Lecture 07

Software Testing

Session Outcomes

- White Box Testing
- White Box Testing Techniques
  - Statement Coverage
  - Branch Coverage
- Unit Testing
  - Junit
- Non-functional Testing

Story So Far …

- So far we have discussed about
  - Types of testing
    - Black Box testing
    - White box testing
  - Black box testing strategies
    - Equivalence Class Testing
    - Boundary Value Testing
  - Software testing levels

- Now lets look into white box testing in more details…

What is Software Testing?

- **“Software Testing is the process of executing a program or system with the intent of finding errors”** [Myers, 79].
- **“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence”** [Dijkstra, 1972]

Why Testing is necessary?

- Executing a program with the intent of finding an *error*.
- To check if the system meets the requirements and be executed successfully in the Intended environment.
- To check if the system is “ Fit for purpose”.
- To check if the system does what it is expected to do.

Verification and Validation

- Verification:

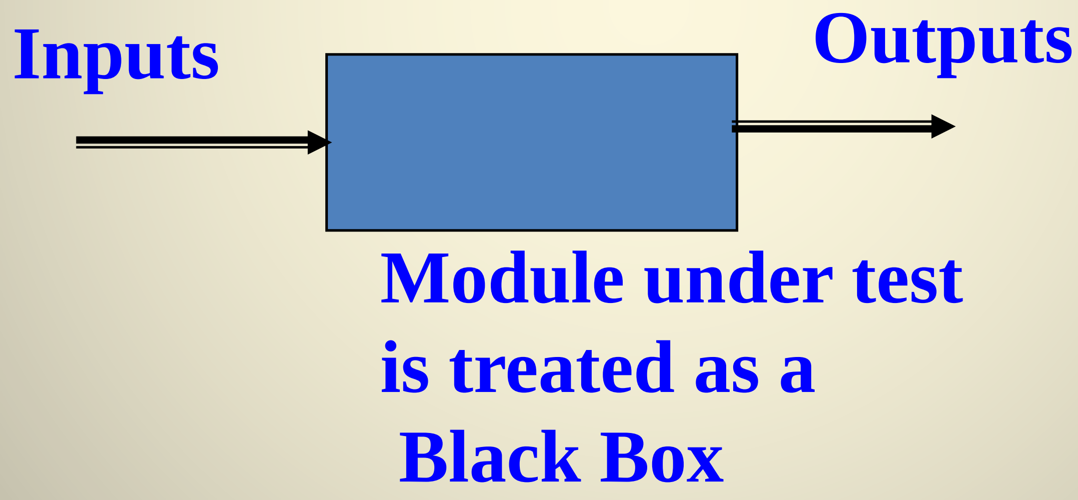
  "Are we building the product right"

  - The software should conform to its specification – functional and non-functional requirements
  - Typically involves reviews and meeting to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meeting.

- Validation:
  "Are we building the right product"
  - The software should do what the user really requires which **might** be different from specification.
  - Typically involves actual testing and takes place after verifications are completed.

Black Box Testing

- Testing focus on the software functional requirements, and input/output.



White Box Testing

– Testing is based on the structure of the program

- In white box testing internal structure of the program is taken into account.
- The test data is derived from the structure of the software.

- Tests are based on coverage of code statements, branches, paths, conditions.

- Most of the defects found in Unit and Integration is done using the white box testing.

White Box Testing - Techniques

Statement Coverage

– Execute all statements at least once

Branch (Decision/Edge) Coverage

– Execute each decision direction at least once

Condition (Predicate) Coverage

– Execute each decision with all possible outcomes at least once

Decision/Condition Coverage

– Execute all possible combinations of condition outcomes in each decision

Multiple Condition Coverage

– Invoke each point of entry at least once

Execute all statements at least once

Statement Coverage

Statement coverage involves execution of all the executable statements in the source code at least once.

Methodology

- Design test cases so that every statement in a program is executed at least once.

Principal Idea

- Unless a statement is executed, we have no way of knowing if an error exists in that statement.

