Testing

Testing

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

Testing

- Unfortunately, it is impossible to completely test a non-trivial system.
- Testing needs to be performed under time and budget constraints.
- As a result, systems are deployed without being completely tested, leading to faults discovered by end users.

Launch of the Space Shuttle Columbia

- The first launch of the Space Shuttle Columbia in 1981,was cancelled because of a problem that was not detected during development.
- The problem was traced to a change made by a programmer 2 years earlier, who erroneously reset a delay factor from 50 to 80 milliseconds.
- This added a probability of 1/67 that any space shuttle launch would fail.
- Unfortunately, despite of thousands of hours of testing after the change was made, the fault was not discovered during the testing phase.
- During the actual launch the fault caused a synchronization problem with the shuttle's five on-board computers that then led to the decision to abort the launch.

Purpose of Testing

- The aim of program testing is to help **identify all defects in a program.**
- However, in practice, even after satisfactory completion of the testing phase, **it is not possible to guarantee that a program is error free.**
- This is because the **input data domain of most programs is very large,** and it is **not practical to test the program exhaustively with respect to each value** that the input can assume.

Consider a function taking a floating point number as argument.
If a tester takes 1sec to type in a value, then **even a million testers would not be able to exhaustively test** it after trying for a million number of years.

Even with this limitation, we should **not underestimate the importance of testing.**

Careful testing can **expose a large percentage of the defects** existing in a program, and provides a practical way of reducing defects in a system.

Testing Principles

Glen Myers states a number of rules that can serve well as testing objectives:

- Testing is a process of executing a program with the **intent of finding an error.**
- A good test case ?

Testing Principles

Glen Myers states a number of rules that can serve well as testing objectives:

- Testing is a process of executing a program with the **intent of finding an error.**
- A good test case is one that has a **high probability of finding an as-yet undiscovered error.**
- **A successful test is one that uncovers** an as-yet-undiscovered error.

Testing Principles

Principle 1. ***All tests should be traceable to customer requirements.***
The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
Principle 2. ***Tests should be planned long before testing begins***

When????????

Testing Principles

Principle 1. ***All tests should be traceable to customer requirements.***

The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2. ***Tests should be planned long before testing begins.***

- **Test planning** can begin as soon as the requirements model is complete.
- **Detailed definition of test cases** can begin as soon as the design model has been solidified.

Therefore, all tests can be planned and designed before any code has been generated.

Test Plan

- Before testing activities start, a test plan is developed.
- The test plan documents the following:
  - Features to be tested
  - Features not to be tested
  - Test strategy
  - Test suspension criteria
  - stopping criteria
  - Test effort
  - Test schedule

Testing Principles

Principle 3. ***The Pareto principle applies to software testing*****.**

- Pareto principle implies that **80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components.**
- The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. ***Testing should begin "in the small" and progress toward testing "in the large."***

HOW???????

Testing Principles

Principle 3. ***The Pareto principle applies to software testing*****.**

- Pareto principle implies that **80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components.**
- The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. ***Testing should begin "in the small" and progress toward testing "in the large."***

- The first tests planned and executed generally **focus on individual components.**
- As testing progresses, focus shifts in an attempt to find errors in **integrated clusters of components and ultimately in the entire system.**

Testing Principles

Principle 5. *Exhaustive testing is not possible.*

- The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.

Terminologies

Standardised by the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

- A **mistake** is essentially **any programmer action that later shows up as an incorrect result during program execution**. A programmer may commit a mistake in almost any development activity. For example,
  - during coding a programmer might commit the mistake of **not initializing a certain variable**, or
  - might overlook the errors that might arise in some exceptional situations such as **division by zero** in an arithmetic operation.
  - Both these **mistakes can lead to an incorrect result**.

- An **error** is **the result of a mistake** committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. Example of an error is:
  - a call made to a wrong function.
- The terms error, fault, bug, and defect are considered to be synonyms in the area of program testing.

Terminologies

A **failure** of a program essentially denotes **an incorrect behaviour exhibited by the program during its execution.**
An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program.

Every failure is caused by some bugs present in the program.
In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large.
Examples:
- The result computed by a program is 0, when the correct result is 10.
- A program crashes on an input.
- A robot fails to avoid an obstacle and collides with it.

It may be noted that **mere presence of an error in a program code may not necessarily lead to a failure during its execution.**

Test Case

- A **test case** is a triplet [I , S, R], where
  - I is the data input to the program under test,
  - S is the state of the program at which the data is to be input. The state of a program is also called its execution mode.
  - R is the result expected to be produced by the program.
- Example- Consider the different execution modes of a certain text editor software.
  - edit
  - View
  - create
  - display.

Test Case

A test case is a set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.

Eg—
[input: “abc”, state: edit, result: abc is displayed], which essentially means that the input abc needs to be applied in the edit mode, and the expected result is that the string abc would be displayed.

Test Case

A test case is said to be a **positive test case** if it is designed to test

- whether the software correctly performs a required functionality.

A test case is said to be **negative test case**, if it is designed to test

- whether the software carries out something, that is not required of the system.

Example ??

Test Case

Example –

Consider a program to manage user login.

- A positive test case can be designed to check if a
  - login system validates a user with the **correct user name and password.**
- A negative test case in this case can be a test case that checks whether the
  - the login functionality validates and admits a **user with wrong or bogus login user name or password.**

Test Scenario

A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output.

A test case can be said to be an implementation of a test scenario.

Test Scenario

- An important automatic test case design strategy is to:
  - first design test scenarios through an analysis of some program abstraction (model) and
  - then implement the test scenarios as test cases.
- In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed.

Test Script and Test Suite

- A **test script** is an **encoding of a test case as a short program.**
  - Test scripts are developed for automated execution of the test cases.

A **test suite** is the set of all test that have been designed by a tester to test a given program.

Test Script and Test Suite

- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance.
- In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

Strategies For Software Testing

- Strategy for software testing provides a road map that describes the steps to be conducted as part of testing,
  - when these steps are planned and then undertaken,
  - how much effort, time, and resources will be required.
- Any testing strategy must incorporate:
  - **TEST PLANNING**
  - **TEST CASE DESIGN**
  - **TEST EXECUTION, AND**
  - **RESULTANT DATA COLLECTION AND EVALUATION.**

Strategies For Software Testing

- A software testing strategy should be:
  - flexible enough to promote a customized testing approach. At the same time,
  - it must be rigid enough to encourage reasonable planning and management tracking as the project progresses.

Software Inspections

- “There is no urge so great as for one man to edit another man’s work.”**-Mark Twain**

A STRATEGIC APPROACH TO SOFTWARE TESTING

A number of software testing strategies have been proposed in the literature. All have the following generic characteristics:

• To perform effective testing, **you should conduct effective technical reviews .** By doing this, many errors will be eliminated before testing commences.

• **Testing begins at the component level and works “outward”** toward the integration of the entire computer-based system.

A STRATEGIC APPROACH TO SOFTWARE TESTING

• **Different testing techniques are appropriate** for different software engineering approaches and **at different points in time.**

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

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

FORMAL TECHNICAL REVIEWS

A *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; and
(5) To make projects more manageable.

FORMAL TECHNICAL REVIEWS

In addition, the FTR serves as a training ground,

- enabling junior engineers to observe different approaches to software analysis, design, and implementation.

The FTR also serves to promote backup and continuity because

- a number of people become familiar with parts of the software that they may not have otherwise seen.

FTR

- Given these constraints, it should be obvious that an FTR **focuses on a specific (and small) part of the overall software.**
- For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components.
- **By narrowing the focus, the FTR has a higher likelihood of uncovering errors.**

The Review Meeting

The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component).

1) The individual who has developed the work product—the *producer*
   —informs the project leader that the work product is complete and that a review is required.
2) The project leader contacts a *review leader*,
   - who evaluates the product for readiness,
   - generates copies of product materials, and
   - distributes them to two or three *reviewers* for advance preparation.

The Review Meeting

3) Each reviewer is expected to
   - spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.

Concurrently, the review leader also :
   - establishes an agenda for the review meeting, which is typically scheduled for the next day.

The Review Meeting

4) The review meeting is attended by the review leader, all reviewers, and the producer.
5) One of the reviewers takes on the role of a *recorder*, that is, the individual who records (in writing) all important issues raised during the review.
6) The FTR begins with an introduction of the agenda and a brief introduction by the producer.
7) The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation.
8) When valid problems or errors are discovered, the recorder notes each.

The Review Meeting

9) At the end of the review, all attendees of the FTR must decide whether to:
   a) Accept the product without further modification,
   b) Reject the product due to severe errors (once corrected, another review must be performed), or
   c) Accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required).
10) After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised.
These are summarized at the end of the review meeting, and a review issues list is produced.

In addition, a formal technical review summary report is completed.

A review summary report answers three questions:
1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

Review Guidelines

1) *Review the product, not the producer*
2) *Set an agenda and maintain it.*
3) *Limit debate and rebuttal*
4) *Enunciate problem areas, but don't attempt to solve every problem noted.*
5) *Take written notes*
6) *Limit the number of participants and insist upon advance preparation*
7) *Develop a checklist for each product that is likely to be reviewed.*
8) *Allocate resources and schedule time for FTRs*
9) *Conduct meaningful training for all reviewers*
10) *Review your early reviews*

Verification and Validation

- Software testing is one element of a broader topic that is often referred to as verification and validation (V&V).
- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Boehm [Boe81] states this another way:
  - Verification: "Are we building the product right?"
  - Validation: "Are we building the right product?"

Verification and Validation

Verification and validation includes a wide array of SQA activities:

- TECHNICAL REVIEWS,
- QUALITY AND CONFIGURATION AUDITS,
- PERFORMANCE MONITORING,
- SIMULATION,
- FEASIBILITY STUDY,
- DOCUMENTATION REVIEW,
- DATABASE REVIEW,
- ALGORITHM ANALYSIS,
- **DEVELOPMENT TESTING,**
- **USABILITY TESTING,**
- **QUALIFICATION TESTING,**
- **ACCEPTANCE TESTING, AND**
- **INSTALLATION TESTING.**

Although testing plays an extremely important role in V&V, **many other activities are also necessary.**

Verification and Validation

- But testing should not be viewed as a safety net.
- As they say, “You can’t test in quality. If it’s not there before you begin testing, it won’t be there when you’re finished testing.”
- Quality is incorporated into software throughout the process of software engineering.
- Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

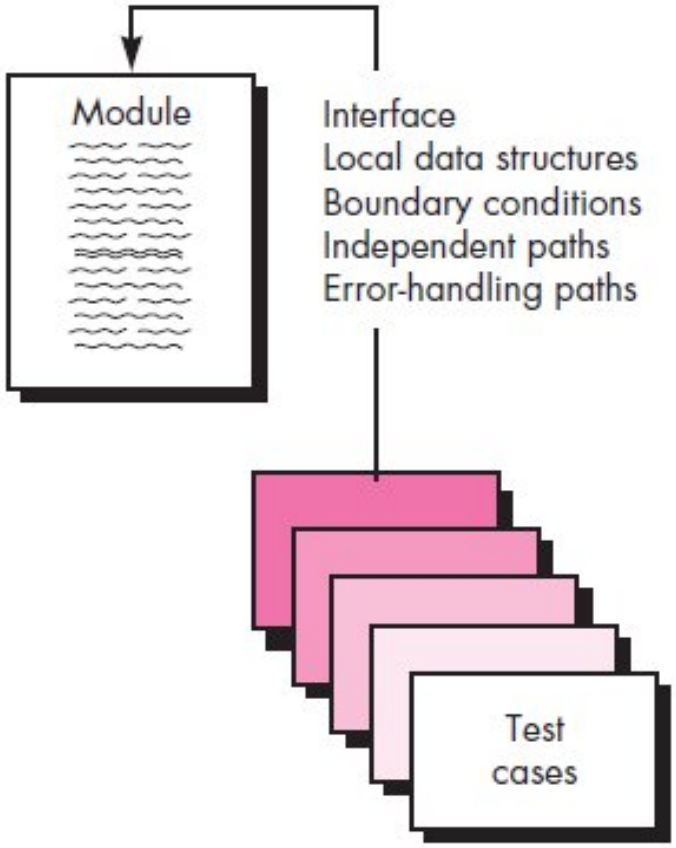
Unit Testing

- *Unit testing* focuses verification effort on the smallest unit of software design—the software component or module.
- Uncover errors within the boundary of the module.
- The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.
- The unit test focuses on the internal processing logic and data structures within the boundaries of a component.
- This type of testing can be conducted in parallel for multiple components.

Unit-test considerations

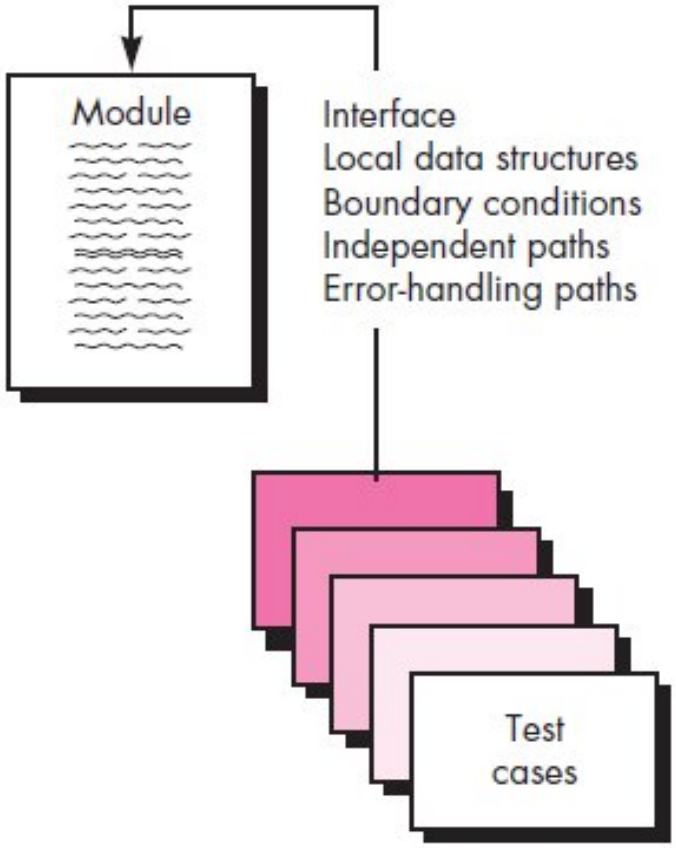
Unit tests are illustrated schematically in Figure.

1) The **module interface is tested** to:
   - ensure that information **properly flows into and out of the program unit** under test.
2) **Local data structures are examined**:
   - to ensure that **data stored temporarily maintains its integrity during all steps in an algorithm's execution.**



Unit-test considerations

3) **All independent paths** through the control structure are **exercised**:
   - to ensure that **all statements in a module have been executed at least once.**
4) **Boundary conditions are tested** to
   - ensure that the module operates properly at boundaries established to limit or restrict processing.
5) And finally, all **error-handling paths are tested.**



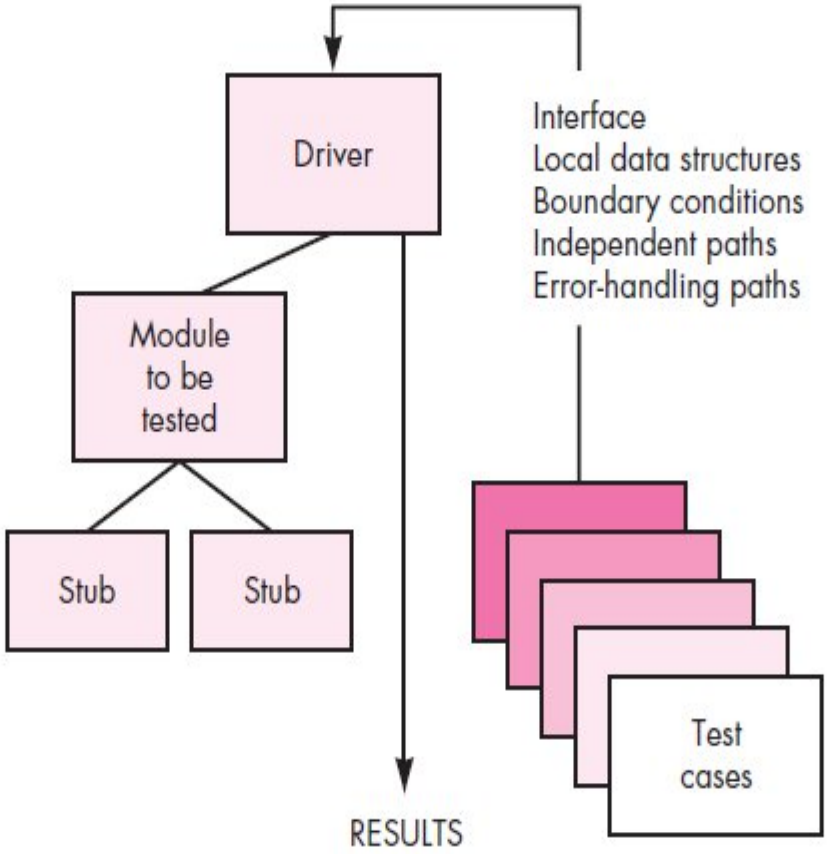
Unit-test procedures

- Unit testing is normally considered as an adjunct to the coding step.
  - The design of unit tests can occur before coding begins or after source code has been generated.
- A review of design information provides guidance for establishing test cases.
  - Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.

Unit-test procedures

- The unit test environment is illustrated in Figure .

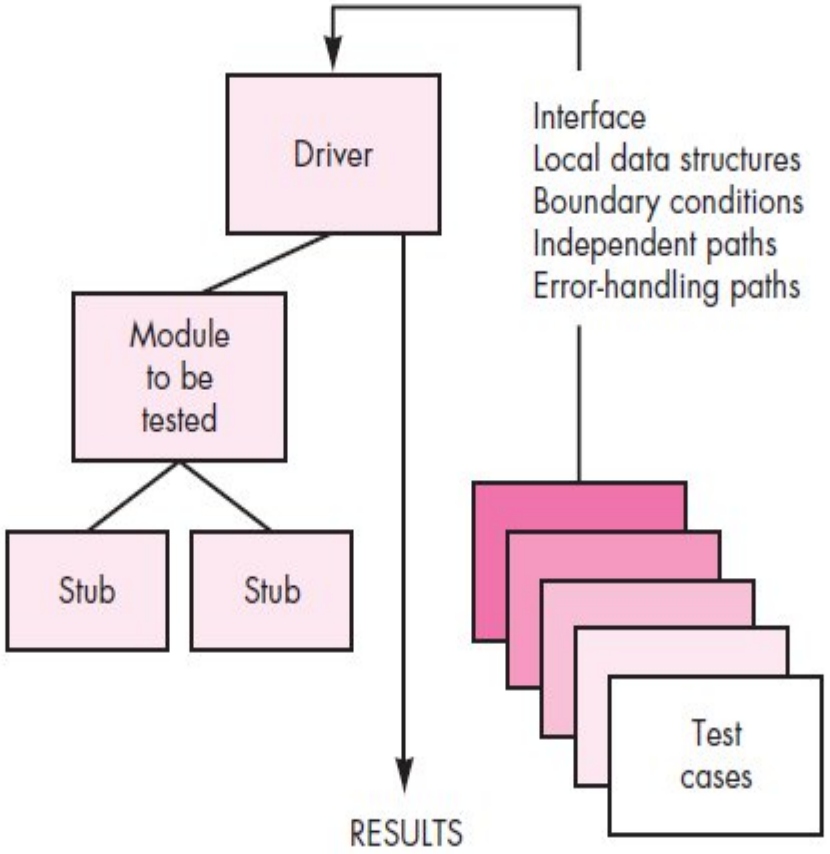
- In most applications a *driver* is nothing more than a "main program" that:
  - **accepts test case data,**
  - **passes such data to the component (to be tested), and**
  - **prints relevant results.**



Unit-test procedures

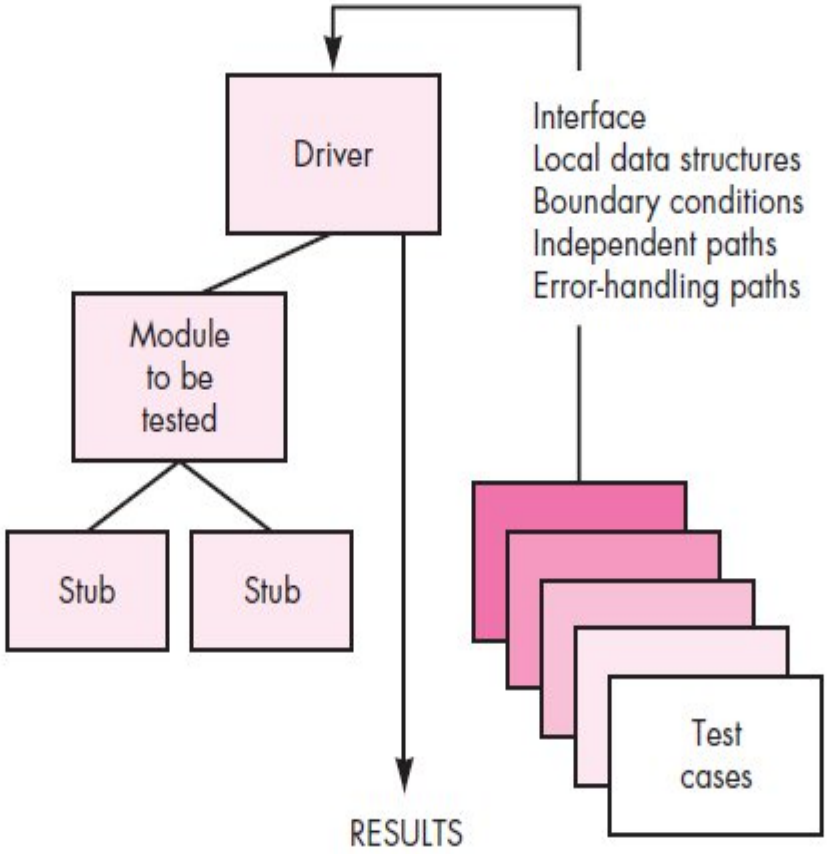
- ***Stubs*** **serve to replace modules that are subordinate (invoked by) the component to be tested.**

- A stub or "dummy subprogram":
  - **uses the subordinate module's interface,**
  - **may do minimal data manipulation,**
  - prints verification of entry, and
  - **returns control to the module undergoing testing.**



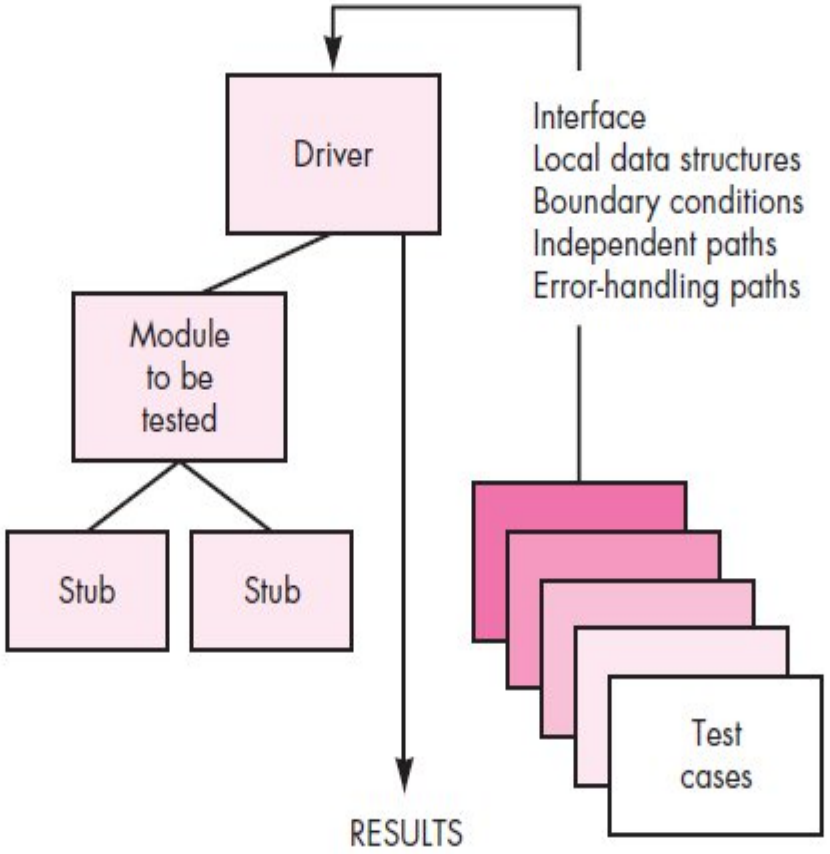
Unit-test procedures

- Drivers and stubs **represent testing "overhead." :**
  - That is, **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-test procedures

- Unfortunately, many components **cannot be adequately unit tested with "simple" overhead** software.
- In such cases, **complete testing can be postponed :**
  - **until the integration test step**



Integration Testing

- Once all modules have been unit tested, The Question arises:

- “If they all work individually, why do you doubt that they’ll work when we put them together?”
- Example????

Example??

- STDNT (INDIVIDUAL EFFORTS)
- Team Efforts

Symphony

- Solo Performances
- Team Performances

Integration Testing

The problem, of course, is “putting them together”—

- Interfacing.
- Data can be lost across an interface
- One component can have an inadvertent, adverse effect on another
- Sub-functions, when combined, may not produce the desired major function
- Individually acceptable imprecision may be magnified to unacceptable levels
- Global data structures can present problems.

