

Architectural Modeling

Introduction

- Patterns and Frameworks
- Component Diagrams
- Deployment Diagrams
- A Detailed Case Study on System Analysis and Design using Unified Approach.

Patterns and Frameworks

- All well-structured systems are full of patterns. A pattern provides a common solution to a common problem in a given context.
- A mechanism is a design pattern that applies to a society of classes; a framework is typically an architectural pattern that provides an extensible template for applications within a domain.
- You use patterns to specify mechanisms and frameworks that shape the architecture of your system. You make a pattern approachable by clearly identifying the slots, tabs, knobs, and dials that a user of that pattern may adjust in order to apply the pattern in a particular context.
- A pattern is a common solution to a common problem in a given context. A mechanism is a design pattern that applies to a society of classes.
- A framework is an architectural pattern that provides an extensible template for applications within a domain.

Patterns and Frameworks

- In all well-structured systems, you'll find lots of patterns at various levels of abstraction.
- **Design patterns** specify the structure and behavior of a society of classes; **architectural patterns** specify the structure and behavior of an entire system.
- In practice, there are two kinds of patterns of interest- design patterns and frameworks- and the UML provides a means of modeling both.
- When you model either pattern, you'll find that it typically stands alone in the context of some larger package, except for dependency relationships bind them to other parts of your system.

Design Patterns

- Patterns support reuse of software architecture and design.
- Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain.
- Frameworks support reuse of detailed design and code.
- A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications.
- Together, design patterns and frameworks help to improve software quality and reduce development time.
- e.g., reuse, extensibility, modularity, performance.
-

Design Patterns

- Design patterns represent solutions to problems that arise when developing software within a particular context
 - i.e., “Patterns == problem/solution pairs in a context”
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
 - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs

Modeling Design Patterns

- To model a design pattern,
 - Identify the common solution to the common problem and reify it as a mechanism.
 - Model the mechanism as a collaboration, providing its structural, as well as its behavioral, aspects.
 - Identify the elements of the design pattern that must be bound to elements in a specific context and render them as parameters to the collaboration.

Frameworks

- 1. Frameworks are semi-complete applications
- Complete applications are developed by inheriting from, and instantiating parametrized framework components.
- 2. Frameworks provide domain-specific functionality
- e.g., business applications, telecommunication applications, window systems, databases, distributed applications, OS kernels
- 3. Frameworks exhibit inversion of control at run-time
- i.e., the framework determines which objects and methods to invoke in response to events

Modeling Architectural Patterns

- To model an Architectural pattern,
 - Harvest the framework from an existing, proven architecture.
 - Model the framework as a stereotyped package, containing all the elements (and especially the design patterns) that are necessary and sufficient to describe the various views of that framework.
 - Expose the slots, tabs, knobs, and dials necessary to adapt the framework in the form of design patterns and collaborations. For the most part, this means making it clear to the user of the pattern which classes must be extended, which operations must be implemented, and which signals must be handled.

Component Diagrams

- Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems.
- A component diagram shows the organization and dependencies among a set of components.
- Component diagrams are used to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.
- Component diagrams are essentially class diagrams that focus on a system's components.
- *Component diagrams are not only important for visualizing, specifying, and documenting component-based systems, but also for constructing executable systems through forward and reverse engineering.*

Component Diagrams

- A component diagram is just a special kind of diagram and shares the same common properties as do all other diagrams- a name and graphical contents that are a projection into a model.
- What distinguishes a component diagram from all other kinds of diagrams is its particular content.

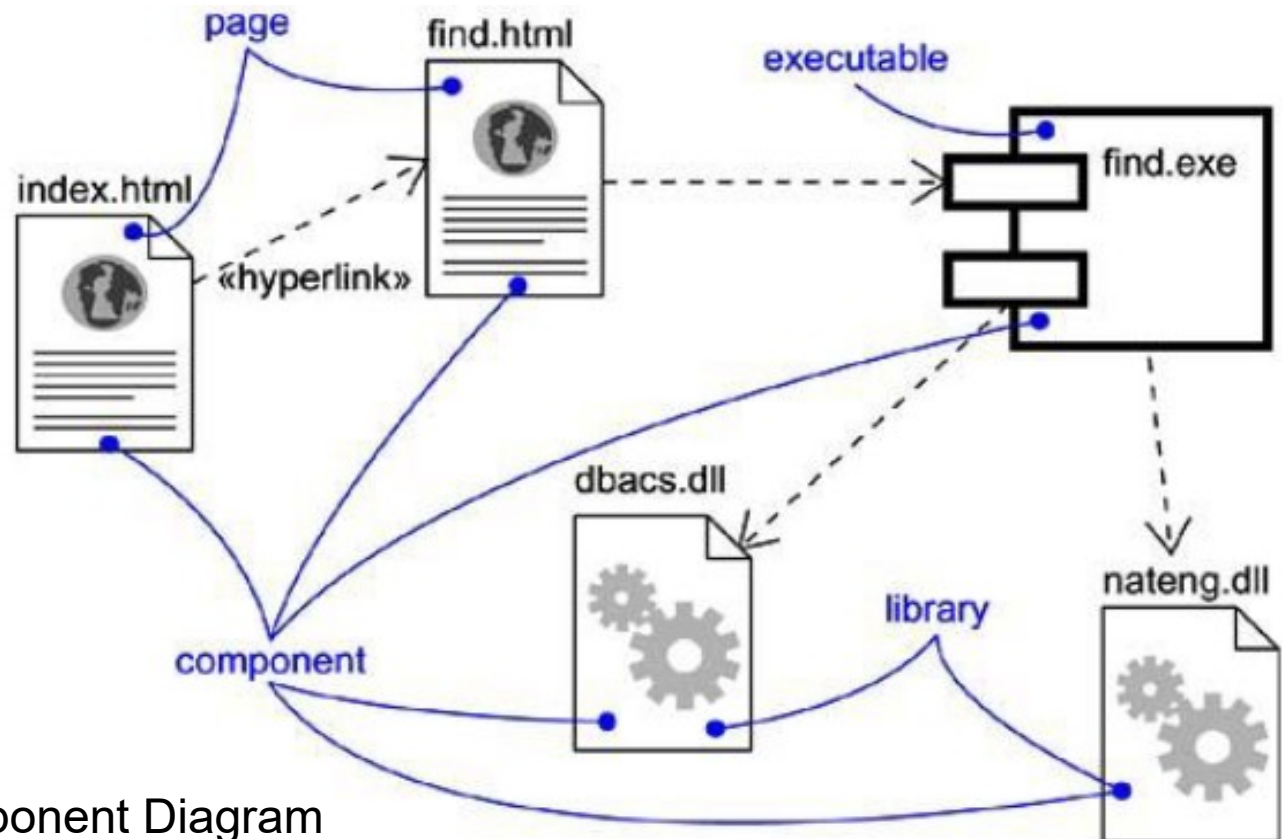


Figure 29-1 A Component Diagram

Component Diagrams

- You create use case diagrams to reason about the desired behavior of your system. You specify the vocabulary of your domain with class diagrams.
- You create sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams to specify the way the things in your vocabulary work together to carry out this behavior.
- Eventually, you will turn these logical blueprints into things that live in the world of bits, such as executables, libraries, tables, files, and documents. You'll find that you must build some of these components from scratch, but you'll also end up reusing older components in new ways.
- With the UML, you use component diagrams to visualize the static aspect of these physical components and their relationships and to specify their details for construction, as in Figure 29-1.

Component Diagrams

- Component diagrams commonly contain
 - Components
 - Interfaces
 - Dependency, generalization, association, and realization relationships
- Like all other diagrams, component diagrams may contain notes and constraints.
- Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks.
- Sometimes, you'll want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems.
- **Note**
- In many ways, a component diagram is just a special kind of class diagram that focuses on a system's components.

Component Diagrams

- Static implementation view represented by component diagrams primarily supports the configuration management of a system's parts, made up of components that can be assembled in various ways to produce a running system.
- When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.
 - 1. To model source code
 - 2. To model executable releases
 - 3. To model physical database
 - 4. To model adaptable systems

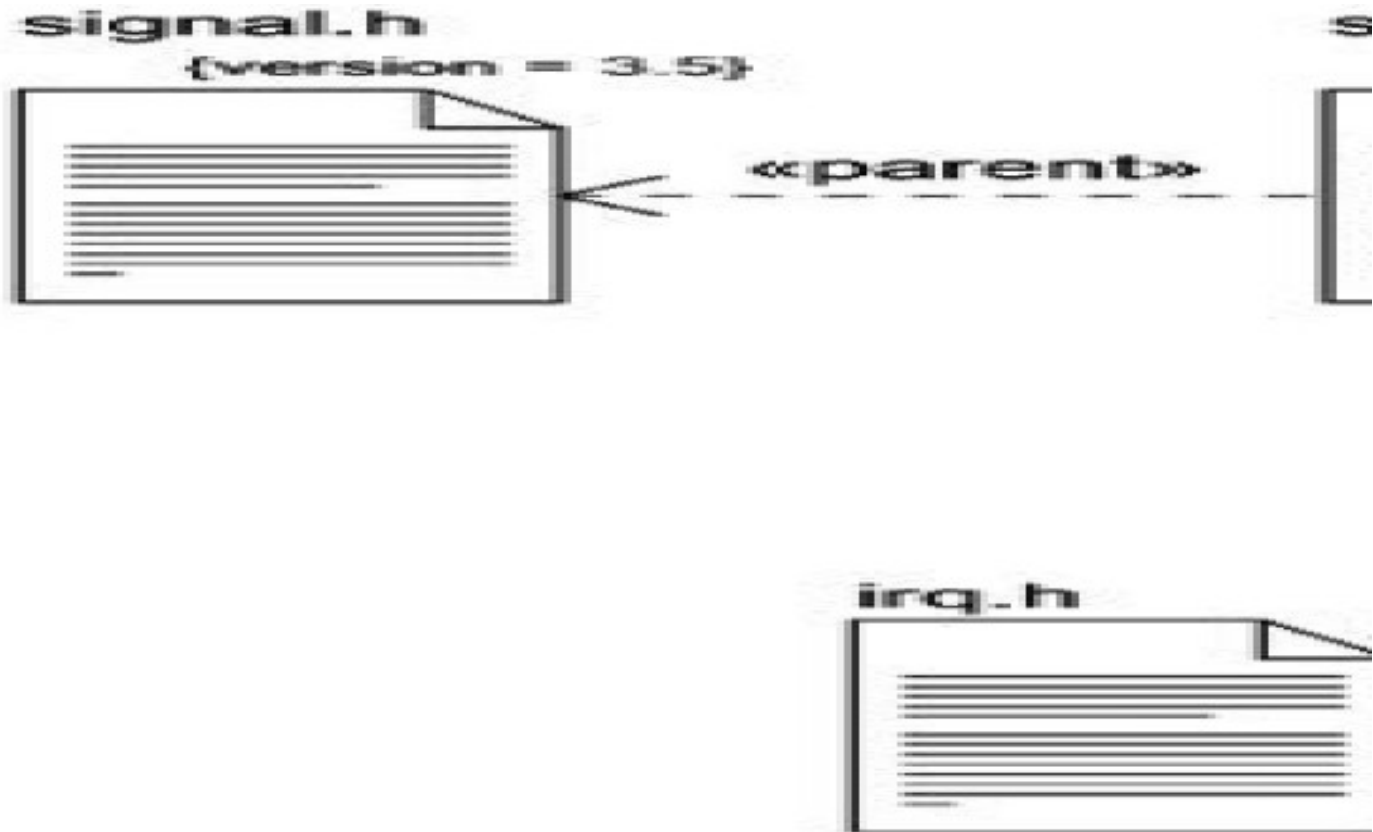
Component Diagrams

- **Modeling Source Code**
- To model a system's source code,
 - Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
 - For larger systems, use packages to show groups of source code files.
 - Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
 - Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies

Component Diagrams

- Modeling Source Code
-

Figure 29-2 Modeling Source Code



Component Diagrams

- **Modeling an Executable Release**
- To model an executable release,
 - Identify the set of components you'd like to model. 1) some or all the components that live on one node, or 2) the distribution of these sets of components across all the nodes in the system.
 - Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents).
 - For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others.

Component Diagrams

- Modeling an Executable Release
-

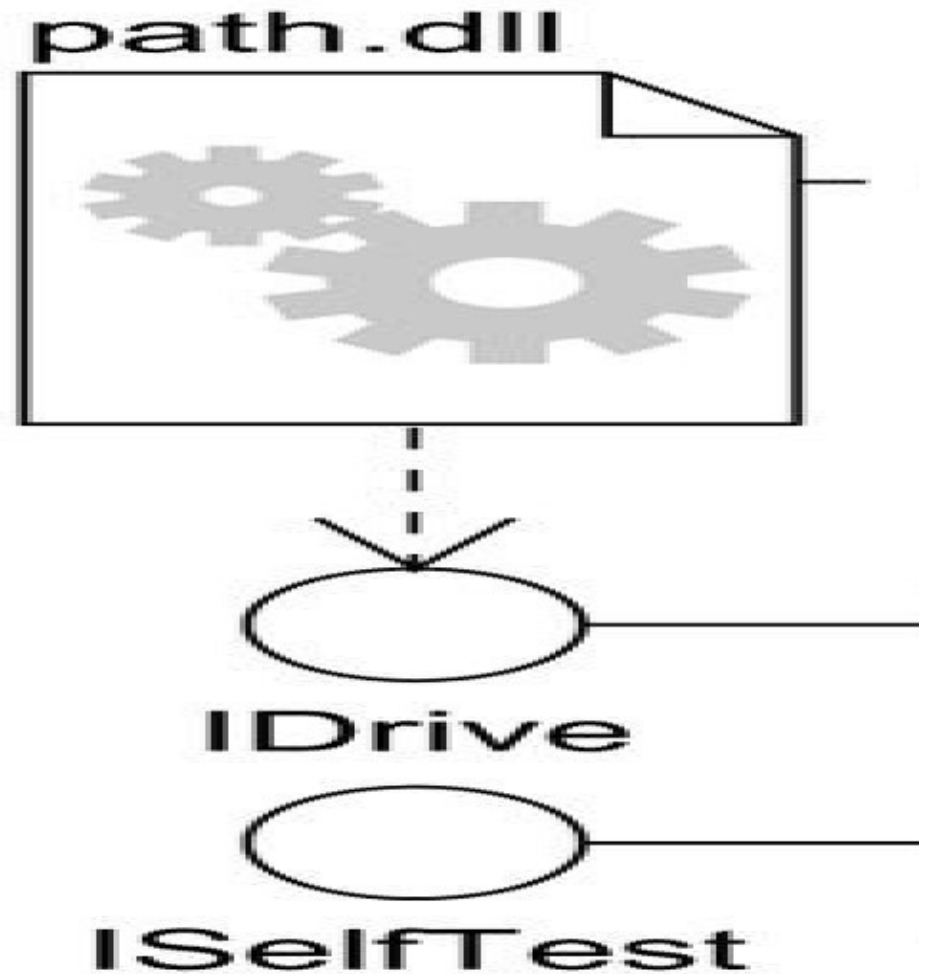


Figure 29-3 Modeling an Executable Release

Component Diagrams

- **Modeling a Physical Database**
- A logical database schema captures the vocabulary of a system's persistent data, along with the semantics of their relationships.
- Physically, these things are stored in a database for later retrieval, either a relational database, an object-oriented one, or a hybrid object/relational database.
- The UML is well suited to modeling physical databases, as well as logical database schemas.
- Mapping a logical database schema to an object-oriented database is straightforward because even complex inheritance lattices can be made persistent directly.
- Mapping a logical database schema to a relational database is not so simple, however.
-

Component Diagrams

- In the presence of inheritance, you have to make decisions about how to map classes to tables. Typically, you can apply one or a combination of three strategies.
- 1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add new child classes or modify your parent classes.
- 2. Collapse your inheritance lattices so that all instances of any class in a hierarchy has the same state. The downside with this approach is that you end up storing superfluous information for many instances.
- 3. Separate parent and child states into different tables. This approach best mirrors your inheritance lattice, but the downside is that traversing your data will require many cross-table joins.

