SOFTWARE ENGINEERING

Spring 2024

BASICS OF TESTING - DEFINITION

- Definition
  - Software testing is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect free in order to produce the quality product.
- Testing types
  - **Manual Testing**

    Manual testing is the process of testing where software manually find the defects. Tester should have the perspective of end users and to ensure all the features are working as mentioned in the requirement document. In this process, testers execute the test cases and generate the reports manually without using any automation tools.

  - **Automation Testing**

    Automation testing is the process of testing the software using an automation tool to find the defects. In this process, testers execute the test scripts and generate the test results automatically by using automation tools.

SOME DEFINITIONS

- Test
  - A process of finding errors through the occurrences of faults
  - Another definition: the act of exercising test cases
- Test case
  - Description of a set of inputs along with expected outputs
- Test data
  - Values that instantiate a test case
- Test Suite
  - A collection of test cases targeted to test a set of properties such as user interface functionalities, reliability, performance and so on.

SOME DEFINITIONS (CONTINUED)

- SUT
  - System (or Software) Under Test
- Test Harness
  - Software along with a set of executable test cases to test a SUT. This is part of an automated testing process.
- Testing and debugging
  - Testing is a process of finding the presence of an error
    - May be performed by several people including the programmer
  - Debugging is a process of finding the source of an error, if it is present
    - Mostly performed by the programmer

PROGRAM TESTING

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using artificial data.
- You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- Can reveal the presence of errors NOT their absence.
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

PROGRAM TESTING GOALS

- To demonstrate to the developer and the customer that the software meets its requirements.
  - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

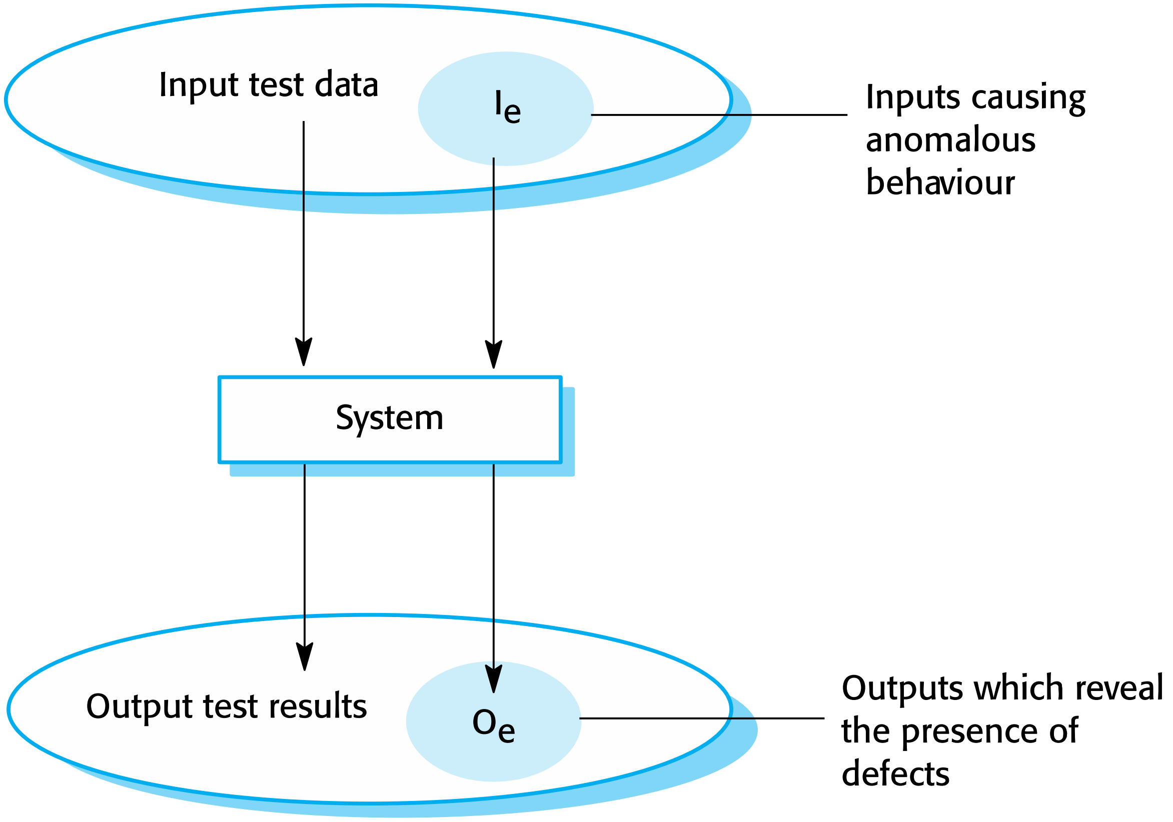
VALIDATION AND DEFECT TESTING

- The first goal leads to validation testing
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- The second goal leads to defect testing
  - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

TESTING PROCESS GOALS

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.
- Defect testing
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

AN INPUT-OUTPUT MODEL OF PROGRAM TESTING


Input test data
I_e
Inputs causing anomalous behaviour
System
Output test results
O_e
Outputs which reveal the presence of defects

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.

V & V CONFIDENCE

- Aim of V & V is to establish confidence that the system is ‘fit for purpose’.
- Depends on system’s purpose, user expectations and marketing environment
  - Software purpose
    - The level of confidence depends on how critical the software is to an organisation.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

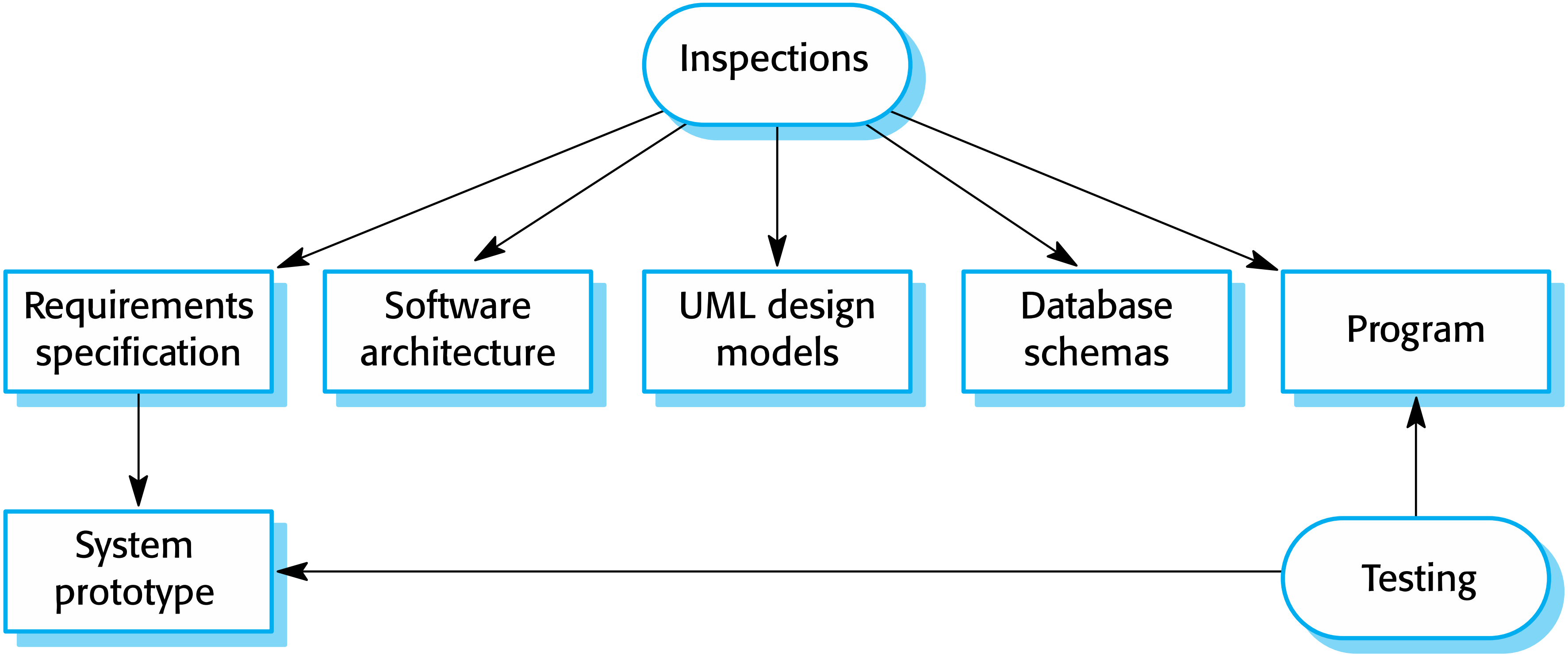
TESTING METHODS

- Static Testing
  - It is also known as **Verification in Software Testing**. Verification is a static method of checking documents and files. Verification is the process, to ensure that whether we are building the product right i.e., to verify the requirements which we have and to verify whether we are developing the product accordingly or not.
- Dynamic Testing
  - It is also known as **Validation** in **Software Testing**. Validation is a dynamic process of testing the real product. Validation is the process, whether we are building the right product i.e., to validate the product which we have developed is right or not.

INSPECTIONS AND TESTING

- 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.

INSPECTIONS AND TESTING



SOFTWARE INSPECTIONS

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design,configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

