements and is designed specifically to evaluate the readiness of a system according to the various criteria which are not covered by functional testing.

Regression testing: Regression Testing is performed after code fixes, upgrades or any other system maintenance to check the new changes has not affected any existing functionality.

9.2 Black Box Testing Techniques

- Equivalence Class Partitioning
- Boundary Value Analysis
- Cause effect Graphing
- Compatibility testing
- Syntax Driven Testing
- State Transition Testing

Equivalence Class Partitioning

The domain of input values to a programme is partitioned into a set of equivalence classes in this approach. This partitioning is done in such a way that the program's behaviour is consistent across all input data of the same equivalence class.

The idea is to partition the input domain of the system into a number of equivalence classes such that each member of class works in a similar way, i.e., if a test case in one class results in some error, other members of class would also result into same error.

The domain of input values to a programme is partitioned into a set of equivalence classes in this approach. This partitioning is done so that the program's behaviour is consistent across all input data of the same equivalence class. The primary notion behind the equivalence classes is that testing code with any one value from an equivalence class is equivalent to testing software with any other value from that equivalence class. Examining the input and output data can be used to create equivalence classes for software. Some general guidelines for creating equivalence classes are listed below.

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

The technique involves two steps

Identification of equivalence class – Partition any input domain into minimum two sets: valid values and invalid values. For example, if the valid range is 0 to 50 then select one valid input like 45 and one invalid like 69.

Generating test cases - To each valid and invalid class of input assign unique identification number.Write test case covering all valid and invalid test case considering that no two invalid inputs mask each other.

 **Example:**For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

 **Example:**Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

The equivalence classes are the following:

- Parallel lines (m1=m2, c1≠c2)
- Intersecting lines (m1≠m2)
- Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5,

5) (7, 7), (10, 10) (10, 10) are obtained.

Boundary Value Analysis

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

The values at the boundaries are the focus of boundary value testing. This method assesses whether or not a set of values is acceptable to the system. It's a great way to cut down on the amount of test cases. It's best for systems with inputs that fall within defined parameters.

Boundary value analysis is a software testing technique for determining test cases that address known areas of common problems at the edges of software component input ranges. The borders of input ranges to a software component, in particular, are prone to faults, according to testing experience. A programmer who has to implement the range 1 to 12 at an input, which may be used to represent the months January to December in a date, has a line in his code that checks for this range. This may appear to be:

if (month > 0 && month < 13)

But a common programming error may check a wrong range e.g. starting the range at 0 by writing:

if (month >= 0 && month < 13)

For more complex range checks in a program this may be a problem which is not so easily spotted as in the above simple example.

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

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers.
3. Apply guidelines 1 and 2 to output conditions.

   Example: Assume that a temperature vs. pressure table is required as output from anengineering analysis program.

   Test cases should be designed to create an output report that produces the maximum (andminimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., an array has a definedlimit of 100 entries), be certain to design a test case to exercise the data structure at itsboundary.
   Most software engineers intuitively perform BVA to some degree. By applying these guidelines,boundary testing will be more complete, thereby having a higher likelihood for error detection.

Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1,5000,5001}.

Cause effect Graphing

One weakness with the equivalence class partitioning and boundary value methods is that they consider each input separately. That is, both concentrate on the conditions and classes of one input. They do not consider combinations of input circumstances that may form interesting situations that should be tested.

Consider all valid combinations of the equivalence classes of input conditions as one approach to exercise combinations of distinct input circumstances. This straightforward approach will generate a disproportionately large number of test cases, many of which will be useless in exposing new errors. For example, if there are n different input conditions, we will have 2n test cases because any combination of the input conditions is legitimate. Cause-effect graphing is a technique for selecting input conditions in a systematic manner so that the number of test cases does not grow unmanageably huge. The technique starts with identifying causes and effects of the system under testing. A cause is a distinct input condition, and an effect is a distinct output condition. Each condition forms a node in the causeeffect graph.

The conditions should be written in such a way that they can be true or untrue. For example, an input condition could be "file is empty," which would be true if the input file was empty and false if it wasn't. Following the identification of causes and effects, we determine the causes that can produce each effect and how the conditions must be coupled to make the effect true for each effect. The Boolean operators "and," "or," and "not," which are represented in the graph by a, V, and zigzag line, respectively, are used to combine conditions.

Then, for each effect, all combinations of the causes that the effect depends on which will make the effect true, are generated (the causes that the effect does not depend on are essentially "don't care").

By doing this, we identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effect true.

Let us illustrate this technique with a small example. Suppose that for a bank database there are two commands allowed:

credit acct-number transaction amount

debit acct-number transaction amount

The account gets credited if the command is credit and the acct-number is correct. The account gets debited if the command is debit, the account number is valid, and the transaction amount is valid (less than the balance). A suitable message is generated if the command is invalid, the account number is invalid, or the debit amount is invalid. From these conditions, we may deduce the following causes and consequences:

Cause:

c1. Command is credit

c2. Command is debit

c3. Account number is valid

c4. Transactionamt. is valid

Effects:

el. Print "invalid command"

e2. Print "invalid account-number"

e3. Print "Debit amount not valid"

e4. Debit account

e5. Credit account

This technique establishes relationship between logical inputs called causes with corresponding actions called effect. The causes and effects are represented using Boolean graphs.

Identify inputs (causes) and outputs (effect).

Develop cause effect graph.

Transform the graph into decision table.

Convert decision table rules to test cases.

Compatibility testing

The test case result not only depend on product but also infrastructure for delivering functionality. When the infrastructure parameters are changed it is still expected to work properly

Syntax Driven Testing

This type of testing is applied to systems that can be syntactically represented by some language. For example- compilers, language that can be represented by context free grammar. In this, the test cases are generated so that each grammar rule is used at least once.

State Transition Testing

This testing technique uses the inputs, outputs, and the state of the system during the testing phase. It checks the software against the sequence of transitions or events among the test data.

Steps of black box testing

- The first step to black-box testing is to understand the requirement specifications of the application under test.
- The next step is to evaluate the set of valid inputs and test scenarios to test the software
- Prepare the test cases to cover a maximum range of inputs.
- The test cases are run in the system to generate output, which is validated with the expected outcome to mark pass or fail.

- The failed steps are marked and sent to the development team to fix them.
- Retest the system using various testing techniques to verify its recurring nature or to pass it.

Black Box Testing and SDLC

Requirement - This is the initial stage of SDLC and in this stage, a requirement is gathered. Software testers also take part in this stage.

Test Planning and Analysis - Testing Types applicable to the project are determined. A Test Plan is created which determines possible project risks and their mitigation.

Design - In this stage Test cases/scripts are created on the basis of software requirement documents

Test Execution- In this stage Test Cases prepared are executed. Bugs if any are fixed and re-tested.

Advantages of Black Box testing

- The tester doesn't need any technical knowledge to test the system. It is essential to understand the user's perspective.
- Testing is performed after development, and both the activities are independent of each other.
- It works for a more extensive coverage which is usually missed out by testers as they fail to see the bigger picture of the software.

Disadvantages of Black-box Testing

- The test inputs needs to be from large sample space.
- It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult.
- Chances of having unidentified paths during this testing. Test cases can be generated before development and right after specification.

9.3 White Box Testing

White box testing (also known as clear box testing, glass box testing, or structural testing) designs test cases based on the internal structure of the system. To identify all paths across the software, you'll need programming expertise. The tester selects test case inputs and finds the relevant outputs for all paths. Every node in a circuit can be probed and measured in electrical hardware testing, for example, in circuit testing (ICT).

Since the tests are based on the actual implementation, if the implementation changes, the tests probably will need to also. For example ICT needs updates if component values change, and needs modified/new fixture if the circuit changes.

This adds financial resistance to the change process, thus buggy products may stay buggy. Automated optical inspection (AOI) offers similar component level correctness checking without the cost of ICT fixtures, however changes still require test updates.

While white box testing can be used at the unit, integration, and system levels, it is most commonly used at the unit level. While it is typically used to test paths within a unit, it can also be used to test paths between units during integration and subsystems during a system level test. Though this type of test design can reveal a large number of test cases, it may miss sections of the specification that haven't been implemented or criteria that haven't been met. You can be certain, however, that all paths through the test object are followed.

White-box testing or glass-box testing is a test case design method that uses the control structure of the procedural design to obtain test cases. Such kind of testing, software engineer can be deriving test cases which:

1. Guarantee that all independent paths within a module have been exercised at least once,

2. Exercise all logical decisions on their true and false sides

3. Execute all loops at their boundaries and within their operational bounds

4. Exercise internal data structures to ensure their validity.

Steps for white box testing

Input: Requirements, Functional specifications, design documents, source code.

Processing: Performing risk analysis for guiding through the entire process.

Test planning: Designing test cases so as to cover entire code. Execute test plan until error-free software is reached

Output: Preparing final report of the entire testing process.

9.4 White Box Testing Techniques

- Statement Coverage
- Branch Coverage
- Path Coverage
- Condition Coverage
- Loop Testing

Statement Coverage

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid's GCD computation algorithm:

intcompute_gcd (x, y)

int x, y;{1 while (x! = y)

{2 if (x>y) then, 3 x= x – y;4 else y= y – x;5}6 return x;}

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed at least once.

Branch Coverage

In this technique, test cases are designed so that each branch from all decision points are traversed at least once. In a flowchart, all edges must be traversed at least once.

Path Testing

In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.

This technique corresponds to testing all possible paths which means that each statement and branch is covered.

Path testing steps

- Make the corresponding control flow graph
- Calculate the cyclomatic complexity
- Find the independent paths
- Design test cases corresponding to each independent path

Condition Coverage

In this technique, all individual conditions must be covered

Example:READ A, B

IF(A == 0 | | B == 0)

PRINT '0'

In this example, there are 2 conditions: A == 0 and B == 0. Now, test these conditions get TRUE and FALSE as their values.

Loop Testing

Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops. Loops are very important constructs for generally all the algorithms. Loop testing is a white box testing technique. It focuses exclusively on the validity of loop constructs.

Simple loops

Nested loops

Concatenated loops

Simple loop: The following set of tests should be applied to simple loop where n is the maximum number of allowable passes thru the loop:

- Skip the loop entirely.

- Only one pass thru the loop.

- Two passes thru the loop.

- M passes thru the loop where m < n.

- N-1, n, n+1 passes thru the loop.

Nested loop: Beizer approach to the nested loop is:

- Start at the innermost loop. Set all other loops to minimum value.

- Conduct the simple loop test for the innermost loop while holding the outer loops at their minimum iteration parameter value.

-Work outward, conducting tests for next loop, but keeping all other outer loops at minimum values and other nested loops to typical values.

- Continue until all loops have been tested.

Concatenated loops: These can be tested using the approach of simple loops if each loop is independent of other. However, if the loop counter of loop 1 is used as the initial value for loop 2 then approach of nested loop is to be used.

Unstructured loop: This class of loops should be redesigned to reflect the use of the structured programming constructs.

Types of White Box Testing

Unit testing – tests written as part of the application code, which test that each component is working as expected.

Mutation testing – a type of unit testing that checks the robustness and consistency of the code by defining tests, making small, random changes to the code and seeing if the tests still pass.

Integration testing – tests specifically designed to check integration points between internal components in a software system, or integrations with external systems.

Static code analysis – automatically identifying vulnerabilities or coding errors in static code, using predefined patterns or machine learning analysis.

White Box Penetration Testing: In this testing, the tester/developer has full information of the application's source code, detailed network information, IP addresses involved and all server information the application runs on. The aim is to attack the code from several angles to expose security threats.

White Box Testing Tools

- NUnit
- PyUnit
- Parasoft Jtest
- EclEmma
- HTMLUnit
- CppUnit

Advantages of White box testing

- White box testing optimizes code so hidden errors can be identified.
- Test cases of white box testing can be easily automated.
- This testing is more thorough than other testing approaches as it covers all code paths.
- It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

- White box testing is too much time consuming when it comes to large-scale programming applications.
- White box testing is much expensive and complex.
- It can lead to production error because it is not detailed by the developers.

9.5 Sanity Testing

Sanity testing is a type of test that is performed to touch on each implementation and its impact but not in-depth or comprehensively; it may encompass functional, UI, version, and other aspects. After obtaining a software build with minor modifications in code or functionality, sanity testing is performed to ensure that the bugs have been fixed and that no new issues have been introduced as a result of the changes.

The objective is "not" to verify thoroughly the new functionality but to determine that the developer has applied some rationality (sanity) while producing the software.If sanity test fails, the build is rejected to save the time and costs involved in a more rigorous testing.

Sanity testing was performed when we are receiving software build (with minor code changes) from the development team. It is a checkpoint to assess if testing for the build can proceed or not.Sanity testing also ensures that the modification in the code or functions does not affect the associated modules. Consequently, it can be applied only on connected modules that can be impacted.

Features of Sanity Testing

Subset of Regression Testing: Sanity testing is a subset of regression testing and focuses on the smaller section of the application.

Unscripted: Most of the times sanity testing is not scripted.

Not documented: Usually sanity testing is undocumented.

Narrow and deep: Sanity testing is narrow and deep approach of testing where limited functionalities are covered deeply.

Performed by testers: Sanity testing is normally performed by testers

Sanity Testing Process

The main purpose of performing sanity testing is to check the incorrect outcomes or defects which are not existing in component procedures. And also, ensure that the newly added features may not affect the functionalities of current features.

Therefore, we need to follow the below steps to implement the sanity testing process gradually:

- Identification
- Evaluation
- Testing

Identification

The first step in the sanity testing process is Identification, where we detect the newly added components and features as well as the modification presented in the code while fixing the bug.

Evaluation

After completing the identification step, we will analyze newly implemented components, attributes and modify them to check their intended and appropriate working as per the given requirements.

Testing

Once the identification and evaluation step are successfully processed, we will move to the next step, which is testing. In this step, we inspect and assess all the linked parameters, components, and essentials of the above analyzed attributed and modified them to make sure that they are working fine.

Advantages of Sanity Testing

- Sanity testing helps in quickly identify defects in the core functionality
- It is less expensive as compared to other types of software testing
- It can be carried out in lesser time as no documentation is required for sanity testing.
- If the defects are found during sanity testing, project is rejected that is helpful in saving time for execution of regression tests.

Disadvantages of Sanity testing

- All the test cases are not covered under sanity testing.
- It is emphasized only on the statement and functions of the application.
- We do not have future references since the sanity testing is unscripted.

 **Example:**Suppose we have an e-commerce application, which contains several modules, but here, we mainly concentrate only a few modules such as the login page, the home page, the new user creation page, the user profile page, etc.

While a new user tries to login into the application, he/she is not able to log in, as there is a bug in the login page.

Because the password field in the login module accepts less than four alpha-numeric characters and based on the specification, the password field should not be accepted below 7-8 characters.

Thus, it is considered as bug, which is reported by the testing team to the development team to fix it.

Once the development team fixes the specified bug and reports back to the testing team, the testing team tests the same feature to verify that the modification that happens in the code is working fine or not.

And the testing team also verifies that the particular modification does not impact other related functionalities.

To modify the password on the user profile page there is a process.

As part of the sanity testing process, we must authenticate the login page and the profile page to confirm that the changes are working fine at both the places.

9.6 Smoke Testing

Smoke Testing is a type of software testing which is usually performed on initial software builds to make sure that the critical functionalities of the program are working fine.It is executed before any detailed functional or regression tests are executed.

Smoke Testing is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing. It consists of a minimal set of tests run on each build to test software functionalities. Smoke testing is also known as "Build Verification Testing" or “Confidence Testing.”

Characteristics of Smoke Testing

- Smoke testing is documented.
- Smoke testing may be stable as well as unstable.

- Smoke testing is scripted.
- Smoke testing is type of regression testing.

Steps for smoke testing

- Identify smoke test cases
- Create smoke tests
- Run smoke test
- Analyze smoke tests

Types of Smoke Testing

Manual method: In general, smoke testing is done manually. It approaches varies from one organization to other. Smoke testing is carried to ensure the navigation of critical paths is as expected and doesn't hamper the functionality. Once the build is released to QA, high priority functionality test cases are to be taken and are tested to find the critical defects in the system. If the test passes, we continue the functional testing. If the test fails, the build is rejected and sent back to the development team for correction. QA again starts smoke testing with a new build version. Smoke testing is performed on new build and will get integrated with old builds to maintain the correctness of the system. Before performing smoke testing, QA team should check for correct build versions.

Automation method: Automation Testing is used for Regression Testing. However, we can also use a set of automated test cases to run against Smoke Test. With the help of automation tests, developers can check build immediately, whenever there is a new build ready for deployment.

Instead of having repeated test manually whenever the new software build is deployed, recorded smoke test cases are executed against the build. It verifies whether the major functionalities still operates properly. If the test fails, then they can correct the build and redeploy the build immediately. By this, we can save time and ensure a quality build to the QA environment.

Using an automated tool, test engineer records all manual steps that are performed in the software build.

Advantages of Smoke Testing

It saves test effort and time.

It makes easy to detect critical errors and helps in correction of errors.

It runs quickly.

It minimizes integration risks.

Summary

- Tests must be conducted to find the highest possible number of errors, must be done systematically and in a disciplined way.
- Testing software can be considered as the only destructive (psychologically) step in the entire life cycle of software production.
- A program developed should be testable i.e. it should be possible to test the program.
- An oracle is a mechanism for determining whether the program has passed or failed a test.
- One of the key problems with oracles is that they can only address a small subset of the inputs and outputs actually associated with any test.
- A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.
- There are two fundamental properties for a testing criterion: reliability and validity.
- Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.
- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- White box-testing referred to as glass box test can be defined as a test case design method which employs control structure of procedural design in order to derive test cases.

Keywords

Basis Path Testing: It allows the design and definition of a basis set of execution paths.

Black-box Testing: It refers to tests that are conducted at the software interfaces; it mainly focuses on the functional requirements of the software.

Boundary Value Analysis: Boundary value analysis is a test case design technique that complements equivalence partitioning.

Data Flow Testing: The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

Equivalence Class: An equivalence class represents a set of valid or invalid states for input conditions.

Equivalence Partitioning: Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

Failure: Failure is the inability of a system or component to perform a required function according to its specifications.

Fault: Fault is a condition that causes a system to fail in performing its required function.

Loop Testing: Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs.

Software Testing: Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.

Statement Coverage: Statement coverage is a measure of the percentage of statements that have been executed by test cases.

White-box Testing: It works on the principle of closely monitoring the procedural details of the software.