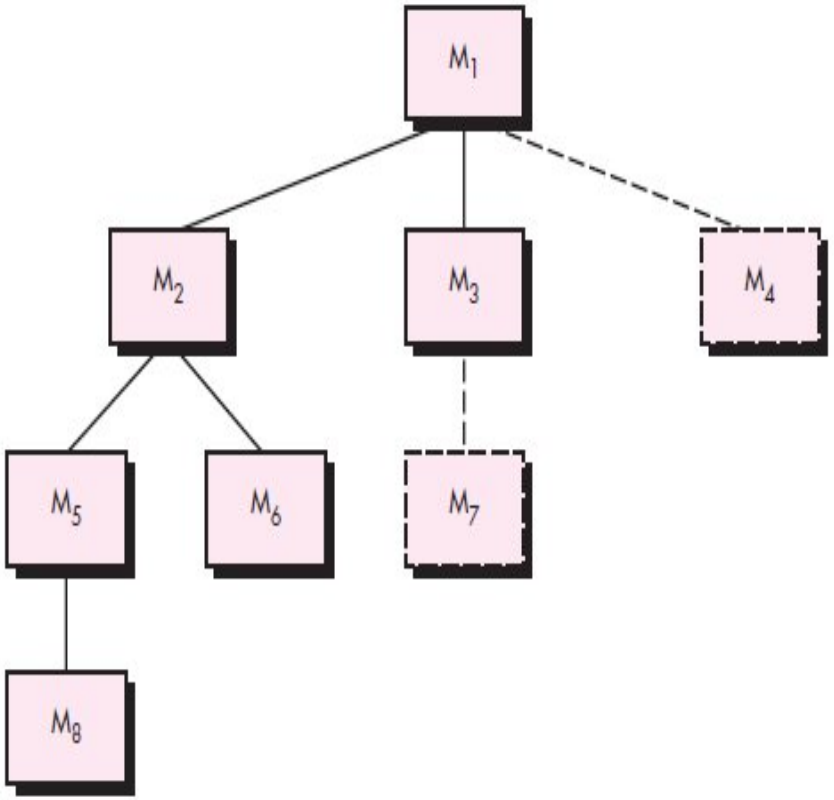
Integration Testing

- Integration testing is a systematic technique for :
  - constructing the software architecture
  - while at the same time conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that has been dictated by design.

Top-down integration

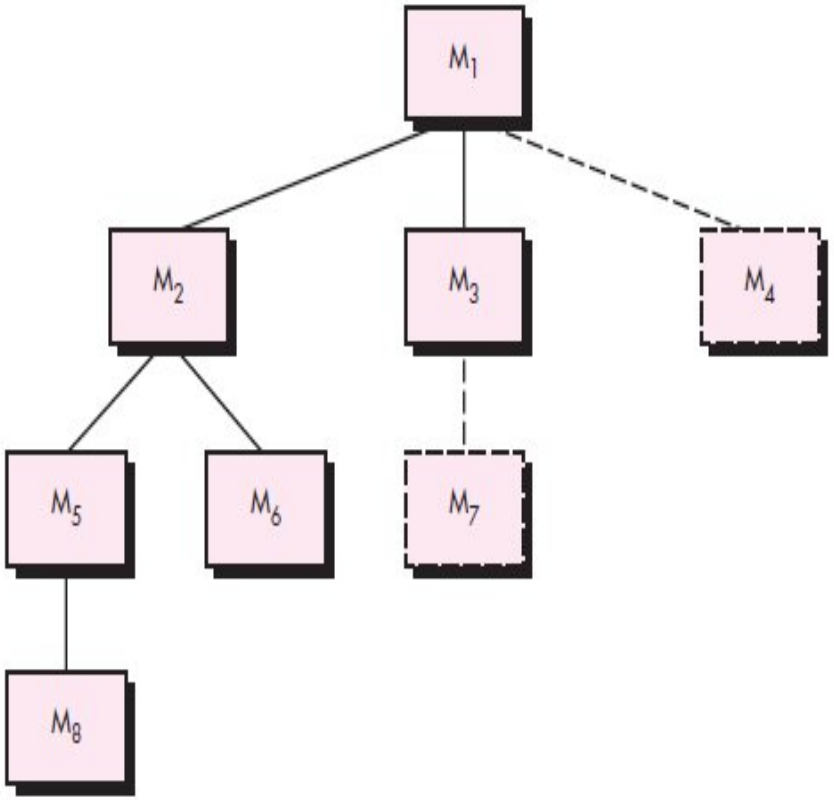
- *Top-down integration testing* is **an incremental approach** to construction of the software architecture.

- **Modules are integrated by moving downward** through the control hierarchy,
  - beginning with the main control module (main program).
  - Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure **in either a depth-first or breadth-first manner.**



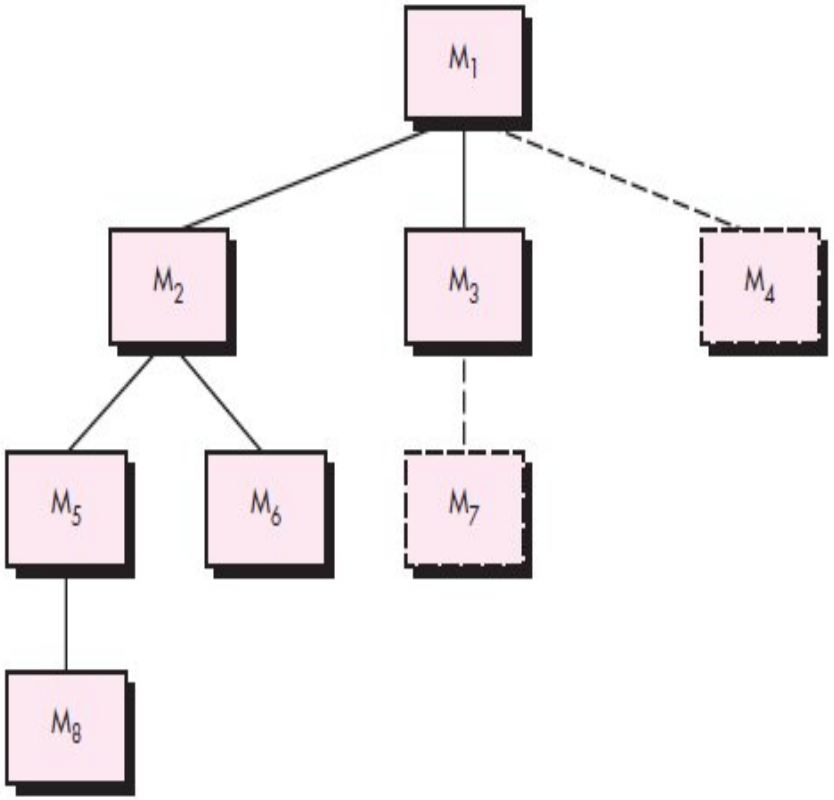
Top-down integration

- ***Depth-first integration*** integrates all components on **a major control path** of the program structure.
- Selection of a major path is **somewhat arbitrary and depends on application-specific** characteristics.
- For example, :
  - **Selecting the left-hand path, components M1, M2 , M5 would be integrated first.**
  - Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated.
  - Then, the central and right-hand control paths are built.



Top-down integration

- ***Breadth-first integration*** **incorporates all components directly subordinate at each level,** moving across the structure horizontally.
  - **M2, M3, and M4 would be integrated first.**
  - The next control level, **M5, M6, and so on**, follows.



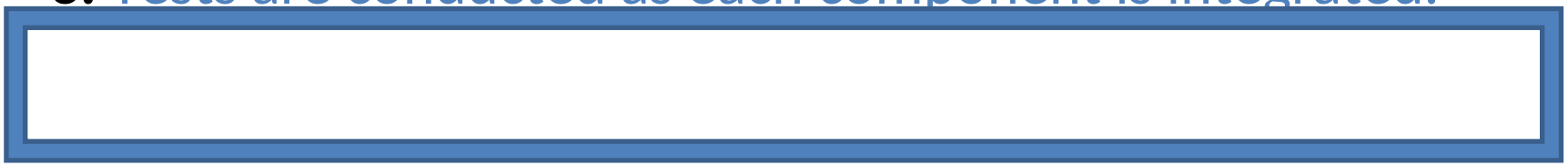
Top-down integration

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.



Top-down integration

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

Bottom-up integration

- *Bottom-up integration testing*, as its name implies
  - begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure).

- Because components are integrated from the bottom up,
  - the functionality provided by components subordinate to a given level is always available and

  - the need for stubs is eliminated.

Bottom-up integration

A bottom-up integration strategy may be implemented with the following steps:
1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.

2. A *driver* (a control program for testing) is written to coordinate test case input and output.

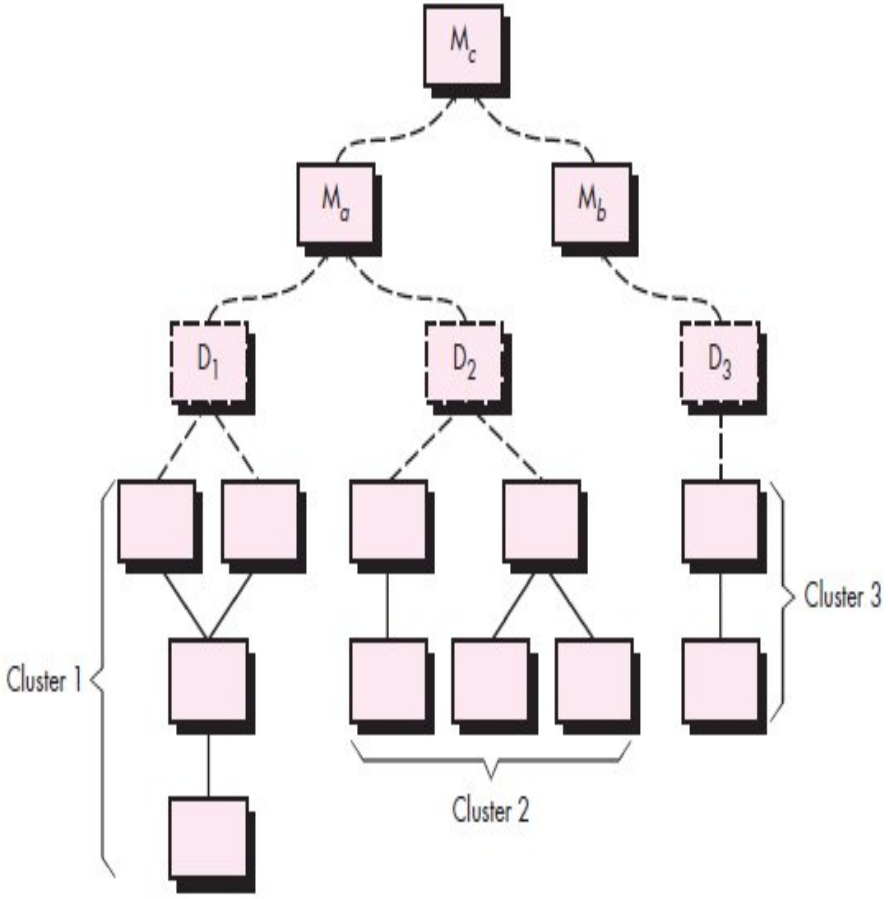
3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Bottom-up integration

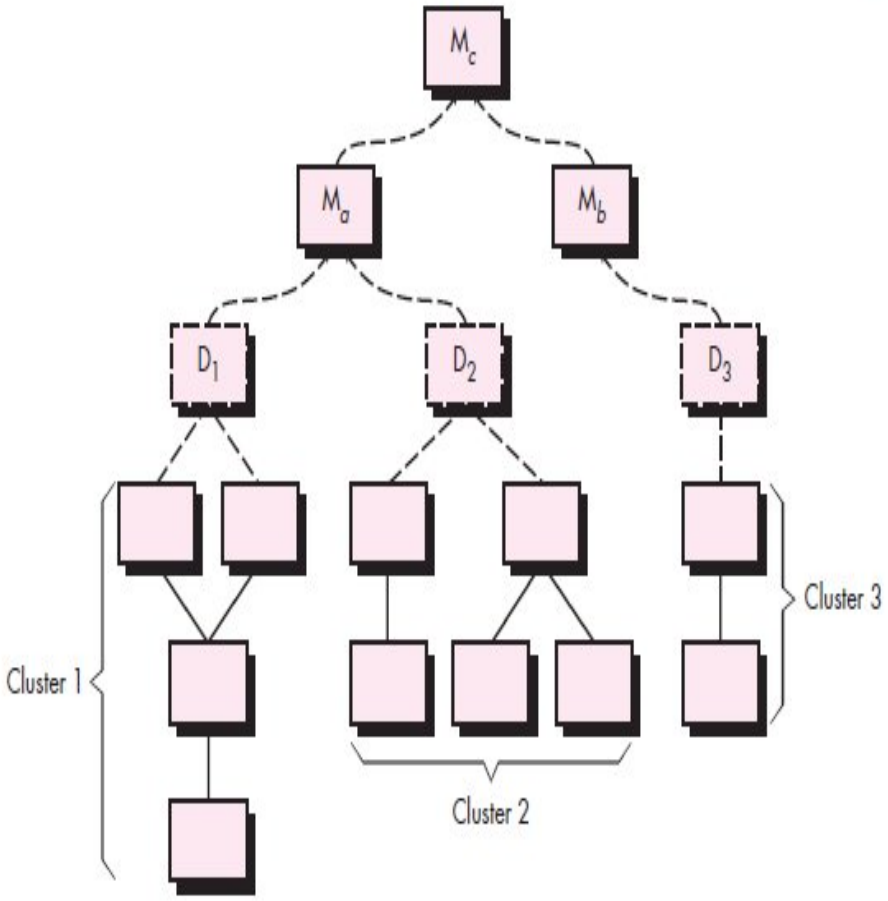
Referring to the Figure.

- Components are combined to form clusters 1, 2, and 3.
- **Each of the clusters is tested using a driver** (shown as a dashed block).
- Components in clusters 1 and 2 are subordinate to M*a*.
- **Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*.**
- **Similarly, driver D3 for cluster 3 is removed** prior to integration with module M*b*.



Bottom-up integration

- Both M*a* and M*b* will ultimately be integrated with component M*c*, and so forth.
- **As integration moves upward, the need for separate test drivers lessens.**
- In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.



Regression testing

- *Regression testing is an important strategy for* ***reducing "side effects."***
- *Run regression tests* ***every time a major change is made to the software*** *(including the integration of new components).*

Regression testing

- Each time a new module is added as part of integration testing, the software changes:
  - **New data flow paths are established,**
  - **New I/O may occur, and**
  - **New control logic is invoked.**
  - These changes may cause problems with functions that previously worked flawlessly.
- In the context of an integration test strategy, *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.**

Regression testing

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

- *Capture/playback tools* enable the software engineer to capture test cases and results for subsequent playback and comparison.

The *regression test suite* (the subset of tests to be executed) contains three different classes of test cases:

- **A representative sample of tests that will exercise all software functions.**
- **Additional tests that focus on software functions that are likely to be affected by the change.**
- **Tests that focus on the software components that have been changed.**

Regression testing

- As integration testing proceeds,
  - the number of regression tests can grow quite large.
- Therefore, the regression test suite should be designed to include
  - only those tests that address one or more classes of errors in each of the major program functions.
- It is **impractical and inefficient to re-execute every test for every program function once a change has occurred.**

System testing

Unit and integration testing focus on finding faults in individual components and the interfaces between the components.

Once components have been integrated, system testing:

- ensures that the **complete system complies with the**
  - **functional and**
  - **Non-functional requirements of the system.**

System testing

There are several system testing activities that are performed:

- **Functional testing**. **Test of functional requirements**
- **Performance testing**. **Test of non-functional requirements**
- **Pilot testing**. Tests of common **functionality among a selected group of end users in the target environment** .

It consists of :

  - **Alpha Testing**
  - **Beta Testing**

- **Acceptance testing.**

Functional testing

- Functional testing, also called requirements testing,:
  - **finds differences between the functional requirements and the system.**

- System testing is a black box technique.

- Test cases are derived from the use case model.

- In systems with complex functional requirements, it is usually not possible to test all use cases for all valid and invalid inputs.

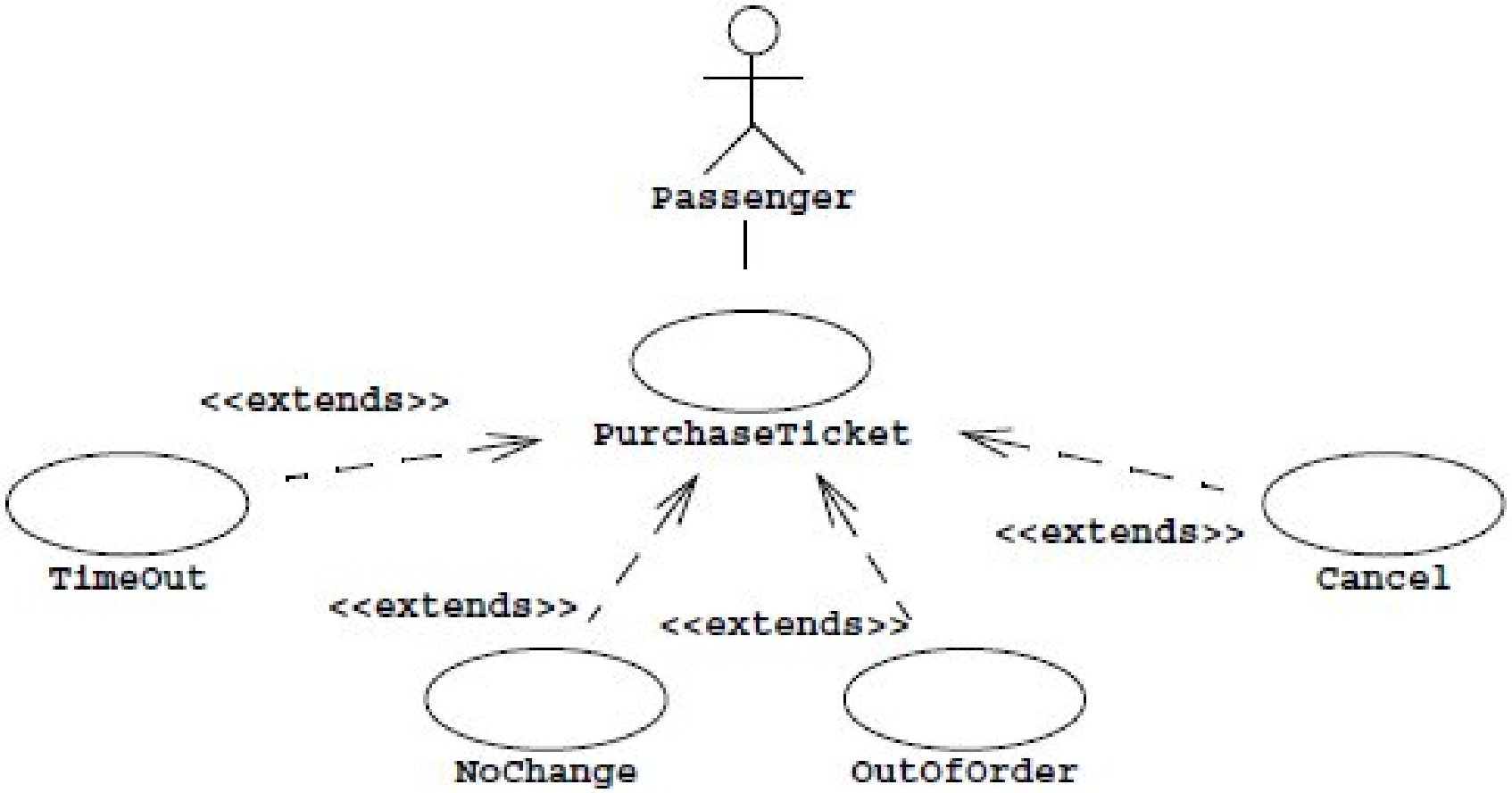
Functional testing

- The goal of the tester is to select those tests that are relevant to the user and have a high probability of uncovering a failure.

- To identify functional tests, we inspect the use case model and identify use case instances that are likely to cause failures.

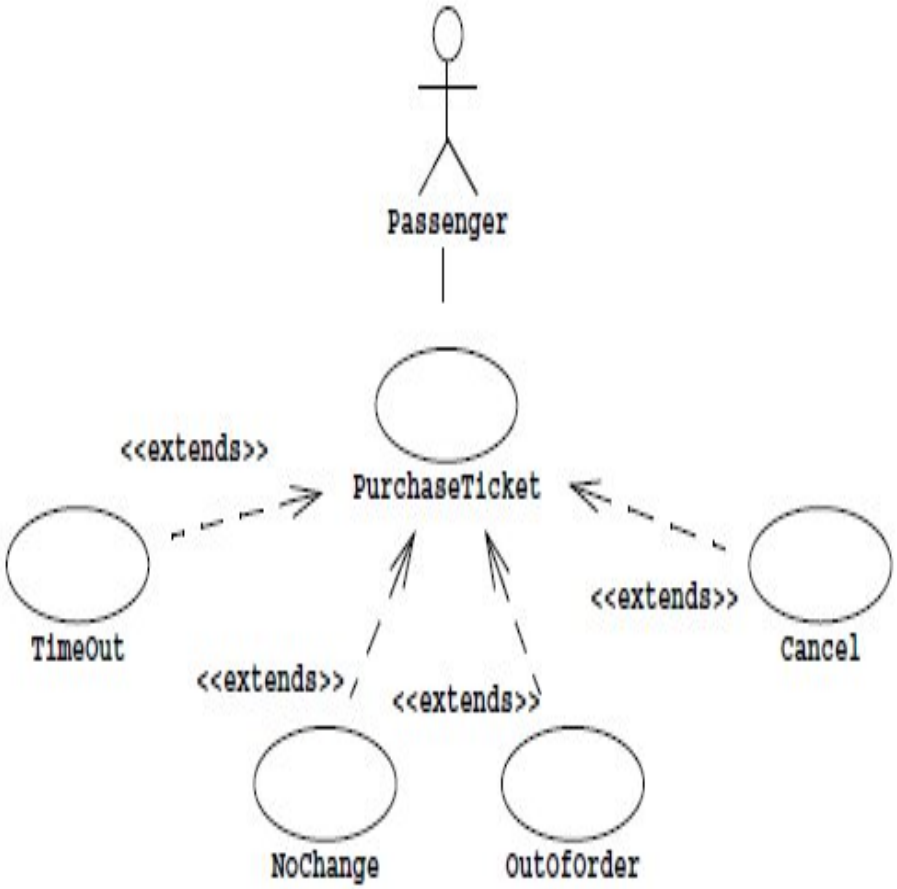
- **Test cases should exercise both:**
  - **common and**
  - **exceptional use cases.**

Functional testing



Functional testing

- Use case model for a subway ticket distributor
- The common case functionality are:
  - PurchaseTicket use case, describing the steps necessary for a Passenger to successfully purchase a ticket.
- The exceptional conditions use cases are:
  - The TimeOut,
  - Cancel,
  - OutOfOrder, and
  - NoChange



Test Cases

Three features of the Distributor are likely to fail and should be tested:

1. The Passenger may press multiple zone buttons before inserting money, in which case the Distributor should display the amount of the last zone

2. The Passenger may select another zone button after beginning to insert money, in which case the Distributor should return all money inserted by the Passenger

3. The Passenger may insert more money than needed, in which case the Distributor should return the correct change.

Performance testing

- Performance testing is carried out to check whether the system meets the non-functional requirements identified in the SRS document.

- For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document.

- All performance tests can be considered as black-box tests.

Performance testing

The following tests are performed during performance testing:

- **Stress Testing**
- **Deployment Testing**
- ***Security testing***
- ***Recovery testing***
- **Volume testing**
- **Documentation testing**
- **Usability testing**

- Stress Testing????

Stress Testing

Stress tests are designed to confront programs with abnormal situations.
In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

Stress testing executes a system in a manner that demands resources in abnormal:

- quantity,
- frequency, or
- volume.

Essentially, the tester attempts to break the program.

Stress Testing

For example,
(1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
(2) input data rates may be increased by an order of magnitude to determine how input functions will respond,
(3) test cases that require maximum memory or other resources are executed,
(4) test cases that may cause thrashing in a virtual operating system are designed,
(5) test cases that may cause excessive hunting for disk-resident data are created.

Sensitivity Testing

- A variation of stress testing is a technique called *sensitivity testing*.
- In some situations (the most common occur in mathematical algorithms),
  - a very small range of data contained within the bounds of valid data for a program
  - may cause extreme and even erroneous processing or profound performance degradation.
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

Deployment Testing

- In many cases, software must execute on a variety of platforms and under more than one operating system environment.
- *Deployment testing*, sometimes called *configuration testing:*
  - exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines :
  1) all **installation procedures** and
  2) **specialized installation software (e.g., "installers")** that will be used by customers, and
  3) all **documentation** that will be used to introduce the software to end users.

Security testing

- *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration

- During security testing, the **tester** plays the role(s) of:
  - **the individual who desires to penetrate the system.**

Security testing

- The tester may attempt to:
  - **acquire passwords** through external clerical means;
  - **may attack the system** with custom software designed to break down any defenses that have been constructed;
  - **may overwhelm the system**, thereby denying service to others; **(DOS)**
  - **may purposely cause system errors**, hoping to **penetrate during recovery;**
  - **may browse through insecure data**, hoping to find the key to system entry.

Security testing

Given enough time and resources, **good security testing will ultimately penetrate a system. !!!!!!**

Then WHAT????????????

Security testing

- Given enough time and resources, **good security testing will ultimately penetrate a system.**

The role of the system designer is to **make penetration cost more than the value of the information that will be obtained.**

Recovery testing

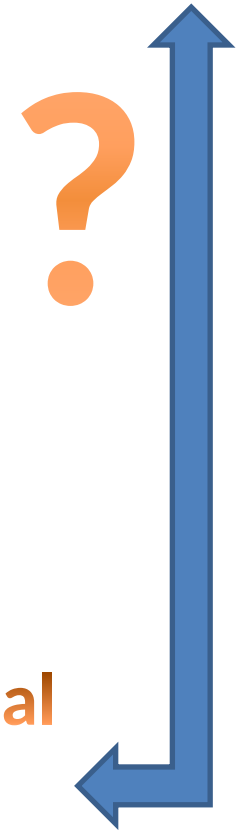
- *Recovery testing* is a system test that:
  - **forces the software to fail in a variety of ways** and
  - **verifies that recovery is properly performed.**
- If recovery is automatic (performed by the system itself):
  - **reinitialization,**
  - **checkpointing mechanisms,**
  - **data recovery, and**
  - **restart**

  are evaluated for correctness.
- If recovery requires human intervention:
  - the **mean-time-to-repair (MTTR) is evaluated** to determine whether it is within acceptable limits.