Component Diagrams

- When designing a physical database, you also have to make decisions about how to map operations defined in your logical database schema.
- Object- oriented databases make the mapping fairly transparent.
- But, with relational databases, you have to make some decisions about how these logical operations are implemented. Again, you have some choices.
 1. For simple CRUD (create, read, update, delete) operations, implement them with standard SQL or ODBC calls.
 2. For more-complex behavior (such as business rules), map them to triggers or stored procedures.

Component Diagrams

- Given these general guidelines, to model a physical database,
 - Identify the classes in your model that represent your logical database schema.
 - Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.



Deployment Diagrams

- Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.
- A deployment diagram shows the configuration of run time processing nodes and the components that live on them.
- You use deployment diagrams to model the static deployment view of a system. For the most part, this involves modeling the topology of the hardware on which your system executes.
- Deployment diagrams are essentially class diagrams that focus on a system's nodes.
- *Deployment diagrams are not only important for visualizing, specifying, and documenting embedded, client/server, and distributed systems, but also for managing executable systems through forward and reverse engineering.*

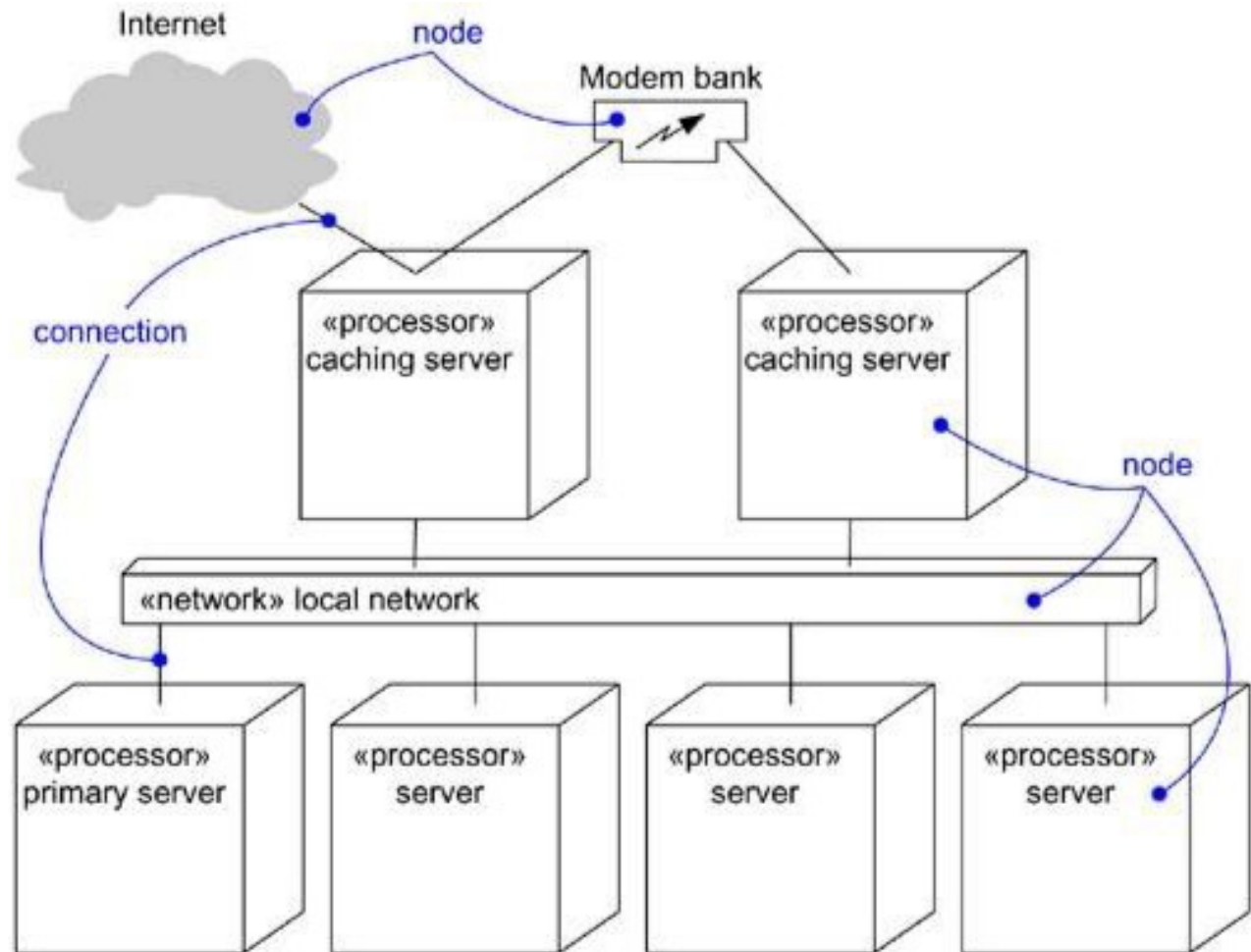
Deployment Diagrams

- The UML is designed to model many of the hardware aspects of a system sufficient for a software engineer to specify the platform on which the system's software executes and for a systems engineer to manage the system's hardware/software boundary.
- We use deployment diagrams to reason about the topology of processors and devices on which software executes.

Deployment Diagrams

- A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.
- Graphically, a deployment diagram is a collection of vertices and arcs.

Figure 30-1 A Deployment Diagram



Deployment Diagrams

- Deployment diagrams commonly contain
 - Nodes
 - Dependency and association relationships
- Like all other diagrams, deployment diagrams may contain notes and constraints.
- Deployment diagrams may also contain components, each of which must live on some node.
- Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your deployment diagrams, as well, especially when you want to visualize one instance of a family of hardware topologies.
- **Note**
- In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

Deployment Diagrams

- Deployment diagrams are used to model the static deployment view of a system. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.
- There are some kinds of systems for which deployment diagrams are unnecessary.
- If you are developing a piece of software that lives on one machine and interfaces only with standard devices on that machine that are already managed by the host operating system (for example, a personal computer's keyboard, display, and modem), you can ignore deployment diagrams.
- On the other hand, if you are developing a piece of software that interacts with devices that the host operating system does not typically manage or that is physically distributed across multiple processors, then using deployment diagrams will help you reason about your system's software-to-hardware mapping.

Deployment Diagrams

- When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.
- 1. To model embedded systems
- 2. To model client/server systems
- 3. To model fully distributed systems

Deployment Diagrams

- **Modeling an Embedded System**
- To model an embedded system,
 - Identify the devices and nodes that are unique to your system.
 - Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
 - Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
 - As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

Deployment Diagrams

- Modeling an Embedded System
-

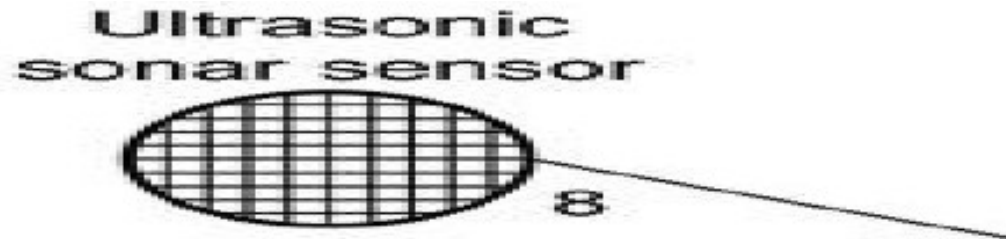
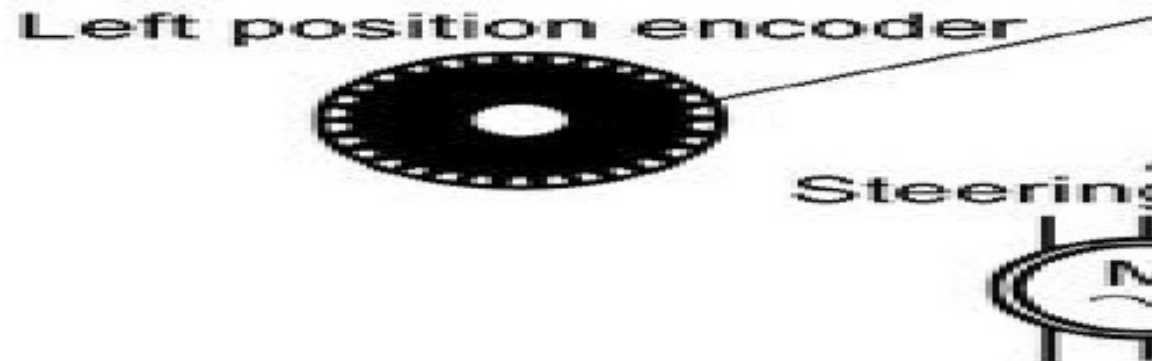


Figure 30-2 Modeling an Embedded System



Deployment Diagrams

- **Modeling a Client/Server System**
- To model a client/server system,
 - Identify the nodes that represent your system's client and server processors.
 - Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
 - Provide visual cues for these processors and devices via stereotyping.
 - Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

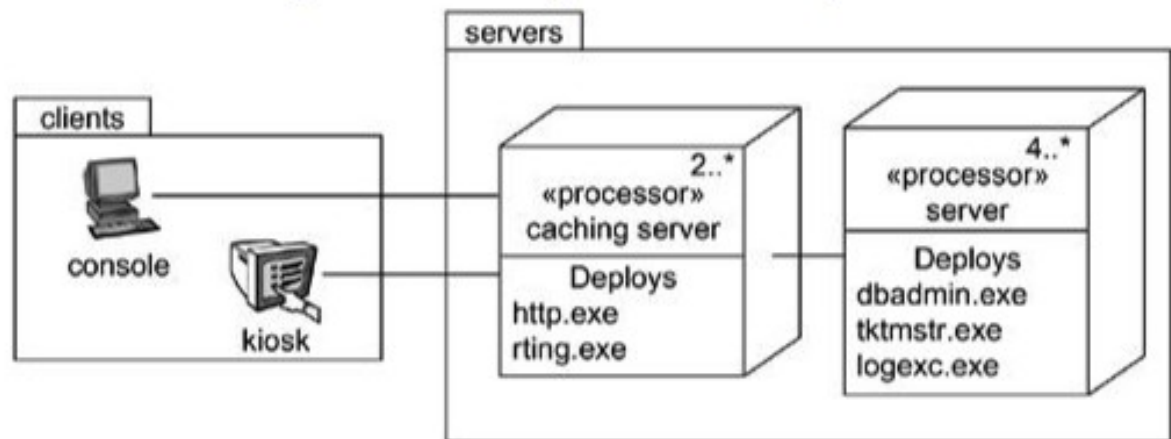


Figure 30-3
Modeling a Client/
Server System

Deployment Diagrams

- **Modeling a Fully Distributed System**
- To model a fully distributed system,
 - Identify the system's devices and processors as for simpler client/server systems.
 - If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
 - Pay close attention to logical groupings of nodes, which you can specify by using packages.
 - Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
 - If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.