Self Assessment

1. Black box testing is____
A. Focuses on the functional requirements of the software
B. No knowledge required for internal code structure, implementation details and internal paths
C. It is used to test the system against external factors responsible for software failures
D. All of above

2. What are the types of Black Box Testing?
A. Functional testing
B. Non-functional testing
C. Regression testing
D. All of above

3. Boundary Value Analysis is part of _
A. White box testing
B. Black box testing
C. Unit testing
D. None of above

4. Which is not part of Black Box Testing Techniques?
A. Cause effect Graphing
B. Compatibility testing
C. Load testing
D. Syntax Driven Testing

5. Generating test cases and Identification of equivalence class are part of__
A. Cause effect Graphing
B. Equivalence Class Partitioning
C. Syntax Driven Testing
D. None of above

6. Cause effect graphing is__
A. Identify inputs (causes) and outputs (effect).
B. Develop cause effect graph.
C. Transform the graph into decision table
D. All of above

7. yntax Driven Testing and State Transition Testing is part of ____
A. Integration testing
B. Unit testing
C. White box testing
D. Black box testing

8. What are different Loop Testing__?
A. Simple
B. Nested
C. Concatenated
D. All of above

9. NUnit and PyUnit is part of ____
A. Integration testing
B. White box testing
C. Unit testing
D. Black box testing

10. What are the types of Smoke testing?
A. Manual method
B. Automation method
C. Hybrid method
D. All of above

Answer for Self Assessment

1. D	2. D	3. B	4. C	5. B
6. D	7. D	8. D	9. B	10. D

Review Questions

1. What are the fundamentals of software testing?
2. Explain the benefits and objectives of testing.
3. Enumerate the various principles of testing.
4. Discuss about white box testing techniques.
5. How smoke testing is different from sanity testing.
6. What are advantages of white box testing?
7. Briefly explain the Black-box testing. Discuss about equivalence partitioning

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Link

- www.guru99.com/software-testing.html
- www.ece.cmu.edu/~koopman/des_s99/sw_testing/
- www.softwarequalitymethods.com/Papers/OracleTax.pdf
- se.fsksm.utm.my/.../IST-AutomatedFrameworkSoftTestOracle-FPage.pdf
- https://www.geeksforgeeks.org/smoke-testing-software-testing/
- https://www.javatpoint.com

Unit 10: Testing Levels

CONTENTS

Objective

- after studying this unit, you will be able to:
- explain unit testing and integration testing
- discuss system testing and acceptance testing
- regression testing

Introduction

Software testing is an investigation that is carried out to offer information to stakeholders regarding the quality of the product or service being tested. Software testing can also give a corporation with an objective, unbiased picture of the software, allowing them to grasp and comprehend the risks associated with software implementation. The process of executing a programme or application with the purpose of detecting software bugs is one example of a test technique (errors or other defects). Testing is the process of assessing a system or its component(s) with the goal of determining whether or not it meets the set requirements. Testing is executing a system in order to identify any gaps, errors or missing requirements in contrary to the actual desire or requirements.

10.1 Unit Testing

Unit testing is the basic level of testing. Unit testing is concerned with the verification of individual software components or modules. Important control pathways are identified and evaluated using the design to discover faults within the module border. It entails isolating a module from the rest of the software, creating test cases, and comparing the actual results to the expected results as defined by the specifications and design. One of its goals is to locate and correct as many software faults as possible.

A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property. The isolated part of the definition is important. In his book "Working Effectively with Legacy Code", author Michael Feathers states that such tests are not unit tests when they rely on external systems: “If it talks to the database, it talks across the network, it touches the file system, it requires system configuration, or it can't be run at the same time as any other test."

Unit testing is the process of testing each unit or component of a software application separately. Unit testing is carried out by developers throughout the development (coding) phase of an application. Unit tests are used to isolate a part of code and ensure that it is correct. A singular function, method, process, module, or object might be considered a unit. Unit testing is a type of White Box testing.

Why Unit Testing

- Unit tests help to fix bugs early in the development cycle and save costs.
- It helps the developers to understand the testing code base and enables them to make changes quickly
- Good unit tests serve as project documentation
- Unit tests help with code re-use. Migrate both your code and your tests to your new project.

Work flow of unit testing

- Creating test cases
- Reviewing test cases
- Base lining test cases
- Executing test cases

Types of Unit testing

Manual: The manual approach has a step by step instructional procedure that helps testers perform this task efficiently.

Automated: The automated method is the most preferred as it is faster and more accurate. A developer writes a section of code in the application just to test the function. Isolate the function to test it more rigorously.Isolating the code helps in revealing unnecessary dependencies between the code being tested and other units or data spaces in the product. Developer uses a Unit Test Framework to develop automated test cases and codes criteria into the test to verify the correctness of the code.

Unit Testing Techniques

Black box testing: that involves testing of user interface along with input and output

White box testing: that involves testing the functional behavior of the software application

Grey box testing: that is used to execute test suites, test methods, test cases and performing risk analysis.

Unit testing best practices

- Ensure Unit Tests are independent of each other
- Always stick to one code unit at once
- Make use of AAA for readability- (Arrange, Act, Assert)
- Fix the bugs before moving for Integration Testing
- Ensure to have proper variable naming
- Always separate test code and production code

Unit Testing Tools

Junit: Junit is a free to use testing tool used for Java programming language. It provides assertions to identify test method. This tool test data first and then inserted in the piece of code.

NUnit: NUnit is widely used unit-testing framework use for all .net languages. It is an open source tool which allows writing scripts manually. It supports data-driven tests which can run in parallel.

JMockit: JMockit is open source Unit testing tool. It is a code coverage tool with line and path metrics. It allows mocking API with recording and verification syntax. This tool offers Line coverage, Path Coverage, and Data Coverage.

EMMA: EMMA is an open-source toolkit for analyzing and reporting code written in Java language. Emma support coverage types like method, line, basic block. It is Java-based so it is without external library dependencies and can access the source code.

PHPUnit: PHPUnit is a unit testing tool for PHP programmer. It takes small portions of code which is called units and test each of them separately. The tool also allows developers to use pre-define assertion methods to assert that a system behave in a certain manner.

The various errors that must be checked for while testing are as under:

1. Error description is unintelligible.

2. Error noted does not correspond to error encountered.

3. Error condition causes system intervention prior to error handling.

4. Exception-condition processing is incorrect.

5. Error description does not provide enough information to assist in the location of the cause of the error.

Advantages of Unit testing

- Isolates a section of code and validates its correctness.
- Helps in identifying and fixing the bugs at the early stage of the SDLC process.
- Assures to reduce the cost as bugs are resolved at the earliest.
- Helps the developers to improve the design by allowing refactoring of the code.
- Assures in simplifying the debugging process.
- With the proper unit testing practice, components that are integrated after the build can assure in achieving a quality product.

Unit Testing Disadvantages

Unit testing can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs.

Unit testing by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

10.2 Integration Testing

Individual modules / components are merged and tested as a group during integration testing. This level of testing is designed to reveal flaws in the interaction of integrated units. Integration testing is concerned with ensuring that data is communicated between various units. It's also known as 'I & T' (Integration and Testing), 'String Testing,' and 'Thread Testing,' among other terms. Unit testing uses modules for testing, and integration testing combines and tests these parts.

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

The big bang technique is the polar opposite of incremental integration. The programme is built and tested in small chunks, making it easier to isolate and rectify faults; interfaces are more likely to be thoroughly verified; and a systematic testing approach can be used.

Top-down Integration

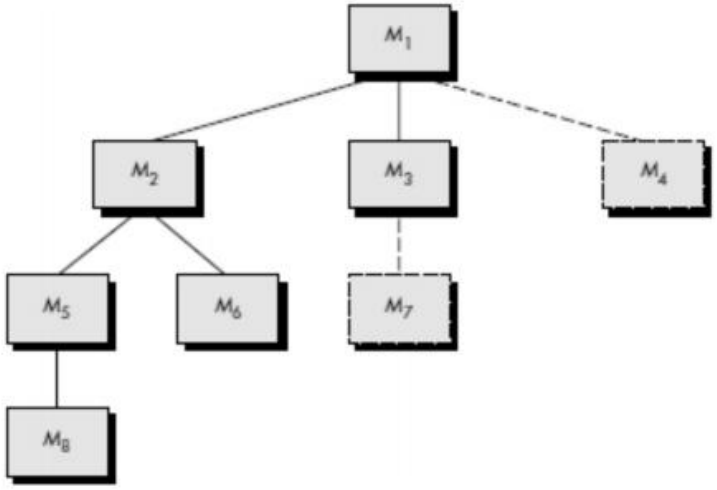
Top-down integration testing is an incremental approach to construction of program structure.Modules are integrated by moving downward through the control hierarchy, beginning withthe main control module. Modules subordinate to the main control module are incorporatedinto the structure in either a depth-first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

Referring to Figure, depth-first integration would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.

Example: Selecting the left-hand path, components M1, M2, M5 would be integrated first.

Next, M8 or if needed for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

Top down Integration



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

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

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

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

Advantages - Top-Down Integration Testing

- Separately debugged module.
- Few or no drivers needed.
- Fault Localization is easier.
- Possibility to obtain an early prototype.
- Critical Modules are tested on priority; major design flaws could be found and fixed first

Bottom-Up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules i.e., components at the lowest levels in the program structure.

Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (also referred as builds) that perform a specific software sub function.

2. A driver is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure

Advantages - Bottom-Up Integration Testing

- In bottom-up testing, no stubs are required.
- A principle advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.
- Fault localization is easier.

Big-Bang Integration Testing: In Big-Bang Integration Testing all the modules are combining and verifying the functionality after the completion of individual module testing. This approach is practicable only for very small systems. Debugging errors reported during big bang integration testing are very expensive to fix.

Mixed/ sandwiched Integration Testing: A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches.

Incremental Testing: In the Incremental Testing approach, testing is done by integrating two or more modules that are logically related to each other and then tested for proper functioning of the application. Then the other related modules are integrated incrementally and the process continues until all the logically related modules are integrated and tested successfully.

Stubs and Drivers: Stubs and Drivers are the dummy programs in Integration testing used to facilitate the software testing activity. These programs act as a substitutes for the missing models in the testing.They do not implement the entire programming logic of the software module but they simulate data communication with the calling module while testing.

Stub: Is called by the Module under Test.

Driver: Calls the Module to be tested.

Best Practices for Integration Testing

First, determine the Integration Test Strategy that could be adopted and later prepare the test cases and test data accordingly.

Study the Architecture design of the Application and identify the Critical Modules. These need to be tested on priority.

Obtain the interface designs from the Architectural team and create test cases to verify all of the interfaces in detail. Interface to database/external hardware/software application must be tested in detail.

After the test cases, it's the test data which plays the critical role.

Always have the mock data prepared, prior to executing. Do not select test data while executing the test cases.

10.3 System Testing

There are two types of system testing: integration testing and acceptance testing. The bottom-up approach, top-down strategy, and sandwich strategy are all methods for merging software components into a working product. To ensure that modules are available for incorporation into the expanding software product when needed, careful planning and scheduling are required.

The integration strategy dictates the order in which modules must be available, and thus exerts a strong influence on the order in which modules are written, debugged, and unit tested.

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests to verify that the implemented system satisfies its requirements. Acceptance tests are typically performed by the quality assurance and/or customer organizations. Depending on local circumstances, the development group may or may not be involved in acceptance testing.

System testing is done to ensure that the software is complete and integrated. Its purpose is to assess the system's end-to-end specs. System testing looks for flaws in both the individual parts and the entire system. The observed behavior of a component or a system when it is tested is the result of system testing.

System testing is performed on the entire system, either in accordance with system requirement requirements, functional requirement specifications, or both. System testing is included in the black box testing area of software testing since it encompasses testing of the software's external functionality.

System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of Black box testing, and as such, should require no knowledge of the inner design of the code. System testing is actually done to the entire system against the Functional Requirement Specifications (FRS) and/or the System Requirement Specification (SRS). Moreover, the System testing is an investigatory testing phase, where the focus is to have almost a destructive attitude and test not only the design, but also the behavior and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specifications. Remaining All Testing Models comes under System Testing.

Steps for System Testing

Test Environment Setup: Create testing environment for the better quality testing.

Create Test Case: Generate test case for the testing process.

Create Test Data: Generate the data that is to be tested.

Execute Test Case: After the generation of the test case and the test data, test cases are executed.

Defect Reporting: Defects in the system are detected.

Regression Testing: It is carried out to test the side effects of the testing process.

Log Defects: Defects are fixed in this step.

Retest: If the test is not successful then again test is performed.

Types of System Testing

Performance Testing: Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application.

Load Testing: Load Testing is a type of software testing which is carried out to determine the behavior of a system or software product under extreme load.

Stress Testing: Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads.

Usability Testing: mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives.

Scalability Testing: Scalability Testing is a type of software testing which is carried out to check the performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

Regression Testing: involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.

Recovery testing: is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

Migration testing: is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

Functional Testing: Functional testing of a system is performed to find if there's any missing function in the system. Tester makes a list of vital functions that should be in the system and can be added during functional testing and should improve quality of the system.

10.4 User Acceptance Testing

User Acceptance Testing performed by the end user or the client to verify/accept the software system before moving the software application to the production environment.The major aim of this test is to evaluate the compliance of the system with the business requirements and assess whether it is acceptable for delivery or not.

After System Testing and before making the system accessible for actual use, Acceptance Testing is the final phase of software testing. It's done in a way that's similar to Black Box testing, with the number of users required to test the system's acceptability level.

This is arguably the most importance type of testing as it is conducted by the Quality Assurance Team who will gauge whether the application meets the intended specifications and satisfies the client's requirements. The QA team will have a set of pre written scenarios and Test Cases that will be used to test the application.

More thoughts regarding the application will be presented, and more testing will be conducted to assess its accuracy and the reasons for the project's inception. Acceptance tests are designed to identify not only basic spelling mistakes, cosmetic faults, and interface gaps, but also any bugs in the application that could cause system crashes or serious issues.

By performing acceptance tests on an application the testing team will deduce how the application will perform in production. There are also legal and contractual requirements for acceptance of the system.

Steps for User Acceptance Testing

Analysis of Business Requirements

Creation of UAT test plan

Identify Test Scenarios

Create UAT Test Cases

Preparation of Test Data (Production like Data)

Run the Test cases

Record the Results

Confirm business objectives

Types of Acceptance Testing

User Acceptance Testing (UAT)

Business Acceptance Testing (BAT)

Contract Acceptance Testing (CAT)

Regulations Acceptance Testing (RAT)

Operational Acceptance Testing (OAT)

Alpha Testing

Beta Testing

Business Acceptance Testing: BAT is used to determine whether the product meets the business goals and purposes or not. BAT mainly focuses on business profits which are quite challenging due to the changing market conditions and new technologies so that the current implementation may have to being changed which result in extra budgets.

Contract Acceptance Testing: CAT is a contract which specifies that once the product goes live, within a predetermined period, the acceptance test must be performed and it should pass all the acceptance use cases.

Regulations Acceptance Testing: RAT is used to determine whether the product violates the rules and regulations that are defined by the government of the country where it is being released.

Operational Acceptance Testing: OAT is used to determine the operational readiness of the product and is a non-functional testing. It mainly includes testing of recovery, compatibility, maintainability, reliability etc.

Alpha testing: Alpha testing is used to determine the product in the development testing environment by a specialized tester's team usually called alpha testers.

Beta testing: Beta testing is used to assess the product by exposing it to the real end-users, usually called beta testers in their environment. Feedback is collected from the users and the defects are fixed. Also, this helps in enhancing the product to give a rich user experience.

User Acceptance testing tools

Fitness tool: It is a java tool used as a testing engine. It to create tests and record results in a table. Users of the tool enter the formatted input and tests are created automatically. The tests are then executed and the output is returned back to the user.

Watir: It is toolkit used to automate browser-based tests during User acceptance testing. Ruby is the programming language used for inter-process communication between ruby and Internet Explorer.

10.5 Regression Testing

Regression Testing is re-execute or re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change.

Regression Testing is done to make sure that new code changes should not have side effects on the existing functionalities. It ensures that the old code still works once the latest code changes are done.

Regression testing is re-running a subset of previously run tests to confirm that modifications have not resulted in unwanted side effects. Manual regression testing, re-executing a portion of all test cases, or employing automated capture or playback technologies are all options. Software engineers can use capture or playback tools to record test cases and results for later replay and comparison.

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

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

The number of regression tests can quickly rise as integration testing progresses. As a result, only those tests should be included in the regression test suite that address one or more classes of defects in each of the key programme functions. Once a change has occurred, it is impractical and inefficient to re-run every test for every software function.

After modifying software, either for a change in functionality or to fix defects, a regression test re-runs previously passing tests on the modified software to ensure that the modifications haven't unintentionally caused a regression of previous functionality. Regression testing can be performed at any or all of the above test levels. These regression tests are often automated.

In integration testing also, each time a module is added, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. Hence, there is the need of regression testing.

Any sort of software testing that aims to find regression bugs is known as regression testing. When software functionality that previously worked as expected stops working or no longer works in the same way that it was anticipated, regression bugs develop. Regression bugs are typically unintentional consequences of programme changes.

Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have reemerged. Experience has shown that as software is developed, this kind of reemergence of faults is quite common. Sometimes it occurs because a fix gets lost through poor revision control practices (or simple human error in revision control), but just as often a fix for a problem will be "fragile" - i.e. if some other change is made to the program, the fix no longer works. Finally, it has often been the case that when some feature is redesigned, the same mistakes will be made in the redesign that were made in the original implementation of the feature.

Therefore, in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools. Such a 'test suite' contains software tools that allows the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically rerun all regression tests at specified intervals and report any regressions.

For small projects, common tactics include running such a system after each successful build, every night, or once a week. The extreme programming software development process includes regression testing. Extensive, repeatable, and automated testing of the complete software package at every stage of the software development cycle replaces design documentation in this methodology.

Steps for regression testing

- Detect Changes in the Source Code
- Prioritize Those Changes and Product Requirements
- Determine Entry Point and Entry Criteria
- Determine Exit Point
- Schedule Tests

Types of Regression Testing

- Corrective Regression Testing
- Retest-all Regression Testing
- Selective Regression Testing
- Progressive Regression Testing
- Complete Regression Testing
- Partial Regression Testing
- Unit Regression Testing

Regression testing techniques

- Re-test All
- Regression test Selection
- Prioritization of test case

Regression Testing Tools

Selenium: This is an open source tool used for automating web applications. Selenium can be used for browser-based regression testing.

