# Software Engineering

## Lecture 7

## (Issue/Bug Tracking)

# Issue Tracking

# Remember

**Software engineering in a nutshell**

Software engineering is concerned with the efficient and timely delivery of software that meets stated requirements.

# Software Bugs

- *"A bug is an unwanted and unintended property of any piece of software, especially one that causes it to malfunction or fail."*

- It is an unwanted feature and we want it out.

- A common bug in most software is they crash for no apparent reason.

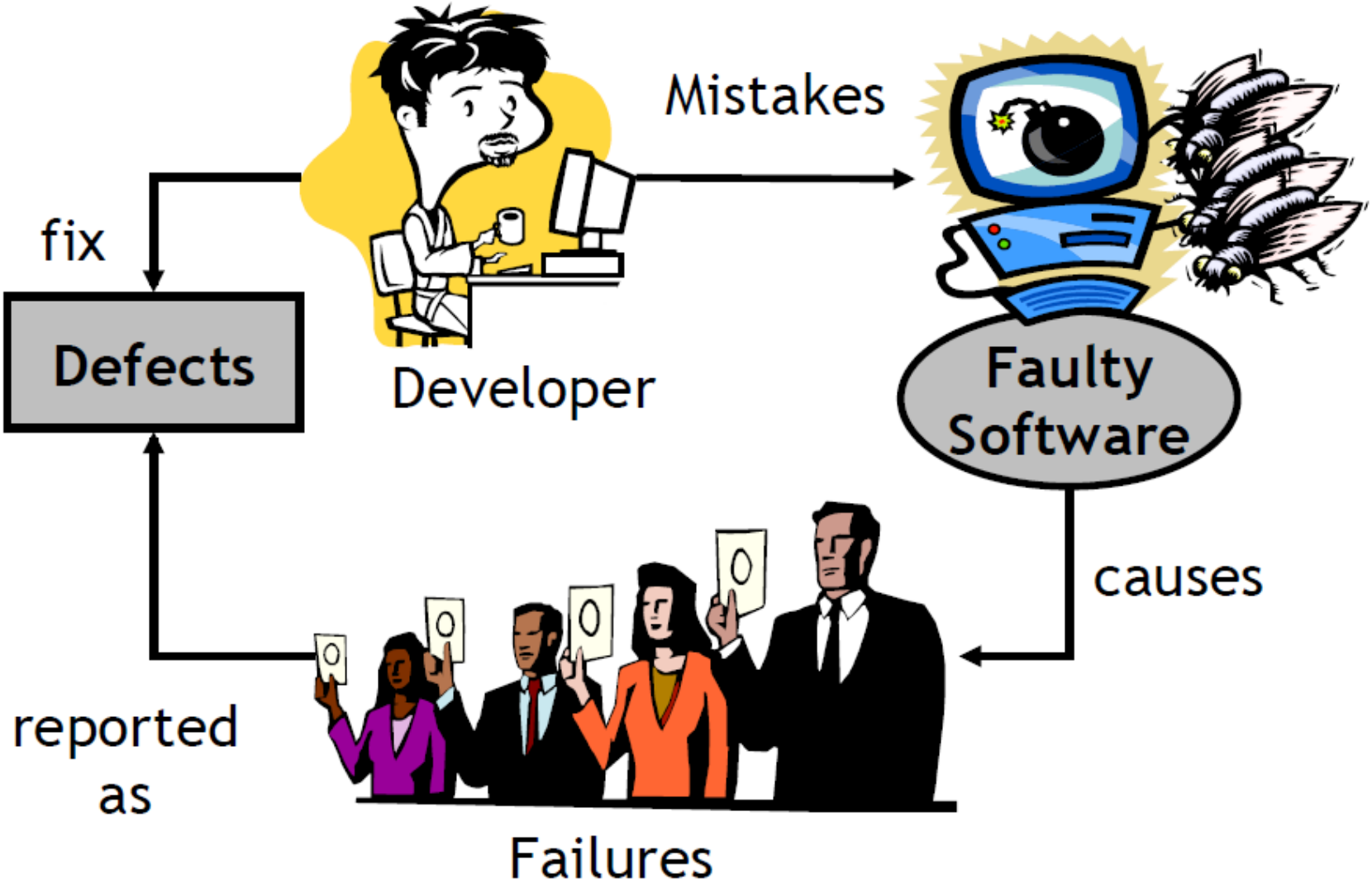
# Mistake, Fault & Failure …

- Developer makes a **mistake**
  - E.g. “>” instead of “>=“
- This generally results in code with some omissions or **faults**.
- These faults manifest as a **failure** when the program is in production or test.

# Life Cycle of a Bug …

- **Failure**s are reported as **defect**s – Developers trace and identify cause of defect – the original fault.
- Defects are removed – hopefully with no side-effects.
- Bugs are exterminated - but new bugs are either found or new errors are made introducing new bugs.

# Life Cycle of a Bug



# Terminology (IEEE Software Quality Management standard)

- Mistake
  - Human mistake results in manifestation of a fault during the execution of the system
- Fault or Defect
  - An incorrect statement, step, process, or data definition in a computer program. This is generally caused by a design or coding mistake. A single fault can cause multiple failures
- Failure
  - A manifest observable violation against specification or client expectations by the system. A failure can be thought of as a "symptom"
- Error
  - Observed difference between a computed result and the correct result i.e., Expected outcome =/= Observed outcome
- The term "**bug**" is used colloquially to mean one or all of the above.

# Terminology

## Mistake

- A human action that produces an incorrect result
- For example, the type *integer* is used instead of *double* causing loss of information during execution.

## Fault / Defect

- Fault is a software defect (incorrect step, process or data definition) that causes a failure. Example, *infinite program loop*.
- The adjudged cause of failure is called a fault. Example: A failure may be cause by a defective block of code.

# Terminology

**Failure**

- The inability of a system or component to perform its required functions within specified performance requirements .
- A manifest observable violation against specification or client expectations by the system.
- For example: Client requirement needs the administrator to be able to add new users, however, the system does not allow it.

**Error**

- The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.
- For example, if an equation should return 100 but is returning 99.95.

# Terminology

## Bug

- A **software bug** is an error, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result.

# Fault and Failure Example

- A patient gives a doctor a list of symptoms
  - Failures
- The doctor tries to diagnose the root cause of the ailment
  - Fault
- The doctor may look for anomalous internal conditions (high blood pressure, irregular heartbeat, bacteria in the blood stream)
  - Errors

# Software Faults, Errors & Failures

- Software Fault : A static defect in the software
- Software Error : An incorrect internal state that is the manifestation of some fault
- Software Failure : External, incorrect behavior with respect to the requirements or other description of the expected behavior

# A Concrete Example

**Fault: Should start searching at 0, not 1**

```
public static int numZero (int [ ] arr)
{  // Effects: If arr is null throw NullPo
   // else return the number of occurrence
   int count = 0;
   for (int i = 1; i < arr.length; i++)
   {
      if (arr [ i ] ==
      {
         count++;
      }
   }
   return count;
}
```

**Test 1**
[ 2, 7, 0 ]
**Expected: 1**
Actual: 1

**Error: i is 1, not 0, on the first iteration**
Failure: none

**Test 2**
[ 0, 2, 7 ]
**Expected:** 1
Actual: 0

**Error: i is 1, not 0**
Error propagates to the variable count
Failure: count is 0 at the return statement

# Terminology

**Example:**

- Consider a medical doctor making a diagnosis for a patient. The patient informs the doctor with a list of **symptoms (i.e. *failures*)**. The doctor then discovers the **root cause (i.e. *fault*)** of the symptoms. The doctor advises the patient to do some medical test and finds that the patient has high blood pressure, irregular heartbeat, high cholesterol etc. (which corresponds to ***error***).

# Why do we have Bugs? ...

- Users specify the wrong requirements.

Business Rule:

"***Salary should be >= 50,000.***"

User Specification:

"***Salary should be > 50,000.***"

# Why do we have Bugs? ...

- Developer misinterprets the requirements.

Requirement:

***"There should be no more than 100 students in each class."***

Interpretation:

***"There should be more than 100 students in each class."***

# Why do we have Bugs? ...

- Requirements are incorrectly recorded.