Deployment Diagrams

- Modeling a Fully Distributed System
-

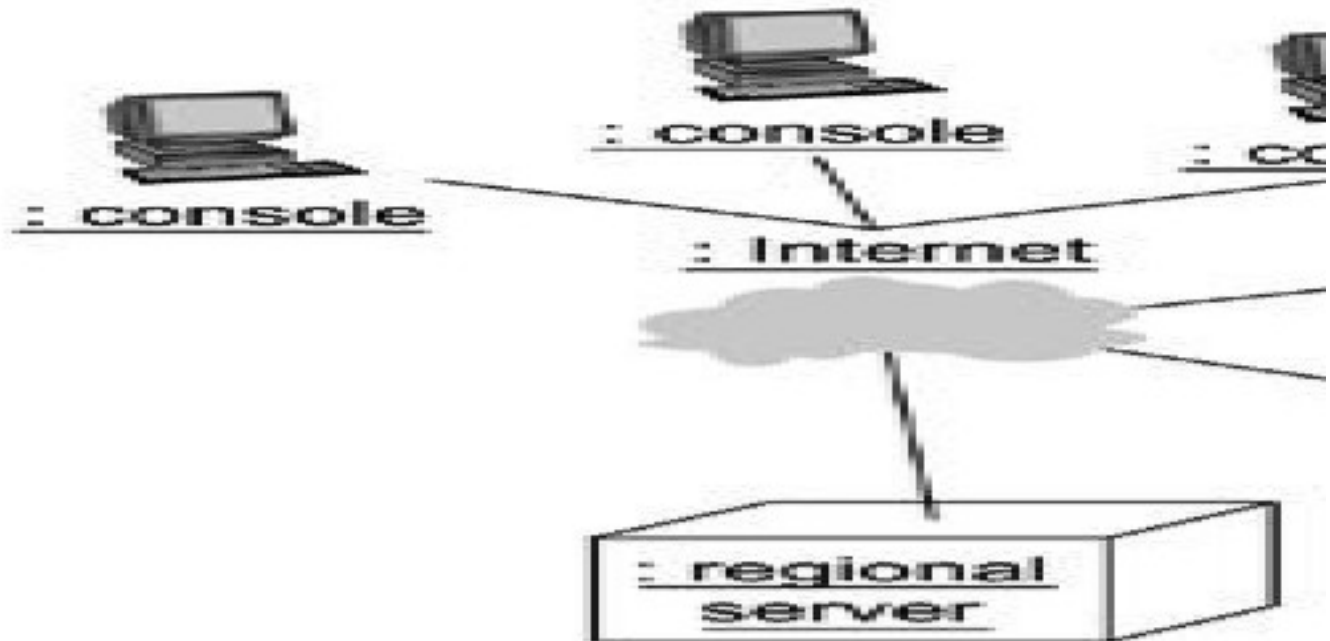


Figure 30-4 Modeling a Fully Distributed System

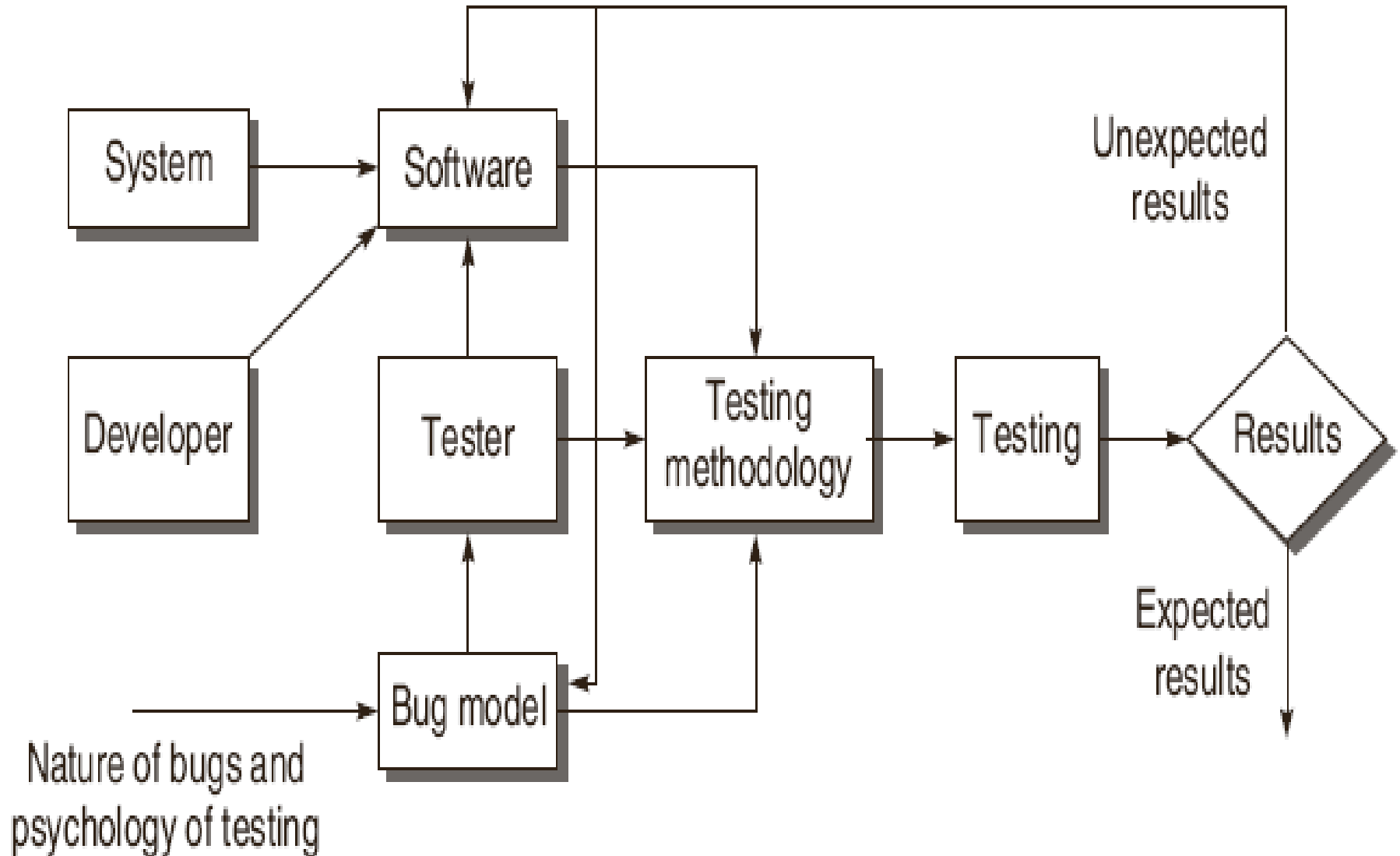
CHAPTER 2

SOFTWARE TESTING TERMINOLOGY AND METHODOLOGY

Objectives

- Difference between error, fault and failure.
- Life Cycle of a bug.
- How does a bug affect economics of software testing?
- How does a bug classified?
- Testing Principles
- Software Testing Life Cycle (STLC) and its models.
- Difference between verification and validation.
- Development of software testing methodology

MODEL FOR SOFTWARE TESTING



MODEL FOR SOFTWARE TESTING

- In effective software testing many terms need to be known.
- Terms like 'error', 'bug', 'defect', 'fault', 'failure' etc. but all are not synonyms.
- Bugs in general are more important during software testing.
- All bugs have different severity, according to that they need to be classified.
- A bug has complete life cycle from initiation to their death.
- These bugs also affect the project cost at various levels of development.
- Software testing is a complete process which has different phases in software testing life cycle.

SOFTWARE TESTING TERMINOLOGY

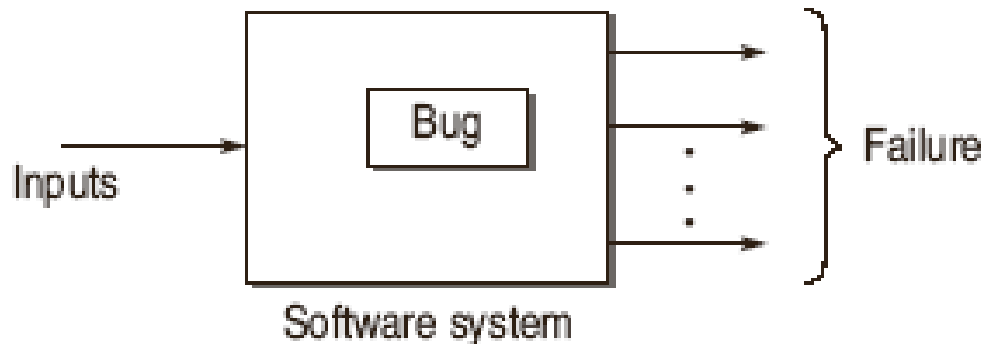
- **Failure**

The inability of a system or component to perform a required function according to its specification. **In other words, when results or behaviors of system under test is different as compared to specified specifications, then failure exists.**

- **Fault / Defect / Bug**

Fault is a condition that in actual causes a system to produce failure. It can be said that failures are manifestation of bugs.

One or more failures may be due to one or more bugs.



SOFTWARE TESTING TERMINOLOGY

- **Error**

Whenever a member of development team makes any mistake in any phase of SDLC, errors are produced.

It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does and so on. **Thus, error is a very general term used for human mistakes.**



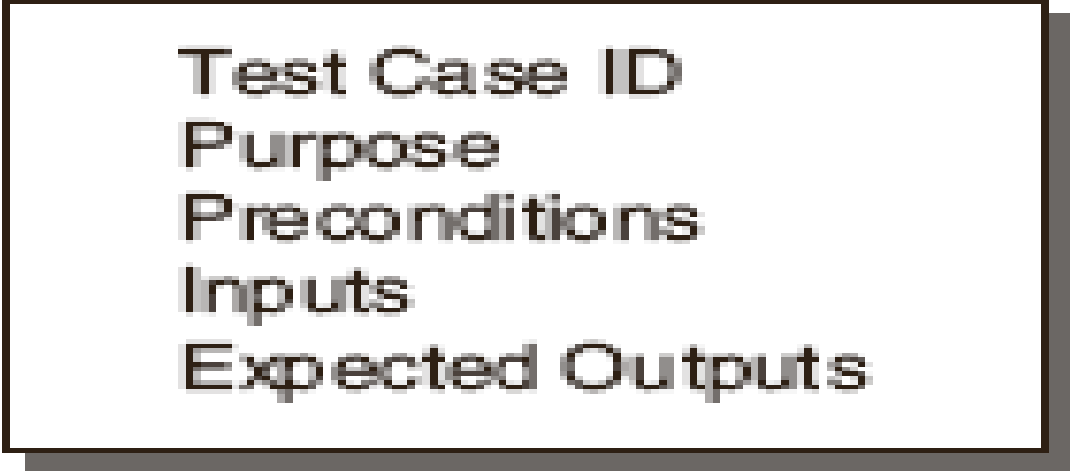
Flow of faults

SOFTWARE TESTING TERMINOLOGY

○ Test Case

Test case is a well documented procedure designed to test the functionality of a feature in the system.

A test case has an identity and is associated with a program behavior. The primary purpose of designing a test case is to find errors in the system.



Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs

SOFTWARE TESTING TERMINOLOGY

- **Test ware**

The documents created during the testing activities are known as Test ware. These are produced by test engineers. It may include test plans, test specifications, test case design, test reports etc.

- **Incident**

The symptom(s) associated with a failure that alerts the user to the occurrence of a failure.

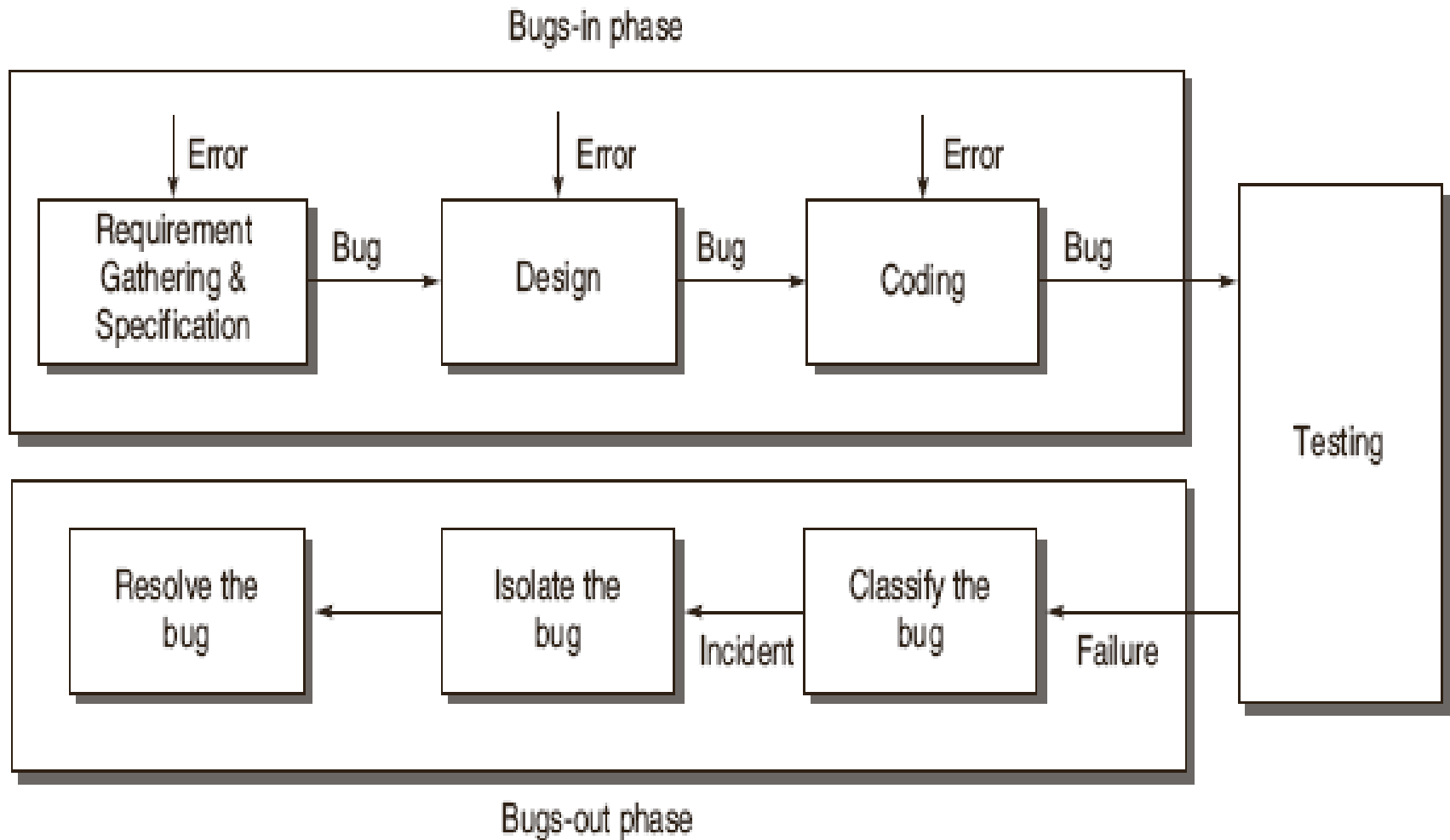
- **Test Oracle**

to judge the success or failure of a test,

LIFE CYCLE OF A BUG

- In testing phase, we analyze the incidents when the failure occurs.
- On the basis of symptoms derived from the incidents, a bug can be classified into certain categories.
- The bug can be isolated in the corresponding phase of SDLC and resolved by finding its exact location.
- The whole life cycle of a bug can be classified into two phases:
 - Bugs-in phase
 - Bugs-out phase

LIFE CYCLE OF A BUG



BUG-IN PHASE

- This phase is in where the errors and bugs are introduced in the software.
- Whenever people commit mistake, it creates errors on a specific location of the software and consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in software.
- This bug is carried out in subsequent phases of SDLC, if not detected.
- A phase may have its own errors as well as bugs received from the previous phase.

BUG-OUT PHASE

- If failures occurs while testing a software product, we come to conclusion that it is affected by bugs.
- In bug-out phase we observe failures the following activities are carried out to free from bugs.
 - **Bug classification:** Classify bug whether it can be critical or catastrophic in nature or it may have no adverse effect on the output behavior.

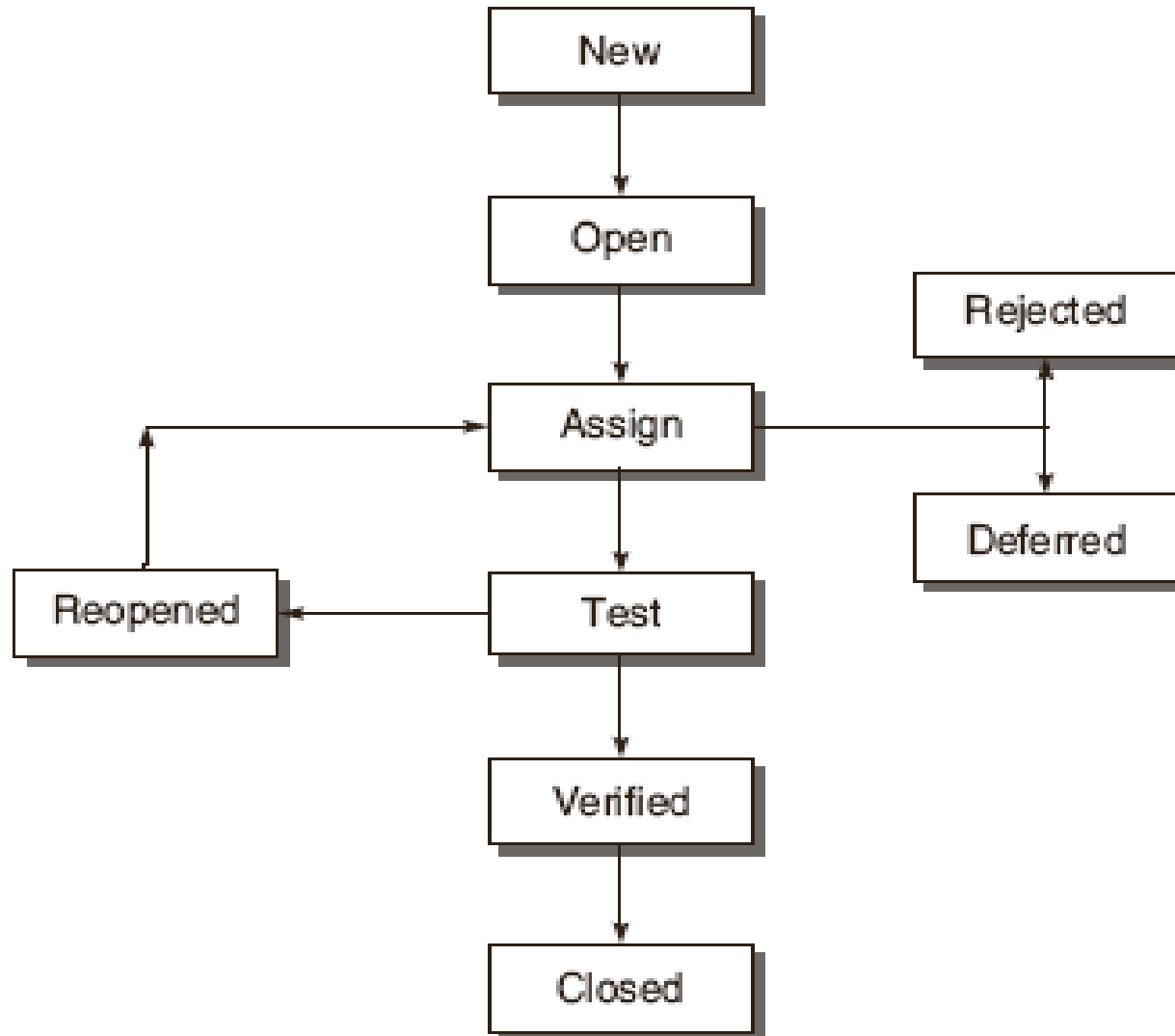
BUG-OUT PHASE

- **Bug isolation:** bug isolation is the activity by which we locate the module in which the bug appears. Incidents observed in failures help in this activity.
- **Bug resolution:** Once we have isolate the bug, we back-trace the design to pinpoint the location of the error. A bug is resolved when we have found exact location of its occurrence.

