Chapter 5

5.0		**Testing and Maintenance**
	5.1	Testing Concepts: Purpose of Software Testing, Testing Principles, Goals of Testing, Testing aspects: Requirements, Test Scenarios, Test cases, Test scripts/procedures,
	5.2	Strategies for Software Testing, Testing Activities: Planning Verification and Validation, Software Inspections,FTR
	5.3	Levels of Testing : unit testing, integration testing, regression testing, product testing, acceptance testing and White-Box Testing
	5.4	Black-Box Testing: Test Case Design Criteria, Requirement Based Testing, Boundary Value Analysis, Equivalence Partitioning
	5.5	Object Oriented Testing: Review of OOA and OOD models, class testing, integration testing, validation testing
	5.6	Reverse and re-engineering, types of maintenance

Purpose of Software Testing

- Testing is the **process of analyzing a system** or system component **to detect the differences** between specified (**required**) and observed (**existing**) behavior.

- Testing is a **set of activities** that can be **planned in advance** and conducted systematically

Principles of Testing

1. All tests should be traceable to customer requirements

- testing proves the presence of errors
- it does not verify that no more bugs exist
- Testing is not a prove that the system is free of errors.
- most severe defects are those that cause the program to fail to meet its requirements.

2. Exhaustive testing is not possible

- An exhaustive test which considers all possible input parameters, their combinations and different pre-conditions can not be accomplished.

Principles of Testing

- Test are always spot tests , hence efforts must be managed.

3. Tests should be planned long before testing begin

4. Test early and regularly

- Testing activities should begin as early as possible within the software life cycle.
- Early testing helps detecting errors at an early stage of the development process which simplifies error correction

Principles of Testing

5. Accumulation of errors

- There is no equal distribution of errors within one test object.
- The place where one error occurs, it’s likely to find some more. The testing process must be flexible and respond to this behavior.

6. Fading effectiveness

- The effectiveness of software testing fades over time.
- If test-cases are only repeated, they do not expose new errors.

Principles of Testing

- Errors, remaining within untested functions may not be discovered.
- to prevent this effect, test-cases must be altered and reworked time by time.

7. Testing depends on context

- No two systems are the same and therefore can not be tested the same way.

Principles of Testing

8. False conclusion: no errors equals usable system

- Error detection and error fixing does not guarantee a usable system matching the users expectations.
- Early integration of users and rapid prototyping prevents unhappy clients and discussions.

9. The Pareto principle applies to software testing

10. Testing should begin "in the small" and progress toward testing "in the large."

11. To be most effective, testing should be conducted by an independent third party.

Goals of Testing

- The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements.

- The first goal leads to **validation testing**, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.

Goals of Testing

2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification.**(Verification)**

- The second goal leads to defect testing, where the test cases are designed to expose defects.

- The set of activities that ensure that software correctly implements a specific function or algorithm

Testing Concepts

- A **test component** is a part of the system that can be isolated for testing.
- A **fault**, also called **bug or defect**, is a design or coding mistake that may cause abnormal component behavior.
- An **erroneous state** is an indication of a fault during the execution of the system.
- A **failure** is a deviation between the specification and the actual behavior. A **failure is triggered** by one or more **erroneous states**.

Requirements of Testing

1. Testability

- Software testability is simply how easily [a computer program] can be tested.
- There are certainly metrics that could be used to measure testability.

2. Operability

- "The better it works, the more efficiently it can be tested.“
- The system has few bugs (bugs add analysis and reporting overhead to the test process).

Requirements of Testing

3. Observability

- **"What you see is what you test."**
- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

Requirements of Testing

4. Controllability.

- "The better we can control the software, the more the testing can

be automated and optimized."

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the

test engineer.

- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

Requirements of Testing

5. Decomposability

- "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."
- The software system is built from independent modules.
- Software modules can be tested independently.

Requirements of Testing

6. Simplicity.

- **"The less there is to test, the more quickly we can test it.“**

- **Functional simplicity** (e.g., the feature set is the minimum necessary to meet requirements).

Requirements of Testing

- **Structural simplicity** (e.g., architecture is modularized to limit the propagation of faults).

- **Code simplicity** (e.g., a coding standard is adopted for ease of inspection and maintenance).

Requirements of Testing

7. Stability

- **"The fewer the changes, the fewer the disruptions to testing."**
- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.

Requirements of Testing

8. Understandability.

- **"The more information we have, the smarter we will test."**
- The design is well understood.
- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated.
- Technical documentation is instantly accessible.
- Technical documentation is well organized.
- Technical documentation is specific and detailed.
- Technical documentation is accurate.

Test Case

- A test case is a set of **conditions or variables under which a tester will determine** whether a **system under test** satisfies requirements or works correctly.

- The process of developing test cases can also help find problems in the requirements or design of an application.

Test Case

- A test case has five attributes:

1. The **name** of the test case allows the tester to distinguish between different test cases.

- **For example:** testing a use case Deposit(), call the test case Test_Deposit.

2. The **location** attribute describes where the test case can be found.

- path name or the URL to the executable of the test program and its inputs.

Test Case

3. **Input (data)** describes the set of input data or commands to be entered by the actor of the test case.

- The test data, or links to the test data, that are to be used while conducting the test.

4. The expected behavior is described by the **oracle** attribute.

5. The **log** is a set of time-stamped correlations of the observed behavior with the expected behavior for various test runs.

Strategies for Software Testing

- Testing is a **set of activities that can be planned in advance** and conducted systematically.

- A number of **software testing strategies provide** the **software developer** with a **template** for testing and have the following generic characteristics:

Strategies for Software Testing

1. Testing begins at the **component level** and works "**outward**" toward the **integration** of the entire computer-based **system**.

2. **Different testing techniques** are appropriate at **different points in time**.

3. Testing is **conducted by the developer** of the software and (**for large projects**) an **independent test group**.

Strategies for Software Testing

4. **Testing and debugging are different activities**, but debugging must be accommodated in any testing strategy.

- A testing strategy must implement low level and high level tests

- A strategy must provide guidance for the practitioner and a set of milestones for the manager.

Strategic Issues

- best strategy will fail if a series of overriding issues are not addressed
- Following are the strategic issues to be considered:

1. ***Specify product requirements in a quantifiable manner long before testing commences.***

- objective of testing is to find errors
- a good testing strategy also assesses other quality characteristics as well.
- Measurable requirements to be specified for unambiguous results

Strategic Issues

2. State testing objectives explicitly

- specific objectives of testing should be stated in measurable terms.

- For example, test effectiveness, test coverage, the cost to find and fix defects, frequency of occurrence, and test work-hours should be stated within the test plan.

Strategic Issues

3. Understand the users of the software and develop a profile for each user category.

- Use cases that describe the **interaction scenario for each class of user** can **reduce overall testing effort** by focusing testing on actual use of the product.

Strategic Issues

4. Develop a testing plan that emphasizes "rapid cycle testing."

- Is mindset and skill set to carry out testing more quickly , less expensive and best results.

- The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Strategic Issues

5. Build "robust" software that is designed to test itself.

- Software should be capable of diagnosing certain classes of errors.

- the design should accommodate automated testing and regression testing.

Strategic Issues

6. Use effective technical reviews as a filter prior to testing

- Technical reviews can be as effective as testing in uncovering errors.

- reviews can reduce the amount of testing effort that is required to produce high quality software.

Strategic Issues

7. Develop a continuous improvement approach for the testing process.

- The test strategy should be measured.

- The metrics collected during testing should be used as part of a statistical process control approach for software testing.

Verification and Validation

- **Verification** refers to the set of activities that ensure that software correctly implements a specific function(**algorithm**).

- **Validation** refers to a different set of activities that ensure that the software that has been built is traceable to **customer requirements**.

- Verification and validation encompasses a wide array of SQA activities.

Verification and Validation

- SQA includes following activities:

- formal technical reviews
- quality and configuration audits
- performance monitoring
- Simulation
- feasibility study
- documentation review
- database review
- algorithm analysis
- development testing
- qualification testing
- installation testing

Verification and Validation

- Testing defines principles for quality assurance and error detection.

Formal Technical Reviews (FTR)

- Formal Technical Review (FTR) is a **software quality control activity** performed by software engineers (and others).

The objectives of an FTR are:

(1) to **uncover errors in function, logic, or implementation** for any representation of the software

(2) To **verify** that the **software** under review **meets its requirements**

(3) to **ensure** that the **software** has been **represented according to predefined standards**

(4) to **achieve software** that is developed in a **uniform manner**

(5) to make **projects more manageable**.