Quick Test Professional (QTP): HP Quick Test Professional is automated software designed to automate functional and regression test cases. It uses VBScript language for automation. It is a Data-driven, Keyword based tool.

Rational Functional Tester (RFT): IBM's rational functional tester is a Java tool used to automate the test cases of software applications. This is primarily used for automating regression test cases and it also integrates with Rational Test Manager.

Uses of regression testing

Regression testing can be used not only for testing the correctness of a program, but it is also often used to track the quality of its output. For instance in the design of a compiler, regression testing should track the code size, simulation time, and compilation time of the test suites.

System testing is a series of different tests and each test has a different purpose but all work to verify that all system elements have been properly integrated and perform allocated functions.

Summary

- The first level of testing is called unit testing. Unit testing focuses on verification of individual units or modules of software.
- The next level of testing is often called integration testing. Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.
- System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.
- System Testing is done after Integration Testing. This plays an important role in delivering a high-quality product.
- Integration Testing focuses on checking data communication amongst these modules. Hence it is also termed as 'I & T' (Integration and Testing), 'String Testing' and sometimes 'Thread Testing'.
- Regression Testing is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.

Keywords

Bottom-up Integration: Bottom-up integration testing begins construction and testing with atomic modules i.e., components at the lowest levels in the program structure.

Integration Testing: Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.

Regression Testing: Regression testing is the re-execution of some subset of tests that have already beenSystem Testing: System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

Top-down Integration: Top-down integration testing is an incremental approach to construction of program structure.

Unit Testing: Unit testing focuses on verification of individual units or modules of software. conducted to ensure that changes have not propagated unintended side effects.

Self Assessment

1. Unit testing is_____
A. It is a White Box testing technique
B. It involves the testing of each unit
C. It is done during the development of an application by the developers
D. All of above

2. What are the steps for work flow of unit testing?
A. Creating test cases
B. Reviewing test cases
C. Base lining test cases
D. All of above

3. Types of Unit testing are _____
A. Manual
B. Automated
C. Both manual and automated
D. None of above

4. White box testing and Grey box testing is part of __

A. Stress testing
B. Regression testing
C. Unit testing
D. None of above

5. Tools used in Unit Testing are_____
A. PHPUnit
B. EMMA
C. JMockit
D. All of above

6. Incremental Testing and Stubs and Drivers are part of ___
A. Unit testing
B. White box testing
C. Integration testing approaches
D. None of above

7. What are the advantages of Top-Down Integration Testing?
A. Few or no drivers needed.
B. Fault Localization is easier.
C. Possibility to obtain an early prototype
D. All of above

8. Which is not part of Integration testing approaches _____
A. Top-Down Integration Testing
B. Black box testing
C. Mixed Integration Testing
D. Incremental Testing

9. System testing is_____
A. Falls under the black box testing
B. It is to evaluate the end-to-end system specifications
C. Performed to validates the complete and fully integrated software product
D. All of above

10. Recovery testing and Migration testing are used in____
A. Unit testing
B. Stress testing
C. System testing
D. All of above

11. Alpha and beta testing is used in___
A. Manual
B. Automated
C. User Acceptance testing
D. None of above

12. Tools used in User Acceptance testing are__
A. Watir
B. Fitness
C. Both watir and fitness
D. None of above

13. Regression testing is _____
A. Re-execute or re-running functional and non-functional tests
B. To make sure that new code changes should not have side effects on the existing functionalities
C. Changes in Existing functionality
D. All of above

14. What are the Steps for regression testing?
A. Detect Changes in the Source Code
B. Prioritize Those Changes and Product Requirements
C. Determine Entry Point and Entry Criteria
D. All of above

15. Partial Testing and Unit Testing is part of___
A. Stress testing.
B. Load testing.
C. Regression Testing
D. All of above

Answer for Self Assessment

1.	D	2.	D	3.	C	4.	C	5.	D
6.	C	7.	D	8.	B	9.	D	10.	C
11.	C	12.	C	13.	D	14.	D	15.	C

Review Questions

1. What is unit testing? What are the advantages of unit testing?
2. Define integration testing. Explain top-down and bottom-up integration.
3. Write down the three different classes of test cases of regression test suite.
4. Discuss about system testing.
5. "Testing commences with a test plan and terminates with acceptance testing". Comment.
6. What are the advantages of regression testing?
7. Differentiate between unit testing and system testing.

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education.

Web Links

- rajeevprabhakaran.wordpress.com/2008/11/20/levels-of-testing
- https://smartbear.com/
- https://www.softwaretestinghelp.com/
- https://www.javatpoint.com/

- https://www.tutorialspoint.com/

Unit 11: Bugs

CONTENTS

Objectives

After studying this unit, you will be able to:

- discuss bug/defect definition,
- bugs life cycle and bug tracking,
- bug tracking tool

Introduction

Testing is the process of discovering flaws, with a flaw defined as any deviation from expected outcomes. "A coding error is called an Error, a defect discovered by a tester is called a Defect, a defect accepted by the development team is called a Bug, and a build that does not satisfy the requirements is called a Failure."

A software bug is an error, flaw, failure, or fault in a computer programme or system that causes it to generate an inaccurate or unexpected result, or to act in unanticipated ways. A software defect is a difference or deviation in the software application from the end user's or original business requirements. A software defect is a coding fault that results in inaccurate or unexpected outputs from a software programme that does not satisfy its intended purpose. During the execution of test cases, testers may come across such flaws.

11.1 Bug, Error, Fault and Defect

A bug is caused by a code mistake. Before the product is shipped to the customer, an error was discovered in the development environment. An error in programming that causes a programme to perform poorly, generate inaccurate results, or crash. A malfunctioning programme caused by a software or hardware fault. The term "bug" is used by testers.

A "Bug" is a most unwelcomed word in the software development process. A bug indicates a fault, error or failure in the software/system being built that produces unexpected results. A bug identified needs to be tracked and fixed to ensure optimum quality in the software/system being developed.

An error in a computer program's step, process, or data specification that leads the programme to behave in an unexpected or unexpected way. As a result of an error, a flaw is introduced into the software. It's a flaw in the software that could lead it to behave erroneously and in ways that aren't specified. It's the outcome of a mistake. The software industry is still unable to agree on definitions for all of the aforementioned terms. In other words, if you use the term to signify one thing, your audience may not understand it to mean that thing.

When the predicted and actual behaviour do not match in software testing, an incident must be raised. An occurrence could be a Bug. When a programmer intends to implement a given behaviour, but the code fails to correctly comply to that behaviour due to faulty coding implementation, it is the programmer's fault. It's also known as a flaw.

An error is a mistake, misconception, or misunderstanding on the part of a software developer. In the category of the developer, we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a de-sign notation, or a programmer might type a variable name incorrectly – leads to an Error. It is the one that is generated because of the wrong login, loop or syntax. The error normally arises in software; it leads to a change in the functionality of the program.

Bugs in history and their impact

In 1996, the $1.0 billion rocket called Ariane 5 was destroyed a few seconds after launch due to a bug in the on-board guidance computer program.

In 1962, a bug in the flight software for the Mariner I spacecraft caused the rocket to change path from the expected path.

In the 1980s, bugs in the code controlling the machine called Therac-25, used for radiation therapy, lead to patient deaths.

In the 1990s, a bug was found in the new release of AT&T's software control #4ESS long distance switches caused many computers to crash.

Bugs and its Source

A bug is actually an error which would have been introduced in the due course of the software development life cycle. The most common sources of Bugs are detailed below.

Ambiguous Requirements – Unclear requirement is a source of Bugs being introduced. This can be avoided by having strong Static Testing techniques, Reviews, Walkthroughs, etc.

Programming Errors – Incorrect coding standards is a source which can be minimized with proper code review and unit testing

Unachievable Deadlines – Stringent timelines often lead to bugs which can be avoided if proper planning is done to ensure timelines are driven based on the estimates

Missing coding - Here, missing coding means that the developer may not have developed the code only for that particular feature.

Every bug not only breaks the functionality it occurs in but also has probabilities of causing a ripple effect in other areas of the application. This ripple effect needs to be evaluated when fixing any bug. Lack of foresight in anticipating such issues can cause serious problems and an increase in bug count.

Bugs in Software Testing and Its Impact

The impact of the Bug is measured based on Severity and Priority. Impact to business is measured in terms of Severity, whereas impact to business execution is measure in terms of Priority.

The table below is a standard definition used across the software industry for the Severities.

Severity	Definition
Critical	A critical defect would create a major disruption to the business operation. A severe application problem causing considerable downtime, financial penalty or loss of integrity with customers.

High	A major defect would result in loss of business functionality and would require a workaround in production.
Medium	A defect which would have a medium impact on business functions, but could be immediately managed or worked around.
Low	A cosmetic only defect which would have no business or user impact

Priority determines the impact of business execution, and subsequently, how quickly the defect needs to be fixed to ensure the project plan is on track.

The table below is a standard definition used across the software industry for the Priorities.

Priority	**Definition**
Immediate	Any problem that is stopping ALL usage of the software by the business, i.e. Nothing else can be executed until this problem has been corrected, tested and re-migrated with No workaround**.**
High	A major function or feature has been disabled or is incorrect causing a severe degradation in service. A workaround is possible, but additional problems/failures could result in critical failure.
Medium	An issue which signifies one of the multiple streams of testing (other than the critical path) is stopped, but other streams of processing can still continue.
Low	Business can still proceed with a workaround and the fix can be applied in isolation

11.2 Classifications of Software Bugs

Software defects by nature

- Functional defects
- Performance defects
- Usability defects
- Compatibility defects
- Security defects

Functional defects are the errors identified in case the behaviour of software is not compliant with the functional requirements. Such types of defects are discovered via functional testing

Performance defects are those bound to software's speed, stability, response time, and resource consumption, and are discovered during performance testing.

Usability defects make an application inconvenient to use and, thus, hamper a user's experience with software.

Compatibility defects: An application with compatibility errors doesn't show consistent performance on particular types of hardware, operating systems, browsers, and devices or when integrated with certain software.

Security defects are the weaknesses allowing for a potential security attack. The most frequent security defects in projects we perform security testing for are encryption errors, susceptibility to SQL injections, XSS vulnerabilities, buffer overflows, weak authentication, and logical errors in role-based access.

Software defects by severity

- Critical defects
- High-severity defects
- Medium-severity defects
- Low-severity defects

Critical defects usually block an entire system's or module's functionality, and testing cannot proceed further without such a defect being fixed.

An example of a critical defect is an application's returning a server error message after a login attempt.

High-severity defects affect key functionality of an application, and the app behaves in a way that is strongly different from the one stated in the requirements, for instance, an email service provider does not allow adding more than one email address to the recipient field.

Medium-severity defects are identified in case a minor function does not behave in a way stated in the requirements.

Low-severity defects are primarily related to an application's UI and may include such an example as a slightly different size or color of a button.

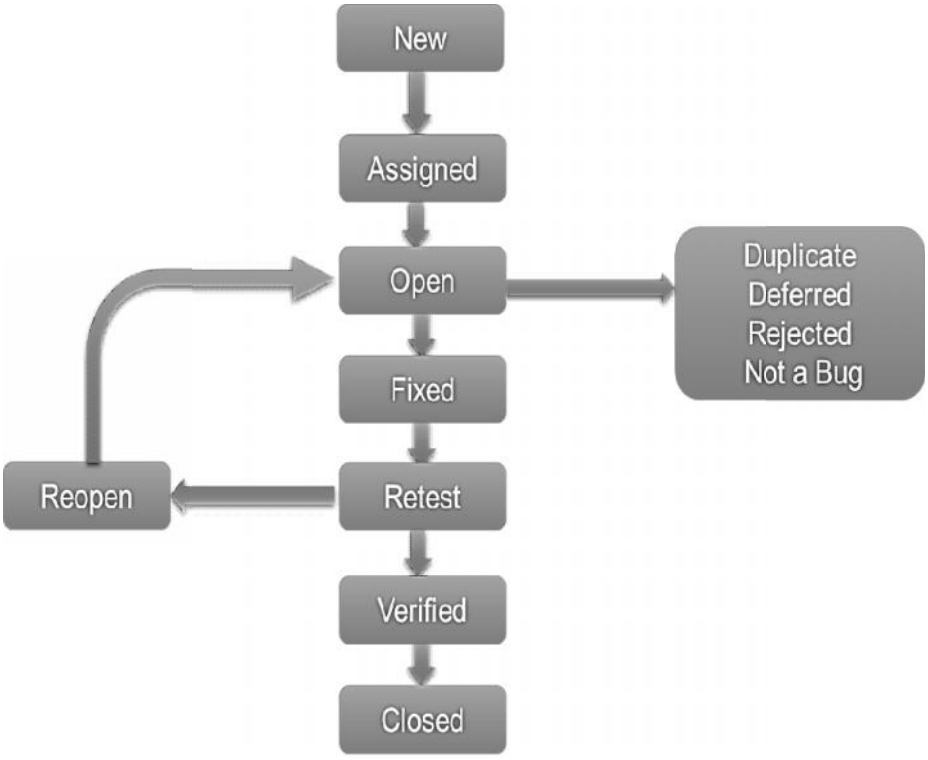
11.3 Bug Life Cycle

Bug Life Cycle is the specific set of states that defect or bug goes through in its entire life.

This starts as soon as any new defect is found by a tester and comes to an end when a tester closes that defect assuring that it won't get reproduced again.

Bug Life cycle is the journey of a defect cycle, which a defect goes through during its lifetime. It varies from organization to organization and also from project to project as it is governed by the software testing process and also depends upon the tools used.

Bug life cycle



New: This is the first state of a defect in the Defect Life Cycle. When any new defect is found, it falls in a 'New' state, and validations and testing are performed on this defect in the later stages of the Defect Life Cycle.

Assigned: In this stage, a newly created defect is assigned to the development team for working on the defect. This is assigned by the project lead or the manager of the testing team to a developer.

Open: Here, the developer starts the process of analyzing the defect and works on fixing it. if required. If the developer feels that the defect is not appropriate then it may get transferred to any of the below four states namely Duplicate, Deferred, Rejected, or Not a Bug-based upon the specific reason.

Fixed: When the developer finishes the task of fixing a defect by making the required changes then he can mark the status of the defect as 'Fixed'.

Pending Retest: After fixing the defect, the developer assigns the defect to the tester for retesting the defect at their end, and till the tester works on retesting the defect, the state of the defect remains in 'Pending Retest'.

Retest: At this point, the tester starts the task of working on the retesting of the defect to verify if the defect is fixed accurately by the developer as per the requirements or not.

Reopen: If any issue persists in the defect then it will be assigned to the developer again for testing and the status of the defect gets changed to 'Reopen'.

Verified: The tester re-tests the bug after it got fixed by the developer. If there is no bug detected in the software, then the bug is fixed and the status assigned is "verified."

Closed: When the defect does not exist any longer then the tester changes the status of the defect to 'Closed'.

Rejected: If the defect is not considered as a genuine defect by the developer then it is marked as 'Rejected' by the developer.

Duplicate: If the developer finds the defect as same as any other defect or if the concept of the defect matches any other defect then the status of the defect is changed to 'Duplicate' by the developer.

Deferred: If the developer feels that the defect is not of very important priority and it can get fixed in the next releases or so in such a case, he can change the status of the defect as 'Deferred'.

Not a Bug: If the defect does not have an impact on the functionality of the application then the status of the defect gets changed to 'Not a Bug'.

Guidelines for Implementing Defect/ bug Life Cycle

- It is very important that before starting to work on the Defect Life Cycle, the whole team clearly understands the different states of a defect (discussed above).
- Defect Life Cycle should be properly documented to avoid any confusion in the future.
- Make sure that each individual who has been assigned any task related to the Defect Life Cycle should understand his/her responsibility very clearly for better results.
- Each individual who is changing the status of a defect should be properly aware of that status and should provide enough details about the status and the reason for putting that status so that everyone who is working on that particular defect can understand the reason of such a status of a defect very easily.
- The defect tracking tool should be handled with care to maintain consistency among the defects and thus, in the workflow of the Defect Life Cycle.

11.4 Bug Tracking

During software testing, bug tracking is the process of logging and monitoring defects or errors. It's also known as issue tracking or defect tracking. Hundreds or thousands of flaws may exist in large systems. For debugging, each must be reviewed, monitored, and prioritized. Bugs may need to be monitored over a long period of time in some circumstances. An effective bug tracking system that is perfectly aligned with your development and quality processes is a priceless asset.

defect tracking is a vital step in software engineering because complicated and business-critical systems have hundreds of flaws. "Managing, evaluating, and prioritising these issues is one of the most difficult aspects. Over time, the number of defects grows, and to successfully manage them, a defect tracking system is utilised to make the task easier.

A bug tracking tool can help record, report, assign and track the bugs in a software development project. There are many defect tracking tools available. You can put this in another way "Better is the bug tracking tool, better the quality of the product."

For Bug tracking software, it is essential to have:

Reporting facility – complete with fields that will let you provide information about the bug, environment, module, severity, screenshots, etc.

Assigning – What good is a bug when all you can do is find it and keep it to yourself, right?

Progressing through life cycle stages – Workflow

History/work logs/comments

Reports – graphs or charts

Storage and Retrieval – Every entity in a testing process needs to be uniquely identifiable. The same rule applies to bugs too. A bug tracking tool must provide a way to have an ID that can be used to store, retrieve (search) and organize bug information.

Functionalities of the Bug Tracking system is:

Creating a new text file and writing the details entered by the user into the text file.

Option to change the status of the bug.

Report of specific bug file.

11.5 Bug Tracking Tools

Following is a list of Top Bug Tracking Tool, with their popular features. The list contains both open source(free) and commercial(paid) software.

BugZilla	BackLog
SpiraTeam	Bird Eats Bug
Zoho bug tracker	Monday
JIRA	Mantis
RedMine	Trac
Axosoft	HP ALM/ Quality Center
eTraxis	Bugnet

BugZilla:

Bugzilla has been a leading Bug Tracking tool widely used by many organizations for quite some time now. It is very simple to use, a web-based interface. It has all the features of the essence, convenience, and assurance.

This tool is an open source software and provides features like

- E-mail notification for change in code
- Reports and Charts
- Patch Viewers
- List of bugs can be generated in different formats
- Schedule daily, monthly and weekly reports
- This bug tracking tool detect duplicate bug automatically
- Setting bug priorities by involving customers
- Predict the time a bug may get fixed

Backlog:

Backlog is a popular bug and project tracking tool in one platform. It's easy for anyone to report bugs and keep track of a full history of issue updates and status changes.