Statement coverage is used to derive scenario based upon the structure of the code under test.

$$\text{Statement Coverage} = \frac{\text{No of executed statements}}{\text{Total no of statements}} * 100\%$$

Example

- Calculate the no of test cases needed for full statement coverage for the given scenario.

```
Printsum (int a, int b)
 {
        int result = a+ b;
        If (result> 0)
                Print ("Positive", result);
        Else
                Print ("Negative", result);
}
```

Step 1: What is the total number of statements in the code?

```
Printsum (int a, int b) {
        int result = a+ b;
        If (result> 0)
                Print ("Positive", result);
        Else
                Print ("Negative", result);
}
```

Total no of statements = 7

Step 2: Find out the executed no of statements for a=3 and b=9.

Test Case 1 – if a=3, b=9

```
Printsum (int a, int b) {
        int result = a+ b;
        If (result> 0)
                Print ("Positive", result);
        Else
                Print ("Negative", result);
}
```

No of executed statements = 5

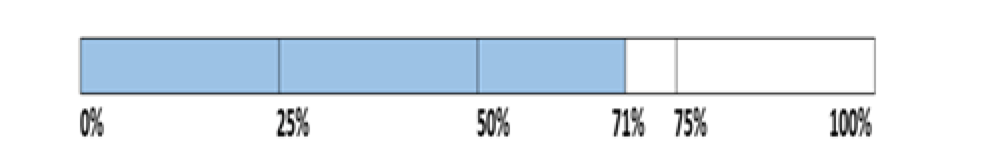
Step 3: Find the statement coverage.

Test Case 1 – if a=3, b=9

```
Printsum (int a, int b) {
        int result = a+ b;
        If (result> 0)
                Print ("Positive", result);
        Else
                Print ("Negative", result);
}
```

No of executed statements = 5
Total no of statements = 7

Statement coverage = 5/7*100 = 71%



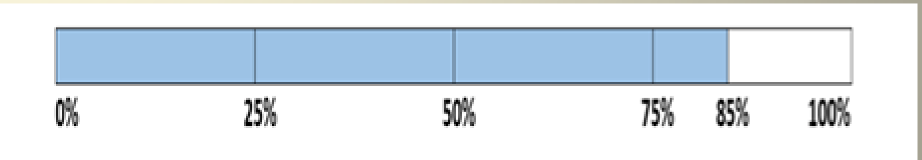
Step 4: Again check the statement coverage when a=-3 and b=-9.

Test Case 2 – if a=-3, b=-9

```
Printsum (int a, int b) {
        int result = a+ b;
        If (result> 0)
                Print ("Positive", result);
        Else
                Print ("Negative", result);
}
```

No of executed statements = 6
Total no of statements = 7

Statement coverage = 6/7*100 = 85%



Overall we can say all the statements are fully covered by using the two test cases. So the overall statement coverage of 100% can be achieved by the above two test cases.

Activity

- Calculate the no of minimum test cases needed for full statement coverage for the given scenario.

```
int f1(int x, int y){
        while (x != y){
                if (x>y)
                        x=x-y;
                else y=y-x;
                }
        return x;       }
```

Branch Coverage

Branch coverage covers both the true and false conditions unlikely the statement coverage.

Methodology

- Test cases are designed such that different branch conditions given true and false values in turn.

Principal Idea

- This technique checks every possible path (decisions).
- A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement.

- A branch is the outcome of a decision, so branch coverage simply measures which decision outcome have been tested.

- This takes more in depth view of the source code compared to statement coverage.

$$\text{branch Coverage} = \frac{\text{No of executed branches}}{\text{Total no of branches}} * 100\%$$

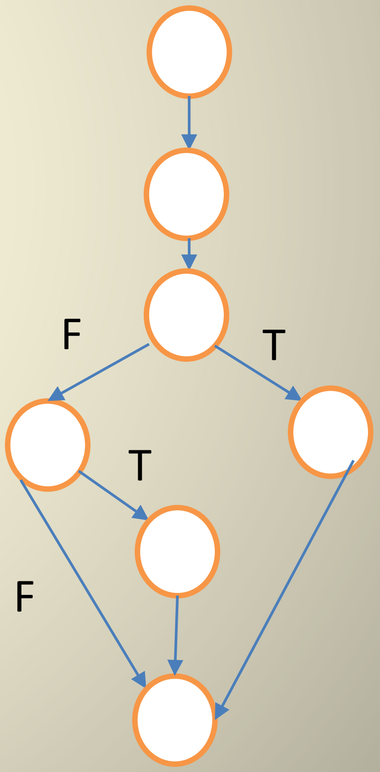
Example

Calculate the no of minimum test cases needed for full branch coverage for the given scenario.

```
Printsum (Int a, Int b) {
        Int result = a+ b;
        If (result> 0)
                Print ("Positive", result);
        else if (result<0)
                Print ("Negative", result);
        else
                do nothing;
}
```