Business Rule for Tax:

***"First $6000 of income is not taxed."***

***"Next $6000 taxed at 10%."***

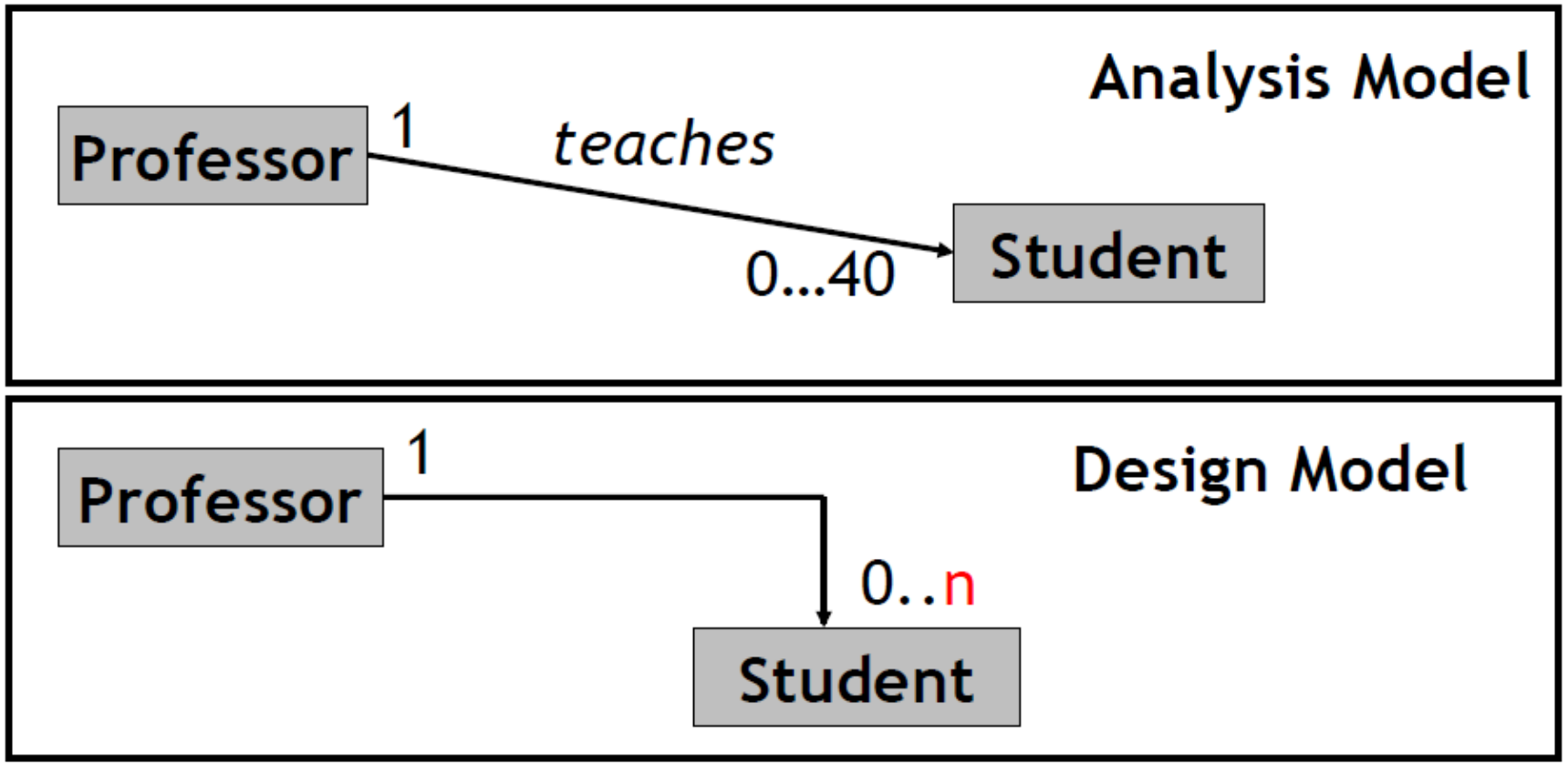
Recorded As:

***"First $600 of income is not taxed."***

***"Next $6000 taxed at 10%."***

# Why do we have Bugs? ...

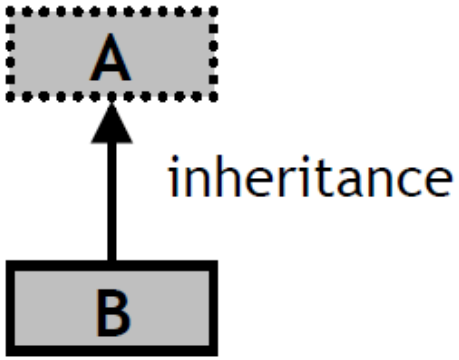
- Design specifications are incorrect or inconsistent with requirements.



# Why do we have Bugs? ...

- Program specifications are incorrect or inconsistent with design.

Design:

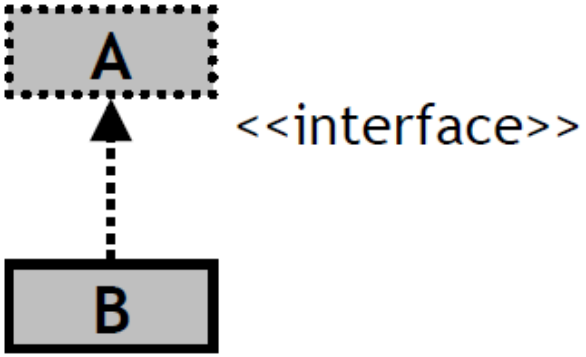



A is abstract class

Implementation:

**Class A is declared as Interface**

**Class B implements realizing interface Class A**



# Why do we have Bugs? ...

- Programming mistakes in the coding phase.

Required:

a >= 52.

Actual:

a >= 5.

# Why do we have Bugs? ...

- Mistakes during data entry - database has invalid data

| Requirement | |
|---|---|
| Salary | Tax Rate |
| **6000** | **0%** |
| **6000-12000** | **10%** |
| **12000 >** | **20%** |

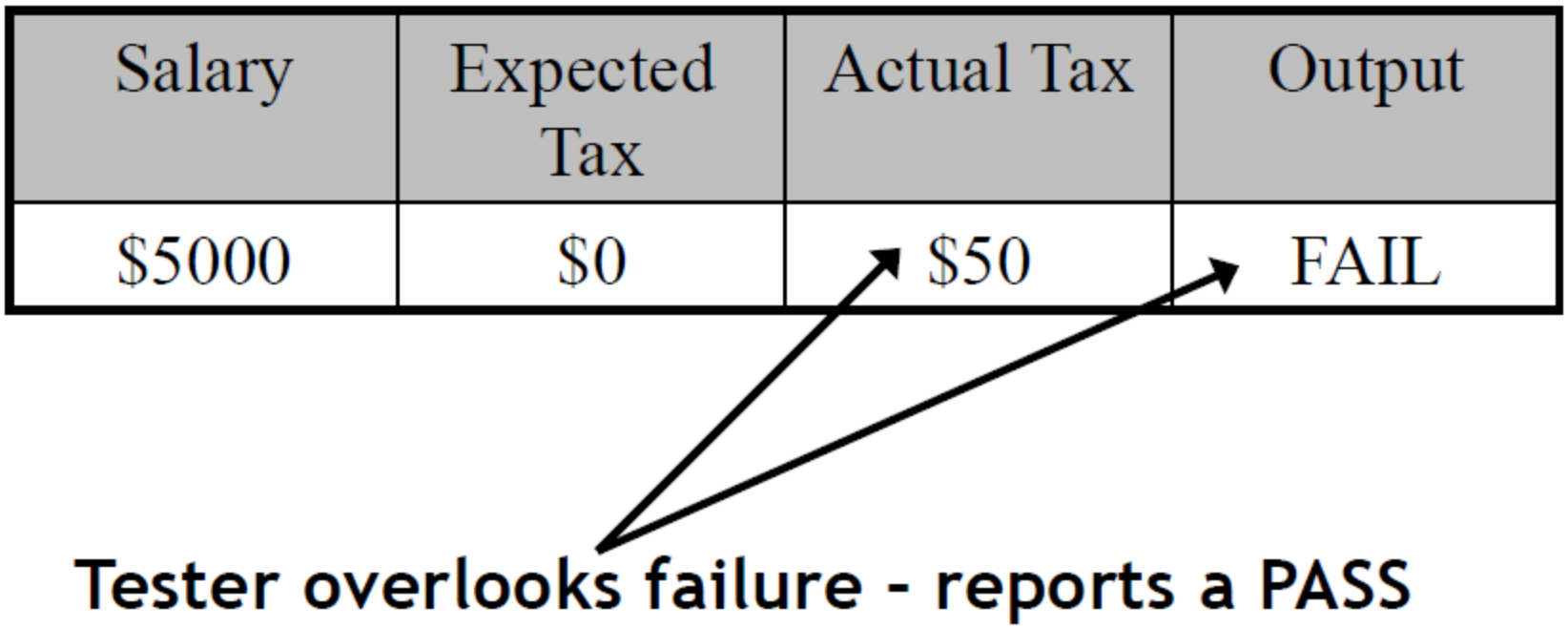
| Entry Made As | | |
|---|---|---|
| Min | Max | Rate |
| 0 | 6000 | 0% |
| 6000 | 12000 | 10% |
| 12000 | > | 20% |

# Why do we have Bugs? ...

- Mistakes during Testing or Verification.

The following represents a faulty test case:

| Salary | Expected Tax | Actual Tax | Output |
|---|---|---|---|
| $5000 | $0 | $50 | FAIL |



# Why do we have Bugs?

- Bug fixes cause new bugs.

# Defective Code Example

```
/** Check if missile can be fired */
public  boolean fireMissile()
{
    if (target = enemy)
    {
        return true;
    }
    else
        return false;
}
```

← MISTAKE!

# Defective Code Example

```
/** Check if missile can be fired */
public  boolean fireMissile()
{
    if (target == enemy)
    {
        return true;
    }
    else
        return false;
}
```

# Defective Code – Thoughts ...

- The consequences of inadequately testing this function are catastrophic.
- Imagine if this code made it into production and the TARGET was friendly?

# Defective Code – Thoughts

- Imagine a large system has 1000 critical bugs like this – How many tests do we have to run to get these all out?