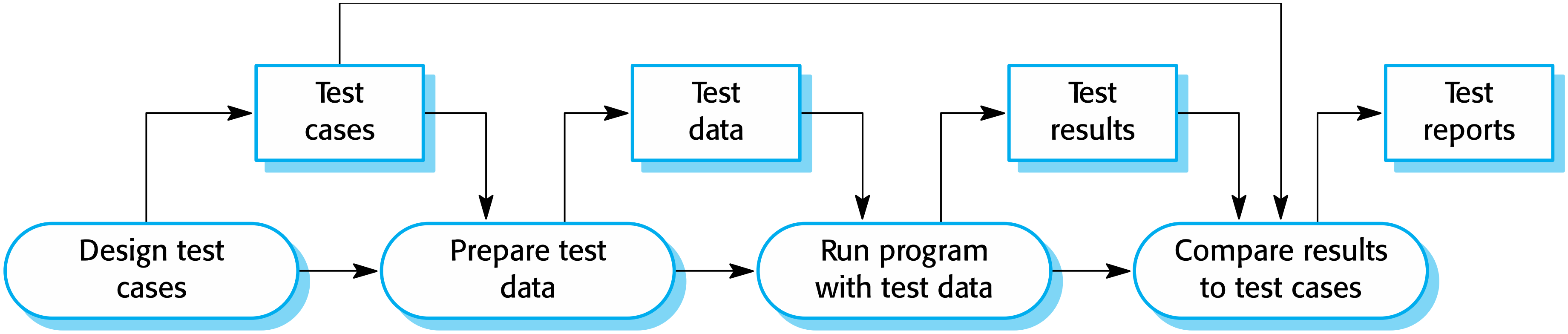
ADVANTAGES OF INSPECTIONS

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

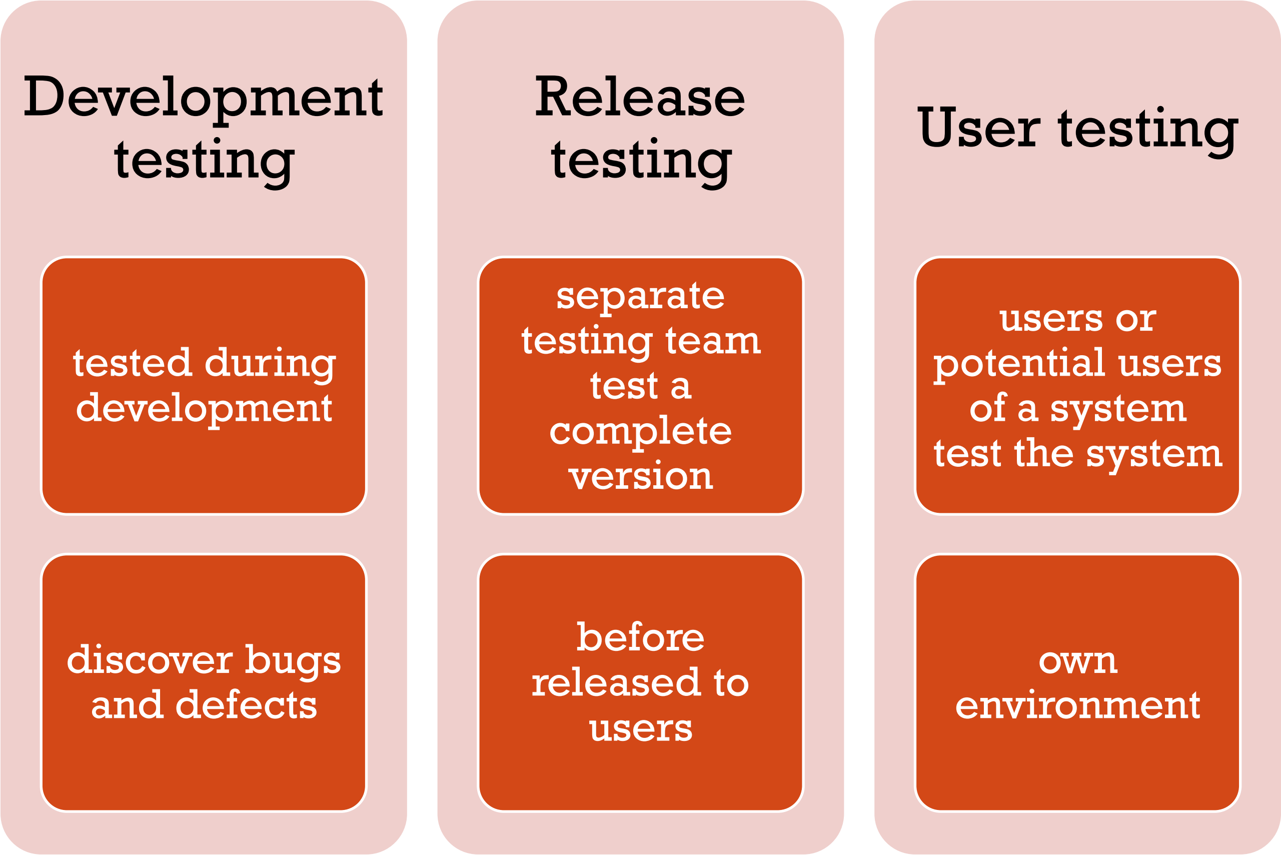
INSPECTIONS AND TESTING

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- 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.

A MODEL OF THE SOFTWARE TESTING PROCESS



STAGES OF TESTING



Development testing

- tested during development
- discover bugs and defects

Release testing

- separate testing team test a complete version
- before released to users

User testing

- users or potential users of a system test the system
- own environment

DEVELOPMENT TESTING

DEVELOPMENT TESTING

Unit testing

- individual program units or object classes are tested.
- focus on testing the functionality of objects or methods.

Component testing

- several individual units are integrated to create composite components.
- Component testing should focus on testing component interfaces.

System testing

- some or all of the components in a system are integrated and the system is tested as a whole
- should focus on testing component interactions.

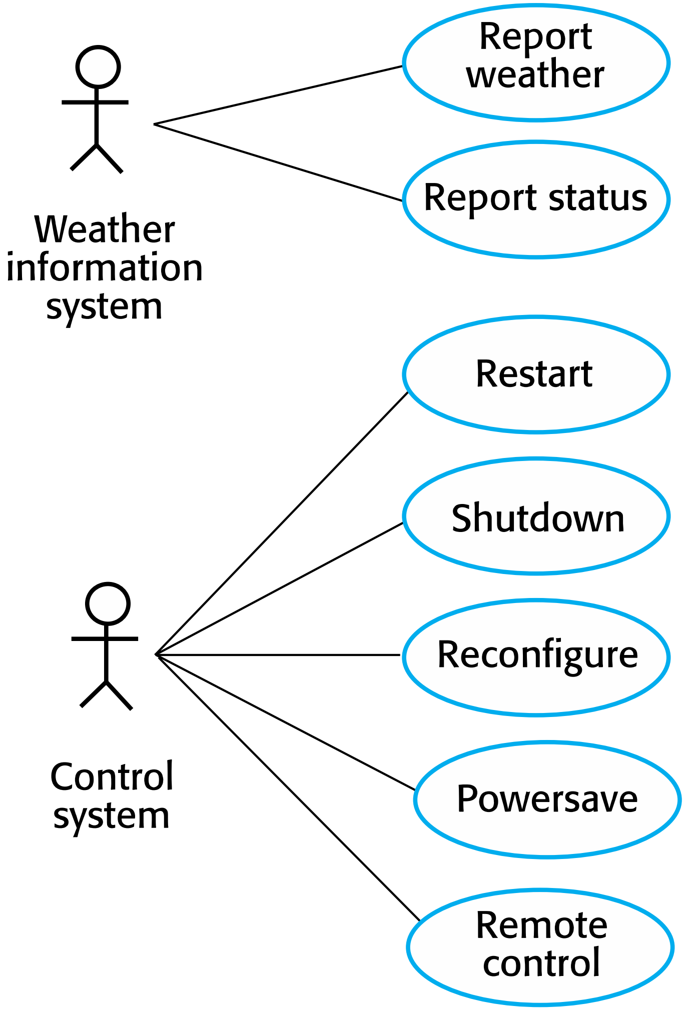
UNIT TESTING

- Unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Units 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.

THE WEATHER STATION OBJECT INTERFACE



WeatherStation

identifier

reportWeather ( )
reportStatus ( )
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

CHOOSING UNIT TEST CASES

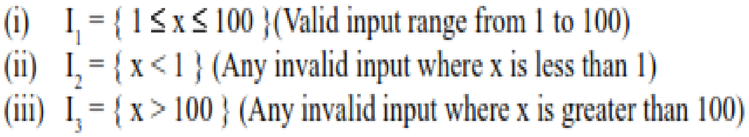
- The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- If there are defects in the component, these should be revealed by test cases.
- This leads to 2 types of unit test case:
  - The first of these should reflect normal operation of a program and should show that the component works as expected.
  - The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

TESTING STRATEGIES

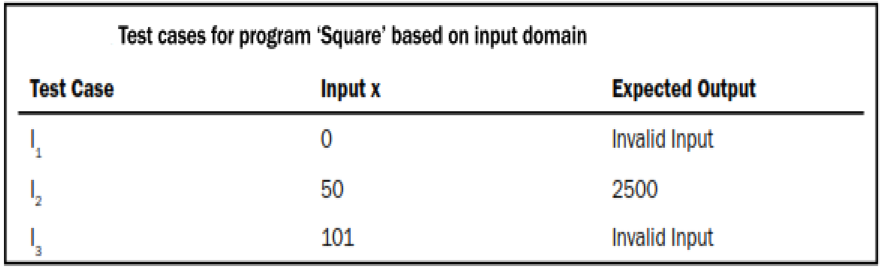
- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - You should choose tests from within each of these groups.
- Guideline-based testing, where you use testing guidelines to choose test cases.
  - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

CREATION OF EQUIVALENCE CLASSES

- The entire input domain can be divided into at least two equivalence classes: one containing all valid inputs and the other containing all invalid inputs.
- Each equivalence class can further be sub-divided into equivalence classes on which the program is required to behave differently.
- Square program(Write a program 'Square' which takes 'x' as an input and prints the square of 'x' as output. The range of 'x' is from 1 to 100)



(i) $I_1 = \{ 1 \leq x \leq 100 \}$(Valid input range from 1 to 100)
(ii) $I_2 = \{ x < 1 \}$ (Any invalid input where x is less than 1)
(iii) $I_3 = \{ x > 100 \}$ (Any invalid input where x is greater than 100)



Test cases for program 'Square' based on input domain

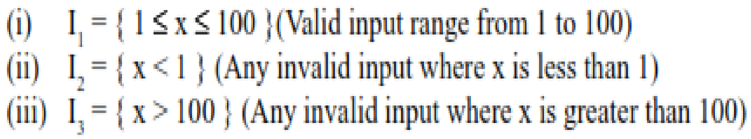
Test Case	Input x	Expected Output
I_1	0	Invalid Input
I_2	50	2500
I_3	101	Invalid Input

EQUIVALENCE CLASS TESTING

- A large number of test cases are generated for any program. It is neither feasible nor desirable to execute all such test cases.
- Select a few test cases and still wish to achieve a reasonable level of coverage.
- Divide input domain into various categories with some relationship and expect that every test case from a category exhibits the same behavior.
- If categories are well selected, we may assume that if one representative test case works correctly, others may also give the same results. This assumption allows us to select exactly one test case from each category and if there are four categories, four test cases may be selected.
- Each category is called an equivalence class and this type of testing is known as equivalence class testing.

CREATION OF EQUIVALENCE CLASSES

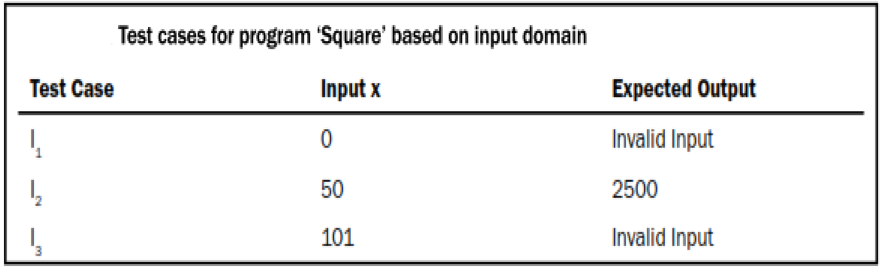
- The entire input domain can be divided into at least two equivalence classes: one containing all valid inputs and the other containing all invalid inputs.
- Each equivalence class can further be sub-divided into equivalence classes on which the program is required to behave differently.
- Square program



(i) $I_1 = \{ 1 \leq x \leq 100 \}$(Valid input range from 1 to 100)

(ii) $I_2 = \{ x < 1 \}$ (Any invalid input where x is less than 1)

(iii) $I_3 = \{ x > 100 \}$ (Any invalid input where x is greater than 100)



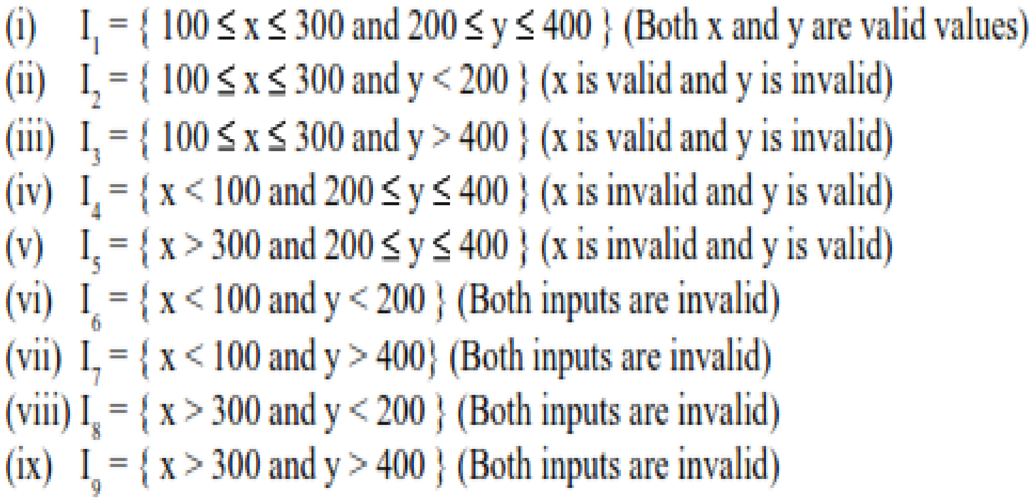
Test cases for program 'Square' based on input domain