Volume testing

- **Volume testing** attempts to find **faults associated with large amounts of data,** such as
  - **static limits imposed by the data structure, or**
  - **high-complexity algorithms, or**
  - **high disk fragmentation.**
- Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.
- For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.

Documentation testing



- It is checked whether the required :
  - **user manual,**
  - **maintenance manuals,**
  - **technical manuals exist and are consistent.**
- If the requirements specify the:
  - **types of audience for which a specific manual** should be designed,
  - then the **manual is checked for compliance** of this requirement.

Usability testing

- Usability testing concerns **checking the user interface** to see:
  - if it meets all user requirements concerning the user interface.
- During usability testing,:
  - **display screens,**
  - **messages,**
  - **report formats**, and
  - other aspects relating to the user interface requirements are tested.

Pilot testing

- During the **pilot test**, also called the **field test**, the system is installed and used by a selected set of users.
- Users exercise the system as if it had been permanently installed. No explicit guidelines or test scenarios are given to the users.

Alpha Test

- An **alpha test** is a pilot test with users exercising the system in the **development environment**.
- The *alpha test* is conducted **at the developer's site** by a representative group of end users.
- The software is used in a natural setting with the **developer "looking over the shoulder"** of the users and **recording errors and usage problems**.
- Alpha tests are conducted in **a controlled environment.**

Beta Test

- In a **beta test**, the acceptance test is performed by a limited number of end users **in the target environment**.
- The *beta test* is conducted at **one or more end-user sites**.
- Unlike alpha testing, the **developer generally is not present**.
- Therefore, the beta test is a **"live" application of the software in an environment that cannot be controlled by the developer.**

Beta Test

- The customer **records all problems** (real or imagined) that are encountered during beta testing and **reports these to the developer at regular intervals**.
- As a result of problems reported during beta tests, you make **modifications and then prepare for release** of the software product to the entire customer base.

Acceptance testing

- Performed **when custom software is delivered** to a customer under contract.
- The customer performs **a series of specific tests in an attempt to uncover errors before accepting the software** from the developer.
- In some cases (e.g., **a major corporate or governmental system)** acceptance testing can be:
  - very formal and
  - encompass **many days or even weeks of testing**.

Acceptance testing

There are three ways the client evaluates a system during acceptance testing.

- **Benchmark test**, the client prepares **a set of test cases that represent typical conditions under which the system should operate.**
- Benchmark tests can be performed with:
  - **actual users or**
  - **by a special test team** exercising the system functions,

  but it is important that the testers be familiar with the functional and non functional requirements so they can evaluate the system.

Acceptance testing

- Another kind of system acceptance testing is used in **reengineering projects**, when the new system replaces an existing system.
  - In **competitor testing,** the new system is tested against an existing system or competitor product.
  - In **shadow testing**, a form of comparison testing, the new and the legacy systems are run in parallel and their outputs are compared.

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 about conditions that have changed and which requirements had to be:
  - **added,**
  - **modified, or**
  - **deleted because of the changes.**
- If the **customer is satisfied, the system is accepted,**
  - possibly **contingent on a list of changes recorded in the minutes of the acceptance test.**

Black-box Approach

- In the black-box approach,
  - test cases are designed using only the **functional specification** of the software.
- Test cases are designed solely based on :
  - analysis of the **input/output behaviour** (that is, functional behaviour)
  - **does not require any knowledge of the internal structure** of a program.
- Also known as **functional testing.**

White-box Approach

- White-box test cases requires:
  - ❑ **a thorough knowledge of the internal structure** of a program,

- Also called **structural testing**.

White/Black box Approach

- Black- box test cases are designed solely based on the **input-output behaviour of a program.**
- In contrast, white-box test cases are based on an **analysis of the code.**
- These two approaches to test case design are **complementary.**
  - That is, a program has to be tested using the test cases designed by both the approaches, and
  - One testing using one approach **does not substitute testing using the other.**

WHITE-BOX TESTING

White-box testing is an important type of unit testing.

A large number of white-box testing strategies exist.

Also called glass-box testing.

Using white-box testing methods, you can derive test cases that
(1) guarantee that **all independent paths** within a module have been exercised at least once,
(2) exercise **all logical decisions** on their true and false sides,
(3) execute **all loops** at their boundaries and within their operational bounds, and
(4) exercise **internal data structures** to ensure their validity.

WHITE-BOX TESTING

It includes:

- **BASIS PATH TESTING**
- **CONTROL STRUCTURE TESTING**

Basis path testing

- *Basis path testing* is a white-box testing technique first proposed by Tom McCabe.
- The basis path method enables the test-case designer to **derive a logical complexity measure of a procedural design.**

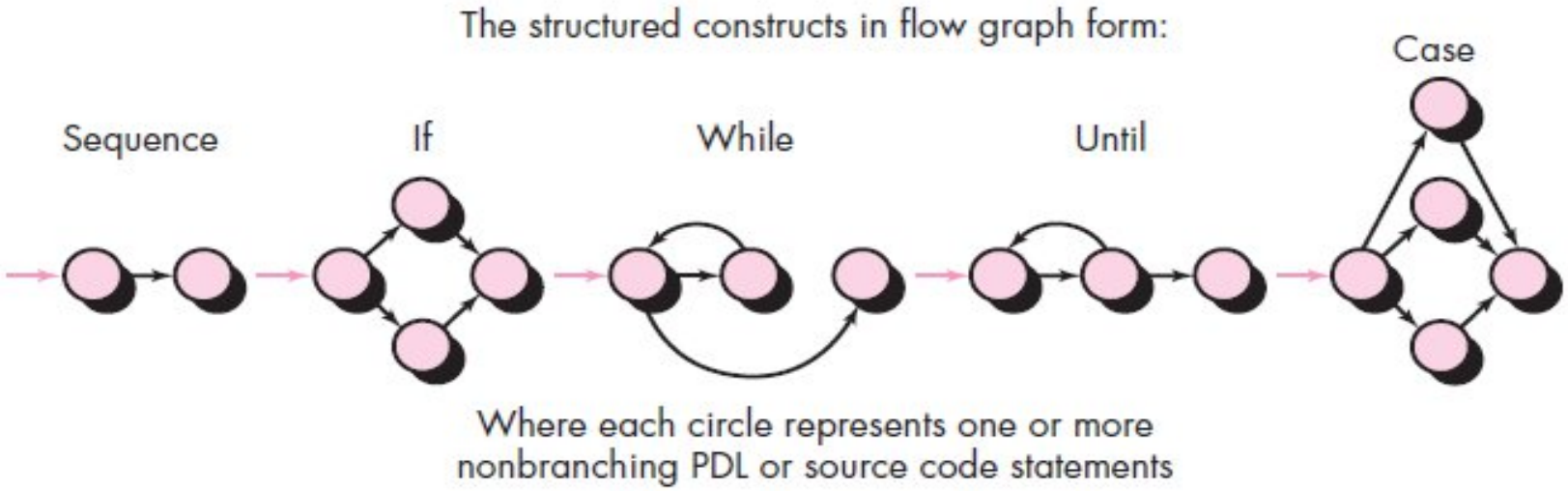
BASIS PATH TESTING

It includes:

- **FLOW GRAPH NOTATION**
- **INDEPENDENT PROGRAM PATHS**
- **DERIVING TEST CASES**
- **GRAPH MATRICES**

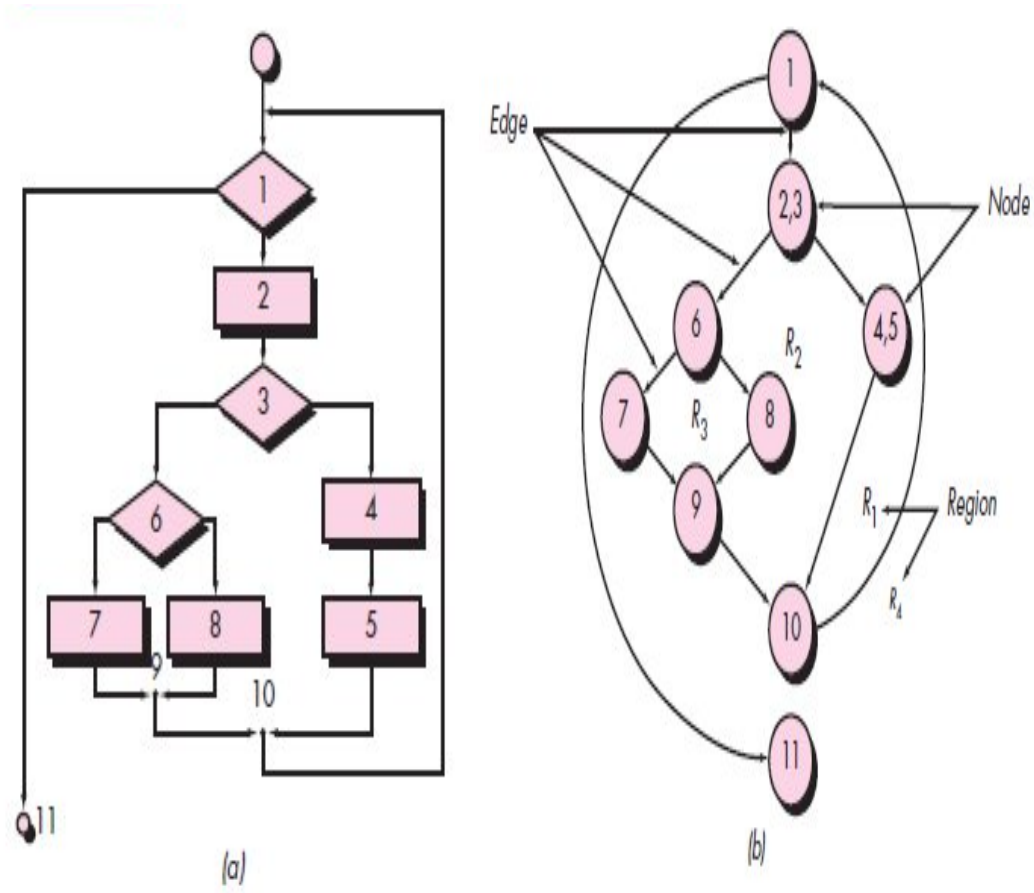
Flow Graph Notation

- The flow graph depicts logical control flow using the notation
- **Each structured construct has a corresponding flow graph symbol.**



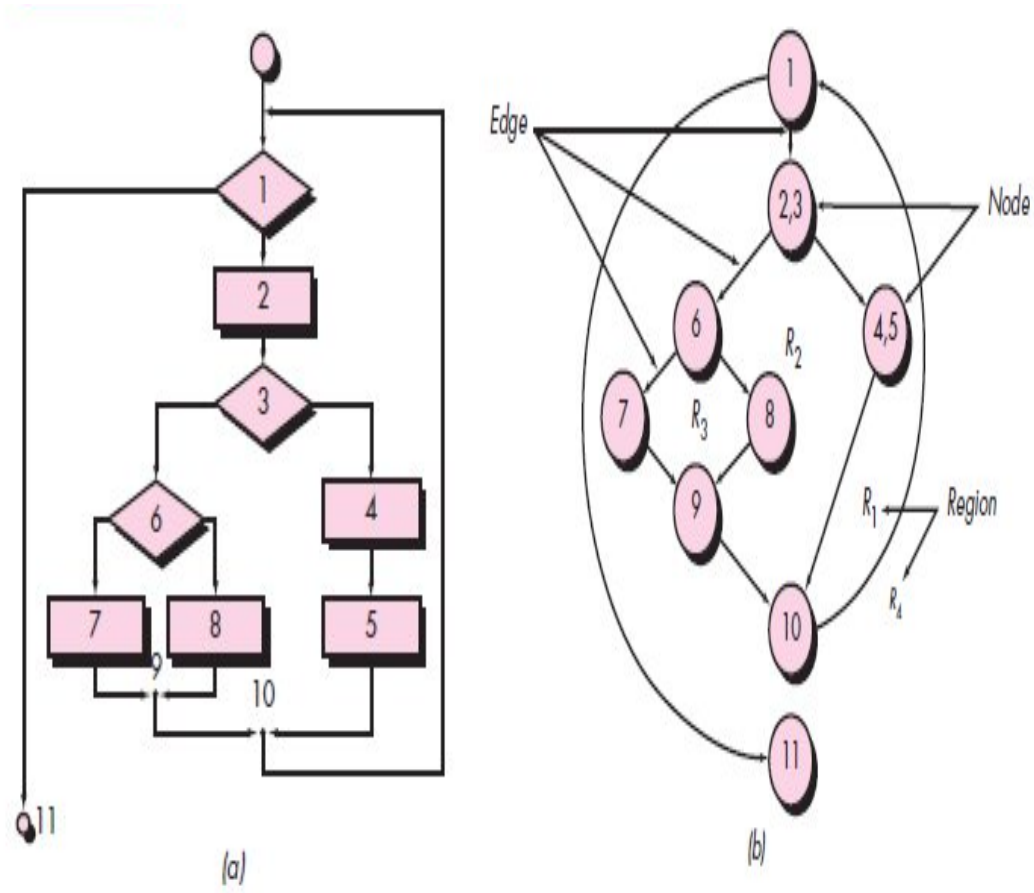
Flow Graph Notation

- Procedural design in Figure (a).
  - A flowchart is used to depict program control structure
- Figure(b) maps the flowchart into a flow graph



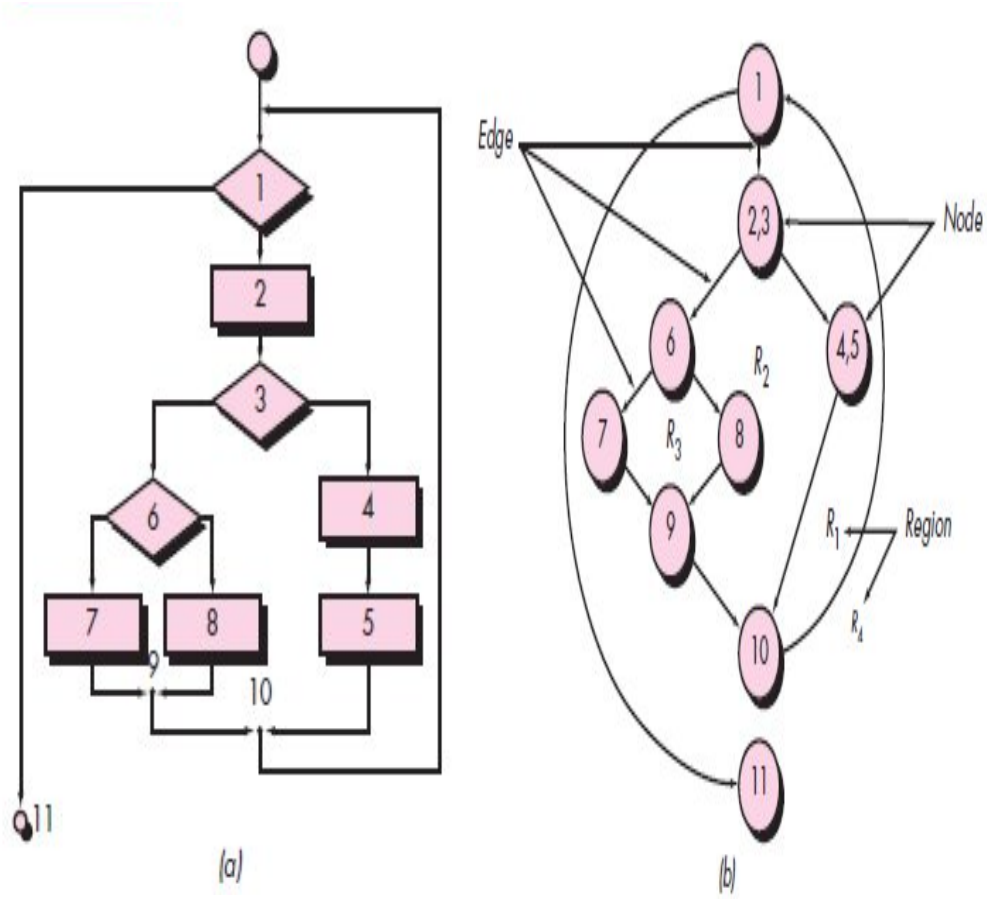
Flow Graph Notation

- **Each circle, called a *flow graph node*,**
  - represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- Check 2,3
- Check 4,5



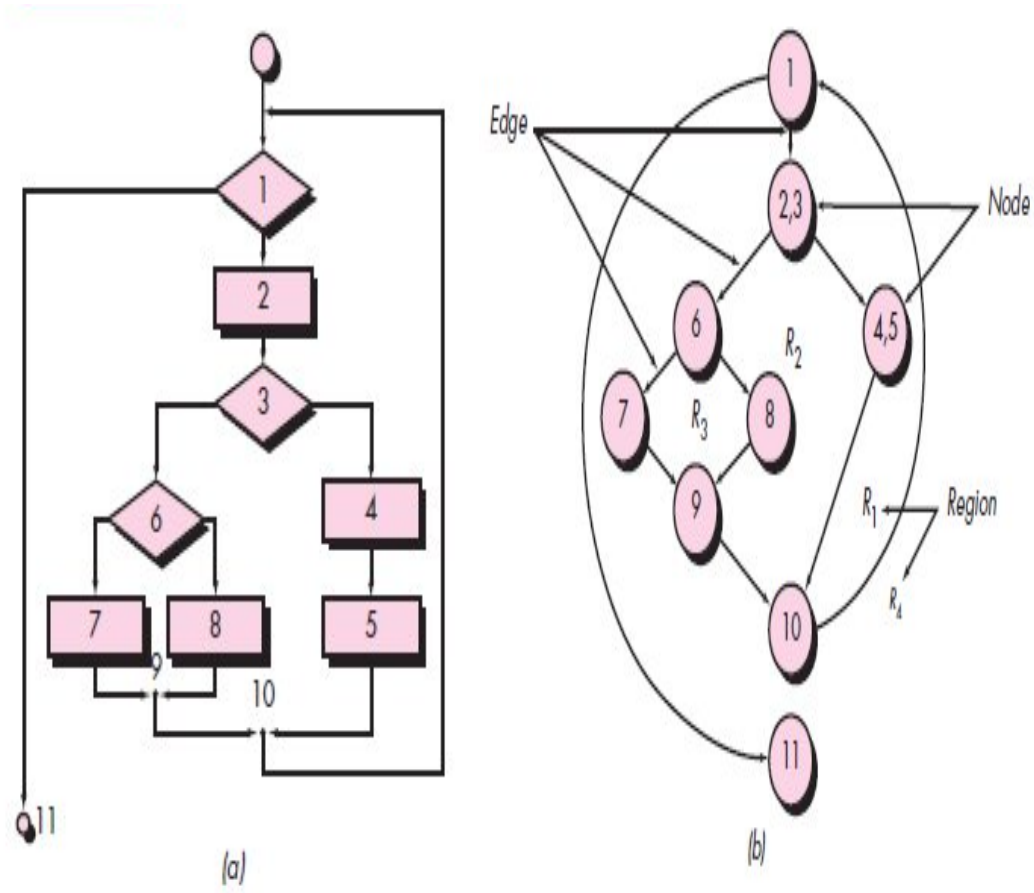
Flow Graph Notation

- The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows.



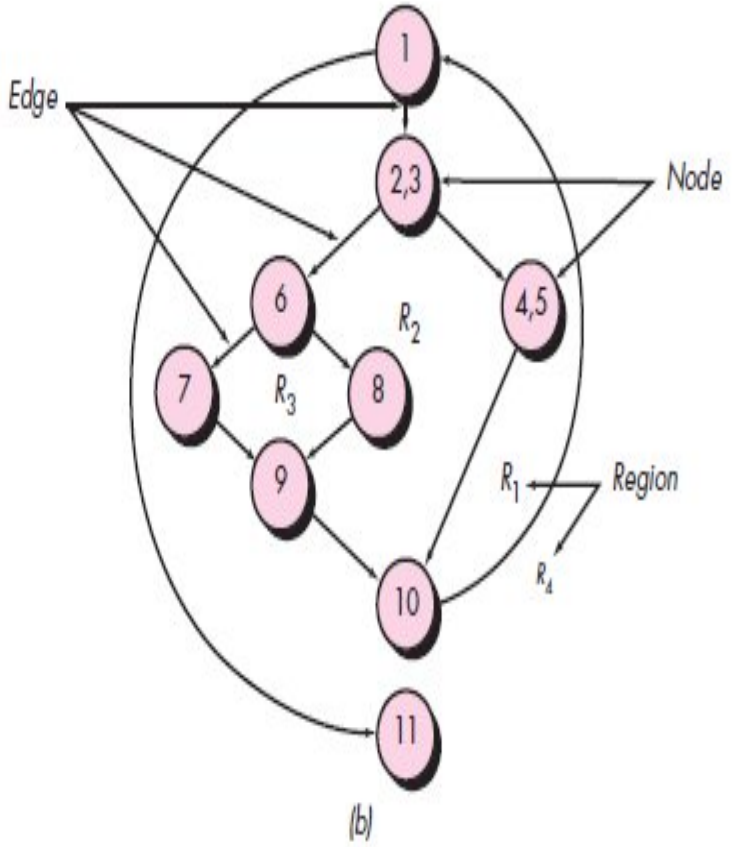
Flow Graph Notation

- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct).
- Check 9, 10



Flow Graph Notation

- **Areas bounded by edges and nodes are called *regions*.**
- When counting regions, we include the **area outside the graph as a region**



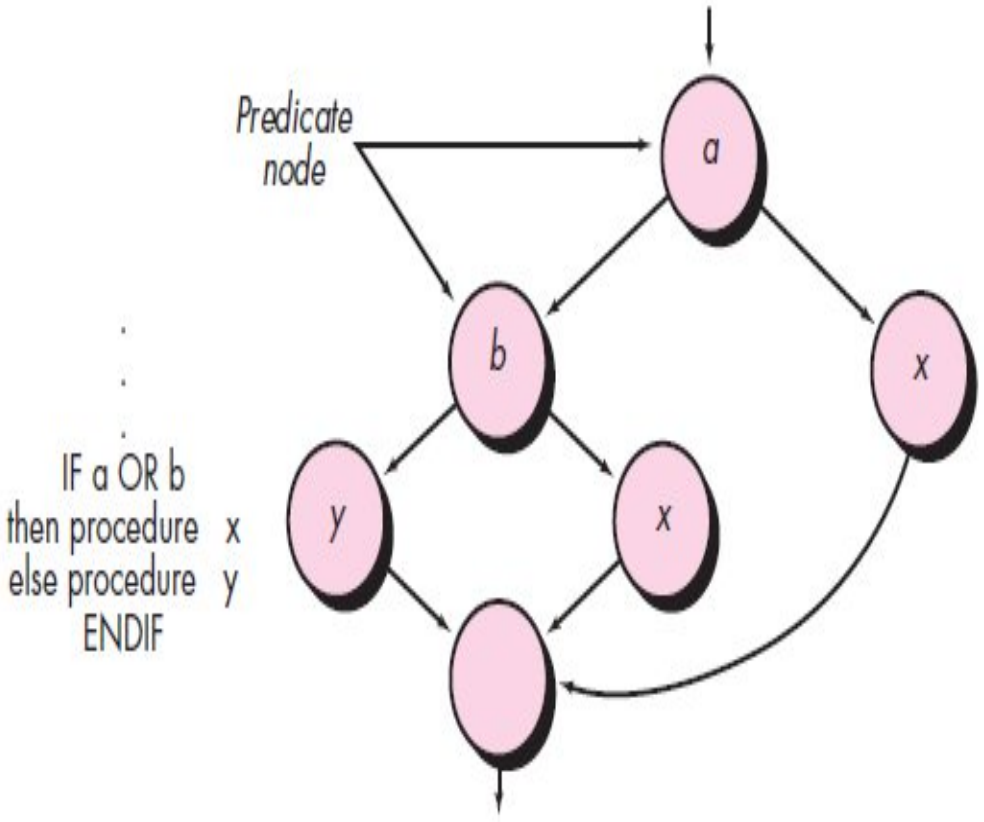
Flow Graph Notation

- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.

- A compound condition occurs when
  - one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.

Flow Graph Notation

- A separate node is created for each of the conditions *a* and *b* in the statement
  - IF *a* OR *b*.
- Each node that contains a condition is called a *predicate node* and
  - **is characterized by two or more edges emanating from it.**

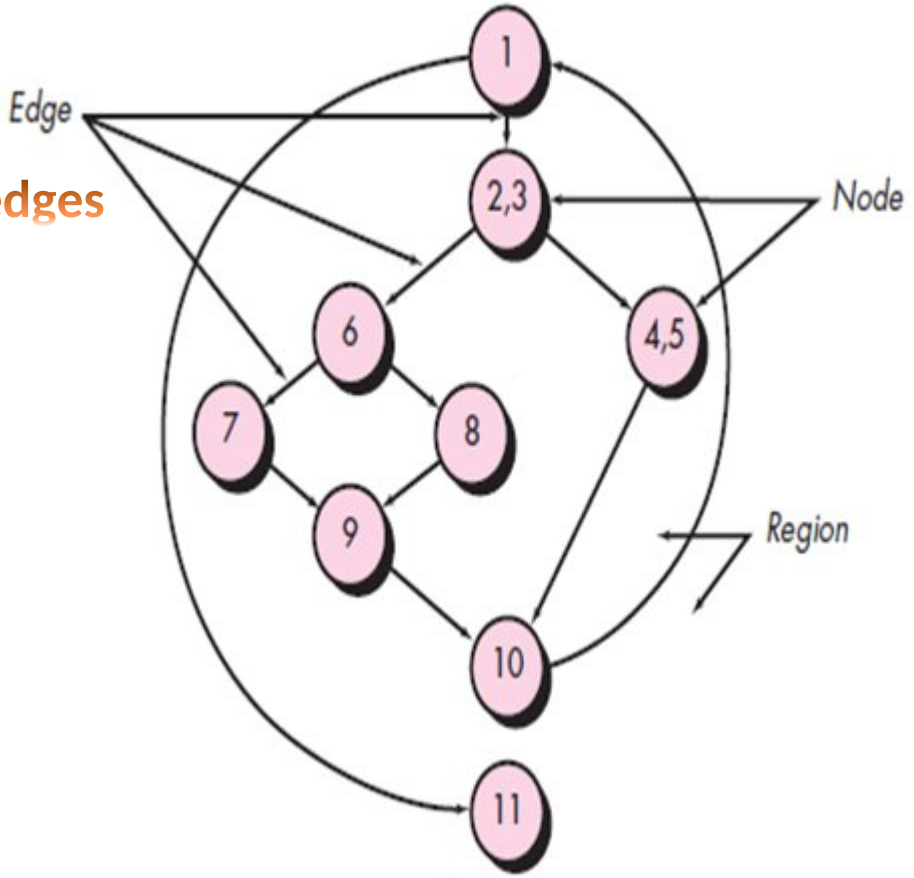


Independent Program Paths

- An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition.
- When stated in terms of a flow graph,
  - an independent path must move along **at least one edge that has not been traversed before** the path is defined.