Step 1: Come up with a simple control flow graph for the given code.

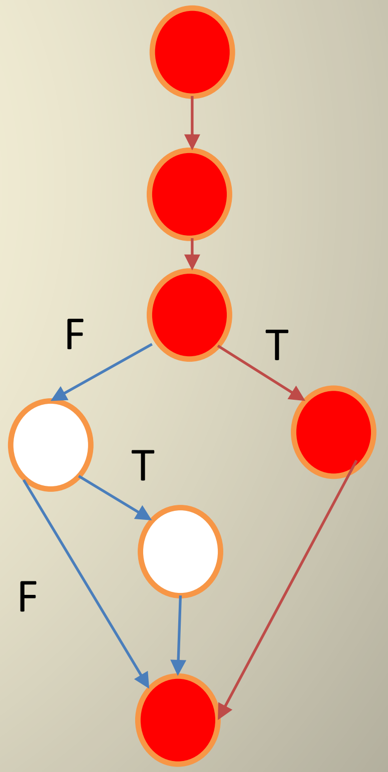
```
Printsum (Int a, Int b){
        Int result = a+ b;
        if(result>0)
                Print ("Positive", result);
        else if (result<0)
                Print ("Negative", result);
        else
                do nothing;
}
```



Step 3: Traverse through the graph when a=3 and b=9.

Test case 1 – a=3,b=9

This test case covers the path highlighted.

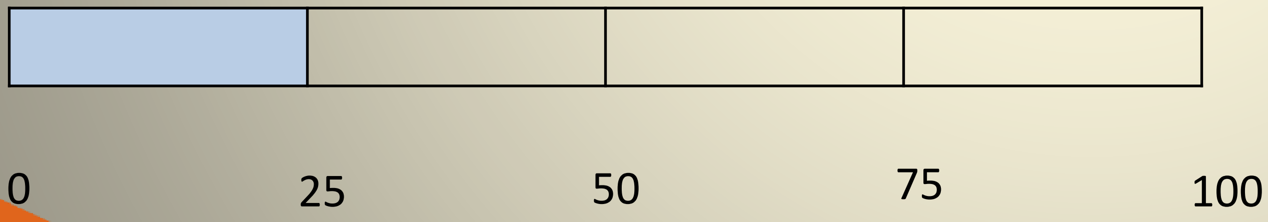


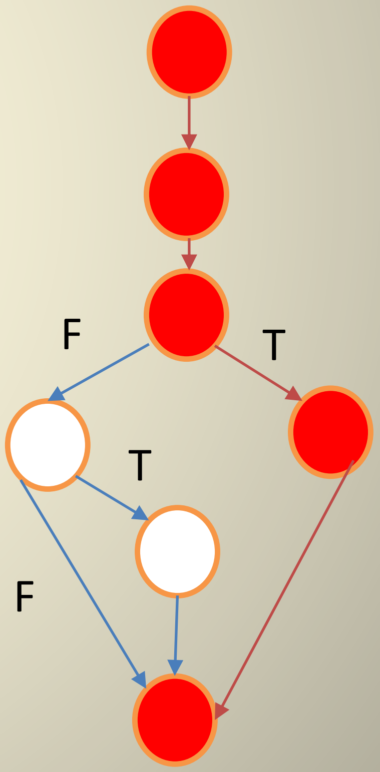
Step 4: Calculate the statement coverage when a=3 and b=9.