Test Case	Input x	Expected Output
I_1	0	Invalid Input
I_2	50	2500
I_3	101	Invalid Input

EQUIVALENCE CLASSES FOR ADDITION PROGRAM

Consider a program ‘Addition’ with two input values x and y and it gives the addition of x and y as an output. The range of both input values are given as:

100<= x <= 300
200 <= y <= 400



(i) $I_1 = \{ 100 \leq x \leq 300 \text{ and } 200 \leq y \leq 400 \}$ (Both x and y are valid values)

(ii) $I_2 = \{ 100 \leq x \leq 300 \text{ and } y < 200 \}$ (x is valid and y is invalid)

(iii) $I_3 = \{ 100 \leq x \leq 300 \text{ and } y > 400 \}$ (x is valid and y is invalid)

(iv) $I_4 = \{ x < 100 \text{ and } 200 \leq y \leq 400 \}$ (x is invalid and y is valid)

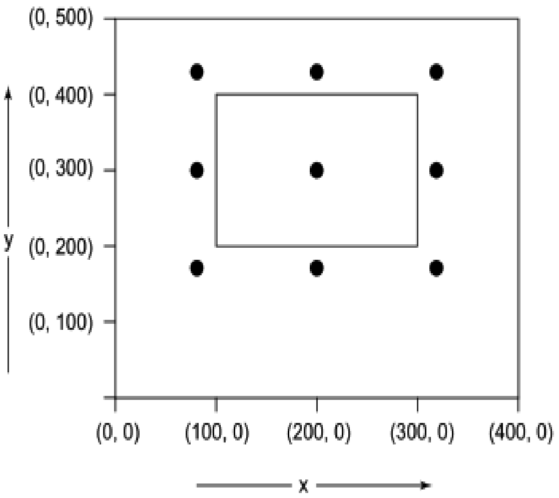
(v) $I_5 = \{ x > 300 \text{ and } 200 \leq y \leq 400 \}$ (x is invalid and y is valid)

(vi) $I_6 = \{ x < 100 \text{ and } y < 200 \}$ (Both inputs are invalid)

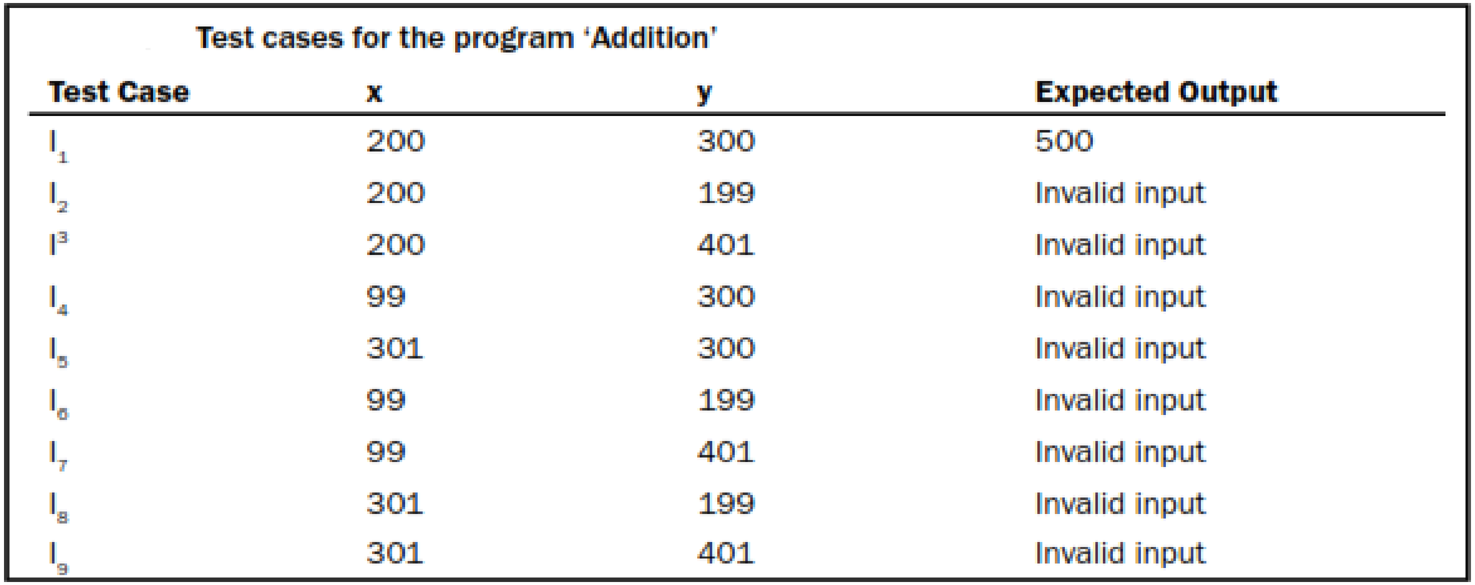
(vii) $I_7 = \{ x < 100 \text{ and } y > 400 \}$ (Both inputs are invalid)

(viii) $I_8 = \{ x > 300 \text{ and } y < 200 \}$ (Both inputs are invalid)

(ix) $I_9 = \{ x > 300 \text{ and } y > 400 \}$ (Both inputs are invalid)



TEST CASES FOR ADDITION



Test cases for the program 'Addition'

Test Case	x	y	Expected Output
I_1	200	300	500
I_2	200	199	Invalid input
I^3	200	401	Invalid input
I_4	99	300	Invalid input
I_5	301	300	Invalid input
I_6	99	199	Invalid input
I_7	99	401	Invalid input
I_8	301	199	Invalid input
I_9	301	401	Invalid input

APPLICABILITY

- Applicable at unit, integration, system and acceptance test levels.
- The basic requirement is that inputs or outputs must be partitioned based on the requirements and every partition will give a test case.
- If one test case catches a bug, the other probably will too. If one test case does not find a bug, the other test cases of the same equivalence class may also not find any bug.
- The design of equivalence classes is subjective and two testing persons may design two different sets of partitions of input and output domains.

EXAMPLE

- Consider a program for the determination of the largest amongst three numbers. Its input is a triple of positive integers (say x, y and z) and values are from interval [1, 300].
- Design the equivalence classes

AUTOMATED TEST COMPONENTS

- A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
- A call part, where you call the object or method to be tested.
- An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

TESTING GUIDELINES (SEQUENCES)

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.

GENERAL TESTING GUIDELINES

- Choose inputs that force the system to generate all error messages
- Design inputs that cause input buffers to overflow
- Repeat the same input or series of inputs numerous times
- Force invalid outputs to be generated
- Force computation results to be too large or too small.

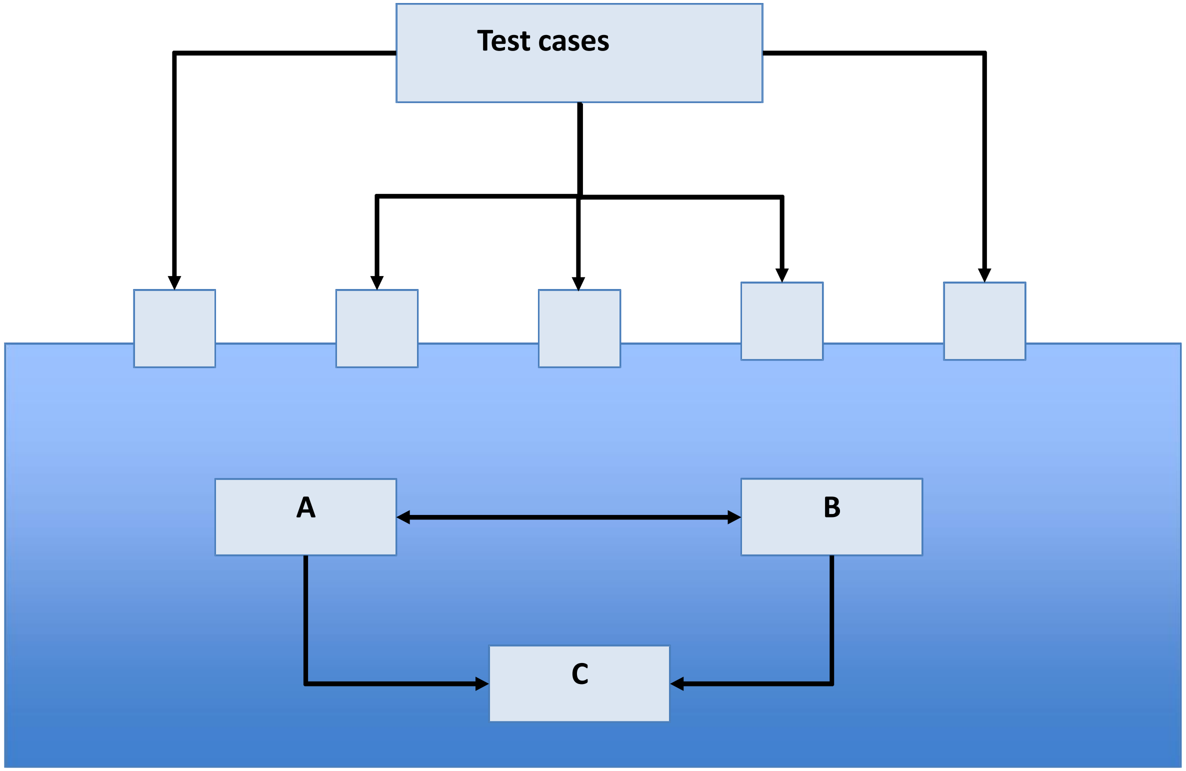
COMPONENT TESTING

- Software components are often composite components that are made up of several interacting objects.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
    - You can assume that unit tests on the individual objects within the component have been completed.

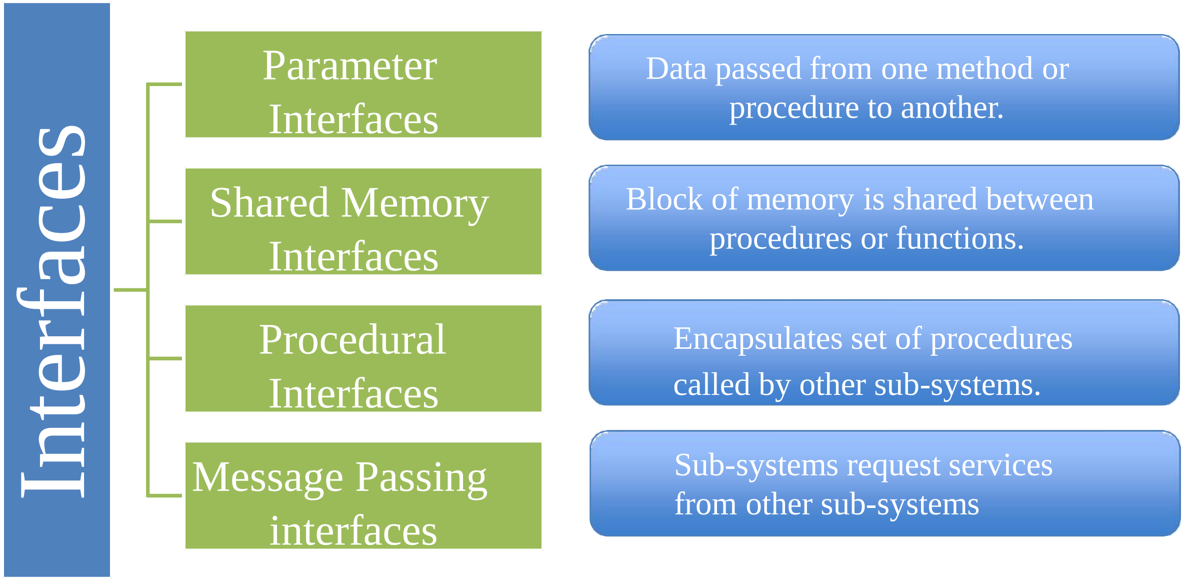
COMPONENT TESTING

- Software component
  - Composite components of several interacting objects.
- Access the functionality
  - Through defined component interface.
- Testing composite components
  - Focus on showing that the component interface behaves according to its specification.