Formal Technical Reviews (FTR)

- the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.

- The FTR is actually a class of reviews that includes ***walkthroughs*** **and** ***inspections.***

- **FTR is conducted as a meeting** and will be successful only if it is properly planned, controlled, and attended.

Formal Technical Reviews (FTR)

- **The Review Meeting**
- Between **three and five people** (typically) should be involved in the review.

- **Advance preparation** should occur but should require no more than two hours of work for each person.

- The **duration** of the review meeting should be **less than two hours**.

Formal Technical Reviews (FTR)

- The review meeting is attended by the review leader, all reviewers, and the producer.
- One of the reviewers takes on the role of a *recorder*
- The producer proceeds to “walk through”

Formal Technical Reviews (FTR)

- At the end of the review, all attendees of the FTR must decide whether to:

(1) Accept the product without further modification

(2) reject the product due to severe errors

(3) accept the product with minor revisions

Formal Technical Reviews (FTR)

- **Review Reporting and Record Keeping**
- During the FTR, a reviewer (the recorder) records all issues that have been raised.
- review issues list produced.
- What was reviewed?
- Who reviewed it?
- What were the findings and conclusions?

Formal Technical Reviews (FTR)

- **Review Guidelines**
- The following represents a minimum set of guidelines for formal technical reviews:

1. **Review the product, not the producer.**
2. **Set an agenda and maintain it.**

An FTR must be kept on track and on schedule.

3. **Limit debate and rebuttal.**

When an issue is raised by a reviewer, there may not be universal agreement on its impact.

Formal Technical Reviews (FTR)

4. Enunciate problem areas, but don’t attempt to solve every problem noted.

A review is not a problem-solving session.

5. Take written notes.

It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.

Formal Technical Reviews (FTR)

6. Limit the number of participants and insist upon advance preparation

Keep the number of people involved to the necessary minimum.

7. Develop a checklist for each product that is likely to be reviewed.

A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues.

Formal Technical Reviews (FTR)

8. Allocate resources and schedule time for FTRs.

For reviews to be effective, they should be scheduled as tasks during the software process.

9. Conduct meaningful training for all reviewers

Levels of Testing (Unit Testing)

1. Unit Testing

- **Unit Testing** is a level of software testing where **individual units/ components** of a software are **tested.**

- It is concerned with **functional correctness of the standalone modules.**

- Important control paths are tested to **uncover errors within the boundary** of the module.

Levels of Testing (Unit Testing)

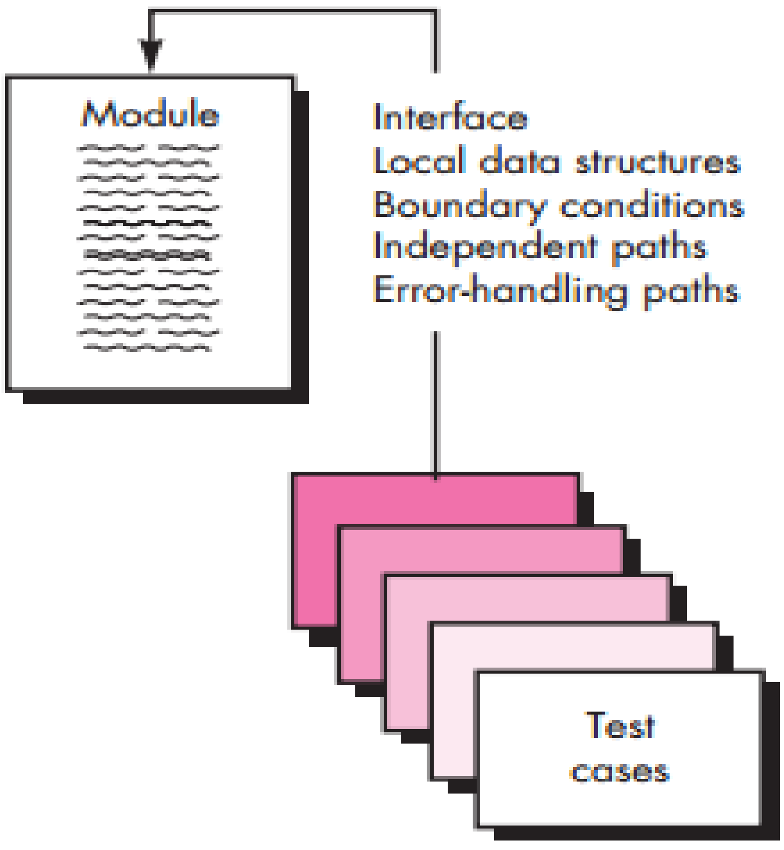
- The **relative complexity of tests and the errors those tests uncover** is limited by the constrained **scope** established for unit testing.

- focuses on the **internal processing logic and data structures**

- can be conducted in **parallel for multiple components**

Levels of Testing (Unit Testing)

- Unit Test



Levels of Testing (Unit Testing)

- The module interface is tested **to ensure that information properly flows into and out** of the program unit under test.
- All **independent paths through control structure exercised** to ensure all statements executed.
- Boundary conditions are tested to limit or restrict processing.
- all error-handling paths are tested.

Levels of Testing (Unit Testing)

- **Data flow** across a component interface is **tested before** any other testing.

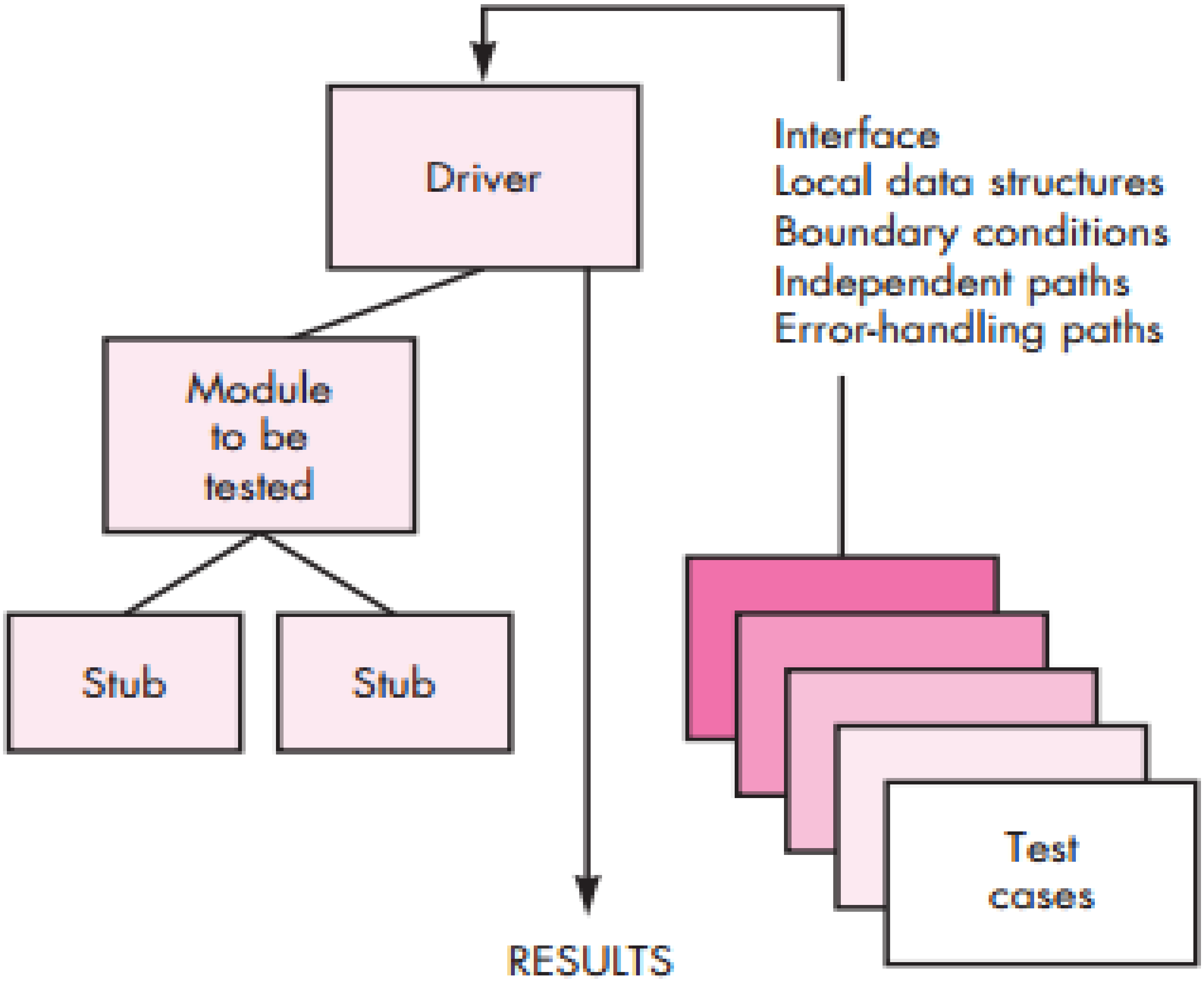
- Test cases should be designed to uncover errors due to **improper computation , comparison and data flow.**

- Boundary testing is one of the most important unit testing tasks.