STATES OF A BUG

- *New*: The tester reported a bug first time the state is new.
- *Open*: If the bug is genuine then its state is open approved by test leader.
- *Assign*: An open bug comes to the development team where team verifies its validity.
 - If the bug is valid, a developer is assigned the job to fix it and its state is ASSIGN.
- *Deferred*: The developer who has been assigned to fix the bug will check its validity and priority.
 - If the priority of the bug is not high or there is not sufficient time to test then that state is deferred.

STATES OF A BUG



STATES OF A BUG

- *Rejected*: It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one.
- *Test*: After fixing the valid bug, the developer sends it back to the testing team for next round of checking.
 - Before sending them to testing team, the developer changes the bug's state to 'TEST'.
 - It specifies that bug has been fixed by the development team but not tested and is released to the testing team.
- *Verified/fixed*: The tester tests the software and verifies whether the reported bug is fixed or not.
 - After verifying, the developer approves that bug is fixed and changes the status to 'VERIFIED'.

STATES OF A BUG

- *Reopened*: If the bug is still there even after fixing it, the tester changes its status to 'REOPENED'. The bug traverses the life cycle once again.
 - In another case a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.
- *Closed*: Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to 'CLOSED'.

WHY DO BUGS OCCUR?

- **To Err is Human:** errors are produced when developers commit mistakes.
- Bugs in earlier stages go undetected and Propagate: Unclear or constantly changing requirements, software complexity, programming errors, timelines, errors in bug tracking, timelines, communication gap, documentation errors.
 - Miscommunication in gathering requirements from the customer is a big source of bugs.
 - New feature added to module or existing feature removed can be linked to other modules or components in the software.

WHY DO BUGS OCCUR?

- Rescheduling of resources, redoing or discarding an already completed task and changes in hardware/ software requirements can affect the software.
- Assigning a new developer to the project midway can cause bugs.
- If proper coding standards are not followed , then new developer might not get all the relevant details of the project.
- Complexity in keeping a track of all the bugs can in turn cause more bugs. This gets harder when a bug has a very complex life cycle, i.e. when the number of times bugs open, closed, not accepted, ignored and reopened.

BUG AFFECTS ECONOMICS OF SOFTWARE TESTING

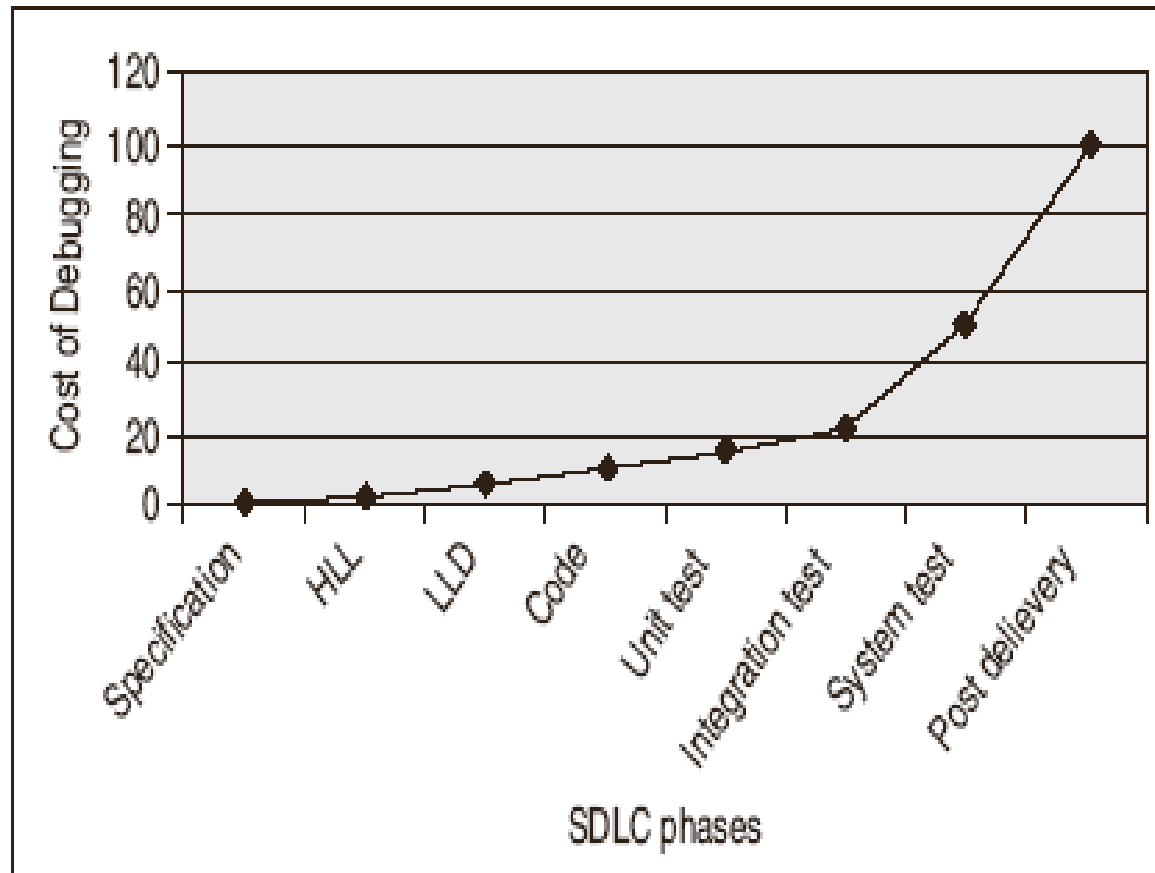
- Studies have demonstrated that testing prior to coding is 50% effective in detecting errors and after coding , it is 80% effective.
- It is at least 10 times as costly to correct an error after coding as before, and 100 times as costly to correct a production error(post-release error).
- If the bugs embedded in earlier stages go undetected, it is more difficult to detect them in later stages.
- So the cost to remove bugs from a system increases, as *the cost of a bug is equal to Detection Cost + Correction Cost*.

BUG AFFECTS ECONOMICS OF SOFTWARE TESTING

- A bug found and fixed early when the specification is being written, costs next to nothing.
- The same bug, if not found until the software is coded and tested, might cost ten times the cost in early stages.
- After the release of the product, if a customer finds the same bug, the cost will be 100 times more.
- Figure 2.6 outlines the relative cost to correct a defect depending on the SDLC in which it is discovered. Therefore, cost increase is Logarithmic.
- Analysis/Specification, Design, coding and Testing

BUG AFFECTS ECONOMICS OF SOFTWARE TESTING

Figure 2.6 Cost of debugging increases if bug propagates



BUG CLASSIFICATION BASED ON CRITICALITY

- Bugs can be classified based on the impact they have on the s/w under test. The classification then can be used to prioritize the bugs.
- **Critical Bugs** the worst effect on the functioning of software such that it stops or hangs the normal functioning of the software.
- **Major Bug**
This type of bug does not stop the functioning of the software but it causes a functionality to fail to meet its requirements as expected.

BUG CLASSIFICATION BASED ON CRITICALITY

- **Medium Bugs**

Medium bugs are less critical in nature as compared to critical and major bugs.

- **Minor Bugs**

These types of bugs do not affect the functionality of the software. These are just mild bugs which occur without any effect on the expected behavior or continuity of the software. For example, typographical error or misaligned printout.

BUG CLASSIFICATION BASED ON SDLC

- ***Requirement Specification:*** The first type of bug in SDLC is in the requirement gathering and specification phase.
- It has been observed that most of the bugs appear in this phase only.
- If these bugs go undetected, they propagate into subsequent phases.

BUG CLASSIFICATION BASED ON SDLC

- Requirement gathering and specification is a difficult phase in the sense that requirements gathered from the customer are to be converted into a *requirement specification* which will become the base for design.
- There may be a possibility that requirements specified are not exactly what the customers want.
- Moreover, specified requirements may be incomplete, ambiguous, or inconsistent.
- Specification problems lead to wrong missing, or superfluous features.

BUG CLASSIFICATION BASED ON SDLC

Design Bugs

- *Control Flow Bugs*: If we look at the control flow of a program then there may be many errors.
- For example, some paths through the flow may- be missing; there may be unreachable paths, etc.
- *Logic Bugs*: Any type of logical mistakes made in the design is a logical bug.
- For example, improper layout of cases, missing cases, improper combination of cases, misunderstanding of the semantics of the order in which a Boolean expression is evaluated.

BUG CLASSIFICATION BASED ON SDLC

Design Bugs

- *Processing Bugs:* Any type of computation mistakes result in processing bugs.
- Examples include arithmetic error, incorrect conversion from one data representation to another, ignoring overflow, improper use of logical operators, etc.

BUG CLASSIFICATION BASED ON SDLC

Design Bugs

- *Data flow bugs* : Control flow cannot identify data errors. For this, 'Ye use data flow (through data flow graph-discussed in Chapter 5).
- There may be data flow anomaly errors like un-initialized data, initialized in wrong format, data initialized but not used, data used but not initialized, redefined without any intermediate use, etc.

BUG CLASSIFICATION BASED ON SDLC

- **Error handling bugs:** *There may be errors about error handling in the software.*
- There are situations in the system when exception handling mechanisms must be adopted.
- If the system fails, then there must be an error message or the system should handle the error in an appropriate way.
- If you forget to do all this, then error handling bugs appear.

BUG CLASSIFICATION BASED ON SDLC

- **Race condition bugs:** *Race conditions there are two inputs A and B. According to design A precedes B in most of the cases.*
- *But, B can also come first I rare and restricted conditions. This is the race conditions, as the possibility of preceding B in restricted condition has not been taken care, resulting in a race condition bug.*
- *In multiprocessing system and interactive systems there may be many race conditions. Race conditions are among the least tested.*

BUG CLASSIFICATION BASED ON SDLC

- **Boundary-related bugs:** *Most of the time, the designers forget to take into consideration what will happen if any aspect of a program goes beyond its minimum and maximum values.*
- For example, there is one integer whose range is between 1 to 100. What will happen if a user enters a value as 1 or 101? When the software fails at the boundary values, then these are known as boundary-related bugs. There may be boundaries in loop, time, memory, etc.

BUG CLASSIFICATION BASED ON SDLC

- **User interface bugs** *There may be some design bugs that are related to users* If the user does not feel good while using the software, then there are use interface bugs.
- Examples include inappropriate functionality of some feature not doing what the user expects; missing, misleading, or confusing information; inappropriate error messages, wrong content in help text. Etc.

BUG CLASSIFICATION BASED ON SDLC

- **Coding Bugs:** There may be a long list of coding bugs.
- If you are a programmer, then are aware of some common mistakes made.
- For example, undeclared routines, dangling code, typographical errors, documentation bugs, i.e. erroneous comments lead to bugs in maintenance.

BUG CLASSIFICATION BASED ON SDLC

- **Interface and Integration Bugs:** External interface bugs include invalid timing or sequence assumptions related to external signals, misunderstanding external input and output formats, and user interface bugs.
- Internal interface bugs include input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call parameter bugs, and misunderstood entry or exit parameter values.

BUG CLASSIFICATION BASED ON SDLC

- Integration bugs result from inconsistencies or incompatibilities between modules discussed in the form of interface bugs.
- There may be bugs in data transfer and data sharing between the modules.

BUG CLASSIFICATION BASED ON SDLC

- **System Bugs:** There may be bugs while testing the system as a whole based on various parameters like performance, stress, compatibility, usability, etc.
- For example, in a real-time system, stress testing is very important, as the system must work under maximum load.
- If the system is put under maximum load at every factor like maximum number of users, maximum memory limit, etc. and if it fails, then there are system bugs.

BUG CLASSIFICATION BASED ON SDLC

- **Testing Bugs:** One can question the presence of bugs in the testing phase because this phase is dedicated to finding bugs.
- But the fact is that bugs are present in testing phase also. After all, testing is also performed by testers - humans.

BUG CLASSIFICATION BASED ON SDLC

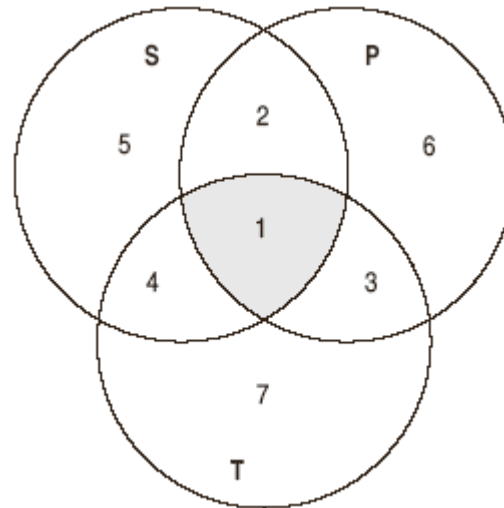
- Some testing mistakes are:
 - failure to notice/report a problem,
 - failure to use the most promising test case,
 - failure to make it clear how to reproduce the problem,
 - failure to check for unresolved problems just before the release,
 - failure to verify fixes,
 - failure to provide summary report.

TESTING PRINCIPLES

- Effective Testing not Exhaustive Testing
- Testing is not a single phase performed in SDLC
- Destructive approach for constructive testing
- Early Testing is the best policy.
- The probability of the existence of an error in a section of a program is proportional to the number of errors already found in that section. => *If there is one bug, then there can be many*
- Testing strategy should start at the smallest module level and expand toward the whole program.