INTERFACE TESTING



INTERFACE TESTING



Interfaces

Parameter Interfaces

Data passed from one method or procedure to another.

Shared Memory Interfaces

Block of memory is shared between procedures or functions.

Procedural Interfaces

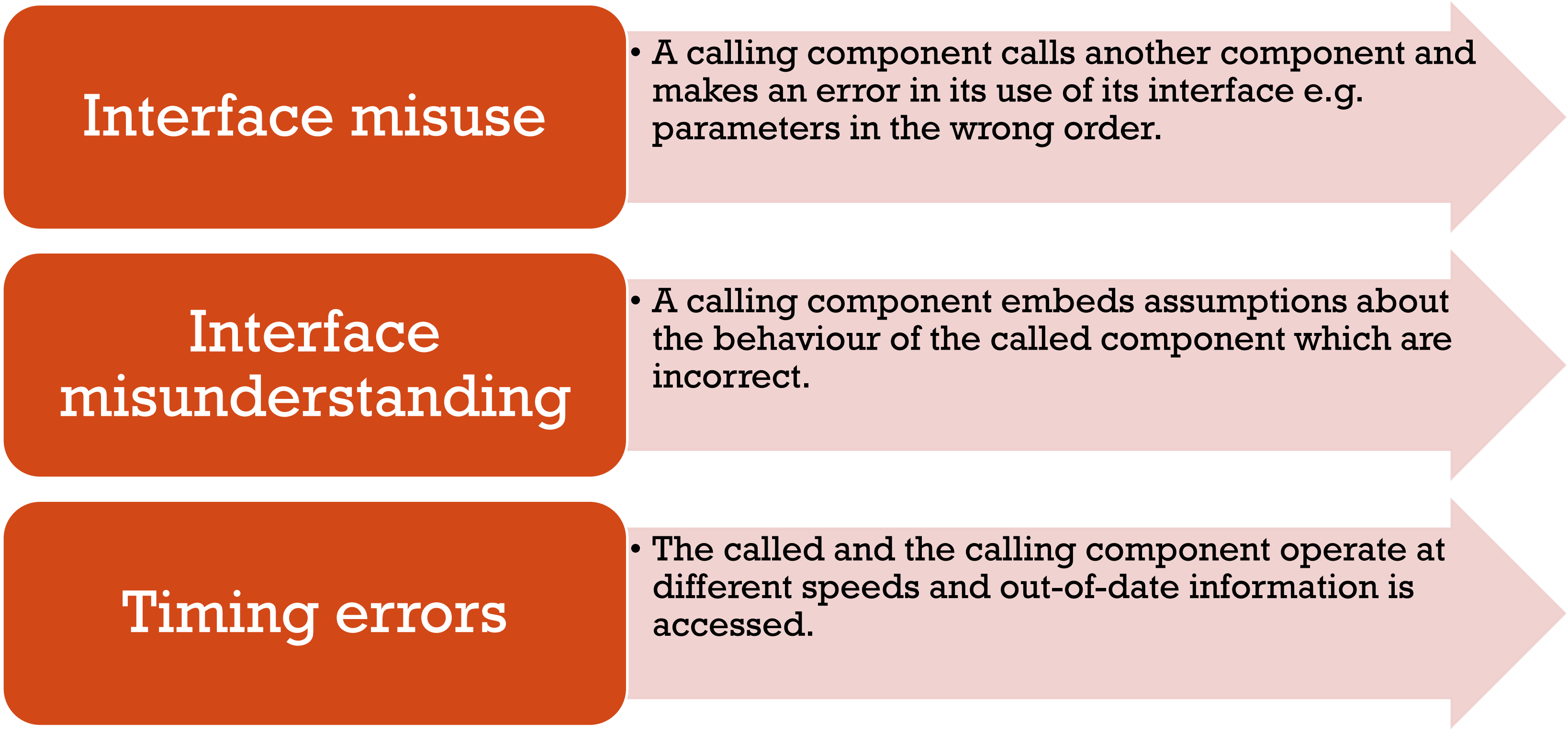
Encapsulates set of procedures called by other sub-systems.

Message Passing interfaces

Sub-systems request services from other sub-systems

Whenever there is a clear relationship between modules, we go for the integration testing. And the main purpose of the integration testing level is to expose the faults at the time of interaction between integrated components or units.

INTERFACE ERRORS



Interface misuse

- A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

Timing errors

- The called and the calling component operate at different speeds and out-of-date information is accessed.

INTERFACE TESTING GUIDELINES

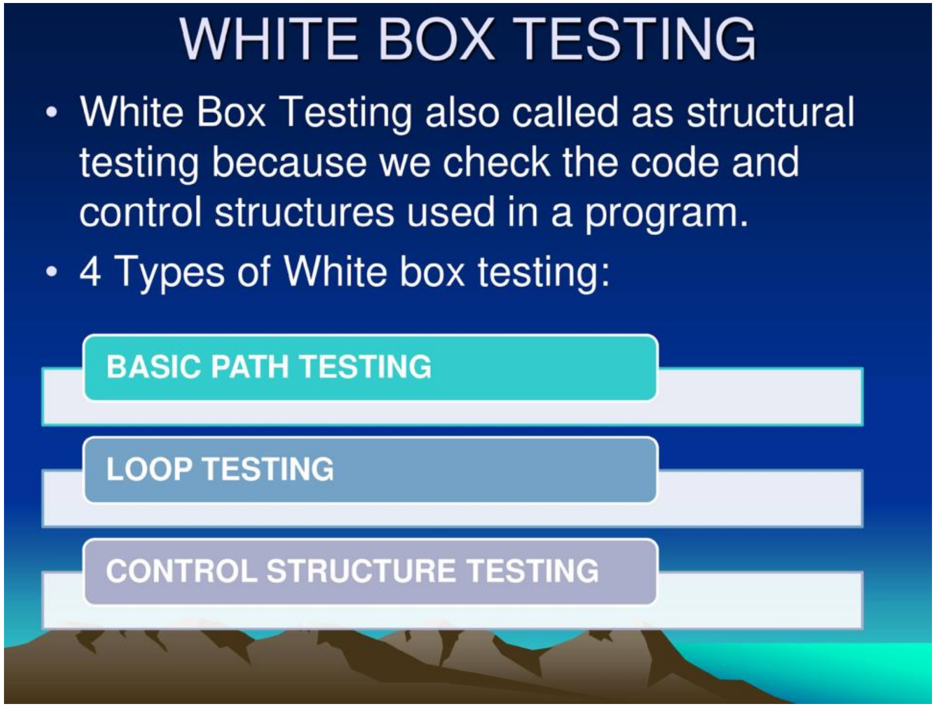
- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

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.



WHITE BOX TESTING

- White Box Testing also called as structural testing because we check the code and control structures used in a program.
- 4 Types of White box testing:

BASIC PATH TESTING

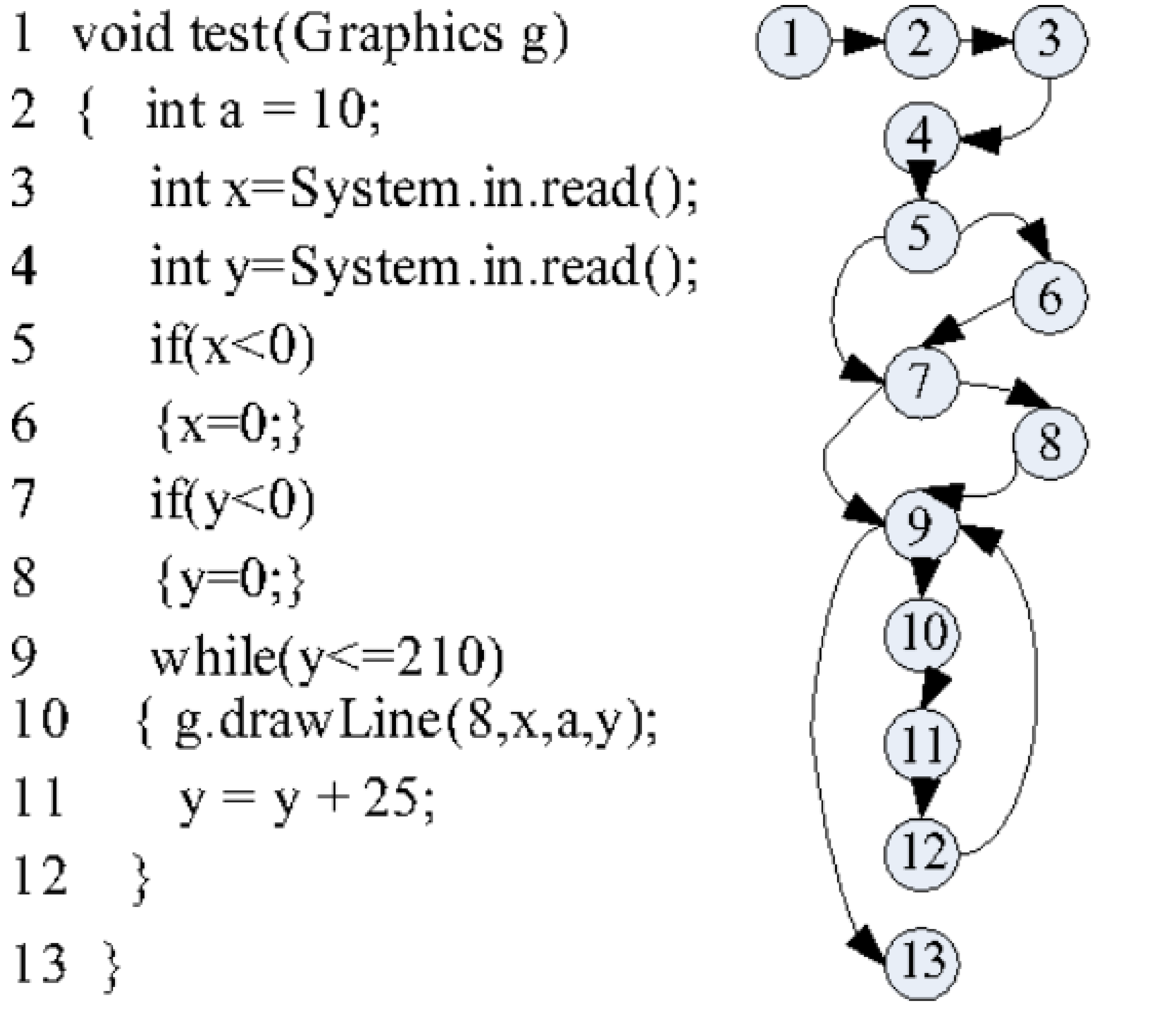
LOOP TESTING

CONTROL STRUCTURE TESTING

PATH TESTING

- The basic steps involved in basis path testing include
  - Draw a control graph (to determine different program paths)
  - Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
  - Find a basis set of paths
  - Generate test cases to exercise each path

Example-01



```
1  void test(Graphics g)
2  {   int a = 10;
3      int x=System.in.read();
4      int y=System.in.read();
5      if(x<0)
6       {x=0;}
7      if(y<0)
8       {y=0;}
9      while(y<=210)
10   { g.drawLine(8,x,a,y);
11       y = y + 25;
12   }
13 }
```

