School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 3a: Test Automation and Junit Introduction

CSI3105:
Software Testing

Review Questions

1. Explain the following terms associated with Control Flow Graphs:
   - Nodes
   - Directed Edges
   - Basic Block
   - Path
2. Draw the Control Flow Graph for the following code segments:

```
Q= 15;
 P = 5;
   while  (Q > P)
 {
                P = R (Q, P);
                Q = Q – 1;
 }
```

```
for ( y = 100; y > x; x--)
{
    y = y – 5;
    x = x + 1;
}
```

3. Assume a predicate P(x) that represents the statement:
   - X is a prime number

What is the truth values for the following:

P(2) - True

P(3) - True

P(4) - False

P(5) - True

Review Questions

An abstract representation of a program/method that models all executions of a program/method by describing control structures

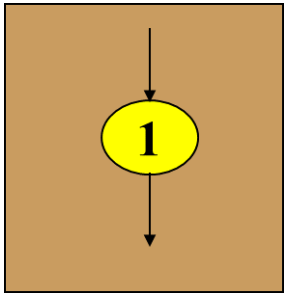
- Nodes : Statements or sequences of statements (basic blocks)
- Directed Edges : Transfers of control (branch; alternate paths)

Basic Block : A sequence of statements such that if the first statement is executed, all statements will also be executed (no branches). It has unique entry and exit points.

Path : A collection of *Nodes* linked with *Directed Edges*

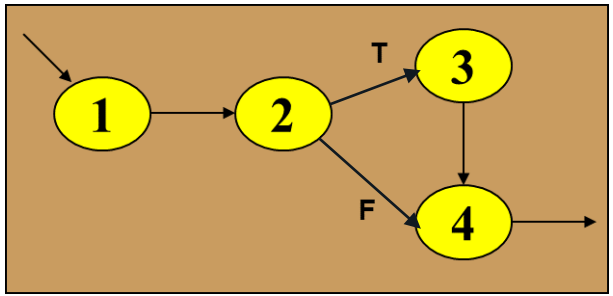
Review Questions

Simple Basic block
represented as one node




```
Statement1;
Statement2;
Statement3;
Statement4;
```

CFG



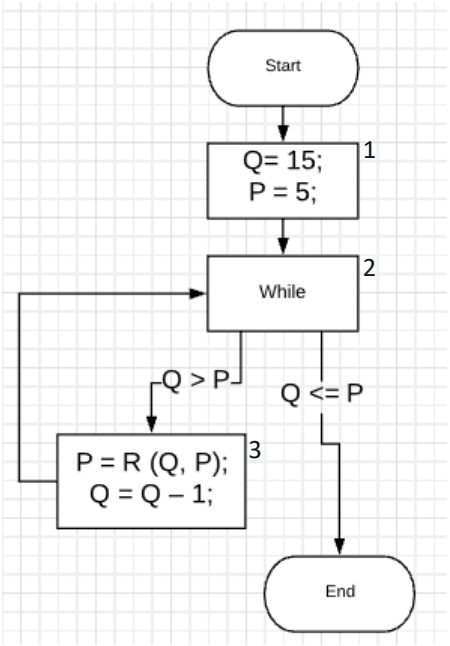

```
Statement1;
Statement2;

if X < 10 then
   Statement3;

Statement4;
```

1
2
3
4

Review Questions

Draw the Control Flow Graph for the following code segments:

```
Q= 15;
 P = 5;
   while  (Q > P)
 {
                P = R (Q, P);
                Q = Q – 1;
}
```



Review Questions

Assume a predicate P(x) that represents the statement:

- X is a prime number

What is the truth values for the following:

P(2) - True

P(3) - True

P(4) - False

P(5) - True

Lecture Summary

The learning goal for this week are:

- To understand what is test automation and issues associated with test automation
- To use JUnit for unit testing

This module relates to Chapter 10 of the textbook

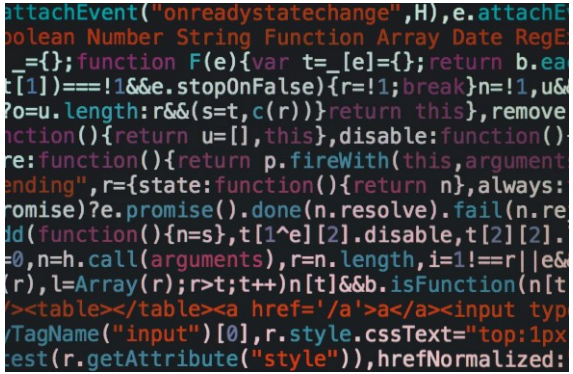
School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 3b: Software Testability and Test Cases

CSI3105:
Software Testing

Software Testing



- We code and introduce errors
- Potentially create a whole mess

…

- Software testing can be hard and laborious
- Some pieces of software can be harder than others to test

What is Software Testability?