TESTING PRINCIPLES

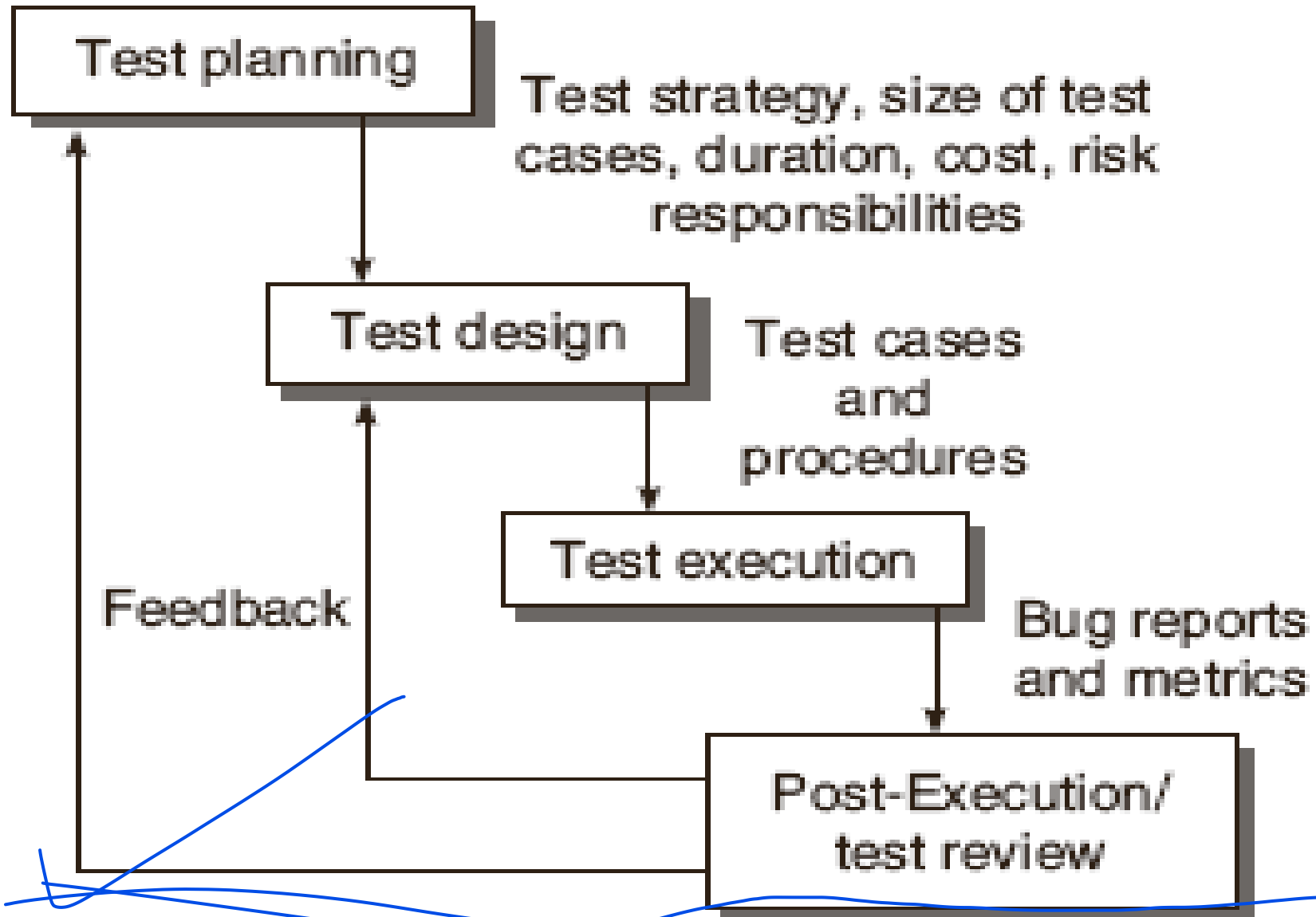
- Testing should also be performed by an independent team.
- Everything must be recorded in software testing.
- Invalid inputs and unexpected behavior have a high probability of finding an error.
- Testers must participate in specification and design reviews.



SOFTWARE TESTING LIFE CYCLE (STLC)

- Since we considered testing as a process, we need to define steps for successful and effective testing.
- This process is called *Software Testing Life Cycle* (STLC).
- Major contribution of STLC is to involve testers at the early stage of development.
- This ensures great benefit in project schedule and cost as well measuring specific milestones.

SOFTWARE TESTING LIFE CYCLE (STLC)



TEST PLANNING

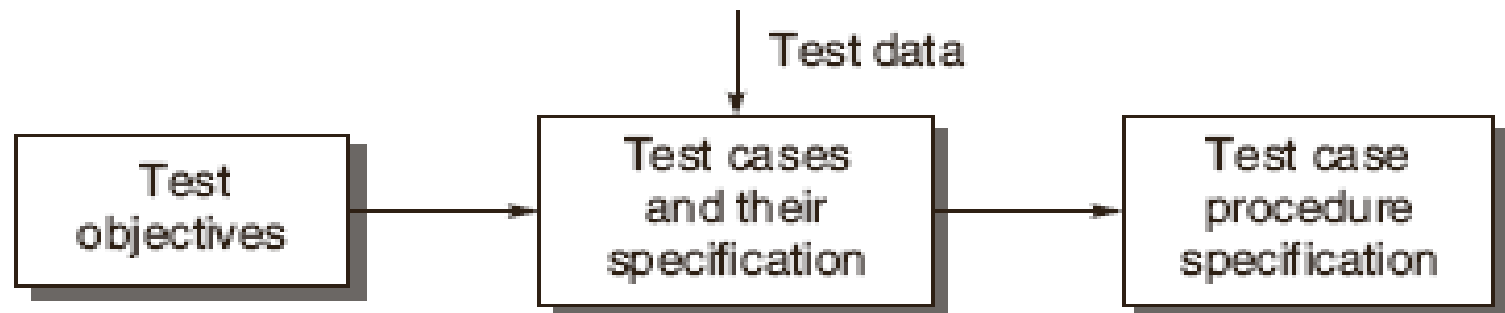
- Goal of test planning is to take up important issues like: resources, schedules, responsibilities, risks, priorities, etc.
 - Defining the Test Strategy
 - Estimate number of test cases, their duration and cost.
 - Plan the resources like the manpower, tools, documents, etc.
 - Identify areas of risks.
 - Defining the test completion criteria.
 - Identification of methodologies, techniques and tools for various test cases.
 - Identifying reporting procedures, bug classification, databases for testing, Bug Severity levels, project metrics.

TEST PLANNING

- Based on planning, analysis is done for various testing activities. The major is test plan document.
- Test plans are developed for each level of testing. After analyzing the issues the following activities are carried out.
 - Develop a test case format
 - Develop test case plans according to every phase of SDLC.
 - Identify test cases to be automated.
 - Prioritize the test cases according to their importance and criticality.
 - Define areas of stress and performance testing.
 - Plan the test cycles required for regression testing.

TEST DESIGN

- Determining the test objectives and their Prioritization
- Preparing List of Items to be Tested
- Mapping items to test cases
- Selection of Test case design techniques
- Creating Test Cases and Test Data
- Setting up the test environment and supporting tools
- Creating Test Procedure Specification



TEST DESIGN

- Determining the test objectives and their prioritization: The test objectives reflect the fundamental elements that need to be tested to satisfy an objective.
- Preparing list of items to be tested The objectives thus obtained are now converted into lists of items that are to be tested under an objective.

TEST DESIGN

- Mapping items to test cases After making a list of items to be tested, there is a need to identify the test cases.
- A matrix can be created for this purpose, identifying which test case will be covered by which item. Thus it permits reusing the test cases.
- This matrix will help in:
 - a) Identifying the major test scenarios.
 - b) Identifying and reducing the redundant test
 - c) Identifying the absence of a test case for a particular objective and as a result, creating them.

TEST DESIGN

- Designing the test cases demands a prior analysis of the program at functional or structural level.
- Some attributes of a good test case are given below:
 - a) A test case that has been designed keeping in view the criticality and high-risk requirements in order to place a greater priority upon, and provide added depth for testing the most important functions[12].
 - b) It should be designed such that there is a high probability of finding an error.
 - c) Test cases should not overlap or be redundant. Each test case should address a unique functionality, thereby not wasting time and resources.

TEST DESIGN

d) Although we can sometimes, combine a series of tests into one, a good test case should be designed with a modular approach for the sake of complexity, re-usability and to be recombined to execute various functional paths.

It also avoids masking of errors and duplication of test-creation efforts [7, 12].

e) A successful test case is one that has the highest probability of detecting an as-yet-undiscovered error [2].

- Selection of test case design techniques While designing test cases, there are two broad categories , namely black- box testing and white-box testing.

TEST DESIGN

- Creating test cases and test data: based on the testing objectives identified. The objective under which a test case is being designed, the i/ps required, and the expected o/ps.
 - In i/p specifications, test data must be chosen and specified with care, to prevent incorrect execution of test cases.
- Setting up the test environment and supporting tools: So details like hardware configurations, testers, interfaces, operating systems, and manuals must be specified during this phase.

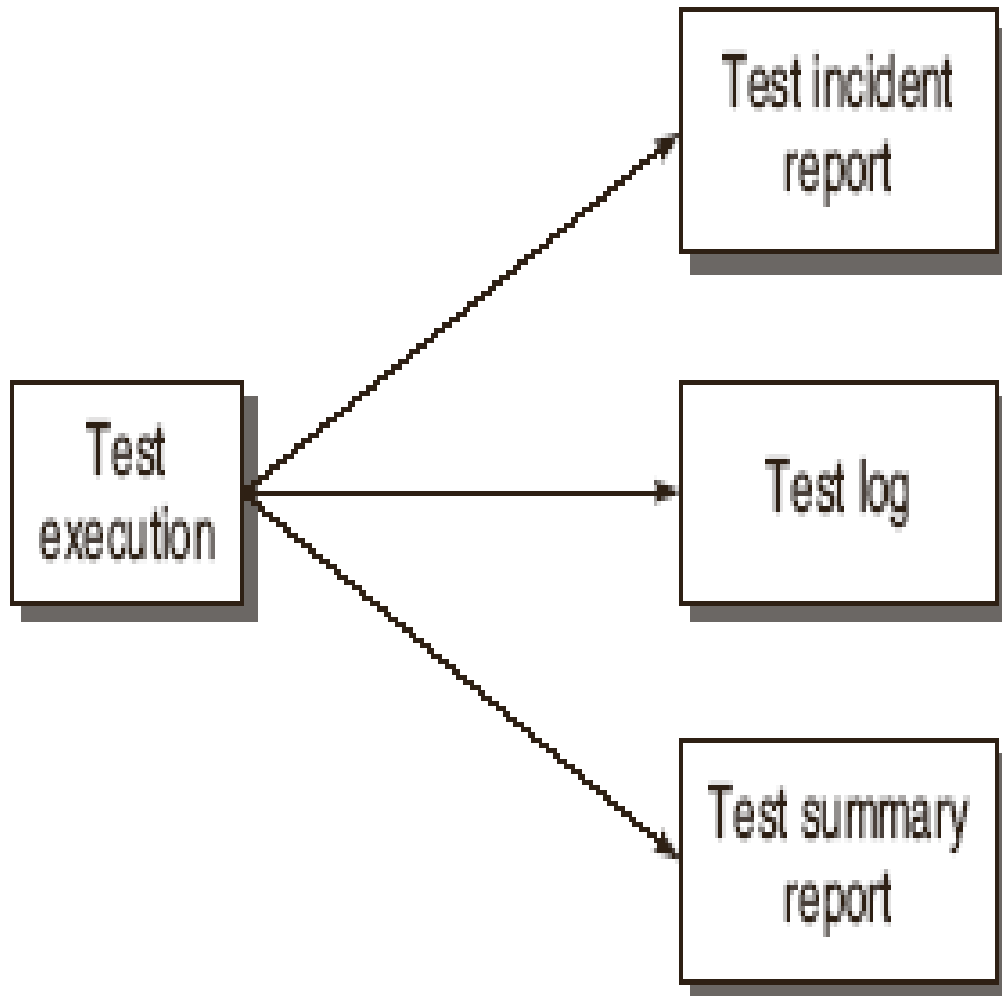
TEST DESIGN

- Creating test procedure specification :description of how the test case will be run, in the form of sequenced steps used by the tester at the time of execution of test cases.
- **The hierarchy for test design phase includes:** developing test objectives, identifying test cases and creating their specifications, and then developing test case procedure specifications as shown in Fig. 2.9.
- Test Design document provides the details of the input specifications, output specifications, environmental needs, and other procedural requirements for **the test case**.
- **Input speci – Required i/p for testing**
- **Output speci – expected output**
- **Needs of other nature like hardware, software**
- **Any other special requirements**

TEST EXECUTION

- In this phase, all test cases are executed including verification and validation.
- Verification test cases are started at the end of each phase of SDLC. Validation test cases are started after the completion of a module.
- It is the decision of the test team to opt for automation or manual execution.
- Test results are documented in the test incident reports, test logs, testing status, and test summary reports.

TEST EXECUTION



Test Execution	Person Responsible
Unit	Developer of the module
Integration	Testers and Developers
System	Testers, End-Users, Developers
Acceptance	Testers, End-Users

POST-EXECUTION / TEST REVIEW

- ***Understanding the bug:*** The developer analyses the bug reported and builds an understanding of its whereabouts.
- ***Reproducing the bug:*** Next, he confirms the bug by reproducing the bug and the failure that exists. (Necessary to cross-check failures). However, some not reproducible bugs increases the problems of developers.
- ***Analyzing the nature and cause of the bug:*** After examining the failures, the developer starts debugging its symptoms and tracks back to the actual location of the error in the design. the modified portion of the software is tested once again.

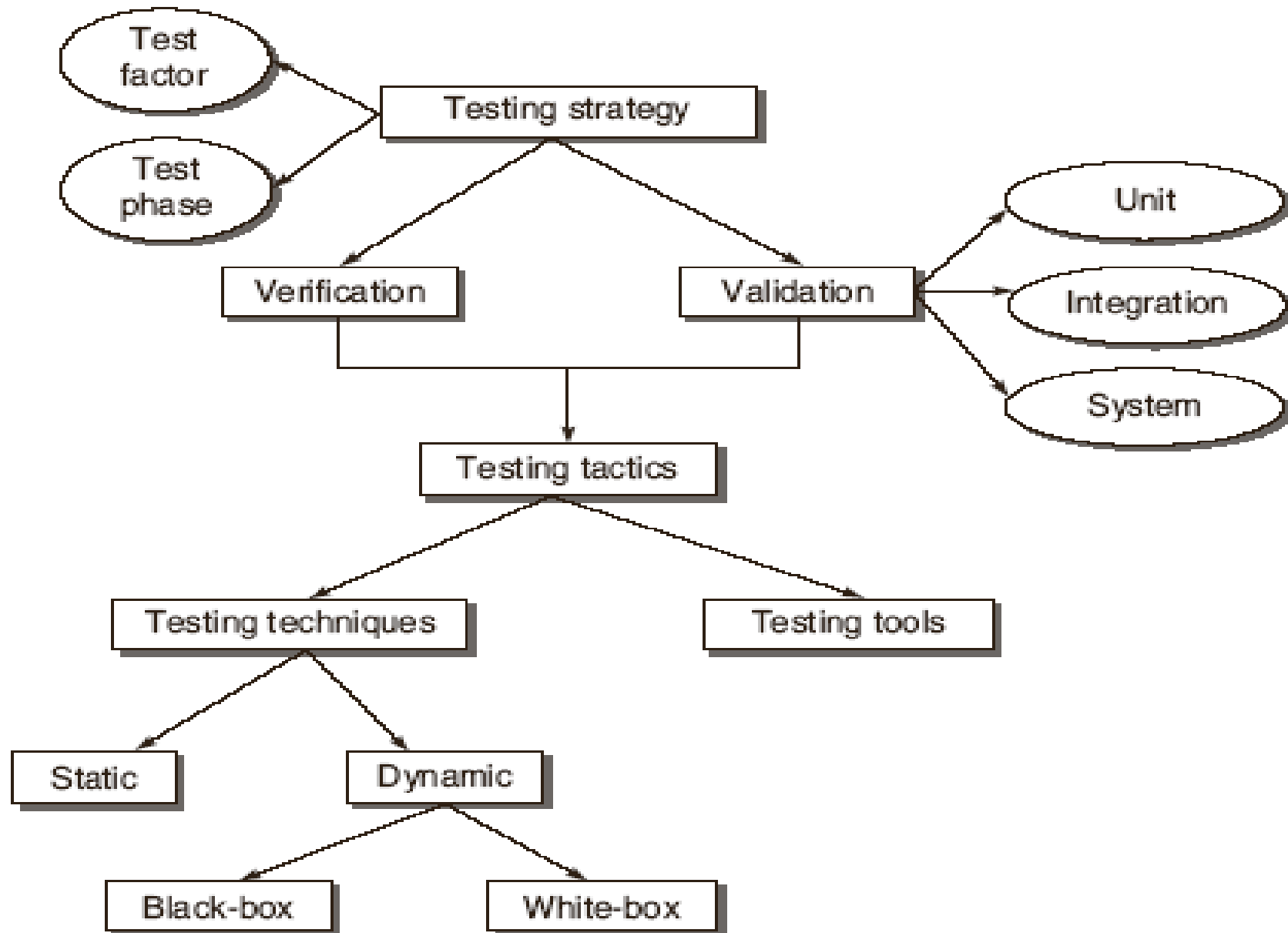
POST-EXECUTION / TEST REVIEW

- The results from manual and automated testing can be collected. The final bug report and associated metrics are reviewed and analyzed for overall testing process.
- The following activities can be done:
- ***Reliability analysis*** to establish whether the s/w meets the predefined reliability goals or not. If so, the product can be released and if not, then time and resources are required to reach reliability goals.
- ***Coverage Analysis:*** used as an alternative to stop testing.
- ***Overall Defect Analysis:*** to identify risk areas and focus on quality improvement.

