SOFT 3416
Software Verification and Validation

Week XIV

Ahmet Feyzi Ateş

Işık University Faculty of Engineering and Natural Sciences
Department of Computer Science Engineering
2023-2024 Spring Semester

Model Based Testing (MBT)

Learning Objectives:

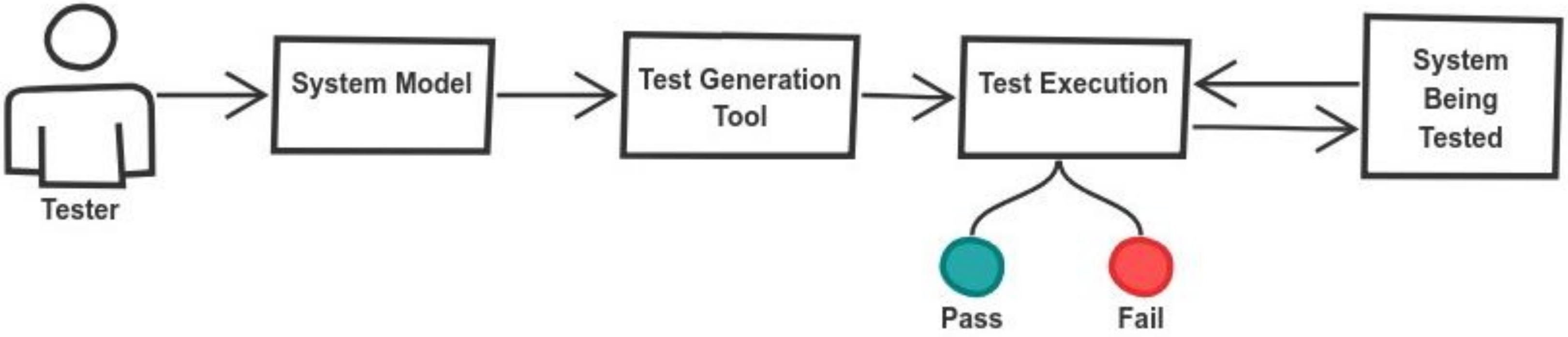
- What is MBT
- Why MBT
- Goals of MBT
- How MBT is achieved

What is Model Based Testing

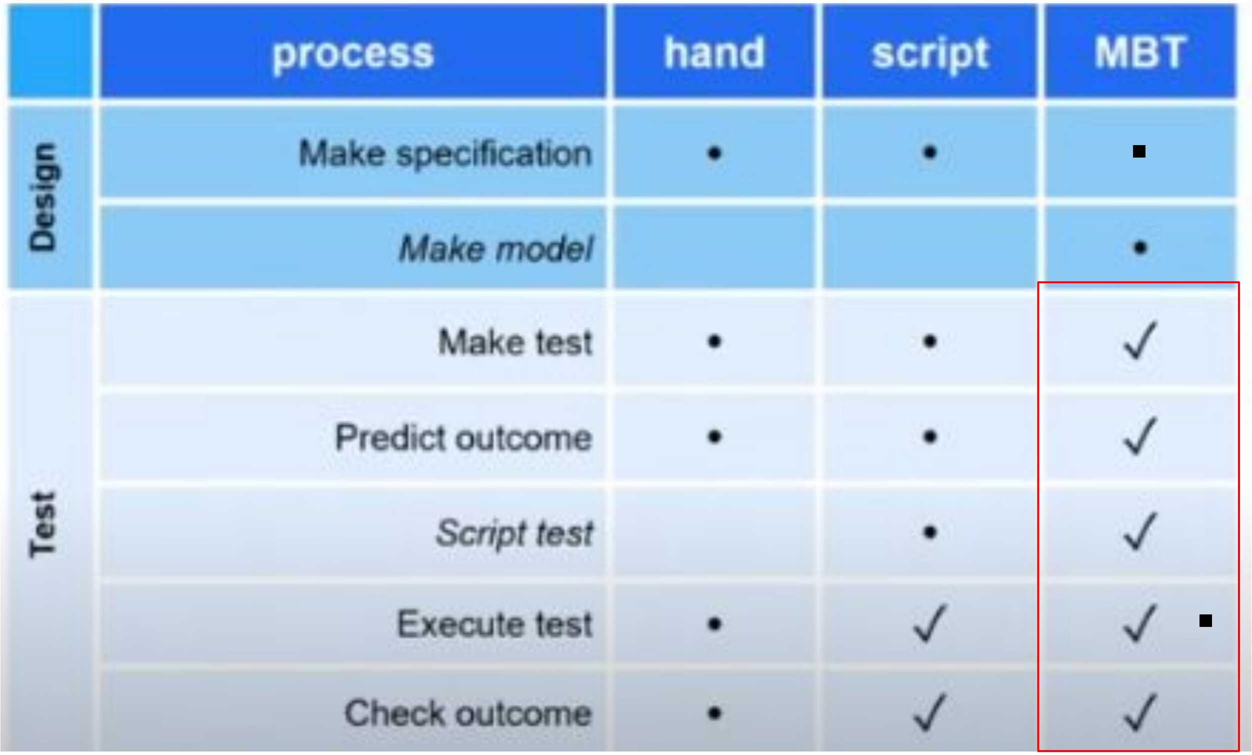
- Model-based testing is a type of software testing method that uses a system’s model under test to generate test cases.
- Test automation tools that use this approach can create tests automatically from the model or semi-automatically with some user input.
- This approach can be used for any software testing but is particularly well suited for testing complex systems with many possible states or behaviors.
- Model based test automation can help reduce the time and effort required to create and maintain manual test cases and can also help improve the coverage and accuracy of tests.

How does Model-Based Testing work

- Model-based test is a methodology that uses a model of the system under test to generate test cases.
- The model can be either static or dynamic.
- Static models are typically used for GUI testing, while dynamic models are used for API testing.
- To generate Model test cases, the tester does the following:
  - Creates a representation of the system under test using a graphical tool, such as a UML diagram, or by writing code.
  - Once the model is created, define each test case's input and expected output values.
  - Executes the test case and compares the actual output to the expected outcome. If there is a discrepancy, then the tester reports a failure.



Test Strategies Comparison



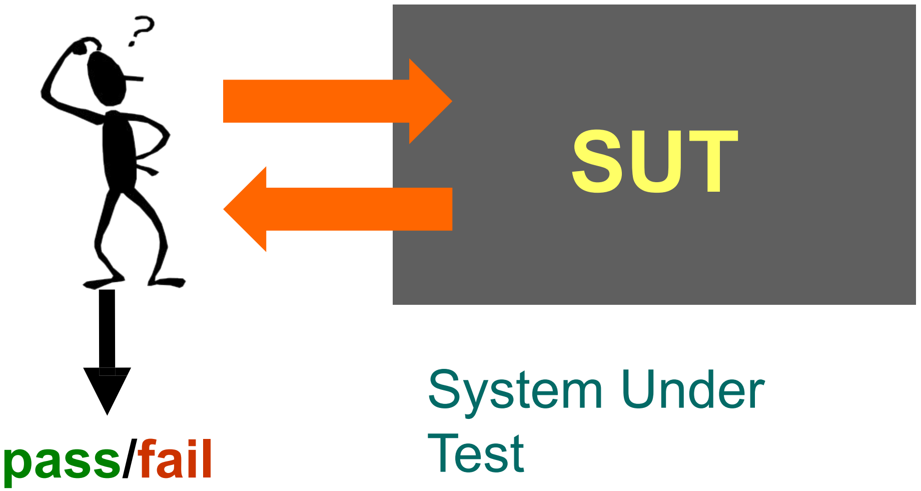
	process	hand	script	MBT
Design	Make specification	•	•	■
	Make model			•
Test	Make test	•	•	✓
	Predict outcome	•	•	✓
	Script test		•	✓
	Execute test	•	✓	✓ ■
	Check outcome	•	✓	✓