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met (from Ammann & Offut, 2016)

Testability of the software module is high, implies finding faults in the system via testing is easier

Testability – how likely the testing will find a fault

Some metrics associated with software testability:

- Observability
- Controllability
- Availability
- Simplicity
- Stability

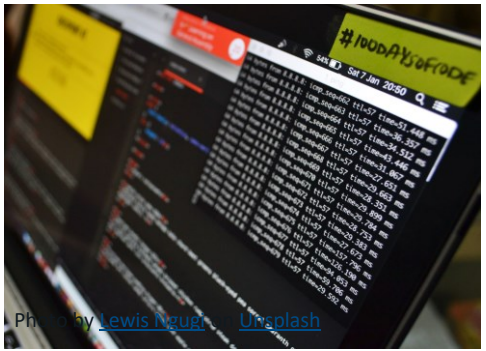
Observability

Observability

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

- Software that affects hardware devices, databases, or remote files have low observability

Hard!



Easier!

Controllability

Controllability

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

- Easy to control software with inputs from keyboards
- Inputs from hardware sensors or distributed software is harder

Data abstraction reduces controllability and observability

Test Cases Introduction

Controllability

- Giving input to the software under test

Observability

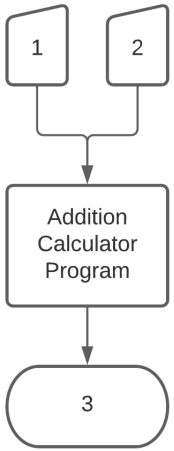
- Seeing what the software does after we give it the input

In a simple sense, to test software we must provide

- Some form of inputs to the system under test
- Observer and compare the output against what we expect to happen

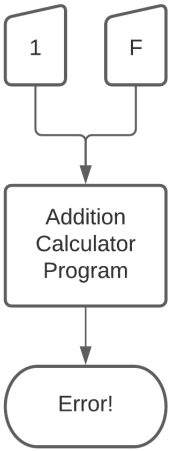
Test Cases

Test case #1:
Input: 1, 2
Expected output: 3



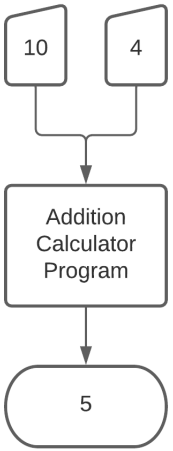
Expected output: 3
Observed output: 3
Passed

Test case #2:
Input: 1, F
Expected output: Error!



Expected output: Error!
Observed output: Error!
Passed

Test case #1:
Input: 10, 4
Expected output: 14



Expected output: 14
Observed output: 5
Failed

3:

Components of a Test Case (3.2)

A test case is a multipart artifact with a definite structure

Input Values

- Test case values

> The input values needed to complete an execution of the software under test

Expected result

Expected results

> The result that will be produced by the test if the software behaves as expected

- A *test oracle* uses expected results to decide whether a test passed or failed

Affecting Controllability and Observability

Prefix values

Inputs necessary to put the software into the appropriate state to receive the test case values

Postfix values

Any inputs that need to be sent to the software after the test case values are sent

1. *Verification Values* : Values needed to see the results of the test case values
2. *Exit Values* : Values or commands needed to terminate the program or otherwise return it to a stable state

Putting Tests Together

Test case

The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test

Test set

A set of test cases

Executable test script

A test case that is prepared in a form to be executed automatically on the test software and produce a report

School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 3c: Manual vs Automated

CSI3105:
Software Testing

Manual Testing

- Software tester essentially takes up the role of the end user

Manual Testing

- Write test cases
- Set up the test environment/ Load Test data
- Execute tests
- Compare results
- Log tests results
- Clear up test environment
- Summarize results into a report
- Analysis of the test reports

Automated Testing

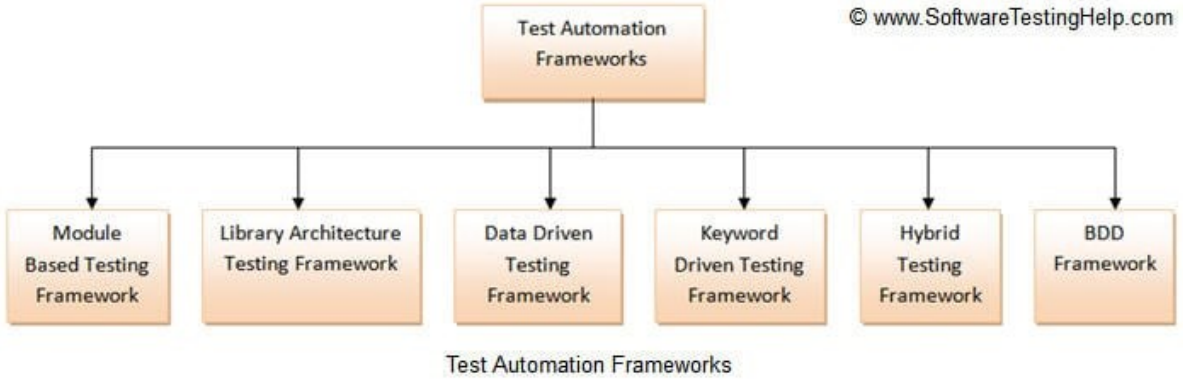
- Manual testing is good..
  - Can be a long and fatiguing process
  - Human resource intensive
  - This can introduce human error
- Test cases are executed with the assistance of tools, scripts, and software.
- Using software or tools we can execute many test cases without human intervention
  - Simulate thousands of concurrent users

