By Shubh Thakare +919595939264

Unit 3. Object-Oriented Design

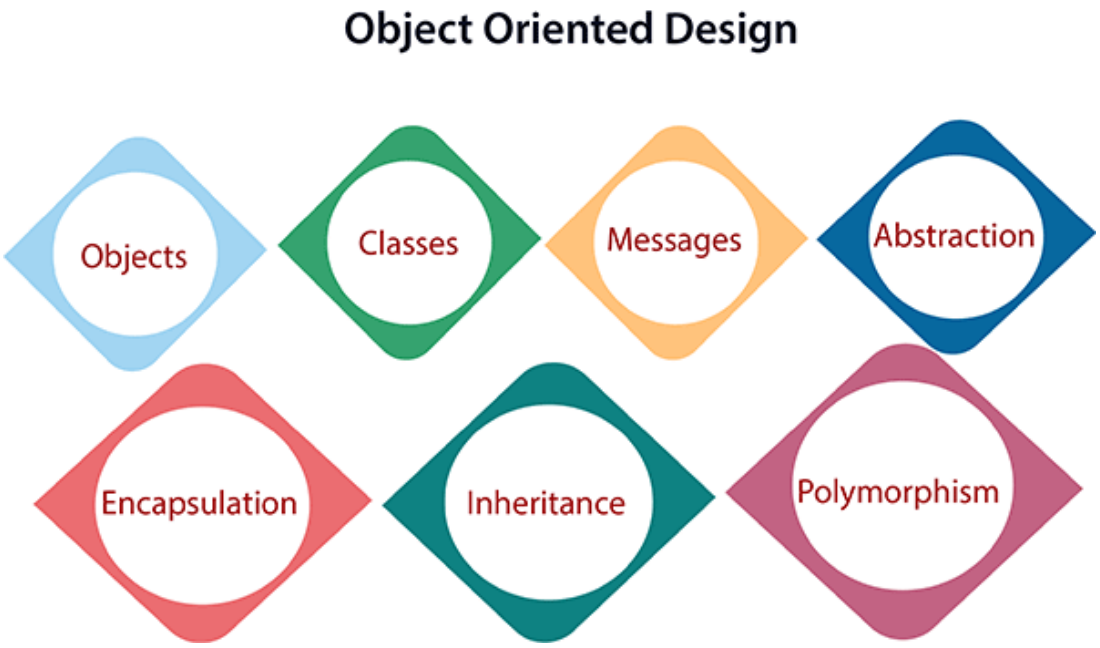
Object-Oriented Design ::

1.Object-Oriented Design :

Object-Oriented Design

In the object-oriented design method, 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. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

The different terms related to object design are:



1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

3. **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.
4. **Abstraction:** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

2.An Object-Oriented design process

Introduction to Object Oriented Design?

Software development consists of several important activities, which are compiled and attached together to create a software that has impeccable qualities, features and functionality. It is these components of the software that decide its scalability, performance, reliability, security, and more, as well as ensures that the software or application is developed as per the requirements of the client. However, to initiate and implement such activities, software engineers are required to prepare a proper design for the software, which can guide them during the process of software development.

What is Object Oriented Design?

Object Oriented Design (OOD) serves as part of the object oriented programming (OOP) process of lifestyle. It is mainly the process of using an object methodology to design a computing system or application. This technique enables the implementation of a software based on the concepts of objects. Additionally, it is a concept that forces programmers to plan out their code in order to have a better flowing program.

The other characteristics of Object Oriented Design are as follow:

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

Process of Object Oriented Design:

Understanding the process of any type of software related activity simplifies its development for the software developer, programmer and tester. Whether you are executing functional testing, or making a test report, each and every action has a process that needs to be followed by the members of the team. Similarly, Object Oriented Design (OOD) too has a defined process, which if not followed rigorously, can affect the performance as well as the quality of the software. Therefore, to assist the team of software developers and programmers, here is the process of Object Oriented Design (OOD):

1. To design classes and their attributes, methods, associations, structure, and even protocol, design axiom is applied.
    - The static UML class diagram is redefined and completed by adding details.
    - Attributes are refined.
    - Protocols and methods are designed by utilizing a UML activity diagram to represent the methods algorithm.
    - If required, redefine associations between classes, and refine class hierarchy and design with inheritance.

- Iterate and refine again.

2. Design the access layer.
    - Create mirror classes i.e., for every business class identified and created, create one access class.
3. Identify access layer class relationship.
4. Simplify classes and their relationships. The main objective here is to eliminate redundant classes and structures.
    - **Redundant Classes:** Programmers should remember to not put two classes that perform similar translate requests and translate results activities. They should simply select one and eliminate the other.
    - **Method Classes:** Revisit the classes that consist of only one or two methods, to see if they can be eliminated or combined with the existing classes.
5. Iterate and refine again.
6. Design the view layer classes.
    - Design the macro level user interface, while identifying the view layer objects.
    - Design the micro level user interface.
    - Test usability and user satisfaction.
    - Iterate and refine.
7. At the end of the process, iterate the whole design. Re-apply the design axioms, and if required repeat the preceding steps again.

Concepts of Object Oriented Design:

In Object Oriented Design (OOD), the technology independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and the interfaces are designed, which results in a model for the solution domain. In short, a detailed description is constructed to specify how the system is to be built on concrete technologies. Moreover, Object Oriented Design (OOD) follows some concepts to achieve these goals, each of which has a specific role and carries a lot of importance. These concepts are defined in detail below:

1. **Encapsulation:** This is a tight coupling or association of data structure with the methods or functions that act on the data. This is basically known as a class, or object (object is often the implementation of a class).

2. **Data Protection:** The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as private or protected to the owning class.

3. **Inheritance:** This is the ability of a class to extend or override the functionality of another class. This so called child class has a whole section that is the parent class and then it has its own set of functions and data.

4. **Interface:** A definition of functions or methods, and their signature that are available for use as well as to manipulate a given instance of an object.

5. **Polymorphism:** This is the ability to define different functions or classes as having the same name, but taking different data type.

Advantages of Object Oriented Design:

The discussion above has elaborated on several advantages of Object Oriented Design (OOD). From enabling the implementation of a software based on the concepts of objects and deleting the shared data areas to distributing and executing the object sequentially or in parallel, the benefits of this approach of software design are numerous. Hence, provided here some of the other advantages of using Object Oriented Design (OOD).

- 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.

3.Design evolution.

Design evolution Design evolution takes place over many years. A product is launched in the market and immediately people try to improve it. There are hundreds of examples of design evolution. The bicycle started its life as a wooden contraption that was pushed along by the rider's feet. There were several evolutionary designs proposed such as the “Penny Farthing,” but the formula of the modern bicycle emerged only when the pedal crank was connected to the drive axle by a roller chain. Modern evolution has kept the general format but improved the materials, brakes, suspension, and tires and reduced the overall weight.

Design concepts

The set of fundamental software design concepts are as follows:

1. Abstraction

- A solution is stated in large terms using the language of the problem environment at the highest level abstraction.
- The lower level of abstraction provides a more detail description of the solution.
- A sequence of instruction that contain a specific and limited function refers in a procedural abstraction.
- A collection of data that describes a data object is a data abstraction.

2. Architecture

- The complete structure of the software is known as software architecture.
- Structure provides conceptual integrity for a system in a number of ways.
- The architecture is the structure of program modules where they interact with each other in a specialized way.
- The components use the structure of data.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework.

3. Patterns

A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

4. Modularity

- A software is separately divided into name and addressable components. Sometime they are called as modules which integrate to satisfy the problem requirements.
- Modularity is the single attribute of a software that permits a program to be managed easily.

5. Information hiding

Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.

6. Functional independence

- The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.
- The functional independence is accessed using two criteria i.e Cohesion and coupling.

Cohesion

- Cohesion is an extension of the information hiding concept.
- A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

Coupling

Coupling is an indication of interconnection between modules in a structure of software.

7. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.

8. Refactoring

- It is a reorganization technique which simplifies the design of components without changing its function behaviour.
- Refactoring is the process of changing the software system in a way that it does not change the external behaviour of the code still improves its internal structure.

9. Design classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user visible.

Evolutionary Design

Evolutionary Design is about using all these tools, patterns, practices, habits and knowledge to be able to create a system that is adapted to the current need, but envisions the future developments. We don't develop according to a plan, like in waterfall, but we develop according to a goal: adding features to serve the users and the client.

By Shubh Thakare +919595939264

Performing User interface design :

1.Golden rules

The Golden Rules of User Interface Design

Place the User in Control

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**
  - The user shall be able to enter and exit a mode with little or no effort
- **Provide for flexible interaction**
  - The user shall be able to perform the same action via keyboard commands, mouse movement, or voice recognition
- **Allow user interaction to be interruptible and "undo" able**
  - The user shall be able to easily interrupt a sequence of actions to do something else (without losing the work that has been done so far)
  - The user shall be able to "undo" any action

Place the User in Control (continued)

- **Hide technical internals from the casual user**
  - The user shall not be required to directly use operating system, file management, networking. etc., commands to perform any actions. Instead, these operations shall be hidden from the user and performed "behind the scenes"
- **Design for direct interaction with objects that appear on the screen**
  - The user shall be able to **manipulate objects on the screen** in a manner similar to what would occur if the object were a physical thing (e.g., stretch a rectangle, press a button, move a slider)