Test case 1 – a=3,b=9

This test case covers the path highlighted.

branch coverage=1/4*100=25%



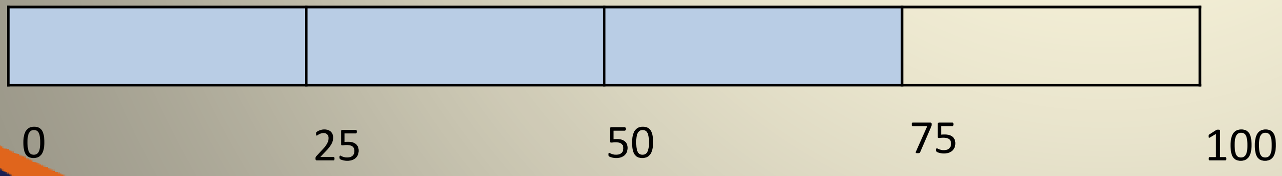


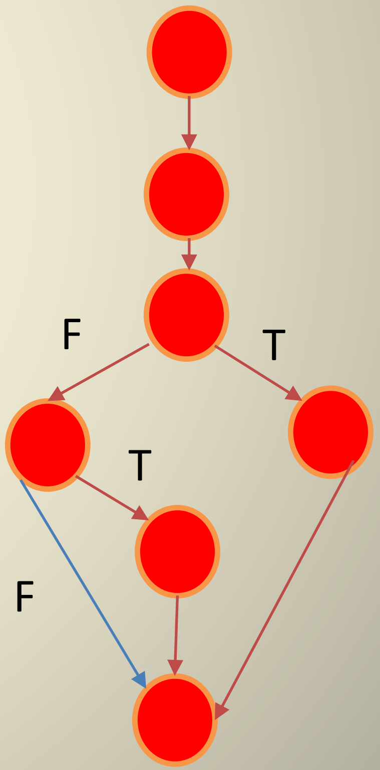
Step 5: Calculate the statement coverage when a=-5 and b=-8.

Test case 2 – a=-5,b=-8

This test case covers the path highlighted.

branch coverage=2/4*100=50%



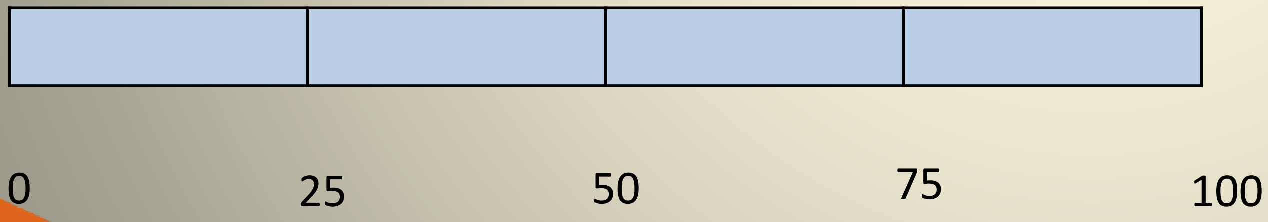


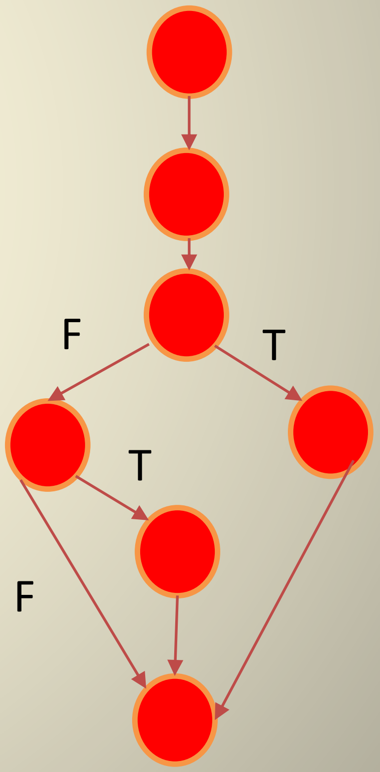
Step 5: Calculate the statement coverage when a and b is 0.