Levels of Testing (Unit Testing)

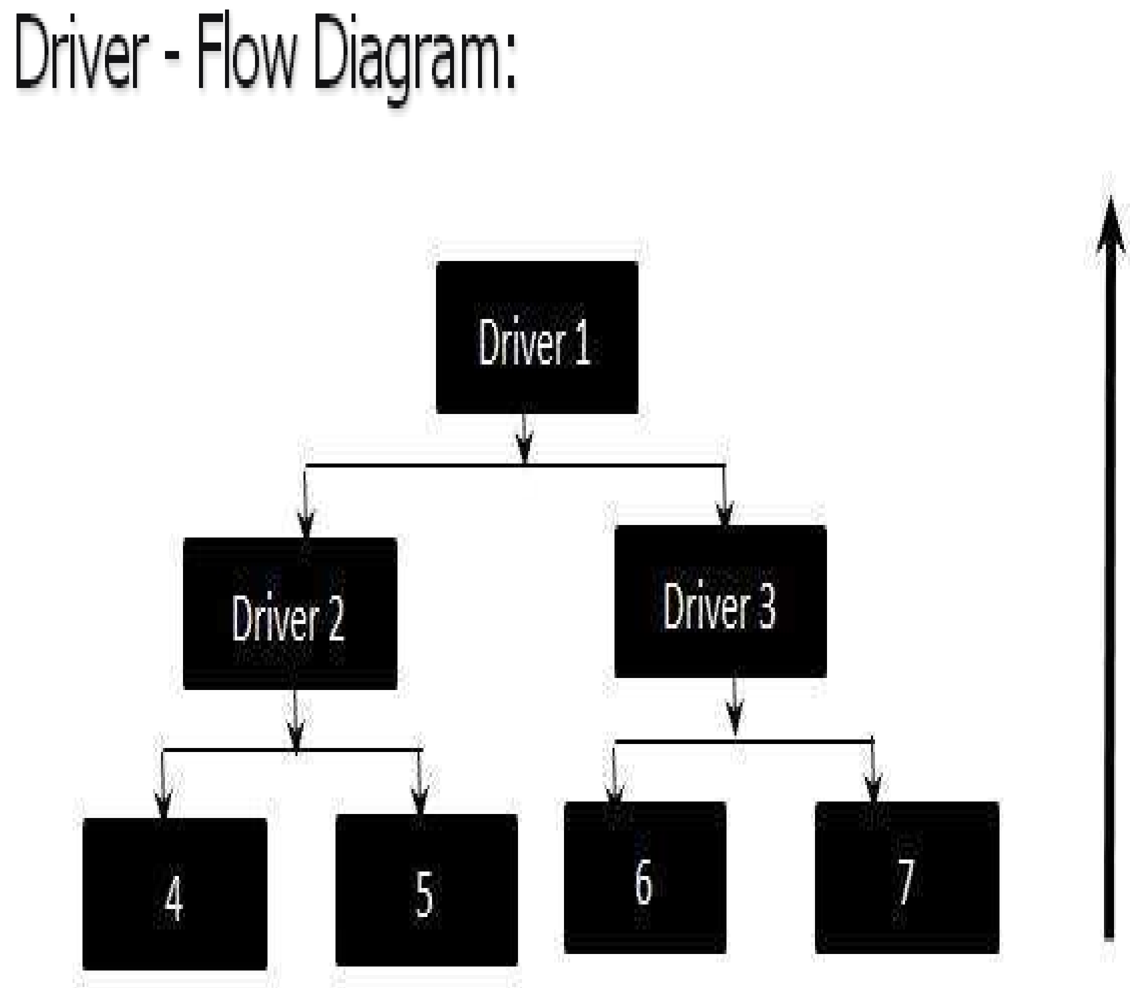
- **errors often occur** when the **maximum or minimum allowable value** is encountered (boundary testing)

- Test cases that exercise **data structure, control flow, and data values just below, at, and just above maxima and minima** are very likely to uncover errors.


Driver
Interface
Local data structures
Boundary conditions
Independent paths
Error-handling paths
Module to be tested
Stub
Stub
Test cases
RESULTS

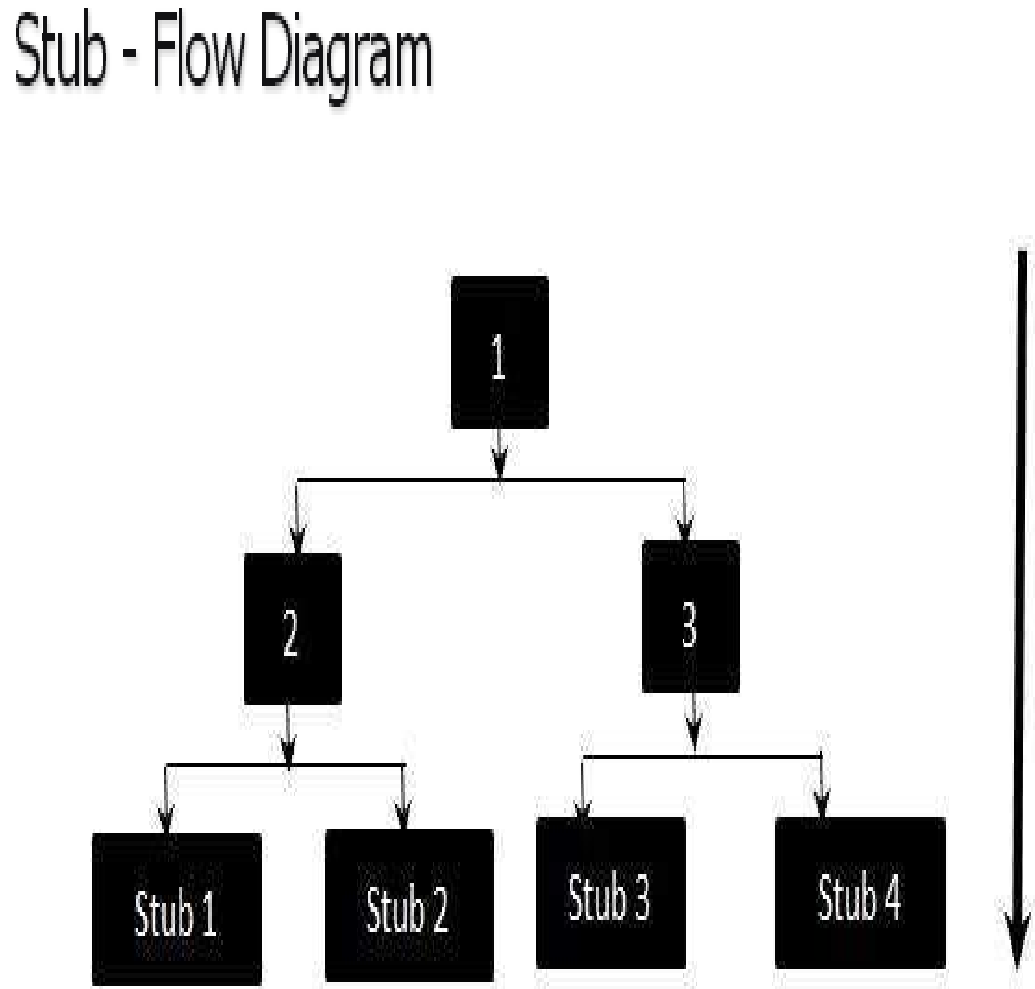
Levels of Testing (Unit Testing)

Drivers are considered as the dummy modules that always **simulate the high level modules.**



Levels of Testing (Unit Testing)

Stubs are considered as the dummy modules that always **simulate the low level modules.**



Levels of Testing (Unit Testing)

- Drivers and stubs represent testing “overhead.”
- both are software that must be written but that is not delivered with the final software product.
- If drivers and stubs are kept simple, actual overhead is relatively low.
- Unit testing is simplified when a component with high cohesion is designed.