Reduce the User's Memory Load

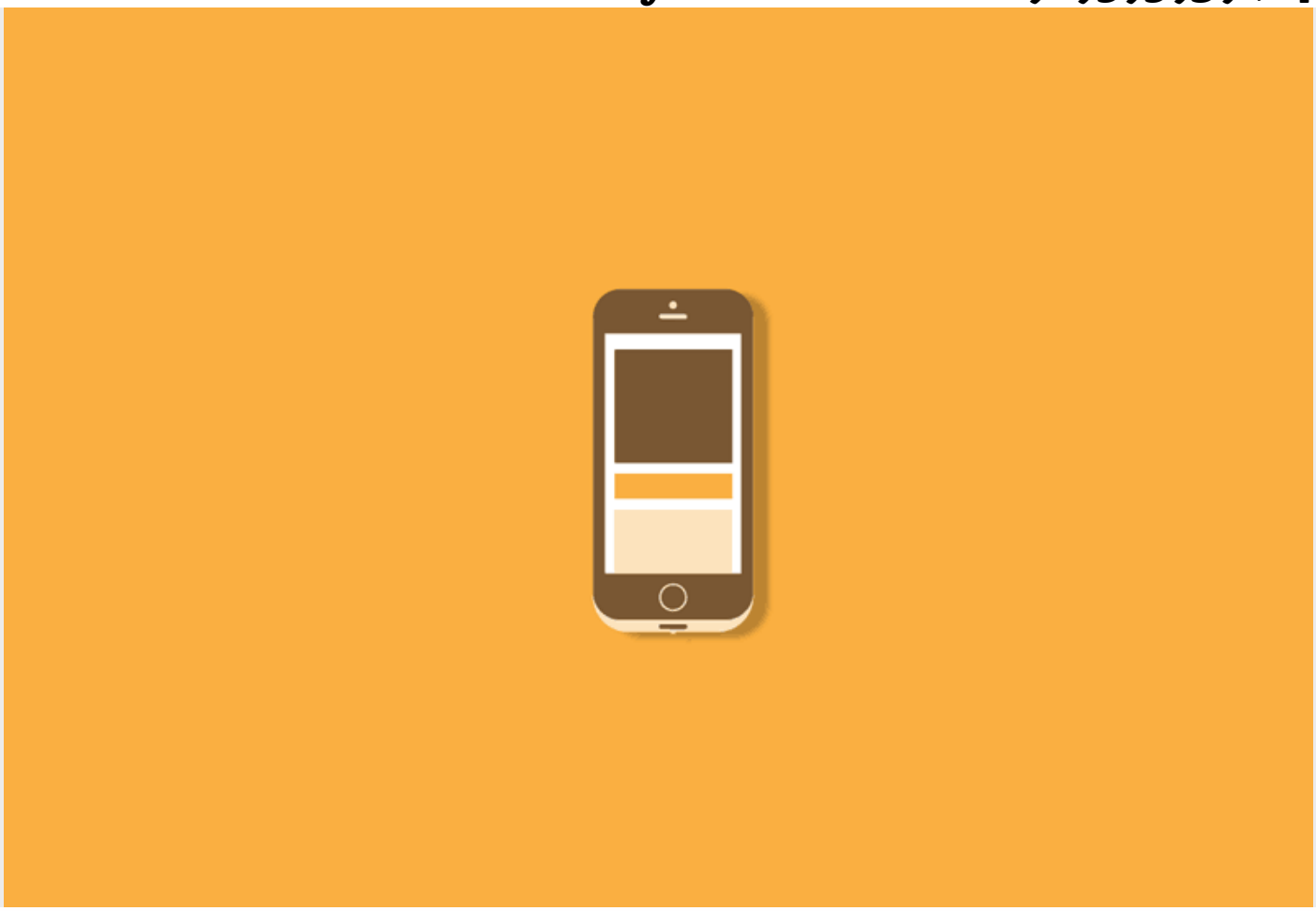
- **Reduce demand on short-term memory**
  - The interface shall reduce the user's requirement to remember past actions and results by providing visual cues of such actions
- **Define shortcuts that are intuitive**
  - The user shall be provided mnemonics (i.e., control or alt combinations) that tie easily to the action in a way that is easy to remember such as the first letter

Reduce the User's Memory Load (continued)

- **The visual layout of the interface should be based on a real world metaphor**
  - The screen layout of the user interface shall contain well-understood visuals that the user can relate to real-world actions
- **Disclose information in a progressive fashion**
  - When interacting with a task, an object or some behavior, the interface shall be organized hierarchically by moving the user progressively in a step-wise fashion from an abstract concept to a concrete action (e.g., text format options → format dialog box)

The more a user has to remember, the more error-prone interaction with the system will be

By Shubh Thakare +919595939264



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.

8 Golden Rules of User Interface Design is a "**rule**" that needs to be considered in making interface design. This Golden Rule was proposed by Ben Schneiderman on August 21, 1947, in his book entitled Designing the User Interface: Strategies for Effective Human-Computer Interaction.

1. Strive for consistency

Designing "**consistent interfaces**" means using the same design patterns and the same sequences of actions for similar situations. This includes, but isn't limited to, the right use of color, typography, and terminology in prompt screens, commands, and menus throughout your user journey.

Remember: a consistent interface will allow your users to complete their tasks and goals much more easily.

2. Enable Frequent Users to Use Shortcuts

Speaking of using UI rules as shortcuts, your users will benefit from shortcuts as well, especially if they need to complete the same tasks often.

Expert users might find the following features helpful:

- Abbreviations
- Function keys
- Hidden commands
- Macro facilities

Read more: The Fundamentals of UX and UI Design

3. Offer informative feedback

For every user action, there should be interface feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial. Visual presentation of the

objects of interest provides a convenient environment for showing changes explicitly.

4. Design dialogs to yield closure

Sequences of actions should be organized into groups with a beginning, middle, and end. Informative feedback at the completion of a group of actions gives users the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans from their minds, and an indicator to prepare for the next group of actions. For example, e-commerce websites move users from selecting products to the checkout, ending with a clear confirmation page that completes the transaction.

5. Offer Simple Error Handling

A good interface should be designed to avoid errors as much as possible. But when errors do happen, your system needs to make it easy for the user to understand the issue and know how to solve it. Simple ways to handle errors include displaying clear error notifications along with descriptive hints to solve the problem.

As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer a simple, comprehensive mechanism for handling the error.

Error Prevention:

- Error prevention over error correction
- Automatic detection of errors

- Clear error notifications
- Hints for solving the problem

6. Permit easy reversal of actions UNDO

This feature relieves anxiety since users know that errors can be undone, and encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data-entry task, or a complete group of actions, such as entry of a name-address block.

7. Keep users in control

Experienced users strongly desire the sense that they are in charge of the interface and that the interface responds to their actions. They don't want surprises or changes in familiar behavior, and they are annoyed by tedious data-entry sequences, difficulty in obtaining necessary information, and inability to produce their desired result.

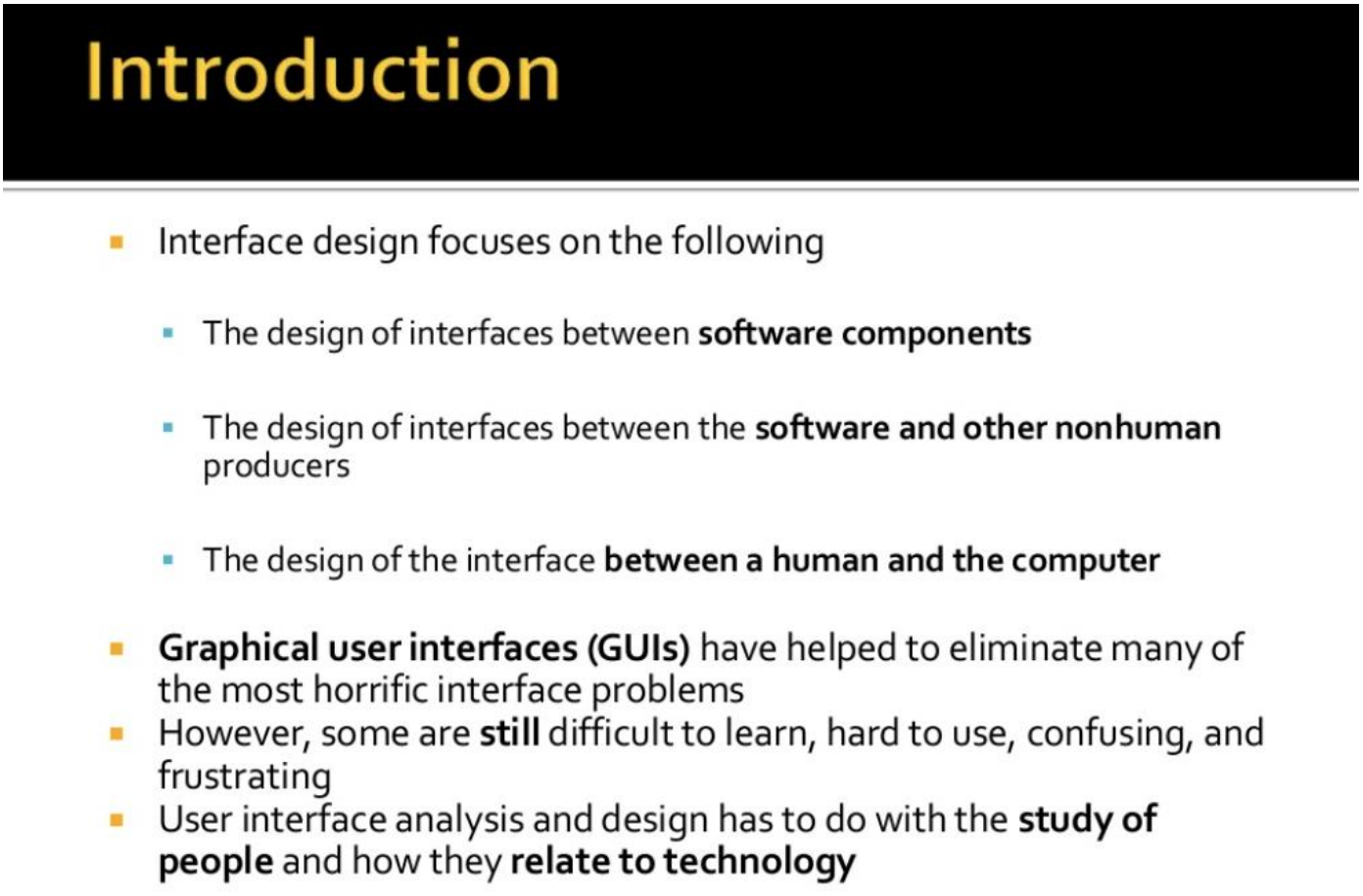
8. Reduce short-term memory load

Humans' limited capacity for information processing in short-term memory (the rule of thumb is that people can remember "seven plus or minus two chunks" of information) requires that designers avoid interfaces in which users must remember information from one display and then use that information on another display. It means that cellphones should not require reentry of phone numbers, website locations should remain visible, and lengthy forms should be compacted to fit a single display.

These underlying principles must be interpreted, refined, and extended for each environment. They have their limitations, but they provide a good

starting point for mobile, desktop, and web designers. The principles presented in the ensuing sections focus on increasing users' productivity by providing simplified data-entry procedures, comprehensible displays, and rapid informative feedback to increase feelings of competence, mastery, and control over the system.The goal for UI designers is to produce user-friendly interfaces: interfaces that encourage exploration without fear of negative consequences. Without any doubt interfaces of the future will be more intuitive, enticing, predictable, and forgiving, but most principles of UI design listed in this article will surely be applicable. 8 These Golden Rules from Schneider ensure a good design but can ease the burden experienced by users in using applications that we make.