SOFTWARE TESTING STRATEGY

- The planning of the whole testing process into a well-planned series of steps.
- A roadmap that includes very specific activities that must be performed by the test team in order to achieve a specific goal.
- **Test Factors:** Test factors are risk factors or issues related to system under development. Risk factors need to be selected and ranked according to a specific task.
- **Test Phase:** This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed. E.g. Strategies different for spiral and iterative waterfall.

SOFTWARE TESTING METHODOLOGY



TEST STRATEGY MATRIX

- Test strategy matrix identifies the concerns that will become the focus of test planning and execution.
- In this way, this matrix becomes an input to develop the testing strategy. The matrix is prepared by test factors and test phase.

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test
Portability	Is portability feature mentioned in specifications according to different hardware?					Is system testing performed on MIPS and INTEL platforms?
Service Level	Is time frame for booting mentioned?	Is time frame incorporated in design of the module?				

TEST STRATEGY MATRIX

- **Select and rank test factors:** Based on the test factors list, the most appropriate factors according to specific systems are selected and ranked from the most significant to the least. These are rows of a matrix.
- **Identify system development phases:** Different phases according to the adopted development model are listed as columns of the matrix. These are called test phases.
- **Identify risks associated with the system under development and its corresponding phase:** In horizontal column under each test phases, the test concern with the strategy used to address this concern is entered.

TEST STRATEGY MATRIX

- The risks may include any circumstance, events, actions that may prevent the test program from being implemented or executed according to schedule, such as late budget approval, delayed arrival of test equipment, or late availability of the software application.
- The test team must carefully examine risks in order to derive effective test and mitigation strategies for a particular application.
- Plan the test strategy for every risk identified so that risks are mitigated.

DEVELOPMENT OF TEST STRATEGY

- When project starts and progresses, testing too starts from the first step of SDLC.
- Rule for development of a test strategy includes testing ‘begins from the smallest unit and progresses to enlarge.
- This gives rise to two basic terms- Verification and Validation.

DEVELOPMENT OF TEST STRATEGY

Verification: “Are we building the product right?” is to check the software with its specification at every development phase such that any defect can be detected at any early stage of testing and will not be allowed to propagate further.

Verification refers to the set of activities that ensures correct implementation of functions in a software.

analysis-design-coding-testing

Are we building the product right?

Are we building the right product?

VALIDATION ACTIVITIES

Validation: “Are we building the right product?”. When different modules of a system is integrated, the system built after integration. This is validation testing.

Validation has the following three activities which are also known as the three levels of validation testing.

Unit Testing: it is a major validation effort performed on the smallest module of the system.

Unit testing is a basic level of testing which cannot be overlooked, and confirms the behavior of a single module according to its functional specifications.

VALIDATION ACTIVITIES

Integration Testing: it is a validation technique which combines all unit tested modules and performs a test on their aggregation?

When one module is combined with another in an integrated environment, interfacing between units must be tested.

We integrate the units according to the design and availability of units. Therefore, tester must be aware of system design.

VALIDATION ACTIVITIES

System Testing: This testing level focuses on testing the entire integrated system.

It incorporates many types of testing. The purpose is to test the validity for specific users and environments.

The validity of the whole system is checked against the requirement specification.

TESTING TACTICS

- A specific test strategy works as given below.
- Software testing techniques: effective testing is a real challenge.
 - Design effective test cases with most of the testing domains covered detecting the maximum number of bugs.
 - Actual methods for designing test cases, software testing technique, implement the test cases on the software.

TESTING TACTICS

- These techniques are categorized into two parts.
- **Static testing:** it is a technique for assessing the structural characteristics of source code, design specifications or any notational representation that conforms to well defined syntactical rules.

TESTING TACTICS

- **Dynamic Testing:** All the methods that execute the code to test a software are known as dynamic testing techniques.
- The code is run on a number of inputs provided by the user and the corresponding results are checked.
- This type of testing is further divided into sub two categories: Black box testing and White box testing.

TESTING TACTICS

- **Black box testing:** It checks only the functionality of system.
- This techniques received inputs given to system and the output is received after processing in the system.
- It is also known as functional testing. It is used for system testing under validation.

- **White box testing:** Every design feature and its corresponding code is checked logically with every possible path execution.
- So it takes care of structural paths instead of just outputs.
- It is known as structural testing and also used for unit testing under verification.

CONSIDERATIONS IN DEVELOPING TESTING METHODOLOGIES

- *Determine project risks:* A test strategy is developed with the help of another team members familiar with business risks associated with the software.
- *Identify test activities according to SDLC Phase:* After identifying all the risk factors in an SDLC phase, testing can be started in that phase. All testing activities must be recognized at all the SDLC phases.

CONSIDERATIONS IN DEVELOPING TESTING METHODOLOGIES

- *Determine the type of Development project:* The environment or methodology to develop the software also affects the testing risks.
 - The risks associated with a new development project will be different from a maintenance project or a purchased software.

BUILD THE TEST PLAN

- A test plan provides:
 - Environment and pre-test background
 - Overall objectives
 - Test team
 - Schedule and budget showing detailed dates for the testing
 - Resource requirements including equipments, software and personnel
 - Testing materials including system documentation, software to be tested, test inputs, test documentation, test tools.
 - Specified business functional requirements and structural functions to be tested
 - Type of testing technique to be adopted in a particular SDLC phase or what are the specific tests to be conducted in a particular phase.

Chapter 6

Static Testing

Introduction

- Static testing is complementary to dynamic testing and improves the software quality.
- Some bugs are detectable only through static testing.
- There are 3 types of static testing: Inspection, Walkthroughs and Reviews.
- Inspections are the most widely used formal technique for static testing at early stage.
- Benefits and effectiveness of inspection process
- Variants of Inspection process
- Walkthrough is a less formal and less rigorous as compared to inspection.
- Review is higher level technique as compared to inspection or walkthrough, as it also includes management.

Static Testing

- Do not execute the code, so no bulk of test cases are required.
- Do not demonstrate that the software is operational or one function of software is working;
- Checks the software product at each SDLC stage for conformance with the required specifications or standards.
- Requirements, design specifications, test plans, source code, user's manuals, maintenance procedures are some of the items that can be statically tested.
- Cost-effective technique of error detection.
- A bug is found at its exact location whereas a bug found in dynamic testing provides no indication to the exact source code location.

Benefits and Objectives of Static Testing

- **Benefits :**
 - A more technically correct base is available for each new phase of development.
 - Immediate evaluation and feedback to the author from his/her peers which will bring about improvements in the quality of future products.
 - The overall software life cycle cost is lower, since defects are found early and are easier and less expensive to fix.

Benefits and Objectives of Static Testing

- **Objectives :**
 - To identify errors in any phase of SDLC as early as possible.
 - To verify that the components of software are in conformance with its requirements
 - To provide information for project monitoring
 - To improve the software quality and increase productivity.

Static Testing

Types of Static Testing

- Software Inspections
- Walkthroughs
- Technical Reviews

Inspections

- Tackles software quality problems because they allow detection and removal of defects after each phase of the software development process.
- An in-process manual examination of an item to detect bugs.
- Embedded in the process of developing products and are done in early stages of each phase.
- Carried out by a group of peers. The group of peers first inspects the product at individual level.

Inspections

- It includes discussion of potential defects of the product observed in a formal meeting.
- Formal process of verification. The documents which can be inspected are SRS, SDD, code and test plan.
- Inspection process involves the interaction of the following elements:
 - Inspection steps
 - Roles for participants
 - Item being inspected

Inspections

- The entry and exit criteria are used to determine whether an item is ready to be inspected or not.
- For e.g. code inspection entry criterion is that the code has been compiled successfully.
- Exit criterion is that once item has been given for inspection it should not be updated, otherwise it will not know how many bugs have been reported and corrected through inspection process.

Inspections

- Inspection Team

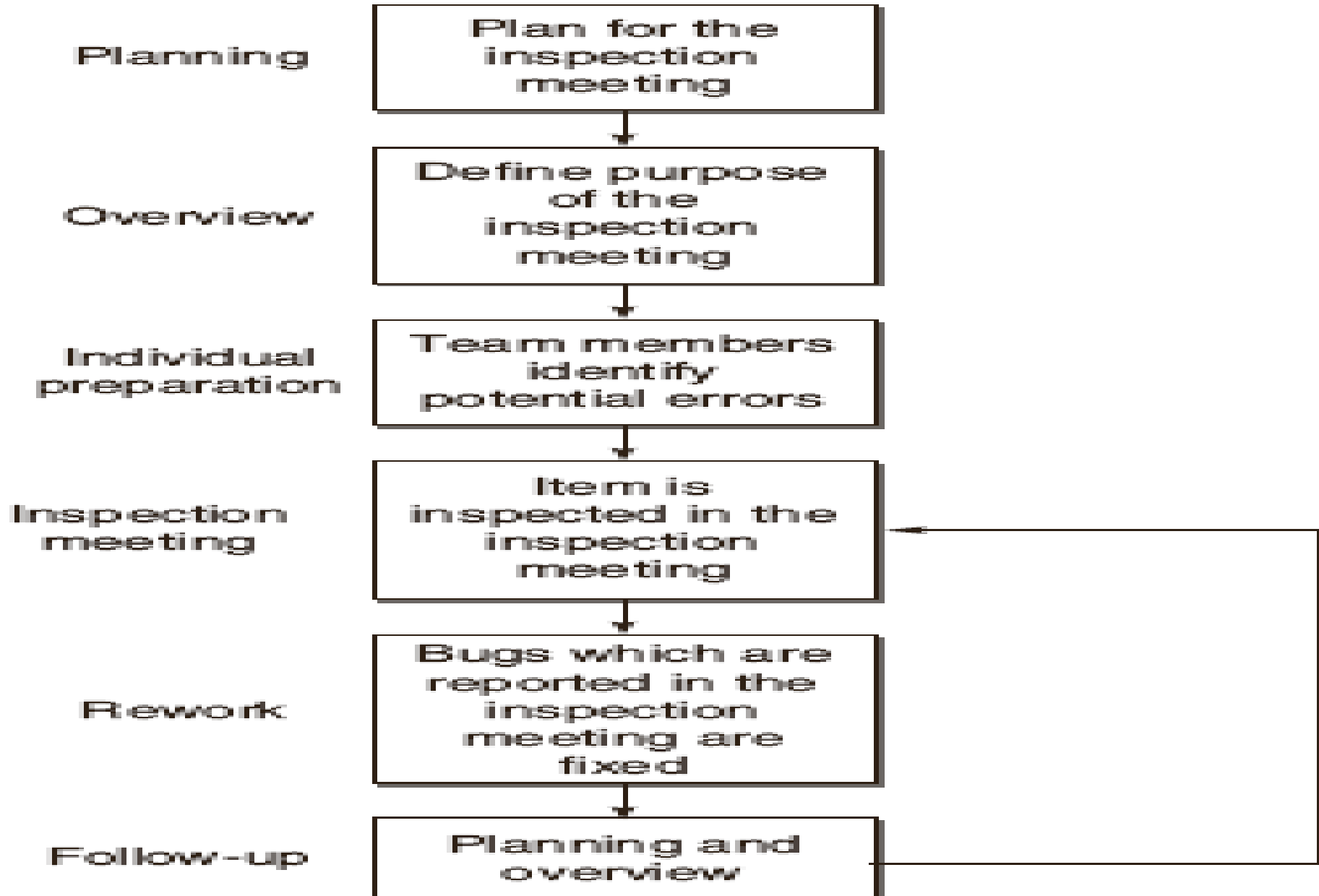
- *Author / Owner / Producer*: The programmer or designer responsible for producing the program or document as well as for fixing defects during the inspection process.
- *Inspector*: A peer member of the team, not a manager or supervisor responsible for finding errors, omissions, and inconsistencies in programs and documents.

Inspections

○ Inspection Team

- *Moderator*: A team member who manages the whole inspection process.
 - He schedules, leads, and controls the inspection session.
 - He is the key person with the responsibility of planning and successful execution of the inspection process.
- *Recorder*: One who records all the results of the inspection meeting.

Inspection Process



Planning

- The product to be inspected is identified.
- A moderator is assigned.
- The objective of the inspection is stated.
- During planning the moderator performs the following activities:
 - Assures that the product is ready for inspection
 - Selects the inspection team and assigns their roles.
 - Schedules the meeting venue and time.
 - Distributes the inspection materials like the item to be inspected, check lists etc.

Overview

- Inspection team is provided with the background information for inspection.
- The author presents the rationale for the product, its relationship to the rest of the products being developed, its function and intended to use, and approach used to develop it.
- This information is necessary for the inspection team to perform a successful inspection.
- The opening meeting is called by moderator. In this meeting the objective of inspection is explained to the team members.
- The idea is that every member should be familiar with the overall purpose of the inspection.

Individual preparation

- After the overview, the reviewer individually prepare them selves for inspection process by studying documents provided them in the overview sessions.
- Author point out errors or problems found in and record them in log. This log is submitted to moderator.
- The moderator compiles the logs of different members and give a copy of this compiled item to the author of the inspected item.

Individual preparation

- The inspector reviews the product for general problems as well as related to their specific area of expertise.
- Checklists are used during this stage for guidance on typical types of errors found.
- After reviewing, the inspectors record defects found on a log and the time spent during preparation.
- Completed logs submitted to the moderator prior to inspection meeting.
- The moderator reviews the logs submitted by each inspector to determine whether team is adequately prepared.

Inspection meeting

- Once all initial preparation is complete, the actual inspection meeting can start.
- The author first discusses every issue raised by different members in the compiled log file.
- The basic goal of inspection meeting is to uncover any bug in the item.
- No effort in meeting to fix the bug. It means that bugs are only being notified to the author, which he will fix it later.

Inspection meeting

- Every activity in meeting should be a constructive engagement so that more and more bugs can be discovered.
- It is duty of moderator to focused towards its objective and the author is not discouraged in any way.
- At the end, moderator concludes the meeting and produces a summary of the inspection meeting.
- This summary basically a list of errors found in the item that need to be resolved by the author.

Rework and Follow-up

○ Rework:

- The summary list of the bugs that arise during the inspection meeting needs to be reworked by the author.
- The author fixes all these bugs and reports back to the moderator.

○ Follow-up:

- It is the responsibility of the moderator to check all the bugs found in the last meeting have been addressed and fixed.
- He prepares a report and checks that all issues have been resolved.
- Then document is then approved for release.
- If this is not the case, then the unresolved issues are mentioned in a report and another inspection meeting is called by the moderator.

Benefits Inspection Process

- Bug Reduction: the number of bugs is reduced through the inspection process.
L . H.
 - Fenton [86] reported that through the inspection process in IBM, the number of bugs per thousand lines of code has been reduced by two-thirds.
- Bug Prevention : Based on experience of previous inspections, analysis can be made for future inspections or projects, thereby preventing bugs which have appeared earlier.