- Manual step

✓ Automated

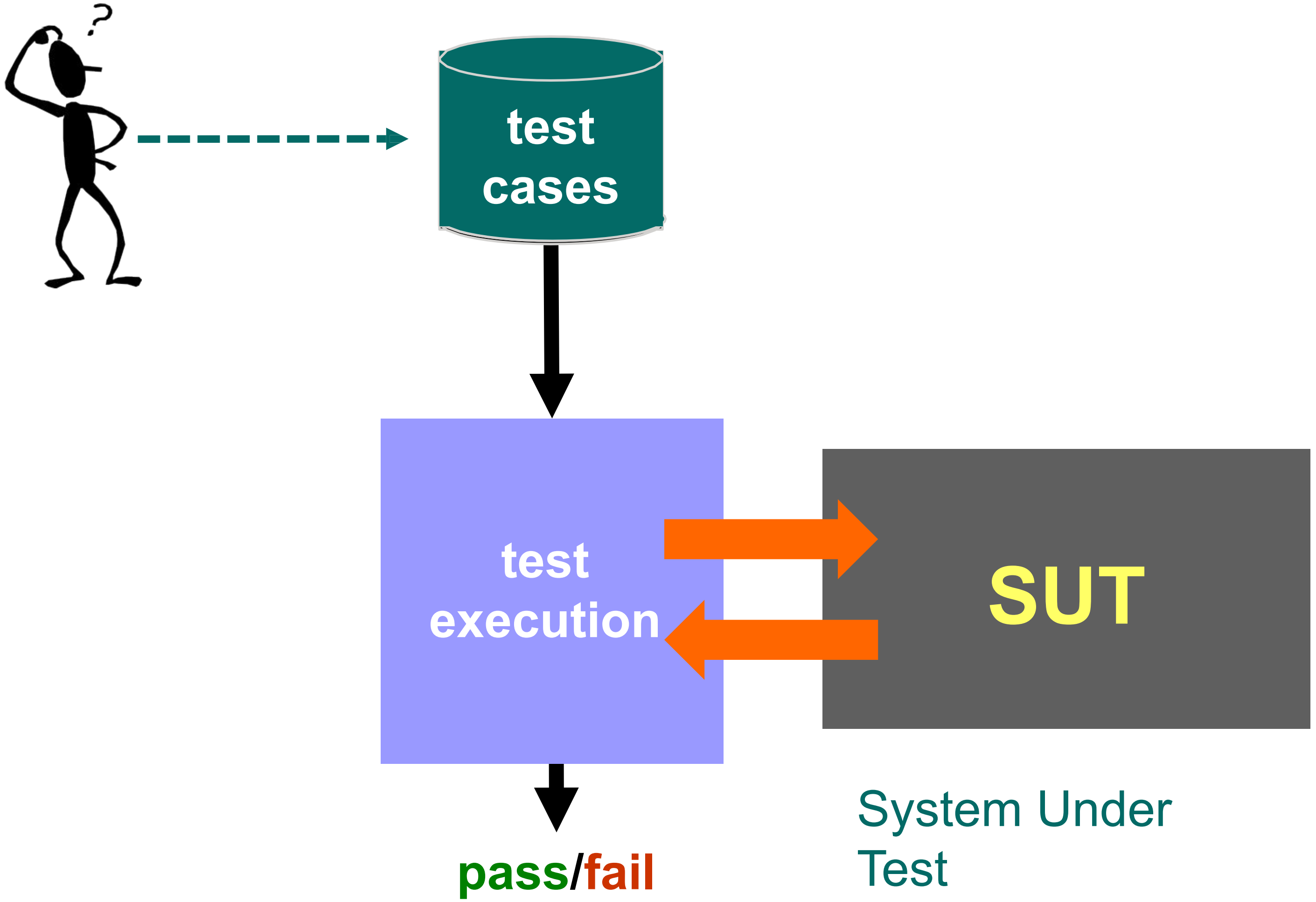
Manual (hand) Testing

All test cases are carried out manually, one by one, by the tester; and a verdict of ***pass/fail*** is arrived for each test case.



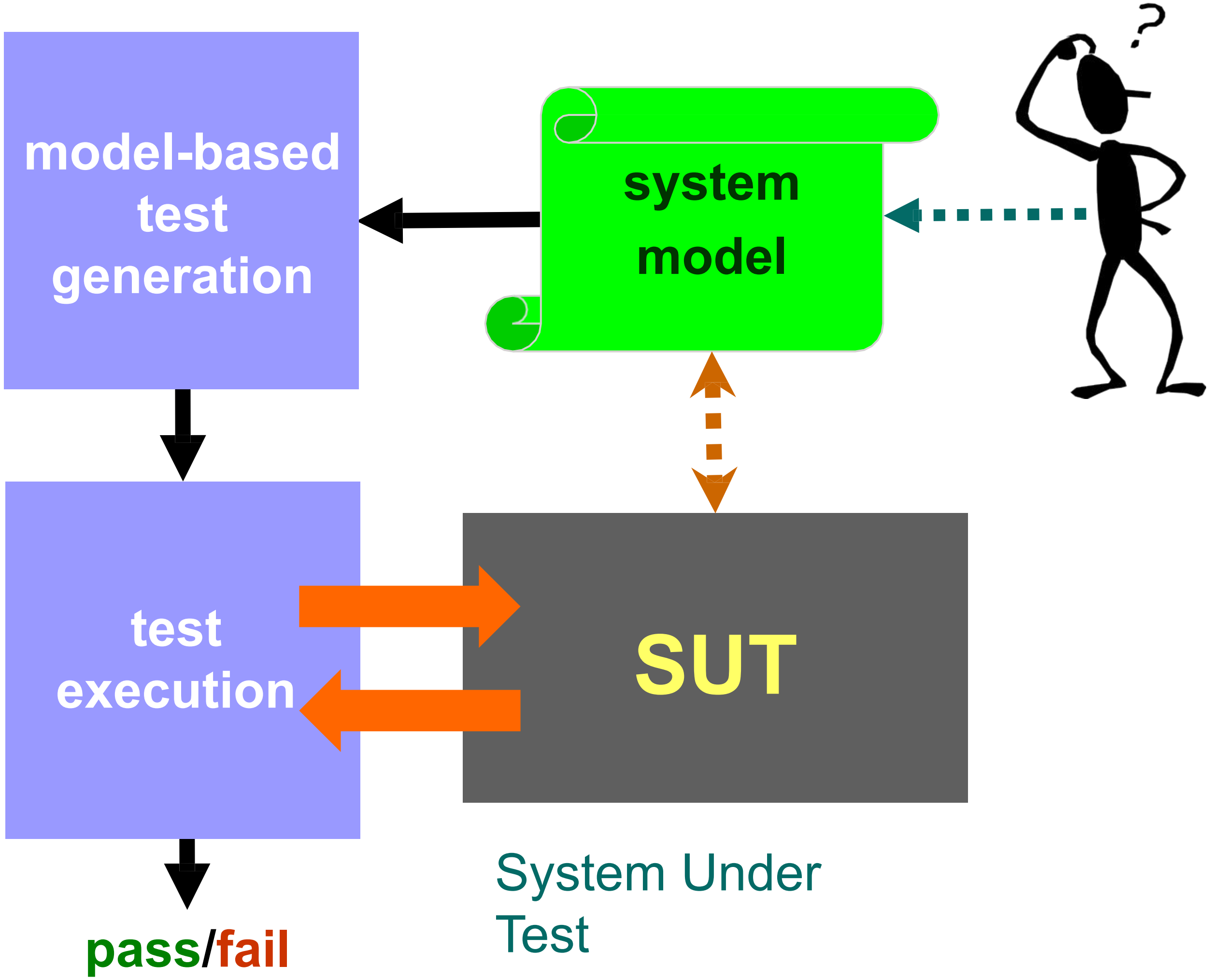
Scripted Testing

Test cases are written by the tester as scripts within a specific test harness tool, and then the tool executes each test case automatically; and the tool arrives at the pass/fail verdict for each test case according to a test oracle.

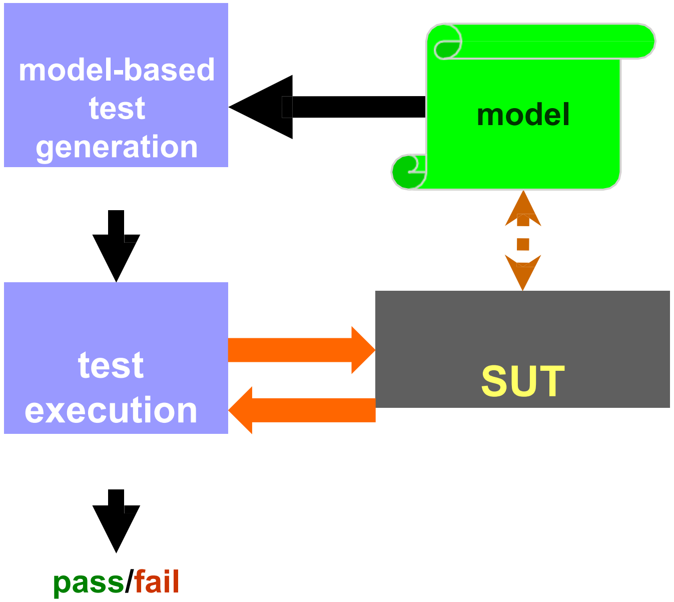


Model Driven Testing

The system is modelled formally or semi-formally in machine-processable formats, test case are generated from the model automatically by tool. Generated test cases can be executed manually or automatically, while automated execution is generally preferred for obvious productivity reasons



MBT Benefits



detecting more bugs

faster and cheaper

MBT: next step in test automation

- **Automatic test generation**

  + test execution + result analysis

- **More, longer, and diversified test cases**

  more variation in test flow and in test data

- **Model is precise and consistent test basis**

  unambiguous analysis of test results

- **Test maintenance by maintaining models**

  improved regression testing

- **Expressing test coverage**

  model coverage

  customer profile coverage

Testing Based on Models

- The essence of model-based testing is the sequence of steps:
  - Model the system.
  - Identify threads of system behavior in the model.
  - Transform these threads into test cases.
  - Execute the test cases (on the actual system) and record the results.
  - Revise the model(s) as needed and repeat the process.

MBT Objectives

- Model-Based Testing (MBT) is an advanced test approach using models for testing.
- It extends and supports classic test design techniques such as equivalence partitioning, boundary value analysis, decision table testing, state transition testing, and use case testing.
- The basic idea is to improve the quality and efficiency of the test design and test implementation activities by:
  - Designing a comprehensive MBT model, typically using tools, based on project test objectives.
  - Providing an MBT model as a test design specification. This model includes a high degree of formal and detailed information that is sufficient to automatically generate the test cases directly from the MBT model.

- The ultimate objective is to avoid the tester writing each test case manually so that he/she can concentrate on tasks with higher added value.

MBT Specific Activities

- Activities specific to MBT are:
  - ***MBT modeling activities***
    - activities related to the administration of the MBT model, development and integration of the modeling approach, definition of modeling guidelines, development of the structure of the MBT model, development of model elements, e.g., specific diagrams of the MBT model, tooling related activities.
  - ***Generation of test cases*** based on the MBT approach and selection criteria,
- At a minimum, model-based testing impacts test analysis and test design activities in the fundamental test process.
- Depending on the test objectives, MBT can also be used with a broader scope relating to all parts of the fundamental test process.

MBT Specific Activities (cont'd)

- The following MBT specific activities should be considered:
  - ***Test planning and control***
    - can include implementation of the MBT specific activities of the fundamental test process (MBT tooling, guidelines, specific metrics, MBT artifacts as part of the baseline for project planning, etc.).
  - ***Test analysis and design***
    - can include MBT modeling activities, test selection criteria and advanced test coverage metrics. Test implementation and execution can include MBT test generation and MBT test adaptation.
  - ***Evaluation of exit criteria and reporting***
    - can include advanced coverage metrics (based on structural and/or explicit model information) and MBT-based impact analysis. Test closure activities can include establishing model libraries for re-use in future projects.

MBT Specific Activities (cont'd)

- A focus point of the impact of MBT on the test process is process automation and artifact generation.

- MBT promotes a shift of test design into earlier project phases when compared with classic test design.

- It can serve as an early requirements verification method and can promote improved communication especially by using graphical models.

- Model-based tests often augment traditional tests. This provides an opportunity for the test team to verify consistency between the automatically generated MBT test cases and the manually created traditional test cases.