2.User interface analysis and design



A Spiral Process

- User interface development follows a spiral process
  - **Interface analysis (user, task, and environment analysis)**
    - Focuses on the profile of the users **who will interact with the system**
    - Concentrates on **users, tasks, content and work environment**
    - Delineates the human- and computer-oriented tasks that are required to achieve system function
  - **Interface design**
    - Defines a set of interface **objects** and **actions** (and their **screen representations**) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system
  - **Interface construction**
    - **Begins with a prototype** that enables usage scenarios to be evaluated
    - Continues with development tools to complete the construction
  - **Interface validation,** focuses on
    - The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements
    - The degree to which the interface is easy to use and easy to learn
    - The **users' acceptance** of the interface as a useful tool in their work

5

3.interface analysis

Elements of the User Interface

- To perform user interface analysis, the practitioner needs to study and understand four elements
  - The **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 **work environment** in which these tasks will be conducted

By Shubh Thakare +919595939264

User Analysis

- **The analyst strives to get the end user's mental model and the design model to converge by understanding**
  - The users themselves
  - How these people use the system
- **Information can be obtained from**
  - **User interviews** with the end users
  - **Sales input** from the sales people who interact with customers and users on a regular basis
  - **Marketing input** based on a market analysis to understand how different population segments might use the software
  - **Support input** from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use
- A set of questions should be answered during user analysis (see next slide)

Task Analysis and Modeling

- **Task analysis strives to know and understand**
  - The work the user **performs in specific circumstances**
  - **The tasks and subtasks** that will be performed as the user does the work
  - The specific problem domain objects that the user manipulates as work is performed
  - The sequence of **work tasks** (i.e., the workflow)
  - The **hierarchy of tasks**
- **Use cases**
  - Show how an end user performs some specific work-related task
  - Enable the software engineer to extract tasks, objects, and **overall workflow** of the interaction
  - Helps the software engineer to identify **additional helpful features**

By Shubh Thakare +919595939264

Content Analysis

- The display content may range from character-based reports, to graphical displays, to multimedia information
- Display content may be
  - Generated by components in other parts of the application
  - Acquired from data stored in a database that is accessible from the application
- The format of the content (as it is displayed by the interface) needs to be considered
- A set of questions should be answered during content analysis (see next slide)

4.interface design steps

Introduction

- User interface design is an **iterative process**, where each iteration elaborate and **refines** the information developed in the preceding step
- General steps for user interface design
  1) Using information developed during user interface analysis, define user interface **objects and actions (operations)**
  2) Define **events (user actions)** that will cause the state of the user interface to change; model this behavior
  3) Depict each interface state as it will actually look to the **end user**
  4) Indicate how the **user interprets** the state of the system from information provided through the interface
- During all of these steps, the designer must
  - **Always follow the three golden rules of user interfaces**
  - Model how the interface will be implemented
  - Consider the **computing environment** (e.g., display technology, operating system, development tools) that will be used

By Shubh Thakare +919595939264

5.Design evaluation

Design and Prototype Evaluation

- Before prototyping occurs, a number of evaluation criteria can be applied during design reviews to the design model itself
  - **The amount of learning required by the users**
  - **The interaction time and overall efficiency**
    - Derived from the number of user tasks specified and the average number of actions per task
  - **The memory load on users**
    - Derived from the number of actions, tasks, and system states
  - **The complexity of the interface and the degree to which it will be accepted by the user**
    - Derived from the interface style, help facilities, and error handling procedures

Design and Prototype Evaluation (continued)

- Prototype evaluation can range from an informal test drive to a formally designed study using statistical methods and questionnaires
- The prototype evaluation cycle consists of prototype creation followed by user evaluation and back to prototype modification until all user issues are resolved
- **The prototype is evaluated for**
  - Satisfaction of user requirements
  - **Conformance to the three golden rules of user interface design**

By Shubh Thakare +919595939264

Testing Strategies :

Software Testing

Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Why Software Testing is Important?

Software Testing is Important because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

What are the benefits of Software Testing?

Here are the benefits of using software testing:

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

1.A strategic approach to software testing

Software Testing Strategies and Approaches

By Shubh Thakare +919595939264

When venturing into a software testing project, there are two major categories that need to be considered: **strategy** and **investment**. Needless to say, strategy comes first. Without knowing what techniques and tools are needed to comprehensively test a website or app, it is not possible to determine how much investment the project will require.

1. Static Testing Strategy

A static test evaluates the quality of a system without actually running the system. While that may seem impossible, it can be accomplished in a few ways.

The static test looks at portions of or elements related to the system in order to detect problems as early as possible. For example, developers review their code after writing and before pushing it. This is called desk-checking, a form of static testing. Another example of a static test would be a review meeting held for the purpose of evaluating requirements, design, and code.

2. Structural Testing Strategy

While static tests are quite useful, they are not adequate. The software needs to be operated on real devices, and the system has to be run in its entirety to find all bugs. Structural tests are among the most important of these tests.

Structural tests are designed on the basis of the software structure. They can also be called white-box tests because they are run by testers with thorough knowledge of the software as well as the devices and systems it is functioning on. Structural tests are most often run on individual components and interfaces in order to identify localized errors in data flows.

3. Behavioral Testing Strategy

Behavioral Testing focuses on how a system acts rather than the mechanism behind its functions. It focuses on workflows, configurations, performance, and all elements of the user journey. The point of these tests, often called “black box” tests, is to test a website or app from the perspective of an end-user.

Behavioral testing must cover multiple user profiles as well as usage scenarios. Most of these tests focus on fully integrated systems rather than individual components. This is because it is possible to effectively gauge system behavior from a user’s eyes, only after it has been assembled and integrated to a significant extent.

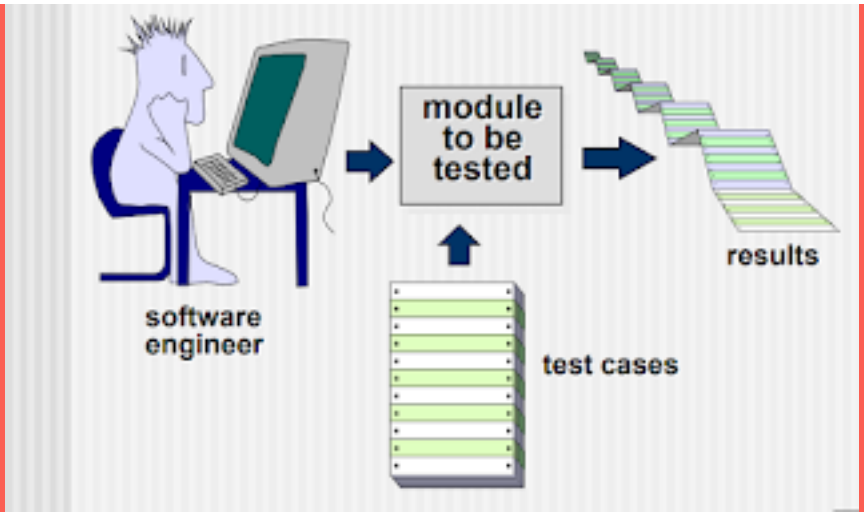
2.test strategies for conventional software

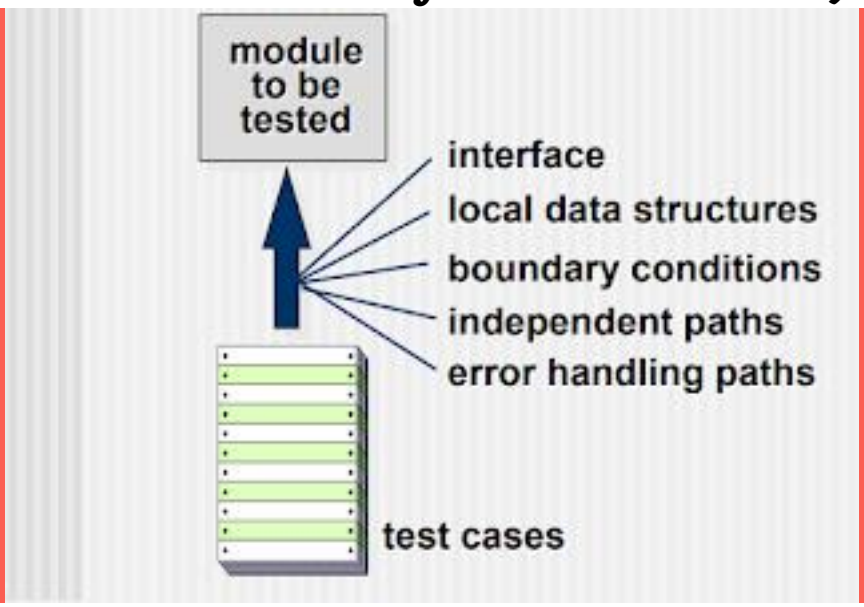
- There are many strategies that can be used to test software.
- At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
  - This approach simply does not work. It will result in buggy software.
- At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.

- This approach, although less appealing to many, can be very effective.
- A testing strategy that is chosen by most software teams falls between the two extremes.
- It takes an incremental view of testing,
  - Beginning with the testing of individual program units,
  - Moving to tests designed to facilitate the integration of the units,
  - Culminating with tests that exercise the constructed system.

Unit Test :

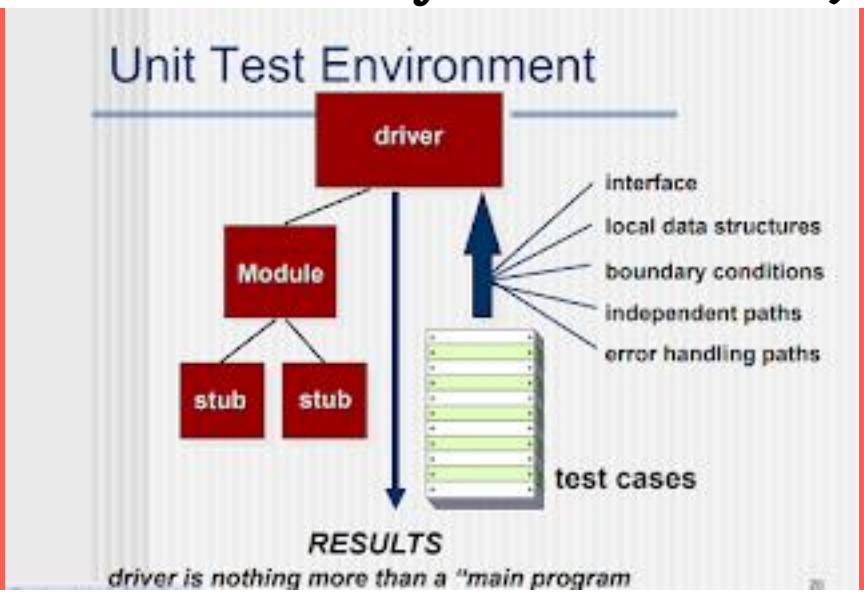
- **Unit testing** focuses verification effort on the smallest unit of software design—the software component or module.
- The unit test focuses on the internal processing logic and data structures within the boundaries of a component.
- This type of testing can be conducted in parallel for multiple components.