Benefits Inspection Process

- Productivity: Since all phases of SDLC may be inspected without waiting for code development and its execution, the cost of finding bugs decreases, resulting in an increase in productivity.
 - Errors are found at their exact places, therefore reducing the need of dynamic testing and debugging.
- Real-time Feedback to Software Engineers: The inspections also benefit software engineers/developers they get feedback on their products on a relatively real time basis.
 - Developers find out the types of errors and the error density.
 - Since they get feedback in early stages of development, they may improve their capability.

Benefits Inspection Process

- Reduction in development resource: the cost of rework is surprisingly high if inspections are not used and errors are found during development or testing.
 - Inspections reduce the effort required for dynamic testing and any rework during design and code, thereby causing an overall net reduction in the development resource.
- Quality Improvement: The direct consequence of static testing also results in the improvement of quality of the final product.
 - Inspections help to improve the quality by checking the standard compliance, modularity, clarity, and simplicity.

Benefits Inspection Process

- Project Management: A project needs monitoring and control. It depends on some data obtained from the development team.
 - Inspection is another effective tool for monitoring the progress of the project.
- Learning through inspection: Inspection also improves the capability of different team members, as they learn from discussions on various types of bugs and the reasons why they occur.
 - It is more beneficial for new members. They can learn about the project in a very short time.

Benefits Inspection Process

- Checking coupling and cohesion: The modules' coupling and cohesion can be checked easily through inspection as compared to dynamic testing.
 - This also reduces the maintenance work.
- Process Improvement: An analysis of why error occurred or frequent places where the errors occurred by inspection team members.
- The issues of process improvements are following:
 - Finding most error-prone modules: through the inspection process, the modules can be analyzed based on their error-density of the individual modules. So we can take some decision to
 - Redesign module,
 - check and rework on the code of module
 - Take extra precautions and test cases to test the module

Benefits Inspection Process

- Distribution of error types: inspections can also provide data according to the error-types.
- We can analyze the data of the percentage of bugs in a particular type of bug category.
- If we get data very early in process , then we can analyze which particular types of bugs may be repeating.

Effectiveness of Inspection Process

- The rate of inspection refers to how much evaluation of an item has been done by the team.
- If the rate is very fast, it means coverage of item to be evaluated is high.
- If the rate is too slow, it means that the coverage is not much.

Effectiveness of Inspection Process

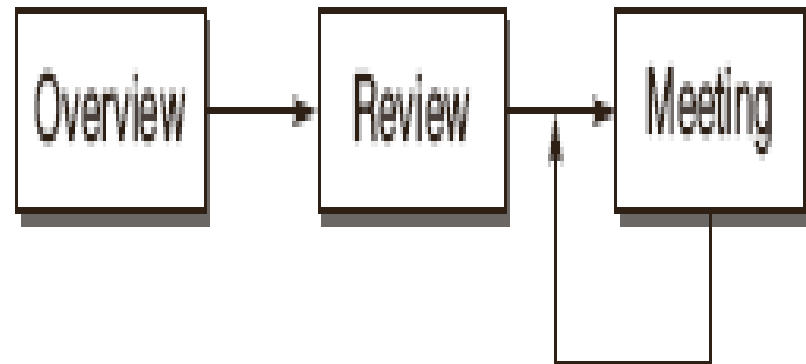
- If the rate is too fast, then it may be possible that it reveals only few errors.
- If the rate is too slow, the cost of project is increases.
- So, a balanced rate of inspection process should be followed so that it concentrates on a reasonable number of defects.
- Error detection Efficiency = $(\text{Error found by an inspection} / \text{Total errors in the item before inspection}) * 100$

Variants of Inspection process

Active Design Reviews (ADRs)	Several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area.
Formal Technical Asynchronous review method (FTArm)	Inspection process is carried out without really having a meeting of the members. This is a type of asynchronous inspection in which the inspectors never have to simultaneously meet.
Gilb Inspection	Defect detection is carried out by individual inspector at his level rather than in a group.
Humphrey's Inspection Process	Preparation phase emphasizes the finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase wherein individual logs are analysed and combined into a single list.
N-Fold inspections	Inspection process's effectiveness can be increased by replicating it by having multiple inspection teams.
Phased Inspection	Phased inspections are designed to verify the product in a particular domain by experts in that domain only.
Structured Walkthrough	Described by Yourdon. Less formal and rigorous than formal inspections. Roles are coordinator, scribe, presenter, reviewers, maintenance oracle, standards bearer, user representative. Process steps are Organization, Preparation, Walkthrough, and Rework. Lacks data collection requirements of formal inspections.

Active Design Reviews

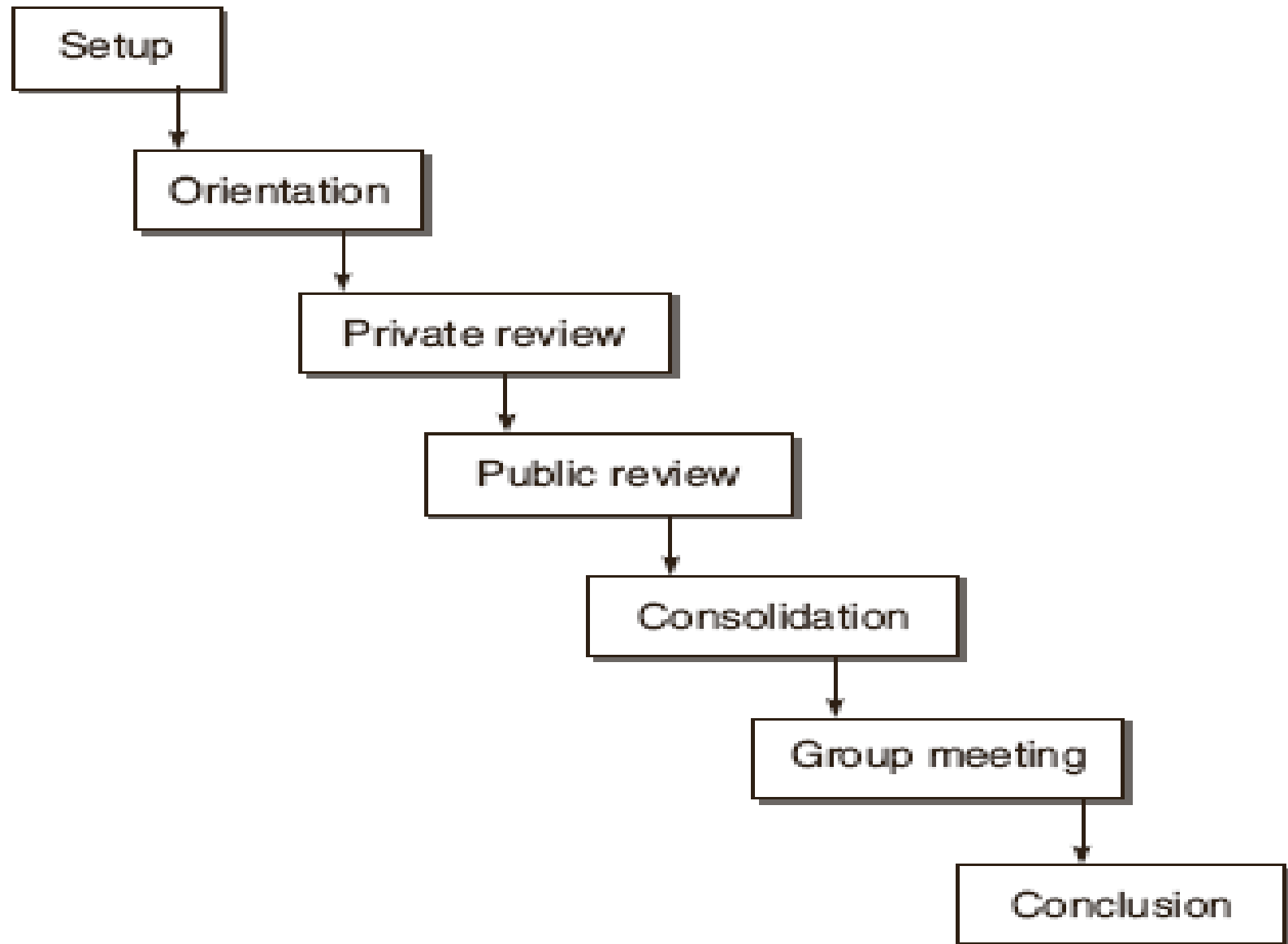
- Active Design Reviews are for inspecting the design stage of SDLC.
- In this process, several reviews are conducted targeting a particular type of bugs and conducted by reviewers who are experts in that area.
- It covers all sections of the design document based on several small reviews instead of only one inspection.
- It is also based on the use of questionnaires to give the reviewers better-defined responsibilities and to make them play a more active role.
- There are three stages:
 - Overview
 - Review
 - Meeting



Active Design Reviews

- Overview: A brief overview of the module being reviewed is presented.
- Review: Reviewers are assigned sections of the document to be reviewed and questionnaires based on the bug type along with the time frame to meet the designers.
- Meeting: The designers read the completed questionnaires and meet the reviewers to resolve any queries that the designers may have about the reviewer's answers to the questionnaires.
 - The interaction continue till reviewers not understand the issues, although an agreement on these not be reached.

Formal Technical Asynchronous review method (FTArm)



Formal Technical Asynchronous review method (FTArm)

- The meeting phase of inspection is considered expensive and therefore, the idea is to eliminate this phase.
- This is an asynchronous type of inspection in which inspectors never have to simultaneously meet.
- For this process, an online version of document is made available to every member so they can add their comments and point out the bugs.

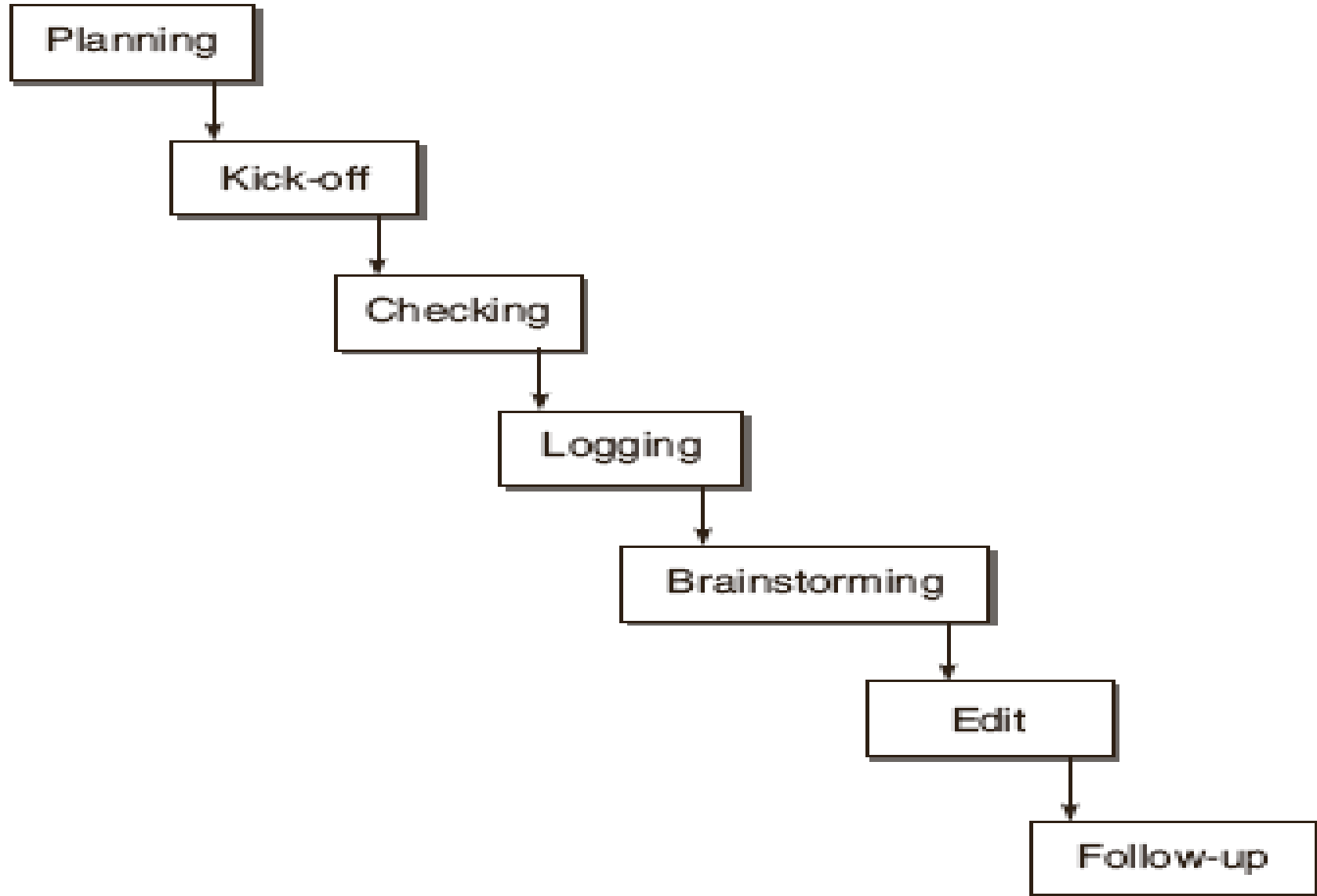
Formal Technical Asynchronous review method (FTArm)

- This process consists of following steps:
 - *Setup*: involves choosing the members and preparing the documents for an asynchronous process.
 - The documents prepared as a hyper text document.
 - *Orientation*: It is same as the overview step of the inspection process.
 - *Private review*: Each reviewer or inspector individually gives his comments on the document being inspected.
 - However, each inspector is unable to see the other inspector's comments.

Formal Technical Asynchronous review method (FTArm)

- *Public review:* All comments provided privately are made public and all inspectors are able to see each other's comments and put forward their suggestions.
- Based on this feedback and with consultation of the author, it is decided that there is a bug.
- *Consolidation:* the moderator analyses the result of private and public reviews and lists the findings and unresolved issues, if any.
- *Group meeting:* if required, any unresolved issues are discussed in this step. But decisions to conduct group meeting is taken in the previous step only by the moderator.
- *Conclusion:* The final report of the inspection process along with the analysis is produced by the moderator.

Gilb Inspection



Gilb Inspection

- Gilb and Graham defined this process.
- In this process defect detection is carried out by individual inspectors at their own level rather than in a group.
- Three different roles are defined in this type of inspection.
 - Leader: is responsible for planning and running the inspection.
 - Author of the document
 - Checker: is responsible for finding and reporting the defects in the document.