(a) source codes

(b) control flow graph

Exercise

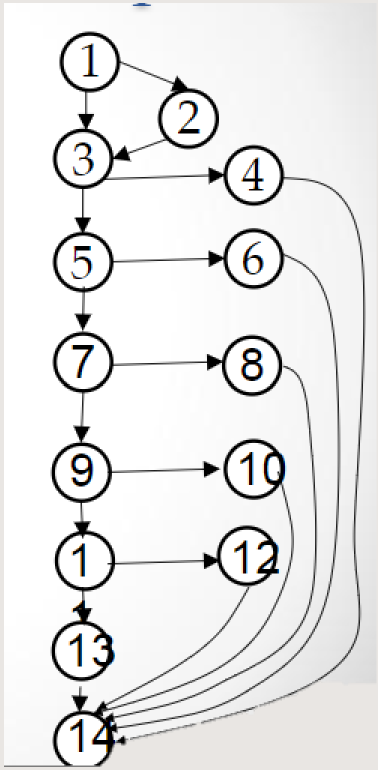
```
 public double calculate(int amount)
    {
-1-  double rushCharge = 0;
-1-  if (nextday.equals("yes") )
     {
-2-      rushCharge = 14.50;
     }
-3-  double tax = amount * .0725;
-3-  if (amount >= 1000)
     {
-4-      shipcharge = amount * .06 + rushCharge;
     }
-5-  else if (amount >= 200)
     {
-6-      shipcharge = amount * .08 + rushCharge;
     }
-7-  else if (amount >= 100)
     {
-8-      shipcharge = 13.25 + rushCharge;
     }
-9-  else if (amount >= 50)
     {
-10-     shipcharge = 9.95 + rushCharge;
     }
-11- else if (amount >= 25)
     {
-12-     shipcharge = 7.25 + rushCharge;
     }
     else
     {
-13-     shipcharge = 5.25 + rushCharge;
     }
-14- total = amount + tax + shipcharge;
-14- return total;
     } //end calculate
```

Basic Path Testing Example

Step 01:

```
 public double calculate(int amount)
   {
-1-  double rushCharge = 0;
-1-  if (nextday.equals("yes") )
    {
-2-      rushCharge = 14.50;
    }
-3-  double tax = amount * .0725;
-3-  if (amount >= 1000)
    {
-4-      shipcharge = amount * .06 + rushCharge;
    }
-5-  else if (amount >= 200)
    {
-6-      shipcharge = amount * .08 + rushCharge;
    }
-7-  else if (amount >= 100)
    {
-8-      shipcharge = 13.25 + rushCharge;
    }
-9-  else if (amount >= 50)
    {
-10-     shipcharge = 9.95 + rushCharge;
    }
-11- else if (amount >= 25)
    {
-12-     shipcharge = 7.25 + rushCharge;
    }
    else
    {
-13-     shipcharge = 5.25 + rushCharge;
    }
-14- total = amount + tax + shipcharge;
-14- return total;
    } //end calculate
```

1, 2, 3, 5,7,9,11,13,14



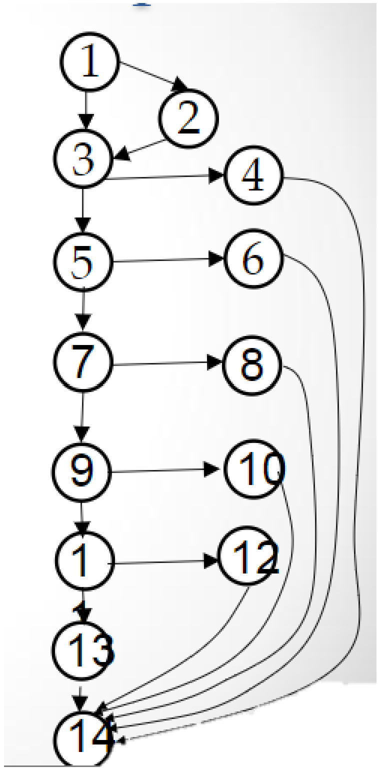
Basic Path Testing Example

- **Step 02:** Determine the Cyclomatic Complexity of the flow graph.
  - V = E-N+2
  - V = 19-14+2
  - V = 7

E-N+2

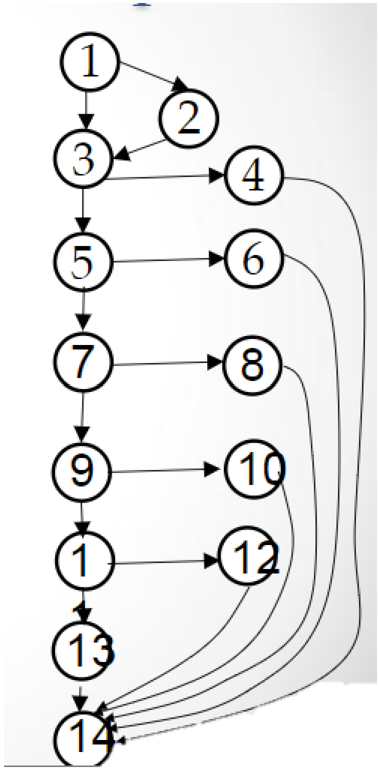
No of edges

No of Nodes

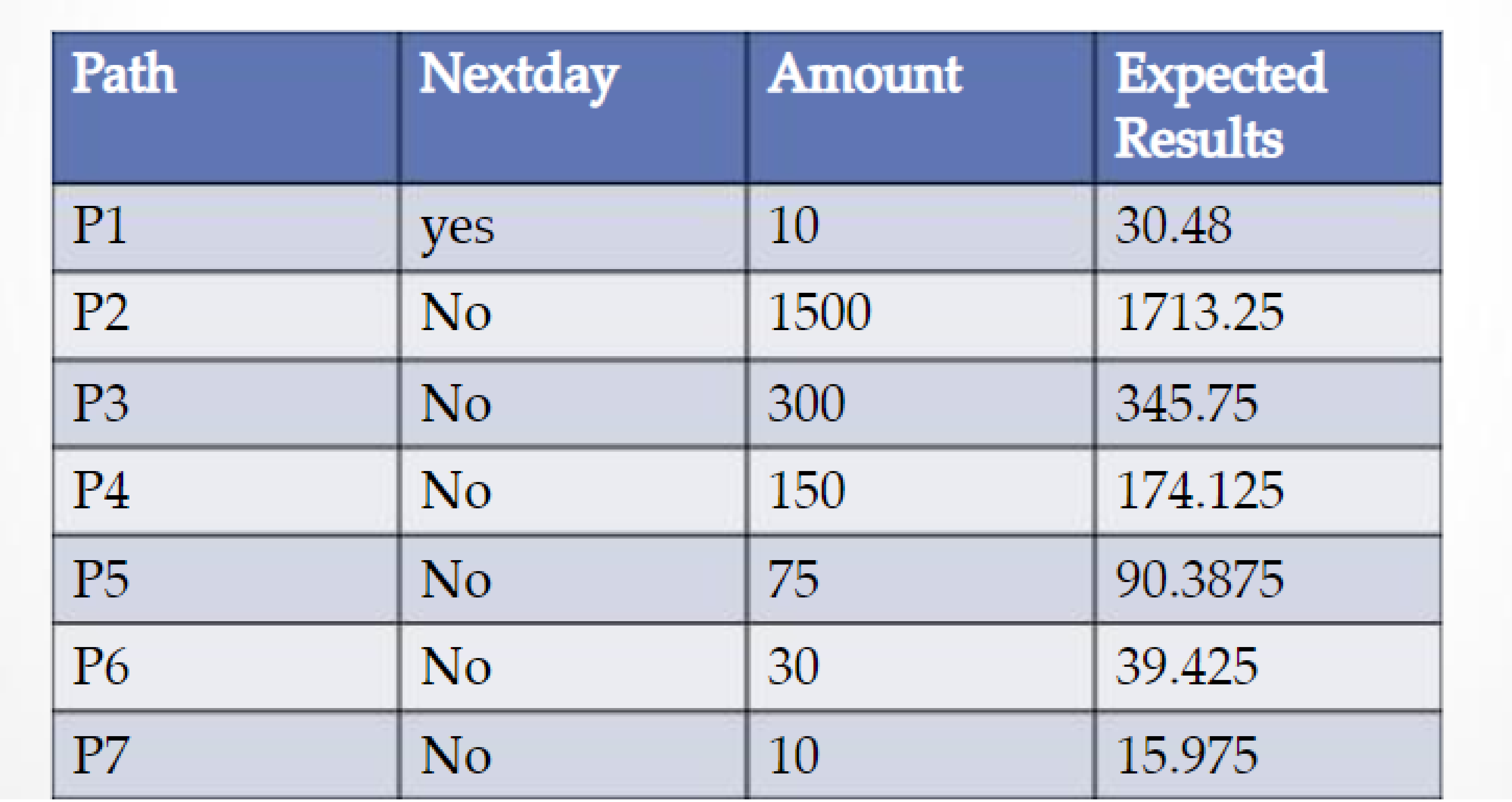


Step: 3 Determine the basis set of independent paths

- P-01: 1, 2, 3, 5,7,9,11,13,14
- P-02: 1,3,4,14
- P-03: 1,3,5,6,14
- P-04: 1,3,5,7,8,14
- P-05: 1,3,5,7,9,10,14
- P-06: 1,3,5,7,9,11,12,14
- P-07: 1,3,5,7,9,11,13,14