- Easy bug tracking tool
- Search and advanced search features
- Full history of issue updates and status changes
- Project and issues with subtasks
- Git and SVN built-in
- Gantt Charts and Burndown charts
- Wikis and Watchlists
- Native mobile apps
- Kanban-style boards for visual workflow

SpiraTeam:

SpiraTeamprovides a complete Application Lifecycle Management (ALM) solution that manages the requirements, tests, plans, tasks, bugs and issues in one environment, with complete traceability.

- Automatic creation of new incidents during the test script execution.
- Fully customizable incident fields including statuses, priorities, defect types and severities
- Ability to link incidents (bugs) to other artifacts and incidents.
- Robust reporting, searching and sorting, plus an Audit log tracking changes.
- Email notifications triggered by the customized workflow status changes.
- Ability to report issues and bugs via email.

Bird Eats Bug:

Bird Eats Bug is a bug reporting tool that helps anyone create interactive data-rich bug reports. While a user makes a screen recording of the issue, Bird's browser extension automatically augments it with valuable technical data like console logs, network errors, browser information, etc.

- Each report auto-includes console logs, network requests and general info (browser/OS, etc.)
- Use mic and/or video recording to explain actions. Save time on typing descriptions.
- Integrations with issue trackers like JIRA, Trello, Github, etc.

Zoho bug tracker:

Zoho bug tracker is a powerful bug tracker that helps you to view issues filtered by priority and severity. It improves the productivity by exactly knowing which bugs are reproducible. It is an online tool that allows you to create projects, bugs, milestone, reports, documents, etc. on a single platform.

Monday:

Monday is a bug tracking tool that that enables you to analyze your performance and manage your team in one place. It provides flexible dashboard for easy visualization of data.

- You can collaborate with other people.
- It can automate your daily work.
- Integration with Mailchimp, Google calendar, Gmail, and more.
- You can track your work progress.
- It enables you to work remotely.

JIRA:

It is easy to use framework. JIRA is a commercial product and helps to capture and organize the team issues, prioritizing the issue and updating them with the project.

It is a tool that directly integrates with the code development environments making it a perfect fit for developers as well.

It comes with many add-ons that make this tool more powerful.

Mantis:

Mantis comes as a web application but also has its own mobile version. It works with multiple databases like MySQL, PostgreSQL, MS SQL and integrated with applications like chat, time tracking, wiki, RSS feeds and many more.

- This is an Open source tool
- This defect tracking tool provides E-mail notification
- Supported reporting with reports and graphs
- Source control integration
- Supports custom fields
- Supports time tracking management
- Multiple projects per instance
- Enable to watch the issue change history and roadmap
- Supports unlimited number of users, issues, and projects

RedMine:

It is an open source bug tracking tool that integrates with SCM (Source Code Management System)too.

It supports multiple platforms and multiple data-bases while for reporting purpose, Gantt charts and calendar are used.

- Gantt chart and calendar
- News, document and files management
- This bug reporting tool provides SCM integration
- Issue creation via e-mail
- This bug tracking software supports multiple database.
- Flexible issue tracking system
- Flexible role based access control
- Multilanguage support

Trac:

Trac is a web based open source issue tracking system that developed in Python.

It is used to browse through the code, view history, view changes, etc. when you integrate Trac with SCM.

It supports multiple platforms like Linux, Unix, Mac OS X, Windows, etc.

Axosoft:

It is a bug tracking system, available for hosted or on-premises software. It is a project management tool for Scrum teams.

It can view each task, its requirement, defects and incidents, in the system, on individual filing cards, through the Scrum planning board.

Axosoft can manage your user stories, defects, support tickets and a real-time snapshot of your progress.

- Scrum planning board
- Scrum burn down charts
- Requirement Management
- Team wiki
- Data visualization
- SCM integration
- Reporting
- Help desk or incident tracking

HP ALM/ Quality Center:

HP ALM is a complete test management solution with a robust integrated bug tracking System within it. HP ALM's bug tracking mechanism is easy and efficient. It supports Agile projects too.

eTraxis:

eTraxis is an open source bug tracking tool supporting multiple languages. This tool is developed in a PHP language and supports multiple-database like Oracle, MySQL, PostgreSQL, and MS Server.

eTraxis gives you a flexible platform involving multiple organizations by providing a central place for all project activity. It allows to create multiple users and projects and at the same time view the bugs assigned.

- Files exchange and supports attachment
- E-mail notification
- Flexible permission
- Powerful filtering on issues
- Custom workflow
- View complete history of all events

IBM Rational ClearQuest:

- IBM ClearQuest tracks, captures and manages any type of bugs
- It support multiplatform, like HP-UX, Linux, Microsoft Windows operating system. It can improve visibility and control of software development projects.
- Integrating with other tools
- Supports real time reporting and metrics
- Increasing team collaboration

Summary

A Software Bug is a failure or flaw in a program that produces undesired or incorrect results. It's an error that prevents the application from functioning as it should.

Reasons For Software Bugs: Miscommunication or No Communication

Software Complexity

Programming Errors

Changing Requirements

Poorly Documented Code

A Bug tracking system is also called as a defect tracking system, this is a software application that captures, reports, and manages data on bugs (errors, exceptions) that occur in software.

Keywords

Bug ***Error***

Fault ***Defect***

Bug tracking

Bug tracking tools

Self Assessment

1. What are the reasons for bug?

A. Wrong coding
B. Missing coding
C. Extra coding
D. All of above

2. What are the classifications of software bugs?

A. Software defects by priority
B. Software defects by severity

C. Software defects by nature
D. All of above

3. Performance defects and Usability defects are part of___
A. Software defects by severity
B. Software defects by nature
C. Software defects by priority
D. None of above

4. Software defects by priority consist of __
A. Critical defects
B. High-severity defects
C. Medium-severity defects
D. None of above

5. Software defects by priority are based upon_____
A. Severity
B. Priority
C. By nature
D. All of above

6. Which is not part of Bug life cycle states?
A. Open
B. Fixed
C. Design
D. Retest

7. Assign, fixed and retest is part of___
A. Software defects by priority
B. Bug life cycle
C. Software defects by nature
D. All of above

8. Which is not bug tracking tool?
A. Zoho tracker
B. Bootstrap
C. Monday
D. JIRA

9. Trac is developed in_
A. Java
B. C++
C. Python
D. None of above

10. eTraxis support___
A. MySQL
B. PostgreSQL
C. MS Server
D. All of above

11. Bug tracking system is__

A. Report of specific bug file
B. It is process of logging and monitoring bugs
C. Option to change the status of the bug
D. All of above

12. Which is not Bug life cycle state?
A. Open
B. Feasibility study
C. Reset
D. None of above

13. eTraxis, Bugnet and Axosoft are__
A. Design tool
B. Testing tools
C. Bug tracking tool
D. None of above

14. Mantis is___
A. Open source tool
B. Bug tracking tool
C. Works with multiple databases like MySQL, PostgreSQL
D. All of above

15. Gantt chart and calendar are used in_
A. Trac
B. RedMine
C. JIRA
D. All of above

Answer for Self Assessment

1.	D	2.	D	3.	B	4.	D	5.	B
6.	C	7.	B	8.	B	9.	C	10.	D
11.	D	12.	B	13.	C	14.	D	15.	B

Review Questions

1. Define bug with example.
2. What are the reasons for bug?
3. Differentiate between bug, error, and fault.
4. Explain any two types of bug.
5. Discuss bug priority and severity.
6. What is significance of bug tracking?
7. Discuss any four bug tracking tools.
8. What are the major features of bug tracing tools?

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition,

Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Links

- www.guru99.com
- www.ece.cmu.edu
- www.softwarequalitymethods.com
- https://www.softwaretestinghelp.com/popular-bug-tracking-software
- https://www.atlassian.com/software/jira/features/bug-tracking
- https://www.softwaretestingmaterial.com/popular-defect-tracking-tools/
- https://www.javatpoint.com/defect-or-bug-tracking-tool

Unit 12: Software Maintenance

CONTENTS

Objectives

After studying this unit, you will be able to:

- discuss software maintenance and software supportability
- describe business process reengineering
- restructuring, economics of reengineering

Introduction

The life of your software does not begin when the code is written and ends when it is released. It instead has a continuous lifespan that stops and starts as needed. Launch marks the beginning of its lifecycle and the start of a significant amount of labour. Software is always changing, and it must be carefully monitored and maintained as long as it is in use. This is mainly to accommodate internal organizational changes, but it is also critical since technology is always evolving.

Your software may require maintenance for a variety of reasons, including keeping it up and running, adding new features, reworking the system to accommodate future changes, moving to the cloud, or making other adjustments. Whatever your incentive for software maintenance is, it is critical to your company's success. As a result, software upkeep entails more than just detecting and addressing flaws. It's about keeping your company's heart beating.

Maintenance may also include adding new features and functionality to an existing software system (using cutting-edge technology). The major goal of software maintenance is to make the software system functional according to the needs of the users and to correct software problems. The issues occur as a result of programme failure or hardware incompatibility with the software.

Program patches are used when only a small portion of the software code needs to be maintained. These patches are only used to correct problems in software code that has errors.

12.1 Software Maintenance

The process of modifying and updating software applications after they have been delivered in order to repair faults and improve performance is known as software maintenance. Software evolution and maintenance improves software performance by Minimising errors, removing unnecessary development, and implementing advanced development techniques.

Software maintenance is a broad term that encompasses optimization, mistake repair, the removal of obsolete functionality, and the augmentation of existing ones. Software maintenance entails making enhancements to a current solution and, on occasion, new software creation to meet changing market demands.

Software maintenance is affected by several constraints such as increase in cost and technical problems with hardware and software. This chapter discusses how software maintenance assists the present software system to accommodate changes according to the new requirements of users.

Let's use the example of an automobile to better grasp the notion of maintenance. When a car is 'used,' its components wear out owing to friction in the mechanical parts, improper use, or environmental factors. The problem is solved by changing the car's components when they become completely unserviceable, and by hiring trained mechanics to tackle complex defects throughout the vehicle's lifetime. The car is occasionally serviced at a service station by the owner. This helps to keep the car in better shape in the future. Similarly, in software engineering, software must be 'serviced' to ensure that it can adapt to changing circumstances (such as business and user needs) in which it operates.

There are number of reasons, why modifications are required, some of them are briefly mentioned below:

Market Conditions - Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.

Client Requirements - Over the time, customer may ask for new features or functions in the software.

Host Modifications - If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.

Organization Changes - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

Why Software maintenance

- Bug Fixing
- Capability Enhancement
- Removal of Outdated Functions
- Performance Improvement
- Market Conditions
- Interface with other systems
- Host Modifications

Bug fixing:

In maintenance management, bug fixing comes at priority to run the software seamlessly. This process contains search out for errors in code and corrects them. The issues can be occurred in hardware, operating systems or any part of the software.

Capability Enhancement:

This comprises an improvement in features and functions to make solutions compatible with the varying market environment. It enhances software platforms, work patterns, hardware upgrades, compilers and all other aspects that affect system workflow.

Removal of Outdated Functions:

The unwanted functionalities are useless. Moreover, by occupying space in solution, they hurt efficiency of the solution. Using software maintenance procedures, such UI and coding elements are removed and replaced with new development using the latest tools and technologies.

Performance Improvement:

To improve system performance, developers detect issues through testing and resolve them. Data and coding restricting as well as reengineering are the part of software maintenance. It prevents the solution from vulnerabilities.

Host Modifications:

If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.

Providing continuity of service:

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

Supporting mandatory upgrades:

Upgrading a software system is made easier with software maintenance. Changes in government regulations or standards may need upgrades. If a web-application system with multimedia capabilities has been established, for example, it may need to be modified in areas where video viewing (through the Internet) is illegal. Upgrading software may also be necessary to stay competitive with other software in the same category.

Improving the software to support user requirements:

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

Facilitating future maintenance work:

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

Changing a Software System

As previously stated, the necessity for software maintenance emerges as a result of changes that must be made to the software system. Anomalies are discovered, new user requirements emerge, and the operating environment changes after a software system has been designed and implemented. This means that, after delivery, software systems are always evolving in response to changing demands.

Lehman was the first to develop the concept of software maintenance and system evolution. He conducted various investigations and offered five laws based on his findings. Large systems are never complete and continue to evolve, according to one of the studies' primary findings. It's worth noting that when systems evolve, they grow more complicated, necessitating some steps to lessen the complexity. The five Lehman laws are presented in Table and explained further below.

Continuing change:

Because systems function in a dynamic context, this law states that change is unavoidable. As the system's environment changes, new requirements emerge, and the system must be adjusted. When the modified system is reintroduced into the environment, it necessitates additional environmental changes. It's worth noting that if a system remains static for a long time, it won't be able to meet the users' ever-changing needs. This is due to the fact that the system may become obsolete over time.

Increasing complexity:

According to this law, as a system evolves, its structure deteriorates (often observed in legacy systems). To avoid this issue, preventive maintenance should be utilized, in which the software's structure is enhanced but no new functionality is added. However, to counteract the effects of structural degradation, extra costs must be paid.

Large software evolution:

Because of organizational elements that are established early in the development process, this law states that software evolution for large systems is mostly dependent on management decisions. This is especially true for major corporations, which have their own internal bureaucracies in charge of decision-making. The rate of change of the system in these organizations is determined by the decision-making processes of the organization. This establishes the overall trends of the system maintenance process and restricts the number of system changes that can be made.

Organizational stability:

This law states that changes to resources such as staffing have unnoticeable effects on evolution. For example, productivity may not increase by assigning new staff to a project because of the additional communication overhead. Thus, it can be said that large software development teams become unproductive because the communication overheads dominate the work of the team.

Conservation of familiarity:

This law indicates that the rate at which new functionality can be added has a limit. This means that adding a significant amount of functionality to a system in a single release will almost certainly result in new system flaws. If a significant increment is introduced, a new release will be necessary 'quite rapidly' to address the new system flaws. As a result, companies should not plan for major functional increments in each release without factoring in the need for bug fixes.

12.2 Types of Software Maintenance

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive maintenance

Corrective Maintenance:

Detecting errors in the existing solution and correcting them to make it works more efficiently. This type of software maintenance aims to eliminate and fix bugs or defects in the software. Corrective software maintenance is often done in the form of small updates frequently.

Corrective software maintenance is the most common and traditional type of maintenance (for software and anything else for that matter). When something goes wrong with a piece of software, such as bugs or errors, corrective software maintenance is required. These can have a large impact on the software's overall functionality, so they must be addressed as soon as feasible.

Many times, software vendors can address issues that require corrective maintenance due to bug reports that users send in. If a company can recognize and take care of faults before users discover them, this is an added advantage that will make your company seem more reputable and reliable (no one likes an error message after all).

Adaptive maintenance:

Modifications in the system to keep it compatible with changing business and technical environment. This type of software maintenance concentrate on software infrastructure.To retain continuity with the software, adaptive maintenance are made in response to new operating systems, hardware, and platforms.

Adaptive software maintenance is concerned with changing technologies as well as your software's policies and procedures. Changes to the operating system, cloud storage, and hardware are just a few examples. When these modifications are made, your programme must adapt in order to match the new requirements and continue to function effectively.

Perfective maintenance:

Fine tuning of all elements, functionalities and abilities to improve system operations and perfectness. The software's accessibility and usability are solved by perfective software maintenance. Perfective maintenance includes altering current software functionality by improving, removing, or inserting new features or functions.

When software is released to the public, new concerns and ideas emerge, just as they do with any other product. Users may recognise the need for more features or requirements in the software in order to make it the greatest tool for their needs. Perfective software maintenance comes into play at this point. Perfective software maintenance strives to fine-tune software by adding new features when needed and deleting components that are no longer relevant or useful in the current software. As the market and user needs evolve, this approach ensures that software remains relevant.

Preventive maintenance:

Preventive software maintenance services help in preventing the system from any upcoming vulnerabilities. Preventive maintenance means improvements to the software, which is done to secure the software for the future. This maintenance is carried out to prevent the product from any potential software alteration.

Preventative software maintenance entails planning ahead of time to ensure that your programme continues to function as intended for as long as possible. This includes making appropriate adjustments, enhancements, and modifications, among other things. Preventative software maintenance can address minor issues that may seem insignificant at the time but could grow into major concerns in the future. These are known as latent flaws, and they must be identified and repaired so that they do not become active faults.

12.3 Software Maintenance Activities

As per IEEE framework sequential maintenance process activities includes…

Identification & Tracing	Analysis
Design	Implementation
System testing	Acceptance Testing
Delivery	Maintenance management

Identification and Tracing: - It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.

Analysis: - The modification is analysed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for.A set of required modifications is then materialized into requirement specifications. The cost of modification /maintenance is analysed and estimation is concluded.

Design: - New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.

Implementation: - The new modules are coded with the help of structured design created in the design step. Every programmer is expected to do unit testing in parallel.

System Testing: - Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures

Acceptance Testing: - After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.

Delivery: - After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.

Maintenance management: - Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

12.4 Software Supportability

Software supportability refers to a software system's ability to be supported during its entire product lifecycle. Software supportability refers to a set of software design attributes, related development tools and methodologies, and environmental infrastructure that make it possible to accomplish software support tasks.

This includes not only meeting any necessary needs or requirements, but also providing the necessary equipment, support infrastructure, new software, facilities, labour, or other resources to keep the programme functioning and capable of performing its purpose.

Supportability is the ability of a complete system design to support operations and readiness needs at a reasonable cost throughout the system's life cycle. It allows for the evaluation of a complete system design's suitability for a set of operational requirements within the anticipated operations and support environment.

Software Support encompass

Key aspects associated to the software

- Operation
- Logistics Management
- Modification

Operation:

Operation covers all aspects associated to the actual use of the software, including the installation, loading (or unloading), configuration, error recovery and execution of the software.

Logistics Management:

Logistics Management covers all aspects related to the handling of the software once a new baseline has been produced, until its delivery to the end user.

Modification:

Modification covers all aspects related to the evolution of the software due to the need of fixing bugs, or adding/changing functionality due to changing user needs.