Levels of Testing (Integration Testing)

- **"If they all work individually, why do you doubt that they'll work when we put them together?"**
- Why integration testing?
- Data can be lost across an interface
- one component can have an adverse effect on another
- Sub functions, when combined, may not produce the desired major function
- global data structures can present problems
- **"putting them together"—interfacing.**

Levels of Testing (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.**

- take unit-tested components and build a program structure by design.

- "**big bang**" approach to integration is a lazy strategy that might result into failure – endless loop

Levels of Testing (Integration Testing)

- **Incremental integration** - The program is constructed and tested in **small increments**, **easier to isolate and correct errors**.

- interfaces are more likely to be tested completely

- Incremental Integration strategies:

1. Top – down integration.
2. Bottom – up integration.

Levels of Testing (Integration Testing)

- **Top Down** is an approach to Integration Testing where top level units are tested first and lower level units are tested step by step after that.

- **depth-first integration** integrates all components on a major control path of the program structure.

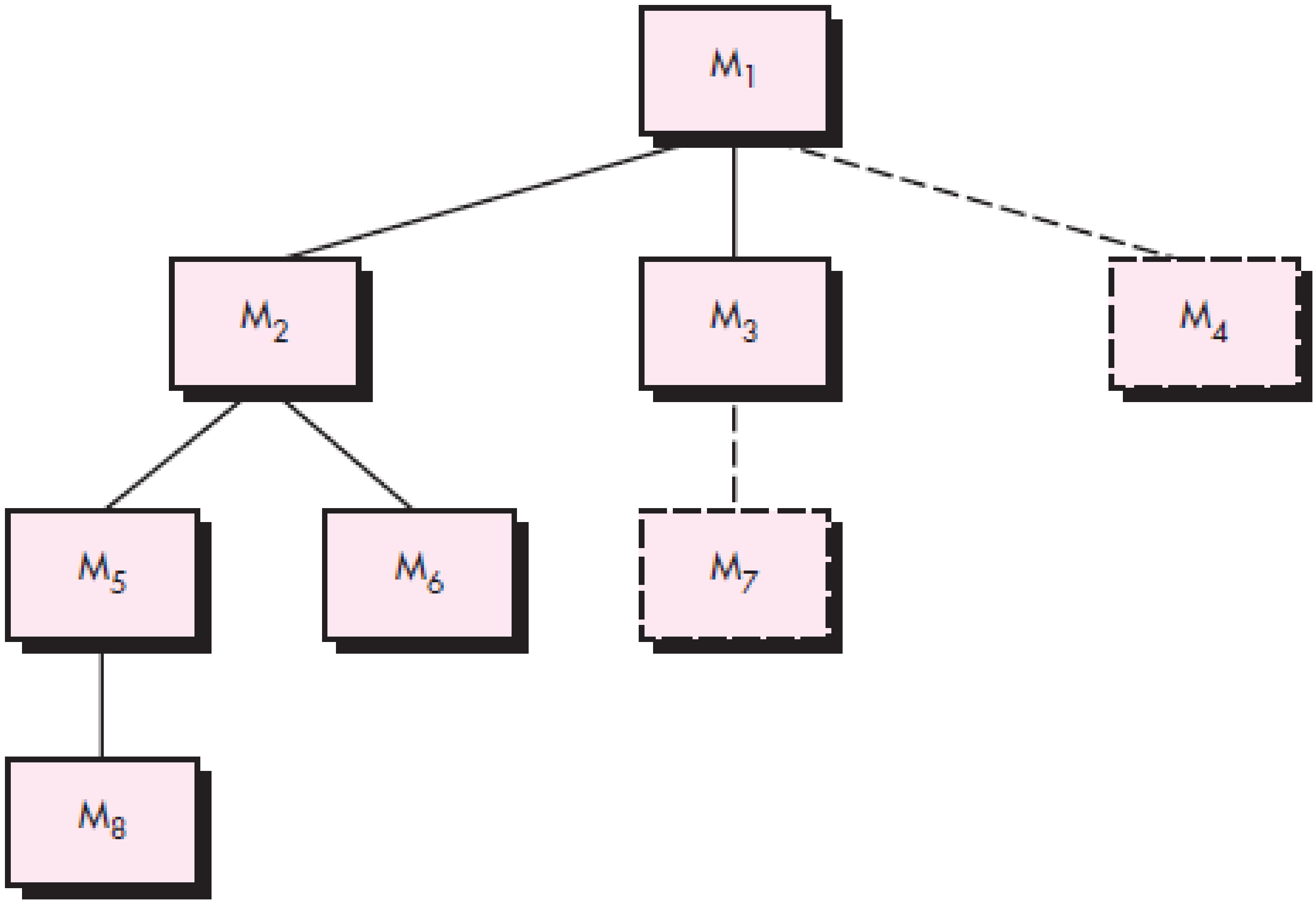
- For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or M6 would be integrated.

Levels of Testing (Integration Testing)

- **Breadth-first integration** incorporates all components directly subordinate at each level, moving across the structure horizontally.

- For example: components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

Top – down integration.



Levels of Testing (Integration Testing)

- **Steps of Integration:**

1. The **main control module is used as a test driver** and **stubs are substituted** for all components directly subordinate to the main control module.
2. Depending on the **integration approach selected** (i.e., depth or breadth first), **subordinate stubs** are **replaced** one at a time with actual components.
3. **Tests are conducted** as each component is integrated.
4. On completion of each set of tests, **another stub is replaced with the real component.**
5. **Regression testing may be conducted** to ensure that new errors have not been introduced.

Levels of Testing (Integration Testing)

- The top-down integration strategy verifies major control or decision points early in the test process.

- Problem with top down strategy - when processing at low levels in the hierarchy is required to adequately test upper levels.

- Bottom – up integration

Levels of Testing (Integration Testing)

- **Bottom Up** is an approach to Integration testing where **bottom level units are tested first** and **upper level units step by step after that.**

- Because components are **integrated** from the **bottom up**, need for **stubs is eliminated.**

- **The whole program does not exist until the last module is integrated.**

Levels of Testing (Integration Testing)

- **Steps of Integration:**

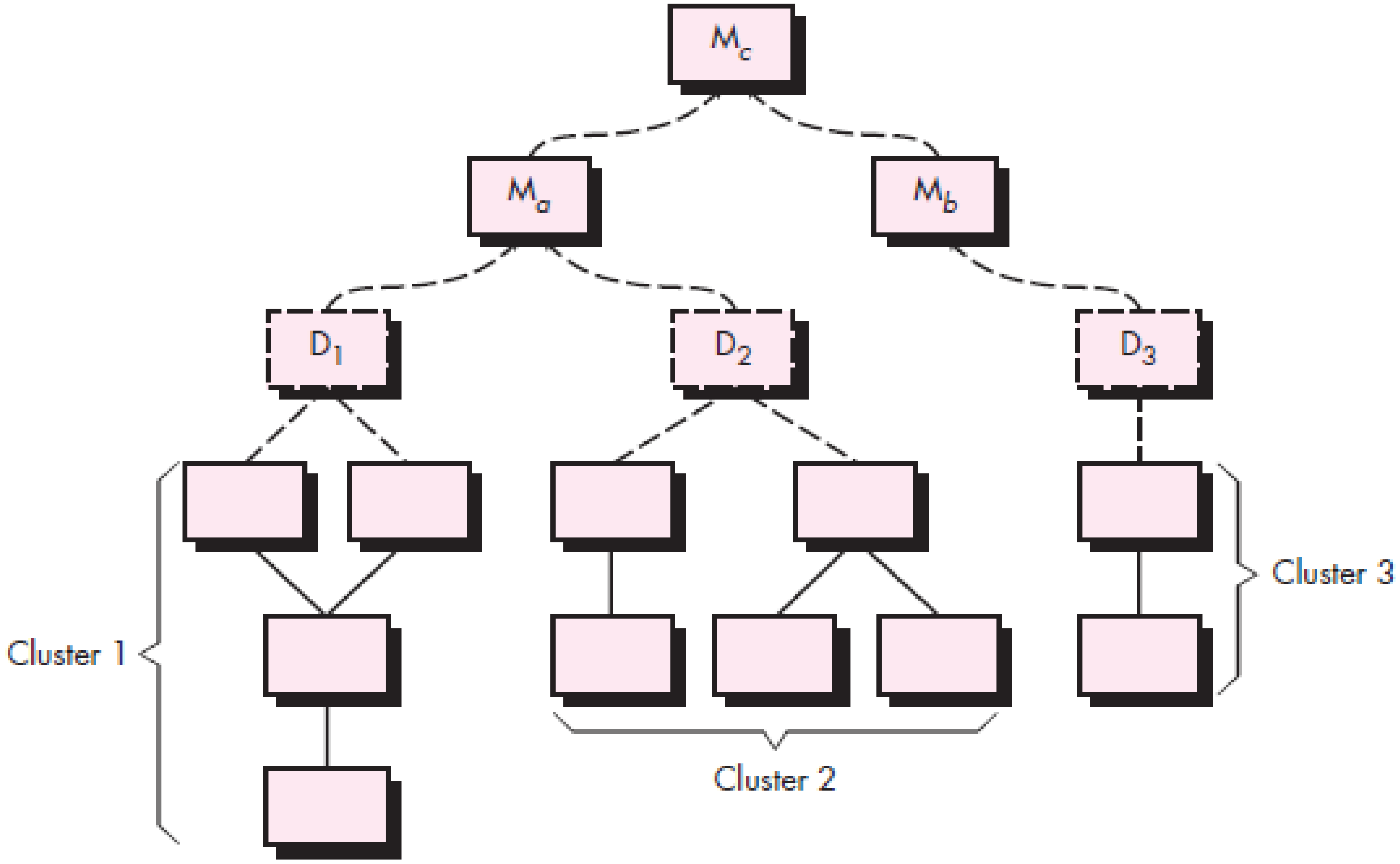
1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.