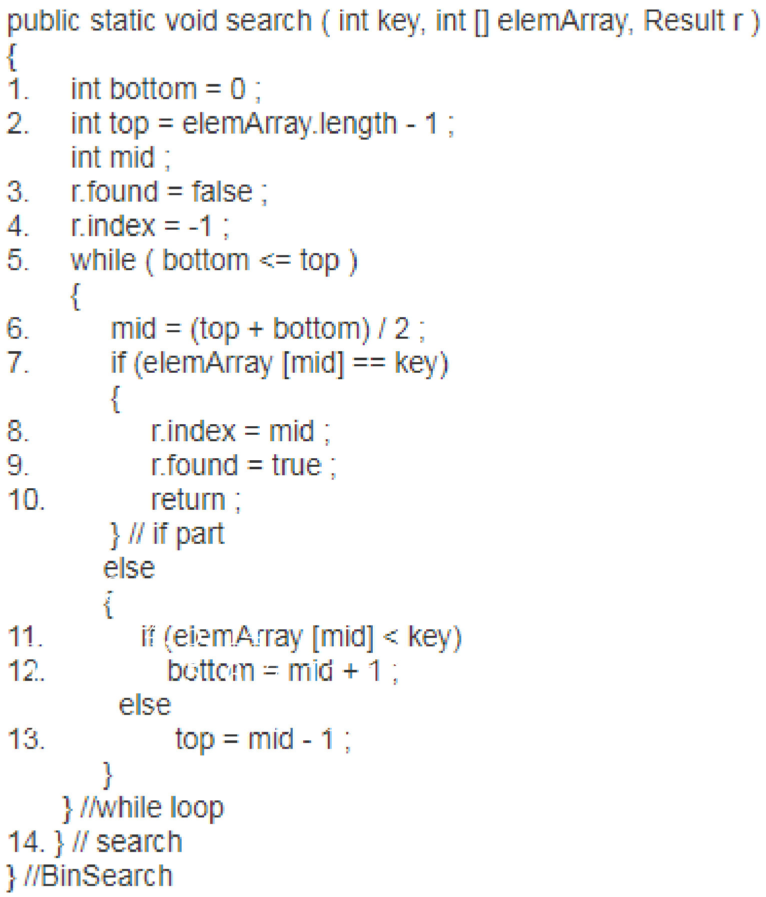


Test Cases-step-04

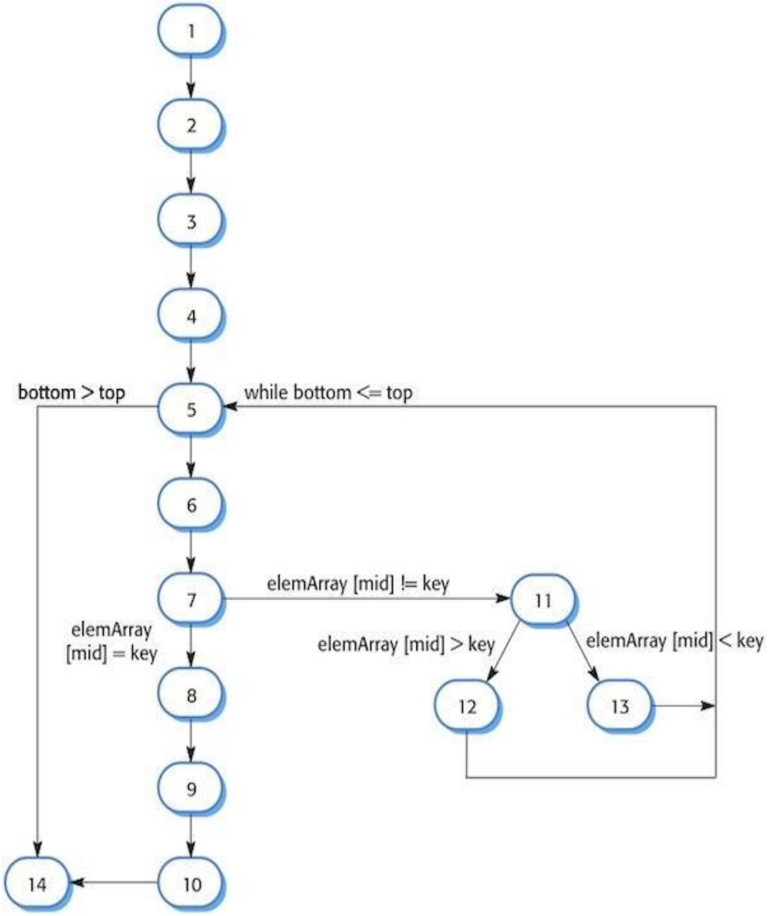


Path	Nextday	Amount	Expected Results
P1	yes	10	30.48
P2	No	1500	1713.25
P3	No	300	345.75
P4	No	150	174.125
P5	No	75	90.3875
P6	No	30	39.425
P7	No	10	15.975

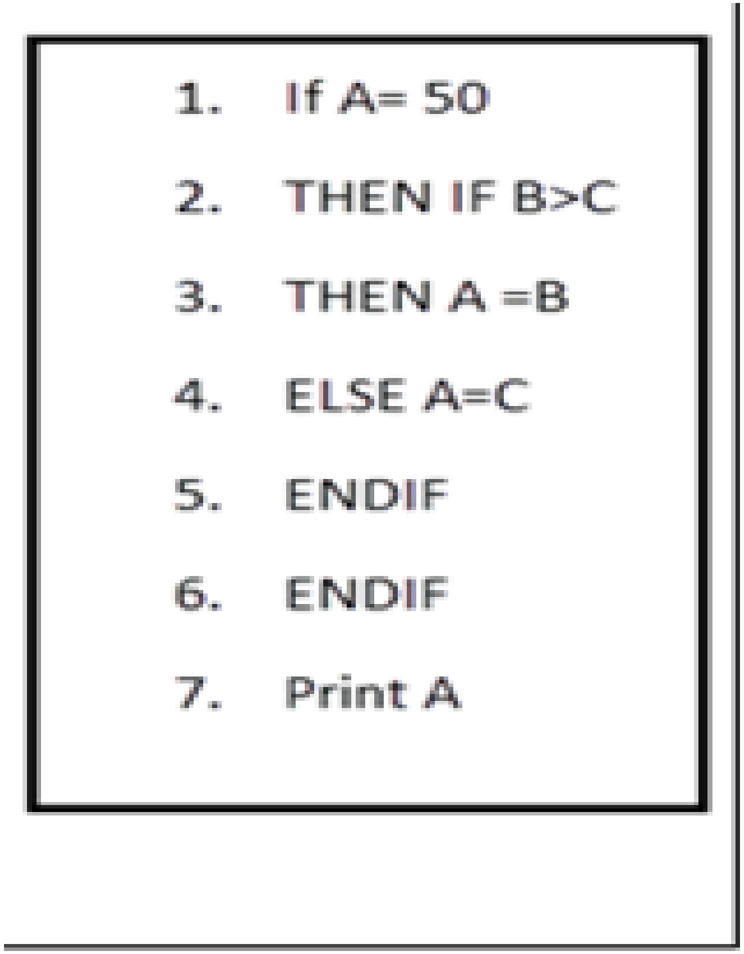
Example



```
public static void search ( int key, int [] elemArray, Result r )
{
1.     int bottom = 0 ;
2.     int top = elemArray.length - 1 ;
       int mid ;
3.     r.found = false ;
4.     r.index = -1 ;
5.     while ( bottom <= top )
       {
6.         mid = (top + bottom) / 2 ;
7.         if (elemArray [mid] == key)
           {
8.             r.index = mid ;
9.             r.found = true ;
10.            return ;
           } // if part
          else
          {
11.           if (elemArray [mid] < key)
12.              bottom = mid + 1 ;
            else
13.              top = mid - 1 ;
          }
      } //while loop
14. } // search
} //BinSearch
```

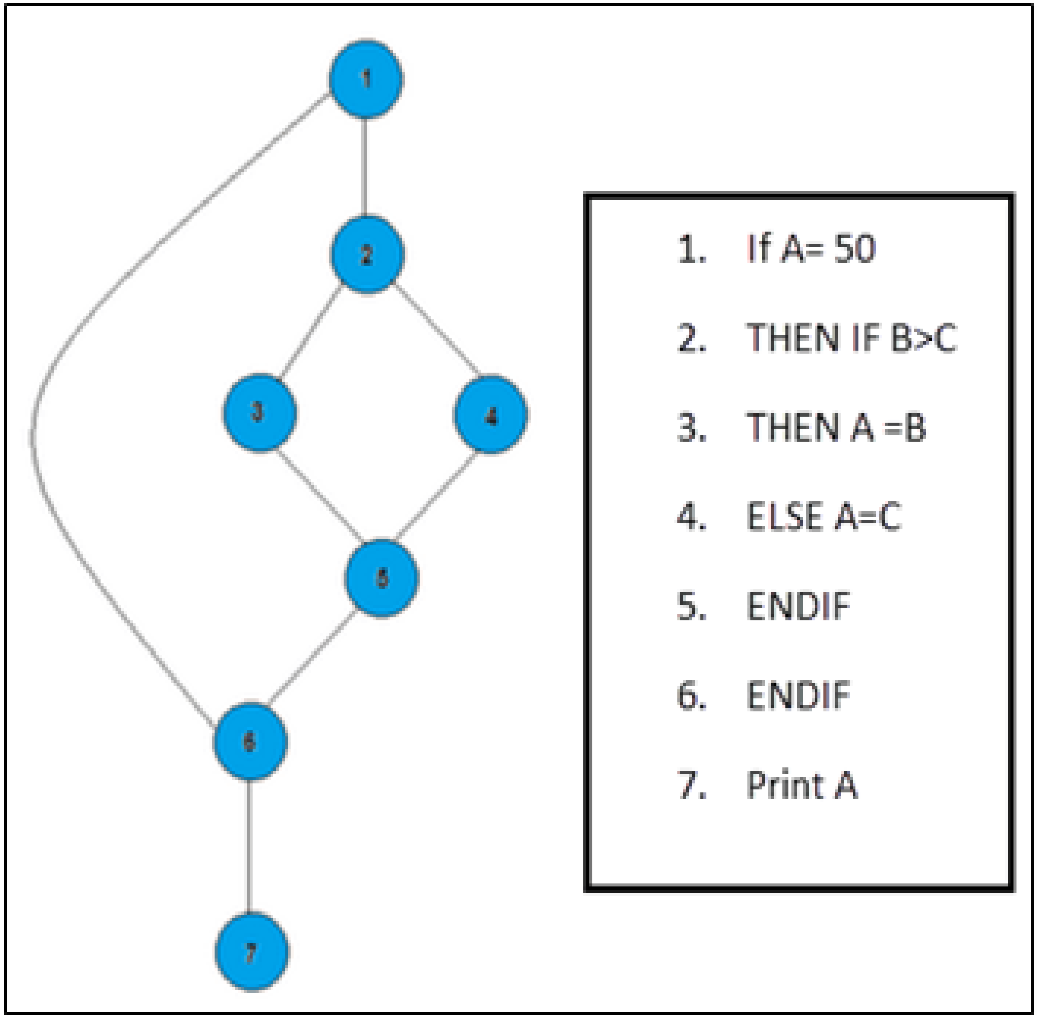


Example



```
If A= 50
THEN IF B>C
THEN A =B
ELSE A=C
ENDIF
ENDIF
Print A
```

imple example, to get a better idea what is basis path testing i



1. If A= 50
2. THEN IF B>C
3. THEN A =B
4. ELSE A=C
5. ENDIF
6. ENDIF
7. Print A

Binary Search Flow Graph

Paths

- **Path 1**: 1,2,3,5,6, 7
- **Path 2**: 1,2,4,5,6, 7
- **Path 3**: 1, 6, 7

Advantages of Basic Path Testing

- It helps to reduce the redundant tests
- It focuses attention on program logic
- It helps facilitates analytical versus arbitrary case design
- Test cases which exercise basis set will execute every statement in a program at least once

SYSTEM TESTING

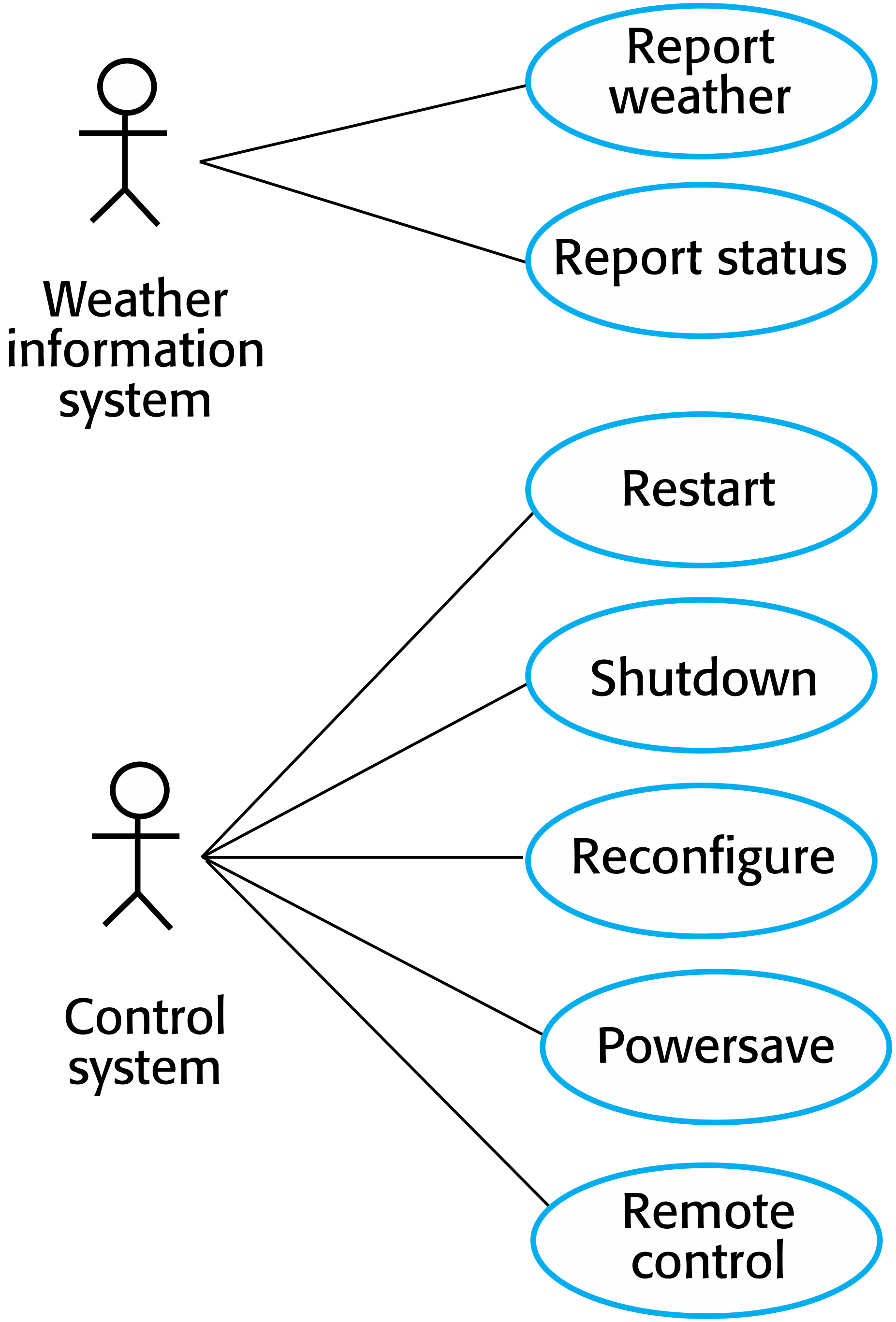
- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the emergent behavior of a system.