Test Automation Framework

An integrated system with a combination of sets the rules, protocols, standards and guidelines to be followed. Components (e.g. function libraries, test data sources, object details and various reusable modules) within this system are the small building blocks that need to be assembled to represent a testing process.

It provides the basis of test automation and simplifies the automation effort.

It is always application independent. It should be scalable and maintainable.



Test Automation Frameworks

Advantages of Test Automation frameworks

Modularity

Understand-ability

Maximum test coverage

Code reuse

Recovery test scenarios

Low-cost maintenance

Minimal manual intervention

Ease of Reporting

Automated Testing

Definition: The act of conducting specific tests via automation (such as executing a set of regression tests) as opposed to conducting them manually.

Lets have a look at how we could automate some tests

School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 3d: Junit Introduction

CSI3105:
Software Testing

What is JUnit?

Open source Java testing framework used to write and run repeatable automated tests

JUnit is open source (junit.org)

A structure for writing test drivers

JUnit features include:

- Assertions for testing expected results
- Test features for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners

JUnit is widely used in industry

JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse

JUnit Tests

JUnit can be used to test …

- … an entire object
- … part of an object – a method or some interacting methods
- … interaction between several objects

It is primarily intended for unit and integration testing, not system testing

Each test is embedded into one test method

A test class contains one or more test methods

Test classes include :

- A collection of test methods
- Methods to set up the state before and update the state after each test and before and after all tests

Get started at junit.org

Junit 5

Architecture – composed of 3 separated modules:

- **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

- JUnit Platform
  - for launching testing frameworks on the JVM. It defines the TestEngine API for developing a testing framework that runs on the platform.

- JUnit Jupiter
  - API for writing Junit 5 tests. Includes new programming model and extension model for writing tests and extensions in JUnit 5 (examples: new assertions, new annotations, and Java 8 Lambda Expressions)

- JUnit Vintage
  - Supports running JUnit 3 and JUnit 4 based tests on the JUnit 5 platform.

Needs Java 8 to run

Junit 5

Why use JUnit?

- Can’t we use the debugging tool in our IDE
- Can’t we add print statements in our code

These methods are good for finding logic errors

- *Can* work for extremely small scale projects
- Limited to our own judgement
    - If we have already made the error, will we be able to catch it?

JUnit provides automated testing

- Can test many scenarios in one execution

Writing Tests for Junit 5

Need to use the methods of the **org.junit.jupiter.api.Assertions** class

Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded

The results from the method calls is used by the test runner to report to the user (in command line mode) or update the display (in an IDE)

All of the methods return void

A few representative methods of **org.junit.jupiter.api.Assertions**

- *assertTrue (boolean)*
- *assertTrue (boolean, String)*
- *fail (String*)
- For more info about these methods, check out: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

Test Fixtures

A test fixture sets up the data needed for every test

- Objects and variables that are used by more than one test
- Initializations (*prefix* values)
- Reset values (*postfix* values)