Test case 2 – a=0,b=0

This test case covers the path highlighted.

branch coverage=2/4*100=50%





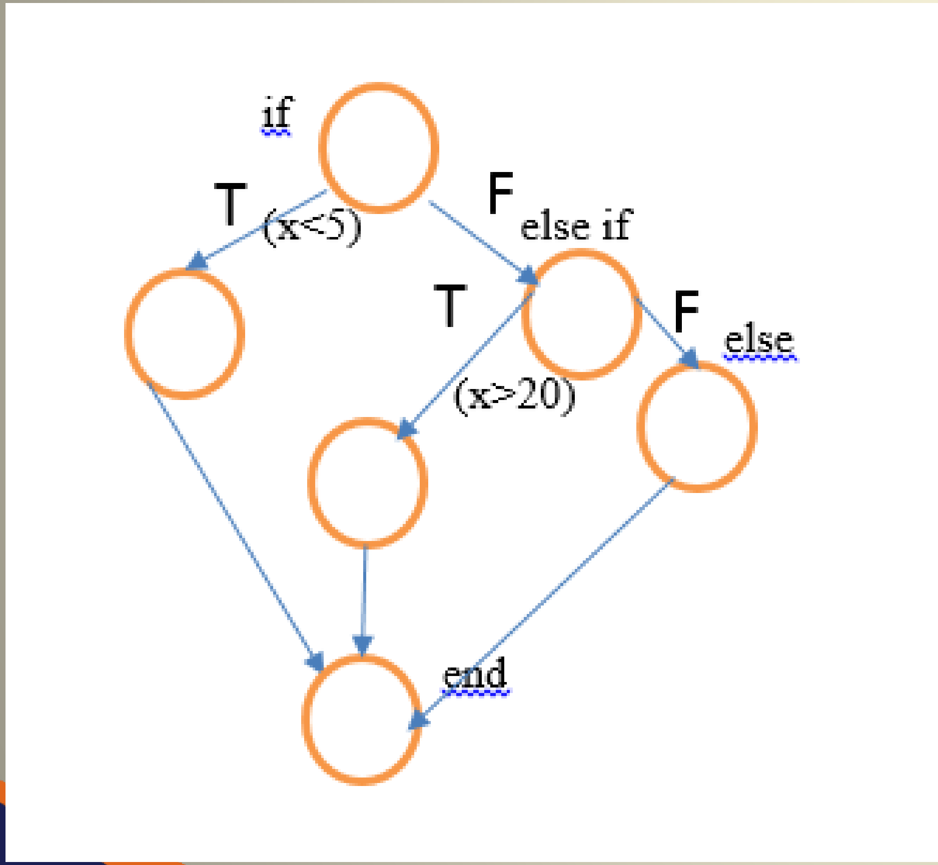
By using minimum three test cases we can test all the branches.

Activity

What is the minimum number of test cases required to achieve full branch coverage for the program segment given below?

```
Void print(int x) {
        if( x < 5){
                ............... }
        else if (x > 20){
                        ............ }
        else {
                        ............ }
}
```

Activity Answer



You need to have minimum three test cases to get full branch coverage.

Eg: { (x=2),(x=25),(x=10) }

What do we test?

- Functional Test
  - Unit Testing
  - Integration Testing
  - System Testing
  - Acceptance Testing

- Non – Functional Test
  - Performance
    - Stress / Load
    - Usability
    - Scalability etc.

Functional Testing

- Unit testing : Individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
- Integration 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. System testing should focus on testing component interactions.
- 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.

Functional Test - Unit Testing

- The most ‘micro’ scale of testing.
- Tests done on particular functions or code modules. it is the testing of single entity (class or method).
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).
- Unit testing can be done in two different ways.
  - Manual Testing
  - Automated Testing

Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.
Time-consuming and tedious – Since test cases are executed by human resources, it is very slow and tedious.	**Fast** – Automation runs test cases significantly faster than human resources.
Huge investment in human resources – As test cases need to be executed manually, more testers are required in manual testing.	**Less investment in human resources** – Test cases are executed using automation tools, so less number of testers are required in automation testing.
Less reliable – Manual testing is less reliable, as it has to account for human errors.	**More reliable** – Automation tests are precise and reliable.
Non-programmable – No programming can be done to write sophisticated tests to fetch hidden information.	**Programmable** – Testers can program sophisticated tests to bring out hidden information.