SYSTEM AND COMPONENT TESTING

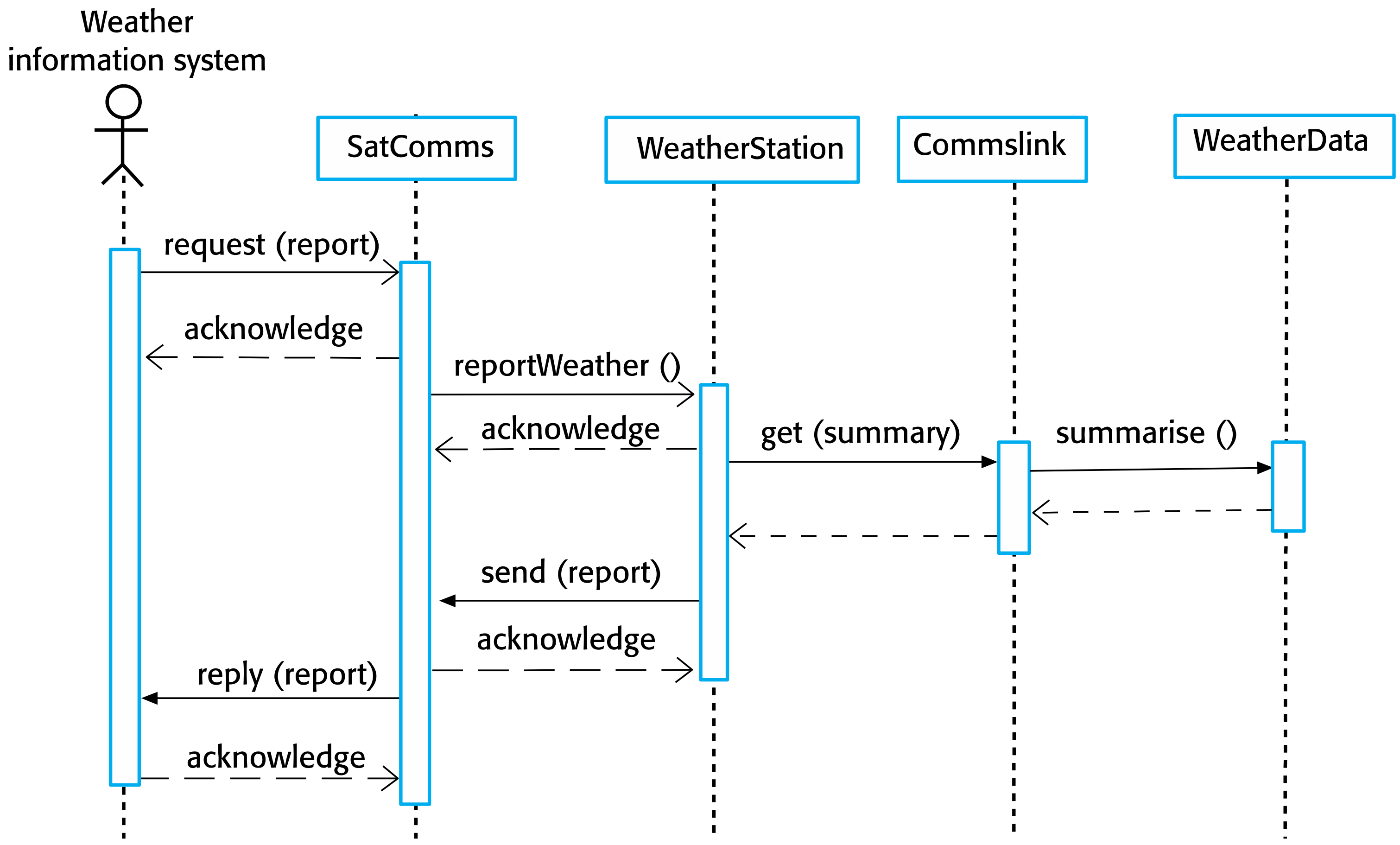
- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

USE-CASE TESTING

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.


Report weather
Report status
Weather information system
Restart
Shutdown
Reconfigure
Powersave
Remote control
Control system

COLLECT WEATHER DATA SEQUENCE CHART



TESTING POLICIES

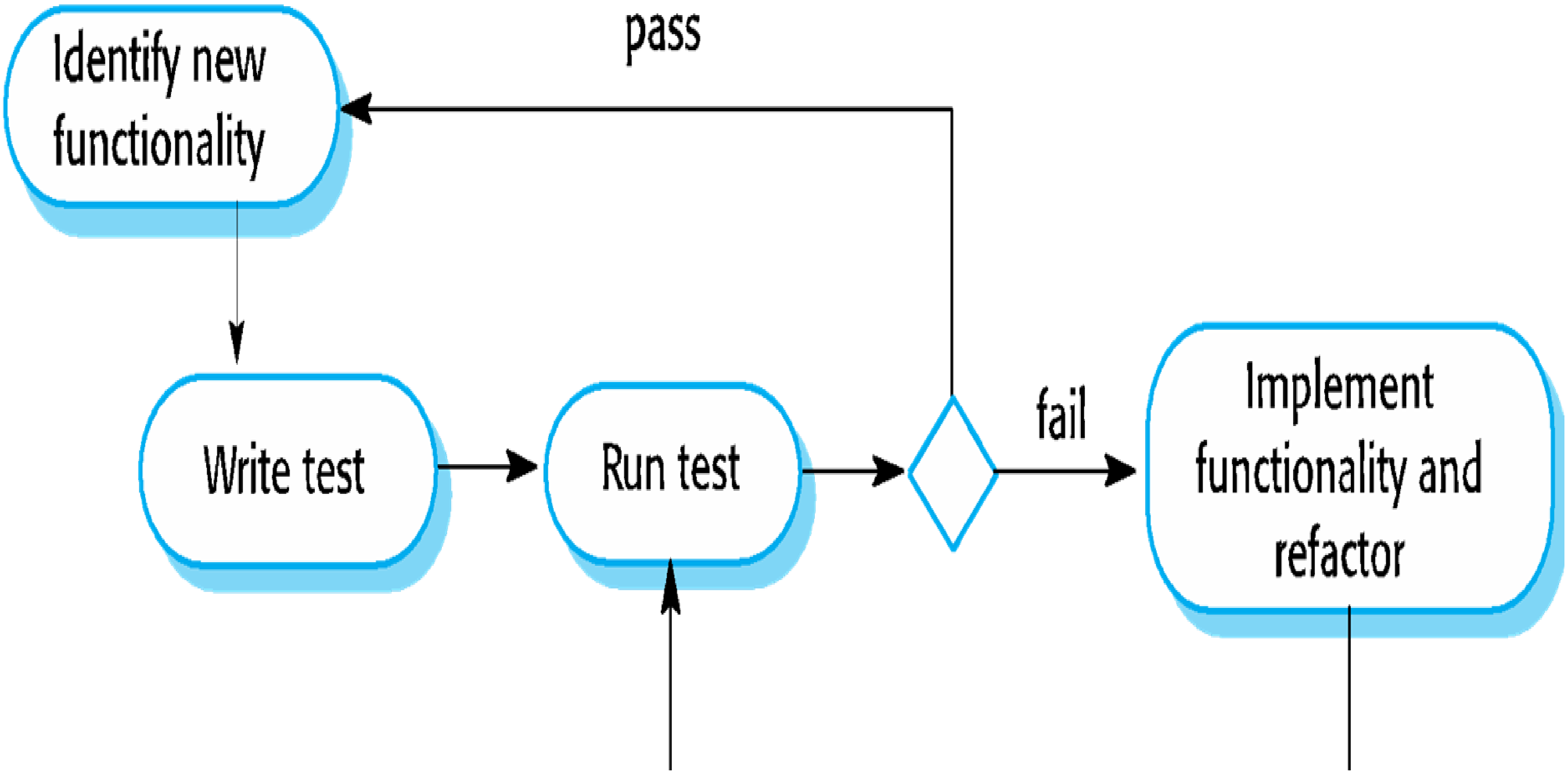
- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - Where user input is provided, all functions must be tested with both correct and incorrect input.

TEST-DRIVEN DEVELOPMENT

TEST-DRIVEN DEVELOPMENT

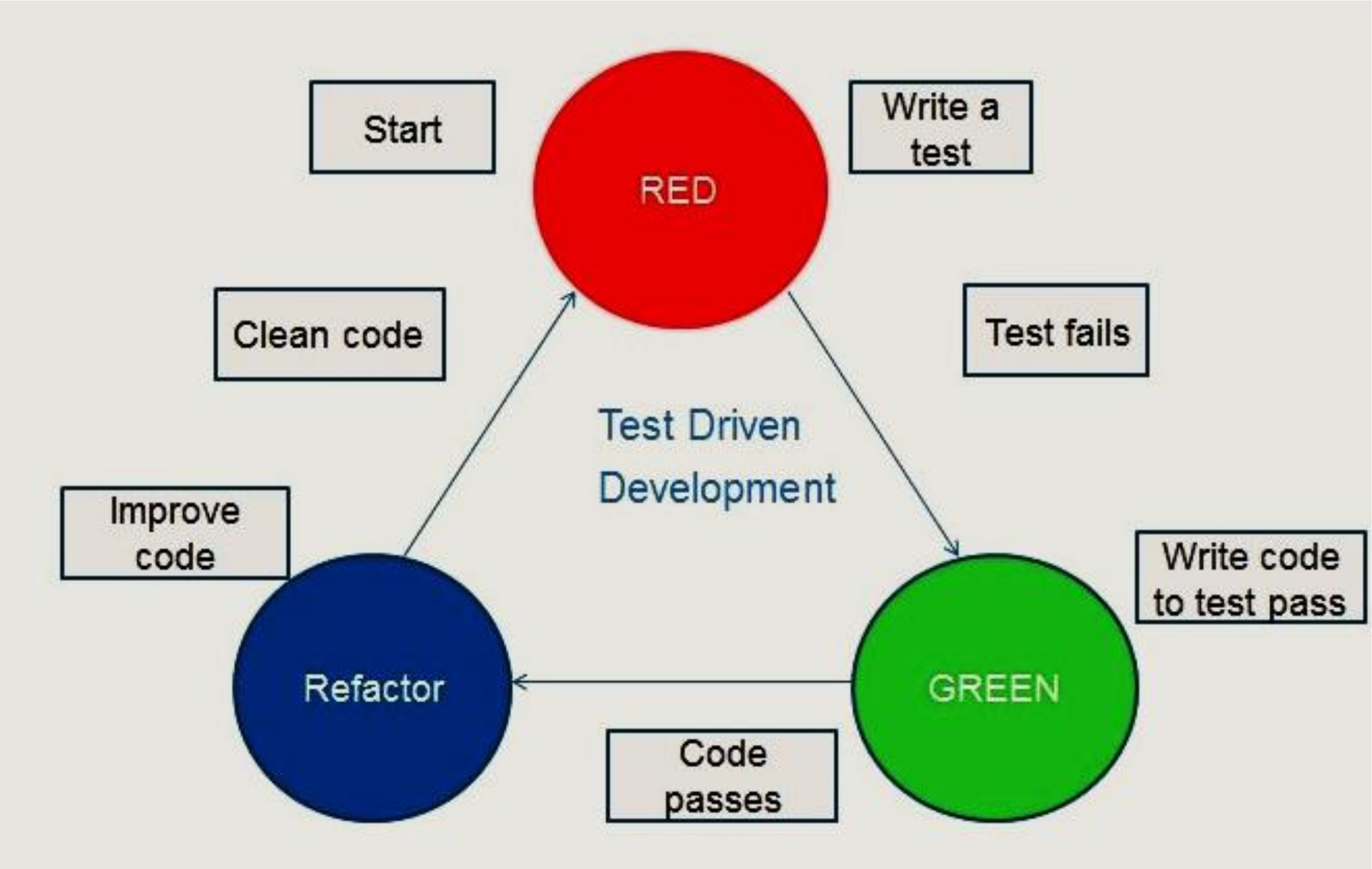
- Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- Tests are written before code and ‘passing’ the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