- Unit tests are illustrated schematically in previous Figure.
- **The module interface** is tested to ensure that information properly flows into and out of the program unit under test.
- **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- **Finally, all error-handling paths are tested**
- Good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur.
- *Unit testing is simplified when a component with high cohesion is designed.*

By Shubh Thakare +919595939264



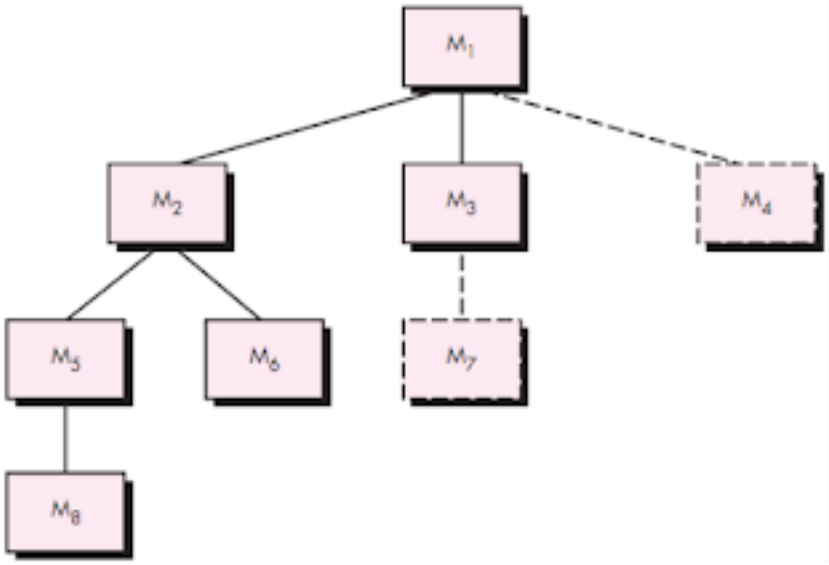
Integration Testing :

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
- **Different Integration Testing Strategies :**
  - Top-down testing
  - Bottom-up testing
  - Regression Testing
  - Smoke Testing

Top-down testing

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

By Shubh Thakare +919595939264



Integration Testing :

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 (discussed later in this section) 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.

Problem encountered during top-down integration testing.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise.

The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels.

Stubs replace low-level modules at the beginning of top-down testing; therefore, no sig-nificant data can flow upward in the program structure.

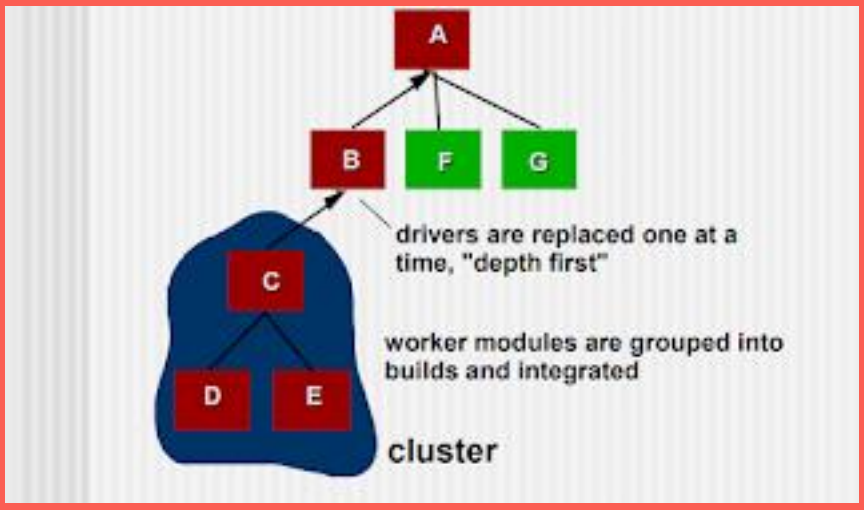
As a tester, you are left with three choices:

(1) delay many tests until stubs are replaced with actual modules,

(2) develop stubs that perform limited functions that simulate the actual module, or

(3) integrate the software from the bottom of the hierarchy upward.

Bottom-Up Integration



Bottom-up Integration Testing :

- Bottom-up integration testing, It 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, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.
- **A bottom-up integration strategy may be implemented with the following steps…**
  - 1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
  - 2. A driver (a control program for testing) is written to coordinate test case input and output.
  - 3. The cluster is tested.
  - 4. Drivers are removed and clusters are combined moving upward in the program structure.

Regression Testing :

- **Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.**
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.
- It is impractical and inefficient to reexecute every test for every program function once a change has occurred….
- **Regression testing is a type of software testing that seeks to uncover new software bugs, OR**
- **Regression testing is the process of testing, changes to computer programs to make sure that the older programming still works with the new changes. Here changes such as enhancements, patches or configuration changes, have been made to them.**

Smoke Testing :

- Smoke testing is an integration testing approach that is commonly used when product software is developed
- **Smoke testing is performed by developers before releasing the build to the testing team and after releasing the build to the testing team it is performed by testers whether to accept the build for further testing or not.**
- It is designed as a pacing (Speedy) mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.
- **Smoke Testing Example**
- For Example in a project there are five modules like login, view user, user detail page, new user creation, and task creation etc. So in this five modules first of all developer perform the smoke testing by executing all the major functionality of modules like user is able to login or not with valid login credentials and after login new user can created or not, and user that is created viewed or not. So it is obvious that this is the smoke testing always done by developing team before submitting (releasing) the build to the testing team.
- Now once the build is released to the testing team than the testing team has to check whether to accept or reject the build by testing the major functionality of the build. So this is the smoke test done by testers
- **Smoke testing provides a number of benefits when it is applied on complex, time critical software projects.**
- I**ntegration risk is minimized.** Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early.
- **The quality of the end product is improved**. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- **Error diagnosis and correction are simplified.**
- **Progress is easier to assess**

By Shubh Thakare +919595939264

3.Black-Box and White-Box testing

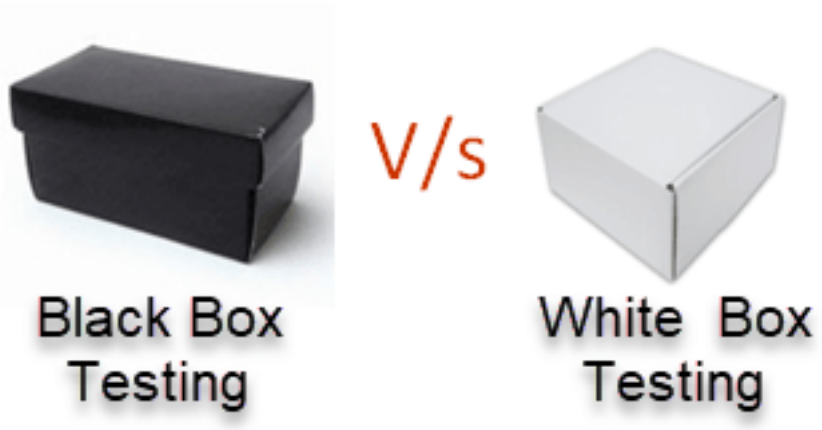
What is Black Box testing?

In Black-box testing, a tester doesn't have any information about the internal working of the software system. Black box testing is a high level of testing that focuses on the behavior of the software. It involves testing from an external or end-user perspective. Black box testing can be applied to virtually every level of software testing: unit, integration, system, and acceptance.

What is White Box testing?

White-box testing is a testing technique which checks the internal functioning of the system. In this method, testing is based on coverage of code statements, branches, paths or conditions. White-Box testing is considered as low-level testing. It is also called glass box, transparent box, clear box or code base testing. The white-box Testing method assumes that the path of the logic in a unit or program is known.

Difference between Black Box testing and White Box testing



Parameter	Black Box testing	White Box testing
Definition	It is a testing approach which is used to test the software without the knowledge of the internal structure of program or application.	It is a testing approach in which internal structure is known to the tester.
Alias	It also knowns as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing,

Parameter	Black Box testing	White Box testing
		code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.
Implementation knowledge	Implementation knowledge is not requiring doing Black Box testing.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	White Box testing is easy to automate.
Objective	The main objective of this testing is to check what functionality of the system under test.	The main objective of White Box testing is done to check the quality of the code.
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.
Granularity	Granularity is low.	Granularity is high.
Testing method	It is based on trial and error method.	Data domain and internal boundaries can be tested.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Algorithm test	Not the best method for algorithm testing.	Best suited for algorithm testing.
Code Access	Code access is not required for Black Box Testing.	White box testing requires code access. Thereby, the code could be stolen if testing is outsourced.
Benefit	Well suited and efficient for large code segments.	It allows removing the extra lines of code, which can bring in hidden defects.

Parameter	Black Box testing	White Box testing
Skill level	Low skilled testers can test the application with no knowledge of the implementation of programming language or operating system.	Need an expert tester with vast experience to perform white box testing.
Techniques	Equivalence partitioning is Black box testing technique is used for Blackbox testing. Equivalence partitioning divides input values into valid and invalid partitions and selecting corresponding values from each partition of the test data. Boundary value analysis checks boundaries for input values.	Statement Coverage, Branch coverage, and Path coverage are White Box testing technique. Statement Coverage validates whether every line of the code is executed at least once. Branch coverage validates whether each branch is executed at least once Path coverage method tests all the paths of the program.
Drawbacks	Update to automation test script is essential if you to modify application frequently.	Automated test cases can become useless if the code base is rapidly changing.

4.Validation testing

Software Testing - Validation Testing

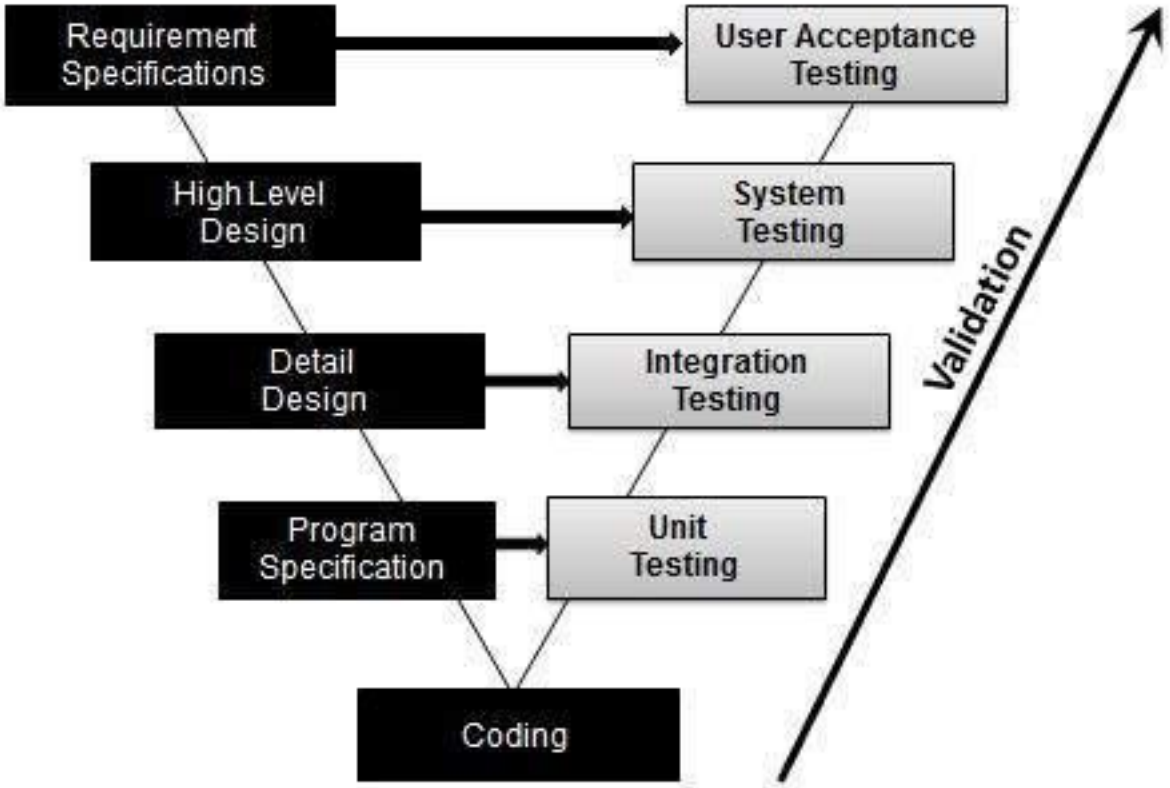
The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

It answers to the question, Are we building the right product?