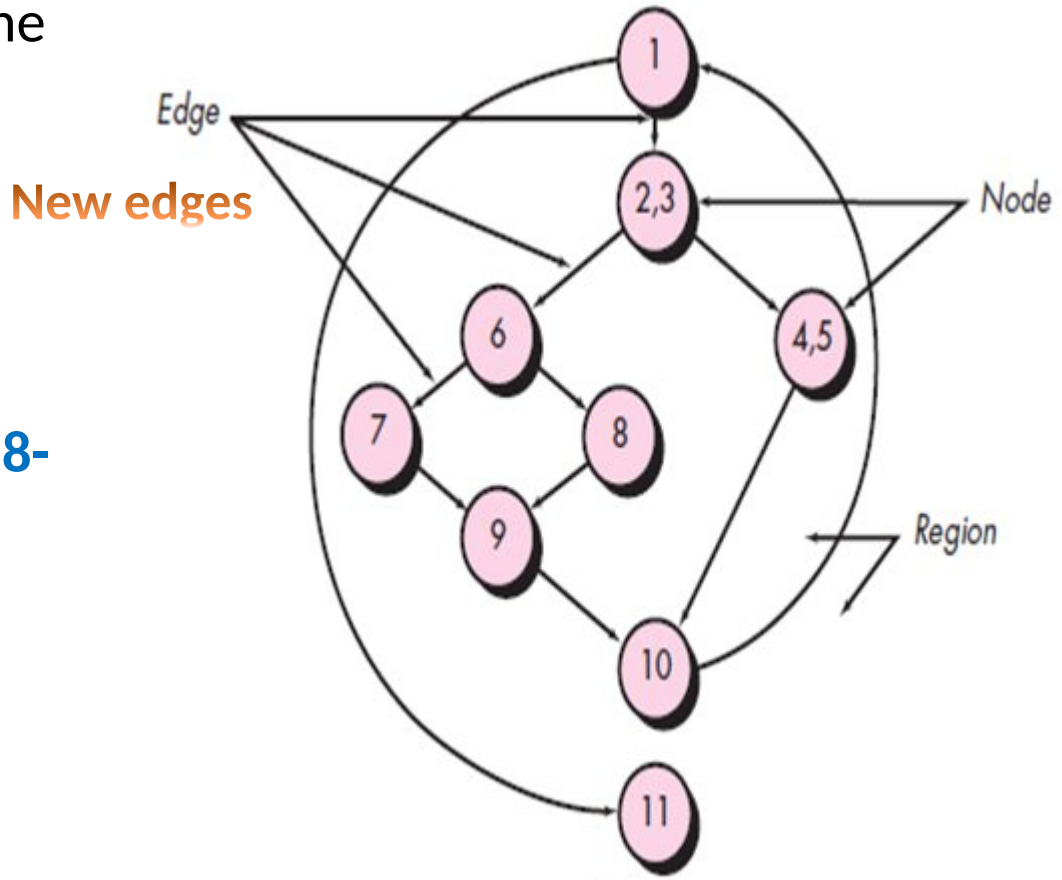
Independent Program Paths

- Set of independent paths for the flow graph in Figure is
  - **Path 1: 1-11**
  - **Path 2: 1-2-3-4-5-10-1-11** **New edges**
  - **Path 3: 1-2-3-6-8-9-10-1-11**
  - **Path 4: 1-2-3-6-7-9-10-1-11**
- Note that each new path introduces a new edge.
- What about the **path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11** ?



Independent Program Paths

- Set of independent paths for the flow graph in Figure is
  - **Path 1: 1-11**
  - **Path 2: 1-2-3-4-5-10-1-11**
  - **Path 3: 1-2-3-6-8-9-10-1-11**
  - **Path 4: 1-2-3-6-7-9-10-1-11**
- Note that each new path introduces a new edge.
- The **path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11** is not considered to be an independent path
  - combination of already specified paths and does not traverse any new edges.



How do you know how many paths to look for?

Cyclomatic complexity

Cyclomatic complexity

- *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines:
  - the **number of independent paths** in the basis set of a program and
  - provides you with an **upper bound for the number of tests that must be conducted** to ensure that all statements have been executed at least once.

Cyclomatic complexity

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*) =*E*-*N*+2**

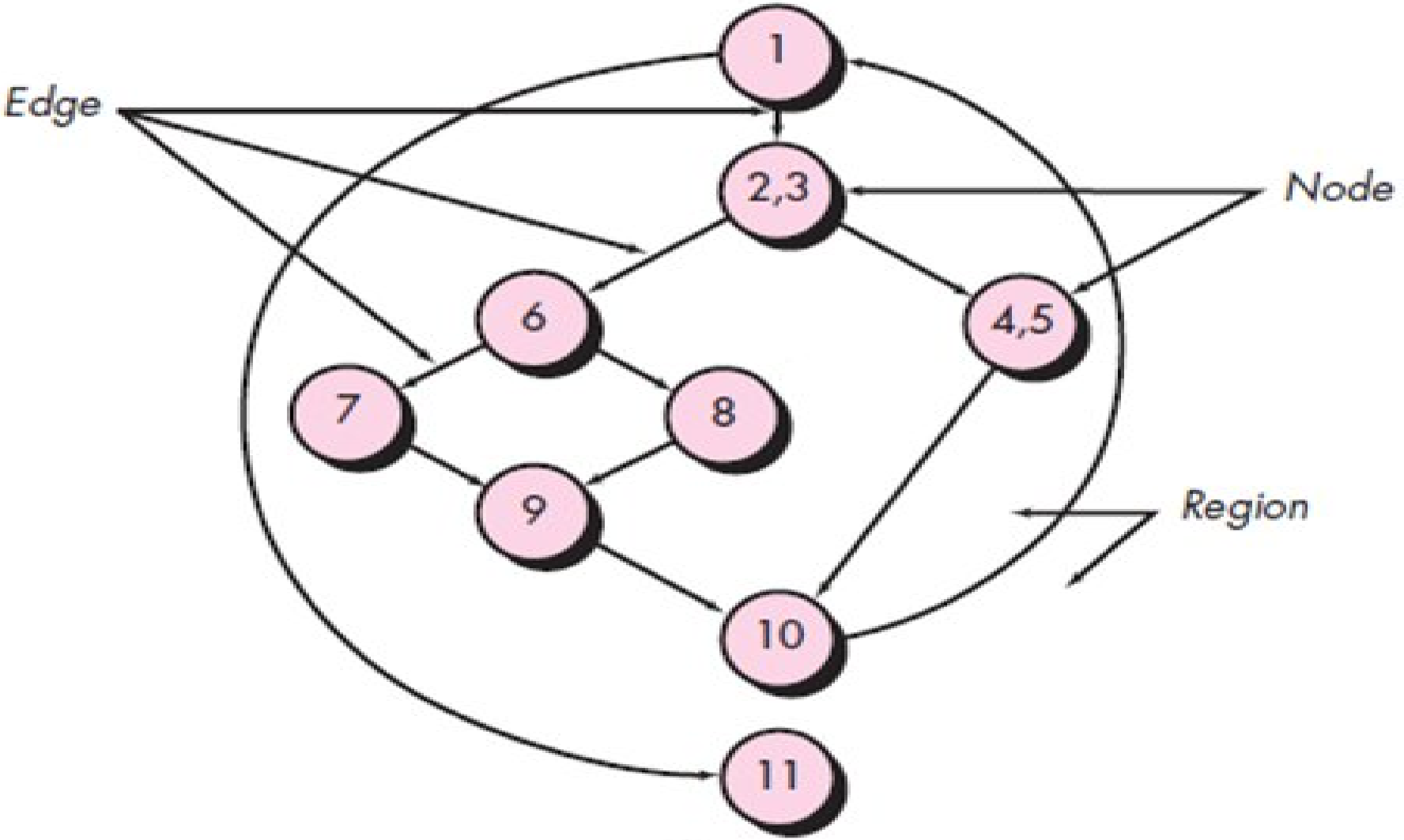
- where *E* is the number of flow graph edges and *N* is the number of flow graph nodes.

3. Cyclomatic complexity *V*(*G*) for a flow graph *G* is also defined as ***V*(*G*)=*P*+1**

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

Cyclomatic complexity

Find Out the Cyclomatic Complexity of the Figure



Cyclomatic complexity

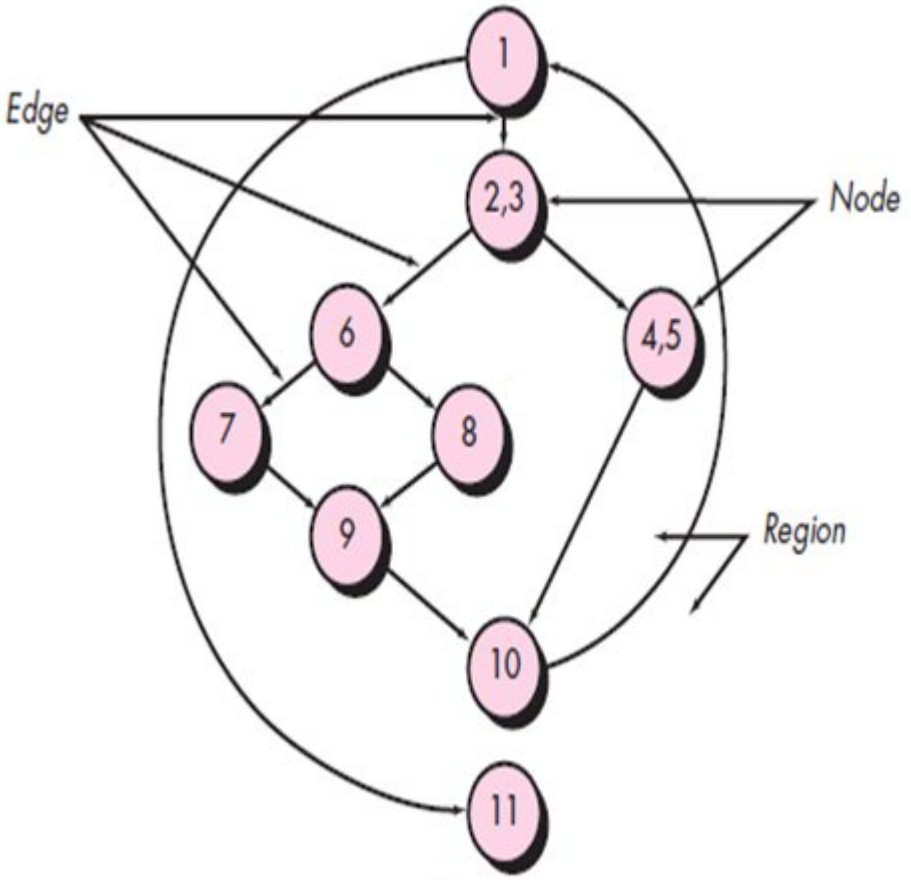
For Figure , the cyclomatic complexity can be computed as:

1. The flow graph has four regions.

2. *V*(*G*) =11 edges - 9 nodes + 2 = 4.

3. *V*(*G*) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure is 4.

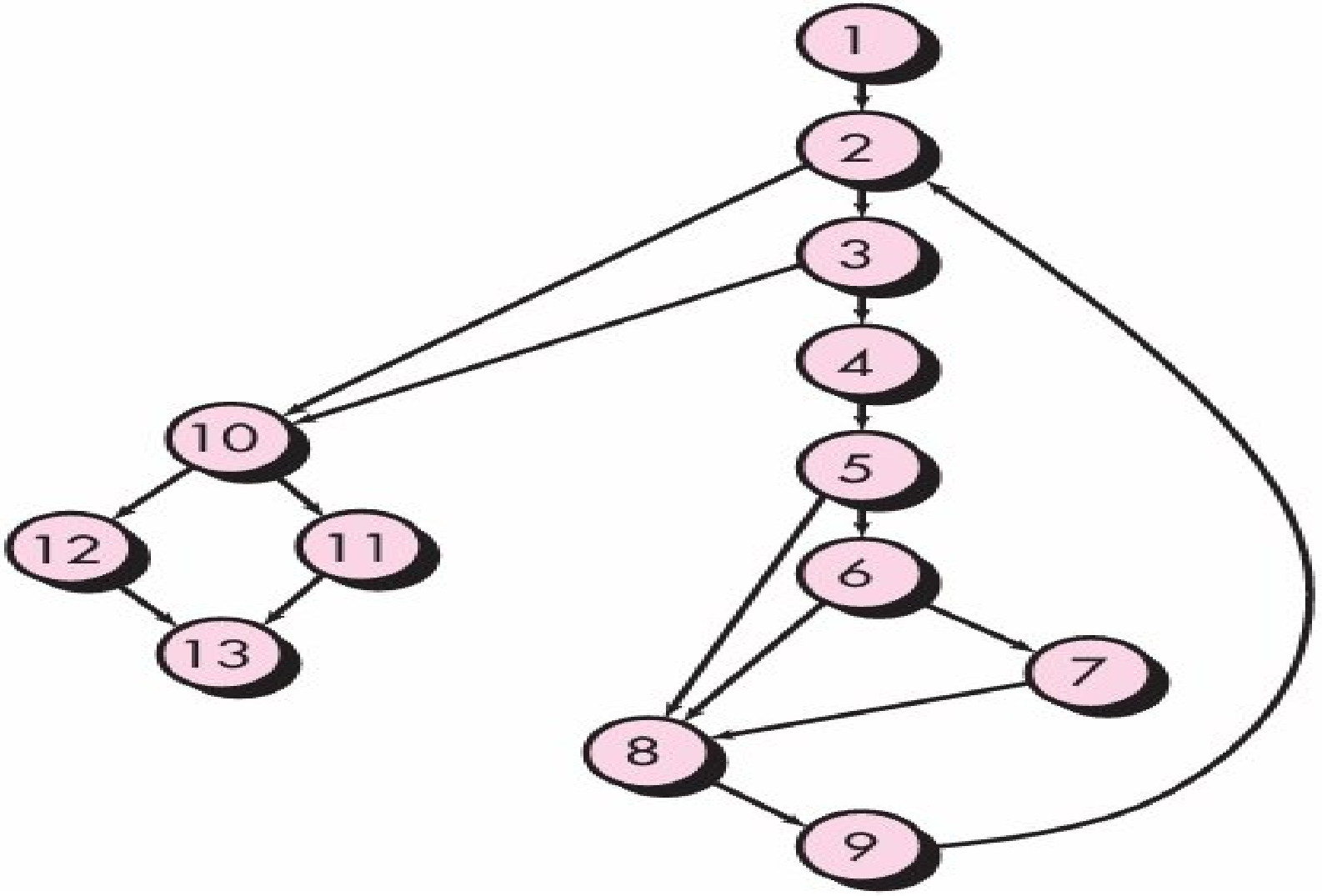


Deriving Test Cases

1) **Using the design or code as a foundation, draw a corresponding flow graph.**
2) **Determine the cyclomatic complexity of the resultant flow graph.**
3) **Determine a basis set of linearly independent paths.**
4) **Prepare test cases that will force execution of each path in the basis set.**

Cyclomatic complexity

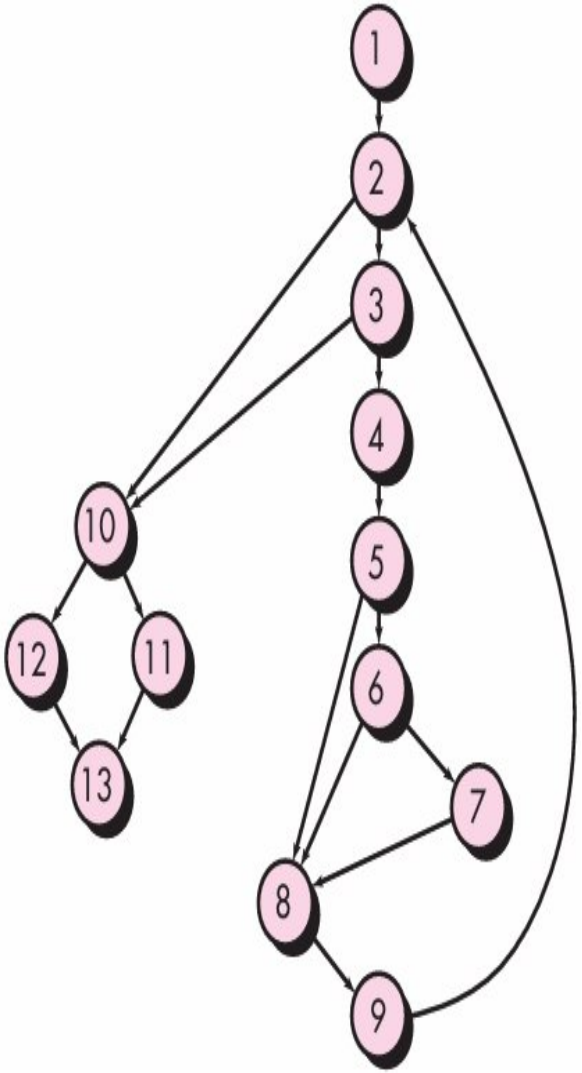
Find Out the Cyclomatic Complexity of the Figure



Cyclomatic complexity

Find Out the Cyclomatic Complexity of the Figure

- V(G) =6 regions
- V(G) =17 edges-13 nodes +2= 6
- V(G) =5 predicate nodes + 1= 6



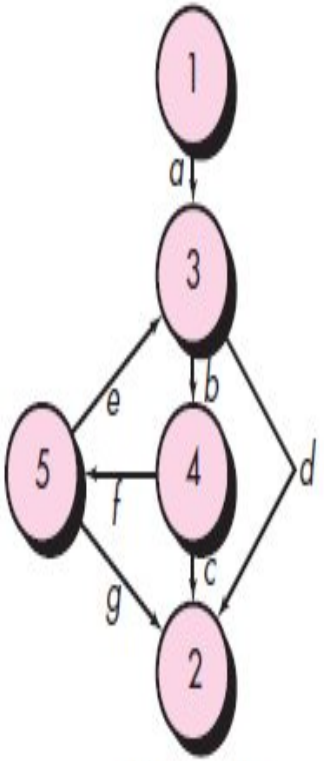
Graph Matrices

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

- A graph matrix is:
  - a square matrix **whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.**
  - Each row and column **corresponds to an identified node,**
  - Matrix entries correspond to **connections (an edge)between nodes**.

Graph Matrices

- A flow graph and its corresponding graph matrix.
  - **Each node on the flow graph is identified by numbers,**
  - **Each edge is identified by letters.**
  - **A letter entry is made in the matrix** to correspond to a connection between two nodes.
  - For example, node 3 is connected to node 4 by edge *b*.

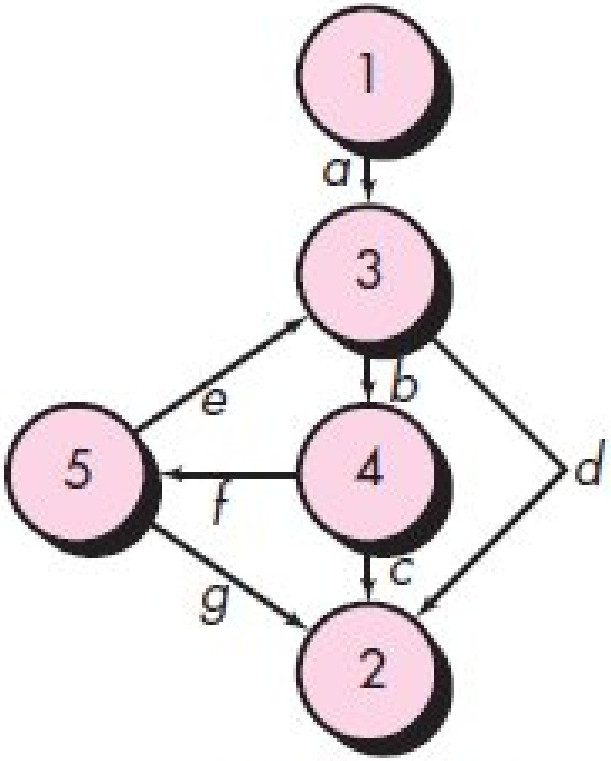


Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Graph Matrices



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Graph Matrices

- To this point, the graph matrix is nothing more than a tabular representation of a flow graph.
- **By adding a *link weight*** to each matrix entry, the graph matrix can **become a powerful tool for evaluating program** control structure during testing.

- The link weight provides additional information about control flow.

- In its simplest form,
  - **the link weight is 1 (a connection exists) or**
  - **0 (a connection does not exist).**

Graph Matrices

But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.

BASIS PATH TESTING

It includes:

- **FLOW GRAPH NOTATION**
- **INDEPENDENT PROGRAM PATHS**
- **DERIVING TEST CASES**
- **GRAPH MATRICES**

CONTROL STRUCTURE TESTING

- Although basis path testing is simple and highly effective, it is not sufficient in itself.
- Control Structure Testing broadens testing coverage and improve the quality of white-box testing.
- It includes:
  - **CONDITION TESTING**
  - **DATA FLOW TESTING**
  - **LOOP TESTING**

Condition Testing

Condition testing is a test-case design method that exercises the logical conditions contained in a program module.

A SIMPLE CONDITION-

- is a Boolean variable or a relational expression,
  - A relational expression takes the form
    - *E*1 <relational-operator> *E*2
    - *E*1 and *E*2 are arithmetic expressions
    - relational-operators are < , <=,> ,>= (nonequality), or =

Condition Testing

COMPOUND CONDITION-

- is composed of two or more simple conditions,
- Boolean/Logical operators, and
- Parentheses.

- Boolean/Logical operators allowed in a compound condition include
  - OR (),
  - AND (&),
  - NOT (¬)

Condition Testing

- **IF A CONDITION IS INCORRECT**
  - then **at least one component of the condition is incorrect**.
- Types of errors in a condition include
  - Boolean/Logical operator errors (incorrect/missing/extra Boolean operators),
  - Boolean variable errors,
  - Parenthesis errors,
  - relational operator errors, and
  - arithmetic expression errors.
- Testing each condition in the program to ensure that it does not contain errors.

Data Flow Testing

- **Selects test paths of a program according to the locations of definitions and uses of variables in the program.**

Assumption:

- **Each statement in a program is assigned a unique statement number**
- Each function does not modify its parameters or global variables.
- For a statement with *S* as its statement number,
  - **DEF(*S*) {*X* | statement *S* contains a definition of *X*}**
  - **USE(*S*) {*X* | statement *S* contains a use of *X*}**

DU

- A ***definition-use (DU) chain* of variable *X*** is of the form **[*X*, *S*, *S'*]**, where:
  - *S* and *S'* are statement numbers,
  - *X* is in DEF(*S*) and USE(*S'*),
  - the definition of *X* in statement *S* is live at statement *S'*.

Data Flow Testing

- If statement *S* is an *if* or *loop statement*,
  - ❑ its DEF set is empty and
  - ❑ its USE set is based on the condition of statement *S*.