Essential MBT Artifacts

- ***Input artifacts:***
  - Test strategy
  - The test basis including requirements and other test targets, test conditions, oral information, and existing design or models.
  - Incident and defect reports, test logs and test execution logs from previous test execution activities Method and process guidelines, tool documents.
- ***Output artifacts:***
  - Different kinds of testware, such as: MBT models
  - *Parts of the test plan* (features to be tested, test environment, ...), test schedule, test metrics, Test scenarios, test suites, test execution schedules, test design specifications
  - Test cases, test procedure specifications, test data, test scripts, test adaptation layer (specifications and code) Bidirectional traceability matrix between generated tests and the test basis, especially requirements, and defect reports

MBT Modelling

- Designing the MBT model is an essential and demanding activity in MBT.

- The quality of the MBT model has a strong impact on the quality of the outcome of the MBT-based test process.

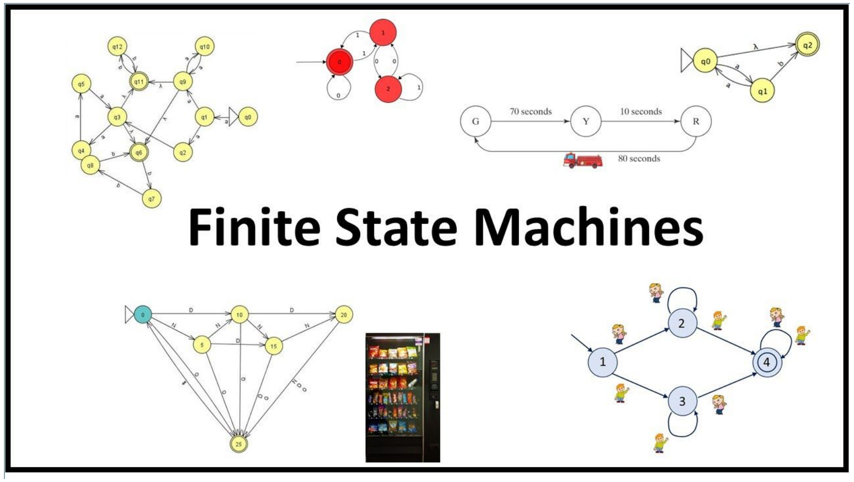
- MBT models are often interpreted by tools and must follow a strict syntax.

- MBT models may also need to support other quality characteristics such as those defined in modeling guidelines.

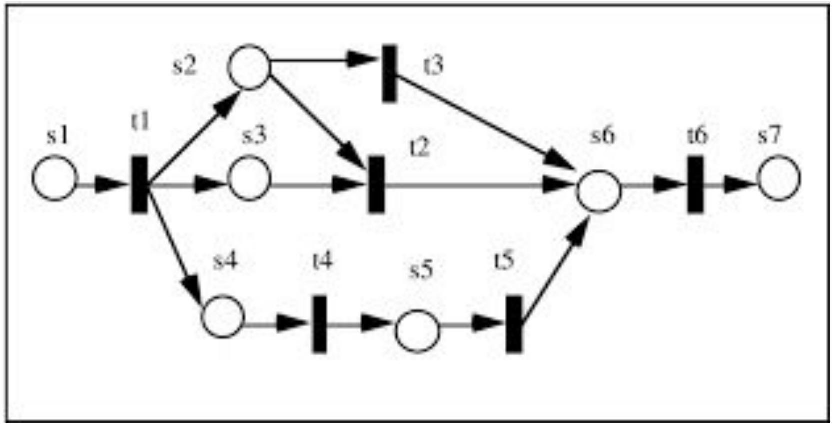
MBT Modelling (cont’d)

- Three types of model subjects are:
  - ***System model:*** model describes the system as it is intended to be.
    - Test cases are generated from this model check if the system conforms to this model.
    - Examples of system models are class diagrams for object-oriented systems or state diagrams describing the states and state transitions of reactive systems.
  - ***Environment model / Usage model:*** describes the environment of the system.
    - Examples of environment models include Markov chain models describing the expected usage of the system.
  - ***Test model:*** is a model of (one or several) test cases.
    - This typically includes the expected behavior of the test object and its evaluation.
    - Examples of test models are (abstract) test case descriptions or a graphical representation of a test procedure

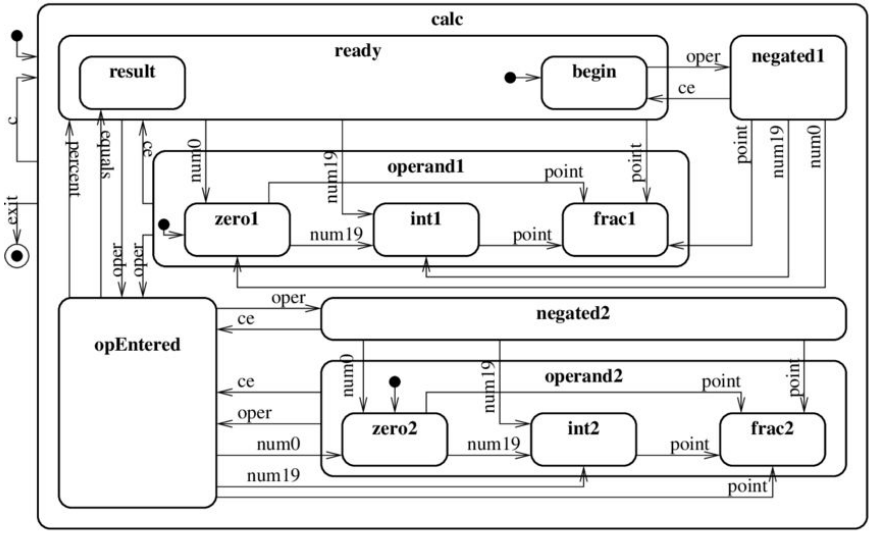
Example Modelling Formalisms


Finite State Machines

Petri Nets

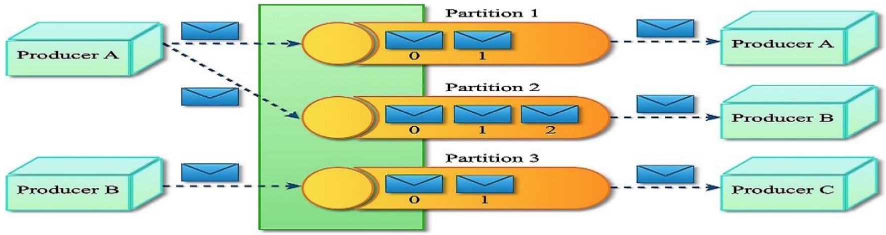


(UML) Statecharts



Messaging System:- Queue

Pub-Subscribe



Languages for MBT Models

- There are many modeling languages that can be used for MBT.

- Modeling languages are defined in the following ways:
  - By their ***concepts*** (also known as abstract syntax and often textually described, but sometimes specified in meta models).
  - By their ***syntax*** (also known as concrete syntax and often defined by grammar rules)
  - By their ***semantics*** (often defined by static and dynamic semantic rules).

- Diagrams are representations of models in a graphical modeling language, such as class diagrams, sequence diagrams or state machine diagrams, in UML.

Languages for MBT Models (cont’d)

- In order to select a good "fit for purpose” language, the tester needs to know the main characteristics that differentiate modeling languages for MBT:
  - ***Modeling the concepts:*** Modeling languages differ in the set of concepts they support.
  - Depending on the test objective, the models can represent
    - structural aspects (e.g., architectures, components, interfaces of the software),
    - data aspects (e.g., formats and semantics of passive objects of the software), or
    - behavioral aspects (e.g., scenarios, interactions, execution of the active objects of the software).

Languages for MBT Models (cont'd)

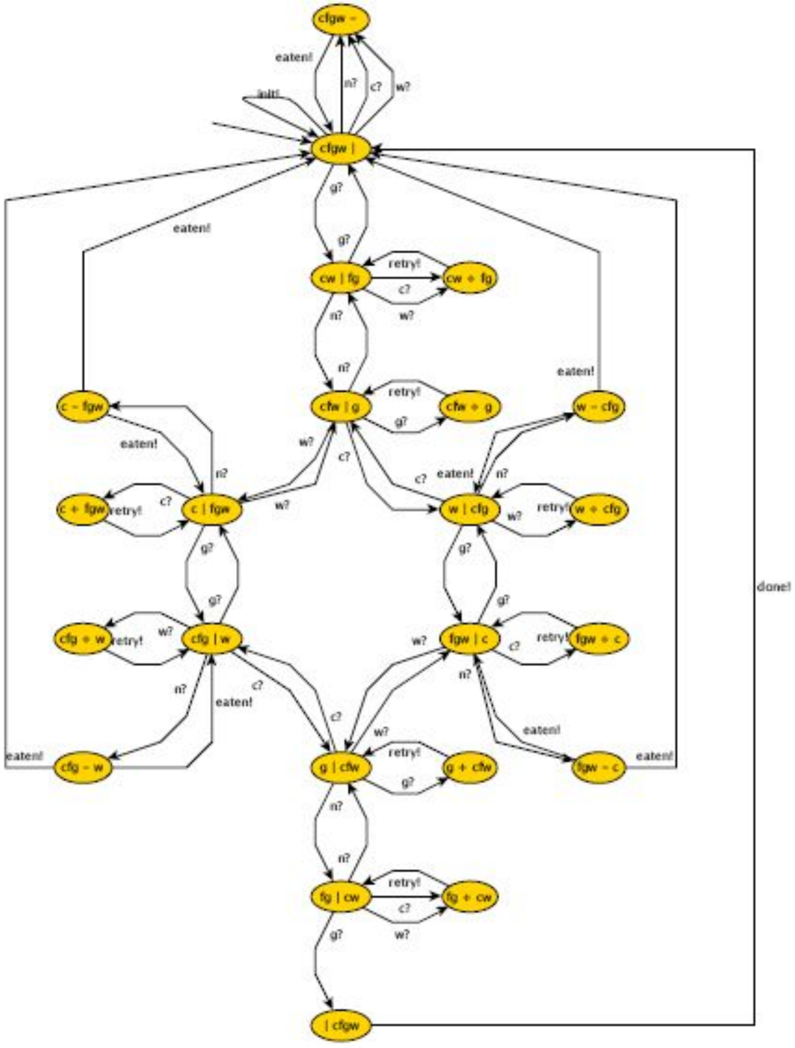
- ***Formalism***: The degree of formalism of a modeling language ranges from limited formalization to full formalization.
- While the latter is the most rigid and enables full-fledged analysis of the models, the former is often more practical.
- At a minimum, the language should have a formal syntax (e.g., including structured text or structured tables). It may also have a formally defined semantics.