Different tests can use the objects without sharing their state

Objects used in test fixtures should be declared as instance variables

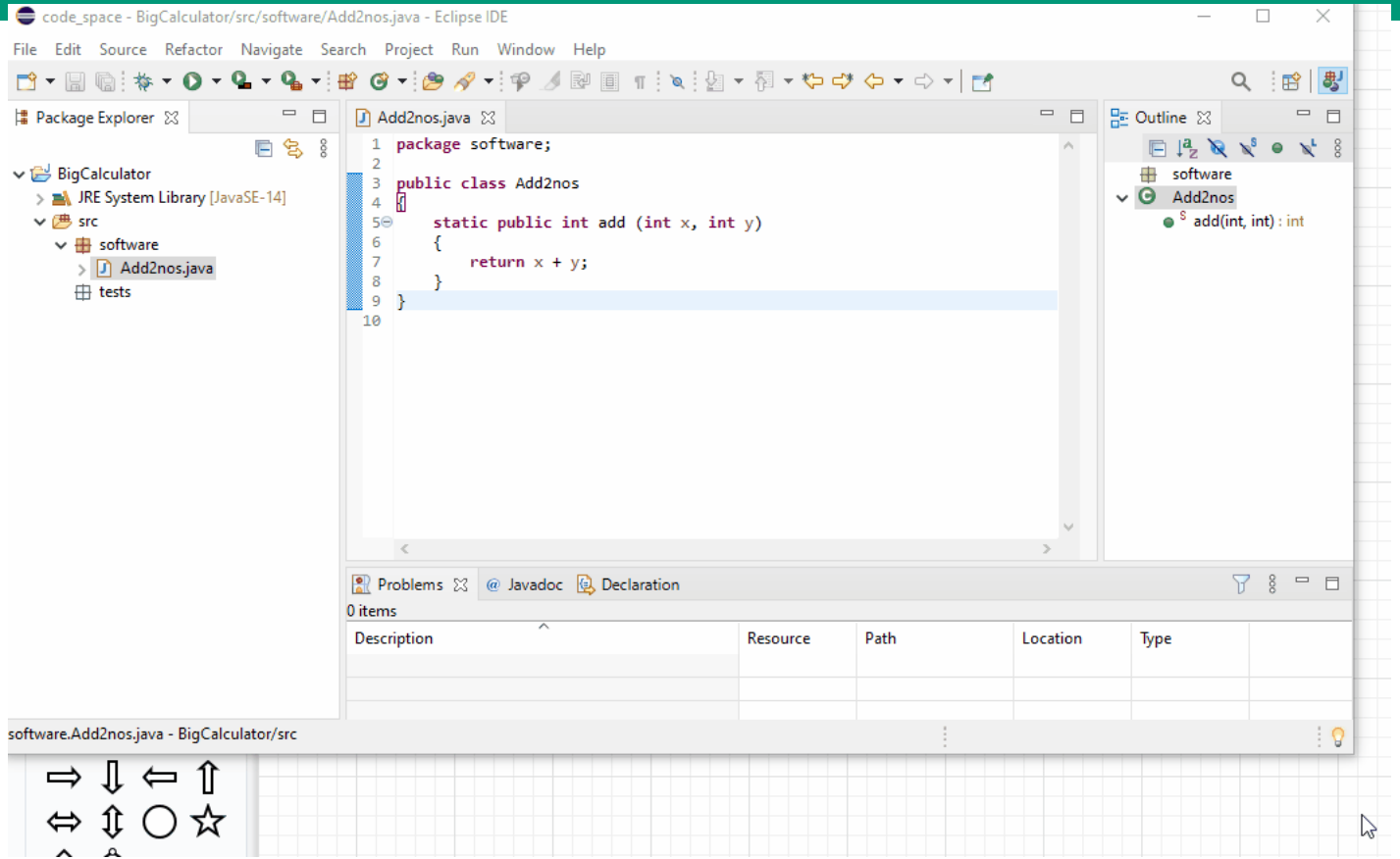
Objects can be initialized in a *@BeforeEach* method