Validation Testing - Workflow:

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.



Activities:

- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing

5.System testing

What is System Testing?

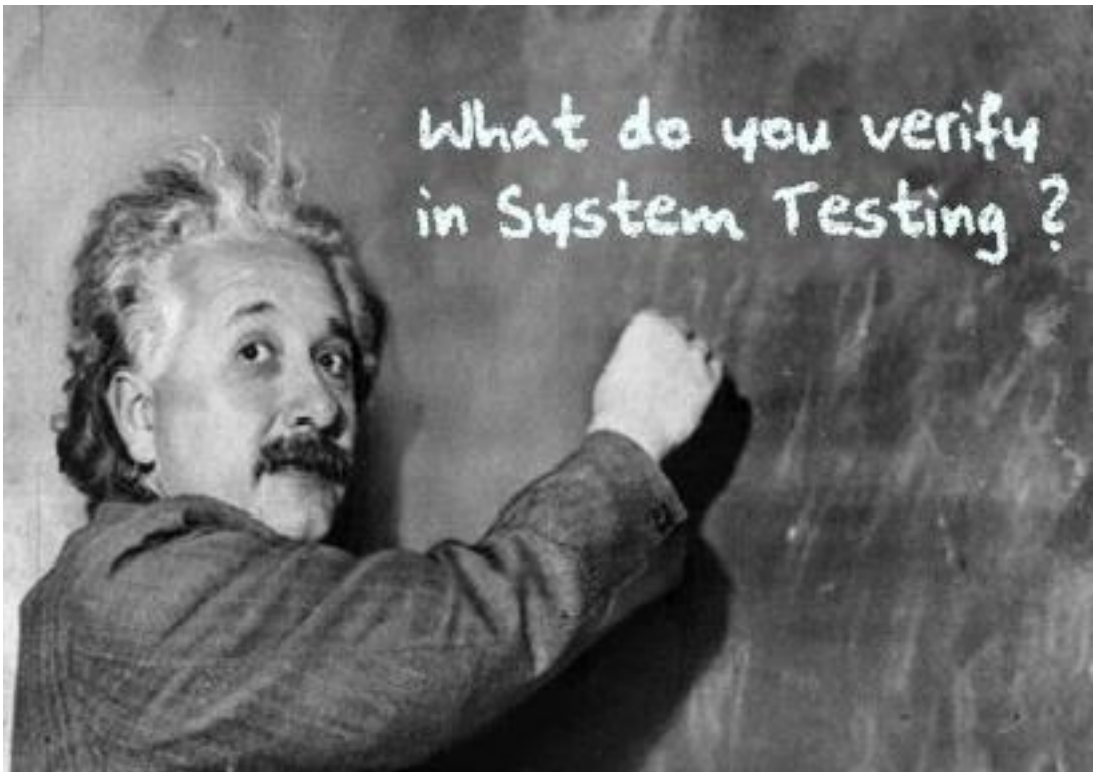
SYSTEM TESTING is a level of testing that validates the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system. Ultimately, the software is interfaced with other software/hardware systems. System Testing is actually a series of different tests whose sole purpose is to exercise the full computer-based system.

.

By Shubh Thakare +919595939264

What do you verify in System Testing?

System Testing involves testing the software code for following

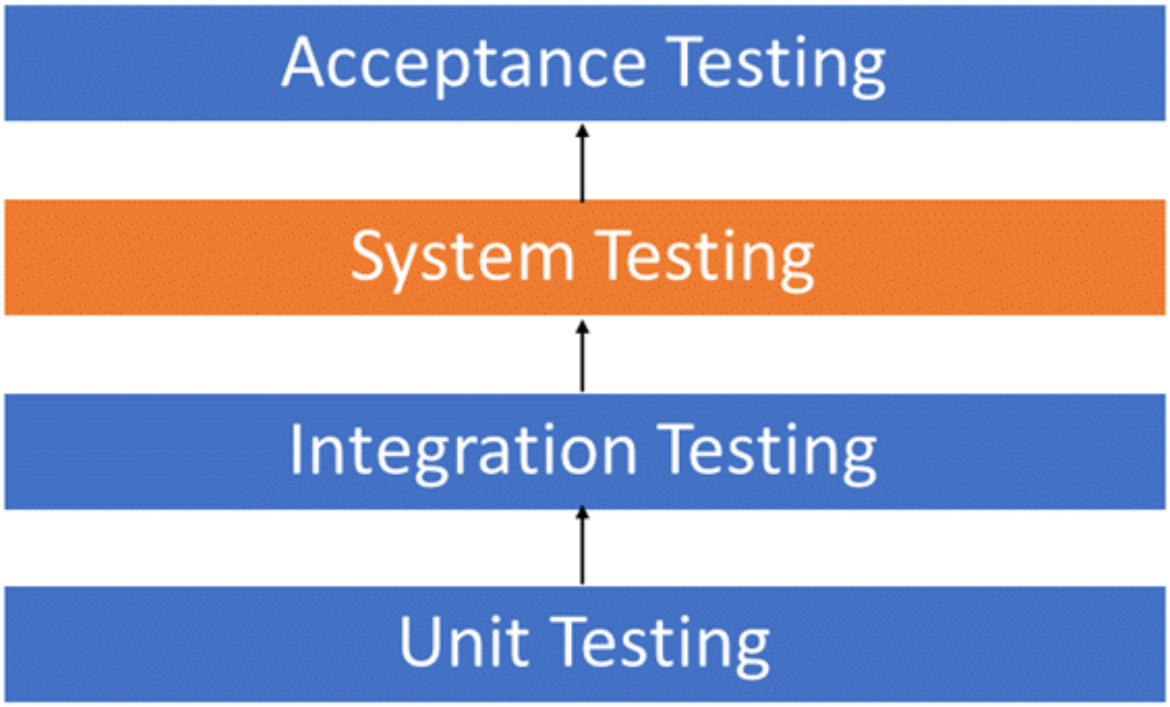


- Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. This is also called End to End testing scenario.
- Verify thorough testing of every input in the application to check for desired outputs.
- Testing of the user's experience with the application.

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

Software Testing Hierarchy

By Shubh Thakare +919595939264



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

- Unit testing performed on each module or block of code during development. Unit Testing is normally done by the programmer who writes the code.

- Integration testing done before, during and after integration of a new module into the main software package. This involves testing of each individual code module. One piece of software can contain several modules which are often created by several different programmers. It is crucial to test each module's effect on the entire program model.

- System testing done by a professional testing agent on the completed software product before it is introduced to the market.

- Acceptance testing - beta testing of the product done by the actual end users.

By Shubh Thakare +919595939264

Different Types of System Testing

There are more than 50 types of System Testing. For an exhaustive list of software testing

Below we have listed types of system testing a large software development company would typically use

1. 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

2. Load Testing- is necessary to know that a software solution will perform under real-life loads.

3. 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.

4. Recovery testing - is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

5. Migration testing- is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

6. Functional Testing - Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.

7. Hardware/Software Testing - IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

What Types of System Testing Should Testers Use?

There are over 50 different types of system testing. The specific types used by a tester depend on several variables. Those variables include:

- Who the tester works for - This is a major factor in determining the types of system testing a tester will use. Methods used by large companies are different than that used by medium and small companies.
- Time available for testing - Ultimately, all 50 testing types could be used. Time is often what limits us to using only the types that are most relevant for the software project.
- Resources available to the tester - Of course some testers will not have the necessary resources to conduct a testing type. For example, if you are a tester working for a large software development firm, you are likely to have expensive automated testing software not available to others.
- Software Tester's Education- There is a certain learning curve for each type of software testing available. To use some of the software involved, a tester has to learn how to use it.
- Testing Budget - Money becomes a factor not just for smaller companies and individual software developers but large companies as well.

6.the art of Debugging

It is a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected. Debugging is harder for complex systems in particular when various subsystems are tightly coupled as changes in one system or interface may cause bugs to emerge in another.

Debugging is a developer activity and effective debugging is very important before testing begins to increase the quality of the system. Debugging will not give confidence that the system meets its requirements completely but testing gives confidence.

Why do we need Debugging?

The process of debugging begins as soon as the **code** of the **software** is written. Then, it continues in successive stages as code is combined with other units of programming to form a software product. Debugging has many benefits such as:

- It **reports** an **error condition immediately**. This allows earlier detection of an error and makes the process of software development stress-free and unproblematic.

- It also provides **maximum useful information** of data structures and allows **easy interpretation.**
- Debugging assists the developer in **reducing useless** and **distracting information**.
- Through debugging the developer can **avoid complex one-use testing code** to save time and energy in software development.

Steps involved in Debugging

The different steps involved in the process of debugging are:

1. Identify the Error: A bad identification of an error can lead to wasted developing time. It is usual that production errors reported by users are hard to interpret and sometimes the information we receive is misleading. It is import to identify the actual error.

2. Find the Error Location: After identifying the error correctly, you need to go through the code to find the exact spot where the error is located. In this stage, you need to focus on finding the error instead of understanding it.

3. Analyze the Error: In the third step, you need to use a bottom-up approach from the error location and analyze the code. This helps you in understanding the error. Analyzing a bug has two main goals, such as checking around the error for other errors to be found, and to make sure about the risks of entering any collateral damage in the fix.

4. Prove the Analysis: Once you are done analyzing the original bug, you need to find a few more errors that may appear on the application. This step is about writing automated tests for these areas with the help of a test framework.

5. Cover Lateral Damage: In this stage, you need to create or gather all the unit tests for the code where you are going to make changes. Now, if you run these unit tests, they all should pass.