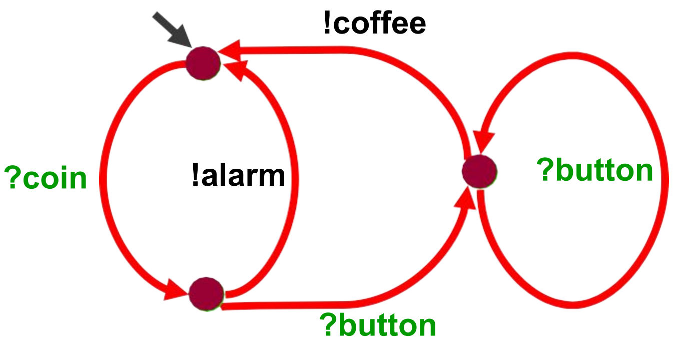
- ***Presentation formats***: Modeling languages can use different presentation formats ranging from textual to graphical formats as well as combinations of these.
- Graphical formats are often more user-friendly and easier to read, while textual formats are often more tool-friendly and efficient to both write and maintain.

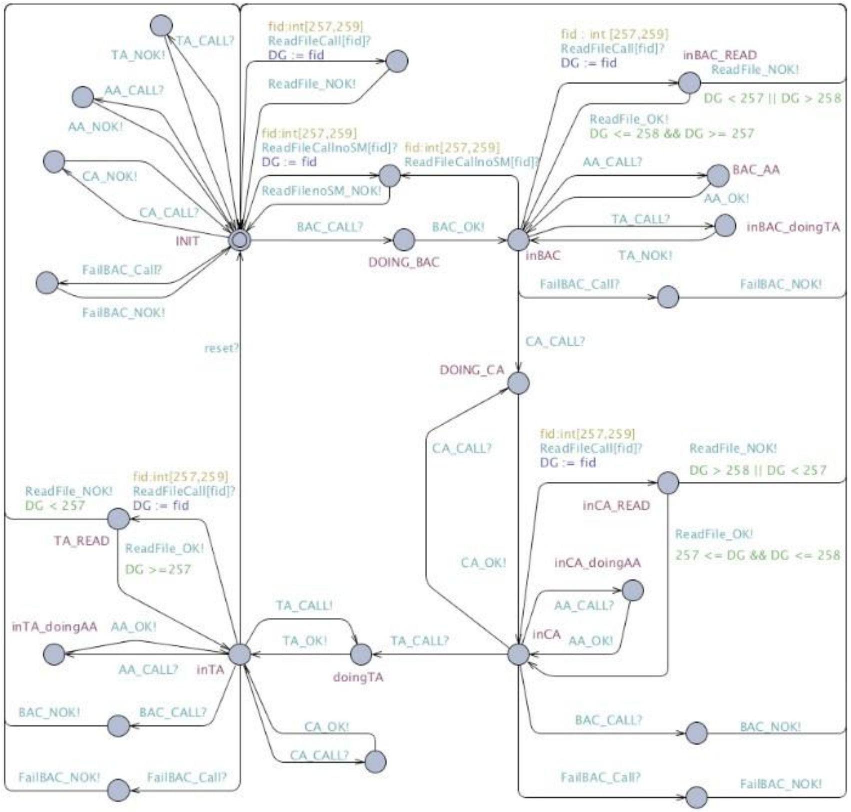
Categories of Modeling Languages

- **Languages for structural models:**
  - Such languages support the specification of structural elements of software such as interfaces, components, and hierarchies.
  - An example is the UML component diagrams.
- **Languages for data models:**
  - Such languages support the specification of the data types and values.
  - Examples include the UML class diagrams and value specifications.
- **Languages for behavioral models:**
  - Such languages support the specification of events, actions, reactions and/or interactions of software.
  - Examples include UML activity or interaction diagrams, state machines or Business Process Modeling and Notation (BPMN).
- **Integrated languages:**
  - Typically, a modeling language is not limited to one of the aspects, but rather provides concepts for a number of aspects.
  - An example is UML itself, in which the different diagrams can be used in combination to denote the different aspects of software

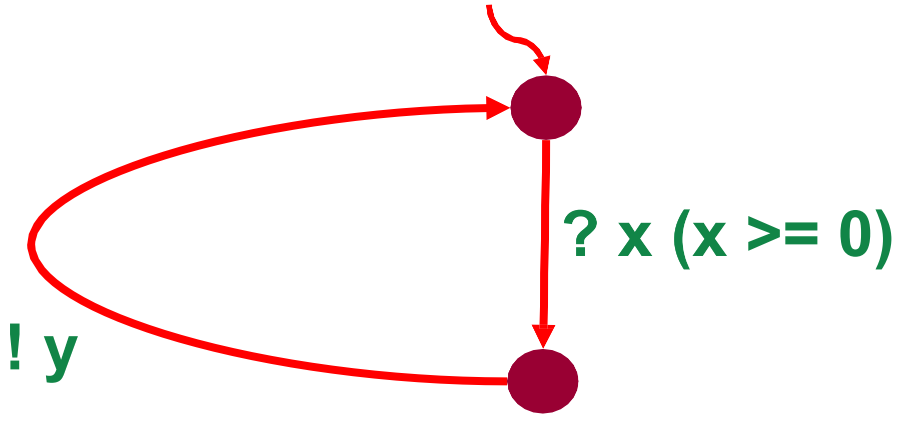
Example MBT Models







Example Model



$y \times y = x$

model of $\sqrt{x}$

- *'?' correponds to input,*
  '!' corresponds to output.
- *specification of* ***properties***

  rather than construction
- ***under-specification***
- ***non-determinism***

Selection Criteria for Test Case Generation

- ***Coverage-based test selection criteria*** relate test generation with coverage items from the MBT model.

- The coverage items may be:
  - ***Requirements linked to the MBT model*****:** This criterion requires that MBT model elements be linked to selected requirements.
  - Full requirements coverage corresponds to a set of test cases that completely covers a selected set of requirements (each requirement is covered by at least one test case).

Selection Criteria for Test Case Generation

- ***MBT model elements*****:** Model coverage is based on the internal structure of the MBT model.
- The tester, test manager or any other role involved defines coverage items and selects the set of test cases that covers a desired quantity of those items.
- Possible coverage items may be model elements such as:
  - States,
  - transitions and decisions in state diagrams (“state transition testing”)
  - Activities and gateways in business process models
  - Conditions and actions in decision tables
  - Statements and conditions in textual models

Selection Criteria for Test Case Generation

- ***Data-related test selection criteria*: These criteria relate to test design techniques such as:**
  - Equivalence partitioning
  - Boundary value analysis

- ***Other test selection criteria:*** include
  - Random,
  - Scenario-based and Pattern-based,
  - Project-driven.

Examples of Test Selection Criteria

- On activity diagrams or business process models:
  - Activity coverage (100% = coverage of each activity in the diagram).
  - Decision / gateway coverage (100% = coverage of each decision point in the diagram).
  - Path coverage (100% = coverage of all business scenarios in the diagram), with or without loops
- On state diagrams:
  - State and transition coverage (100% = coverage of each state or transition respectively).
  - Transition pair coverage (100% = coverage of each consecutive pair of transitions in the diagram).
  - Path coverage (100% = coverage of all paths, typically without loops, in the diagram).
- On decision tables:
  - Every condition
  - Every action
  - Every rule

Examples of Test Selection Criteria (cont'd)

- On data domains defined in the structural part of the model:
  - Equivalence partition coverage (e.g., with one representative per class).
  - Boundary value coverage (e.g., considering boundaries on ranges of numerical data).
  - Pairwise testing on defined domains (ensuring that all possible discrete combinations of each pair of input parameters are produced).
- On a textual model:
  - Statement coverage (100% = each executable statement is covered) Decision coverage (100% = each decision is covered).
  - Decision condition coverage (100% = all condition outcomes and all decision outcomes have been exercised by the test suite).
  - Multiple condition coverage (all combinations of conditions in each decision are covered – leads to a high number of generated test cases).

Test Execution

- MBT generated test cases may be executed manually or automatically.

- For ***manual test execution***, test cases are generated by the MBT tool.
  - These test cases must be generated in a format usable for manual test execution.
  - It may also be beneficial to be able to export the cases into a test management tool.
  - Test execution and defect management are then performed in accordance with the test process.

- For ***automated test execution***, the test cases must be generated in a form that is executable.
  - There are three main approaches that are used to manage automated execution from abstract test cases (next slide):

Test Execution (cont’d)

- **The adaptation approach** :
  - With this approach, test adaptation layer code is written to bridge the abstraction gap.
  - This code is essentially a wrapper around the test object.