Software supportability process

- Program planning and control
- Mission, development & support systems definition
- Preparation and evaluation of alternatives
- Determination of SAS requirements
- Software supportability assessment

Program planning and control:

- Development of an early Software supportability Strategy
- Program and Design Reviews

Mission, development & support systems definition:

- Use Study
- Technological Opportunities
- Standardization
- Design Factors
- Comparative Analysis
- Integration of ADP Systems

Preparation and evaluation of alternatives:

- Functional and Non-Functional Requirements
- Support System Alternatives
- Evaluation of Alternatives & Trade-Offs

Determination of SAS requirements:

- Operational Task Analysis
- Software Exception/Problem Support Analysis
- Software Modification Analysis
- Software Transition Analysis
- Post-Deployment Software Support/Logistics Management Analysis

Software supportability assessment:

- Operation
- Modification
- Problem Reaction
- Logistics Management
- Lessons Learned

Software Supportability Principles

Designing for supportability requires diligence on the part of both managers and engineers

What can managers do to identify vulnerabilities?

"What if" scenarios

Ask your technical team what the Achilles heel is – They will tell you!

- What can engineers to improve supportability through design?
- Use a fully replicated production environment for pre-release testing
- Parameterize using configuration files
- Use frameworks to control design
- Carefully evaluate COTS products before incorporating into the design
- Incorporate distributed component design up front

Check the "ilities":

- Security
- Reliability
- Flexibility
- Maintainability
- Scalability
- Availability

Manage change:

- Don't attempt too much change at once
- Evaluate system impacts with changing requirements
- Use the CCB (Configuration Control Board)
- Resist the temptation to "just add it in this time"

Quality Control:

- Monitor to maintain quality and identify new risks

  Keep CMMI inspections technical

  Develop processes and follow them

- Enforce Independent Verification and Validation

  At a minimum, developers should not test their own code

- QA person should report to Program Manager

Organize for Supportability:

- Supportability failures often occur between teams or areas of expertise
  i.e., software team, network team, SA, Security, etc.

- Mitigation strategy

Assign someone the specific role of enforcing cross- disciple technical quality

12.5 Factors Affecting Software Supportability

Change traffic: Changing traffic is a complex function of requirement stability, software integrity and system usage. Changing traffic will affect the task quantity of software support, and changing traffic will require more software modification work.

Expansion: Extension ability is an attribute related to system design, and insufficient extension ability may limit the scope of software modification and have an important impact on the cost of modification.

System number and deployment location: The number and distribution of the system will have a great impact on the cost of support, but also affect the formulation of software support requirements and software support workload. It also affects the distribution of software support facilities.

Modularization: Modularization facilitates the implementation of software support, and the poor degree of modularization usually leads to an increase in the cost of modification, as other parts of the software need to be modified accordingly.

Software scale: The size of the software has an impact on change traffic, in addition to the amount of resources required to make changes.

Confidentiality: The secret level of data, executable code and documents may impose restrictions and requirements on software support activities, as well as special processing requirements on the system. These effects will limit the use of software and put forward corresponding requirements for design, thus requiring special software support tasks and equipment.

Personnel skills: The software modification needs the personnel who have the corresponding software engineering skill to complete, the software personnel skill level cannot reach the request will affect the software support realization, at the same time, the stipulation skill also will have the influence to the training request.

Standardization: Software development processes and related software documentation to meet standardization requirements can reduce the variety of tools, skills and facilities required and help improve software supportability.

Documentation: Documentation refers to all records, including electronic and paper records, i.e. requirements, design, implementation, testing and using documents related to software products. Documentation is the main factor affecting software supportability.

12.6 Reengineering

It is process of redesign of business processes and the associated systems and organizational structures to achieve a dramatic improvement in business performance.

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written. Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

Motivations for reengineering

- Reduce costs/expenses
- Improve financial performance
- Reduce external competition pressure
- Reverse erosion of market share
- Respond to emerging market opportunities
- Improve customer satisfaction
- Enhance quality of products and services

Reengineering project guidelines

- Planning and launching
- Current state assessing and learning from others
- Designing solution
- Developing business case justification
- Developing solution
- Implementing solution

12.7 Business Process Reengineering

Business process re-engineering is the radical redesign of business processes to achieve dramatic improvements in critical aspects like quality, output, cost, service, and speed. Business process reengineering (BPR) aims at cutting down enterprise costs and process redundancies on a very huge scale.

Business process reengineering is the act of recreating a core business process with the goal of improving product output, quality, or reducing costs.It involves the analysis of company workflows, finding processes that are sub-par or inefficient, and figuring out ways to get rid of them or change them.

Steps of business process reengineering

- Map the current state of your business processes
- Analyze them and find any process gaps or disconnects
- Look for improvement opportunities and validate them
- Design a cutting-edge future-state process map
- Implement future state changes and be careful of dependencies

Map the current state of your business processes:

Gather data from all resources–both software tools and stakeholders. Understand how the process is performing currently.

Analyze them and find any process gaps or disconnects:

Identify all the errors and delays that hold up a free flow of the process. Make sure if all details are available in the respective steps for the stakeholders to make quick decisions.

Look for improvement opportunities and validate them:

Check if all the steps are absolutely necessary. If a step is there to solely inform the person, remove the step, and add an automated email trigger.

Design a cutting-edge future-state process map:

Create a new process that solves all the problems you have identified. Don't be afraid to design a totally new process that is sure to work well. Designate KPIs for every step of the process.

Implement future state changes and be careful of dependencies:

Inform every stakeholder of the new process. Only proceed after everyone is on board and educated about how the new process works. Constantly monitor the KPIs (Key Performance Indicators).

12.8 Software Reengineering

Software Reengineering is the process of updating software without affecting its functionality.It is a process of software development which is done to improve the maintainability of a software system. Re-engineering is the examination and alteration of a system to reconstitute it in a new form.

This process may be done by developing additional features on the software and adding functionalities that may or may not be required but considered to make the software experience better and more efficient.It affects positively at software cost, quality, service to the customer and speed of delivery.

Need of software Re-engineering

Processes in continuity: The functionality of older software product can be still used while the testing or development of software.

Boost up productivity: Software reengineering increase productivity by optimizing the code and database so that processing gets faster.

Reduction in risks: Instead of developing the software product from scratch or from the beginning stage here developers develop the product from its existing stage to enhance some specific features that are brought in concern by stakeholders or its users.

Drastic change in technology: The situation when the initially promising technology is replaced by more successful and advanced alternatives is common in IT. The market is constantly evolving and, if the company wants to keep up with technology, reengineering process becomes a necessity.

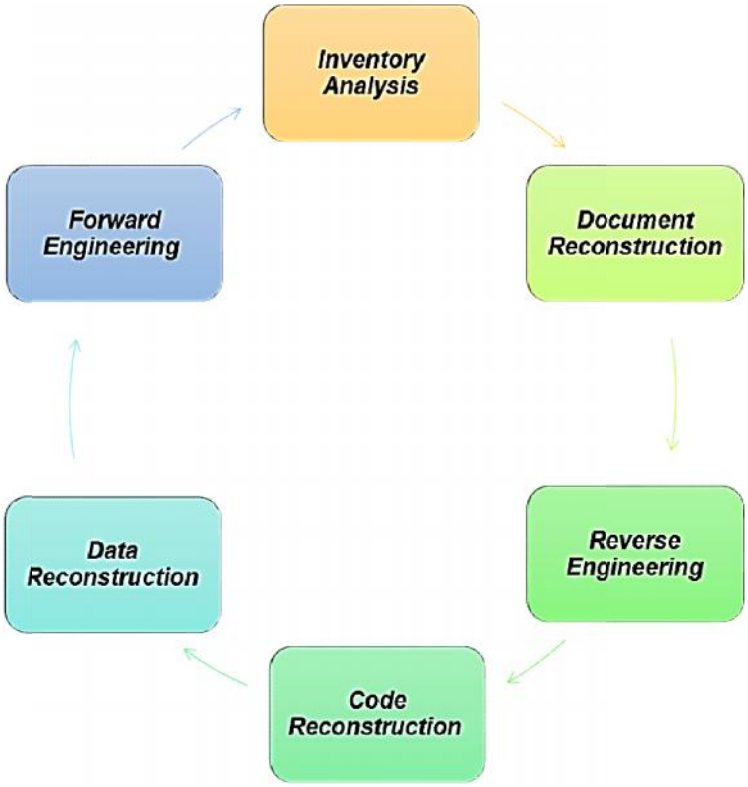
Saves time: As we stated above here that the product is developed from the existing stage rather than the beginning stage so the time consumes in software engineering is lesser.

Optimization: This process refines the system features, functionalities and reduces the complexity of the product by consistent optimization as maximum as possible.

This is an approach by which you can find useless steps and resources that have been implemented in the latest software and then you can remove them from the application

12.9 Steps for Software Re-Engineering

- Inventory Analysis
- Document Reconstruction
- Reverse Engineering
- Code Reconstruction
- Data Reconstruction
- Forward Engineering



Inventory Analysis:

Inventory containing information that provides a detailed description of every active application.By sorting this information according to business criticality, longevity, current maintainability and other local important criteria, candidates for re-engineering appear.The resource can then be allocated to a candidate application for re-engineering work.

Document reconstructing:

- Documentation of a system either explains how it operates or how to use it.
- Documentation must be updated.
- It may not be necessary to fully document an application.
- The system is business-critical and must be fully re-documented.

Reverse Engineering:

Reverse engineering involves the exploration of the existing system to recapture current requirements (including all connections and interdependencies), interfaces between software components, data structure, and data design.

To extract the lost info about the application state, business analysts interview stakeholders, and programmers, study the existing application documentation, perform its lexical and syntactic code analysis, investigate control and data flows, explore use and test cases.

Code Reconstructing:

To accomplish code reconstructing, the source code is analysed using a reconstructing tool. Violations of structured programming construct are noted and code is then reconstructed.The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced.

Data Restructuring:

Data restructuring begins with a reverse engineering activity.Current data architecture is dissected, and the necessary data models are defined.Data objects and attributes are identified, and existing data structure are reviewed for quality.

Forward Engineering:

Forward engineering also called as renovation or reclamation not only for recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

Advantages of Re-engineering

Productivity increase. By optimizing the code and database the speed of work is increased.

Improvement opportunity. You can not only refine the existing product, but also expand its capabilities by adding new features.

Risk reduction. Development from scratch is always a more risky exercise, as opposed to a phased upgrade of the existing system.

Time saving. Instead of starting development from scratch, the existing solution is simply transferred to a new platform, saving all business-logic.

Optimization potential. You can refine the system functionality and increase its flexibility, ensuring better compliance with the enterprise's current objectives.

Processes continuity. The old product can be used while testing the new system until all work is completed.

12.10 Economics of Reengineering and Restructuring

Economics of Re-Engineering

The objective of re-engineering is to reduce maintenance costs. It can achieved by increasing quality and reducing complexity. The costs of re-engineering are driven by the size and interdependency of the software as well as by the degree of automation available.

Parameters for cost/benefit analysis model for reengineering

- P1 = current annual maintenance cost for an application.
- P2 = current annual operation cost for an application.
- P3 = current annual business value of an application.
- P4 = predicted annual maintenance cost after reengineering.
- P5 = predicted annual operations cost after reengineering.
- P6 = predicted annual business value after reengineering.
- P7 = estimated reengineering costs.
- P8 = estimated reengineering calendar time.
- P9 = reengineering risk factor (P9 = 1.0 is nominal).
- L = expected life of the system.

The cost associated with continuing maintenance of a candidate application

$$C_{maint} = [P3 - (P1 + P2)] \times L$$

The costs associated with reengineering are defined using the following relationship

$$C_{reeng} = [P6 - (P4 + P5) \times (L - P8) - (P7 \times P9)]$$

Using the costs presented in equations, the overall benefit of reengineering can be computed as

$$\text{Cost benefit} = C_{reeng} - C_{maint}$$

Cost of maintenance = cost annual of operation and maintenance over application lifetime

Cost of reengineering = predicted return on investment reduced by cost of implementing changes and engineering risk factors

Cost benefit = Cost of reengineering - Cost of maintenance

The cost/benefit analysis presented in the equations can be performed for all high priority applications identified during inventory analysis.Those application that show the highest cost/benefit can be targeted for reengineering, while work on others can be postponed until resources are available.

Software Restructuring

Software restructuring is a form of perfective maintenance that modifies the structure of a program's source code. Its goal is increased maintainability to better facilitate other maintenance activities, such as adding new functionality to, or correcting previously undetected errors within a software system.

Restructuring of software is usually performed in either higher or lower levels of granularity, where the first indicates broader changes in the system's structural architecture and the latter indicates re-factorings performed to fewer and localized code elements.

Types of Restructuring

Code restructuring

Program logic modeled using Boolean algebra and series of transformation rules are applied to yield restructured logic. Create resource exchange diagram showing data types, procedure and variables shared between modules, restructure program architecture to minimize module coupling.

Types of Restructuring

- Data restructuring
- Analysis of source code
- Data redesign
- Data record standardization
- Data name rationalization
- File or database translation

Summary

Maintaining a system is not less important than Web Application Development. It keeps solutions athletic to deal with developing technology and business environment.

Re-Engineering: Restructuring or rewriting part or all of a system without changing its functionality.

Applicable when some (but not all) subsystems of a larger system require frequent maintenance. Reengineering involves putting in the effort to make it easier to maintain. The reengineered system may also be restructured and should be re-documented.

Types of maintenance

Corrective Software Maintenance

Adaptive Software Maintenance

Perfective Software Maintenance

Preventive Software Maintenance

Keywords

Software Maintenance

Software reengineering

Restructuring

Economics of reengineering

Forward Engineering

Code Reconstruction

Data Reconstruction

Self Assessment

1. Software maintenance is used for__
A. Removal of Outdated Functions
B. Performance Improvement
C. Market Conditions
D. Above all

2. Which is not type of Software Maintenance?
A. Unit maintenance
B. Adaptive maintenance
C. Perfective maintenance
D. Above all

3. Software Maintenance Activities includes_
A. Design
B. Implementation
C. System testing
D. Above all

4. Software Support encompass__
A. Operation
B. Logistics Management
C. Modification
D. Above all

5. Software supportability process not include__
A. Program planning and control
B. System testing
C. Preparation and evaluation of alternatives

D. Determination of SAS requirements

6. What are the Software Supportability Principles
A. Check the "ilities"
B. Manage Change
C. Control Quality
D. Above all

7. What are the Steps of business process reengineering?
A. Analyze them and find any process gaps or disconnects
B. Look for improvement opportunities and validate them
C. Design a cutting-edge future-state process map
D. Above all

8. Which is not part of Reengineering project guidelines__
A. Designing solution
B. Developing business case justification
C. Redesign the system
D. Developing solution

9. What are the Motivations for reengineering?
A. Reduce external competition pressure
B. Reverse erosion of market share
C. Respond to emerging market opportunities
D. Above all

10. What are the parameters for Economics of Re-Engineering?
A. Current annual maintenance cost for an application.
B. Current annual operation cost for an application.
C. Estimated reengineering costs
D. Above all

Answer for Self Assessment

1. D	2. A	3. D	4. D	5. B
6. D	7. D	8. C	9. D	10. D

Review Question

1. What is significance of software maintenance?
2. Why software maintenance is required?
3. Differentiate between Corrective maintenance and Adaptive maintenance.
4. Differentiate between Perfective maintenance and Preventive maintenance.
5. Explain different Software Maintenance Activities
6. Discuss the concept of Software reengineering.
7. Define Forward Engineering.

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering—A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education.

Web Links

- http://www.scribd.com
- http://en.wikipedia.org
- https://radixweb.com/blog/why-software-maintenance-is-necessary
- https://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm
- https://www.javatpoint.com/software-engineering-software-maintenance
- https://cpl.thalesgroup.com/software-monetization/four-types-of-software-maintenance

Unit 13: Product Metrics

CONTENTS

Objectives

After studying this unit, you will be able to:

- Describe measure, metrics and indicators
- Demonstrate measurement principles
- Metricsfor specification quality

Introduction

Metrics are only useful if they have undergone statistical validation. The control chart is a simple way to do this while also looking at the variation and distribution of metrics findings. Culture changes as a result of measurement. To launch a metrics programme, three steps must be completed: data collecting, metrics computation, and metrics analysis. A goal-driven strategy, in general, assists an organization in focusing on the correct metrics for its business.Software engineers and their managers can obtain a better understanding of the work they do and the product they make by developing a metrics baseline, which is a database comprising process and product measurements. An indicator is a measure or set of metrics that provides information about the software development process, a software project, or a product. An indicator gives information that allows the project manager or software engineers to improve the process, the project, or the process by adjusting the process, the project, or the process.

13.1 Software Metrics

A software metric is a quantitative or countable measure of programme properties. Software metrics are useful for a variety of purposes, including evaluating software performance, planning work items, calculating productivity, and more. Software metrics are useful for a variety of purposes, including assessing software performance, planning work items, and determining productivity. Many indicators are intertwined throughout the software development process. Software metrics are comparable to management's four functions: planning, organization, control, and improvement.

Characteristics of software Metrics

Quantitative: Metrics must possess quantitative nature. It means metrics can be expressed in values.

Understandable: Metric computation should be easily understood, the method of computing metric should be clearly defined.

Applicability: Metrics should be applicable in the initial phases of development of the software.

Repeatable: The metric values should be same when measured repeatedly and consistent in nature.

Economical: Computation of metric should be economical.

Language Independent: Metrics should not depend on any programming language

13.2 Product Metrics

Product metrics are software product measures at any stage of their development, from requirements to established systems. Product metrics are related to software features only.

Why have Software Product Metrics?

- Help software engineers to better understand the attributes of models and assess the quality of the software
- Help software engineers to gain insight into the design and construction of the software
- Focus on specific attributes of software engineering work products resulting from analysis, design, coding, and testing
- Provide a systematic way to assess quality based on a set of clearly defined rules
- Provide an "on-the-spot" rather than "after-the-fact" insight into the software development

Types of Metrics

Internal metrics: Internal metrics are the metrics used for measuring properties that are viewed to be of greater importance to a software developer. For example, Lines of Code (LOC) measure.

External metrics: External metrics are the metrics used for measuring properties that are viewed to be of greater importance to the user, e.g., portability, reliability, functionality, usability, etc.

Hybrid metrics: Hybrid metrics are the metrics that combine product, process, and resource metrics. For example, cost per FP where FP stands for Function Point Metric.

Project metrics: Project metrics are the metrics used by the project manager to check the project's progress. Data from the past projects are used to collect various metrics, like time and cost; these estimates are used as a base of new software.

Software Metrics classification

Product metrics − Describes the characteristics of the product such as size, complexity, design features, performance, and quality level.

Process metrics − these characteristics can be used to improve the development and maintenance activities of the software.