Objects can be deallocated or reset in an *@AfterEach* method

Basic Template for Writing Test

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class TemplateTests {


    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

}
```

Simple Example



Simple Example

```
Public class Add2nos
{
   static public int add (int x, int y)
   {
    return x + y;
   }
}
```

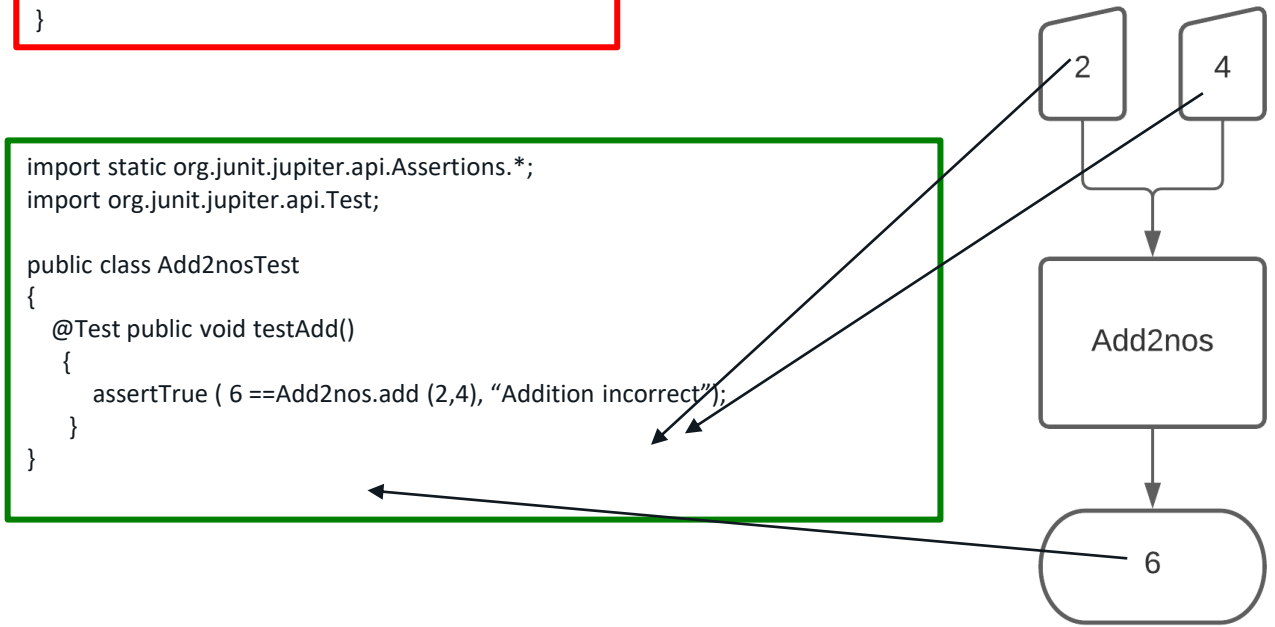
```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Add2nosTest
{
   @Test public void testAdd()
    {
        assertTrue ( 6 ==Add2nos.add (2,4), “Addition incorrect”);
     }
}
```

Simple Example

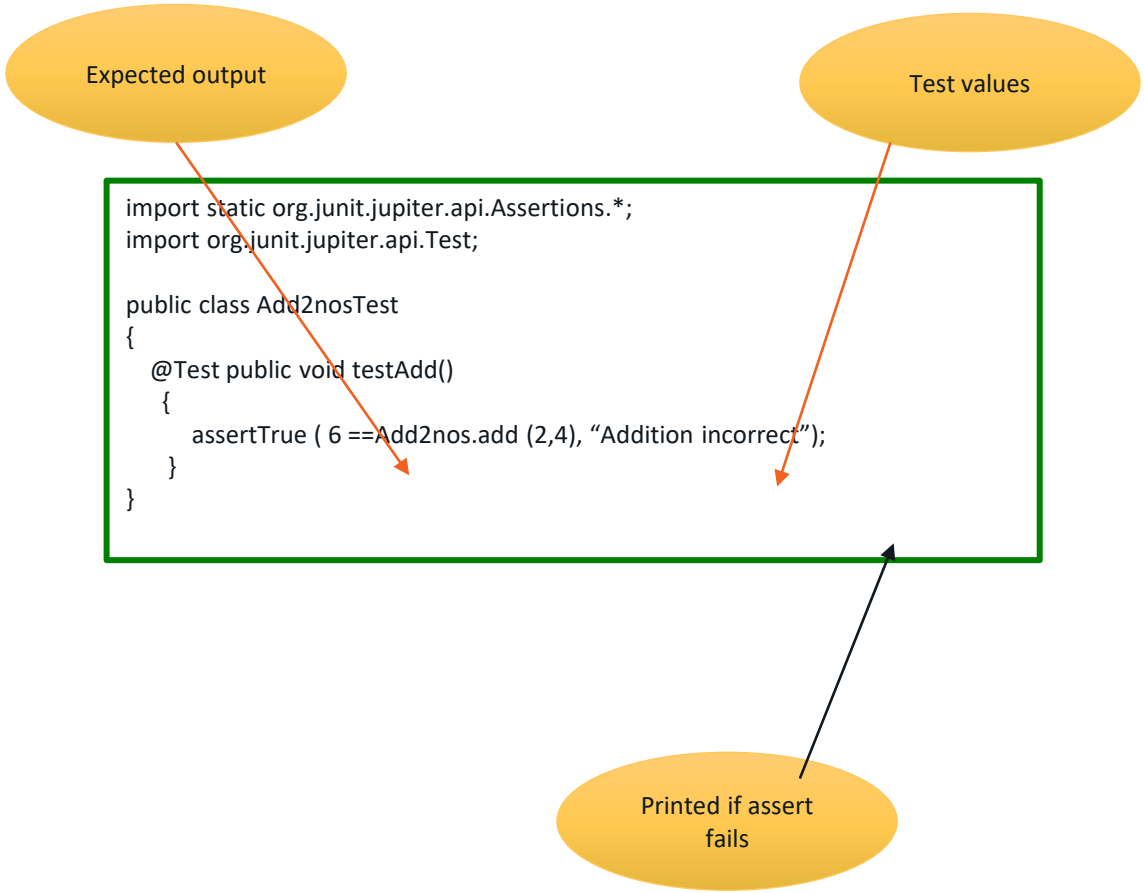
```
Public class Add2nos
{
   Static public int add (int x, int y)
   {
    return x + y;
   }
}
```

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class Add2nosTest
{
   @Test public void testAdd()
    {
        assertTrue ( 6 ==Add2nos.add (2,4), “Addition incorrect”);
     }
}
```