# Some Statistics

- Studies have shown the following about the Defect Rate:

| Novice Developer (1-3 years experience) | Experienced Developer (3-7 years experience) | Expert Developer (over 7 years experience) |
|---|---|---|
| 5-7 mistakes per 100 LOC | 1 mistake per 100 LOC | 0.5 mistake per 100 LOC |

- **LOC – Lines of Code**


FIXED BUG IN CODE
WITHOUT CREATING NEW BUGS
memegenerator.ne

# Software Development - Quality Control

- To ensure functional and quality requirements are met we need a quality control mechanism
- Verification and Validation (V&V) provides this control mechanism
  - Ensure high quality software
  - Combine preventative and corrective measures

# Validation and Verification...

- Validation
  - "Are we building the right product?"
  - The software should do what the user expects
  - More on user side
- Verification
  - "Are we building the product right?"
  - The software should conform to its specification
  - More on product/process side

# Verification & Validation (*IEEE*)

- Verification : The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

- Validation : The process of evaluating software at the end of software development to ensure compliance with intended usage

*IV&V stands for "independent verification and validation"*

# Verification

- "Is the software being built right?"
- "Does the implementation fully and correctly realise the specification?"
- "Do all components exhibit good workmanship, sufficient performance, and conform to applicable standards?"
- Verification addresses the extent of fulfilment of work
- Verification can be
  - Static
  - Dynamic

# Validation

- "Are we building the right product?"
- "Is the specification correct?"
- "Are all user requirements met?"
- Validation addresses the extent of fulfilment of the customer requirements, in any phase
- Techniques include:
  - System Testing and Acceptance Testing
  - Formal reviews

# How To Ensure Your Program Meets Its Requirements?

- Stare at your code for a while and convince yourself it works
  - Easy to make mistakes
  - Easy to be lazy
  - Suffers from “tunnel vision”
  - Most software is “tested” this way
- Prove that it is correct
  - Difficult (very)
  - Not always possible
- Ship it and wait for your customers to complain
  - Not very nice

# How To Ensure Your Program Meets Its Requirements?

- Testing and Debugging

- Testing is a means of detecting/ revealing errors.
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.

# Terminology

**Testing**

- Finding faults or defects in the program or system by using various inputs and conditions.
- Verifying correct behaviour.
- It is the process of evaluating a system with the intent to find whether it satisfies the specified requirements or not.
- In simple words, testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

**Debugging**

- Fix a specific problem of the program or the system by finding the defect and the cause of the defect.
- Checks, detects and corrects errors or bugs to allow proper program operation according to set specifications.

# Real programmers don't test (?)

- I want to get this done fast – testing is going to slow me down.
- I started programming when I was 2. Don't insult me by testing my perfect code!
- Testing is for incompetent programmers who cannot hack.
- We're not MelbourneU students – our code actually works!
- "Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them."

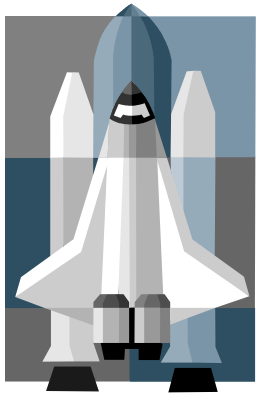
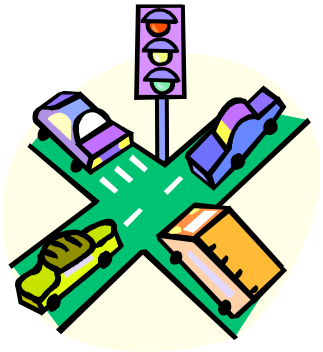
# Software is a Skin that Surrounds Our Civilization










Quote due to Dr. Mark Harman

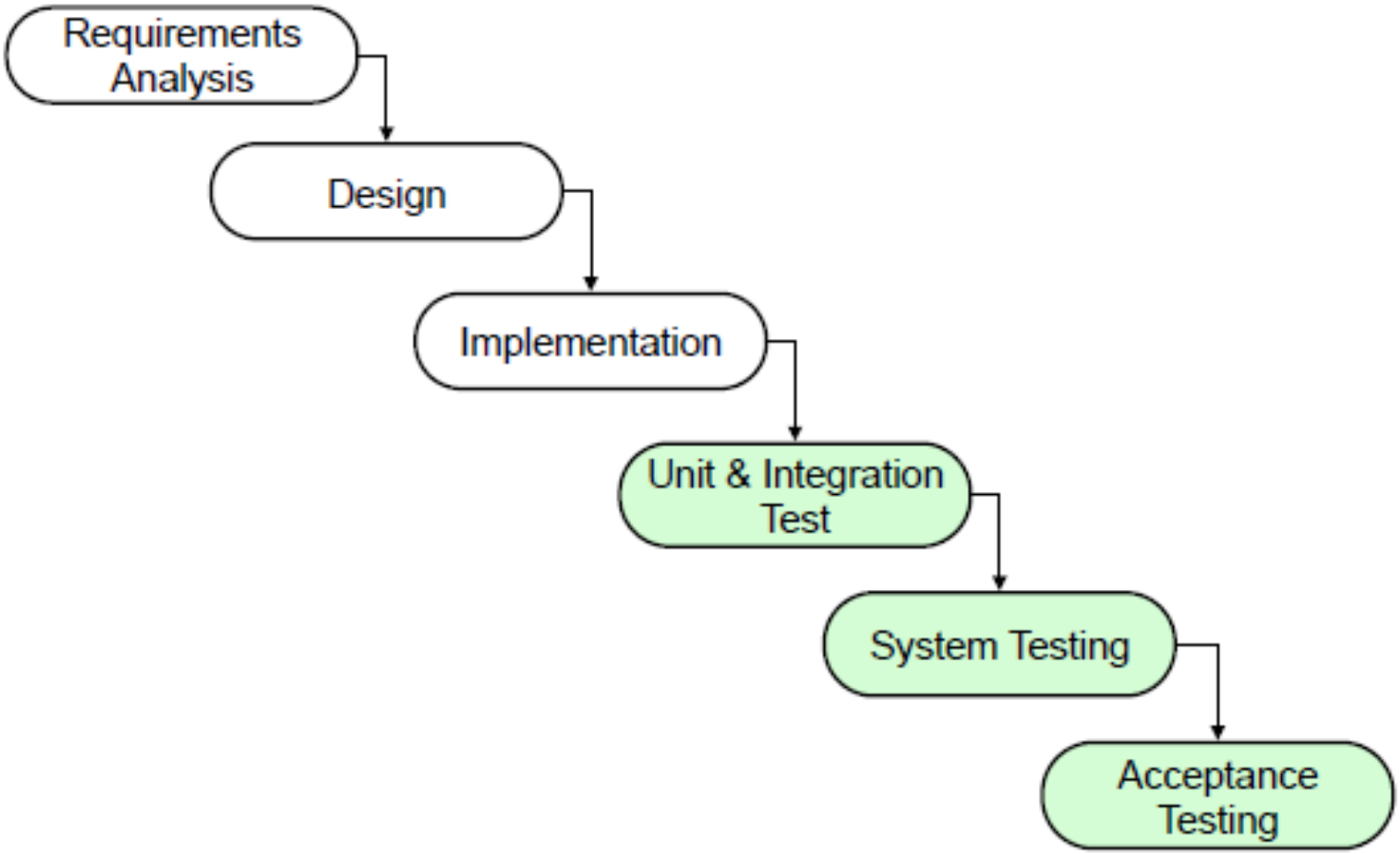
# Testing

- Testing and code review/inspection are the most common quality-assurance methods.
- A practice supported by a wealth of industrial and academic research and by commercial experience.
- "Testing is the process of executing a program with the intention of finding errors." – **Myers**
- "**Testing can show the presence of bugs but never their absence.**" - **Dijkstra**