- The **definition of variable *X* at statement *S* is said to be *live* at statement *S'***
  - **if there exists a path from statement *S* to statement *S'* that contains no other definition of *X*.**

DU TESTING STRATEGY

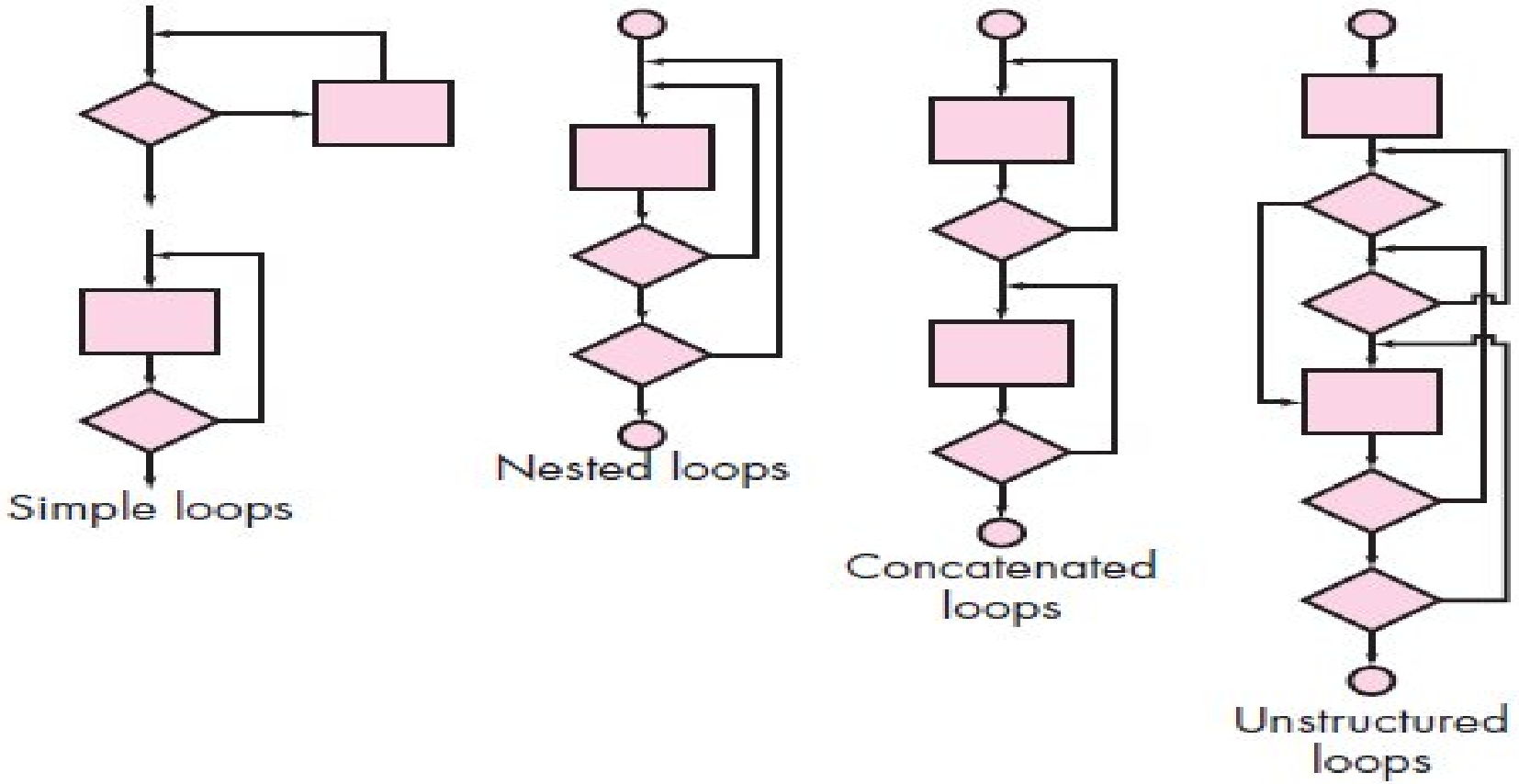
- Data flow testing strategy is to require that **EVERY DU CHAIN BE COVERED AT LEAST ONCE.**

- **DU TESTING DOES NOT GUARANTEE THE COVERAGE OF ALL BRANCHES OF A PROGRAM.**
- However, a branch is not guaranteed to be covered by DU testing only in rare situations such as
  - if-then-else constructs in which the
  - *then part* has no definition of any variable and
  - the *else part* does not exist.
  - In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

Loop Testing

- Loops are part of majority of all algorithms implemented in software.
  - yet, we often **pay them little heed** while conducting software tests.
- *Loop testing* is a white-box testing technique that focuses exclusively on **the validity of loop constructs.**
- Four different classes of loops can be defined:
  - **Simple loops**
  - **Concatenated loops**
  - **Nested loops**
  - **Unstructured loops**

Loop Testing



Simple loops

The following **set of tests can be applied to simple loops**, where ***n* is the maximum number of allowable passes through the loop**.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. *m* passes through the loop where $m<n$.
5. $n-1$, n, $n+1$ passes through the loop.

Nested loops

- If we were to extend the test approach for simple loops to nested loops :
  - the number of possible tests would grow geometrically as the level of nesting increases.
  - This would result in an impractical number of tests.

Nested loops

Beizer suggests an approach to reduce the number of tests:

1. **Start at the innermost loop.**
   - Set all **other loops to minimum** values.
2. Conduct **simple loop tests for the innermost loop** while
   - holding the outer loops at their minimum iteration parameter (e.g., loop counter) values.
   - Add other tests for out-of-range or excluded values.
3. **Work outward**,
   - conducting tests for the next loop, but
   - keeping **all other outer loops at minimum values** and
   - other nested loops to "typical" values.
4. Continue **until all loops have been tested**.

Concatenated loops

- Concatenated loops can be tested
- If each of the loops **is independent of the other.**
  - using the approach defined for **simple loops,**
- When the loops are **not independent,**
  - **the approach applied to nested loops is recommended**.
- If two loops are concatenated and
  - the loop counter for loop 1 is used as the initial value for loop 2,
  - then the loops are not independent.

Unstructured loops

- Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

Black-box testing

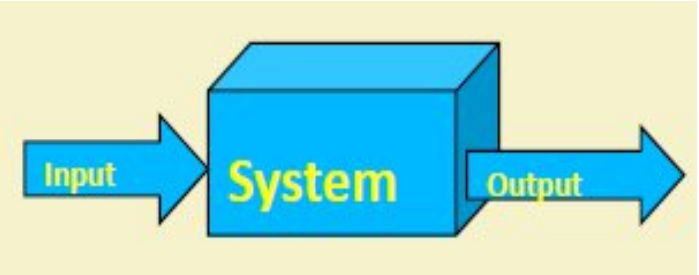
Black-box testing

Software considered as a black box:

- Test data derived from the specification
- **No knowledge of code necessary**

Also known as:

- **Data-driven or**
- **Input/output driven testing**



The goal is to achieve the thoroughness of exhaustive input testing:

- **With much less effort!!!!**

Black-box testing

- Also called
  - *behavioral testing.*
  - *functional testing.*
- Focuses on the **functional requirements of the software**.
- Black-box testing techniques enable you to **derive sets of input conditions that will fully 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, and
  (5) initialization and termination errors.

Black-box testing

Tests are designed to answer the following questions:

- How is functional validity tested?
- **How are system behavior and performance tested?**
- **What classes of input will make good test cases?**

?

- **Is the system particularly sensitive to certain input values?**
- **How are the boundaries of a data class isolated?**
- **What data rates and data volume can the system tolerate?**
- **What effect will specific combinations of data have on system operation?**

?

Black-box testing

It includes:

1) **EQUIVALENCE PARTITIONING**
2) **BOUNDARY VALUE ANALYSIS**
3) **GRAPH-BASED TESTING METHODS**

Equivalence Partitioning

- *Equivalence partitioning* is a black-box testing method
  - that **divides the input domain of a program into classes of data** from which test cases can be derived.

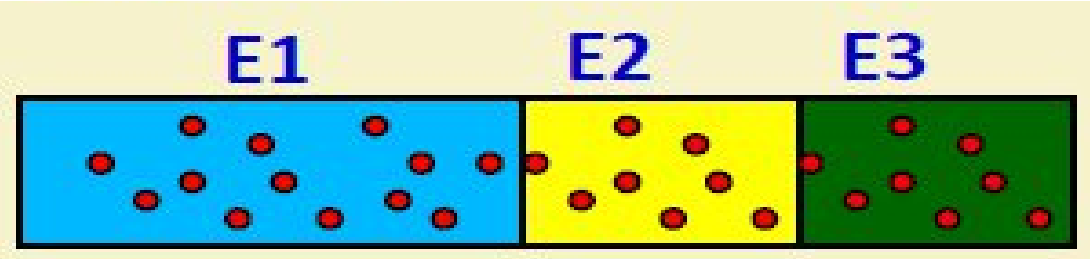
Equivalence Class Partitioning

- The input values to a program:
  - Partitioned into equivalence classes.
- Partitioning is done such that:
  - **Program behaves in similar ways to every input value belonging to an equivalence class.**
  - **At the least, there should be as many equivalence classes as scenarios.**

Why Define Equivalence Classes?

Premise:

- Testing code with **any one representative value from a equivalence class**:
- As good as **testing using any other values** from the equivalence class.



Equivalence Partitioning

- Test-case design is based on an **evaluation of *equivalence classes* for an input condition.**
- If a **set of objects can be linked** by relationships that are
  - **symmetric,**
  - **transitive,**
  - **reflexive,**

  an equivalence class is present

WHEN?

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

- An input condition is either
  - ❑ a specific numeric value,
  - ❑ a range of values,
  - ❑ a set of related values, or
  - ❑ a Boolean condition.

WHAT?

Equivalence Class Partitioning

- How do you identify equivalence classes?
  - **Identify scenarios**
  - **Examine the input data.**
  - **Examine output**

- Few guidelines for determining the equivalence classes can be given…

Guidelines for Equivalence classes

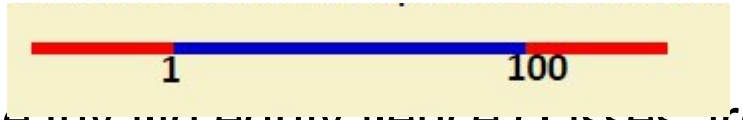
1. If an input condition **specifies a range**, one valid and two invalid equivalence classes are defined.
2. If an input condition requires **a specific value,** one valid and two invalid equivalence classes are defined.

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.

These Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Guidelines to Identify Equivalence Classes

- If an input is a range, one valid and two invalid equivalence classes are defined.
  - Example: 1 to 100



• If an input is a set, one valid and one invalid equivalence classes are defined.
  - **Example: {a,b,c}**

• If an input is a Boolean value, one valid and one invalid class are defined.

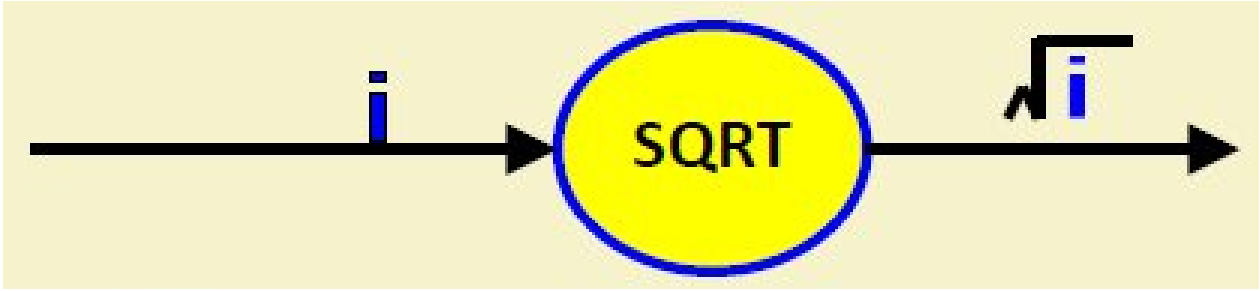
Example:

•**Area code:** input value defined between 10000 and 90000--- **RANGE**

•**Password:** string of six characters --- **SET**

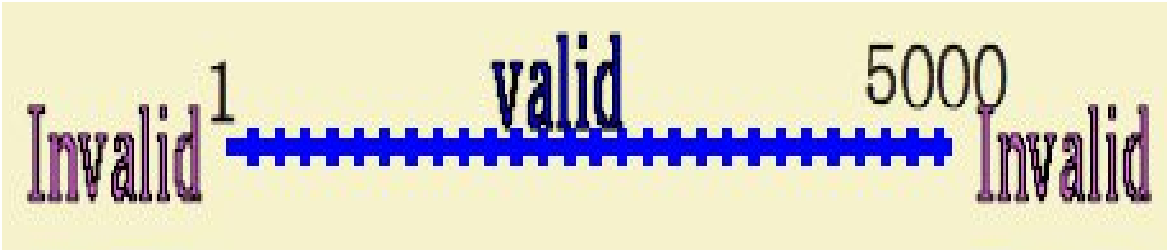
Example

- A program reads an input value in the range of 1 and 5000:
  - Computes the **square root of the input number**



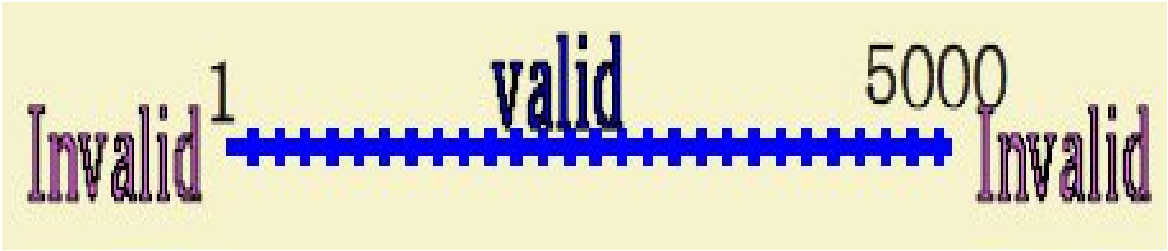
Example (cont.)

- Three equivalence classes:
  - The set of negative integers,
  - Set of integers in the range of 1 and5000,
  - Integers larger than 5000.



Example (cont.)

- The test suite must include:
  - Representatives from each of the three equivalence classes:
  - A possible test suite can be: {-5,500,6000}.



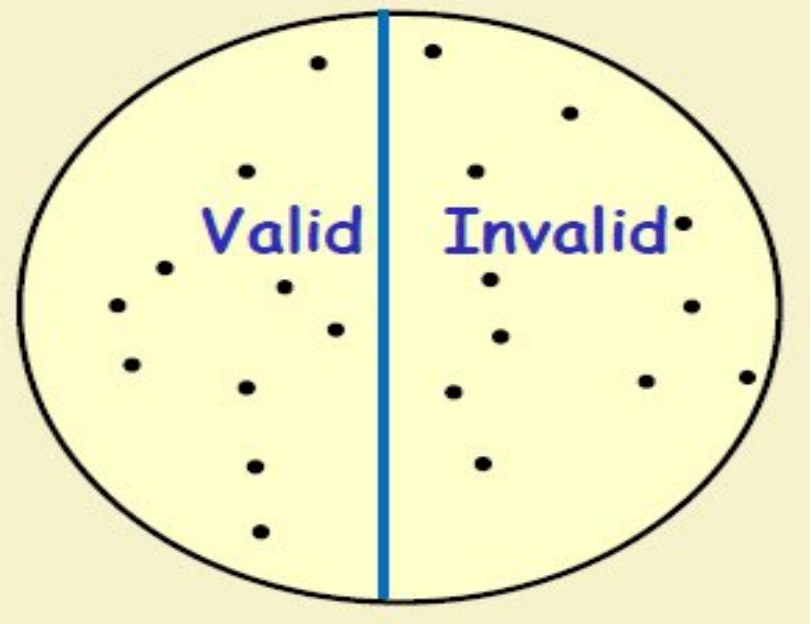
Equivalence Class Partitioning

- If input is an enumerated set of values, e.g.
  - {a,b,c}

- Define:
  - One equivalence class for valid input values.
  - Another equivalence class for invalid input values..

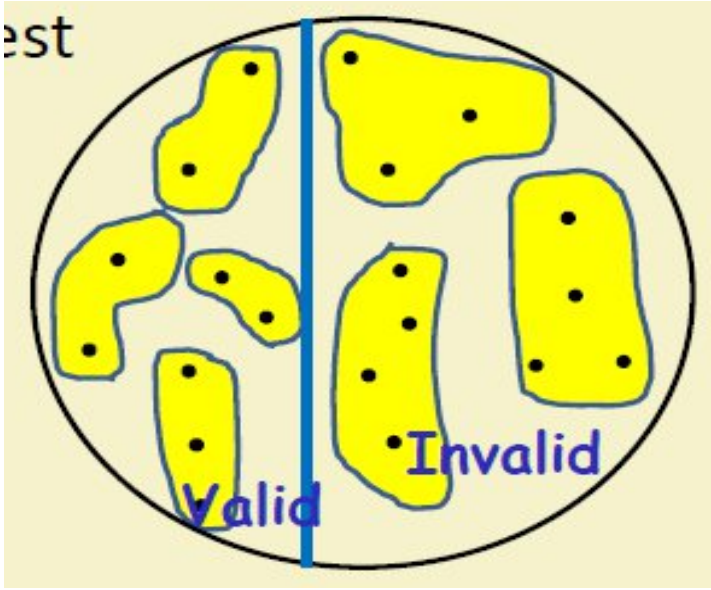
Equivalence Partitioning

- First-level partitioning:
  - Valid vs. Invalid test cases



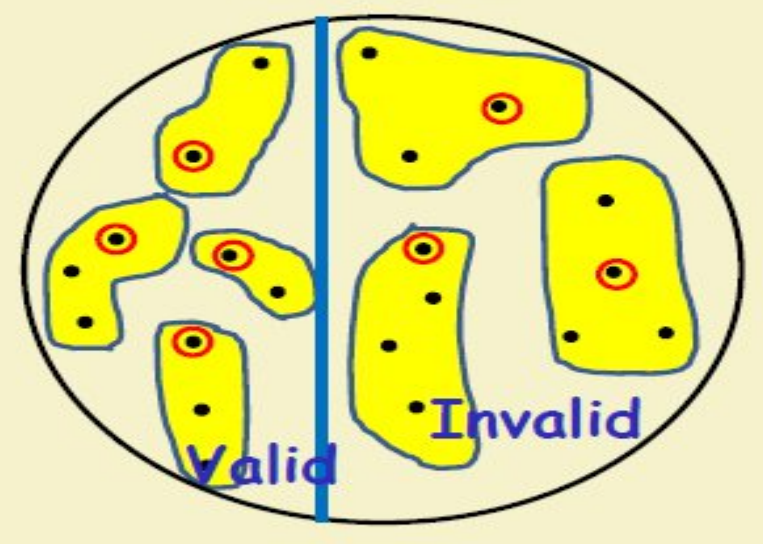
Equivalence Partitioning

- Further partition valid and invalid test cases into equivalence classes



Equivalence Partitioning

- Create a test case using at least one value from each equivalence class

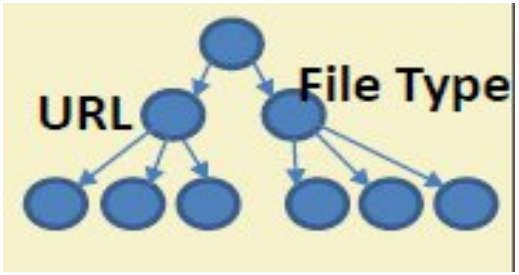


Equivalence Partitioning

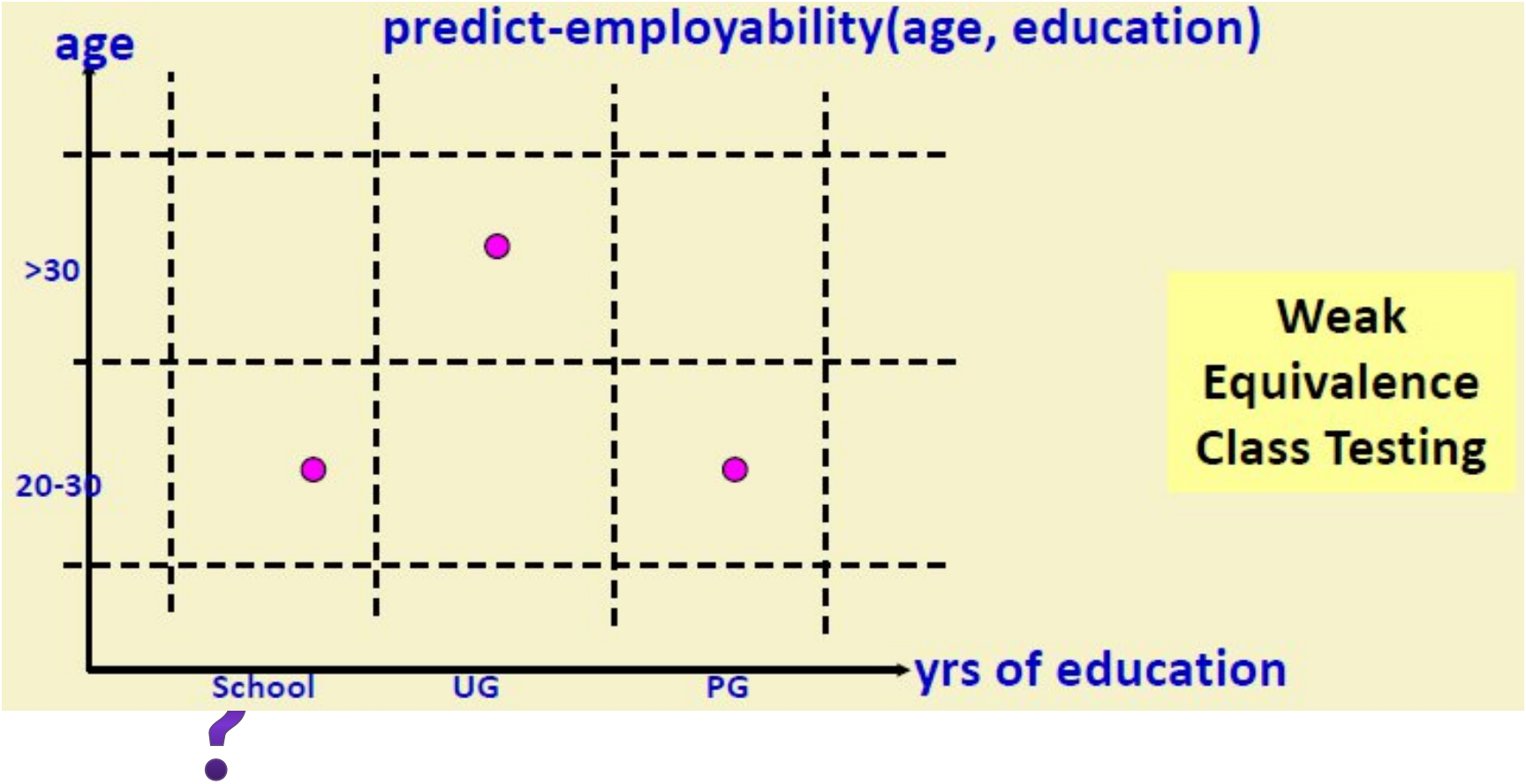
- **A set of input values constitute an equivalence class if the tester believes that these are processed identically:**

Equivalence Partitioning: Example 2

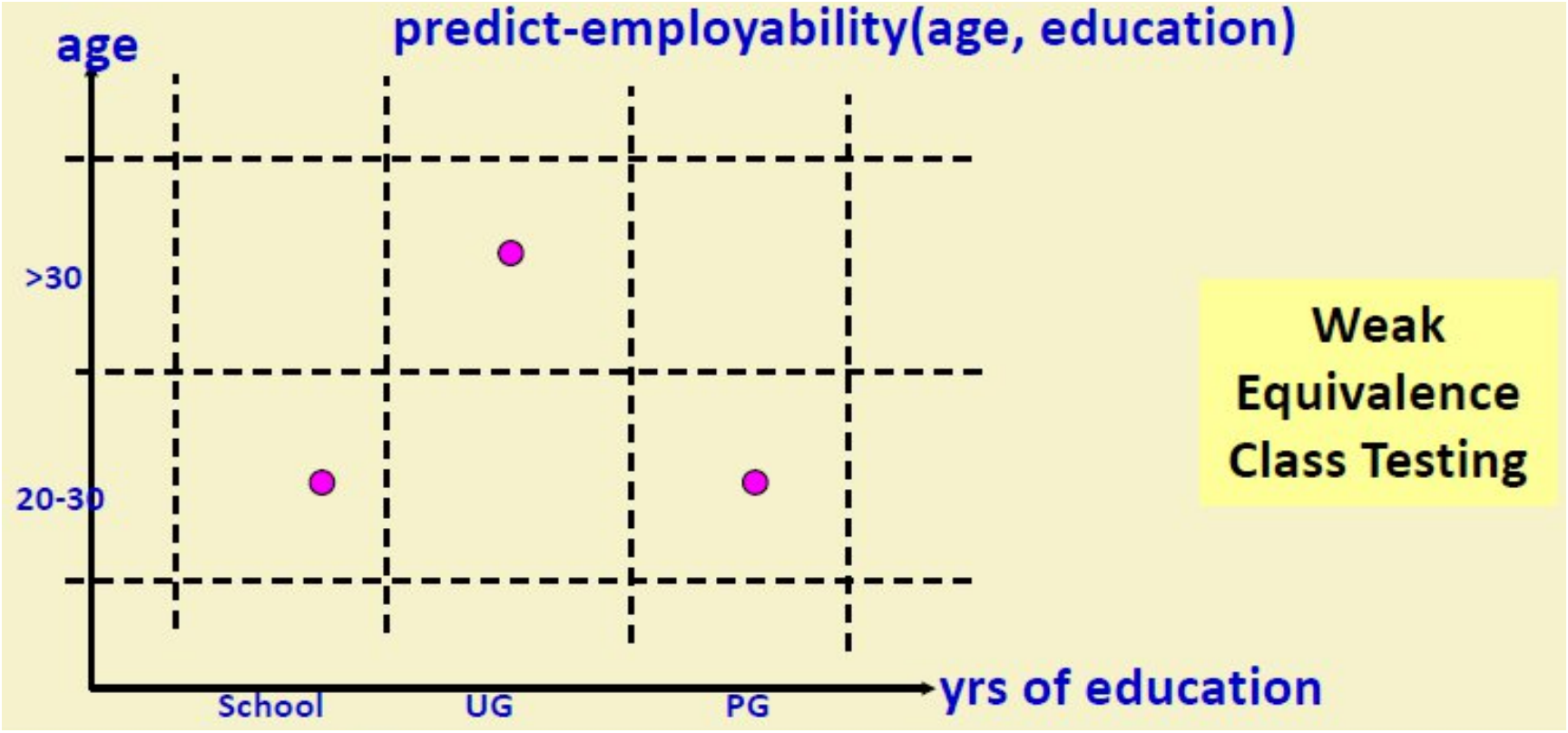
- Example: **Image Fetch-image(URL)**



  - **Equivalence Definition 1:** Partition based on URL protocol (“http”, “https”, “ftp”, “file”, etc.)
  - **Equivalence Definition 2:** Partition based on type of file being retrieved (HTML, GIF, JPEG, Plain Text, etc.)