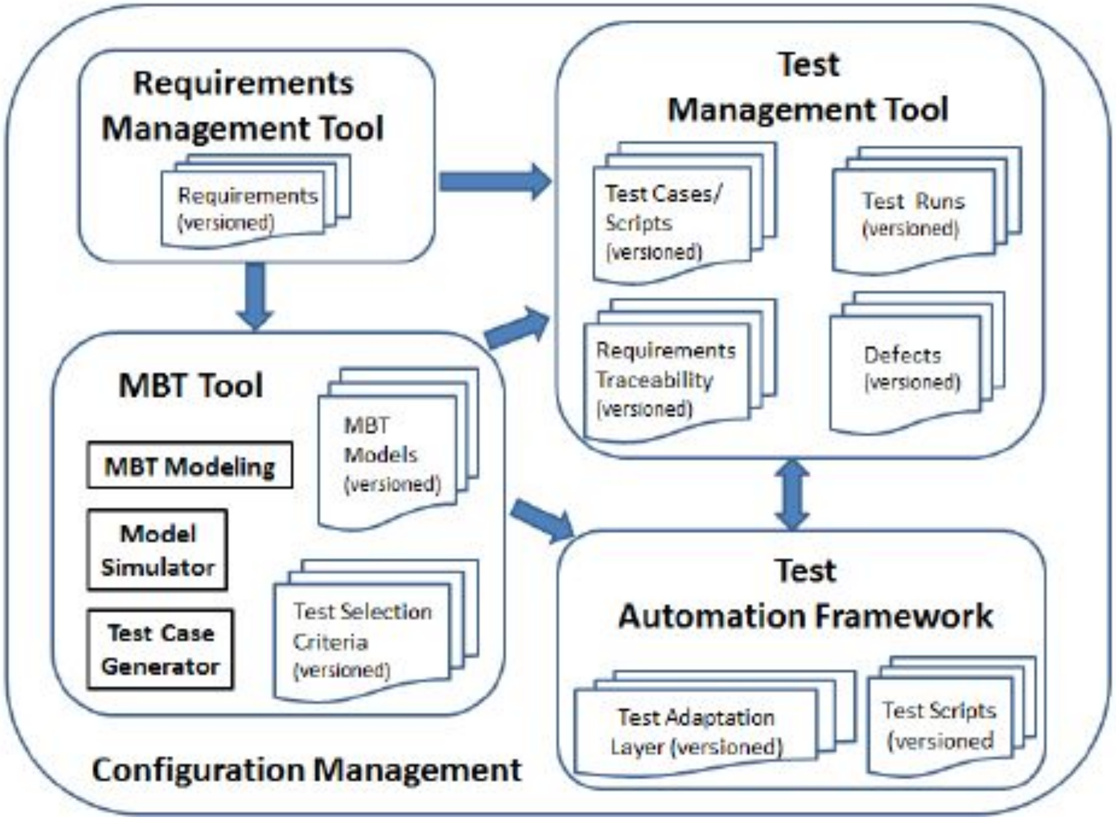
- **The transformation approach** :
  - With this approach, the MBT generated test cases are directly converted into automated test scripts (no test adaptation layer is required).

- **The mixed approach** :
  - This is a combination of the two previously mentioned approaches in which the abstract test cases are transformed into concrete test cases, which are wrapped with adaptation layer code around the test object.

Integration of MBT Tool

- Typical tool chain supports the main activities of the MBT-based fundamental test process, including:
  - Iterative MBT model development, review and validation using the modeling tool, the model simulator and the traceability with the requirements
  - Test generation by applying test selection criteria to the MBT model supported by the test case generator
  - Generated test cases and test scripts as well as traceability links that can be exported to the test management tool and the test automation framework (in case of test execution automation)
  - Configuration management for the MBT testware such as the MBT model and the adaptation layer.

Integration of MBT Tool



Software Verification, Validation and Testing; A Summary

Verification vs Validation

- **Verification**:
  "Are we building the product right”.
  - The software should conform to its specification.

- **Validation**:
  "Are we building the right product”.
  - The software should do what the user really requires.

The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.

- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.

V & V goals

- Verification and validation should establish confidence that the software is fit for purpose
- This does NOT mean completely free of defects
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed

Static and Dynamic Verification

- **Software inspections.** Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis
- **Software testing.** Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

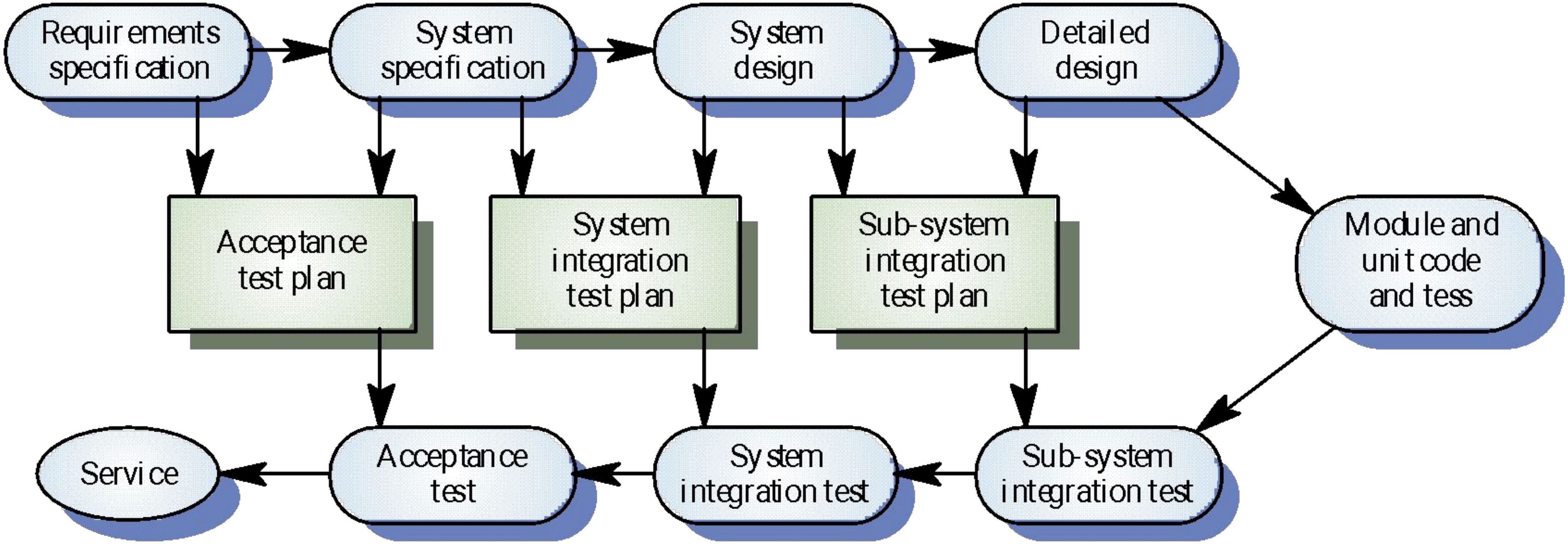
Software testing

- Can reveal the presence of errors NOT their absence
- A successful test is a test which discovers one or more errors
- The only validation technique for non-functional requirements
- Should be used in conjunction with static verification to provide full V&V coverage

Testing and debugging

- Defect testing and debugging are distinct processes
- Verification and validation is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

The V-model of development



Requirements specification

System specification

System design

Detailed design

Acceptance test plan

System integration test plan

Sub-system integration test plan

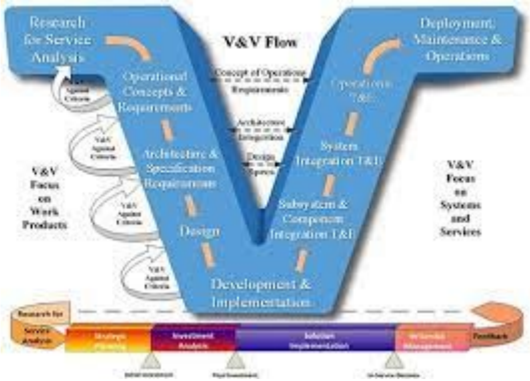
Module and unit code and tess

Service

Acceptance test

System integration test

Sub-system integration test

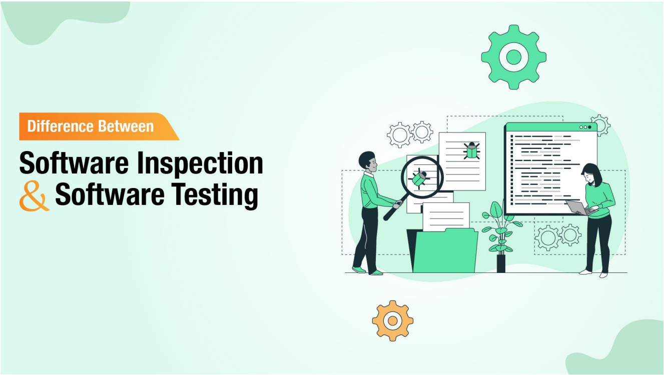

Research for Service Analysis
V&V Flow
Deployment, Maintenance & Operations
Operational Concepts & Requirements
Concept of Operations
Requirements
Operations T&E
Architecture Integration
Design Specs
Architecture & Specification Requirements
System Integration T&E
V&V Against Criteria
V&V Focus on Work Products
V&V Focus on Systems and Services
Design
Subsystem & Component Integration T&E
Development & Implementation
Research for Service Analysis
Feedback

Software Inspections

- Software Inspection involves examining the source representation with the aim of discovering anomalies and defects without execution of a system.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- Corresponds to static analysis of the software and codes.