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.

Bottom Up Integration



Levels of Testing (Regression Testing)

- Regression testing is a type of software testing that **intends to ensure that changes** (enhancements or defect fixes) **to the software have not adversely affected it**.

- **new module is added** as part of integration testing, the **software changes**.

- New data flow paths, new I/O, and new control logic.

Levels of Testing (Regression Testing)

- **changes may cause problems** with functions that previously worked flawlessly.

- ***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.**

- Discovery of error and then correction changes some aspect of software configuration.

Levels of Testing (Regression Testing)

- Regression testing ensures **additional errors are not introduced.**

- Regression testing may be conducted **manually or by automated playback capture tools.**

- Effective Regression testing

Levels of Testing (Regression Testing)

- The regression test suite contains three different classes of test cases:

1. A representative sample of tests that will **exercise all software functions.**

2. **Additional tests** that focus on **software functions** that are **likely** to be **affected by the change.**

3. Tests that focus on the **software components** that have been **changed**.

Levels of Testing (Acceptance Testing)

- **Acceptance testing**, a **testing** technique performed to determine whether or not the **software** system has **met** the **requirement specifications.**

- The main purpose of this test is to evaluate the **system's business requirements and verify delivery to end users.**

- Usually, Black Box Testing method is used in Acceptance Testing.

Levels of Testing (Acceptance Testing)

- ***benchmark test***, the client prepares a set of test cases that represent typical conditions under which the system should operate.

- ***competitor testing,*** the new system is tested against an existing system or competitor product.

- ***shadow testing***, a form of comparison testing, the new

and the legacy systems are run in parallel and their outputs are compared.

Levels of Testing (Acceptance Testing)

- After acceptance testing, **the client reports to the project manager which requirements are not satisfied.**
- Acceptance testing also gives the opportunity for a **dialog** between the developers and client.
- If **requirements must be changed** form the basis for **another iteration** of the software life-cycle process.
- If the customer is satisfied, the system is accepted

Levels of Testing (White Box Testing)

- White Box Testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

Levels of Testing (White Box Testing)

- Using white-box testing methods, you can derive test cases that

(1) all independent paths within a module have been exercised

(2) exercise **all logical decisions** on their true and false sides

(3) Execute **all loops at their boundaries** and within their **operational bounds**

(4) Exercise **internal data structures** to ensure their validity.

Levels of Testing (White Box Testing)

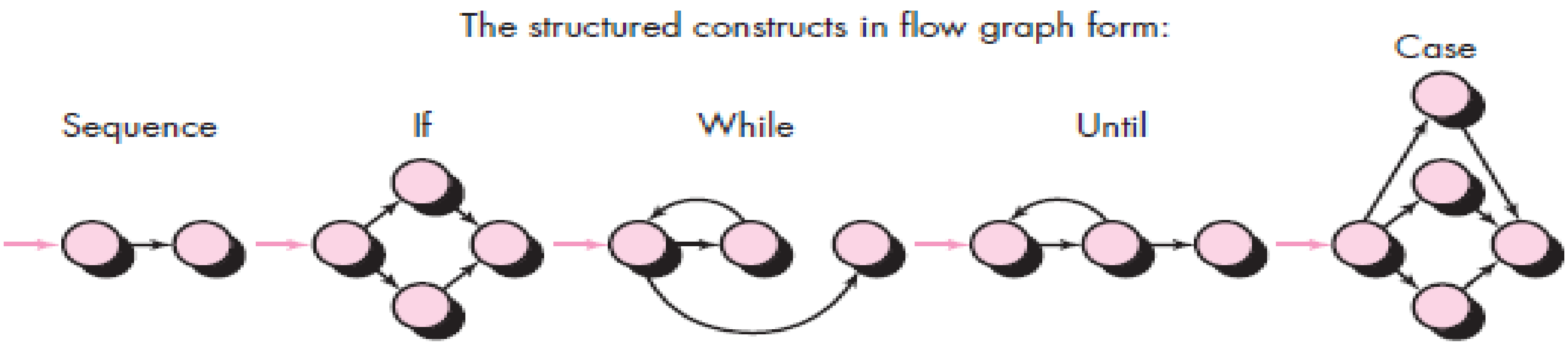
- **Basis path testing** is a white-box testing technique.

- The basis path method – complexities , execution paths(data flow), independent paths and procedural design.

- Test cases derived guaranteed to execute every statement in the program

Levels of Testing (White Box Testing)

- **Flow Graph Notation**



- each circle, called a ***flow graph node,*** represents one or more procedural statements.

- ***edges* or *links*,** represent flow of control and are analogous to flowchart arrows.

Levels of Testing (White Box Testing)