6. Fix & Validate: The final stage is the fix all the errors and run all the test scripts to check if they all pass.

Debugging Strategies

- It is important to **study** the **system** in depth in order to understand the system. It helps the debugger to construct different representations of systems that are to be debugged.
- **Backward analysis** of the problem traces the program backward from the location of failure message in order to identify the region of faulty code. You need to study the region of defect thoroughly to find the cause of defects.
- **Forward analysis** of the program involves tracking the program forward using breakpoints or print statements at different points in the program. It is important to focus on the region where the wrong outputs are obtained.
- You must use the **past experience** of the software to check for similar problems. The success of this approach depends on the expertise of the debugger.

Debugging Tools

Debugging tool is a computer program used to test and debug other programs. There are a lot of public domain software like **gdb** and **dbx** that you can use for debugging. Also, they offer console-based command-line interfaces. Some of the automated debugging tools include code-based tracers, profilers, interpreters, etc.

Here is a list of some of the widely used debuggers:

- **Radare2**
- **WinDbg**
- **Valgrind**

Product metrics :

Software Metrics

A software metric is a measure of software characteristics which are measurable or countable. Software metrics are valuable for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.

Within the software development process, many metrics are that are all connected. Software metrics are similar to the four functions of management: Planning, Organization, Control, or Improvement.

Classification of Software Metrics

Software metrics can be classified into two types as follows:

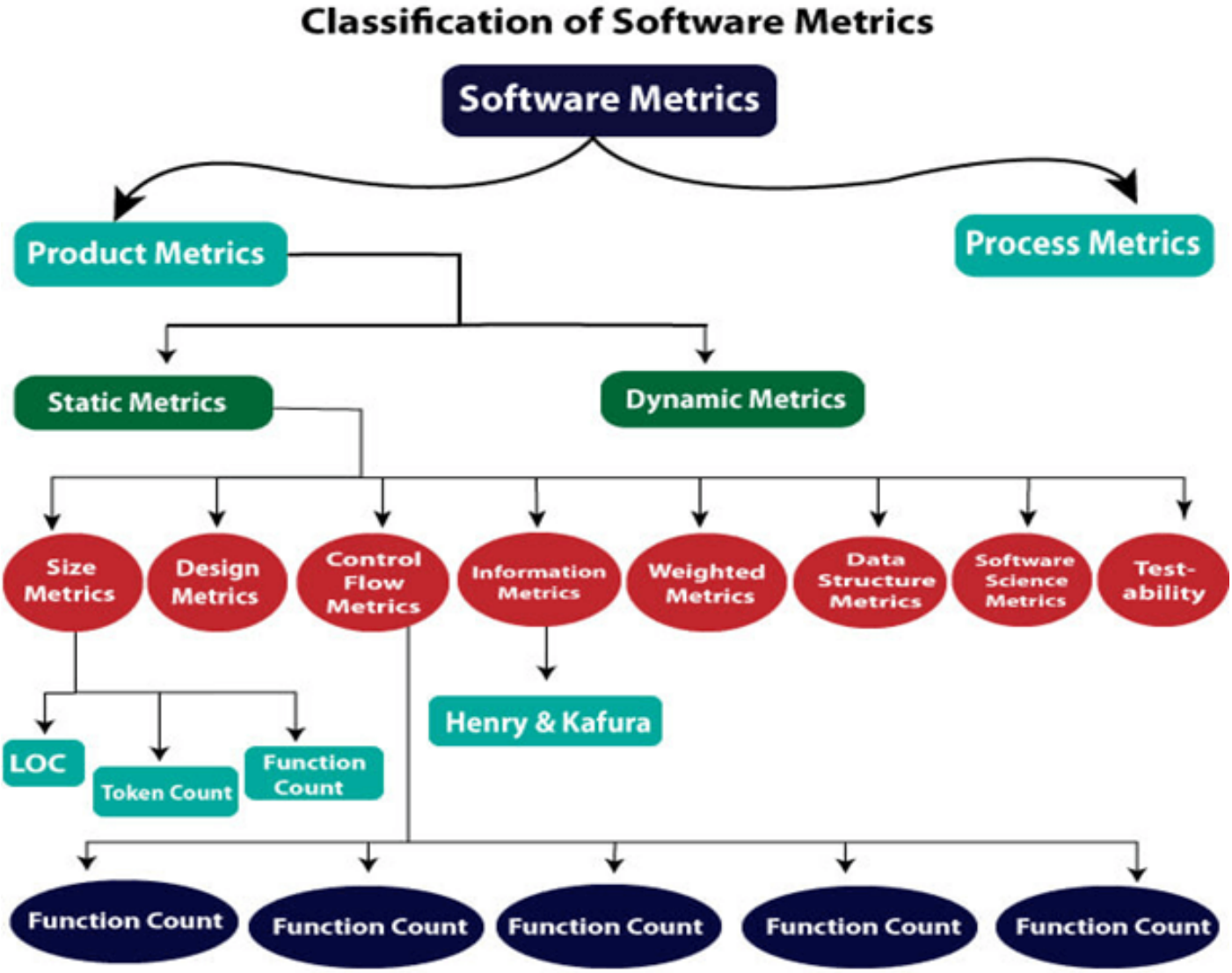
1. Product Metrics: These are the measures of various characteristics of the software product. The two important software characteristics are:

1. Size and complexity of software.
2. Quality and reliability of software.

These metrics can be computed for different stages of SDLC.

2. Process Metrics: These are the measures of various characteristics of the software development process. For example, the efficiency of fault detection. They are used to measure the characteristics of methods, techniques, and tools that are used for developing software.

By Shubh Thakare +919595939264



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. Note that as the project proceeds, the project manager will check its progress from time-to-time and will compare the effort, cost, and time with the original effort, cost and time. Also understand that these metrics are used to decrease the development costs, time efforts and risks. The project quality can also be improved. As quality improves, the number of errors and time, as well as cost required, is also reduced.

Advantage of Software Metrics

Comparative study of various design methodology of software systems.

For analysis, comparison, and critical study of different programming language concerning their characteristics.

In comparing and evaluating the capabilities and productivity of people involved in software development.

In the preparation of software quality specifications.

In the verification of compliance of software systems requirements and specifications.

In making inference about the effort to be put in the design and development of the software systems.

In getting an idea about the complexity of the code.

In taking decisions regarding further division of a complex module is to be done or not.

In guiding resource manager for their proper utilization.

In comparison and making design tradeoffs between software development and maintenance cost.

In providing feedback to software managers about the progress and quality during various phases of the software development life cycle.

In the allocation of testing resources for testing the code.

Disadvantage of Software Metrics

The application of software metrics is not always easy, and in some cases, it is difficult and costly.

The verification and justification of software metrics are based on historical/empirical data whose validity is difficult to verify.

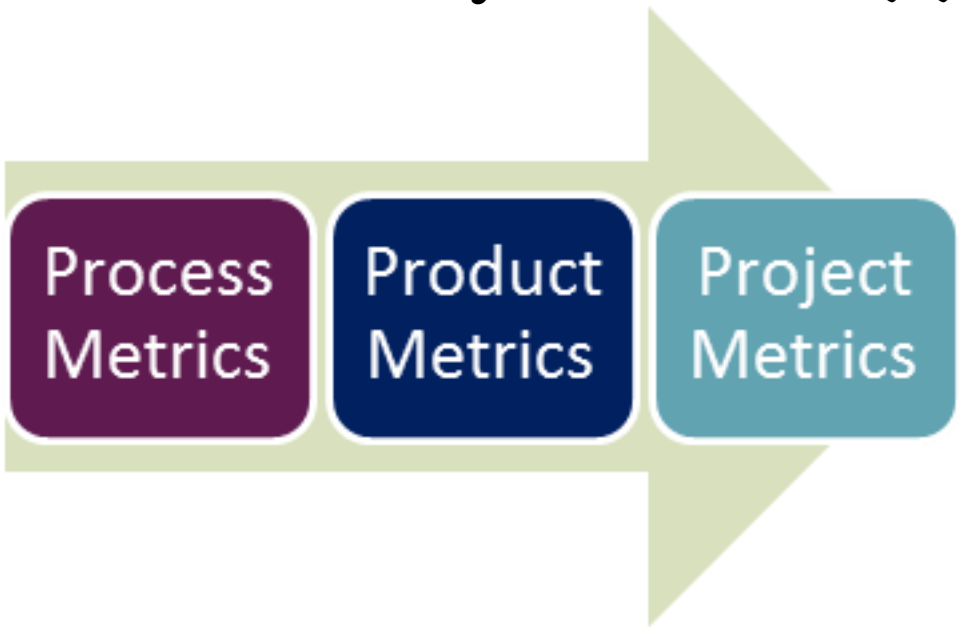
These are useful for managing software products but not for evaluating the performance of the technical staff.

The definition and derivation of Software metrics are usually based on assuming which are not standardized and may depend upon tools available and working environment.

Most of the predictive models rely on estimates of certain variables which are often not known precisely.

Types of Test Metrics

By Shubh Thakare +919595939264



- **Process Metrics:** It can be used to improve the process efficiency of the SDLC (Software Development Life Cycle)
- **Product Metrics:** It deals with the quality of the software product
- **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members

1.Software Quality

Software Quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software Quality Management System

A quality management system is the principal methods used by organizations to provide that the products they develop have the desired quality.

A quality system subsists of the following:

Managerial Structure and Individual Responsibilities: A quality system is the responsibility of the organization as a whole. However, every organization has a sever quality department to perform various quality system activities. The quality system of an arrangement should have the support of the top management. Without help for the quality system at a high level in a company, some members of staff will take the quality system seriously.

Quality System Activities: The quality system activities encompass the following:

Auditing of projects

Review of the quality system

Development of standards, methods, and guidelines, etc.

Production of documents for the top management summarizing the effectiveness of the quality system in the organization.

Evolution of Quality Management System

Quality systems have increasingly evolved over the last five decades. Before World War II, the usual function to produce quality products was to inspect the finished products to remove defective devices. Since that time, quality systems of organizations have undergone through four steps of evolution, as shown in the fig. The first product inspection task gave method to quality control (QC).