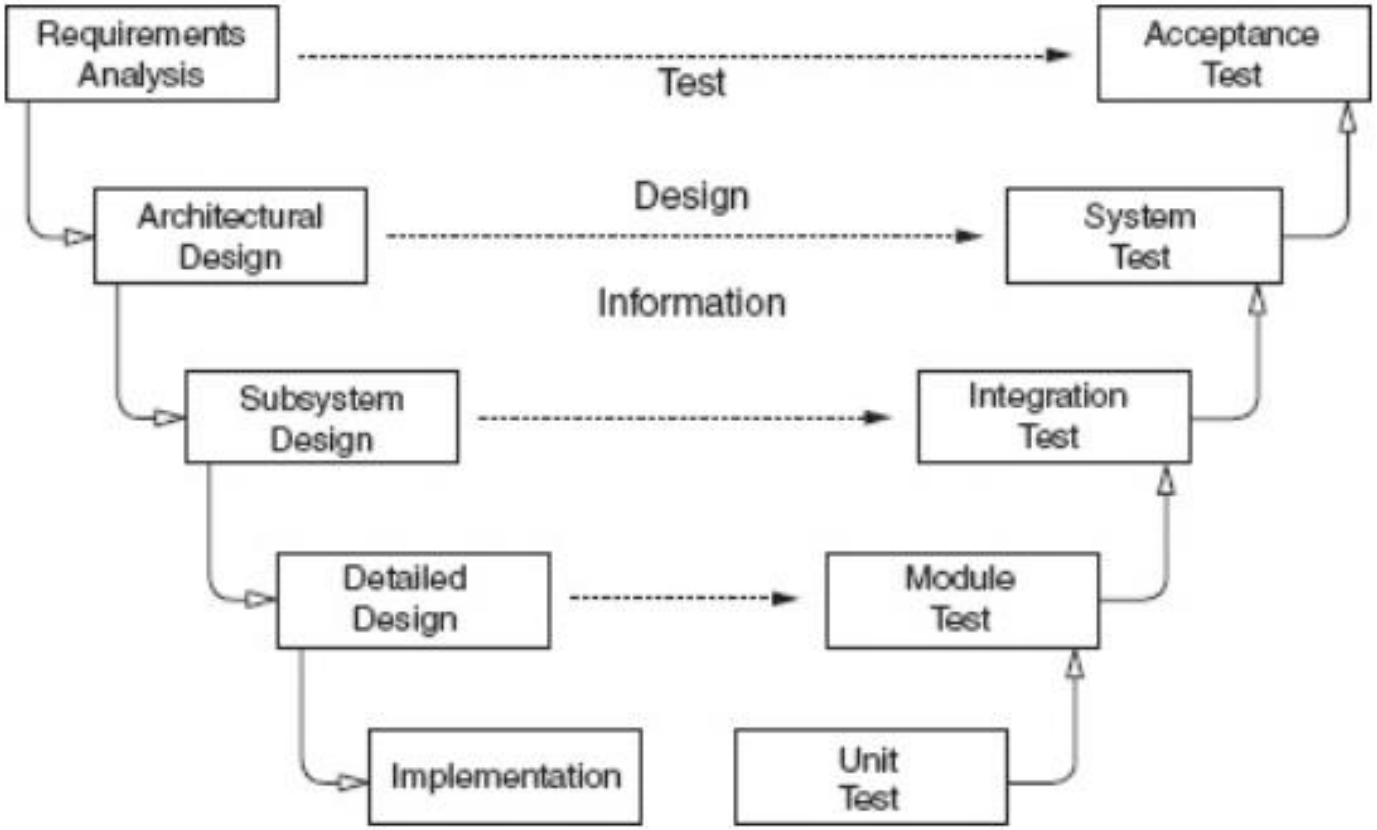
# Testing in the SDLC

- Waterfall
- V-Model
- Exploratory programming
- Throwaway prototyping
- Incremental
- Spiral
- Unified Process
- Agile …

# Validation & Verification in Waterfall Model



# Traditional Testing Levels



# V-Model

- The *requirements analysis* phase of software development captures the customer's needs.
- *Acceptance testing* is designed to determine whether the completed software in fact meets these needs. In other words, acceptance testing probes whether the software does what the users want.
  - Acceptance testing must involve users or other individuals who have strong domain knowledge.

# V-Model Cont.

- The *architectural design* phase of software development chooses *components and connectors* that together realize a system whose specification is intended to meet the previously identified requirements.
- *System testing* is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole.
  - This level of testing usually looks for design and specification problems.
  - It is a very expensive place to find lower-level faults and is usually not done by the programmers, but by a separate testing team

# V-Model Cont.

- The *subsystem design* phase of software development specifies the structure and behavior of subsystems, each of which is intended to satisfy some function in the overall architecture. Often, the subsystems are adaptations of previously developed software.
- *Integration testing* is designed to assess whether the interfaces between modules (defined below) in a subsystem have consistent assumptions and communicate correctly.
  - Integration testing must assume that modules work correct.
  - Integration testing is usually the responsibility of members of the development team.

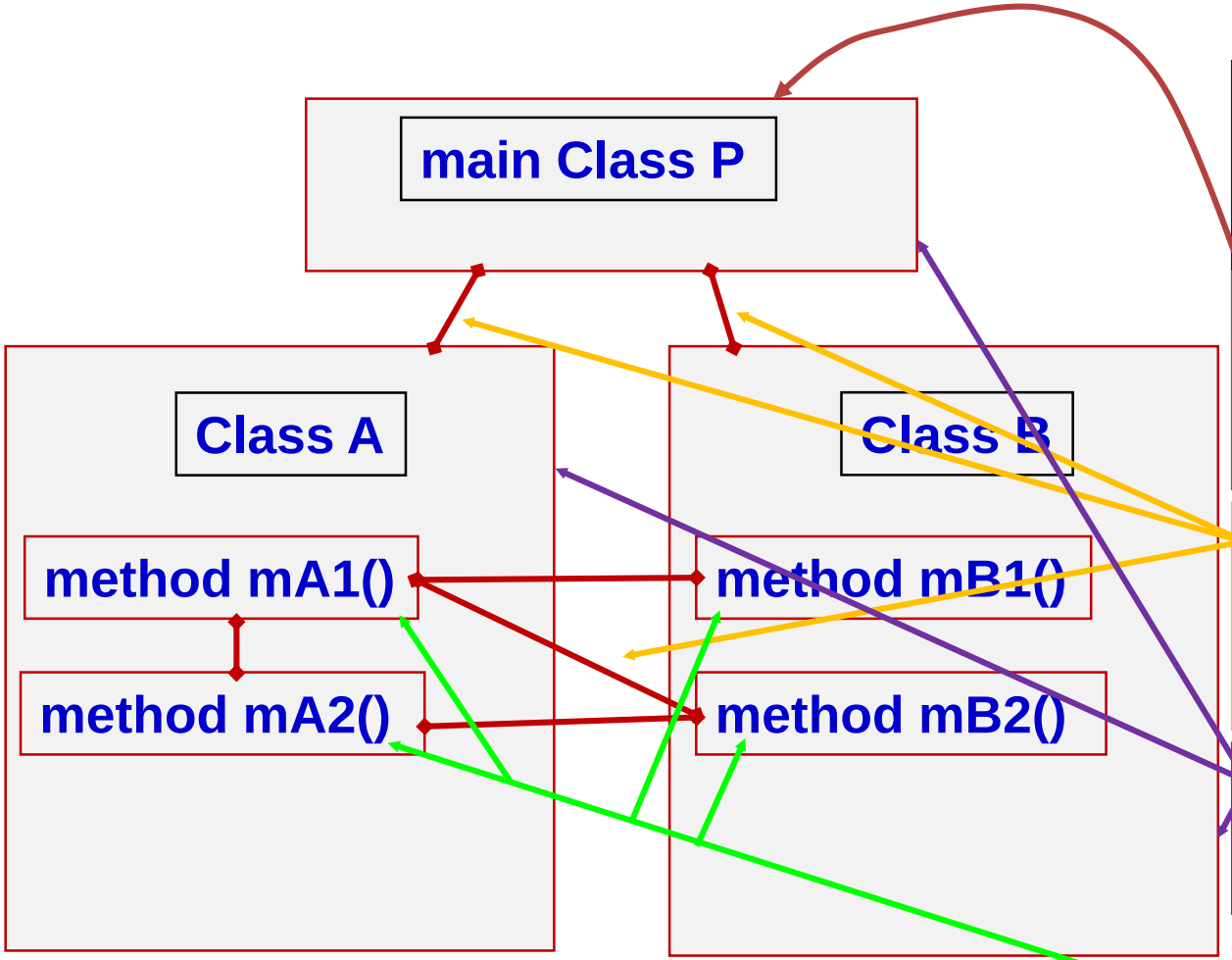
# V-Model Cont.

- The *detailed design* phase of software development determines the structure and behavior of individual modules. A *module* is a collection of related units that are assembled in a file, package, or class.
  - This corresponds to a file in C, a package in Ada, and a class in C++ and Java.
- *Module testing* is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures.
  - Most software development organizations make module testing the responsibility of the programmer; hence the common term *developer testing*.

# V-Model Cont.

- *Implementation* is the phase of software development that actually produces code. A program *unit*, or procedure, is one or more contiguous program statements, with a name that other parts of the software use to call it.
  - Units are called functions in C and C++, procedures or functions in Ada, methods in Java, and subroutines in Fortran.
- *Unit testing* is designed to assess the units produced by the implementation phase and is the "lowest" level of testing.
  - As with module testing, most software development organizations make unit testing the responsibility of the programmer, again, often called developer testing.