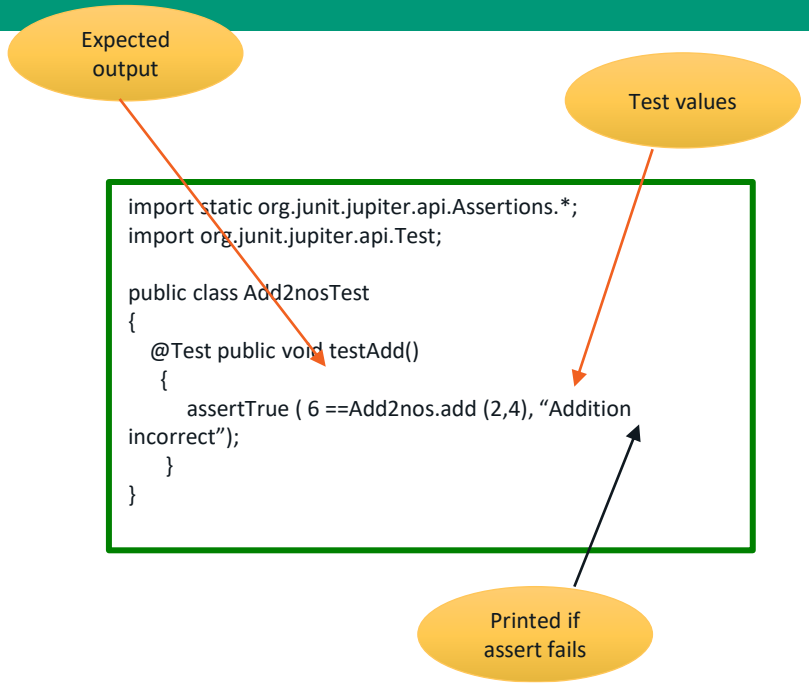
Test case #1:
Input: 2, 4
Expected output: 6



Simple Example

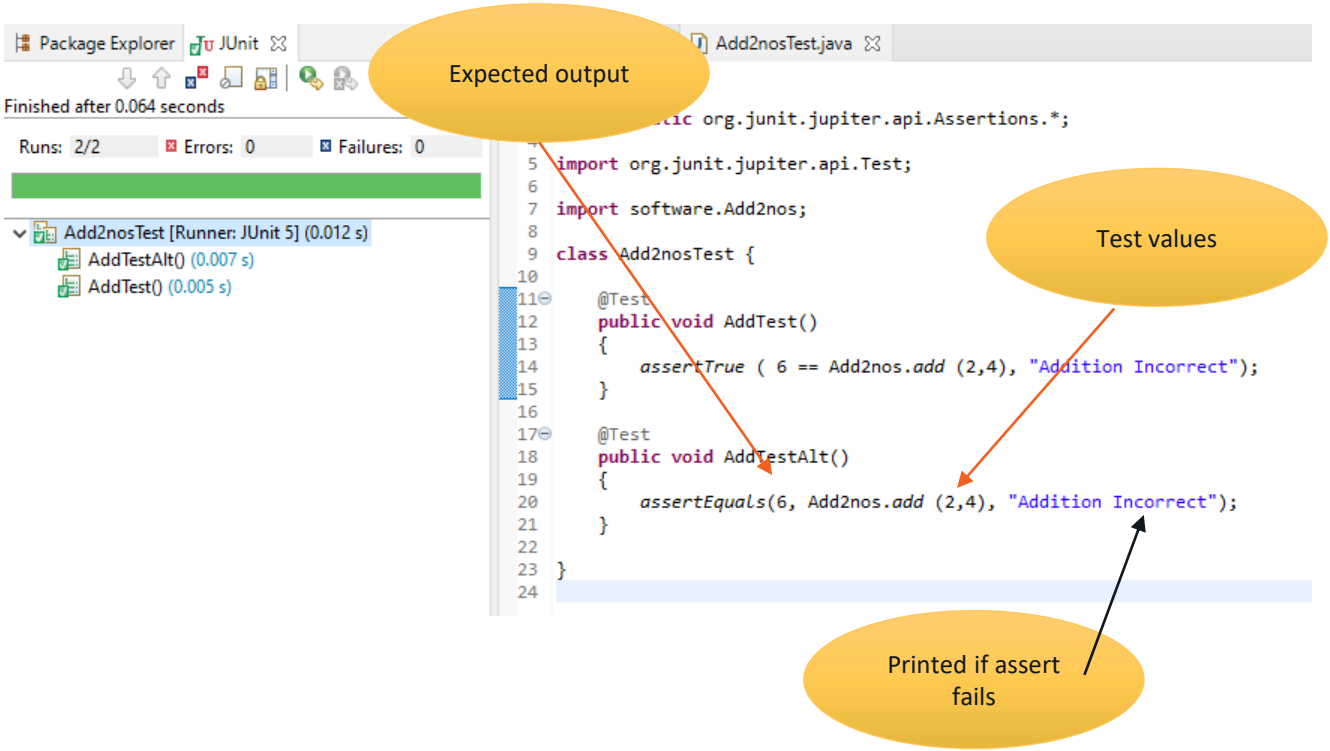


Simple Example



Test id	Test Objective	Technique	Expected Input	Expected Output
Add2nos_1	Valid input returns the correct value	E	2,4	6

Simple Example Alternate



Exception Testing – assertThrows( )

Junit 5 introduced a static method for dealing with exceptions – assertThrows( ).