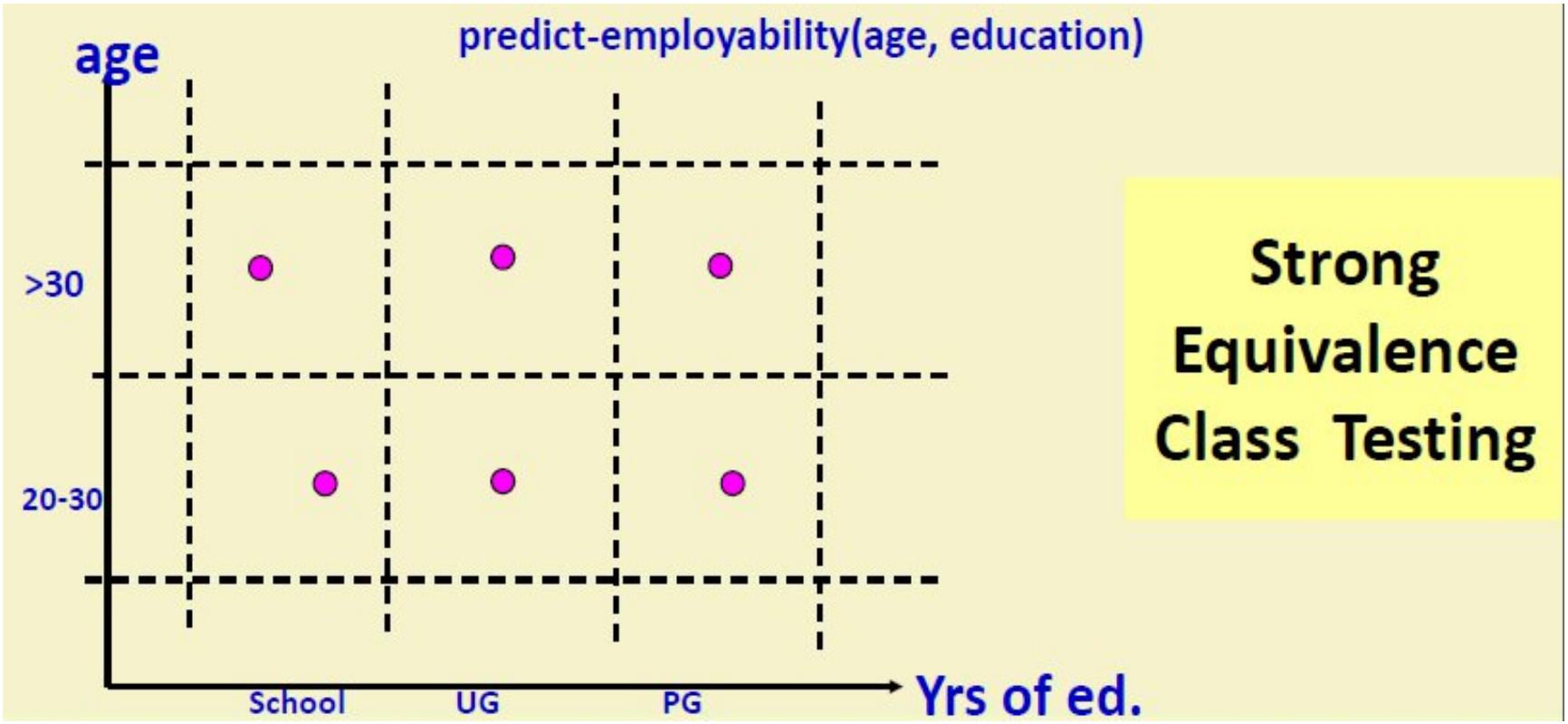

predict-employability(age, education)
age
>30
20-30
School
UG
PG
yrs of education
Weak
Equivalence
Class Testing

- Once we identify Equivalence Classes for the two parameters
- Combine the representative values for these two parameters
- Weak:
  - **Representative from every EC is present.**

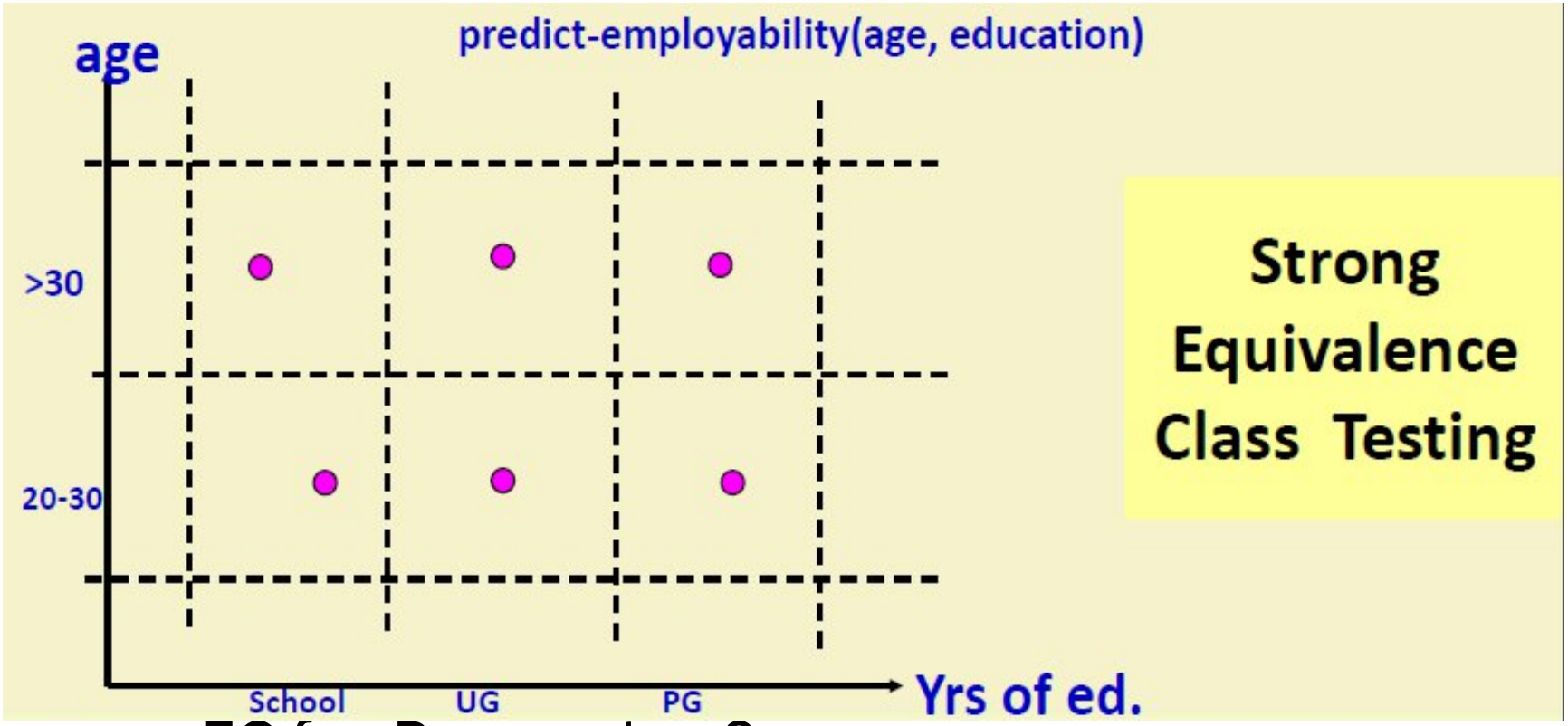


- m= EC for Parameter 2
- No of test cases=max(n,m)

n= 3 for yrs of education
m=2 for age
No of test cases=Max(3,2)=3

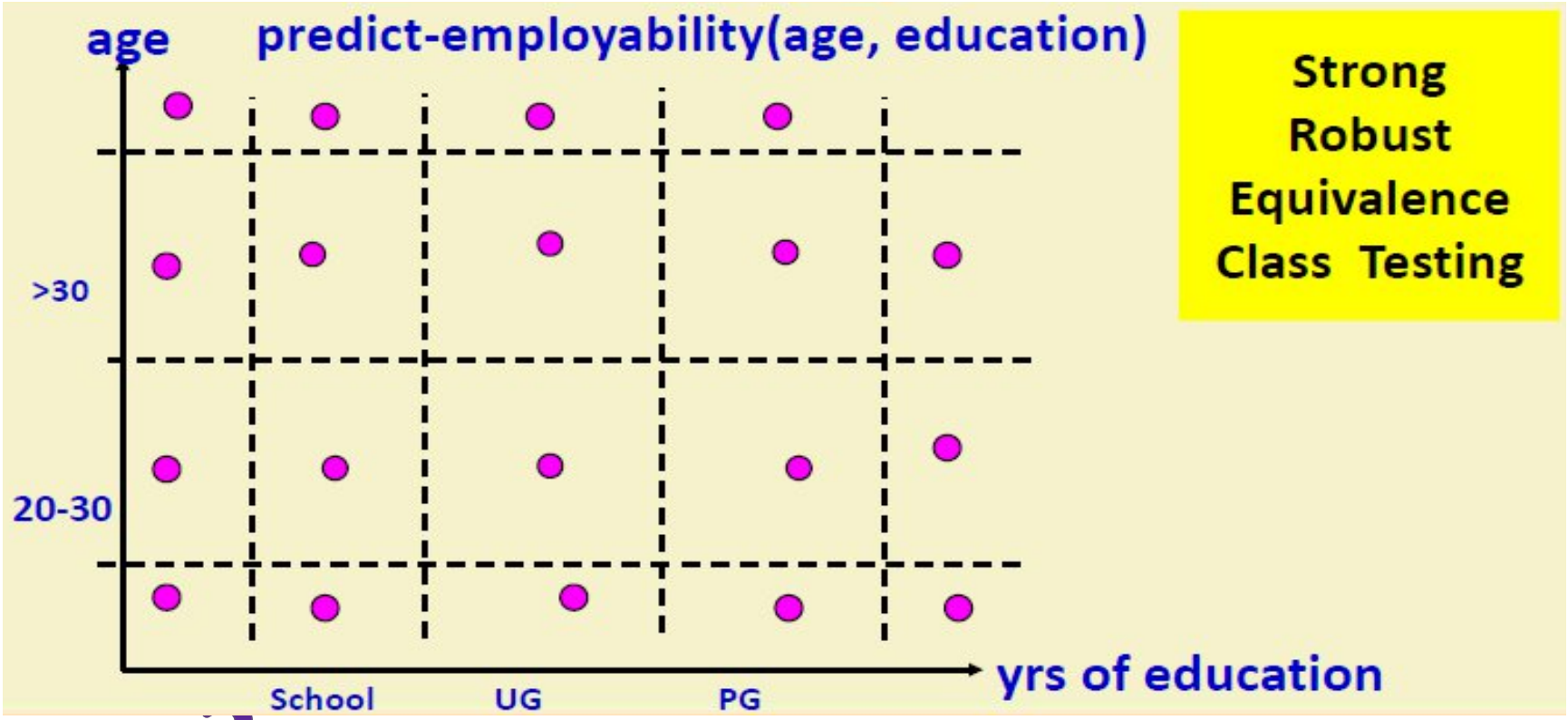

predict-employability(age, education)
age
>30
20-30
School
UG
PG
Yrs of ed.
Strong
Equivalence
Class Testing
?

- Once we identify Equivalence Classes for the two parameters
- Combine the representative values for these two parameters
- Strong:
  - **Every value of 1 Ec of one parameter is combined with every Value of EC of other parameter.**

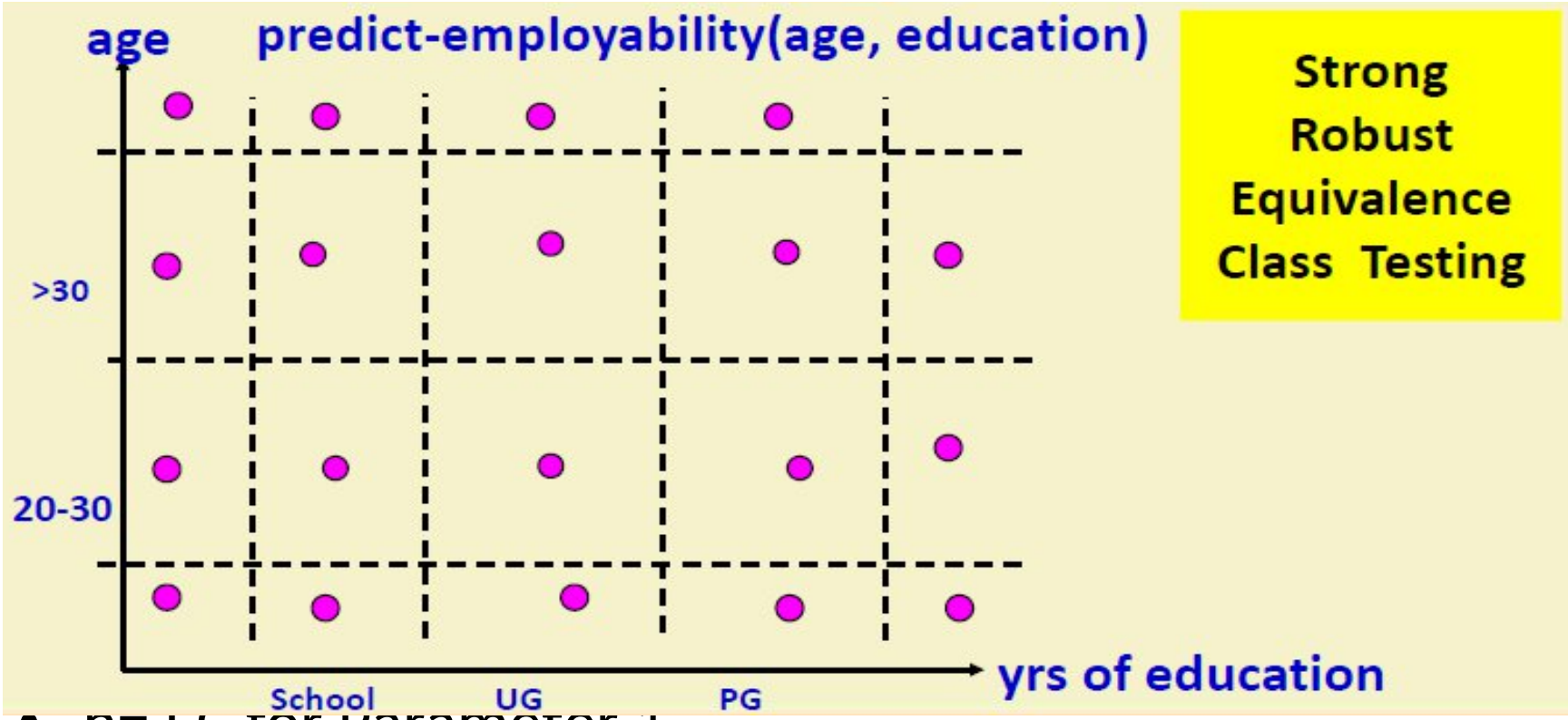


- m= EC for Parameter 2
- No of test cases=n x m

n= 3 for yrs of education
m=2 for age
No of test cases=3x2=6


age
predict-employability(age, education)
Strong
Robust
Equivalence
Class Testing
>30
20-30
School
UG
PG
yrs of education

?



- n= EC for Parameter 1
- m= EC for Parameter 2
- No of test cases=n x m (Invalid Test cases Included)

n= 5 for yrs of education
m=4 for age
No of test cases=5x4=20

EXAMPLE 1

Design Equivalence class test cases: **compute-interest(days)**
A bank pays different rates of interest on a deposit depending on the deposit period.

- 3% for deposit up to 15 days
- 4% for deposit over 15days and up to 180 days
- 6% for deposit over 180 days upto1 year
- 7% for deposit over 1 year but less than 3 years
- 8% for deposit 3 years and above

EXAMPLE 1

Only one parameter is there.
We examine the Output and each of them correspond to a scenario.
We consider **1 Equivalence Class corresponding for each case and then 1 Equivalence Class for Invalid Set of Inputs which is invalid date.**

- 3% for deposit up to 15 days=1 EC
- 4% for deposit over 15days and up to 180 days=1 EC
- 6% for deposit over 180 days upto1 year=1 EC
- 7% for deposit over 1 year but less than 3 years=1 EC
- 8% for deposit 3 years and above =1 EC
- Total=5EC + 1 Invalid EC=6EC

EXAMPLE 2

Design Equivalence class test cases:

- For deposits of less than or equal to Rs. 1 Lakh, rate of interest:
  - 6% for deposit upto1 year
  - 7% for deposit over 1 year but less than 3 years
  - 8% for deposit 3 years and above

- For deposits of more than Rs. 1 Lakh, rate of interest:
  - 7% for deposit upto1 year
  - 8% for deposit over 1 year but less than 3 years
  - 9% for deposit 3 years and above

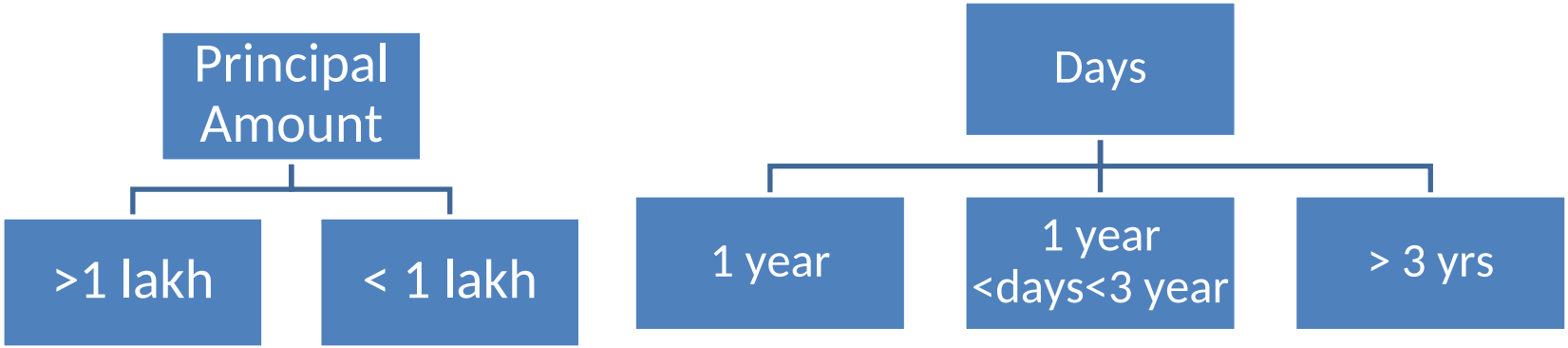
EXAMPLE 2

- Takes 2 parameter
  - Principal amount
  - Days

  - Use Either Weak Equivalence Class Testing or Strong Equivalence Class testing

EXAMPLE 2

- Takes 2 parameter
  - Principal amount
  - Rate



  - For Strong Robust Equivalence Class testing, we can use invalid class also.
  - **EC= Invalid Principal Amount**
  - **EC=Invalid Days**

Boundary Value Analysis

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

- BVA has been developed as a testing technique.
- Complements equivalence partitioning

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

- Boundary value analysis leads to a selection of test cases that
  - **exercise bounding values.**
  - **at the "edges" of the class.**

Guidelines for BVA

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

2) If an input condition specifies **a number of values**, test cases should be developed that exercise
   - ❑ **the minimum and maximum numbers**.
   - ❑ Values just **above and below minimum and maximum.**

FOR INPUT

Guidelines for BVA

3) Apply guidelines 1 and 2 to output conditions.

For example, assume that a temperature versus pressure table is required as output from an engineering analysis program.

- ❑ Test cases should be designed to create an output report that produces the
  - ❑ maximum and
  - ❑ minimum allowable number of table entries.

FOR OUTPUT

3) If internal program data structures have prescribed boundaries **(e.g., a table has a defined limit of 100 entries),** be certain to design a test case to **exercise the data structure at its boundary.**

Boundary Value Analysis: Guidelines

- If an input is a range, bounded by values a and b:
  - **Test cases should be designed with value a and b, just above and below a and b.**

- **Example 1:** Integer D with input range [-3, 10],
- test values:-3, 10, 9, -2, 0

- **Example 2:** Input in the range: [3,102]
- test values: 3, 102, 101, 80, 5

Boundary Value Testing: Example 3

- Process employment applications based on a person's age.

0-16	Do not hire
16-18	May hire on part time basis
18-55	May hire full time
55-99	Do not hire

- Notice the problem at the boundaries. Age "16" is included in two different equivalence classes (as are 18 and 55).

Boundary Value Testing: Code Example 3

If (applicantAge>= 0 && applicantAge**<=16**) hireStatus="NO";

If (applicantAge**>= 16** && applicantAge<=18) hireStatus="PART";

If (applicantAge>= 18 && applicantAge<=55) hireStatus="FULL";

If (applicantAge>= 55 && applicantAge<=99) hireStatus="NO";

Boundary Value Testing: Example 3 (cont)

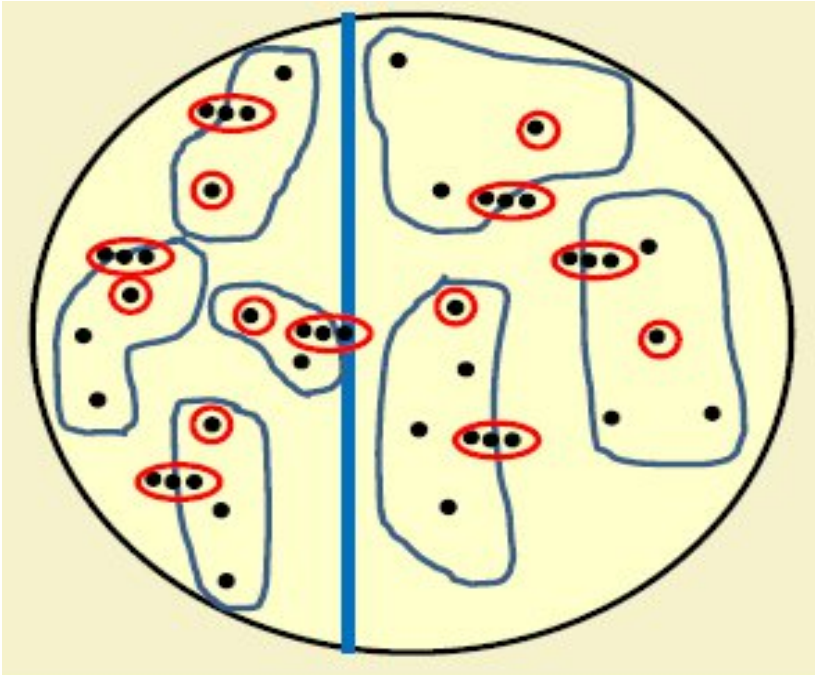
- Corrected boundaries:

0–15	Don't hire
16–17	Can hire on a part-time basis only
18–54	Can hire as full-time employees
55–99	Don't hire

- What about ages -3 and 101?
- The requirements do not specify how these values should be treated.

Boundary Value Analysis

- Create test cases to **test boundaries of equivalence classes**

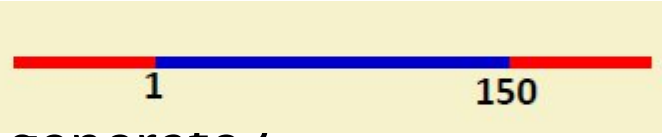


Example 4

- Consider a program that reads the “age” of employees and computes the average age.

input (ages) → Program → output: average age

Assume valid age is 1 to 150

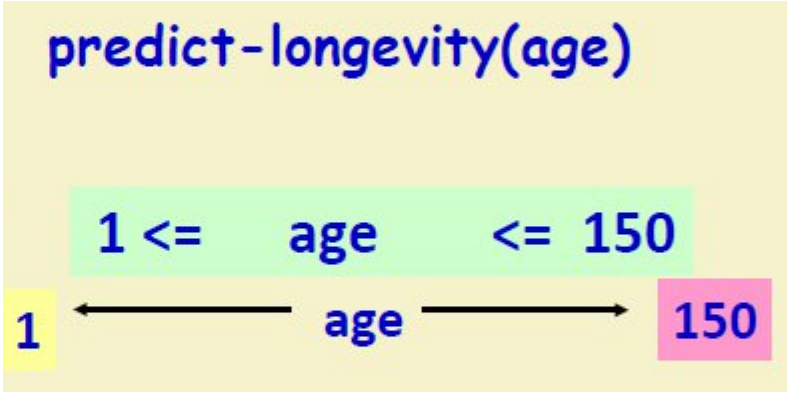


- How would you test this?
  - How many test cases would you generate?
  - What type of test data would you input to test this program?