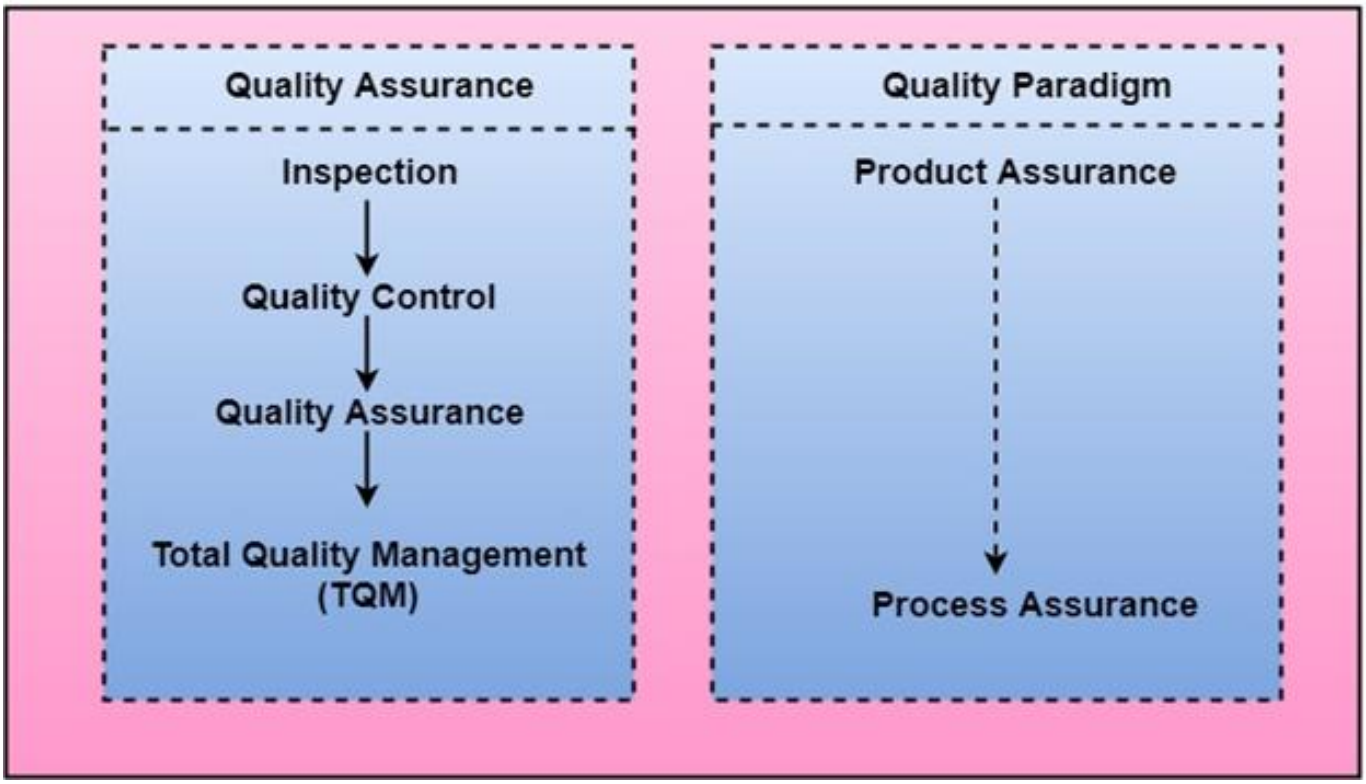
Quality control target not only on detecting the defective devices and removes them but also on determining the causes behind the defects. Thus, quality control aims at correcting the reasons for bugs and not just rejecting the products. The next breakthrough in quality methods was the development of quality assurance methods.

The primary premise of modern quality assurance is that if an organization's processes are proper and are followed rigorously, then the products are obligated to be of good quality. The new quality functions include guidance for recognizing, defining, analyzing, and improving the production process.

Total quality management (TQM) advocates that the procedure followed by an organization must be continuously improved through process measurements. TQM goes stages further than quality assurance and aims at frequently process improvement. TQM goes beyond documenting steps to optimizing them through a redesign. A term linked to TQM is Business Process Reengineering (BPR).

BPR aims at reengineering the method business is carried out in an organization. From the above conversation, it can be stated that over the years, the quality paradigm has changed from product assurance to process assurance, as shown in fig.

Evolution of quality system and corresponding shift in the quality paradigm



2.Metrics for Analysis Model

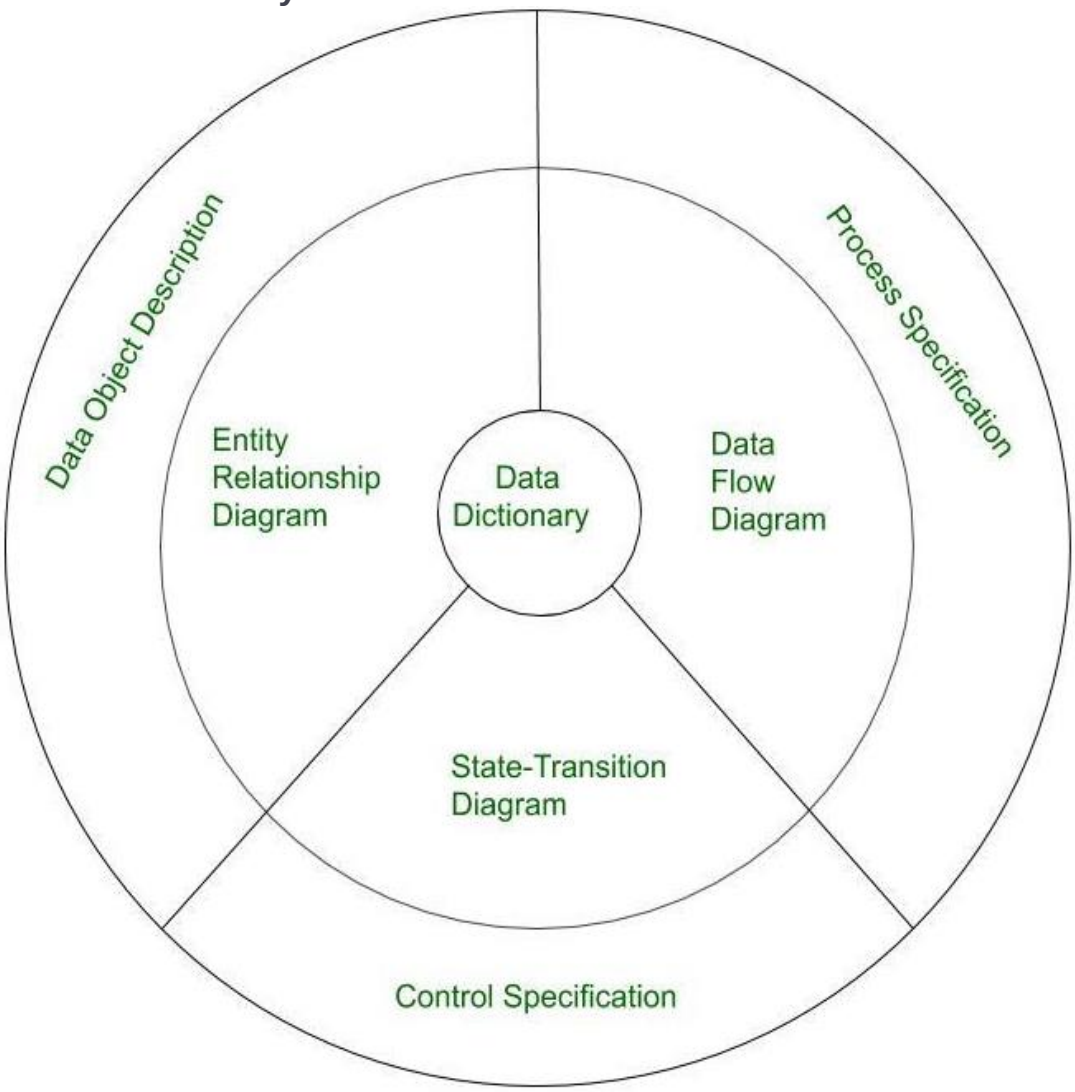
Measurement is done by metrics. Three parameters are measured: process measurement through process metrics, product measurement through product metrics, and project measurement through project metrics.

Analysis Model is a technical representation of the system. It acts as a link between system description and design model. In Analysis Modelling, information, behavior and functions of the system is defined and translated into the architecture, component and interface level design in the design modeling.

Objectives of Analysis Modelling:

1. It must establish a way of creation of software design.
2. It must describe requirements of customer.
3. It must define set of requirements which can be validated, once the software is built.

Elements of Analysis Model:



1. **Data Dictionary:**
   It is a repository that consists of description of all data objects used or produced by software. It stores the collection of data present in the software. It is a very crucial element of the analysis model. It acts as a centralized repository and also helps in modelling of data objects defined during software requirements.

2. **Entity Relationship Diagram (ERD):**
   It depicts relationship between data objects and used in conducting of data modelling activity. The attributes of each object in the Entity Relationship Diagram can be described using Data object description. It provides the basis for activity related to data design.

3. **Data Flow Diagram (DFD):**
   It depicts the functions that transform data flow and it also shows how data is transformed when moving from input to output. It provides the additional information which is used during the analysis of information domain and serves

as a basis for the modeling of function. It also enables the engineer to develop models of functional and information domain at the same time.

4. **State Transition Diagram:**
   It shows various modes of behavior (states) of the system and also shows the transitions from one state to other state in the system. It also provides the details of how system behaves due to the consequences of external events. It represents the behavior of a system by presenting its states and the events that cause the system to change state. It also describes what actions are taken due to the occurrence of a particular event.

5. **Process Specification:**
   It stores the description of each functions present in the data flow diagram. It describes the input to a function, the algorithm that is applied for transformation of input, and the output that is produced. It also shows regulations and barriers imposed on the performance characteristics that are applicable to the process, and layout constraints that could influence the way in which the process will be implemented.

6. **Control Specification:**
   It stores the additional information about the control aspects of the software. It is used to indicate how the software behaves when an event occurs and which processes are invoked due to the occurrence of the event. It also provides the details of the processes which are executed to manage events.

7. **Data Object Description:**
   It stores and provides the complete knowledge about a data object present and used in the software. It also gives us the details of attributes of the data object present in Entity Relationship Diagram. Hence, it incorporates all the data objects and its attributes.

.

3.Metrics for Design Model

Software Engineering-Metrics for Design model

SOFTWARE ENGINEERING

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design

metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative .
We can examine some of the more common design metrics for computer software. Each can provide the designer with improved insight and all can help the design to evolve to a higher level of quality.

Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

4.Metrics for source code

Source code metrics

These are measurements of the source code that make up all your software. Source code is the fundamental building block of which your software is made, so measuring it is key to making sure your code is high-caliber. (Not to mention there is almost always room for improvement.) Look closely enough at even your best source code, and you might spot a few areas that you can optimize for even better performance.