Inspections and Testing

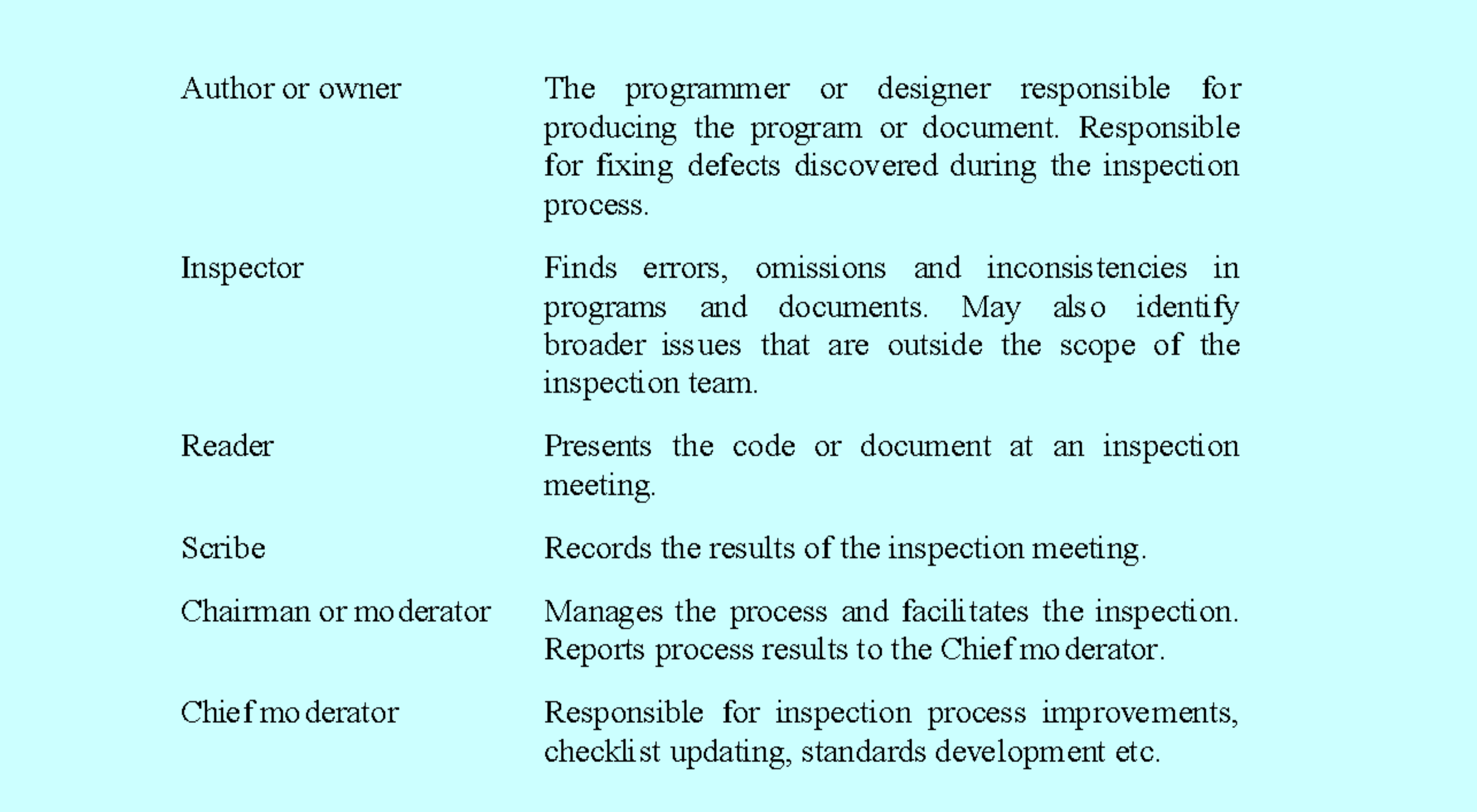
- Inspections and testing are complementary and not opposing verification techniques.
- Inspections can check conformance with a specification but not conformance with the customer’s real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.
- Management should not use inspections for staff appraisal i.e. finding out who makes mistakes.



Inspection Procedure

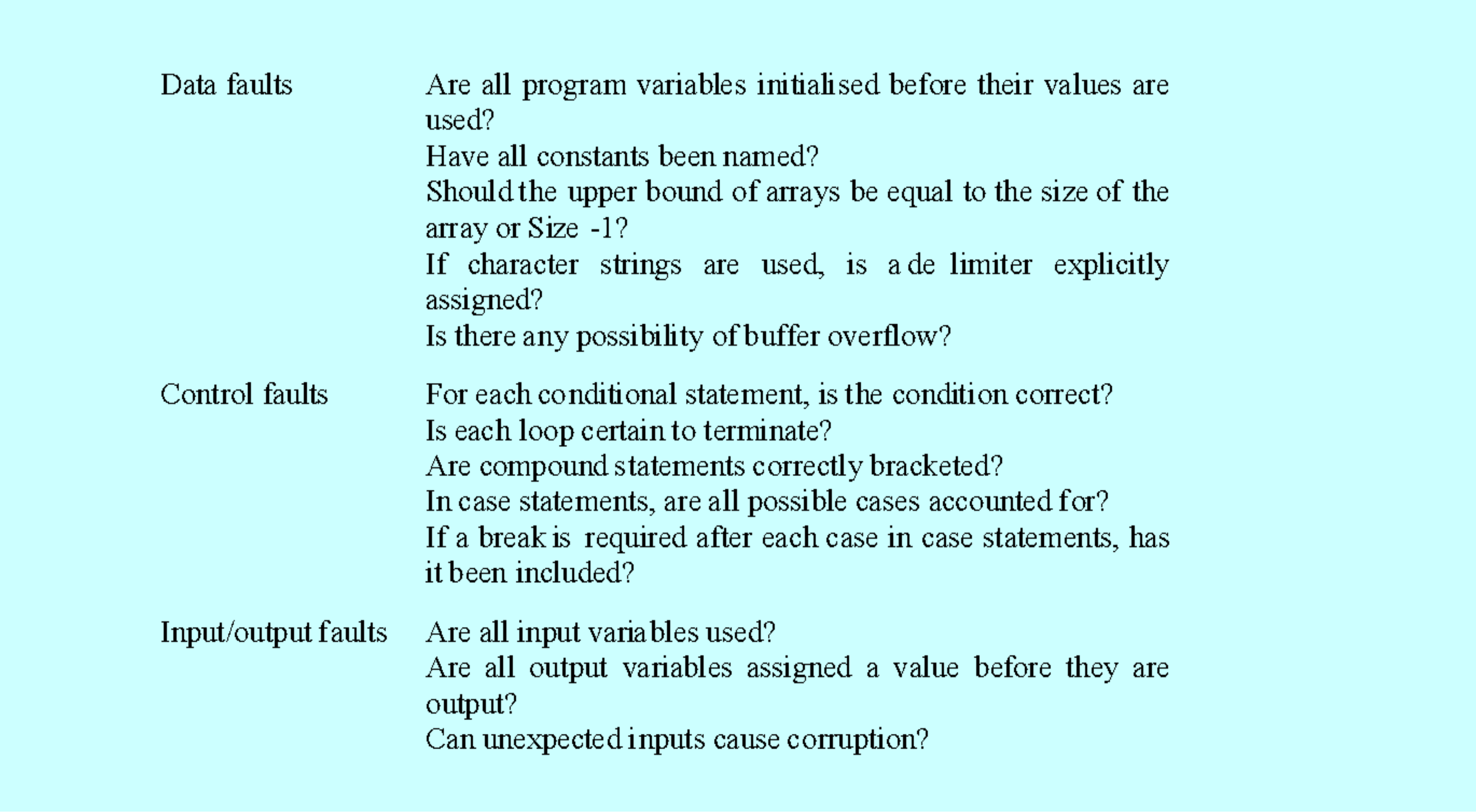
- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair errors.
- Re-inspection may or may not be required.
- Checklist of common errors should be used to drive the inspection. Examples: Initialization, Constant naming, loop termination, array bounds…

Inspection Roles



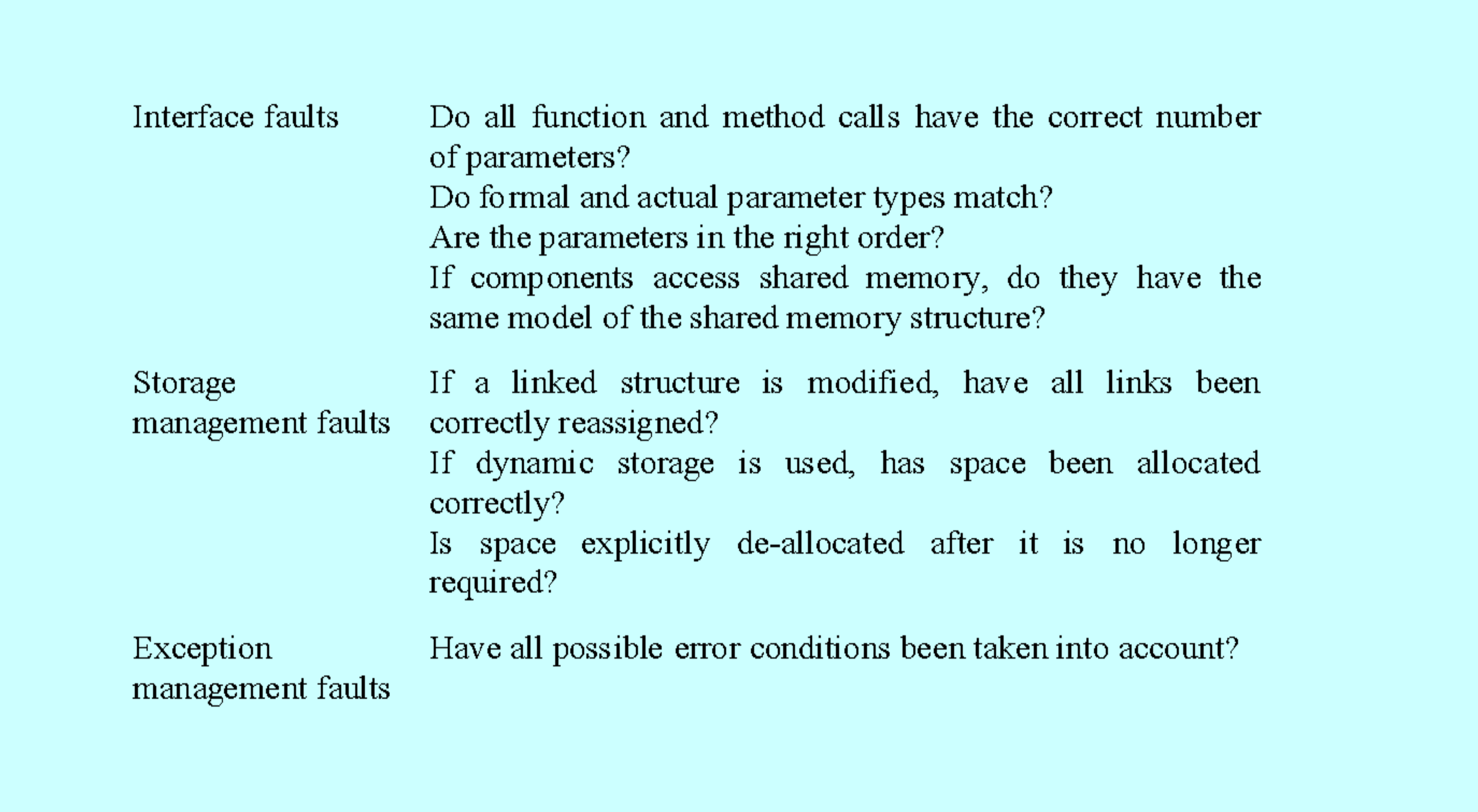
Author or owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting.
Scribe	Records the results of the inspection meeting.
Chairman or moderator	Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
Chief moderator	Responsible for inspection process improvements, checklist updating, standards development etc.