Example 4

The “basic” boundary value testing would include 5 test cases:

1. -at minimum boundary
2. -immediately above minimum
3. -between minimum and maximum (nominal)
4. -immediately below maximum
5. -at maximum boundary



Example 4

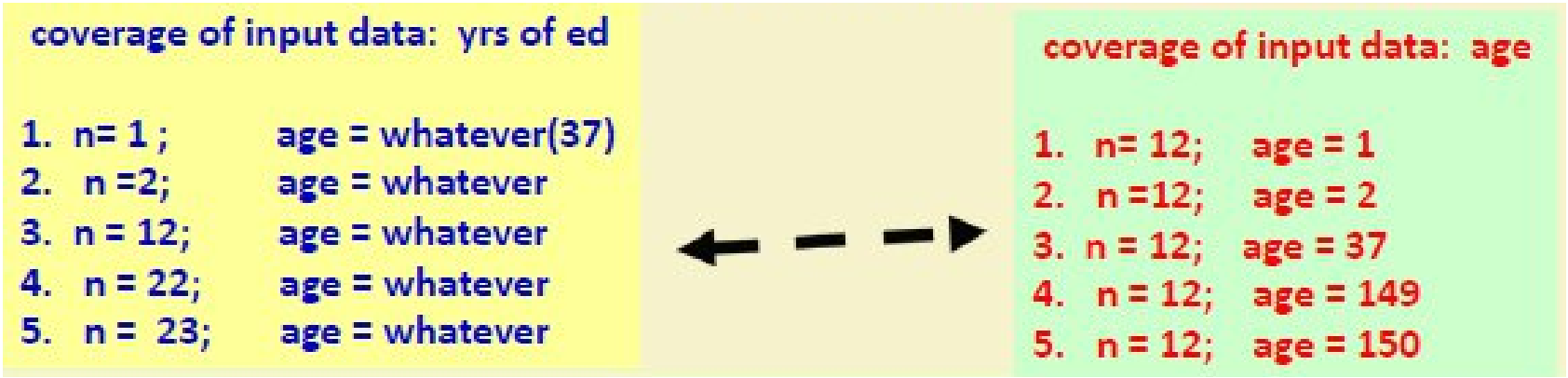
- How many test cases for the example ?answer : **5**

- Test input values? :
  1 at the minimum
  2 at one above minimum
  45 at middle
  149 at one below maximum
  150 at maximum

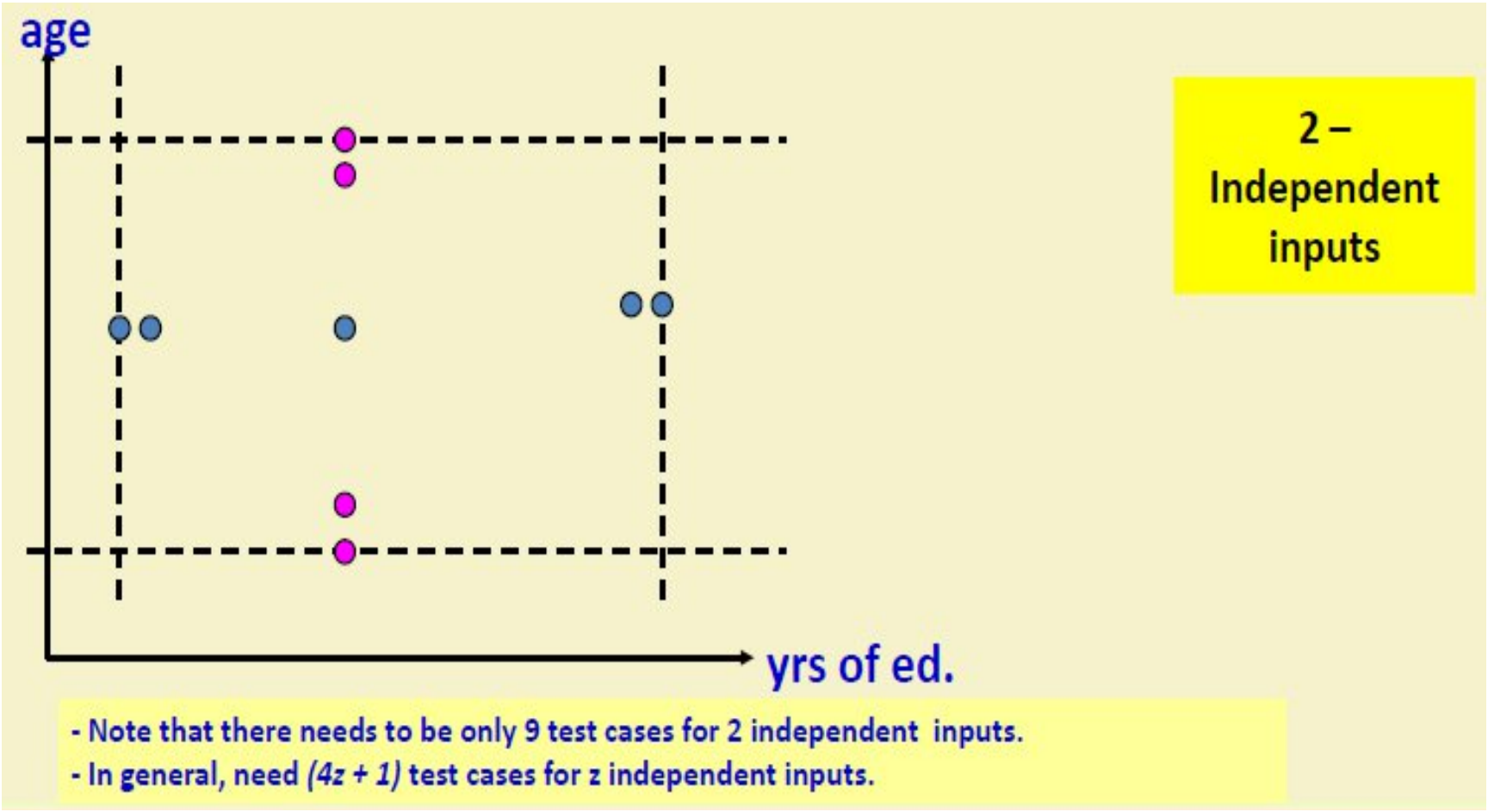
```
predict-longevity(age)
```

Multiple Parameters: Independent distinct Data

- Suppose there are 2 “distinct” inputs that are assumed to be independent of each other.
- Input field 1: years of education (say 1 to 23)
- Input field 2: age (1 to 150)
- If they are independent of each other, then we can start with 5 + 5 = 10 sets.

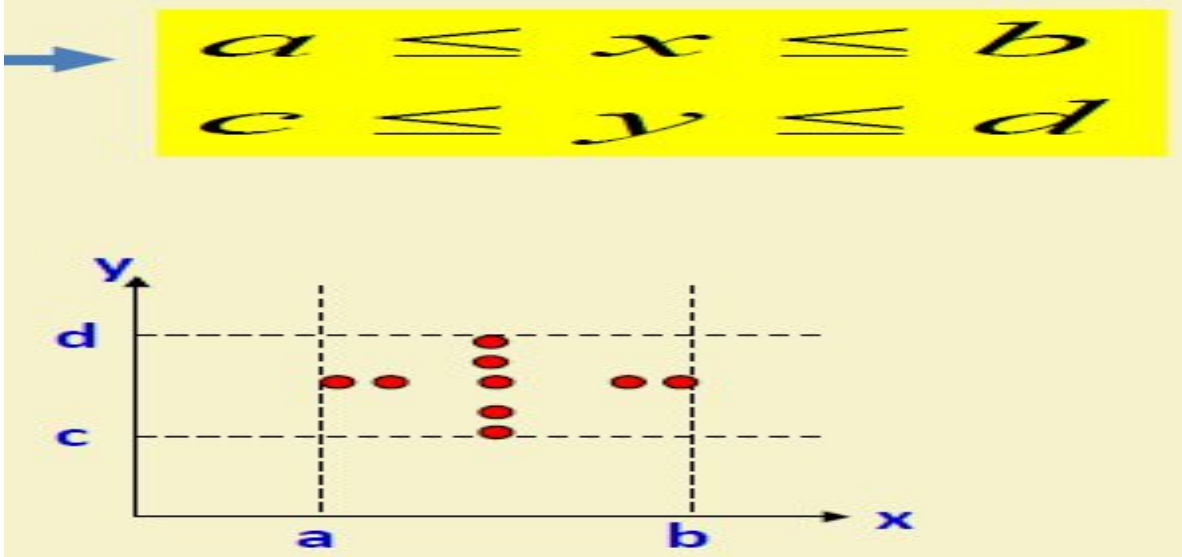


Multiple Parameters: Independent distinct Data



Boundary Value Test

- Given **f(x,y)** with constraints
- Boundary Value analysis focuses on the boundary of the input space to identify test cases.
- Defined as input variable value **at min, just above min, a nominal value, just below max, and at max.**



Weak Testing: Single Fault Assumption

- **Premise:** “Failures rarely occur as the result of the simultaneous occurrence of two (or more) faults”
- Under this: **Hold the values of all but one variable at their nominal values, and let that one variable assume its extreme values.**

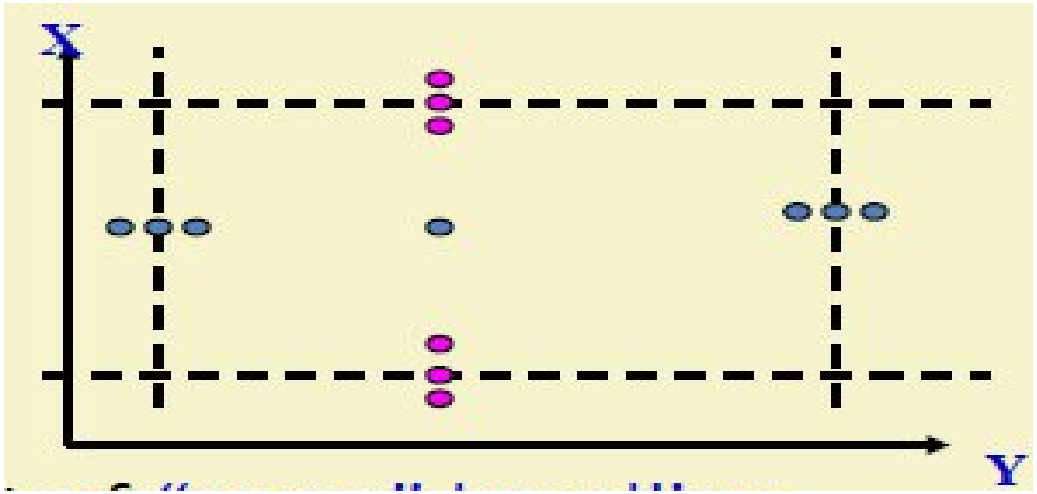
Robustness Boundary Value testing

- This is just an extension of the Boundary Values **to include invalid values:**
  - **Less than minimum**
  - **Greater than maximum**

- There are 7 test cases for each input
- The testing of robustness is really a test of "error" handling.
  1) *Did we anticipate the error situations?*
  2) *Do we issue informative error messages?*
  3) *Do we support some kind of recovery from the error?*

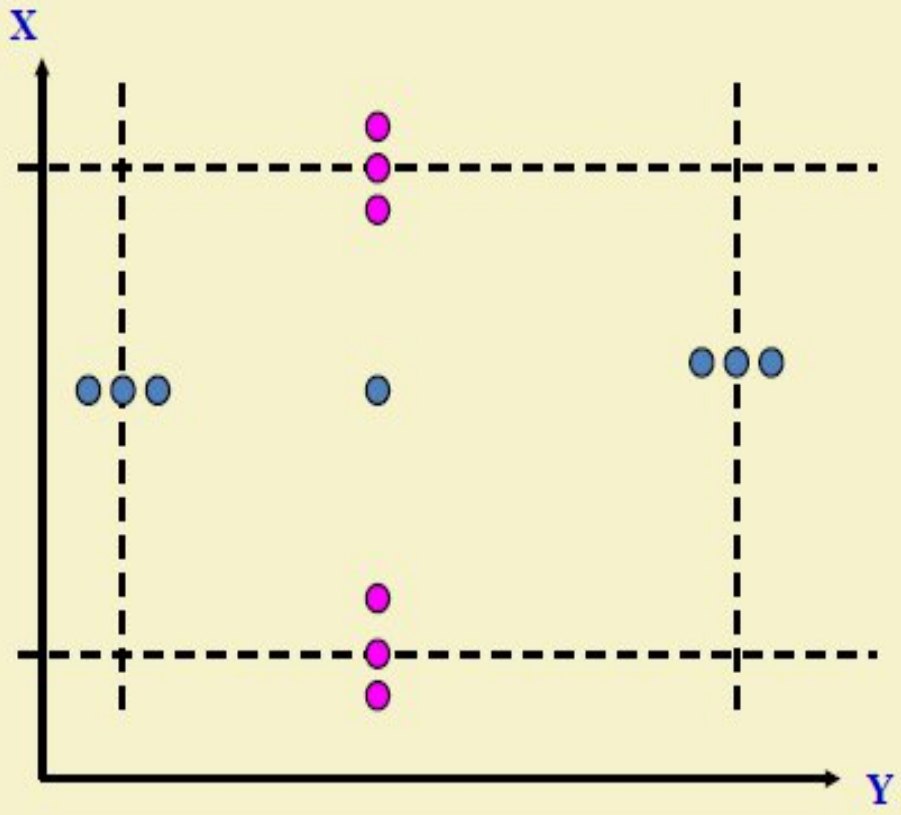
Robustness Boundary Value testing

- This is just an extension of the Boundary Values **to include invalid values:**
  - **Less than minimum**
  - **Greater than maximum**



- There are 7 test cases for
- The testing of robustness is really a test of "error" handling.
  1) *Did we anticipate the error situations?*
  2) *Do we issue informative error messages?*
  3) *Do we support some kind of recovery from the error?*

2 – independent inputs for robustness test



- Note that there needs to be only 13 test cases for 2 independent variables or inputs.
- In general, there will be *(6n+1)* test cases for n independent inputs.

Some Limitations of Boundary Value Testing

- How to handle a set of values?
- How about set of Boolean variables?
  - True
  - False
- May be radio buttons
- What about a non-numerical variable whose values are text?

Graph-Based Testing Methods

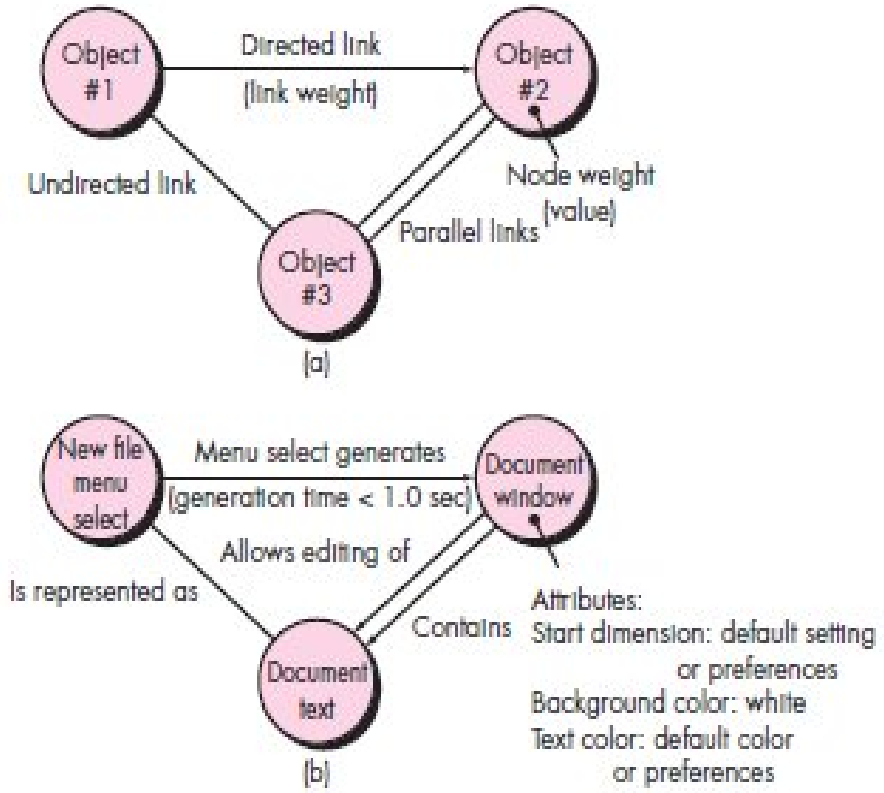
- Define a series of tests that verify "all objects have the expected relationship to one another"
- Creating
  - a graph objects and
  - their relationships and
  - devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

Creating a *graph*

- Collection of ***nodes*** that represent objects,
- ***Links*** that represent the relationships between objects, *node*
- ***Weights*** that describe the properties of a node (e.g., a specific data value or state behavior)
- ***Link weights*** that describe some characteristic of a link.
- ***A directed link*** (represented by an arrow) indicates that a relationship moves in only one direction.
- ***A bidirectional link***, also called a *symmetric link*, implies that the relationship applies in both directions.
- ***Parallel links*** are used when a number of different relationships are established between graph nodes.

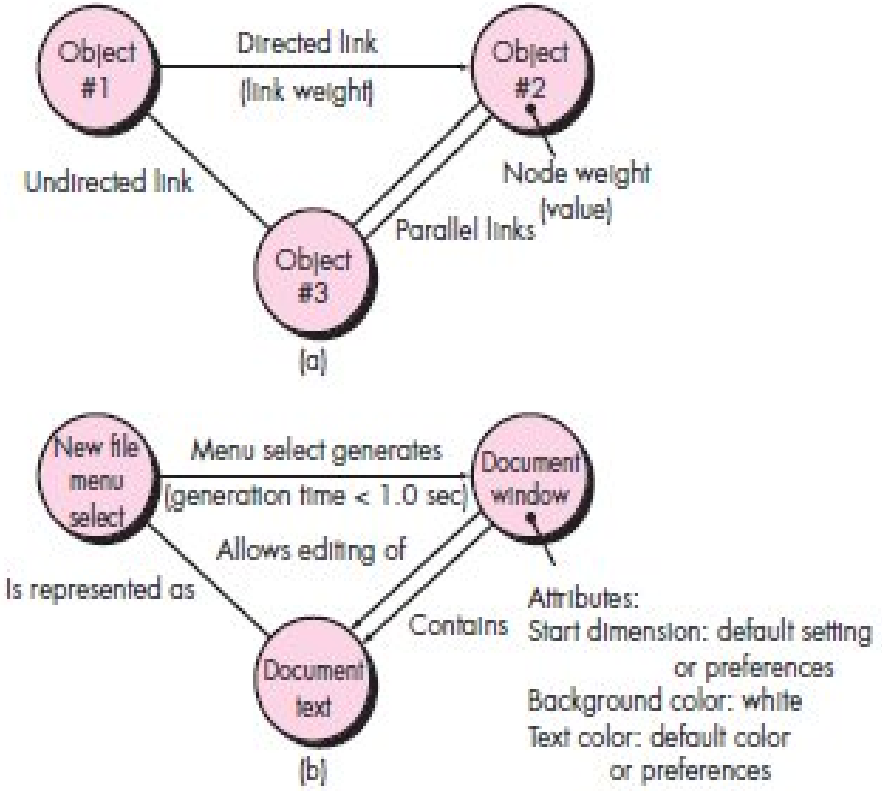
Graph-Based Testing Methods

- A portion of a graph for a word-processing application in Figure where
  - *Object #1*=**newFile** (menu selection)
  - *Object#2*=**documentWindow**
  - *Object #3*=**documentText**
- A menu select on **newFile** generates a document window.



Graph-Based Testing Methods

- The **node weight** of **documentWindow** provides **a list of the window attributes** that are to be expected when the window is generated.
- The **link weight** indicates that the **window must be generated in less than 1.0 second.**
- An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,**
- Parallel links indicate relationships between **documentWindow** and **documentText.**



Graph-Based Testing Methods

- Derive test cases by **traversing the graph and covering each of the relationships.**
  - These test cases are designed in an attempt to **find errors in any of the relationships.**

Graph-Based Testing Methods

Beizer describes a number of behavioral testing methods that can make use of graphs:

- **Transaction flow modeling-**The **nodes represent steps in some transaction** and the links represent the **logical connection between steps**
- **Finite state modeling-**The **nodes represent different user-observable states** of the software and the links represent the **transitions that occur to move from state to state**
- **Data flow modeling-**The **nodes are data objects**, and the links are the **transformations that occur to translate one data object into another.**
- **Timing modeling-**The **nodes are program objects**, and the links are the **sequential connections between those objects**. Link weights are used to **specify the required execution times as the program executes.**

Black-box testing

It includes:

1) **EQUIVALENCE PARTITIONING**
2) **BOUNDARY VALUE ANALYSIS**
3) **GRAPH-BASED TESTING METHODS**

Test-Case Design

Objective: To assist the software team in developing a complete set of test cases for both black-box and white-box testing.

Mechanics: These tools fall into two broad categories:

- **Static testing tools**
- **Dynamic testing tools.**
- Three different types of static testing tools are used in the industry:
  - **code-based testing tools,**
  - **specialized testing languages,**
  - **requirements-based testing tools.**

Test-Case Design

- **Static Testing**
  - **Codebased testing** tools accept source **code as input** and perform a number of analyses that result in the **generation of test cases.**
  - **Specialized testing** languages (e.g., ATLAS) enable a software engineer to **write detailed test** specifications that describe each test case and the logistics for its execution.
  - **Requirements-based testing** tools isolate specific user requirements and **suggest test cases (or classes of tests) that will exercise the requirements.**
- **Dynamic testing tools interact with an executing program**, checking **path coverage, testing assertions** about the value of specific variables, and otherwise instrumenting the execution flow of the program.

The scenario

- An application has served the business needs of a company for 10 or 15 years.
- During that time it has been corrected, adapted, and enhanced many times.
- People approached this work with the best intentions, but good software engineering practices were always shunted to the side (due to the press of other matters).
- Now the application is unstable.
- It still works, but every time a change is attempted, unexpected and serious side effects occur.
- Yet the application must continue to evolve. What to do?
- Unmaintainable software is not a new problem.

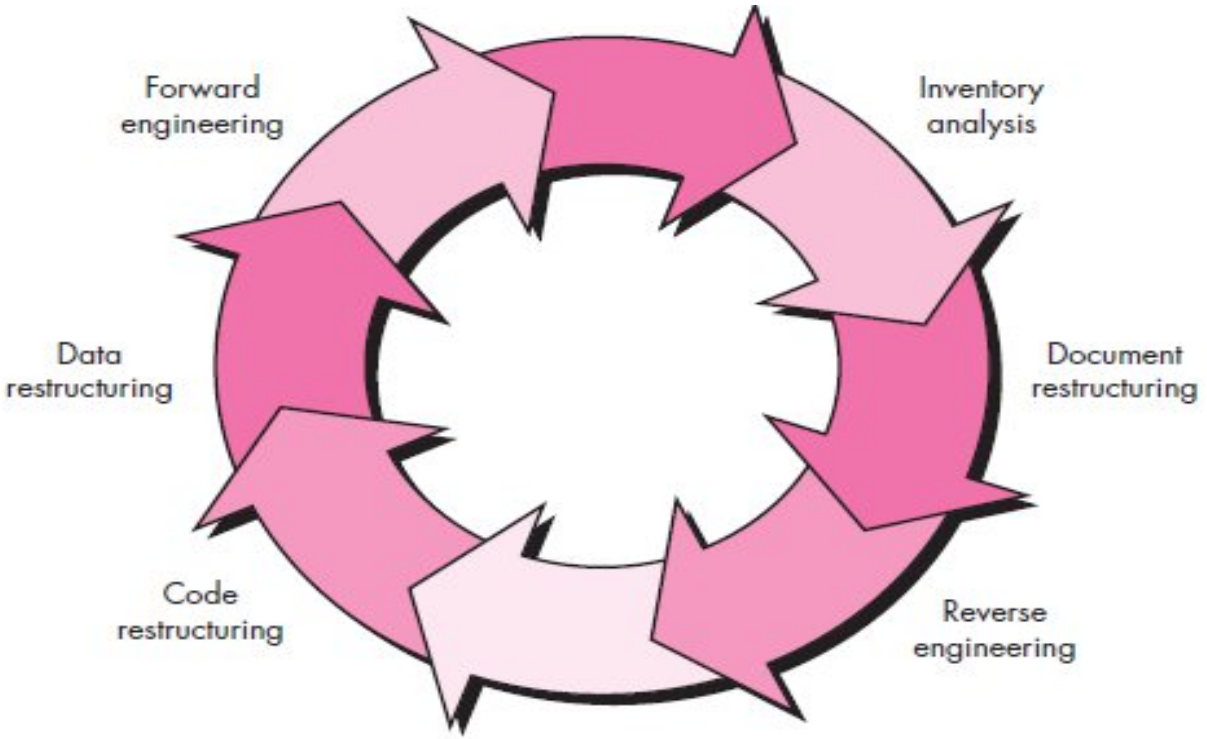
Soln?

Software Reengineering

Software Reengineering Activities

- The reengineering paradigm shown in Figure is a cyclical model.
- Each of the activities presented as a part of the paradigm may be revisited.

A software reengineering process model



Software Reengineering Activities

Inventory analysis.

- Every software organization should have an inventory of all applications.
- 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.**
- **By sorting this information** according to:
  - **business criticality,**
  - **longevity,**
  - **current maintainability and**
  - **supportability**,

  and other locally important **criteria, candidates for reengineering appear.**
- **Resources can then be allocated to candidate applications for reengineering work.**

Software Reengineering Activities

Document restructuring.
Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options?

1. *Creating documentation is far too time consuming.*

If the system works, you may choose to live with what you have.

?

Software Reengineering Activities:Document restructuring

It is not possible to re-create documentation for hundreds of computer programs.

If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!

?

Software Reengineering Activities:Document restructuring

2. ***Documentation must be updated, but your organization has limited resources.***
You'll use a "document when touched" approach.
It may not be necessary to fully redocument an application. Rather, those portions of the system that are currently undergoing change are fully documented.
Over time, a collection of useful and relevant documentation will evolve.
3. ***The system is business critical and must be fully redocumented.***
Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Software Reengineering Activities:Reverse engineering

- The term *reverse engineering* has its origins in the hardware world.
  - A company disassembles a competitive hardware product to understand its competitor's design and manufacturing "secrets."
  - These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained.
  - But these documents are proprietary and unavailable to the company doing the reverse engineering.
- In essence, successful reverse engineering derives one or more design and manufacturing specifications for a product by examining actual specimens of the product.

Software Reengineering Activities:Reverse engineering

- Reverse engineering for software is quite similar.
- The program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier).
- The "secrets" to be understood are obscure because no specification was ever developed.
- Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code.
- Reverse engineering is a process of *design recovery*.
- Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Software Reengineering Activities:Code restructuring

- The most common type of reengineering is *code restructuring*.
  - Some legacy systems have a relatively solid program architecture,
  - but individual modules were coded in a way that makes them difficult to understand, test, and maintain.
- In such cases, the code within the suspect modules can be restructured.