Project metrics − this metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

Product metrics fall into two classes:

Dynamic metrics that are collected by measurements made from a program in execution.Dynamic metrics are usually quite closely related to software quality attributes. It is relatively easy to measure the execution time required for particular tasks and to estimate the time required to start the system. These are directly related to the efficiency of the system failures and the type of failure can be logged and directly related to the reliability of the software.

Static metrics that are collected by measurements made from system representations such as design, programs, or documentation.Static matrices have an indirect relationship with quality attributes. A large number of these matrices have been proposed to try to derive and validate the relationship between the complexity, understandability, and maintainability. Several static metrics which have been used for assessing quality attributes.

Software Product Metrics:

Fan-in/Fan-out

Fan-in is a measure of the number of functions that call some other function (say X). Fan-out is the number of functions which are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of the control logic needed to coordinate the called components.

Length of code

This is measure of the size of a program. Generally, the large the size of the code of a program component, the more complex and error-prone that component is likely to be.

Cyclomatic complexity

This is a measure of the control complexity of a program. This control complexity may be related to program understandability. It is invented by Thomas McCabe (1974) to measure the complexity of a program's conditional logic. It counts the number of decisions in the program, under the assumption that decisions are difficult for people. It makes assumptions about decision-counting rules and linear dependence of the total count to complexity.

Cyclomatic complexity, also known as V(G) or the graph theoretic number, is probably the most widely used complexity metric in software engineering. Defined by Thomas McCabe, it's easy to understand and calculate, and it gives useful results. This metric considers the control logic in a procedure. It's a measure of structural complexity. Low complexity is desirable.

How to calculate cyclomatic complexity?

CC = Number of decisions + 1

The cyclomatic complexity of a procedure equals the number of decisions plus one. What are decisions? Decisions are caused by conditional statements. In Visual Basic they are If. Else If. Else, Case, For.Next, Until, While, Catch and When. In order to get CC, one simply counts the conditional statements. A multi-way decision, the Select Case block, typical counts as several conditional statements. The decisions counted for each statement or construct are listed below.

Decisions Counted for each Statement

Construct	Decisions	Reasoning
If..Then	+1	An "If statement" is a single decision.
ElseIf..Then	+1	ElseIf adds a new decision.
Else	0	Else does not cause a new decision. The decision is at the If.
#If..#ElseIf..#Else	0	Conditional compilation adds no run-time decisions.
Select Case	0	Select Case initiates the following Case branches, but does not add a decision alone.
Case	+1	Each Case branch adds a new decision.
Case Else	0	Case Else does not cause a new decision. The decisions were made at the other Cases.
For [Each] .. Next	+1	There is a decision at the For statement.
Do While| Until	+1	There is a decision at the start of the Do..Loop.
Loop While| Until	+1	There is a decision at the end of the Do..Loop.
Do..Loop alone	0	There is no decision in an unconditional Do..Loop without While or Until.*
While	+1	There is a decision at the start of the While..Wend or While..End While loop.
Catch	+1	Each Catch branch adds a new conditional path of execution. Even though a Catch can be either conditional (catches specific exceptions) or unconditional (catches all exceptions), we treat all of them the same way.*
Catch..When	+2	The When condition adds a second decision.*

The minimum limit for cyclomatic complexity is 1. This happens with a procedure having no decisions at all. There is no maximum value since a procedure can have any number of decisions.

Variations: CC, CC2 and CC3

Cyclomatic complexity comes in a couple of variations as to what exactly counts as a decision. Project Analyzer supports three alternative cyclomatic complexity metrics. CC is the basic version. CC2 and CC3 use slightly different rules.

CC does not count Boolean operators such as And and Or. Boolean operators add internalcomplexity to the decisions, but they are not counted in CC. CC and CC3 are similar what comesto Booleans, but CC2 is different.

CC2 Cyclomatic Complexity with Booleans ("Extended Cyclomatic Complexity")

CC2 = CC + Boolean operators

CC2 extends cyclomatic complexity by including Boolean operators in the decision count.

Whenever a Boolean operator (And, Or, Xor, Eqv, And Also, Or Else) is found within a conditionalstatement, CC2 increases by one. The statements considered are: If, Else If, Select, Case, Until,While, When.

The reasoning behind CC2 is that a Boolean operator increases the internal complexity of adecision. CC2 counts the "real" number of decisions, regardless of whether they appear as asingle conditional statement or split into several statements. Instead of using Boolean operatorsto combine decisions into one (x = 1 And y = 2), you could as well split the decisions into severalsub-conditions (If x = 1 Then If y = 2 Then). CC2 is immune to this kind of restructuring, whichmight be well justified to make the code more readable. On the other hand, one can decrease CCsimply by combining decisions with Boolean operators, which may not make sense. Including Boolean operators in cyclomatic complexity was originally suggested by Thomas McCabe. In this sense, both CC and CC2 are "original" cyclomatic complexity measures. Alternative names: CC2 is also known as ECC extended cyclomatic complexity or strict cyclomatic complexity.

CC3 Cyclomatic Complexity without Cases ("Modified Cyclomatic Complexity")

CC3 = CC where each Select block counts as one

CC3 equals the regular CC metric, but each Select Case block is counted as one branch, not as multiple branches. In this variation, a Select Case is treated as if it were a single big decision.

This leads to considerably lower complexity values for procedures with large Select Case statements. In many cases, Select Case blocks are simple enough to consider as one decision, which justifies the use of CC3.

Alternative Name: CC3 is sometimes called modified cyclomatic complexity.

Summary of cyclomatic complexity metrics is shown in the table.

Summary of Cyclomatic Complexity Metrics

Metric	Name	Boolean operators	Select Case	Alternative name
CC	Cyclomatic complexity	Not counted	+1 for each Case branch	Regular cyclomatic complexity
CC2	Cyclomatic complexity with Booleans	+1 for each Boolean	+1 for each Case branch	Extended or strict cyclomatic complexity
CC3	Cyclomatic complexity without Cases	Not counted	+1 for an entire Select Case	Modified cyclomatic complexity

CC, CC2 or CC3—which one to use? This is your decision. Pick up the one that suits your use best. CC and CC2 are "original" metrics and probably more widely used than CC3. The numeric values are, in increasing order: CC3 (lowest), CC (middle) and CC2 (highest). In a sense, CC2 is the most pessimistic metric. All of them are heavily correlated, so you can achieve good results with any of them.

Cyclomatic Complexity and Select Case

The use of multi-branch statements (Select Case) often leads to high cyclomatic complexity values. This is a potential source of confusion. Should a long multi-way selection be split into several procedures?

McCabe originally recommended exempting modules consisting of single multi-way decision statements from the complexity limit.

Although a procedure consisting of a single multi-way decision may require many tests, each test should be easy to construct and execute. Each decision branch can be understood and maintained in isolation, so the procedure is likely to be reliable and maintainable. Therefore, it is reasonable to exempt procedures consisting of a single multi-way decision statement from a complexity limit.

Total Cyclomatic Complexity (TCC)

The total cyclomatic complexity for a project or a class is calculated as follows:

TCC = Sum (CC) - Count (CC) + 1

TCC equals the number of decisions + 1 in a project or a class. It's similar to CC but for several procedures.

Sum (CC) is simply the total sum of CC of all procedures. Count (CC) equals the number of procedures. It's deducted because we already added +1 in the formula of CC for each procedure.

TCC is immune to modularization, or the lack of modularization. TCC always equals the number of decisions + 1. It is not affected by how many procedures the decisions are distributed.

TCC can be decreased by reducing the complexity of individual procedures. An alternative is to eliminate duplicated or unused procedures.

DECDENS (Decision Density)

Cyclomatic complexity is usually higher in longer procedures. How much decision is there actually, compared to lines of code? This is where you need decision density, which also known as cyclomatic density.

DECDENS = Sum(CC)/LLOC

This metric shows the average cyclomatic density in your project. The numerator is sum of CC over all your procedures. The denominator is the logical lines of code metric. DECDENS ignores single-line procedure declarations since cyclomatic complexity isn't defined for them.

Length of identifiers

This is a measure of the average length of distinct identifier in a program. The longer the identifiers, the more understandable the program.

Depth of conditional nesting

This is a measure of the depth of nesting of if statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.

Fog index

This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document may be to understand.

13.3 Product Quality Metrics

Mean Time to Failure: It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

Defect Density: It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

Customer Problems: It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.

Customer Satisfaction: Customer satisfaction is often measured by customer survey data through the five-point scale

Very satisfied

Satisfied

Neutral

Dissatisfied

Very dissatisfied

Advantages of software metrics

- The design methodology of the software systems can be studied comparatively.
- The characteristics of various programming languages can be studied for analysis and comparison using software metrics.
- The software quality specifications can be prepared using software metrics.
- The compliance of requirements and specifications of software systems can be verified.
- The complexity of the code can be determined.
- The decision of whether to divide a complex module or not can be done
- The utilization of resource managers to their fullest can be guided.
- Design trade-offs and comparing maintenance costs and software development costs can be done.

Disadvantages of software metrics

- The available tools and the working environment is used to define and derive the software metrics, and there is no standard in defining and deriving them.
- Certain variables are estimated based on the predictive models, and they are not known so often.
- It is not easy to apply metrics in all cases. It is difficult and expensive in some cases.
- It is difficult to verify the validity of historical or empirical data on which the verification and justification.
- Software products can be managed, but the technical staff's performance cannot be evaluated using software metrics.

A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process

As per IEEE a metric is "a quantitative measure of the degree to which a system, component, or process possesses a given attribute"

An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself

13.4 Software Measurement

A measurement is an manifestation of the size, quantity, amount or dimension of a particular attributes of a product or process. It is an authority within software engineering. Software measurement process is defined and governed by ISO Standard.

The most important product metric is its size. Size may be expressed in terms of the LOC, number of classes or number of bytes of source code. But LOC is the most popular metric. Software is measured for the following reasons:

(a) To indicate the quality of the product.

(b) To assess the productivity of the people.

(c) To form a baseline for estimation.

(d) To justify the needs of new tools and staff training.

(e) To access advantages for new tools and methods.

Size-Oriented Metrics/LOC Metrics/Direct Measures

LOC or thousands of LOC (KLOC) has been a very traditional and direct method of metrics of software. These metrics are derived by normalizing the quality and productivity measures by considering the size of the product as a metric. Please remember the following points regarding

LOC measures:

1. In size-oriented metrics, LOC is considered to be the normalization value.

2. It is an older method that was developed when FORTRAN and COBOL programming were very popular.

3. Productivity is defined as KLOC / EFFORT where effort is measured in person-months.

4. Size-oriented metrics depend on the programming language used.

5. As productivity depends on KLOC, so assembly language code will have more productivity.

6. LOC measure requires a level of detail which may not be practically achievable.

7. The more expressive is the programming language, the lower is the productivity.

8. LOC method of measurement is not applicable to projects that deal with visual (GUIbased) programming. As already explained, Graphical User Interfaces (GUIs) use formsbasically. LOC metric is not applicable there.

9. It requires that all organizations must use the same method for counting LOC. This is sobecause some organizations use only executable statements, some use comments and somedo not. So, a standard needs to be established.

10. These metrics are not universally accepted.

Based on the LOC/KLOC count of software, many other metrics can be computed:

(a) Errors/KLOC.

(b) $/ KLOC.

(c) Defects/KLOC.

(d) Pages of documentation/KLOC.

(e) Errors/PM.

(f) Productivity = KLOC/PM (effort is measured in person-months).

(g) $/ Page of documentation.

Advantages and disadvantages of LOC measure:

Proponents (in favor) of LOC measure claim that LOC is an artifact of all softwaredevelopment projects that can be easily counted.

Opponents (against), however, argue that LOC measures are programming languagedependent. That is, a C program will take lesser LOC as compared to a COBOL/assemblylanguage program for a given task. For example, a bubble sort program in C will take lessernumber of LOC as compared to a COBOL program of bubble sort.

Text-Based Metrics

Another metrics that may be used is the size of the text of SRS. The size could be the number of pages, number of paragraphs, number of functional requirements, etc.

Advantages and disadvantages of text-based measure:

Proponents (in favor) of text-based metric say that it is used mostly to convey a general sense about the size of the product or project.

Opponents (against) say that such metrics are dependent on the authors of the document and are not the accurate indicators of the size of the project.

Function Oriented Metrics

In 1977, A. J. Albrecht of IBM developed a method of software metrics based on the functionality of the software delivered by an application as a normalization value. He called it the Function Points (FPs). They are derived using an empirical relationship based on direct measures of software's information domain and assessments of software complexity. Note that FPs try to quantify the functionality of the system, i.e., what the system performs. This is taken as the method of measurement as FPs cannot be measured directly. Also note that FP is not a single characteristic but is a combination of several software features/characteristics. A non-profit organization called International Function Point User Group (IFPUG) was formed in 1986 that took the responsibility of developing the counting rules and FP definitions. Today, IFPUG has more than 3000 members and has affiliates in 24 countries. FP also has many variants like backfired FPs, COSMIC FPs, Feature Points, The Netherlands FPs, etc. Remember the following points regarding FPs:

1. FPs of an application is found out by counting the number and types of functions used in the applications. Various functions used in an application can be put under five type

Types of FP Attributes

Measurement Parameter	*Examples*
1. Number of external inputs (EI)	Input screen and tables.
2. Number of external outputs (EO)	Output screens and reports.
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories.
5. Number of external interfaces (EIF)	Shared databases and shared routines.

All these parameters are then individually assessed for complexity.

2. FP characterizes the complexity of the software system and hence can be used to depict the project time and the manpower requirement.

3. The effort required to develop the project depends on what the software does.

4. FP is programming language independent.

5. FP method is used for data processing systems, business systems like information systems.

6. The 5 parameters mentioned above are also known as information domain characteristics.

7. All the above-mentioned parameters are assigned some weights that have been experimentally determined

Measurement Parameter	*Low*	*Average*	*High*

Weights of 5-FP Attributes

1. Number of external inputs (EI)	7	10	15
2. Number of external outputs (EO)	5	7	10
3. Number of external inquiries (EQ)	3	4	6
4. Number of internal files (ILF)	4	5	7
5. Number of external interfaces (EIF)	3	4	6

The functional complexities are multiplied with the corresponding weights against each function and the values are added up to determine the UFP (Unadjusted Function Point) of the subsystem.

Computing FPs

Measurement Parameter	*Count*		*Weighing factor*			
			Simple	*Average*	*Complex*	
1. Number of external inputs (EI)	—	*	3	4	6 =	—
2. Number of external outputs (EO)	—	*	4	5	7 =	—
3. Number of external inquiries (EQ)	—	*	3	4	6 =	—
4. Number of internal files (ILF)	—	*	7	10	15 =	—
5. Number of external interfaces (EIF)	—	*	5	7	10 =	—
Count-total →						—

Note here that weighing factor will be simple, average or complex for a measurement parameter type.

The Function Point (FP) is thus calculated with the following formula

FP = Count-total * [0.65 + 0.01 * ∑(Fi)]

= Count-total * CAF

Where Count-total is obtained from the above Table.

CAF = [0.65 + 0.01 * ∑(Fi)]

and S(Fi) is the sum of all 14 questionnaires and show the complexity adjustment value/ factor-CAF (where i ranges from 1 to 14). Usually, a student is provided with the value of S (Fi). Also note that S (Fi) ranges from 0 to 70, i.e.,

0 <= ∑ (Fi) <=70

and CAF ranges from 0.65 to 1.35 because

(a) When ∑ (Fi) = 0 then CAF = 0.65

(b) When ∑ (Fi) = 70 then CAF = 0.65 + (0.01 * 70) = 0.65 + 0.7 = 1.35

Based on the FP measure of software many other metrics can be computed:

(a) Errors/FP

(b) $/FP.

(c) Defects/FP

(d) Pages of documentation/FP

(e) Errors/PM.

(f) Productivity = FP/PM (effort is measured in person-months).

(g) $/Page of Documentation.

8. LOCs of an application can be estimated from FPs. That is, they are interconvertible. This process is known as backfiring. For example, 1 FP is equal to about 100 lines of COBOL code.

9. FP metrics is used mostly for measuring the size of Management Information System

(MIS) software.

10. But the function points obtained above are unadjusted function points (UFPs). These

(UFPs) of a subsystem are further adjusted by considering some more General System

Characteristics (GSCs). It is a set of 14 GSCs that need to be considered. The procedure for adjusting UFPs is as follows:

(a) Degree of Influence (DI) for each of these 14 GSCs is assessed on a scale of 0 to 5.

(b) If a particular GSC has no influence, then its weight is taken as 0 and if it has a strong influence then its weight is 5.

(c) The score of all 14 GSCs is totaled to determine Total Degree of Influence (TDI).

(d) Then Value Adjustment Factor (VAF) is computed from TDI by using the formula:

VAF = (TDI * 0.01) + 0.65

Remember that the value of VAF lies within 0.65 to 1.35 because

(a) When TDI = 0, VAF = 0.65

(b) When TDI = 70, VAF = 1.35

(e) VAF is then multiplied with the UFP to get the final FP count:

FP = VAF * UFP

Extended Function Point (EFP) Metrics

FP metric has been further extended to compute:

(a) Feature points.

(b) 3D function points.

Feature Points:

According to Jones,

1. Feature point is the superset of function point measure that can be applied to systems and engineering software applications.
2. The feature points are used in those applications in which the algorithmic complexity is high like real-time systems where time constraints are there, embedded systems, etc.
3. Feature points are computed by counting the information domain values and are weighed by only single weight.
4. Feature point includes another measurement parameter – ALGORITHM.
5. The table for the computation of feature point.

Feature Point Calculations

Measurement Parameter	*Count*		*Weighing factor*	
1. Number of external inputs (EI)	–	*	4	—
2. Number of external outputs (EO)	–	*	5	—
3. Number of external inquiries (EQ)	–	*	4	—
4. Number of internal files (ILF)	–	*	7	—
5. Number of external interfaces (EIF)	–	*	7	—
6. Algorithms used	–	*	3	—
Count-total →				—

The feature point is thus calculated with the following formula:

FP = Count-total * [0.65 + 0.01 * ∑ (Fi)]

= Count-total * CAF

where count-total is obtained from the above table.

CAF = [0.65 + 0.01 * ∑ (Fi)]

and ∑ (Fi) is the sum of all 14 questionnaires and show the complexity adjustment value/ factor CAF (where i ranges from 1 to 14). Usually a student is provided with the value of S (Fi).