Inspection Checks



Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned? Is there any possibility of buffer overflow?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? If a break is required after each case in case statements, has it been included?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output? Can unexpected inputs cause corruption?

Inspection Checks (cont'd)



Interface faults	Do all function and method calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Stages of Static Analysis

- *Control flow analysis.* Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- *Data flow analysis.* Detects uninitialised variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- *Interface analysis.* Checks the consistency of routine and procedure declarations and their use

Stages of Static Analysis (cont'd)

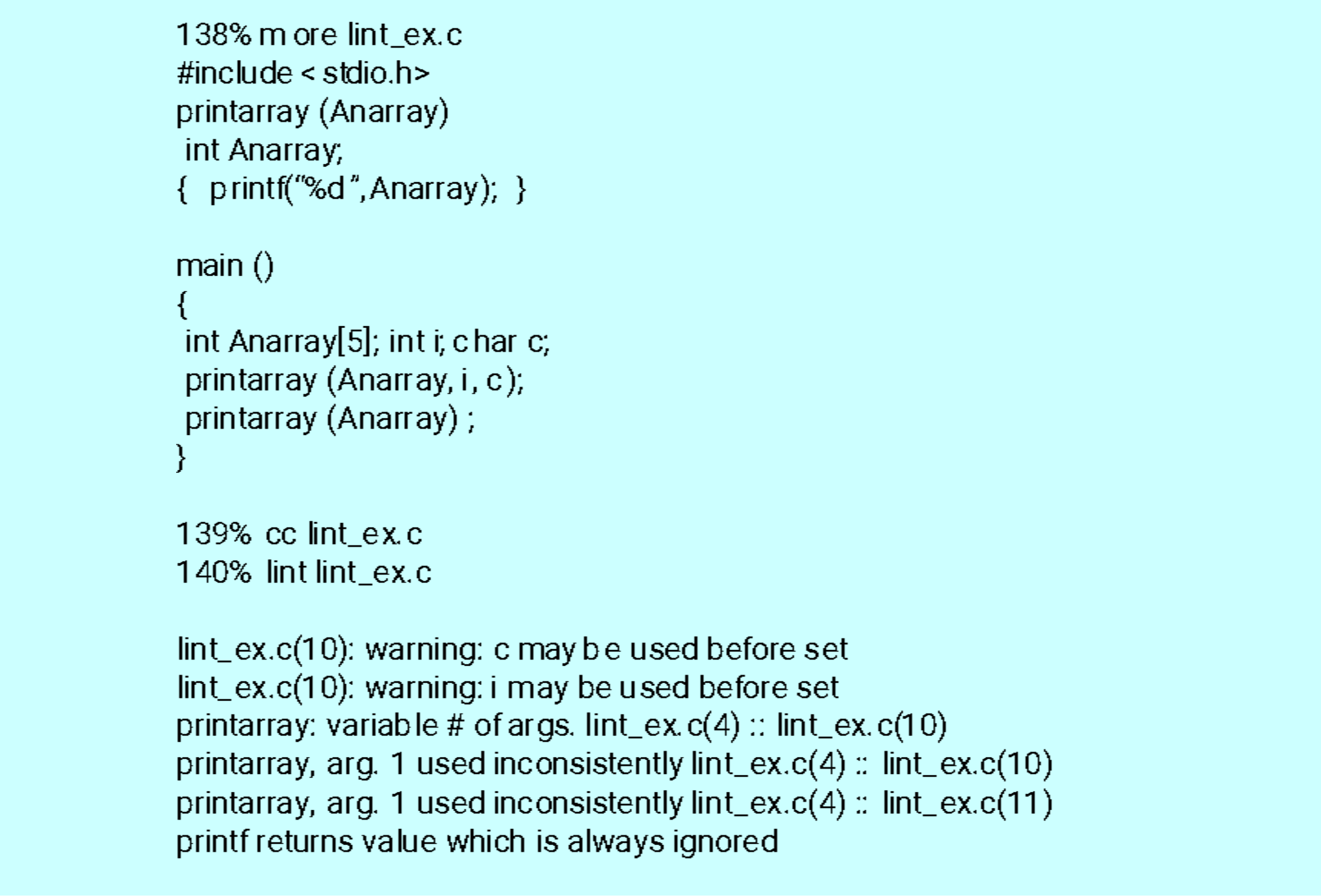
- *Information flow analysis.* Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- *Path analysis.* Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. Must be used with care.

Automated Static Analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

LINT Static Analysis Example

Lint: is an automated source code checker for programmatic and stylistic errors. A tool known as a linter, analyzes C code without executing it.



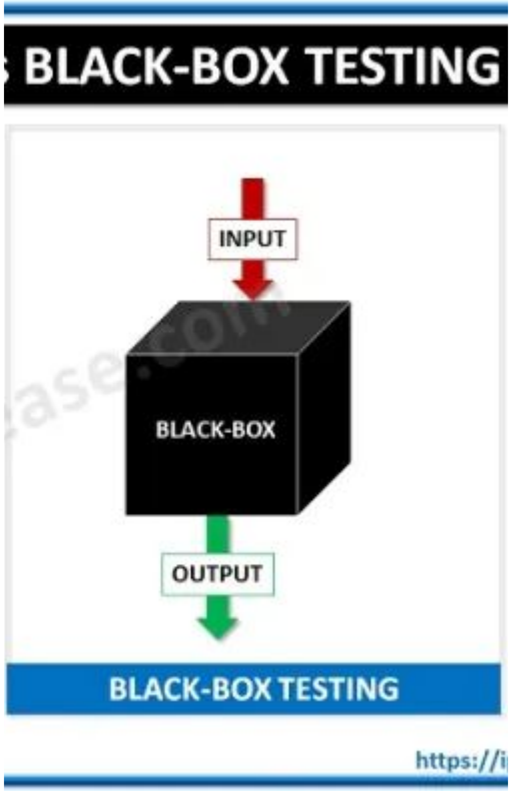
```
138% more lint_ex.c
#include < stdio.h>
printarray (Anarray)
 int Anarray;
{  printf(“%d”,Anarray);  }

main ()
{
 int Anarray[5]; int i; char c;
 printarray (Anarray, i, c);
 printarray (Anarray) ;
}

139%  cc lint_ex.c
140%  lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::  lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::  lint_ex.c(11)
printf returns value which is always ignored
```

Use of static analysis

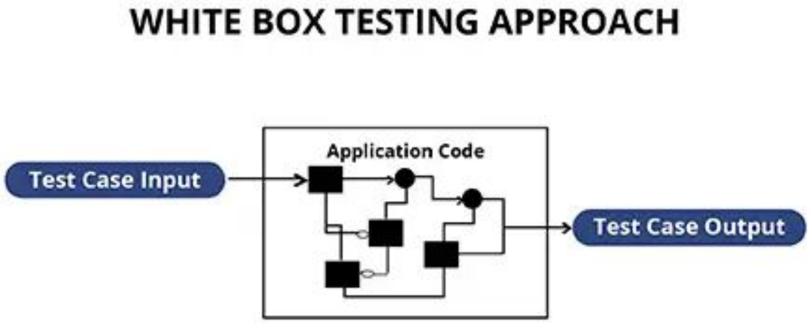
- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler
- Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation

Black-Box Testing

- Also known as Behavioral Testing.
- Involves testing a system with no prior knowledge of its internal workings.
- Testing process is carried out by simulating the software with proper input, and observing the outputs.
- Objective is makes it possible to identify how the system responds to expected and unexpected user actions, its response time, usability issues and reliability issues.



White-Box Testing



- Sometime called structural testing.
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases.
- Objective is to exercise all program statements (not all path combinations).

Component Testing

- Component or unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Components may be:
  - Individual functions or methods within an object;
  - Object classes with several attributes and methods;
  - Composite components with defined interfaces used to access their functionality.

Object Class Testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object;
  - Setting and interrogating all object attributes;
  - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

Interface Testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Particularly important for object-oriented development as objects are defined by their interfaces.