- Areas bounded by edges and nodes are called ***regions.***

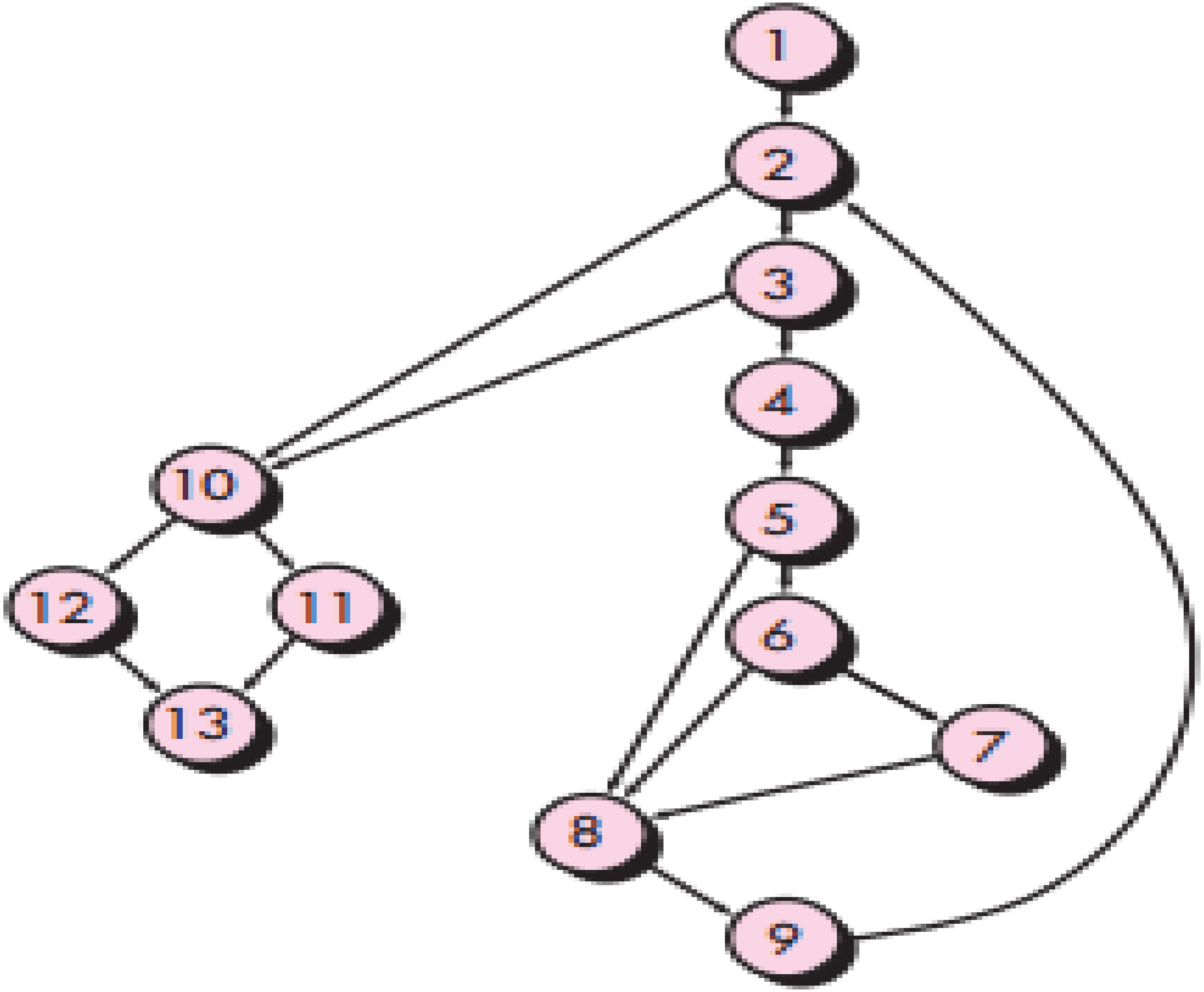
- Each node that contains a condition is called a ***predicate node.***

- An ***independent path*** is any path through the program that introduces at least **one new set of processing statements or a new condition**.

Levels of Testing (White Box Testing)

- ***Cyclomatic complexity*** is a software metric that provides a quantitative measure of the logical complexity of a program.

- Cyclomatic complexity **defines number of independent paths** which can be further used in development of test cases.


1
2
3
4
5
6
7
8
9
10
11
12
13

Levels of Testing (White Box Testing)

- **Complexity is computed in one of three ways:**

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

$$V(G) \quad E - N + 2$$

where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

$$V(G) \quad P + 1$$

where P is the number of predicate nodes contained in the flow graph G.

Levels of Testing (White Box Testing)

- **white-box test design technique:** Procedure to derive and/or **select test cases** based on an **analysis of the internal structure of a component or system.**

- **Deriving Test Cases**

1. **Using the design or code as a foundation, draw a corresponding flow graph.**

Levels of Testing (White Box Testing)

2. Determine the cyclomatic complexity of the resultant flow graph.

3. Determine a set of linearly independent paths.

The value of $V(G)$ provides the upper bound on the number of linearly independent paths.

Levels of Testing (White Box Testing)

4. Prepare test cases that will force execution of each path in the set.

- Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
- Each test case is executed and compared to expected results.
- be sure that all statements in the program have been executed at least once.

Levels of Testing (White Box Testing)

- A data structure, called a ***graph matrix,*** can be quite useful for developing a **software tool that assists in basis path testing.**

Levels of Testing (Black Box Testing)

- ***Black-box testing***, also called ***behavioral testing,*** focuses on the **functional requirements of the software.**

- exercise all functional requirements for a program

- **Black-box testing attempts to find errors in the following categories:**

(1) incorrect or missing functions

(2) interface errors

(3) errors in data structures or external database access

(4) behavior or performance errors

(5) initialization and termination errors.

Levels of Testing (Black Box Testing)

- Black-box testing is not an alternative to white-box techniques.

- it is a complementary approach that is likely to uncover a different class of errors

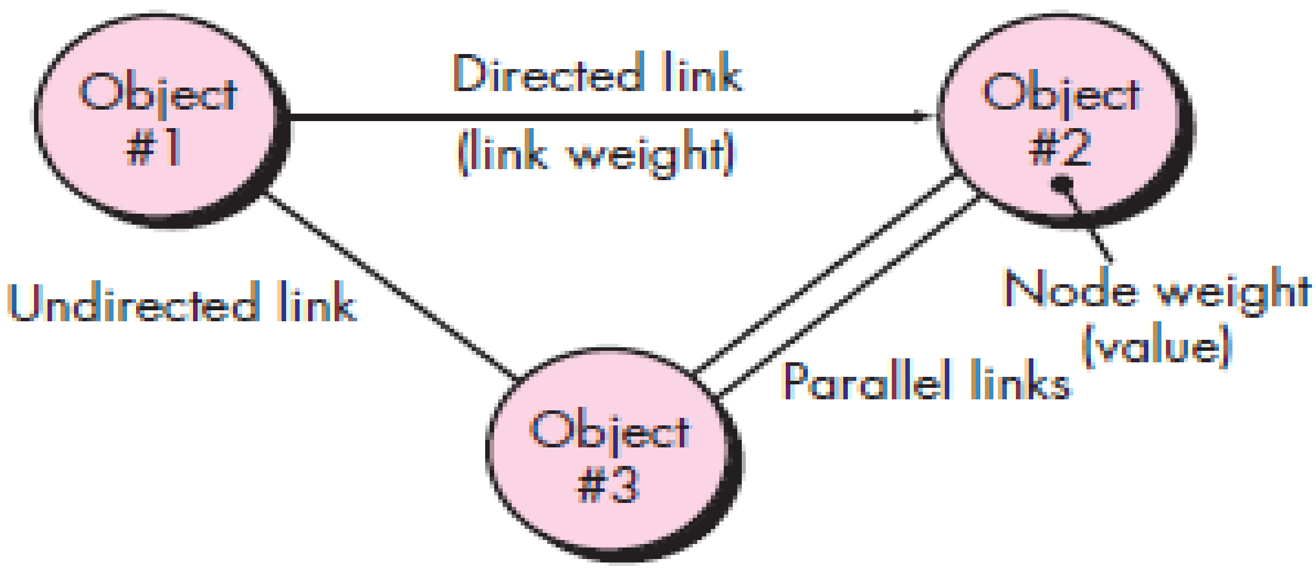
- **Criteria for testing**
- Identifying classes or errors.
- additional test cases that must be designed to achieve reasonable testing.

Graph-Based Testing Methods

- **understand the objects** that are modeled in software and the **relationships that connect these objects.**
- Software testing begins by **creating a graph of important objects and their relationships**
- define a series of tests that **verify all objects have the expected relationship to one another.**
- tests that will cover the graph - each object and relationship is exercised and errors are uncovered.