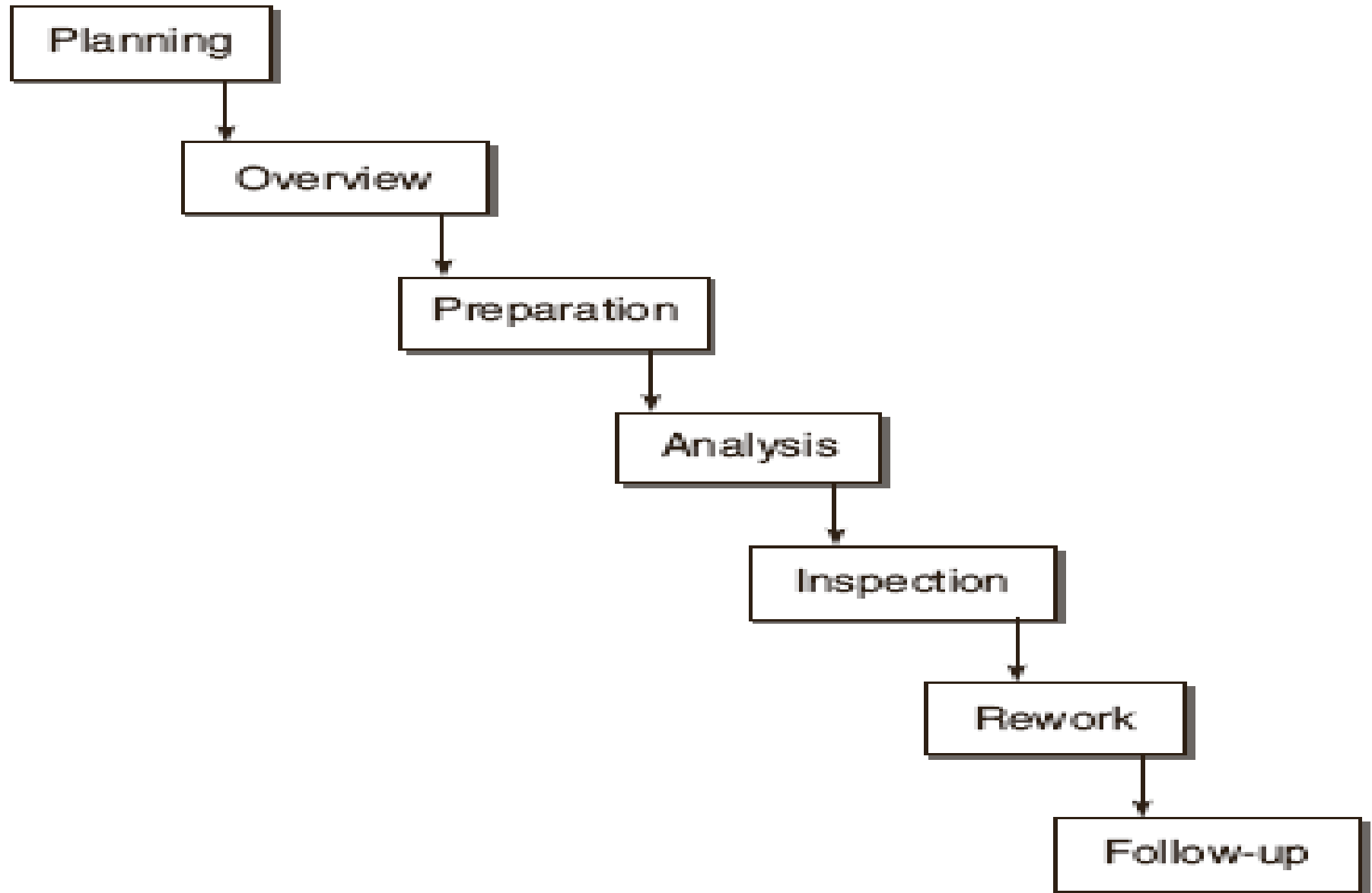
Gilb Inspection

- Entry: The document must pass through an entry criteria so that the inspection time is not wasted on a document which is fundamentally flawed.
- Planning: The leader determines the inspection participants and schedules the meeting.
- Kick off[begin]: The relevant documents are distributed, participants are assigned roles and briefed about the agenda of the meeting.
- Checking: Each checker works individually and finds defects.
- Logging: Potential defects are collected and logged.

Gilb Inspection

- Brain storm: In this stage, process improvement suggestions are recorded based on the reported bugs.
- Edit: After all the defects have been reported, the author takes the list and works accordingly.
- Follow-up: The leader ensures that the edit phase has been executed properly.
- Exit: The inspection must pass the exit criteria as fixed for the completion of the inspection process.

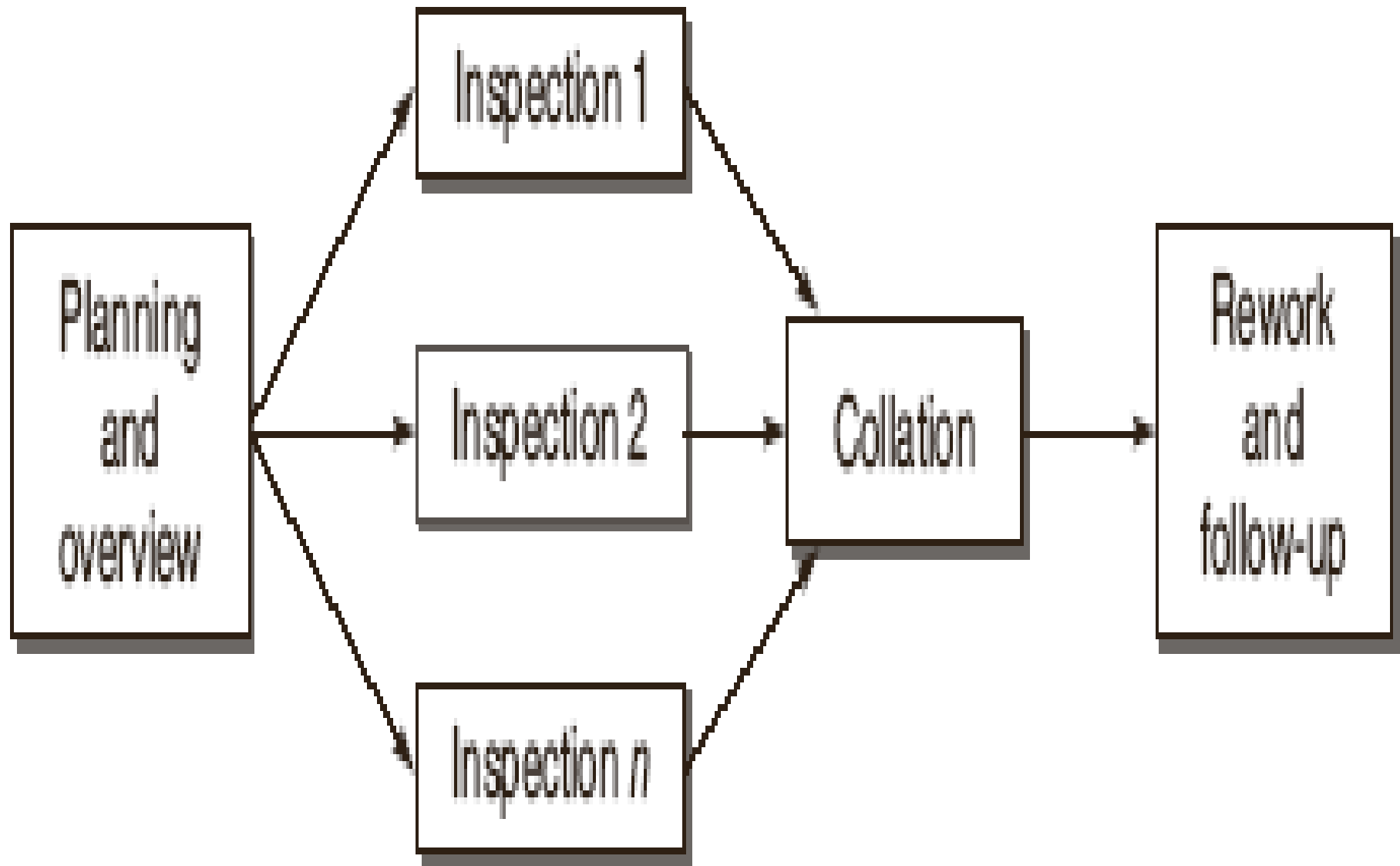
Humphrey's Inspection Process



Humphrey's Inspection Process

- It was described by Watts Humphrey.
- In this process, the preparation phase emphasizes on finding and logging of bugs, unlike Fagan inspection.
- Instead of familiarizing reviewers with documents and being prepared to put forward defects, they are required to produce a list of defects which are then analyzed.
- During analysis phase Producer arranges bugs according to their criticality and arrange them with their priority.
- In analysis phase, individual logs are analyzed and combined into a single list.
- The meeting is centered around this defect list, where producer addresses each defect in turn but may seek clarification from the reviewers.

N-Fold Inspection



N-Fold Inspection

- It is based on the idea that the effectiveness of the inspection process can be increased by replication of it.
- If we increase the number of teams inspecting item, the percentage of defects found may increase.
- Originally, this process was used for inspecting requirement specifications, but it can also be used for any phase.
- This process needs a coordinator who coordinates various teams, collects and collates the inspection data received by them.
- For this he also meets the moderator of every inspection team.

N-Fold Inspection

- This process consists of the following stages:
- *Planning and Overview*: the formal planning and overview stage.
 - It also includes the planning of how many teams will participate in the inspection process.
- *Inspection Stages*: There are many inspection processes, so every team may not choose the same inspection process. The team is free to adopt any process.
- *Collation phase*: The results from each inspection process are gathered, collated, and a master list of all detected defects is prepared.
- *Rework and follow-up*: this step is same as the tradition Fagan inspection process.

Phased Inspection

- Phased inspections are designed to verify the product in a particular domain. If we feel that in a particular area, we may encounter errors, then phased inspections are carried out.
- Only experts who have experience in that particular domain are called.
- This inspection process provides a chance to utilize human resources.
- There are two types of phases, as discussed below:
 - *Single Inspector*: in this phase, a rigorous checklist is used by a single inspector to verify whether the features specified are there in the item to be inspected.

Phased Inspection

- *Multiple Inspector:* The item cannot be verified just with the questions mentioned in the checklist.
 - There are many inspectors who are distributed the required documents for verification of an item.
- Each inspector examines this information and develops a list of questions of their own based on a particular feature.
- These questions are not in binary form as in single inspection.
- The item is then inspected individually by all the inspectors based on a self-developed checklist which is either application or domain specific.
- After individual checking by the inspectors, a reconciliation meeting is organized where inspectors compare their findings about the item.

Reading Techniques

- A reading technique can be defined as a series of steps or procedures to guide an inspector in acquiring a deep understanding of the inspected software product.
- A mechanism or strategy for the individual inspector to detect defects in the inspected product.
- **Ad-hoc Method:** there is no direction or guidelines provided for inspection.
 - The word ad hoc only refers to the fact that no technical support on how to detect defects in a software artifact is given to them.

Reading Techniques

- **Checklists:** A checklist is a list of items that focus the inspector's attention on specific topics such as common defects or organizational rules, while reviewing software documents.
- The checklists are in the form of general questions and are used by the inspection team members to verify the item.
- Checklists have some drawbacks:
 - The questions, being general in nature, are not sufficiently tailored to take into account a particular development environment.
 - Instructions about using a checklist is often missing.
 - There is a probability that some defects are not taken care of. It happens in the case of those type of defects which have not been previously detected.

Reading Techniques

- **Scenario based Reading**
- Checklists are general in nature and do not cover different types of bugs.
- It is based on scenarios, wherein inspectors are provided with more specific instructions than typical checklists.
- The following methods have been developed based on the criteria given by scenario-based reading.

Scenario based Reading

- **Perspective based Reading**
- Software item should be inspected from the perspective of different stakeholders.
- Inspectors of an inspection team have to check software quality as well as the software quality factors of a software artifact from different perspectives.
- For each perspective, either one or multiple scenarios are defined, consisting of repeatable activities an inspector has to perform, and the questions an inspector has to answer.

Scenario based Reading

- **Usage based Reading**
- This method given is applied in design inspections.
- Design documentation is inspected based on use cases, which are documented in requirements specification.
- Since use cases are the basis of inspection, the focus is on finding functional defects, which are relevant from the users' point of view.

Scenario based Reading

- **Abstraction driven Reading**
- This method is designed for code inspections.
- In this method, an inspector reads a sequence of statements in the code and abstracts the functions these statements compute.
- An inspector repeats this procedure until the final function of the inspected code artifact has been abstracted and can be compared to the specification.
- Thus, the inspector creates an abstraction level specification based on the code under inspection to ensure that the inspector has really understood the code.

Scenario based Reading

- **Task driven Reading**

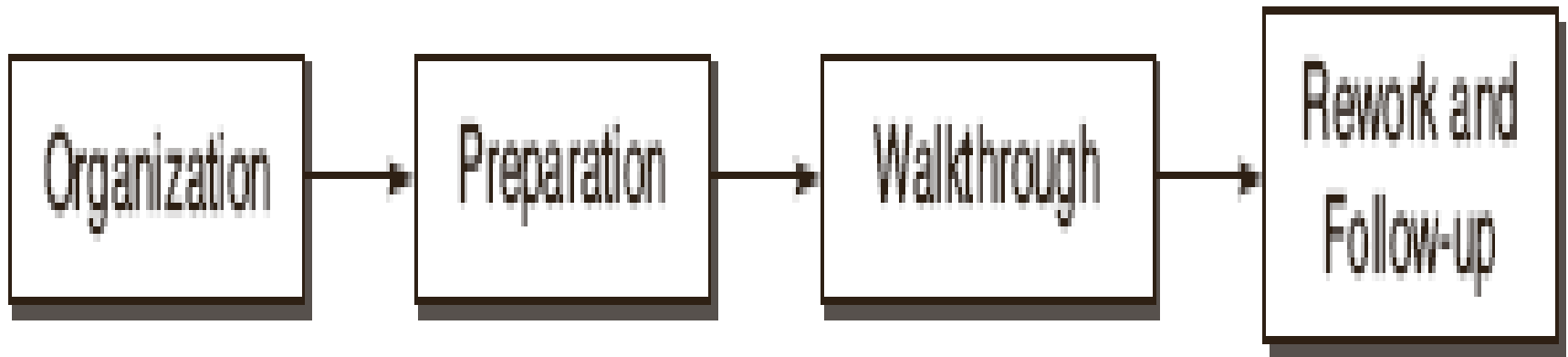
- This method is also for code inspections.
- In this method, the inspector has to create a data dictionary, a complete description of the logic and a cross-reference between the code and the specifications.

- **Function-point based Scenarios**

- This is based on scenarios for defect detection in requirements documents [103].
- The scenarios, designed around function-points are known as the Function Point Scenarios.
- A Function Point Scenario consists of questions and directs the focus of an inspector to a specific function-point item within the inspected requirements document.

Structured Walkthroughs

- The very common term used in the literature for static testing is Inspection but it is for very formal process.
- If you want to go for a less formal having no bars of organized Meeting, then walkthroughs are a good option.



Structured Walkthrough

1. The idea of structured walkthrough was proposed by yourdon.
2. It is less formal and less rigorous technique as compared to inspection.
3. A typical structured walkthrough team consists of following:
 1. Coordinator
 2. Presenter/Developer (Optional)
 3. Scribe/Recorder
 4. Reviewer/Tester
 5. Maintenance Oracle
 6. Standards Bearer
 7. User Representative/Accreditation Agent

Structured Walkthrough

1. Walkthroughs differ significantly from inspection process.
2. A walkthrough is less formal, has fewer steps and does not use a checklist to guide or a written report to document the team's work.
3. The person designated as a tester comes to the meeting armed with a small set of paper test cases- representative sets of inputs and expected outputs for the program or module.
4. During the meeting each test case is mentally executed.
5. That is, the test data are walked through the logic of the program.
6. The state of program is monitored on a paper or any other presentation media.

Technical Reviews

- A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process carried out according to stated objectives.
- A review is similar to an inspection or walkthrough, except that the review team also includes management.
- Therefore, it is considered a higher-level technique than inspection or walkthrough.

Technical Reviews

- A technical review team is generally comprised of management-level representatives of the User and Project Management.
- Review agendas should focus less on technical issues and more on oversight than an inspection.
- The purpose is to evaluate the system relative to specifications and standards, recording defects and deficiencies.

Technical Reviews

- The moderator should gather and distribute the documentation to all team members for examination before the review.
- He should also prepare set of indicators to measure the following points.
- Appropriateness of the problem definition and requirements.
- Adequacy of all underlying assumptions
- Adherence to standards
- Consistency
- Completeness
- Documentation
- The moderator may also prepare a checklist to help the team focus on the key points. The result of the review should be a document recording the events of meeting, deficiencies identified, and review team recommendations.