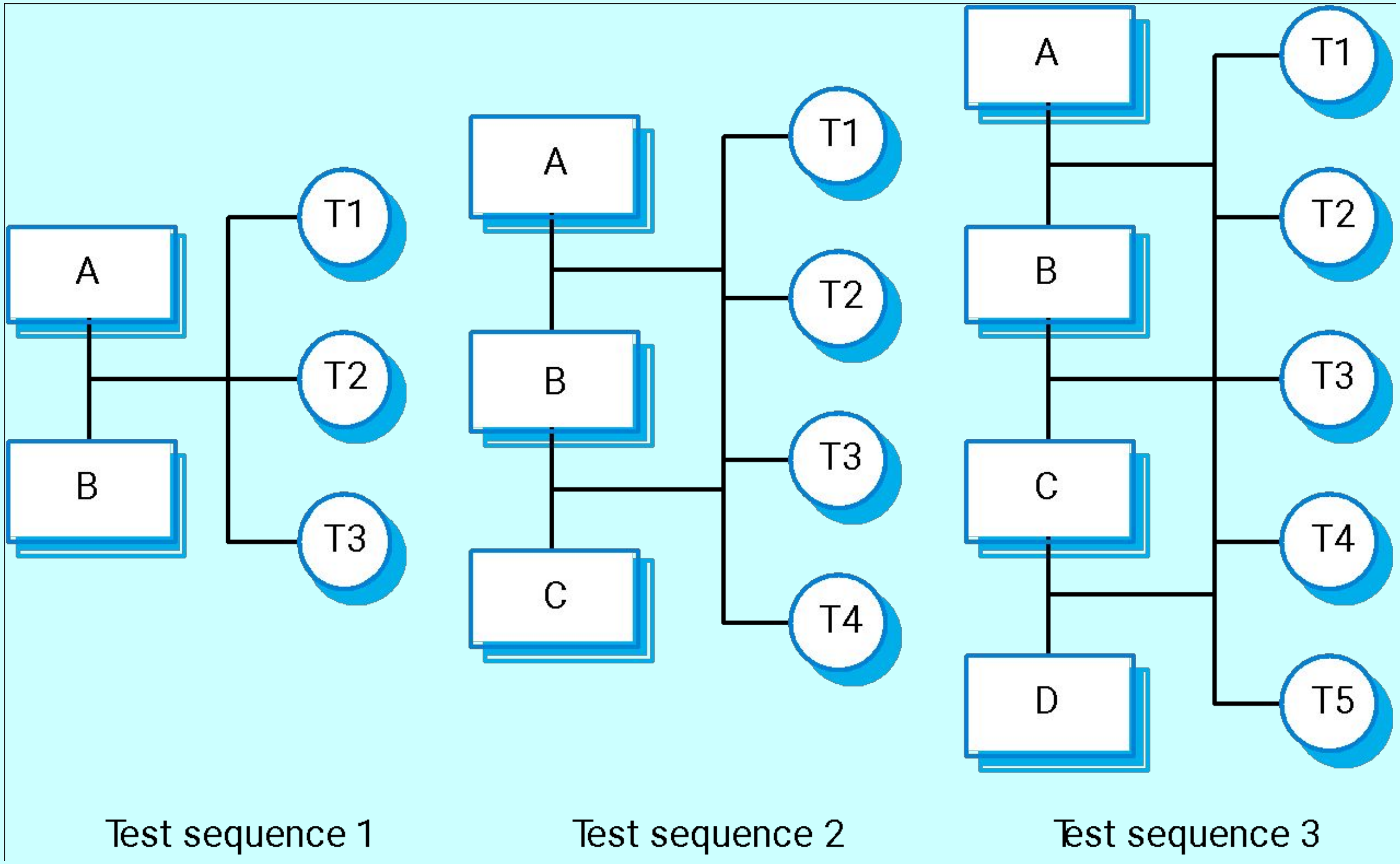
Interface Types

- Parameter interfaces
  - Data passed from one procedure to another.
- Shared memory interfaces
  - Block of memory is shared between procedures or functions.
- Procedural interfaces
  - Sub-system encapsulates a set of procedures to be called by other sub-systems.
- Message passing interfaces
  - Sub-systems request services from other sub-system.

System Testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
  - Integration testing - the test team have access to the system source code. The system is tested as components are integrated.
  - Release testing - the test team test the complete system to be delivered as a black-box.

Incremental Integration Testing



Release Testing

- The process of testing a release of a system that will be distributed to customers.
- Primary goal is to increase the supplier’s confidence that the system meets its requirements.
- Release testing is usually black-box or functional testing
  - Based on the system specification only;
  - Testers do not have knowledge of the system implementation.

Stress Testing

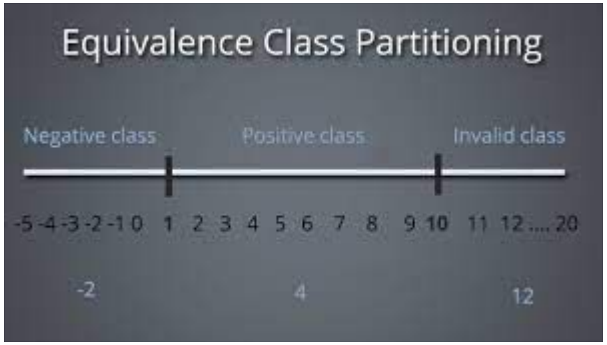
- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.
- There is a slight difference between stress testing and performance testing
    - Stress testing pushes the system beyond its dimensioned limits, while performance testing aims to the system behavior when it is working within the border of its limits, but very close to them.

Test Case Design

- Involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Design approaches:
  - Requirements-based testing (i.e. trace test cases to the requirements)
  - Partition based testing,
  - Boundary value testing,
  - Path based testing.

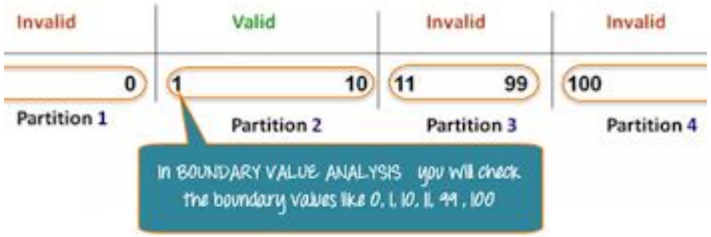
Partition Testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.



Boundary Value Testing

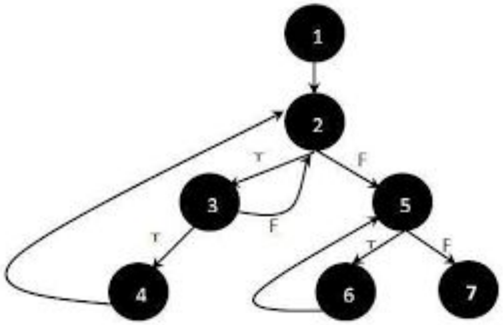
- Based on testing the boundary values of valid and invalid partitions.
- The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition,
- So boundaries are an area where testing is likely to yield defects.



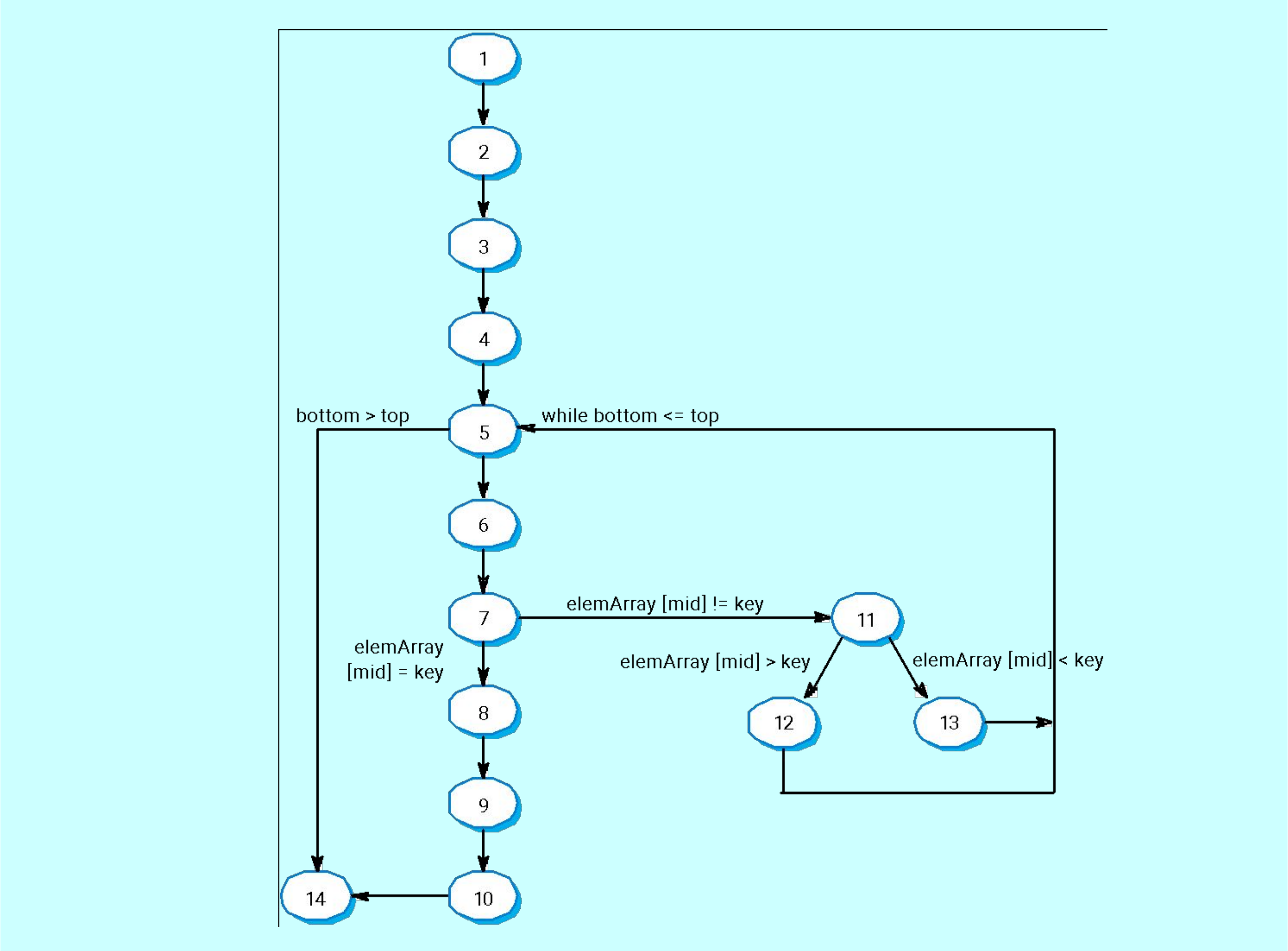
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99 , 100

Path Testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.



Binary Search Flow Graph



Independent Paths

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …
- Test cases should be derived so that all of these distinct paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Test Automation

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as ***Junit*** support the automatic execution of tests.
- Most testing workbenches are open systems because testing needs are organisation-specific.

Unit Testing in Java Using
the JUnit Framework
The JUnit Framework can be easily integrated with Eclipse.
Unit testing accelerates programming speed.
JAVA

Junit as a Test Automation Framework

- JUnit is a software testing framework that helps developers test their applications.
- It allows developers to write tests in Java and run them on the Java platform.
- JUnit also has a built-in reporter that can print out the results of the tests.
- JUnit provides several features that make it easy to create and run tests, including:
  - **Assertions:** Assertions are used to verify the expected behavior of a system. JUnit provides a set of assertion methods that can be used to check the results of a test.
  - **Test runners:** Test runners are used for executing the tests and reporting the results. JUnit provides a graphical test runner that can run tests and view the results.
  - **Test suites:** Test suites are used to group related tests. JUnit provides a way to create test suites that can be run together.
  - **Reporting:** When you run your tests, JUnit can help you analyze the results. It provides a built-in reporter that prints out information about the executed tests.

JUnit - Basic Usage

https://www.tutorialspoint.com/junit/index.htm

```
/*
* This class prints the given message on
console.
*/

public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed

   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
      return message;
   }
}
```

A Class whose behavior to be tested

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {

   String message = "Hello World";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      assertEquals(message,messageUtil.printMessage());
   }
}
```

A Test Class in **Junit** for testing the class under test.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result =
JUnitCore.runClasses(TestJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

A Test Runner Class to run the above defined test case.