6. Function point and feature point both represent systems functionality only.

7. For real-time applications that are very complex, the feature point is between 20 and 35% higher than the count determined using function point above.

3D-Function Points

According to Boeing,

1. Three dimensions may be used to represent 3D function points–data dimension, functional dimension and control dimension.

2. The data dimension is evaluated as FPs are calculated. Herein, counts are made for inputs, outputs, inquiries, external interfaces and files.

3. The functional dimension adds another feature—Transformation, that is, the sequence of steps which transforms input to output.

4. The control dimension that adds another feature—Transition that is defined as the total number of transitions between states. A state represents some externally observable mode.

Computing FP's

Measurement Parameter	*Count*		*Weighing factor*			
			Simple	*Average*	*Complex*	
1. Number of external inputs (EI)	32	*	3	4	6 =	128
2. Number of external outputs (EO)	60	*	4	5	7 =	300
3. Number of external inquiries (EQ)	24	*	3	4	6 =	96
4. Number of internal files (ILF)	8	*	7	10	15 =	80
5. Number of external interfaces (EIF)	2	*	5	7	10 =	14
Count-total →						618

Now Fi for average case = 3.

So sum of all Fi (i ¨ 1 to 14) = 14 * 3 = 42

FP = Count-total * [0.65 + 0.01 * ∑ (Fi)]

= 618 * [0.65 + 0.01 * 42]

= 618 * [0.65 + 0.42]

= 618 * 1.07

= 661.26

and feature point = (32 *4 + 60 * 5 + 24 * 4 + 80 +14) * 1.07 + {12 * 15 *1.07}

= 853.86

Shortcomings of Function Point Metric

Subjective Evaluations: It needs subjective evaluation with a lot of judgement involved.

Conversion need: Many efforts and models are based on LOC, a function point need to be converted.

Less Researched Data: Less research data is available on function point as compared to LOC.

Low Accuracy: It has low accuracy of evaluating as a subjective judgement is involved.

Time consuming: It is a time consuming method as less research data is available which generate low accuracy and less effective results.

Late performance: It is performed after creation of design specification.

Long learning curve: As the learning curve is quite long it's not easy to gain proficiency.

Classification of Software Measurement

Direct Measurement: In direct measurement the product, process or thing is measured directly using standard scale.

Indirect Measurement: In indirect measurement the quantity or quality to be measured is measured using related parameter i.e. by use of reference.

13.5 Measurement Principles

The objectives of measurement should be established before data collection begins

Each technical metric should be defined in an unambiguous manner

Metrics should be tailored to best accommodate specific products and processes

Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable)

Activities of a Measurement Process

Formulation: The derivation (i.e., identification) of software measures and metrics appropriate for the representation of the software that is being considered

Collection: The mechanism used to accumulate data required to derive the formulated metrics

Analysis: The computation of metrics and the application of mathematical tools

Interpretation: The evaluation of metrics in an effort to gain insight into the quality of the representation

Feedback: Recommendations derived from the interpretation of product metrics and passed on to the software development team

13.6 ISO Standards

Mark-II − ISO/IEC 20968:2002 Software engineering - Ml II Function Point Analysis - Counting Practices Manual.

NESMA − ISO/IEC 24570:2005 Software engineering - NESMA function size measurement method version 2.1 - Definitions and counting guidelines for the application of Function Point Analysis.

COSMIC − ISO/IEC 19761:2011 Software engineering. A functional size measurement method.

FiSMA − ISO/IEC 29881:2008 Information technology - Software and systems engineering - FiSMA 1.1 functional size measurement method.

IFPUG − ISO/IEC 20926:2009 Software and systems engineering - Software measurement - IFPUG functional size measurement method.

Case Tools For Software Metrics

Many CASE tools (Computer Aided Software Engineering tools) exist for measuring software.

They are either open source or are paid tools. Some of them are listed below:

1. Analyst4j tool is based on the Eclipse platform and available as a stand-alone Rich Client

Application or as an Eclipse IDE plug-in. It features search, metrics, analyzing quality,and report generation for Java programs.

2. CCCC is an open source command-line tool. It analyzes C++ and Java lines andgenerates reports on various metrics, including Lines of Code and metrics proposed by

Chidamber & Kemerer and Henry & Kafura.

3. Chidamber & Kemerer Java Metrics is an open source command-line tool. It calculatesthe C&K object-oriented metrics by processing the byte-code of compiled Java.

4. Dependency Finder is an open source. It is a suite of tools for analyzing compiled Javacode. Its core is a dependency analysis application that extracts dependency graphs andmines them for useful information. This application comes as a command-line tool, a

Swing-based application and a web application.

5. Eclipse Metrics Plug-in 1.3.6 by Frank Sauer is an open source metrics calculation anddependency analyzer plugin for the Eclipse IDE. It measures various metrics and detectscycles in package and type dependencies.

6. Eclipse Metrics Plug-in 3.4 by Lance Walton is open source. It calculates various metricsduring build cycles and warns, via the problems view, of metrics 'range violations'.

7. OOMeter is an experimental software metrics tool developed by Alghamdi. It accepts

Java/C# source code and UML models in XMI and calculates various metrics.

8. Simmel is an Eclipse plug-in. It provides an SQL like querying language for object-oriented code, which allows searching for bugs, measure code metrics, etc.

Summary

- The standard of measure for the estimation of quality, progress and health of the software testing effort is called software metrics.
- Types of Software Metricscan be divided into three groups: product metrics, process metrics, and project metrics.
- Cyclomatic Complexity counts the number of decisions in the program, under the assumption that decisions are difficult for people.
- Cyclomatic complexity, also known as V(G) or the graph theoretic number, is probably the most widely used complexity metric in software engineering.
- Cyclomatic complexity comes in a couple of variations as to what exactly counts as a decision.
- Boolean operators add internal complexity to the decisions, but they are not counted in
- CC. CC and CC3 are similar what comes to Booleans, but CC2 is different.
- Cyclomatic complexity equals the minimum number of test cases you must execute to cover every possible execution path through your procedure.
- The properties which are of great importance to a software developer can be measured using the metrics called internal metrics.
- The properties which are of great importance to a user can be measured using the metrics called external metrics. An example is portability, reliability, usability, etc.

Keywords

Metrics	Product metrics
Process metrics	Project metrics.
Function point	External Inputs(EI)
External Output (EO)	External inquiries (EQ)
Internal files (ILF)	External interfaces (EIF)

Self Assessment

1. What are the characteristics of software Metrics?

A. Quantitative
B. Applicability
C. Repeatable
D. Above all

2. Which is not type of Software Metrics?
A. Internal metrics
B. Product metrics
C. External metrics
D. Project metrics

3. Product metrics classes are_
A. Dynamic
B. Static
C. Both dynamic and static
D. None of above

4. Software Product Metrics include
A. Fan-in/Fan-out
B. Length of code
C. Cyclomatic complexity
D. All of above

5. Which is not part of Product Quality Metrics?
A. Defect Density
B. Product metrics
C. Customer Problems
D. Customer Satisfaction

6. What are the point scale for Customer Satisfaction?
A. Very satisfied
B. Satisfied
C. Neutral
D. Above all

7. Why we need of software measurement?
A. Create the quality of the current product or process.
B. Anticipate future qualities of the product or process.
C. Enhance the quality of a product or process.
D. Above all

8. Software Measurement include?
A. Direct Measurement
B. Indirect Measurement
C. Both direct and indirect measurement
D. None of above

9. Which is not activities of a measurement process
A. Testing
B. Formulation
C. Collection
D. Feedback

10. How many types of data functions
A. 4
B. 3
C. 2
D. 5

11. Transaction Functions include?
A. External Inputs
B. External Outputs
C. External Inquiries
D. All of above

12. Which is not types of functional point attributes?
A. EQ
B. AI
C. EI
D. ILF

13. What are the shortcomings of function point metric?
A. Conversion need
B. Low Accuracy
C. Time consuming
D. All of above

14. What are the parameter for computing the unadjusted function point (UFP)
A. Number of inputs
B. Number of inquiries
C. Number of files
D. All of above

Answer for Self Assessment

1.	D	2.	B	3.	C	4.	D	5.	B
6.	D	7.	D	8.	C	9.	A	10.	C
11.	D	12.	B	13.	D	14,	D		

Review Questions

1. What is significance of software metric?
2. Explain the concept of Cyclomatic complexity.
3. Discuss the different variations of Cyclomatic complexity.
4. Discuss Product metrics.
5. Differentiate between Process MetricsInternal Metrics.
6. What are the shortcomings of function point metric?
7. What is functional point attributes?
8. Differentiate between Internal Logical Files and External Interface Files.

Further Reading

- Books Rajib Mall, Fundamentals of Software Engineering, 2nd Edition, PHI.
- Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.
- R.S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, Tata
- McGraw Hill Higher education.
- Sommerville, Software Engineering, 6th Edition, Pearson Education

Web Links

- http://c2.com/cgi/wiki?CyclomaticComplexityMetric
- http://www.guru99.com/cyclomatic-complexity.html
- http://sunnyday.mit.edu/16.355/classnotes-metrics.pdf
- https://www.javatpoint.com/software-engineering-software-metrics
- https://www.tutorialspoint.com/software_quality_management/software_quality_management_metrics.htm

Unit 14: Software Process Improvement

Objective

After studying this unit, you will be able to:

- Discuss approaches to Software process improvement
- Explain maturity models
- Discuss SPI process

Introduction

Software Process Improvement (SPI) is a systematic strategy to improving a software development organization's capacity to build and deliver high-quality software on time and on budget. Software Process, also known as Software Development Process or Software Development Life Cycle (SDLC), is a method of breaking down the software development process into a series of actions or phases that contribute to the creation or development of software. Designing, implementing, changing, testing, and maintaining are examples of these activities.

These activities are carried out in order to improve the project's design and administration. Customizing and improving software processes are other strategies or activities used to improve the overall development process. Software projects especially major software projects are on average 66 percent over budget and 33 percent behind time; up to 17 percent of projects are so terrible that they jeopardize the company's existence.

To address this issue, certain methodologies and strategies for addressing software process concerns, suggesting various improvements and identifying faults and inefficiencies in the process, began to emerge. These strategies have become a standard that firms can use to improve their software development process.

Software process improvement is a method for improving processes. The ultimate goal of software process improvement is to produce a better result than the standard procedure. In previous decades, a crisis occurred from 1965 and 1985. At that time, the rate of project failure was rapidly

increasing. The output would not be as good as it needs to be. The crisis manifested in a variety of ways:

1. Project running over budget

2. Project running over time

3. Software was low quality

4. Software often did not meet requirements

5. Project was unmanageable and code difficult to maintain. A lot of problems have come in to the project which leads to not a better result. For example Lack of management, lack of communication among the member of organization. Software process improvement is a set of activities by which processes can achieve a high quality in the organization.

14.1 Software Process Improvement

The Software Process Improvement (SPI) approach is characterised as a set of tasks, tools, and strategies for planning and executing improvement activities with specific goals in mind, such as boosting development speed. Software Process Improvement (SPI) aims to reduce the time, cost, and quality of software engineering and management operations.

SPI activities in software businesses are usually carried out using well-defined reference models such as CMMI and ISO 15504. These models are used to improve development speed, product quality, and cost reduction. SPI is a process re-engineering or change management initiative that identifies inefficiencies in the software development lifecycle and resolves them to improve the process. To be of genuine benefit to the organization, this process should be mapped and connected with organizational goals and change drivers.

A change in the process to produce a better result rather than the original outcome is known as process improvement. A variety of factors influence the software development process. For long-term business, projects should be completed on time, on budget, and with higher quality. It is possible to reuse software process enhancement for the following project. Software process improvement is a set of operations that help organizations mature their processes, achieve all of their objectives, and produce a high-quality output. We can define process improvement as "Software process improvement is a deliberate, planned methodology following standardized documentation practices to capture on paper (and in practice) the activities, methods, practices, and transformations that people use to develop and maintain software and the associated products. As each activity, method, practice and transformation is documented, each is analyzed against the standard of value added to the organization". Software process improvement improves not only quality but also the process of organization.

Software process improvement term is used for improving the organization.

There are some activities for this;

1. Identify the current process

2. Identify the current status of the organization

3. Identify the strength and weakness of the organization

4. Analyse where the changes are needed

5. What will be the effect of these changes?

6. Understand these changes that how much they are affected

Some steps for the process improvement are following:

1. Aware for organization culture

2. Consistence should be there

3. Make target and keep in mind

4. Put a option in the process

5. Set up a well communication team member and senior management

6. Treat process improvement like a project

7. Make changes according to the priority

8. Make a profile of the company

9. Introduce a software process engineering group (SEPG) to your organization

10. Have patience

11. Reuse the previous result

Why do we need for Software Process Improvement?

An immature organization becomes a mature organization as a result of software process improvement. A young company would be unable to provide a high-quality product. There are numerous reasons why high-quality software fails. There is no rationale in the work practices, and there is no support from senior management and management. There is no way to reuse this project because it is just working. The lack of sufficient project documentation renders it unusable for future projects. The project does not meet the needs of the customers. Customer requirements differ from what the project can provide. The project is not completed on schedule or within budget. There are a lot of shortcomings in immature organization. But mature organization provides a good quality to costumer with in proper time and budget. There is commitment between the employees. If there is any problem or any problem between management then senior management handle the problem and give a better solution.

14.2 Steps for SPI

- Current Situation Evaluation
- Improvement Planning
- Improvement Implementation
- Improvement Evaluation

Steps for SPI

Current Situation Evaluation

To assess the current situation of the software process by eliciting the requirements from the stakeholders, analyzing the current artefacts and deliverables, and identifying the inefficiencies from the software process.

The elicitation can be conducted through different techniques. For example, individual interviews, group interview, use-case scenarios, and observations. The key considerations in this step to identify organization goals and ask the solution-oriented questions.

Improvement Planning

After analyzing the current situation and the improvement goals, the findings should be categorized and prioritized according to which one is the most important or have the most severity.

In this step, the gap between the current level and the target level should be planned in terms of a set of activities to reach that target. These activities should be prioritized with the alignment of the involved stakeholders and the organization goals.

Improvement Implementation

In this step, the planned activities are executed and it puts the improvements into practice and spreads it across the organization, what can be effective at the 2nd, 3rd, and 4th step that planning and implementation could be an iterative way.

This iterative way of implementation will help the organization to realize the early benefits from the SPI programme early or even adopt the plan if there is no real impact measured from the improvement.

Improvement Evaluation

In this stage measure the changes in case of Improvement. the impact measurement is applied compared with the GQM. The before improvement measures, after the improvement measures, and the target improvement measure. Measurement, in general, permits an organization to compare the rate of actual change against its planned change and allocate resources based on the gaps between actual and expected progress.

Different SPI methods

Similar to the SDLC, SPI has a lot of methods and you can as well define your own method if it is effective or combine between more than one if you do not have any preferences or organization need to adopt a specific method.

SPI methods mainly categorized into two categories: inductive (bottom-up approach) and prescriptive (top-down approach) and most of them are cyclic with four main steps. The table below contains examples of the methods in each category.

SPI methods mainly categorized into two categories:

Inductive (bottom-up approach)

Quality Improvement Paradigm (QIP)

Improvement framework utilizing lightweight assessment and improvement planning (iFLAP)

Prescriptive (top-down approach)

Capability Maturity Model Integration (CMMI)

Software Process Improvement and Capability determination (SPICE)

Advantages of SPI

Standardization and Process consistency: To have a standard and practical process for software development mapped to organization goals and strategy

Cost Reduction: To improve projects cost by enhancing the process and eliminate issues, redundancies, and deficiencies.

Competitive Edge: Being certified in CMMI for example, can put the company in higher competitive edge and make it gain more sales due to the evidence of existing mature software process based on standard method.

Meeting targets and reduce time to market: Meeting organization goals, projects delivery, quality standards, valuable products, professional documentation are outputs from SPI.

Improve customer's satisfaction: Project delivery on time and based on the specification with high quality will improve customer's satisfaction and improve the sales process.

Automation and Autonomy: Introducing tools to automate things and improve quality and ensure consistency. Moreover, enabling different employees to play different roles in the project.

Demotivation factors for SPI

Budget Constraints: SPI is a costly process, because it needs time and dedicated resources, and not only that but also skilled resources especially in SPI. Company may need SPI consultant and train the resources and orient them on SPI initiative.

Time pressure: Due to the nature of the companies to deliver the projects on time, they faced a lot of time pressure which make it harder for them to dedicate time to the SPI project.

Inadequate metrics: Most of the small companies do not have metrics to measure and compare their progress or improvement which make it sometimes impossible to identify measure the improvements of the SPI.

Lack of Management Commitment: It is mainly because the management cannot understand the benefit from SPI and they do not fully support doing this change as well as the other factors like lack of resources, budget, time,etc.

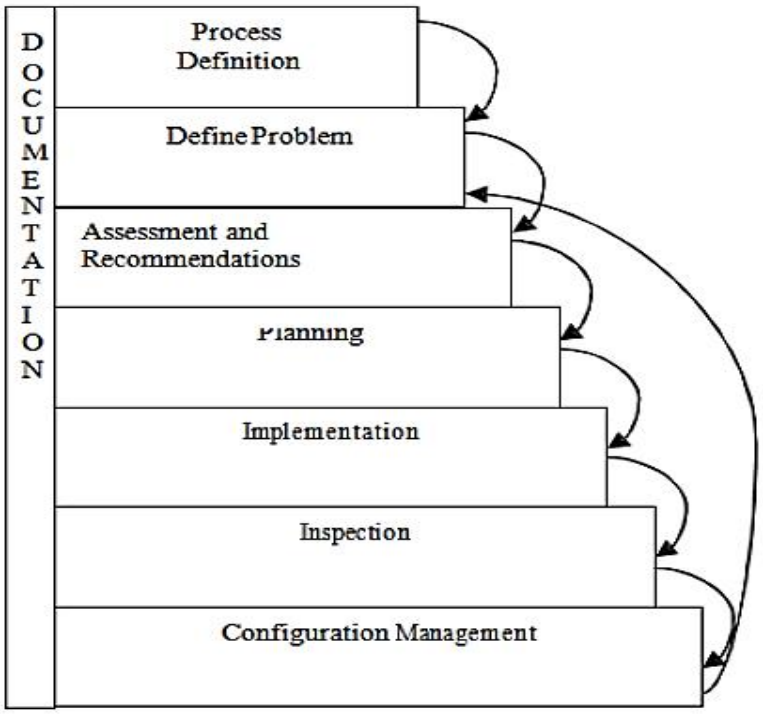
Staff turnover: Sometimes, the company has a high staff turnover which can be an issue to impose the SPI culture change and this can lead to endless SPI.