# Traditional Testing Levels

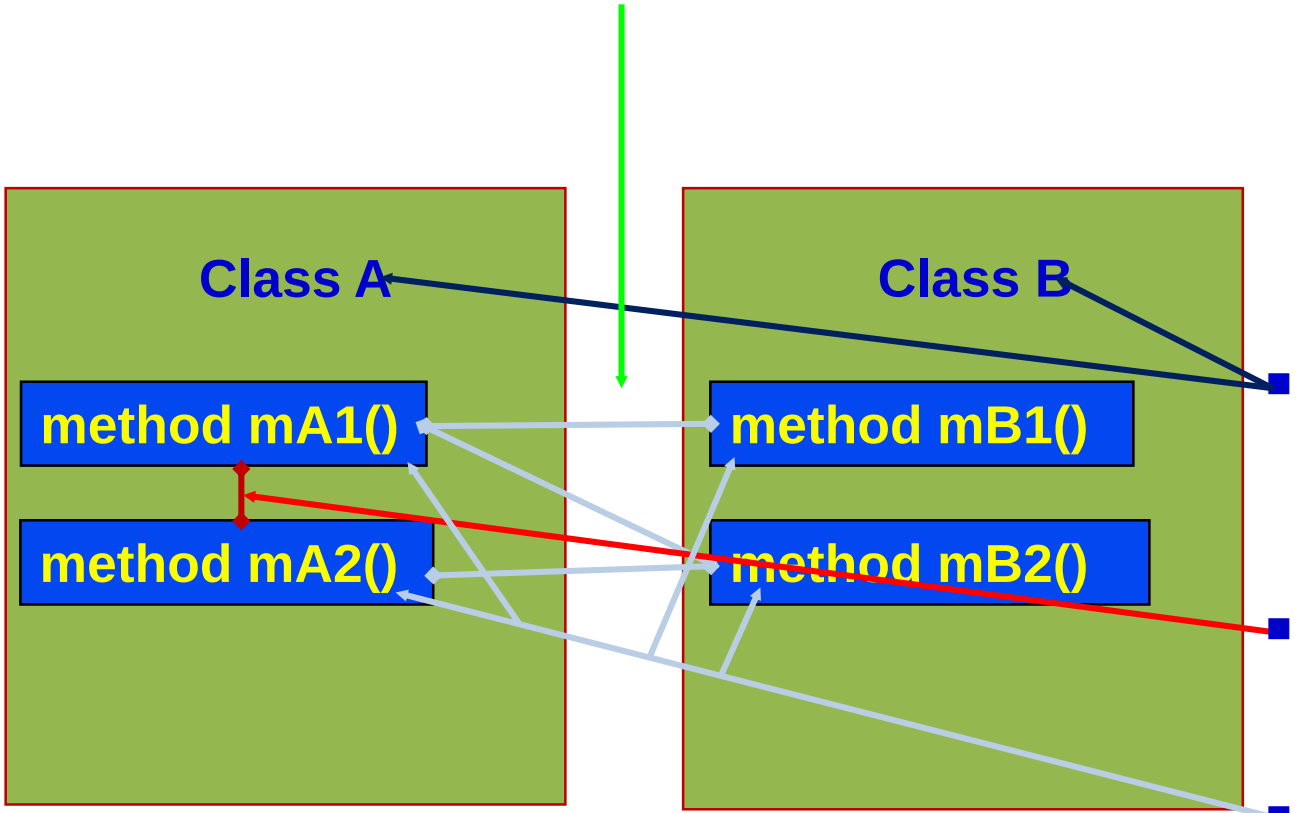



- Acceptance testing : Is the software acceptable to the user?
- System testing : Test the overall functionality of the system
- Integration testing : Test how modules interact with each other
- Module testing (developer testing) : Test each class, file, module, component
- Unit testing (developer testing) : Test each unit (method) individually

**This view obscures underlying similarities**

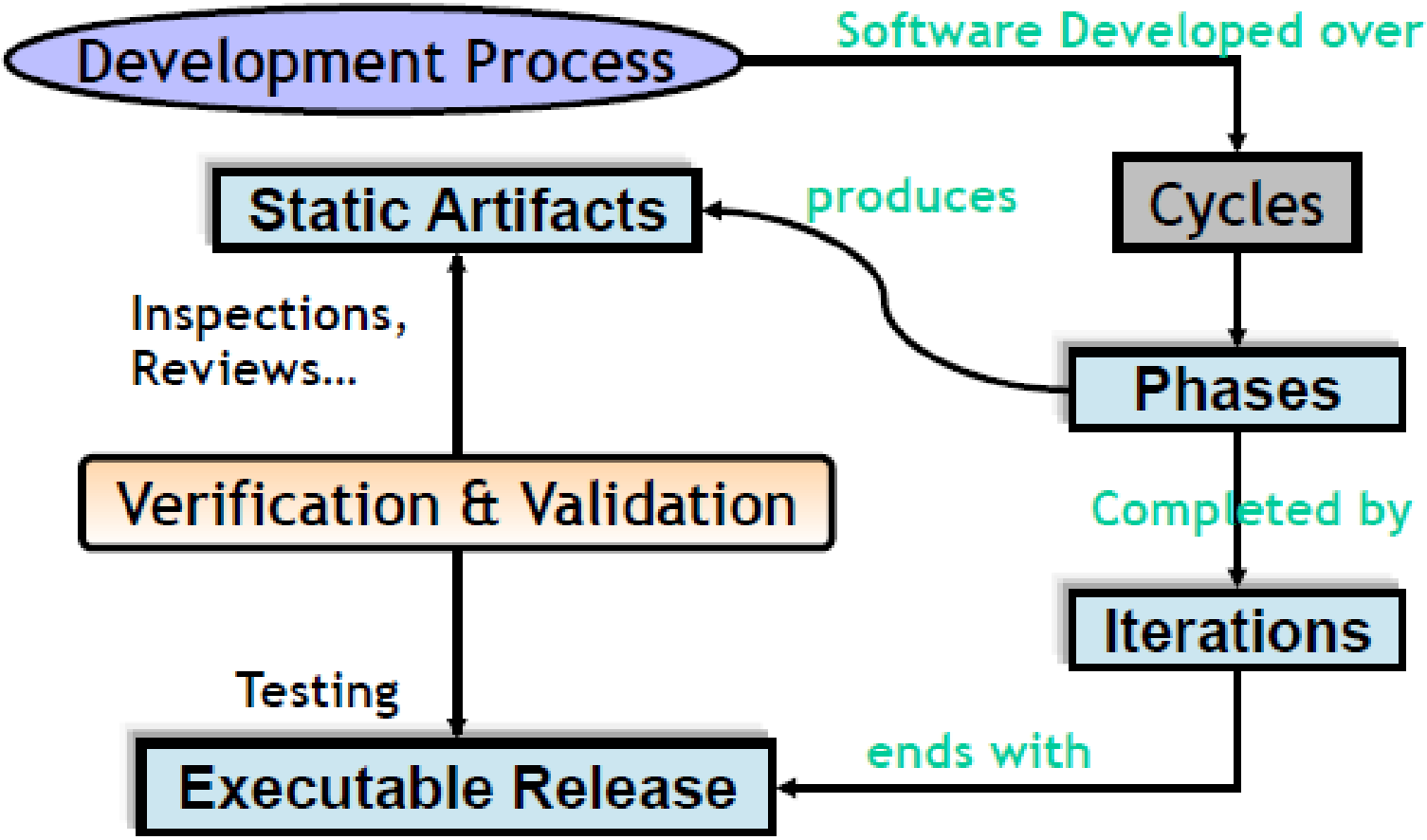
# Object-Oriented Testing Levels

- Inter-class testing : Test multiple classes together




- Intra-class testing : Test an entire class as sequences of calls
- Inter-method testing : Test pairs of methods in the same class
- Intra-method testing : Test each method individually

# Verification and Validation in UP



# Programming is …

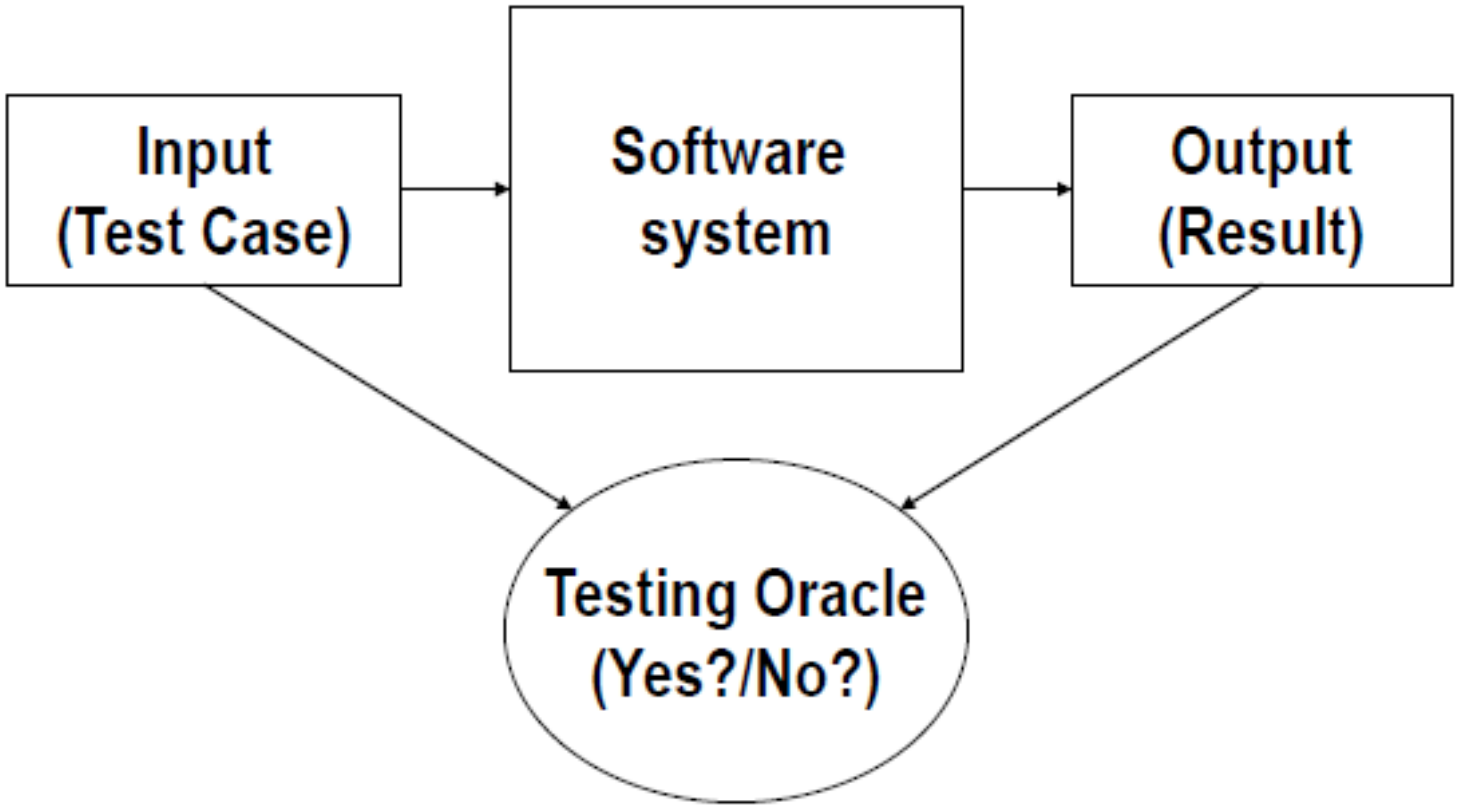
# Kinds of Testing

- ***Unit testing*** is the execution of a complete class, routine, or small program or team of programmers.
- ***Component testing*** is the execution of a class, package, small program, or other program element
- ***Integration testing*** is the combined execution of two or more classes, packages, components, or subsystems.
- ***System testing*** is the execution of the software in its final configuration, including integration with other software and hardware systems.
- ***Regression testing*** is the repetition of previously executed test cases for the purpose of finding defects.