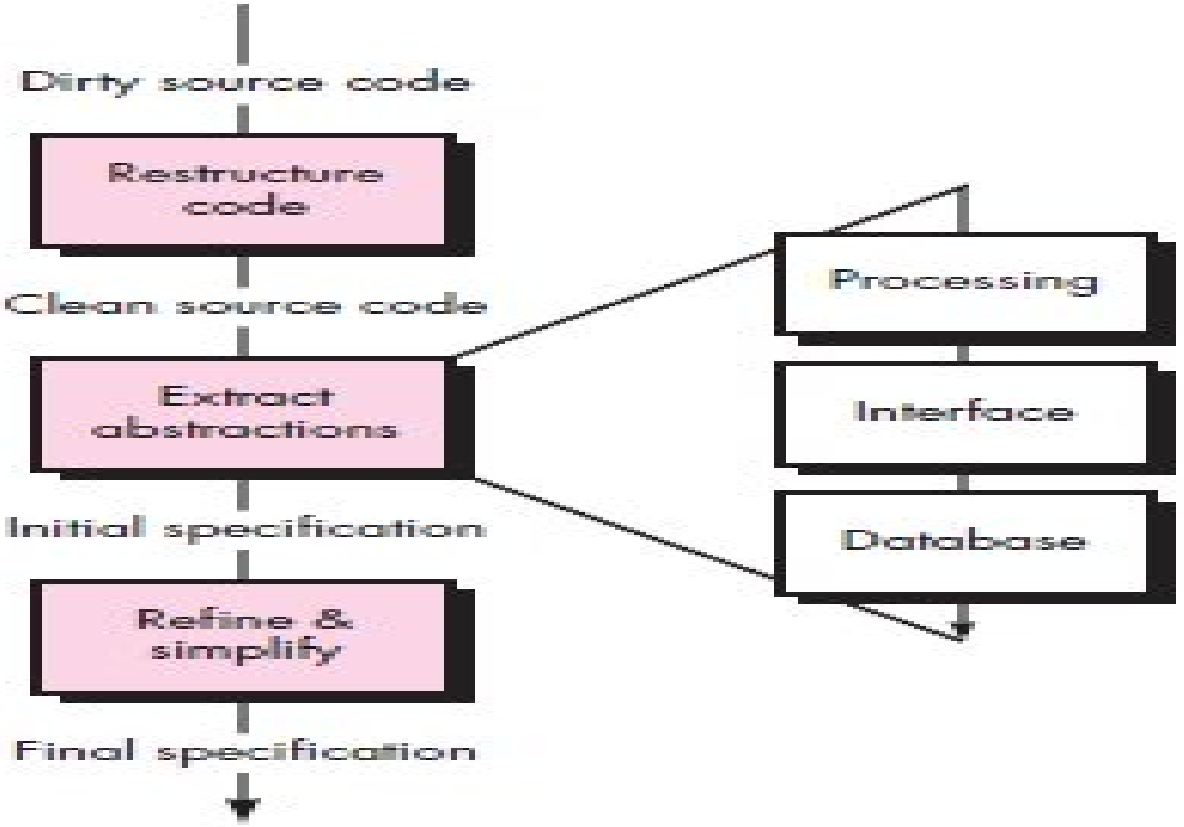
Software Reengineering Activities:Data restructuring

- A program with weak data architecture will be difficult to adapt and enhance.
  - For many applications, **information architecture has more to do with the long-term viability** of a program than the source code itself.
- Data restructuring begins with a reverse engineering activity.
  - 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 Activities:Forward engineering

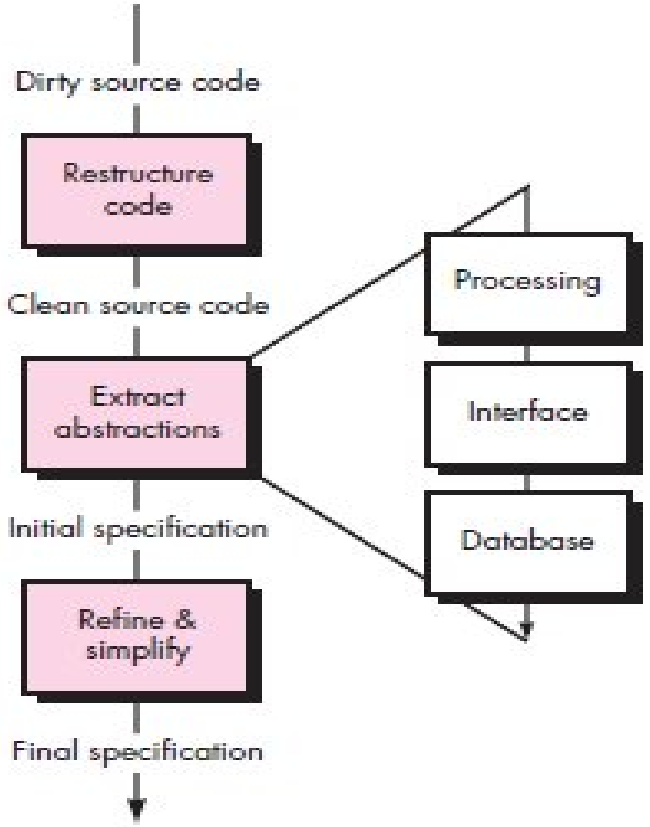
- In an ideal world, applications would be rebuilt using an automated “reengineering engine.” The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality.
- Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

Reverse engineering



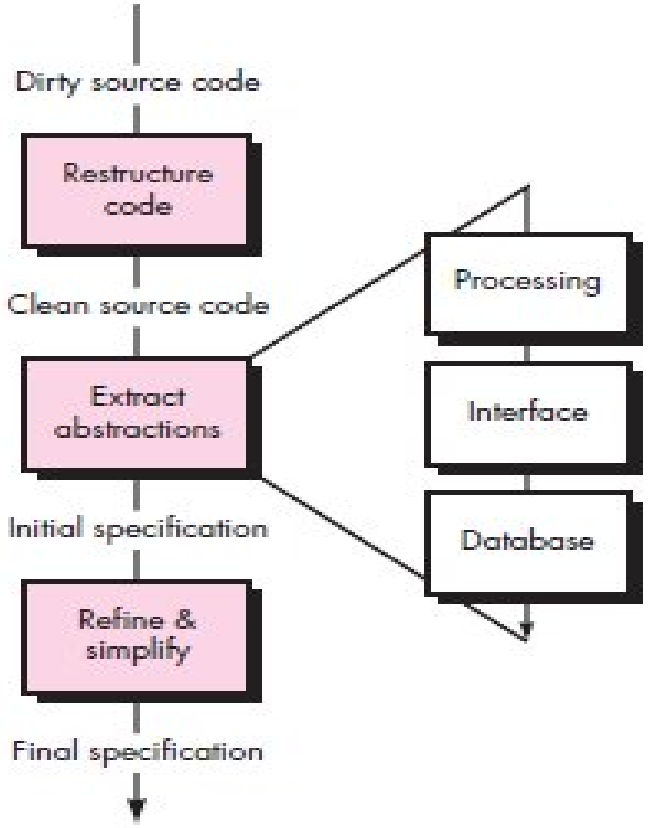
Reverse engineering

- Before reverse engineering activities can commence:
  - **unstructured ("dirty") source code is restructured** so that it contains only the structured programming constructs.
  - This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.



Reverse engineering

- The core of reverse engineering is an activity called *extract abstractions*.
  - Evaluate the old program and from the (often undocumented) source code,
  - Develop a meaningful specification of the processing that is performed,
  - The user interface that is applied, and
  - The program data structures or database that is used.



Reverse Engineering to Understand Data

Reverse engineering of data occurs at different levels of abstraction and is often the first reengineering task.

- **At the program level,**
  - internal program data structures must often be reverse engineered as part of an overall reengineering effort.
- **At the system level,**
  - global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems).

Reverse Engineering to Understand Data

Internal data structures.

- Reverse engineering techniques for internal program data focus on:
  - the **definition of classes of objects.** This is accomplished by **examining the program code with the intent of grouping related program variables.** In many cases, the data organization within the code identifies abstract data types.

Reverse Engineering to Understand Data

Database structure.

Reengineering one database schema into another requires an understanding of existing objects and their relationships.

Reverse Engineering to Understand Processing

- Reverse engineering to understand processing begins with an **attempt to understand and then extract procedural abstractions** represented by the source code.
- To understand procedural abstractions, the code is analyzed at varying levels of abstraction:
  - system, program, component, pattern, and statement.

Reverse Engineering User Interfaces

- Sophisticated GUIs have become popular for computer-based products. Therefore, the **redevelopment of user interfaces has become one of the most common types of reengineering** activity.
- **But before a user interface can be rebuilt, reverse engineering should occur.**

- To fully understand an existing user interface,
  - **the structure and behavior of the interface must be specified.**

Reverse Engineering User Interfaces

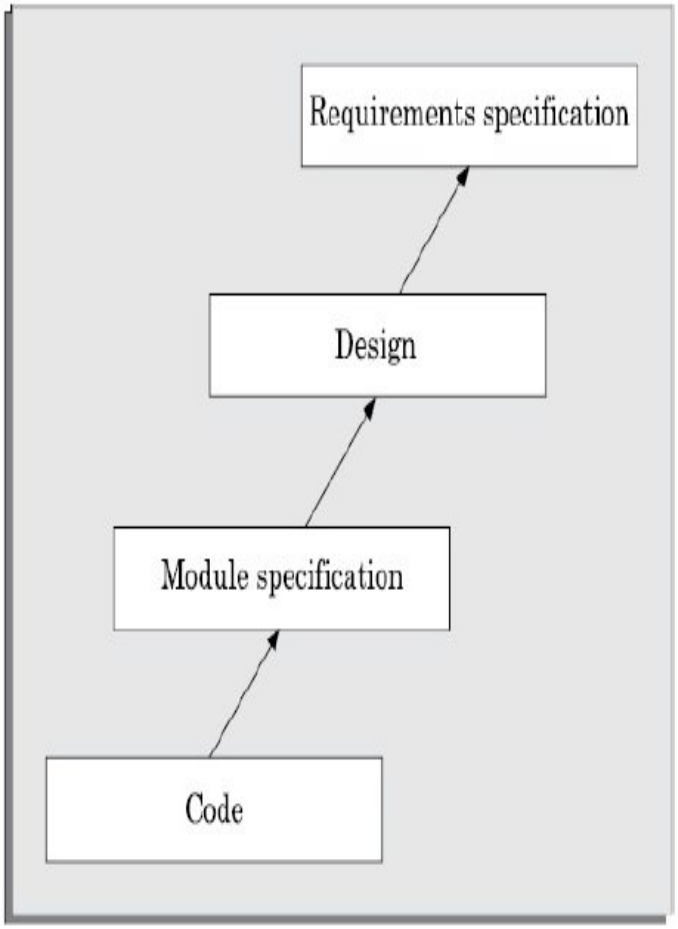
3 basic questions that must be answered as reverse engineering of the UI commences:

- **What are the basic actions** (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of **the behavioral response of the system to these actions**?
- What is meant by **a "replacement," or more precisely, what concept of equivalence of interfaces** is relevant here?

(Recap)Software reverse engineering

- Software reverse engineering is the process of
  - **recovering the design and the requirements specification of a product from an analysis of its code.**

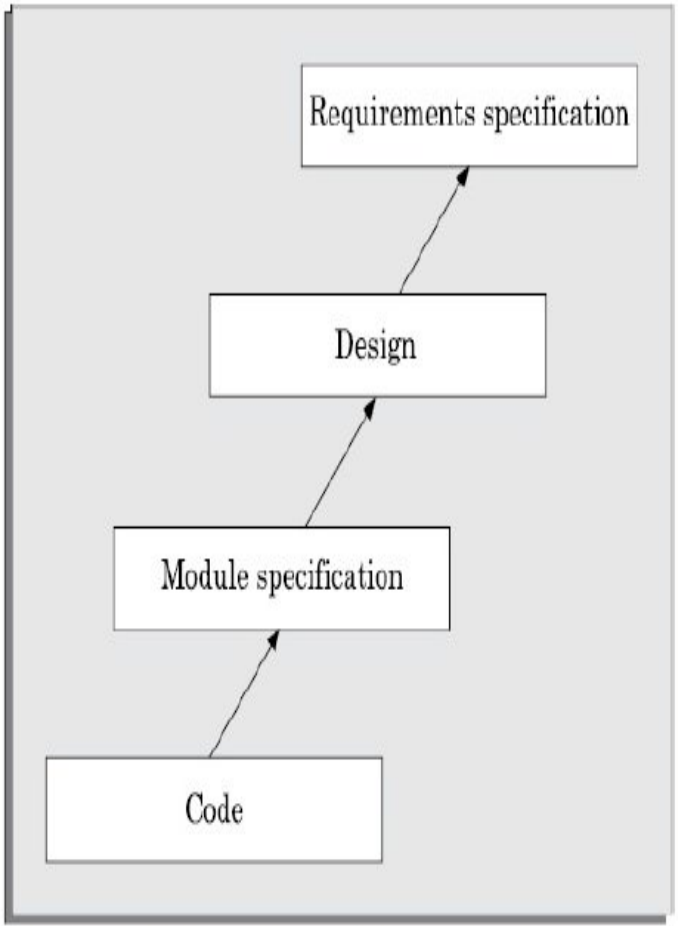
- The purpose of reverse engineering is
  - **to facilitate maintenance work**
  - **to produce the necessary documents for a legacy system.**



(Recap)Software reverse engineering

- The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its
  - readability,
  - structure, and
  - understandability,
  - without changing any of its functionalities.

- After the cosmetic changes have been carried out ,
  - the process of **extracting the code, design, and the requirements specification can begin.**

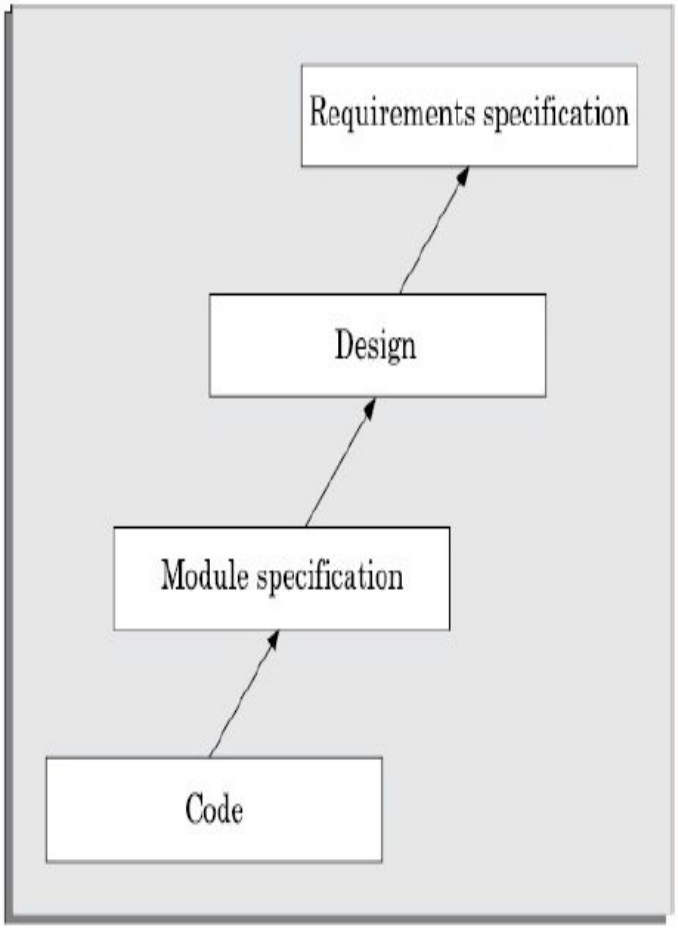


(Recap)Software reverse engineering

- In order to **extract the design**, a full understanding of the code is needed.
- Some automatic tools can be used to derive the data flow and control flow diagram from the code.

- The **structure chart should also be extracted.**

- **The SRS document can be written once the full code has been thoroughly understood and the design extracted.**



Software Maintenance

- The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself.

- Ratio of relative effort of developing a software product and the total effort spent on its maintenance is roughly 40:60.

Types of Software Maintenance

There are three types of software maintenance, which are described as follows:

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Perfective: A software product needs maintenance to

- support the **new features** that users want it to support,
- to **change different functionalities** of the system according to customer demands,
- or to **enhance the performance** of the system.

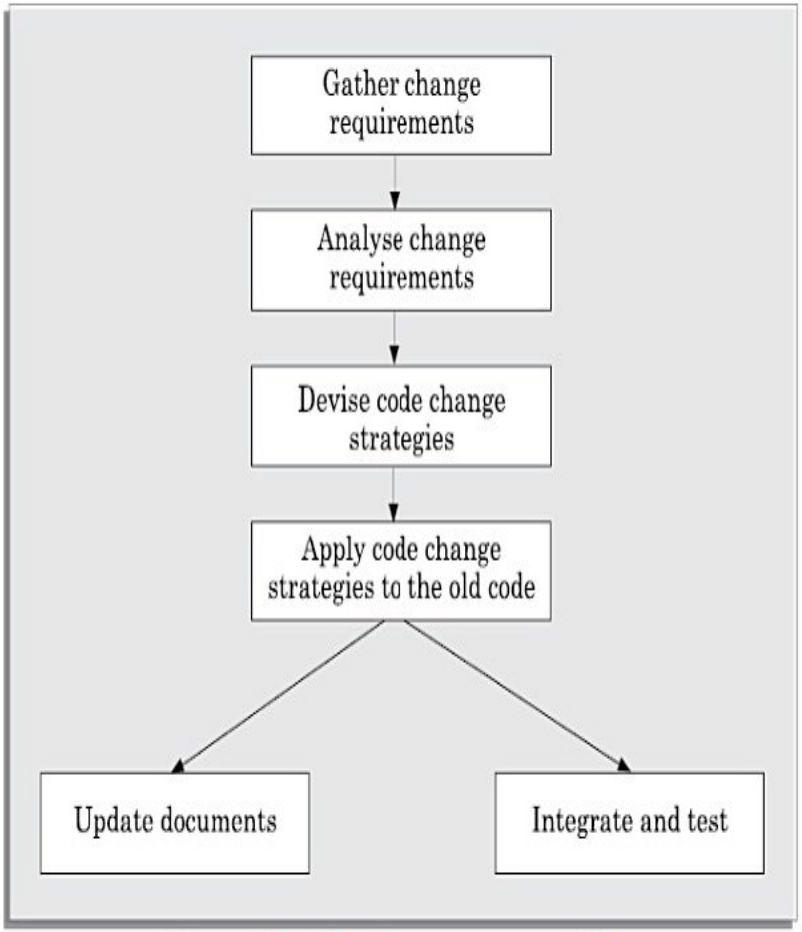
Types of Software Maintenance

Adaptive: A software product might need maintenance when the customers need the product to

- run on new platforms,
- on new operating systems, or
- when they need the product to interface with new hardware or software.

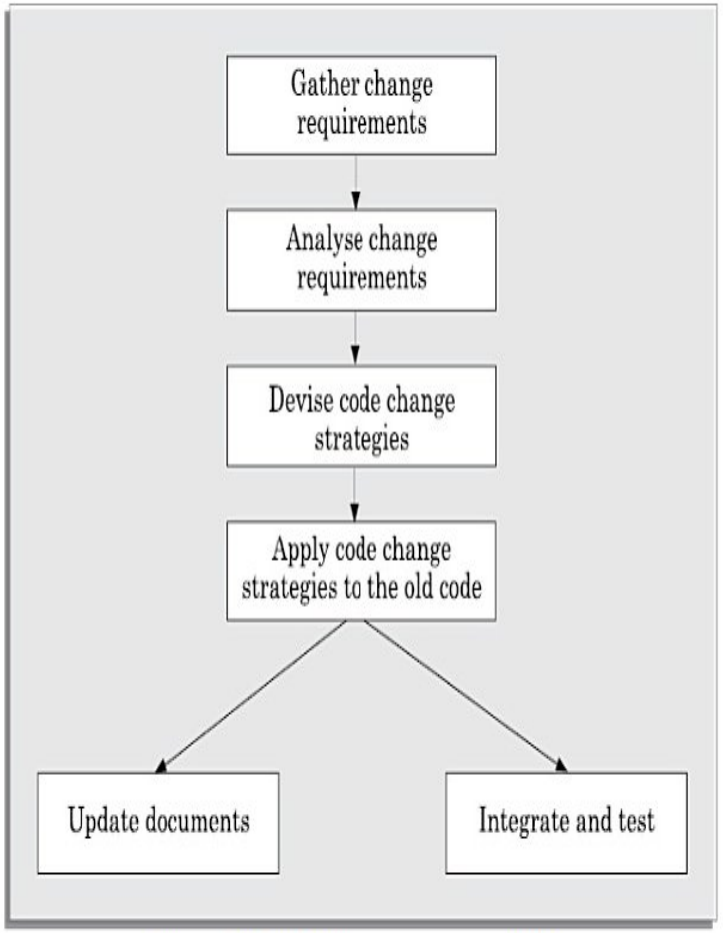
Software Maintenance Process Model: First model

- Preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later.
- Project starts by gathering the requirements for changes.
- The requirements are next analysed to formulate the strategies to be adopted for code change.



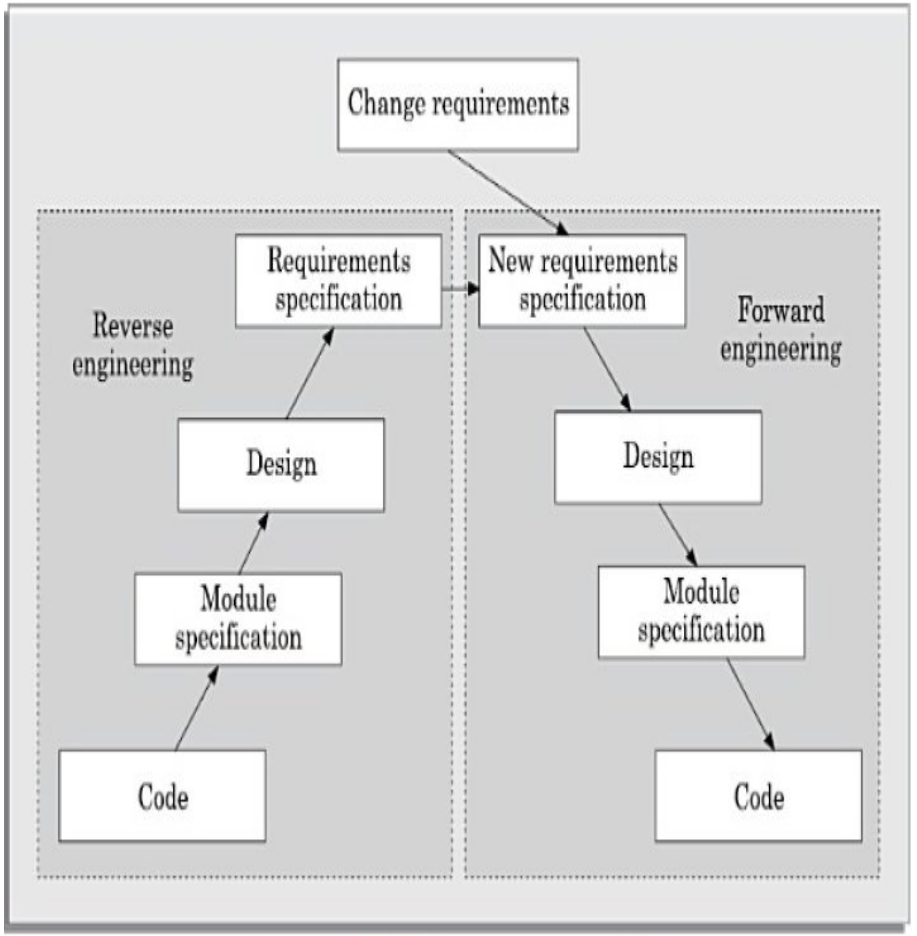
First model

- At this stage, the **association of at least a few members of the original development team** goes a long way in
  - **reducing the cycle time,**
  - especially for projects involving unstructured and inadequately documented code.
- The **availability of a working old system to the maintenance engineers** at the maintenance site greatly facilitates the task of the maintenance team as
  - they **get a good insight into the working of the old system and**
  - also can compare the working of their modified system with the old system.



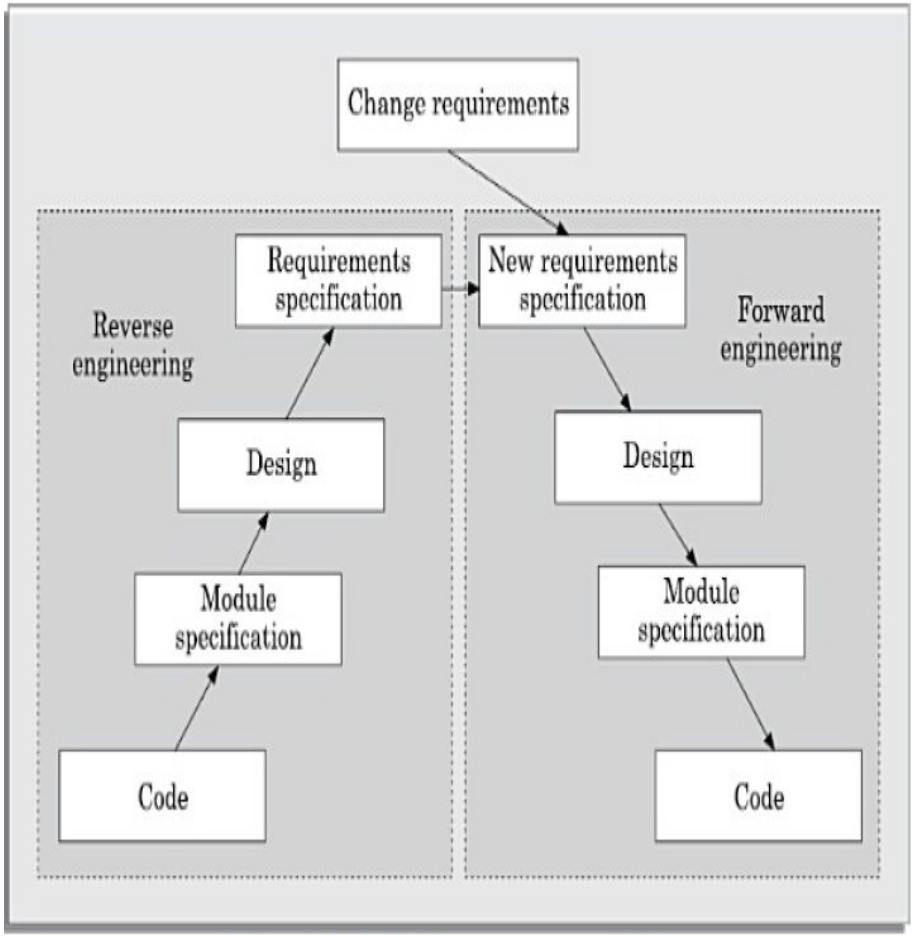
Software Maintenance Process Model: Second model

- The second model is preferred for projects where the amount of rework required is significant.
- This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering.
- This process model is depicted in Figure.



Second model

- During the reverse engineering, the old code is analysed (abstracted) to **extract the module specifications.**
- **The module specifications** are then analysed to produce the design.
- **The design is analysed (abstracted)** to produce the original requirements specification.
- The change requests are then applied to this requirements specification to arrive at the new requirements specification.



Second model

- At this point **a forward engineering is carried out to produce the new code.**
  - At the design, module specification, and coding a substantial reuse is made from the reverse engineered products.
- An important advantage of this approach is that it produces a
  - **more structured design compared to what the original product** had,
  - produces good documentation, and
  - very often results in increased efficiency.