Micro Organization: Some organization are very small and have very few resources. The SPI will be too big for that kind of companies.

14.3 Software Process Improvement Model (SPIM)

Software process improvement model is an approach or method or both by which it improves process and give better result rather than a normal process. With the software process improvement a better and high quality product could be found within budget and time. The researcher proposes a Software process improvement model in this paper. This model is an iterative as well as cyclic model and it consists of eight steps.

Software process improvement model (SPIM)



Problem in project

There are three problems which are mainly to be solved in the software project.

1. Projects are not of good quality

2. Projects do not deliver in time

3. Project goes over budget

Process Definition and Problem Define

A doctor gives the medicine when he knows about the patient diseases. Without knowing they cannot do anything. Similarly any process improvement model can work accurately if one know the exact problems in the process. Therefore the main issue is to make problem clear at first. When problem become clear then a solution can be found.

A lot of reason may be part of unsuccessful project.

1. Lack of management

2. Lack of right communication inside organization

3. Lack of right communication between costumer and organization

4. Technical problem and lack of resources

5. Lack of man power

6. Lack of good training

7. Ego problem

8. Not working in adequate way

Assessment and Recommendation

Once the problem has been described, it is simple to identify areas that need to be improved, and an assessment team is formed based on the problem and the recommendations of higher authorities. The size of the problem and the size of the team are determined by the management. The team should be made up of personnel that are highly qualified and experienced. If the problem is modest, a team of two to three researchers will suffice. If the improvement area is large, a team will be formed based on the problem's size. Sometimes software enhancement is ineffective because no one can claim or guarantee that a methodology or process will improve the project's quality or the organization's culture. So if area of improvement is small like in small project, then risk is reduced. But if area of improvement is of higher level then it may be a great risk. A project/organization take time for improvement and have no guarantee for success, but only probability for the success of same. So at this condition it is better to take sponsorship by the third party. But it requires higher authority's permission. They can take decision about timing of sponsorship needed. They have a meeting with senior management about the improvement of process and assessment of team like who should be the member of team for process improvement. This is very important step because team should be consist of highly qualified member and as small as possible with their role defined and known to them. Member of the team must be certified by SEI (software engineering institute) as only then they can take better decision in critical situation.

Planning

They begin working on the project after the evaluation team has been formed. They devise a strategy for improving the process by identifying improvements that are required. First and foremost, the scope of the problem should be clearly identified. The team is now creating surveys for individuals from various levels inside the business. After analyzing the company's existing state, a profile for the company's future is created. They do this by interviewing as many people as possible and devising a strategy. If there are multiple strategies for improving a process, they should choose the one that yields the best results. With the approach defined, a detailed implementation plan can be developed by taking consideration of all the factors like, technical and nontechnical. When the team complete the plan, they have a meeting with the senior management and there they give a presentation to the seniors. By doing this they reduce risk level. At this situation assessment team is not responsible for any casualty. If the senior management comfortable with your plan, only then team would be allow to move towards next step of implementation plan. If they don't satisfied with plan, then alternative plan have to make along with the presentation. This plan includes schedule, tasks, milestones, decision points, resources, responsibilities,

measurement, tracking mechanisms, risks and mitigation strategies, and any other elements required by the organization.

Implementation

When the plan is established in accordance with the problem, the solution is created in accordance with the plan. Make a paper solution first, based on the evaluation team's experience, knowledge, and expertise. After that, he makes another presentation to the higher authorities. If they offer any suggestions for improving the solution, adjust the plan accordingly, and if they are satisfied with the solution, begin implementation. They opt for a better option. The plan should be discussed with top management and the organization before it is implemented. Begin working on the solution once they have reached an agreement. It is a step-by-step method that does not apply to all of the solutions in one go. Makes a baseline after every step and take a pilot test and check the result if the result is positive then next continue the solution. If there are more than one solution. Then one shouldn't take decision by himself in this situation. Again discuss with the higher authorities. Whatever solution seems better, apply the solution. By doing this risk is less because at this situation team is not responsible and if the plan be successful the all the credit goes to the assessment team.

Inspection and Configuration Management

In most cases, an inspection is a systematic examination or a formal evaluation process. It entails the application of measurements, tests, and gauges to specific properties of an object or activity. The results are frequently compared to pre-determined requirements and standards to see if the item or activity meets these goals. The majority of inspections are nondestructive. The accuracy of the work plan and execution cannot be guaranteed. A team of moderator, reader, and inspectors is assembled for the inspection. There is a formal meeting. The moderator's job is to keep the meeting on track and guarantee that the agenda is followed. Role of themoderator is like a leader of the inspection team. So moderator should behighly qualified and skillful. Moderator tells the team how the inspection startsand leads them. Inspection team takes the interview of assessment team. Theymake a questionnaire for every member of team.

Review of the work done by all, and they analyses the result. What resultwill come after process improvement? Review all the factors came in thetechnique. Main motive of the inspection is to find the defect in the module.If there are no defects then team will give the clean chit for the processimprovement. If there is any problem in the technique, then they tell where theproblem is or what problem can be faced. What change should be done in process? They specify where the changes are need. This work is handling bythe configuration management. Before a configuration item makes a baseline ofthe work. A baseline is a software configuration management concept thathelps us to control changes without seriously impeding justifiable changes.Once a baseline is established, we pass through a one way door. After thatchanges inspection team again inspect the process. This process will continuetill the defect does not end. And the improvement has done.

Documentation

In this paradigm, documentation is an umbrella activity that occurs concurrently with the process. Each phase necessitates documentation. Documentation is required for future work so that it can be reused and new work can be completed. There are a number of requirements related with the documents linked with a software project and the system being developed:

They should act as a communication medium between members of thedevelopment team. They should be a system information repository to be used by maintenance engineers.

Software Process Improvement Models

- CMMI
- SIX SIGMA
- SPICE
- TickIT Guide ISO 9001
- BOOTSTRAP

CMMI

CMMI is a framework that provides a process improvement setup for the software engineering and product development. CMMI combined best practices and basic principles and interlinked them in order to improve the process inside an organization.

CMMI currently addresses three areas of interest:

Product and service development - CMMI for Development (CMMI-DEV)

Service establishment, management, and delivery - CMMI for Services (CMMI-SVC)

Product and service acquisition - CMMI for Acquisition (CMMI-ACQ)

SIX SIGMA

Six Sigma is a process improvement management framework to achieve bottom-line results, and customer's loyalty. The objective of Six Sigma is the implementation of a measurement based strategy that is focused on process improvement and variation reduction.

Six Sigma is a statistically-based process improvement methodology that aims to reduce defects to a rate of 3.4 defects per million defect opportunities by identifying and eliminating causes of variation in business processes. SIX SIGMA is iterative methodology reduced the defect one by one. Six Sigma focuses on prioritizing and solving specific problems which are selected based on the strategic priorities of the company and the problems which are causing the most defects. SIX SIGMA methodology works on two approaches DMAIC, DMADV.

SPICE

The purpose of SPICE framework is to analyse and assess the software process and on these assessments, provide information about process strengths, weaknesses and capability to achieve its goals. SPICE was inspired by CMM and other softwareprocess improvement models.

TickIT Guide ISO 9001

The purpose of stimulating software system developers to think about process quality, how to achieve the desired quality and how to manage this quality in order to perform continuous improvement. TicketIT also provides a practical framework for the management of software development quality.

BOOTSTRAP

BOOTSTRAP is a European method for software process assessment and improvement that was designed and developed to speed up the application of software engineering technology.

14.4 Maturity Model

A maturity model is applied within the context of an SPI framework.The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization.

- an indication of the quality of the software process, the degree to which practitioner's understand and apply the process,
- The general state of software engineering practice.

Capability Maturity Model (CMM)

Capability Maturity Model is used as a standard to measure the maturity of an organization's software process. CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987.

Capability Maturity Model (CMM) specifies an increasing series of levels of a software development organization. The higher the level, the better the software development process.

It isn't a model of a software process. It is a framework for analysing any organization's approach and processes for developing software products. It also includes advice for further improving the maturity of the software development process. It is built on the most effective feedback and growth strategies used by the world's most successful firms.

This model describes a strategy for software process improvement that should be followed by moving through 5 different levels.Each level of maturity shows a process capability level. All the levels except level-1 are further described by Key Process Areas (KPA's).

Key Process Areas (KPA's)

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

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

CMM Maturity Levels

- Initial
- Repeatable
- Defined
- Initial Managed
- Optimizing

CMM -three principal areas

Organization and resource management. This deals with functional responsibilities, personnel, and other resources and facilities.

Software engineering process and its management. This concerns the scope, depth, and completeness of the software engineering process and the way in which it is measured, managed, and improved.

Tools and technology. This deals with the tools and technologies used in the software engineering process and the effectiveness with which they are applied. This section is not used in maturity evaluations or assessments.

Limitations of CMM Models

- CMM determines what a process should address instead of how it should be implemented.
- It does not explain every possibility of software process improvement.
- CMM will not be useful in the project having a crisis right now.
- It concentrates on software issues but does not consider strategic business planning, adopting technologies, establishing product line and managing human resources.
- It does not tell on what kind of business an organization should be in.

14.5 Maturity models – CMMI

CMMI model is a proven set of best practices organized by critical business capabilities which improve business performance.It is designed to be understandable, accessible, flexible, and integrate with other methodologies such as agile.

CMMI is a framework that provides a process improvement setup for the software engineering and product development. CMMI combined best practices and basic principles and interlinked them in order to improve the process inside an organization.

CMMI highlights

- Reduce rework by 54%.
- Improve product quality up to 70%.
- Improve productivity up to 54%.
- Average improvement in defect containment 23%
- Development velocity increased up to 38%

CMMI V2.0 Key Improvements

Improve Business Performance

Business goals are tied directly to operations in order to drive measurable, improved performance against time, quality, budget, customer satisfaction and other key drivers.

Build Agile Resiliency and Scale

Direct guidance on how to strengthen agile with Scrum project processes with a focus on performance.

Increase Value of Benchmarking

New performance-oriented appraisal method improves reliability and consistency of benchmarks while reducing preparation time and lifecycle costs.

Accelerate Adoption

Online access and adoption guidance make the benefits of CMMI more accessible than ever.

14.6 Appraisal

An appraisal is an activity that helps you to identify the strengths and weaknesses of your organization's processes and to examine how closely the processes relate to CMMI best practices.

Need of Appraisals

Determine how well the organization's processes compare to CMMI best practices, and identify areas where improvement can be made.Inform external customers and suppliers about how well the organization's processes compare to CMMI best practices.Meet customer contractual requirements.

CMMI Appraisal Method

The CMMI appraisal method is used to identify process implementation strengths and weaknesses as well as process persistence and habit.

The method is also used to link demonstrated business performance relative to the model adoption

Standard CMMI Appraisal Method for Process Improvement (SCAMPI)

There are three appraisal classes: Class A, B and C.

SCAMPI A: The most rigorous appraisal method, SCAMPI A is most useful after multiple processes have been implemented. It provides a benchmark for businesses and is the only level that results in an official rating.

SCAMPI B: This appraisal is less formal than SCAMPI A; it helps find a target CMMI maturity level, predict success for evaluated practices and give the business a better idea of where they stand in the maturity process.

SCAMPI C: This appraisal method is shorter, more flexible and less expensive than Class A or B. It's designed to quickly assess a business's established practices and how those will integrate or align with CMMI practices. It can be used at a high-level or micro-level, to address organizational issues or smaller process or departmental issues.

The new appraisal method is capable of supporting appraisals in a variety of contexts:

- Benchmarking
- Internal performance and process improvement
- Process monitoring
- Supplier selection
- Risk reduction

14.7 CMMI Maturity Levels

- Initial
- Repeatable

- Defined
- Initial Managed
- Optimizing

Initial: Processes are viewed as unpredictable and reactive. At this stage, "work gets completed but it's often delayed and over budget." This is the worst stage a business can find itself in - an unpredictable environment that increases risk and inefficiency.

Managed: There's a level of project management achieved. Projects are "planned, performed, measured and controlled" at this level, but there are still a lot of issues to address.

Defined: At this stage, organizations are more proactive than reactive. There's a set of "organization-wide standards" to "provide guidance across projects, programs and portfolios."Businesses understand their shortcomings, how to address them and what the goal is for improvement.

Quantitatively managed: This stage is more measured and controlled. The organization is working off quantitative data to determine predictable processes that align with stakeholder needs. The business is ahead of risks, with more data-driven insight into process deficiencies.

Optimizing: Here, an organization's processes are stable and flexible. At this final stage, an organization will be in constant state of improving and responding to changes or other opportunities. The organization is stable, which allows for more "agility and innovation," in a predictable environment.

CMMI model components

Process areas

24 process areas that are relevant to process capability and improvement are identified. These are organized into 4 groups.

Goals

Goals are descriptions of desirable organizational states. Each process area has associated goals.

Practices

Practices are ways of achieving a goal - however, they are advisory and other approaches to achieve the goal may be used.

CMMI Process Area

A Process Area is a cluster of related practices in an area that, when implemented collectively, satisfy a set of goals considered important for making significant improvement in that area. All CMMI process areas are common to both continuous and staged representations.

The CMMI Process Areas PAs can be grouped into the four categories to understand their interactions and links with one another regardless of their defined level

- Process Management
- Project Management
- Engineering
- Support

Each process area is defined by a set of goals and practices. Generic goals and practices: They are part of every process area.Specific goals and practices: They are specific to a given process area. There are 22 process areas in CMMI that indicate the various aspects of product development.

Project Management

Integrated Project Management (IPM)

Project Monitoring and Control (PMC)

Project Planning (PP)

Quantitative Project Management (QPM)

Requirements Management (REQM)

Risk Management (RSKM)

Supplier Agreement Management (SAM)

Engineering

Product Integration (PI)

Requirements Development (RD)

Technial Solution (TS)

Validation (VAL)

Verification (VER)

Process Management

Organizational Performance Management (OPM)

Organizational Process Definition (OPD)

Organizational Process Focus (OPF)

Organizational Process Performance (OPP)

Organizational Training (OT)

Support

Casual Analysis and Resolution (CAR)

Configuration Management (CM)

Decision Analysis and Resolution (DAR)

Measurement and Analysis (MA)

Process and Product Quality Assurance (PPQA)

Summary

Software Process Improvement (SPI) methodology is defined as definitions of sequence of tasks, tools and techniques to be performed to plan and implement improvement activities.

SPI mainly consists of 4 cyclic steps:

Current Situation Evaluation	Improvement Planning
Improvement Implementation	Improvement Evaluation

CMM means Capability Maturity Model. It is a series standard to assess the software capability and maturity. It also provides the methodologies for software assessment.

Capability Maturity Model Integration (CMMI) is a successor of CMM and is a more evolved model that incorporates best components of individual disciplines of CMM like Software CMM, Systems Engineering CMM, People CMM, etc.

Keywords

SPI	CMM
CMMI	SIX SIGMA
SPICE	ISO
SDLC	BOOTSTRAP
QIP	

Self Assessment

1. What are the steps for Software process improvement?

A. Current Situation Evaluation
B. Improvement Planning
C. Improvement Implementation
D. Above all

2. Software process improvement methods are__

A. Inductive
B. Prescriptive
C. Both inductive and prescriptive
D. None of above

3. What are the advantages of Software process improvement?

A. Cost reduction
B. Competitive Edge
C. Standardization and Process consistency
D. All of above

4. What are the Demotivation factors for Software process improvement?

A. Budget Constraints
B. Time pressure
C. Inadequate metrics
D. All of above

5. CMMI and SPICE are part of__

A. Prescriptive
B. Inductive
C. Both inductive and prescriptive
D. None of above

6. Which is not Software Process Improvement Model?

A. SIX SIGMA
B. IBM 02
C. SPICE
D. TickIT Guide ISO 9001

7. What are the areas of interest for CMMI?

A. Product and service development
B. Service establishment, management, and delivery
C. Product and service acquisition
D. All of above

8. BOOTSTRAP is__

A. Asian method
B. European method
C. American method
D. All of above

9. What are the principal area of CMM?

A. Organization and resource management
B. Software engineering process and its management
C. Tools and technology
D. All of above

10. Which is not part of CMM Maturity Level?

A. Repeatable
B. Defined
C. Testing
D. Initial Managed

11. CMMI V2.0 Key Improvements are__

A. Improve Business Performance
B. Build Agile Resiliency and Scale
C. Build Agile Resiliency and Scale
D. Above all

12. Which is not types of CMMI Appraisals?

A. Benchmark Appraisal
B. System design appraisal
C. Sustainment Appraisal
D. Action Plan Reappraisal

13. What are the CMMI Maturity Levels?

A. Repeatable
B. Defined
C. Initial Managed
D. All of above

14. CMMI model components are__

A. Process areas
B. Goals
C. Practices
D. All of above

15. Process Management and Project Management are part of__

A. Practices

B. Goals
C. CMMI Process Areas
D. None of above

Answers for Self Assessment

1.	D	2.	B	3.	D	4.	D	5.	A
6.	B	7.	D	8.	B	9.	D	10.	C
11.	D	12.	B	13.	D	14.	D	15.	C

Review Questions

1. What Is Maturity Level?
2. Define Is Software Process?
3. Differentiate between CMM and CMMI.
4. What is significance of Software Process Improvement?
5. Discuss Key Process Areas.
6. What are the CMMI model components?
7. Differentiate between SIX SIGMA and SPICE.
8. What is the role of ISO, give a suitable example.

Further Readings

O'Regan, Introduction to Software Process Improvement, springer

ShukorSanimMohdFauzi,NuraminahRamli, Software Process Improvement and Management: Approaches and Tools for Practical Development, igi-global

William A. Florac, Measuring the Software Process: Statistical Process Control for Software Process Improvement, Pearson education

Conradi, R., Dybå, T., Sjøberg, D.I.K., Ulsund, T., Software Process Improvement, Springer

Web Links

https://melsatar.blog/2018/06/26/the-software-process-improvement-spi-reward-or-risk/

https://www.geeksforgeeks.org/software-process-customization-and-improvement/

https://resources.sei.cmu.edu/

https://link.springer.com/chapter/10.1007/978-3-642-22206-1_9

https://www.cio.com/article/2437864/process-improvement-capability-maturity-model-integration-cmmi-definition-and-solutions.html

https://www.tutorialspoint.com/cmmi/cmmi_overview.htm

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in