Why Unit Testing?

- Faster Debugging
- Faster Development
- Better Design
- Excellent Regression Tool
- Reduce Future Cost

Unit Testing Frameworks

- What are Unit Testing Frameworks?
  - It's a set of guidelines which will help to run the unit testing.

- Examples of UTFs and Where to get them?
  - www.junit.org
  - www.nunit.org
  - www.xprogramming.com

Characteristics of UTFs

- Most UTFs target OO and web languages
- UTFs encourage separation of business and presentation logic
- Tests written in same language as the code
- Tests are written against the business logic
- GUI and command line test runners
- Rapid feedback

JUnit (www.junit.org)

- Java-based unit testing framework
- Elegantly simple
- Easy to write unit tests
- Easy to manage unit tests
- Open source = Free!
- Use to incrementally build a test suite
  - write the tests as you write the code...
  - JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented.

What is a Unit Test Case?

- A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected.
- A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed.
- There must be at least two unit test cases for each requirement – one positive test and one negative test.

Eg: Try to delete an existing employee in the system -> Positive Test Case

Try to delete a non-existing employer in the system -> Negative Test Case

Unit Testing on JUnit

1. A unit test consists of a “test class” normally corresponding to a specific class in your project – for a class named MyClass the test might look like this,

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class MyClassTest {
        …
}
```

2. This test class can have a few methods for which you provide so called “decorators”. The decorator tells JUnit that this is a special test method.

```
Public class MyClassTest {
        @Test          → Decorator
        public void testMyMethod()
        {
        }
}
```

- One JUnit class will normally have multiple @Test methods for the different public methods
- It can also have a few other methods in your class under test.

Method Name	Decorator	Description
setUpBeforeClass	@BeforeClass	Runs before all the @Test test methods in your Test class.
tearDownBeforeClass	@AfterClass	Runs after all the @Test test methods in your Test class.
setUp	@Before	Runs before each @Test
tearDown	@After	Runs after each @Test
testMethod	@Test	Used for each individual test.

3. Inside each test method you will be running some code usually calling one method in the class you are testing.

Assertions:

```
String expectedValue = “my expected value”;
String actualValue = myClass.method();
assertEquals(expectedValue, actualValue);
```

There are multiple types of assertions you can make – but the most important ones are assertEquals, assertNotEquals, assertTrue and assertFalse.

Non-Functional Testing

- Non functional testing us used to test the non-functional requirements of the system.

What are Non-functional requirements?

Basically non functional requirements describe how the system works.

eg: Usability, reliability, Performance etc.

Non functional Test - Performance Testing

- Performance testing, a non-functional testing technique performed to determine the system parameters in terms of responsiveness and stability under various workload.

- Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage.

Load testing

It is the simplest form of testing conducted to understand the behavior of the system under a specific load.

Load testing will result in measuring important business critical transactions and load on the database, application server, etc., are also monitored.

Stress testing

It is performed to find the upper limit capacity of the system and also to determine how the system performs if the current load goes well above the expected maximum.

Spike testing

Spike testing is performed by increasing the number of users suddenly by a very large amount and measuring the performance of the system.

The main aim is to determine whether the system will be able to sustain the workload.

Load Testing

This testing usually identifies,

- The maximum operating capacity of an application.
- Determine whether current infrastructure is sufficient to run the application.
- Sustainability of application with respect to peak user load.
- Number of concurrent users that an application can support, and scalability to allow more users to access it.

- Load testing is commonly used for the Client/Server, Web based applications.

Why?

- Some extremely popular sites have suffered serious downtimes when they get massive traffic volumes.

Examples:

- Popular toy store Toysrus.com, could not handle the increased traffic generated by their advertising campaign resulting in loss of both marketing dollars, and potential toy sales.
- An Airline website was not able to handle 10000+ users during a festival offer.
- Encyclopedia Britannica declared free access to their online database as a promotional offer. They were not able to keep up with the onslaught of traffic for weeks.

- There are lot different tools available for performance testing.

References

- Software Engineering, I.Sommerville, 10th ed. , Pearson Education.
- Junit 5 User Guide: https://junit.org/junit5/docs/current/user-guide/