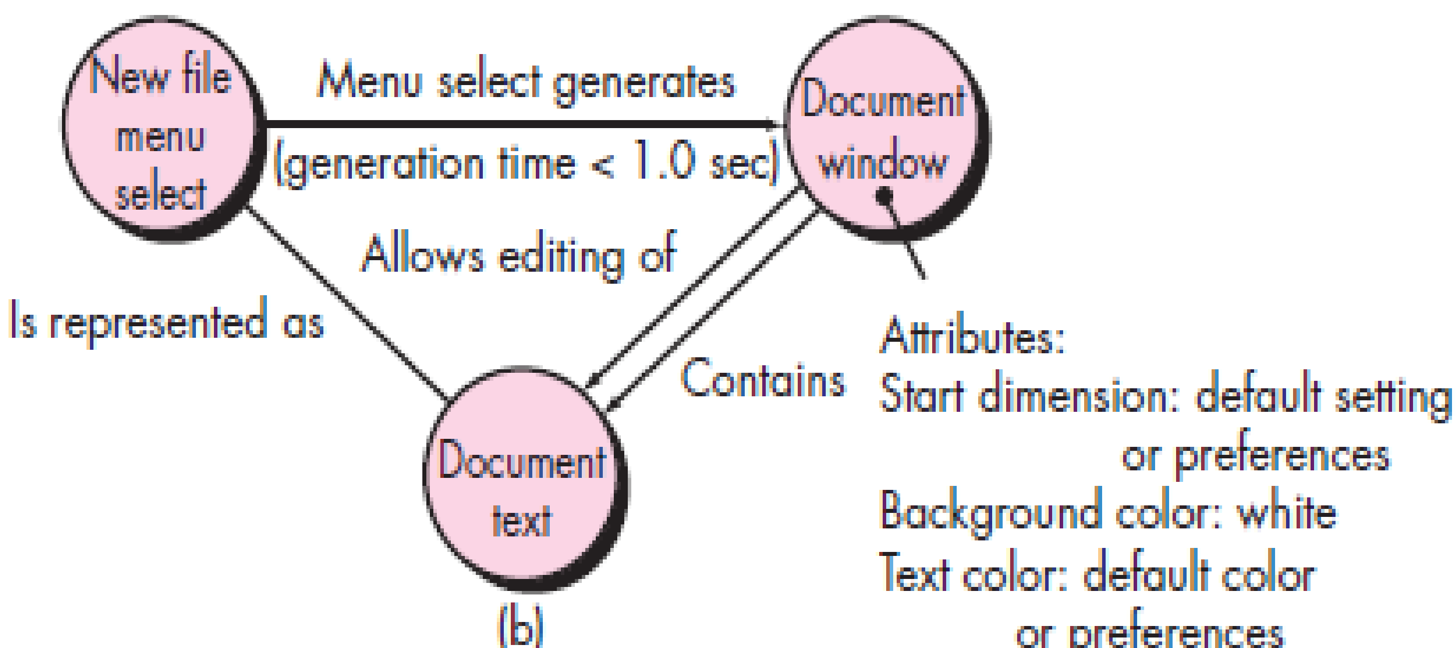
Graph-Based Testing Methods

- Graph notation



Graph-Based Testing Methods

- Example



Graph-Based Testing Methods

- test cases are designed in an attempt to **find errors in any of the relationships.**

- number of behavioral testing methods that can make use of graphs:

1. Transaction flow modeling

Example: airline reservation with validation

Data flow diagram can be used in creating graphs

Graph-Based Testing Methods

2. Finite state modeling

- different user observable states of the software

- **Example: order-information is verified during** ***inventory-*** *availability look-up and is followed by* ***customer-billing-information***

- **State transition diagram** can be used in creating graphs

Graph-Based Testing Methods

3. Data flow modeling

- transformations that occur to translate one data object into another.

4. Timing modeling

sequential connections between objects

specify the required execution times as the program executes

Equivalence Partitioning

- Equivalence partitioning is a software testing technique that **divides the input and/or output data of a software unit into partitions** of data **from which test cases can be derived**.

- The equivalence partitions are **usually derived from the requirements specification for input.**

- Test cases are designed to **cover each partition at least once.**

Equivalence Partitioning

- Equivalence partitioning technique **uncovers classes of errors.**

- An equivalence class represents a set of **valid or invalid states for input conditions.**

- Typically, an input condition is either a specific **numeric value, a range of values, a set of related values, or a Boolean condition.**

Equivalence Partitioning

- **Guidelines for equivalence classes:**

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

Equivalence Partitioning

- Test cases are selected so that the **largest number of attributes of an equivalence class are exercised at once.**
- **For example, a savings account in a bank has a different rate of interest depending on the balance in the account.**

Invalid partition	Valid (for 3% interest)	Valid (for 5%)	Valid (for 7%)
-$0.01	$0.00 $100.00	$100.01 $999.99	$1000.00

Boundary Value Analysis

- A **greater number of errors occurs at the boundaries** of the input domain rather **than in the "center".**

- Boundary value analysis leads to a **selection of test cases that exercise bounding values.**

- BVA leads to the **selection of test cases at the "edges" of the class.**

Boundary Value Analysis

- Rather than **focusing solely on input conditions**, **BVA derives test cases from the output domain as well.**

- **Examples:** temperature versus pressure table
- internal program data structures with prescribed boundaries

Boundary Value Analysis

- **Guidelines for BVA**
- If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.

- If an input condition specifies a number of values then test case designed for maximum and minimum values.

Distinguish White & Black Box Testng

- [..\..\Desktop\The Differences Between Black Box and White Box Testing.docx]

Review of OOA and OOD models

- The **construction of object-oriented** software begins with the **creation of requirements (analysis) and design models.**

- **review of OO analysis and design** models is especially useful because the same semantic **constructs various levels of software product.**

Review of OOA and OOD models

- **Earlier review may help in avoiding following problems in analysis:**

1. Unnecessary creation of special subclasses to **accommodate invalid attributes** is avoided.

2. A misinterpretation of the class definition may lead to **incorrect or extraneous class relationships.**

3. The **behavior of the system** or its classes may be **improperly characterized** to accommodate the extraneous attribute.

Review of OOA and OOD models

- **Earlier review may help in avoiding following problems in design:**

1. **Improper allocation of the class to subsystem** and/or tasks may occur during system design.

2. **Unnecessary design work** to create the procedural design for the operations that address the **extraneous attribute.**

3. The messaging model will be incorrect

Review of OOA and OOD models

- latter stages of their development, (OOA) and (OOD) **models provide substantial information about the structure and behavior of the system.**

- models should be **subjected to rigorous review** prior to the generation of code.

Review of OOA and OOD models

- **Correctness of OOA and OOD Models**
- syntactic correctness is judged on proper use of the symbology.
- each model is reviewed to ensure that proper modeling conventions have been maintained.
- If the model accurately reflects the real world , then it is semantically correct.

Review of OOA and OOD models

- **Consistency of Object-Oriented Models**
- The consistency judged by “considering the relationships among entities in the model.

- Each class and its connections to other classes – examine consistency

- class-responsibility-collaboration (CRC) model used to measure consistency.

Review of OOA and OOD models

- Steps to evaluate class model:

1. Revisit the CRC model and the object-relationship model – **requirements**

2. **Inspect** the description of each CRC index card to determine if a delegated responsibility is part of the collaborator’s definition.

3. **Invert** the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.

Review of OOA and OOD models

4. determine whether classes are valid or whether responsibilities are properly grouped among the classes.

5. Determine whether widely requested responsibilities might be combined into a single responsibility

Object Oriented Testing Strategies

- classical software testing strategy begins with “testing in the small” and works outward toward “testing in the large.”

1. Unit Testing in the OO Context

- Classes and objects
- Each class and object have attributes and operations
- Here smallest testable unit is class.

Object Oriented Testing Strategies

- Class (superclass) has operations defined , which are inherited by subclasses.

- Because **operation *X()*** is used varies in subtle (**indirect) ways**, it is necessary to test operation ***X()* in the context of each of the subclasses.**

- This means that testing operation *X()* in a vacuum (the traditional unit-testing approach) is ineffective in the object-oriented context.

Object Oriented Testing Strategies

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

2. Integration Testing in the OO Context

- There are two different strategies for integration testing of OO systems

1. Thread-based testing

- integrates the set of classes required to respond to one input or event for the system.

- Each thread is integrated and tested individually to ensure that no side effects occur.

Object Oriented Testing Strategies

2. Use-based testing

- begins the construction of the system by testing independent classes.

- Dependent classes, that use the independent classes are tested.

- This sequence of testing layers of dependent classes continues until the entire system is constructed.

- Use of drivers and stubs is to be avoided.