# Testing

- ***Black-box testing*** refers to tests in which the test cannot see the inner workings of the item being tested.
- ***White-box testing*** refers to tests in which the tester is aware of the inner workings of the item being tested.

# Testing – Overview



# Testing - Definition

"The process of operating a system or component under specified conditions, observing or recording results, and making an evaluation of some aspect of the system or component" - IEEE 610

- Simply put the process of testing is:
  - Looking for bugs
  - Checking that the stated requirements are met
  - Verifying that system is reliable and of sufficient quality for release

# Testing - Formal definition ...

- Software Testing
  - The dynamic *verification* of the behavior of a program on a finite set of test cases, suitably selected from the usually *infinite* executions domain, against the specified expected behavior

(Bertolino, 2001)

# Focus of Testing

- Testing starts with two 2 primary goals:
  - Fault detection
  - Establishment of confidence that software conforms to requirements
- Testing ends when:
  - The customer is confident that the system meets stated requirements
  - System exhibits an acceptable level of quality – a low probability of failure, high reliability
  - Testing budget is exhausted

# Views of Testing

Testing can be seen to be:

- Fault-Directed
  - Intent is to reveal faults through failures
  - Also known as negative or dirty testing
- Conformance-Directed
  - Intent is to demonstrate conformance to required capabilities and quality requirements
  - Also known as positive testing

# Testing Terminology ...

- Test Case -
  - A set of *inputs*, execution conditions, and expected results developed for a particular *objective*
  - Exercises a component with the purpose of causing failures and detecting faults
- Test Criteria
  - The criteria that a system or component must meet in order to pass a given test

# Testing Terminology…

- Test Suite
  - A *related* collection of test cases
  - A collection of test case sequences that serve a particular testing goal for a particular version of an Implementation Under Test
  - A whole-part relationship exists among test cases
  - Also known as a Test Bed
- Test Design
  - The process of producing a suite of test cases using a testing strategy
- Test Strategy
  - An algorithm or heuristic to create test cases from a representation, an implementation, or a test model

# Testing Terminology ...

- Test Stub
  - A partial implementation of components on which the tested component depends
  - A stub may simulate the response of a component that has not yet been implemented
- Test Driver
  - A software system used to invoke a component under test
- Test Harness
  - A system of test drivers and other tools to support test execution
- Test stubs and drivers enable components to be isolated from the rest of the system from testing

# Testing Terminology ...

- Test Message
  - Code in a test driver that is sent to a method of the object under test
  - Two or more test messages that follow one another make up a test sequence
- Test Script
  - A program written in a scripting language that executes a set of test suite(s)

# Testing Terminology ...

- Fault Model
  - Identifies relationships and components of the system under test that are most likely to have faults
  - It may be based on common sense, experience, suspicion, analysis, or experiment
  - Each test design strategy has an explicit fault model

# Testing Terminology

- Test Model
  - A testable representation of the relationships among elements of a representation or implementation
  - The test model typically highlights elements of the Implementation Under Test indicated by a fault model
- Example
  - A use case model that has been enhanced with extra information to help with test case design

# Text case - Example

| | | | | | |
|---|---|---|---|---|---|
| **Project Name** | | | | | |
| | BJCS | | | | |
| | | | | | |
| **Test Case ID** | "03.001" | **Test Designed by:** | | | |
| **Test Priority (Low/M** | Medium | **Test Designed date:** | | | |
| **Module Name** | Login Page | **Test Executed by:** | | | |
| **Objective** | Verify login with valid user name and password | **Test Executed date:** | | | |
| **Description** | Test the login page with the valid login details for the work manager | | | | |
| | | | | | |
| **Preconditions** | User has the valid user name and password | | | | |
| **Test setup:** | | | | | |
| **Test teardown:** | | | | | |
| | | | | | |
| **Steps** | **Test Steps** | **Test Data** | **Expected Resul** | **Actual Results** | **Status (Pass/Fai** |
| 1 | Navigate to BJCS login page | http://www | User sees the login page | | |
| 2 | Enter user name | suet@boroona.gov.au | | | |
| 3 | Enter password | 1234 | | | |
| 4 | Click on Login button | | user should be | user is not logg | FAIL |
| | | | | | |
| **Postconditions** | User is validated with the user database and successfully login to account. The account session details are logged in the da | | | | |

# Test case (simplified) - Example

| Test ID: 03.001 | |
|---|---|
| **Type of testing** | Blackbox Testing |
| **Testing objective** | Functional testing calculating income tax to be paid based on the salary. |
| **Environment** | Individual income tax rates (2014-2015) |
| **Test input** | 79,000 |
| **Expected output (or one metamorphic relation)** | 17,222 |
| **Notes** | |

# Activities in Testing

- Identification of test artefact at each level of granularity
- Specification of test objectives
- Test planning
- Test case design
- Execution of test cases
- Capture of results
- Test result analysis
- Regression testing

# Test Levels

- Testing progresses in various levels of increasing detail
  - This models the software system being tested
- The various test levels are:
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
  - Installation testing

# Different Levels of Testing

- Unit Testing
  - test individual unit
- Integration Testing
  - test two or more units together
- System Testing
  - test the system as a whole
  - Function/Performance/User Acceptance Testing
- Installation Testing
- Regression Testing

# Unit Testing

- A unit may be
  - a module (structural approach); or
  - a method in a class (OO approach)
- Test the functionality of each unit

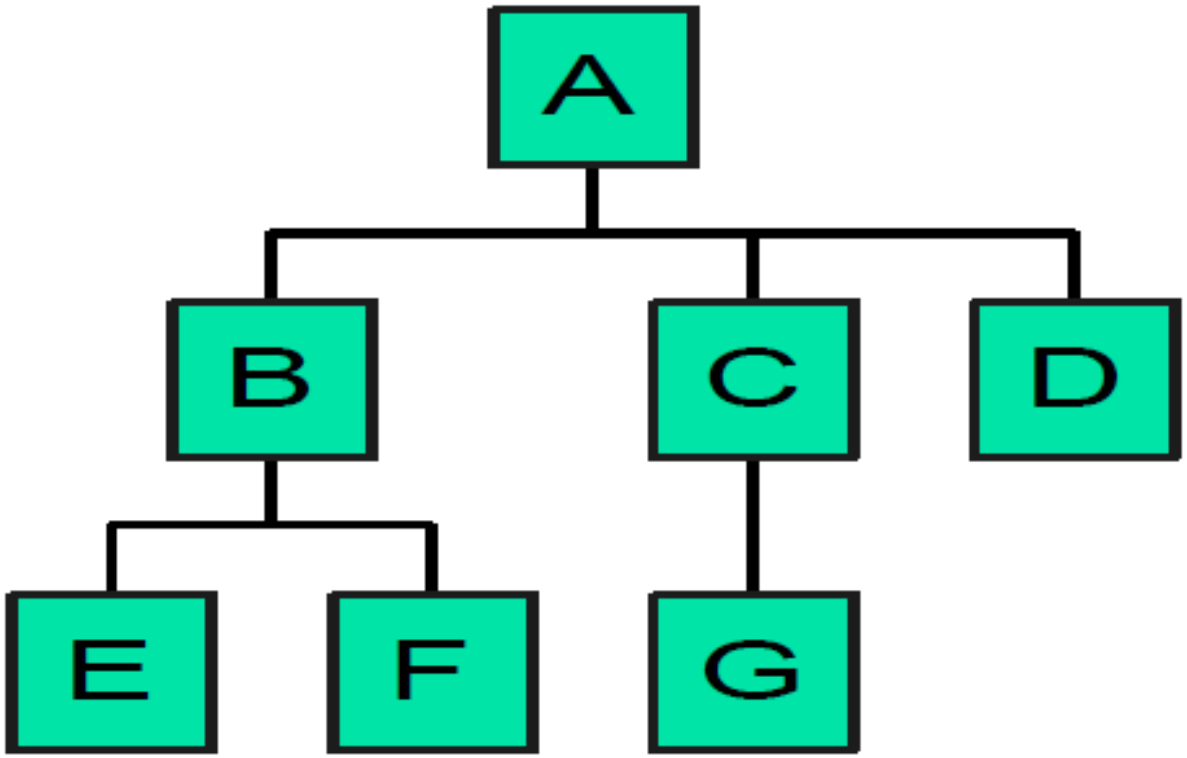
# Integration Testing

- Test two or more units integrated together
- Detect faults related to parameters passing between units to be tested
- Need driver modules and stub modules