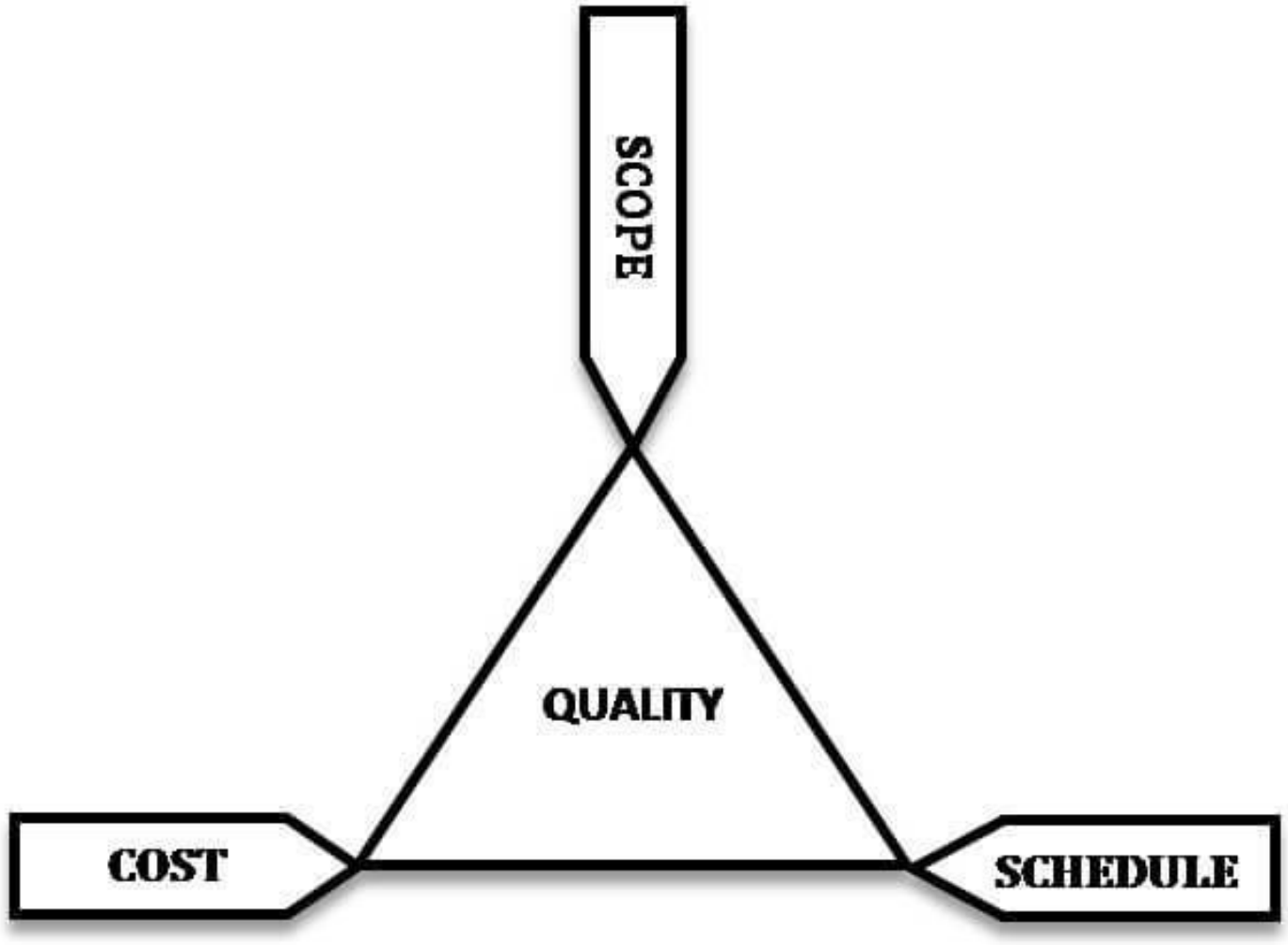
When measuring source code quality make sure you're looking at the number of lines of code you have, which will ensure that you have the appropriate amount of code and it's no more complex than it needs to be. Another thing to track is how compliant each line of code is with the programming languages' standard usage rules. Equally important is to track the percentage of comments within the code, which will tell you how much maintenance the program will require. The less comments, the more problems when you decide to change or upgrade. Other things to include in your measurements is code duplications and unit test coverage, which will tell you how smoothly your product will run (and at when are you likely to encounter issues).

5.Metrics for testing

By Shubh Thakare +919595939264

Software Testing Metrics

Software Testing Metrics are the quantitative measures used to estimate the progress, quality, productivity and health of the software testing process. The goal of software testing metrics is to improve the efficiency and effectiveness in the software testing process and to help make better decisions for further testing process by providing reliable data about the testing process.



Software testing metrics - Improves the efficiency and effectiveness of a software testing process.

Software testing metrics or software test measurement is the quantitative indication of extent, capacity, dimension, amount or size of some attribute of a process or product.

Example for software test measurement: Total number of defects

By Shubh Thakare +919595939264

6.Metrics for maintenance.

Software Quality Metrics Overview

Metrics for Software Maintenance

When development of a software product is complete and it is released to the market, it enters the maintenance phase of its life cycle. During this phase the defect arrivals by time interval and customer problem calls (which may or may not be defects) by time interval are the de facto metrics.

The following metrics are therefore very important:

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix Backlog and Backlog Management Index

Fix backlog is a workload statement for software maintenance. It is related to both the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process. Another metric to manage the backlog of open, unresolved, problems is the backlog management index (BMI).

$$\text{BMI} = \frac{\text{Number of problems closed during the month}}{\text{Number of problem arrivals during the month}} \times 100\%$$

As a ratio of number of closed, or solved, problems to number of problem arrivals during the month, if BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased. With enough data points, the techniques of control charting can be used to calculate the backlog management capability of the maintenance process. More investigation and analysis should be triggered when the value of BMI exceeds the control limits. Of course, the goal is always to strive for a BMI larger than 100. A BMI trend chart or control chart should be examined together with trend charts of defect arrivals, defects fixed (closed), and the number of problems in the backlog.

Figure 4.5 is a trend chart by month of the numbers of opened and closed problems of a software product, and a pseudo-control chart for the BMI. The latest release of the product was available to customers in the month for the first data points on the two charts. This explains the rise and fall of the problem arrivals and closures. The mean BMI was 102.9%, indicating that the capability of the fix process was functioning normally. All BMI values were within the upper (UCL) and lower (LCL) control limits—the backlog management process was in control. (*Note:* We call the BMI chart a pseudo-control chart because the BMI data are autocorrelated and therefore the assumption of independence for control charts is violated. Despite not being "real" control charts in statistical terms, however, we found pseudo-control charts such as the BMI chart quite useful in software quality management. In Chapter 5 we provide more discussions and examples.) A variation of the problem backlog index is the ratio of number of opened problems (problem backlog) to number of problem arrivals during the month. If the index is 1, that means the team maintains a backlog the same as the problem arrival rate. If the index is below 1, that means the team is fixing problems faster than the problem arrival rate. If the index is higher than 1, that means the team is losing ground in their problem-fixing capability relative to problem arrivals. Therefore, this variant index is also a statement of fix responsiveness.

Fix Response Time and Fix Responsiveness

For many software development organizations, guidelines are established on the time limit within which the fixes should be available for the reported defects. Usually the criteria are set in accordance with the severity of the problems. For the critical situations in which the customers' businesses are at risk due to defects in the software product, software developers or the software change teams work around the clock to fix the problems. For less severe defects for which circumventions are available, the required fix response time is more relaxed. The fix response time metric is usually calculated as follows for all problems as well as by severity level:

Mean time of all problems from open to closed

If there are data points with extreme values, medians should be used instead of mean. Such cases could occur for less severe problems for which customers may be satisfied with the circumvention and didn't demand a fix. Therefore, the problem may remain open for a long time in the tracking report.

In general, short fix response time leads to customer satisfaction. However, there is a subtle difference between fix responsiveness and short fix response time. From the customer's perspective, the use of averages may mask individual differences. The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer. For example, John takes his car to the dealer for servicing in the early morning and needs it back by noon. If the dealer promises noon but does not get the car ready until 2 o'clock, John will not be a satisfied customer. On the other hand, Julia does not need her mini van back until she gets off from work, around 6 P.M. As long as the dealer finishes servicing her van by then, Julia is a satisfied customer. If the dealer leaves a timely phone message on her answering machine at work saying that her van is ready to pick up, Julia will be even more satisfied. This type of fix responsiveness process is indeed being practiced by automobile dealers who focus on customer satisfaction.

In this writer's knowledge, the systems software development of Hewlett-Packard (HP) in California and IBM Rochester's systems software development have fix responsiveness processes similar to the process just illustrated by the automobile examples. In fact, IBM Rochester's practice originated from a benchmarking exchange with HP some years ago. The metric

for IBM Rochester's fix responsiveness is operationalized as percentage of delivered fixes meeting committed dates to customers.

Percent Delinquent Fixes

The mean (or median) response time metric is a central tendency measure. A more sensitive metric is the percentage of delinquent fixes. For each fix, if the turnaround time greatly exceeds the required response time, then it is classified as delinquent:

$$\text{Percent delinquent fixes} = \frac{\text{Number of fixes that exceeded the response time criteria by severity level}}{\text{Number of fixes delivered in a specified time}} \times 100\%$$

This metric, however, is not a metric for real-time delinquent management because it is for closed problems only. Problems that are still open must be factored into the calculation for a real-time metric. Assuming the time unit is 1 week, we propose that the percent delinquent of problems in the active backlog be used. *Active backlog* refers to all opened problems for the week, which is the sum of the existing backlog at the beginning of the week and new problem arrivals during the week. In other words, it contains the total number of problems to be processed for the week—the total workload. The number of delinquent problems is checked at the end of the week. Figure 4.6 shows the real-time delivery index diagrammatically.

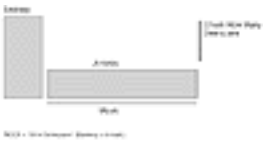


FIGURE 4.6 Real-Time Delinquency Index

It is important to note that the metric of percent delinquent fixes is a cohort metric. Its denominator refers to a cohort of problems (problems closed in a given period of time, or problems to be processed in a given week). The cohort concept is important because if it is operationalized as a cross-sectional measure, then invalid metrics will result. For example, we have seen practices in which at the end of each week the number of problems in backlog (problems still to be fixed) and the number of delinquent open problems were counted, and the percent delinquent problems was calculated. This cross-sectional counting approach neglects problems that were processed and closed before the end of the week, and will create a high delinquent index when significant improvement (reduction in problems backlog) is made.

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. From the customer's perspective, it is bad enough to encounter functional defects when running a business on the software. It is even worse if the fixes turn out to be defective. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction.

The metric of percent defective fixes is simply the percentage of all fixes in a time interval (e.g., 1 month) that are defective. A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure, the second is a process measure. The difference between the two dates is the latent period of the defective fix. It is meaningful to keep track of the latency data and other information such as the number of customers who were affected by the defective fix. Usually the longer the latency, the more customers are affected because there is more time for customers to apply that defective fix to their software system.

There is an argument against using percentage for defective fixes. If the number of defects, and therefore the fixes, is large, then the small value of the percentage metric will show an optimistic picture, although the number of defective fixes could be quite large. This metric, therefore, should be a straight count of the number of defective fixes. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.