There are three overloaded versions.

- *static T assertThrows(Class expectedType, Executable executable)*
  - Asserts that execution of the supplied executable throws an exception of the expectedType and returns the exception.
- *static T assertThrows (Class expectedType, Executable executable, String message)*
  - Asserts that execution of the supplied executable throws an exception of the expectedType and returns the exception.
- *static T assertThrows (Class expectedType, Executable executable, Supplier messageSupplier)*
  - Asserts that execution of the supplied executable throws an exception of the expectedType and returns the exception.

If no exception is thrown, or if an exception of a different type is thrown, this method will fail.

Exception Testing – Examples

Example1:

```
1  package tests;
2
3  import static org.junit.jupiter.api.Assertions.*;
4  import org.junit.jupiter.api.Test;
5  import org.junit.jupiter.api.DisplayName;
6
7  import software.Add2nos;
8
9
10 class ExceptionExampleOne
11 {
12     @Test
13     @DisplayName("No argument supplied")
14     public void testAddZeroArgument()
15     {
16         int [] numbersToAdd = null;
17         assertThrows(NullPointerException.class, ()-> Add2nos.add(numbersToAdd));
18     }
19 }
```

Line 13 – display the message for the exception

Line 17 – the call to assertThrows has two arguments:

- The exception class that is expected to be thrown
- An anonymous function (or lambda) that is to be called.

Execution of line 17, calls the anonymous function, resulting in an exception thrown. The test succeed.

The assertThrows() returns a caught Throwable.

Exception Testing – Examples

To verify a NullPointerException is thrown:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;

class ExceptionTest1 {

    @Test
    @DisplayName("throw the NullPointer exception")
    void  ThrowNull_PtException() {
        assertThrows(NullPointerException.class,() -> { throw new NullPointerException(); });
    }
}
```

Exception Testing – Examples

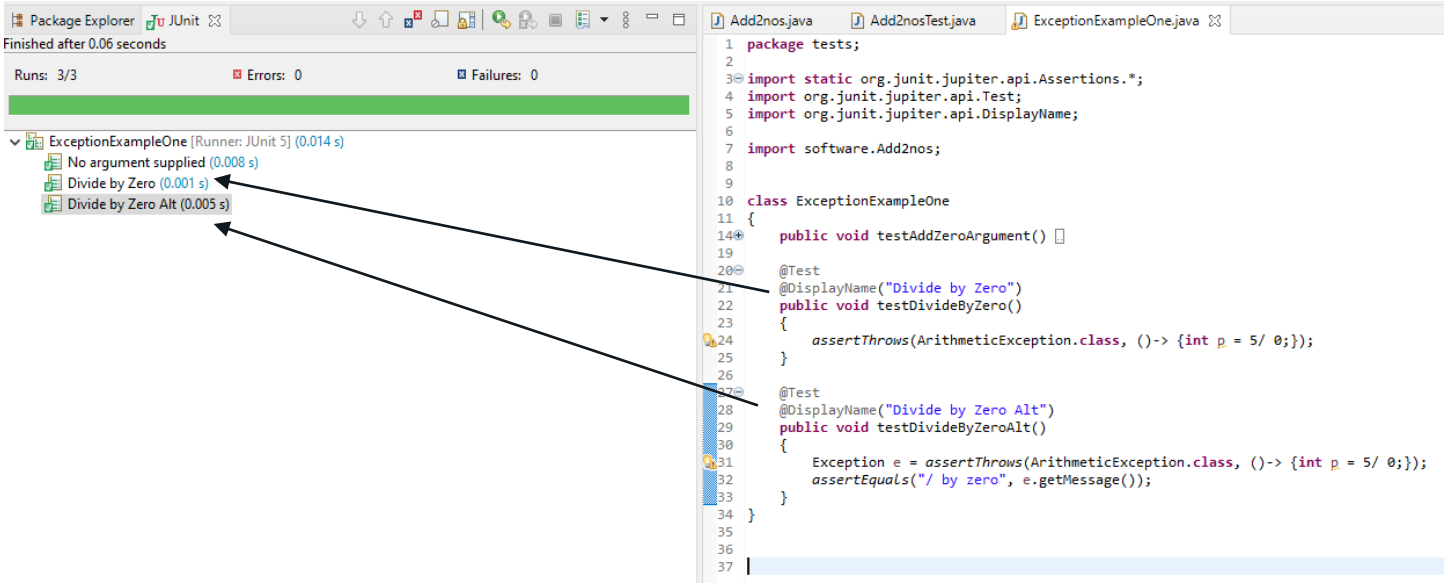
Example – Arithmetic Exception

Add2nos.java | Add2nosTest.java | ExceptionExampleOne.java