Object Oriented Testing Strategies

3. Validation Testing in an OO Context

- Like conventional validation, the validation of OO software **focuses on user-visible actions and user-recognizable outputs from the system.**
- Tester should draw upon **use cases based on requirement model.**
- The use case provides a scenario that has a **high likelihood of uncovered errors in user-interaction requirements.**

Software Rejuvenation

- **Re-documentation**
  - Creation or revision of alternative representations of software
    - at the same level of abstraction
  - Generates:
    - data interface tables, call graphs, component/variable cross references etc.
- **Restructuring**
  - transformation of the system’s code without changing its behavior

Software Rejuvenation

- **Reverse Engineering**
  - Analyzing a system to extract information about the behavior and/or structure
    - also Design Recovery - recreation of design abstractions from code, documentation, and domain knowledge
  - Generates:
    - structure charts, entity relationship diagrams, DFDs, requirements models
- **Re-engineering**
  - Examination and alteration of a system to reconstitute it in another form
  - Also known as renovation, reclamation

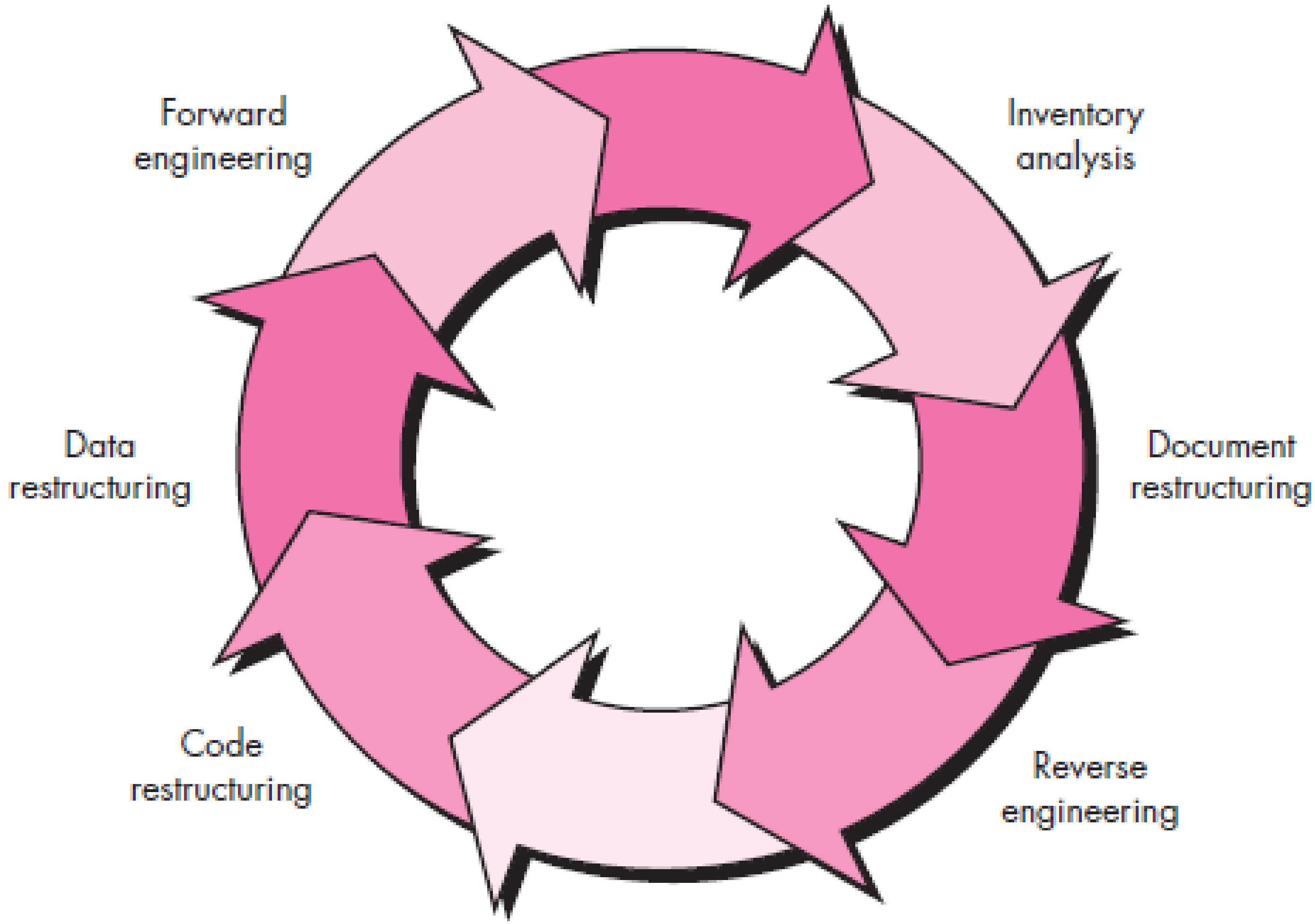
Reengineering

- Reengineering – cost , time , resources - reengineering is not accomplished in a few months or years.

- every organization needs a pragmatic strategy for software reengineering.

- Reengineering is a rebuilding activity.

Software Reengineering Process Model



Software Reengineering Process Model

1. Inventory analysis.

- The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application.

- As status of application can change any time , inventory should be revisited on regular basis.

Software Reengineering Process Model

2. Document restructuring.

- Weak documentation is the trademark of many legacy systems.

1. *Creating documentation is far too time consuming. –* static programs

2. *Documentation must be updated, but your organization has limited resources –* re-document only changed portion

Software Reengineering Process Model

3. The system is business critical and must be fully re-documented - Even in this case, an intelligent approach is to pare documentation to an essential minimum.

3. Reverse engineering

- A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets."

- Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Software Reengineering Process Model

4. Code restructuring.

5. Data restructuring

- A program with weak data architecture will be difficult to adapt and enhance.

- Current data architecture is dissected, and necessary data models are defined.

- Data objects and attributes are identified, and existing data structures are reviewed for quality.

Software Reengineering Process Model

6. Forward engineering

- Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to **improve its overall quality.**

Reverse engineering

- The process of recreating a design by analyzing a final product.

- The **abstraction** level of a reverse engineering **refers to the sophistication of the design information** that can be extracted from source code.

- The **completeness** of a reverse engineering process **refers to the level of detail that is provided at an abstraction level.**

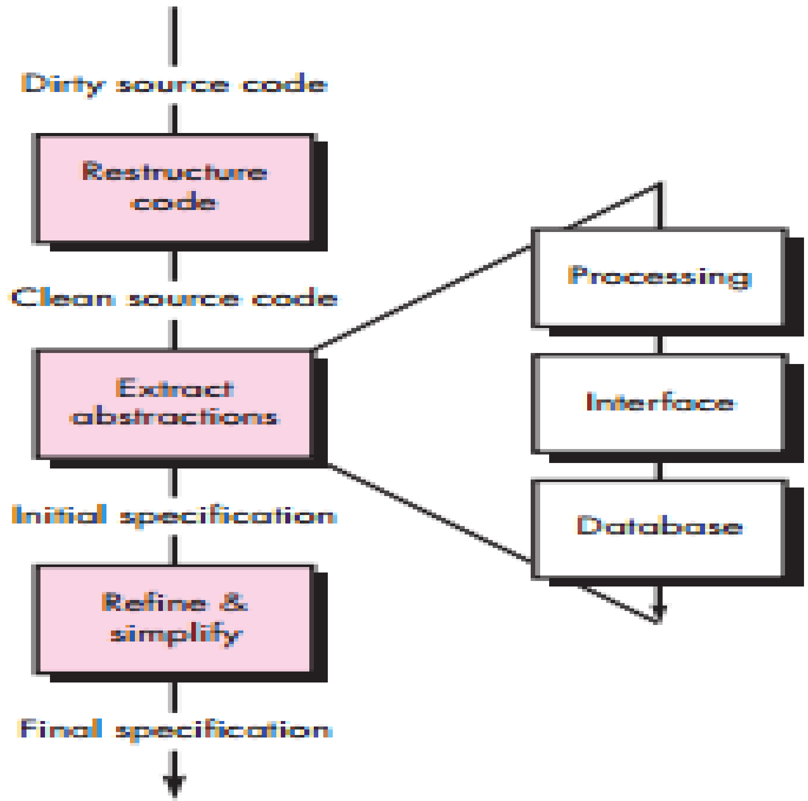
Reverse engineering

- **Interactivity** refers to the **degree to which the human is "integrated" with automated tools** to create an effective reverse engineering process.

- Directionality – one way (maintenance activity) or two way (restructure)

Reverse engineering

- Reverse Engineering Process



Software Maintenance

- Software maintenance is the general process of changing a system after it has been delivered.

- The change may be simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancement to correct specification error or accommodate new requirements.

Software Maintenance

- **There are three different types of software maintenance:**

1. Fault repairs

- Coding errors are usually relatively cheap to correct.

- Design errors are more expensive as they may involve rewriting several program components.

- Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.

Software Maintenance

2. Environmental adaptation

- This type of maintenance is required when some aspect of the system's environment changes.
- Example: hardware, the platform operating system, or other support software changes.
- The application system must be modified to adapt it to cope with these environmental changes.

Software Maintenance

3. Functionality addition

- *This type of maintenance is necessary when the system* requirements change.

- The scale of the changes required to the software is often much greater than for the other types of maintenance.

Software Maintenance

- Other types of software maintenance with different names:

- **Corrective maintenance** is universally used to refer to maintenance for fault repair.

- **Adaptive maintenance** sometimes means adapting to a new environment

- **Perfective maintenance** sometimes means perfecting the software by implementing new requirements