# Integration Testing - Different Ways

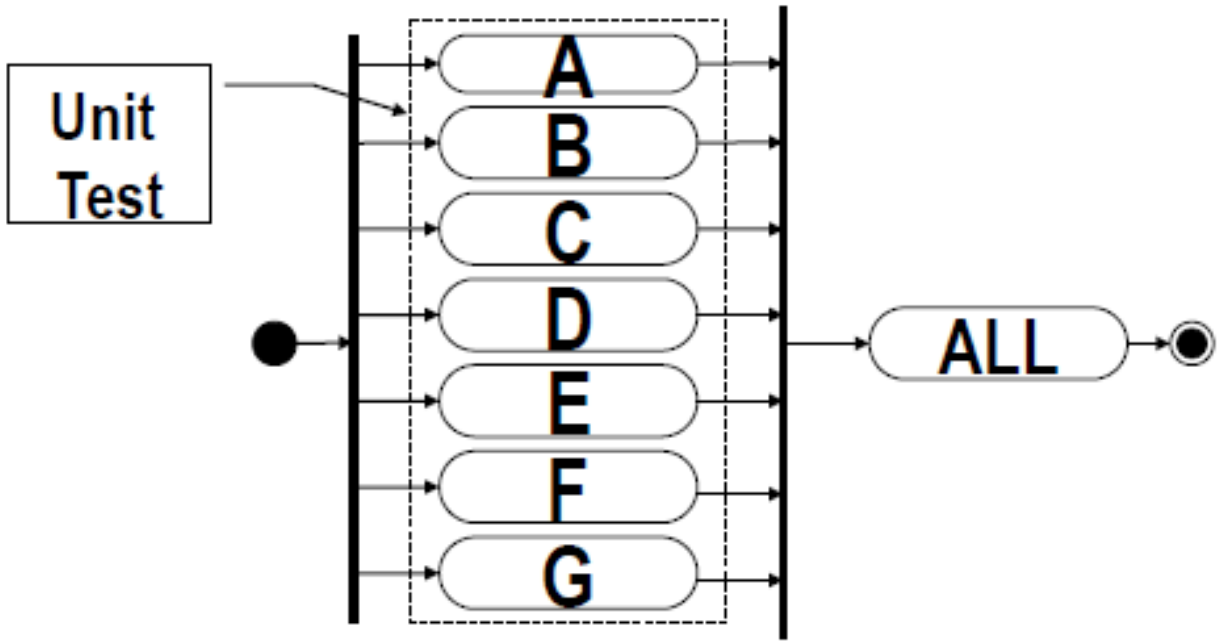
- Non-Incremental
  - Big-bang integration
- Incremental
  - Bottom-up integration
  - Top-down integration
  - Hybrid/Sandwich integration

# Integration Testing - Example



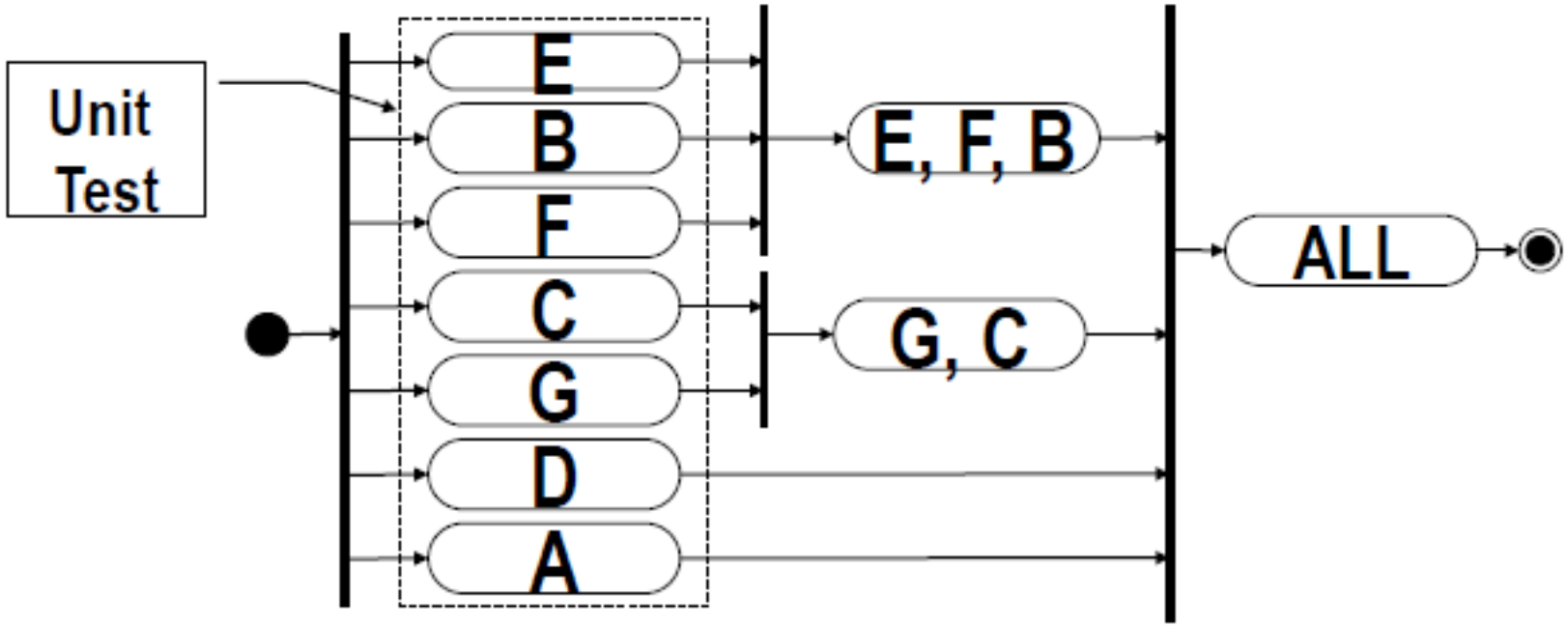
# Big-bang Integration Testing

- After A,B,C,D,E,F,G are unit tested, test ALL of them in one trial



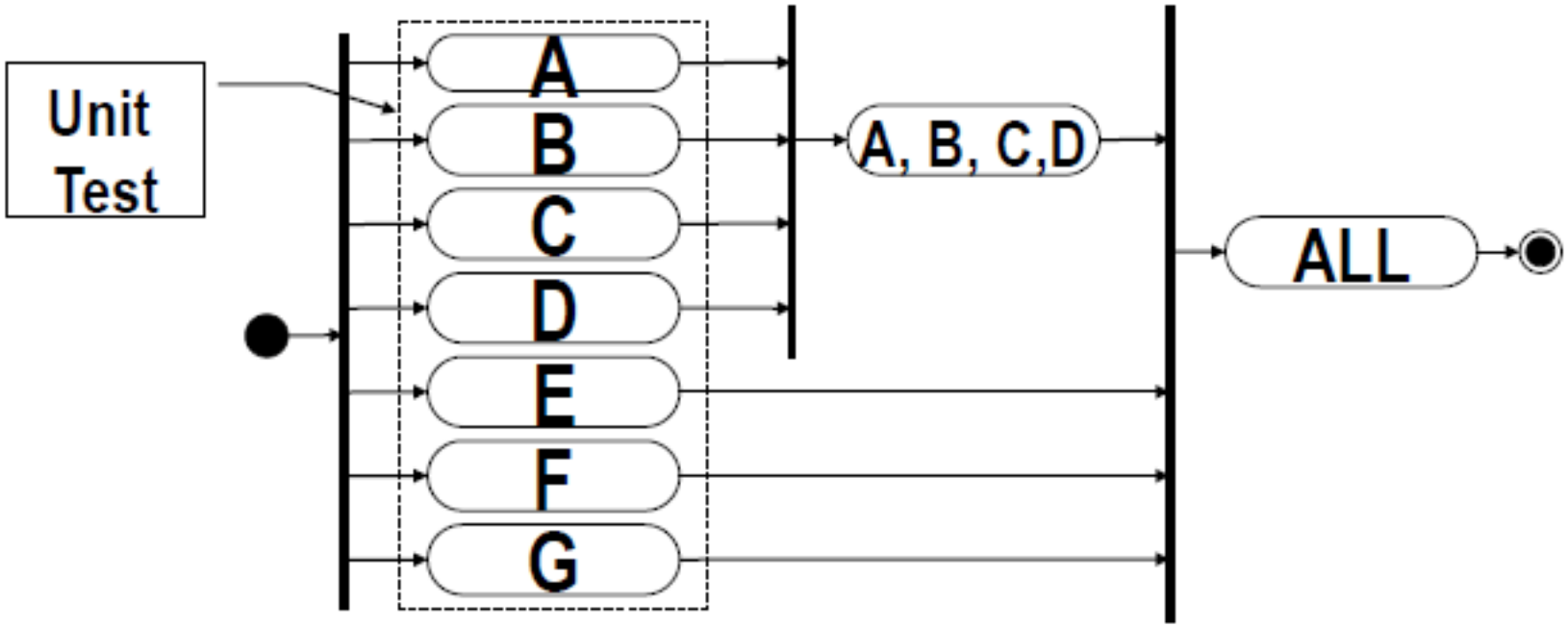
# Bottom-up Integration Testing

- Integrate the units from bottom to top



# Top-down Integration Testing

- Integrate the units from top to bottom



# Hybrid Integration Testing

- A mix between bottom-up and top-down
- One possible way
  - B-E-F and C-G
  - ALL
- Another possible way
  - A-B, A-B-E, A-B-E-F
  - A-B-E-F-C, A-B-E-F-C-G
  - ALL

# System Testing

- Function Testing
  - test the functions of the software
- Performance Testing
  - test the performance of the software
- User Acceptance Testing
  - test the functions and performance of the software in the presence of users

# Function Testing

- Concentrates on correctness of the functions provided by the software
- Need valid inputs as well as invalid inputs

# Performance Testing

- Concentrates on the performance issues such as
  - Volume tests
    - high volumes of data are involved
  - Stress tests
    - design for system that allows multiple users
    - large number of users are involved