TEST-DRIVEN DEVELOPMENT

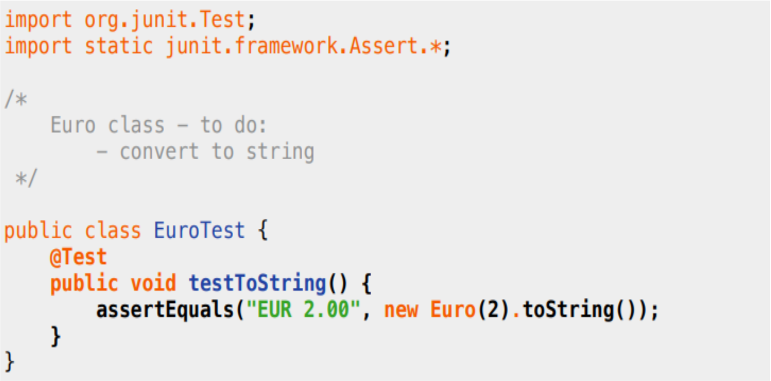


pass
Identify new functionality
Write test
Run test
fail
Implement functionality and refactor

TEST-DRIVEN DEVELOPMENT



Write a new test

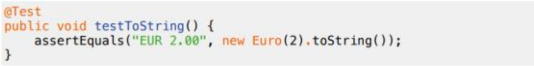


```
import org.junit.Test;
import static junit.framework.Assert.*;

/*
    Euro class - to do:
        - convert to string
 */

public class EuroTest {
    @Test
    public void testToString() {
        assertEquals("EUR 2.00", new Euro(2).toString());
    }
}
```

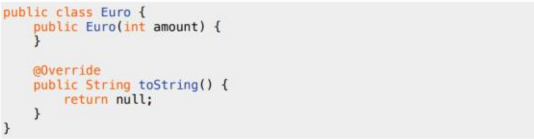
Run it ...

Cannot find symbol class Euro

Make the code compile



```
@Test
public void testToString() {
    assertEquals("EUR 2.00", new Euro(2).toString());
}
```

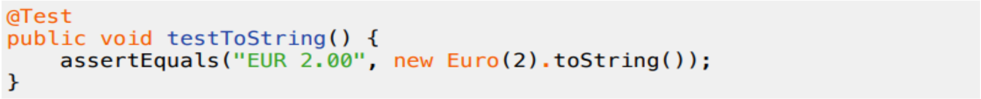


```
public class Euro {
    public Euro(int amount) {
    }

    @Override
    public String toString() {
        return null;
    }
}
```

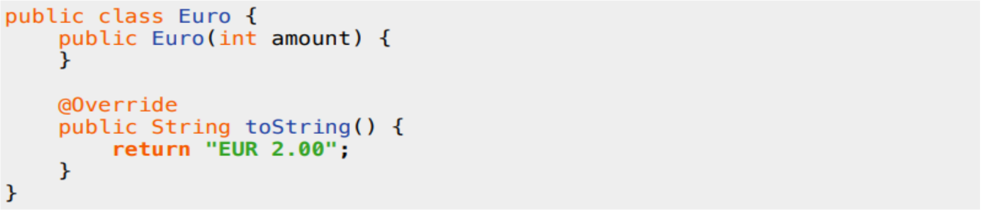
Run it again ...

junit.framework.ComparisonFailure:
Expected: EUR 2.00
Actual: <null>

Add a tiny bit of production code



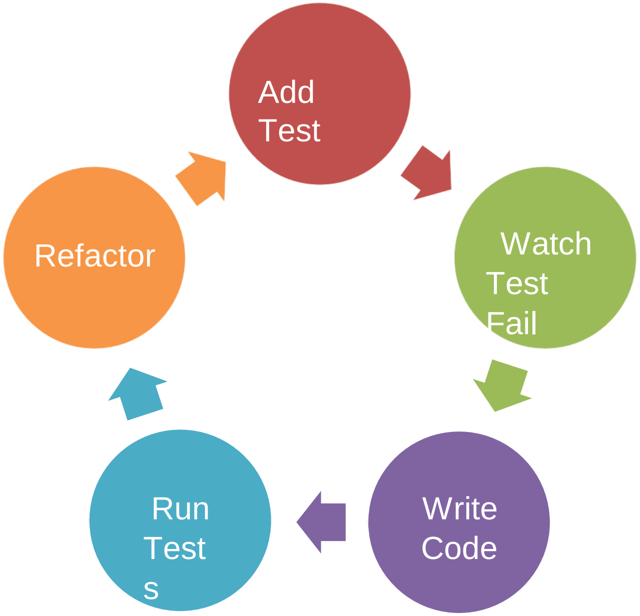
```
@Test
public void testToString() {
    assertEquals("EUR 2.00", new Euro(2).toString());
}
```



```
public class Euro {
    public Euro(int amount) {
    }

    @Override
    public String toString() {
        return "EUR 2.00";
    }
}
```

Run once more ...

Passed!



TEST-DRIVEN DEVELOPMENT

- You start by identifying the increment of functionality that is required.
- You write a test for this functionality and implement it as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
- You then run the test, IF fail.
- You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

BENEFITS OF TEST-DRIVEN DEVELOPMENT

- ***Code coverage***
  - every code segment that you write should have at least one associated test.
- ***Regression testing***
  - A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.
- ***System documentation***
  - The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.
- ***Simplified debugging***
  - When a test fails, it should be obvious where the problem lies.

REGRESSION TESTING

- One of the most important benefits of TDD is that it reduces the costs of regression testing.
- Regression testing involves running test sets that have successfully executed after changes have been made to a system.
- The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code.
- Regression testing is expensive and sometimes impractical when a system is manually tested, as the costs in time and effort are very high. You have to try to choose the most relevant tests to re-run..

RELEASE TESTING

RELEASE TESTING

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

RELEASE TESTING AND SYSTEM TESTING

- Release testing is a form of system testing.
- Important differences:
  - A separate team that has not been involved in the system development, should be responsible for release testing.
  - System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

REQUIREMENTS BASED TESTING

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- Mentcare system requirements:
  - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
  - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

REQUIREMENTS TESTS

Requirement Test	Action	Result
Patient record with no known allergies	Prescribe medication for allergies	Warning message should not be issued by the system
Patient record with a known allergy	Prescribe the medication to that the patient is allergic to	Warning message should be issued by the system
Patient record in which allergies to two or more drugs are recorded	Prescribe both of these drugs separately	Correct warning message should be issued by the system for each drug
Patient record with two known allergic drug	Prescribe two drugs that the patient is allergic to	Two warnings should be issued correctly by system
Overrule the Warning	Prescribe a drug that issues a warning	The system should require the user to provide information explaining why the warning was overruled

A USAGE SCENARIO FOR THE MENTCARE SYSTEM

George is a nurse who specializes in mental healthcare. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients that he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so he notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. Jim agrees so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation and the system re-encrypts Jim's record.

After, finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients who He has to contact for follow-up information and make clinic appointments.

FEATURES TESTED BY SCENARIO

- Authentication by logging
- Downloading and uploading patient records
- Home visit scheduling.
- Encryption and decryption of patient records
- Record retrieval and modification.
- Links with the drugs database & side-effect information
- System for call prompting.

PERFORMANCE TESTING

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

TYPES OF PERFORMANCE TESTING

- **Load testing –** checks the application's ability to perform under anticipated user loads. The objective is to identify performance bottlenecks before the software application goes live.
- **Stress testing –** involves testing an application under extreme workloads to see how it handles high traffic or data processing. The objective is to identify the breaking point of an application.

EXAMPLE PERFORMANCE TEST CASES

- Verify response time is not more than 4 secs when 1000 users access the website simultaneously.
- Verify response time of the Application Under Load is within an acceptable range when the network connectivity is slow
- Check the maximum number of users that the application can handle before it crashes.
- Check database execution time when 500 records are read/written simultaneously.
- Check CPU and memory usage of the application and the database server under peak load conditions
- Verify response time of the application under low, normal, moderate and heavy load conditions.

USER TESTING

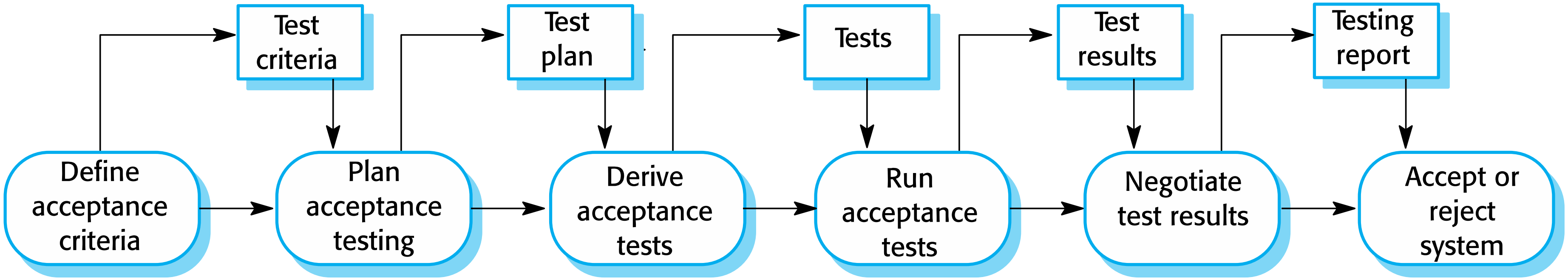
USER TESTING

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

TYPES OF USER TESTING

- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- Acceptance testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

THE ACCEPTANCE TESTING PROCESS



STAGES IN THE ACCEPTANCE TESTING PROCESS

- Define acceptance criteria
- Plan acceptance testing
- Derive acceptance tests
- Run acceptance tests
- Negotiate test results
- Reject/accept system

AGILE METHODS AND ACCEPTANCE TESTING

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is whether or not the embedded user is ‘typical’ and can represent the interests of all system stakeholders.