# Usability Testing

- A kind of performance testing
- A controlled experiment to address user interface requirements
- Focus on usability parameters
- End-users explore the look and feel of screens, messages, reports, etc. in a laboratory environment (such as the one in Swinburne Computer Human Interaction Laboratory SCHIL)
- 3 types
  - Scenario test
  - Prototype test
  - Product test

# User Acceptance Testing

- Testing done with the presence of users to demonstrate the software works according to the users' requirements
- Alpha test
  - within the developer's organization
- Beta test
  - outside the developer's organization

# Installation Testing

- Testing after installation of system in its working environment
- Final test before system becomes operational

# Regression Testing

- Testing of new version/release to identify any new faults introduced during correction or upgrade of system
- Regression testing of new functionalities
  - test all functions to be affected by the new code
  - test essential functions of the previous version
  - perform function testing of the new version

# Test Levels and Focus Area

| Test Level | Core focus |
|---|---|
| **Unit** | **Method, Class, Component** |
| **Integration** | **Cluster, Package, Sub-System** |
| **System** | **Entire System – verification of function & performance** |
| **Acceptance** | **Entire system from users view** |
| **Installation** | **Entire system from deployment view** |

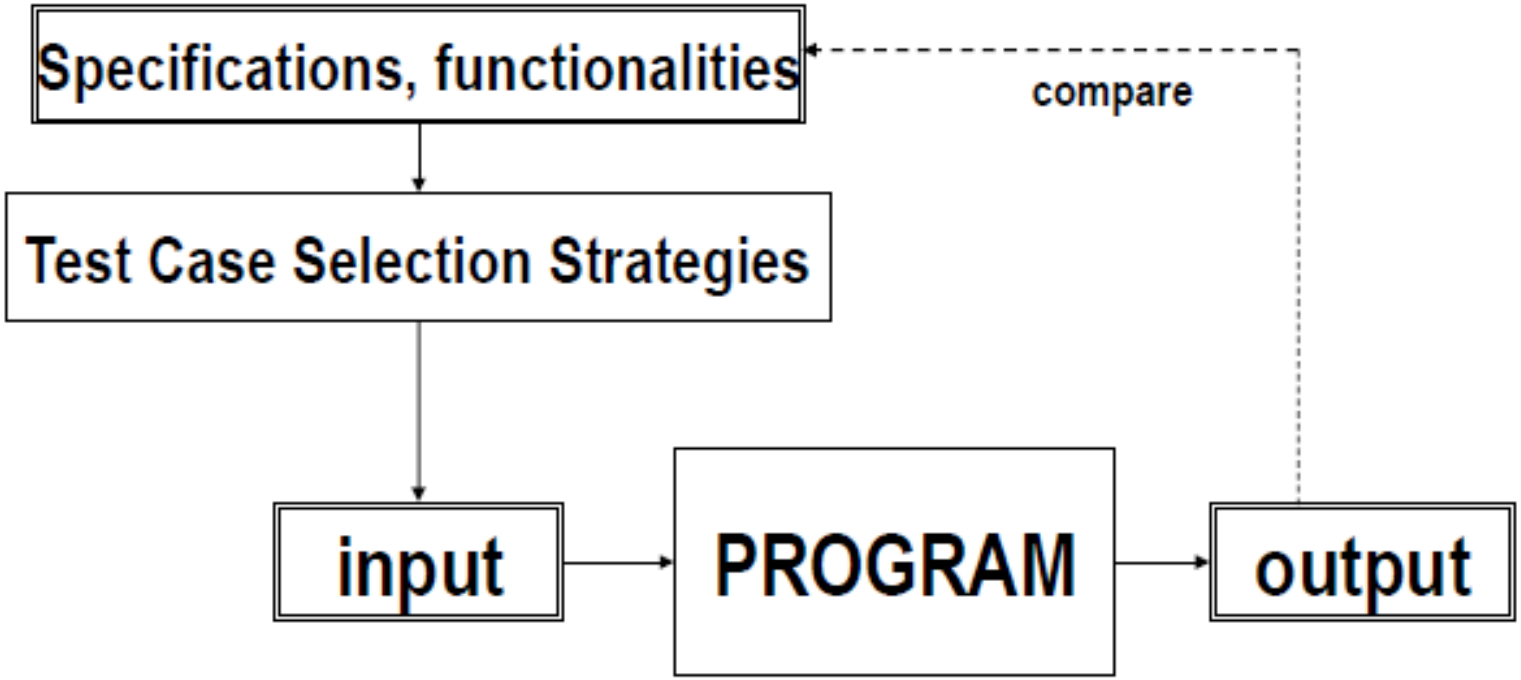
# Test Design Methods

- Tests can be designed from a functional or a structural point of view
- These three views provide for key forms of testing:
  - Functional Testing – Black-Box Testing
  - Structural Testing – White-Box or Glass-Box Testing
  - Hybrid Testing – A combination of the above two

# Black Box Testing

- The software is treated as a black-box – testers are not aware of the internals
- Focus is on verifying that the system functions as specified
- Takes the end-users' point of view
- In principle can detect all bugs but would take an infinite time to do so
- Also known as responsibility based testing
- Functional test design is used more at later stages of the testing steps – Function, Acceptance, Installation

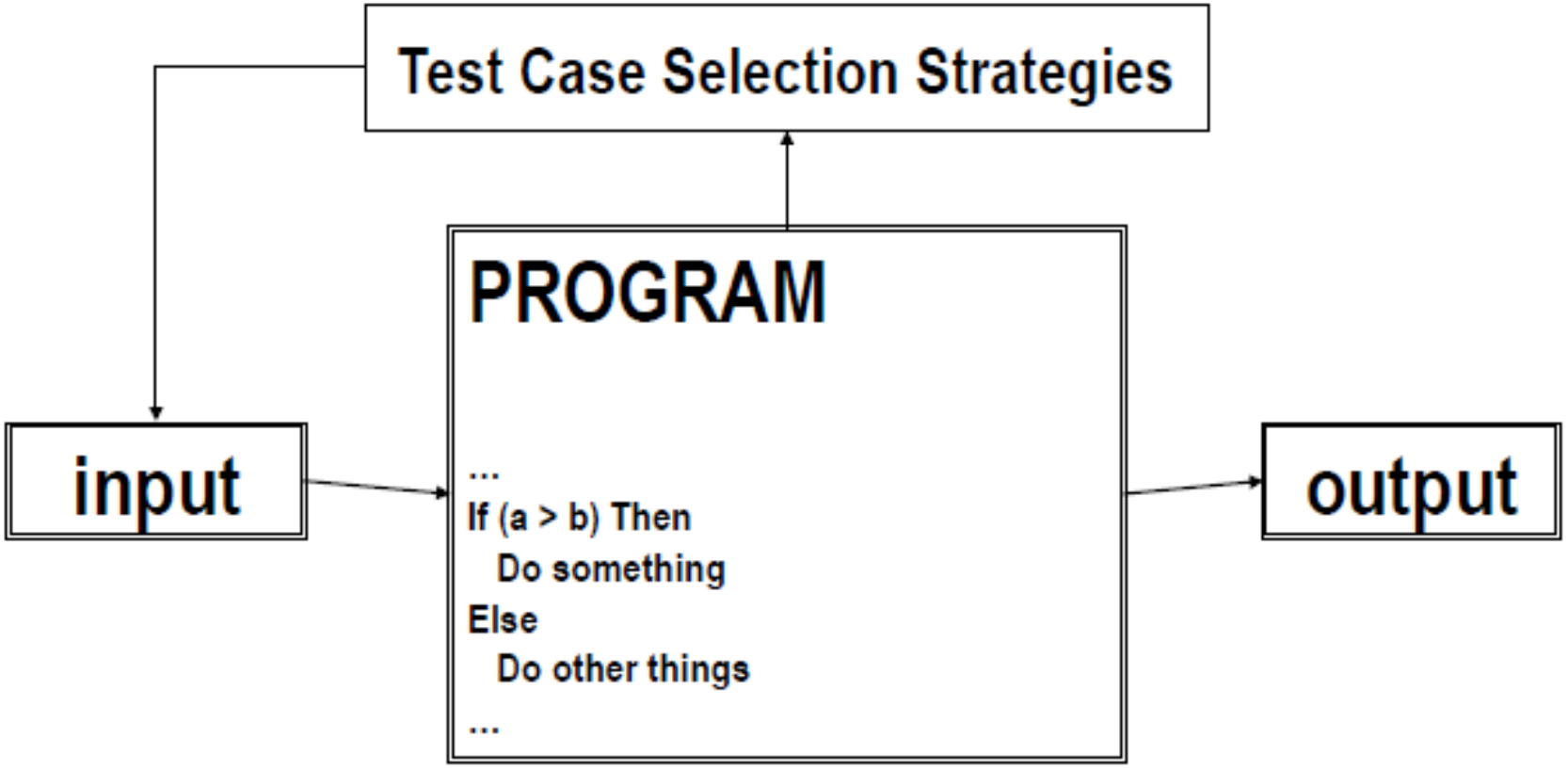
# Black Box Testing – A common view



# White Box Testing

- Looks at the implementation and design of the code to develop the test cases
- This method of testing can help in verifying that we cover all possible paths of execution through the program
- This is also known as Glass-Box Testing or Implementation Based Testing
- Structural test design techniques are applied at early testing stages – Unit and Integration

# White Box Testing – A common view



# Models, Testing and Validation