```
1  package tests;
2
3  import static org.junit.jupiter.api.Assertions.*;
4  import org.junit.jupiter.api.Test;
5  import org.junit.jupiter.api.DisplayName;
6
7  import software.Add2nos;
8
9
10 class ExceptionExampleOne
11 {
14     public void testAddZeroArgument() ..
19
20     @Test
21     @DisplayName("Divide by Zero")
22     public void testDivideByZero()
23     {
24         assertThrows(ArithmeticException.class, ()-> {int p = 5/ 0;});
25     }
26
27     @Test
28     @DisplayName("Divide by Zero Alt")
29     public void testDivideByZeroAlt()
30     {
31         Exception e = assertThrows(ArithmeticException.class, ()-> {int p = 5/ 0;});
32         assertEquals("/ by zero", e.getMessage());
33     }
34 }
35
36
37
```

Exception Testing – Examples

Example – Arithmetic Exception



Parameterized Tests

Allows testing of combinations of parameters by executing the same test method with different values – that is running a test multiple times with different arguments.

Junit 5 – supports multiple data-set source types: Examples include:

- @valueSource (example-- specify a String array as the source of arguments as in the example below)
- Example (Taken from Junit5 User Guide):

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
 void palindromes(String candidate) {
 assertTrue(isPalindrome(candidate));
}
```

- @CsvFileSource – use CSV files from the classpath. Each line from the fileresults in one invocation of the parameterized test.

More info about source types: Check out info at:
https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

Parameterized Tests

In JUnit 5, you can inject the test values into each individual test method directly.

Steps to set up parameterized test–

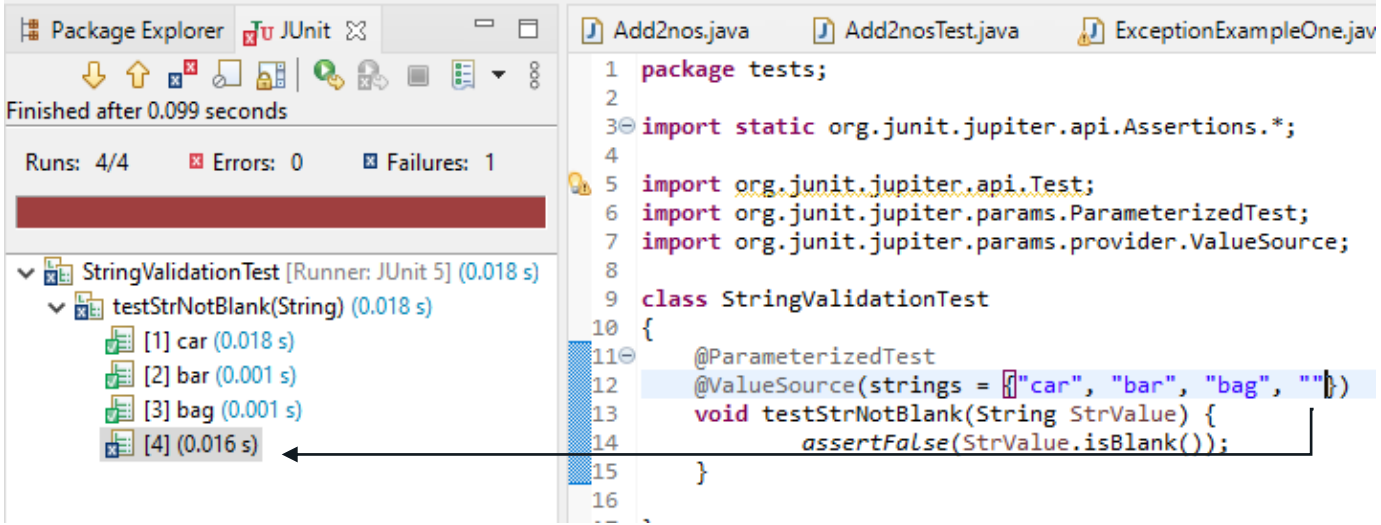
- first annotate a test method with @ParameterizedTest to indicate that this test is going to receive a series of test arguments.
- Second, define the source of the test arguments by providing one of a number of source annotations.

- Example:

```
@ParameterizedTest
@ValueSource(strings = {“car", "bar", "bag"})
void testStrNotBlank(String StrValue) {
      Assertions.assertFalse(StrValue.isBlank());
}
```

Parameterized Tests

In JUnit 5, you can inject the test values into each individual test method directly.



Parameterized Tests

Example:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

@DisplayName("Pass the method parameters using @ValueSource annotation")
class ValueSourceTest {

    @DisplayName("Input a message to our test method")
    @ParameterizedTest
    @ValueSource(strings = {"Hello", "World"})
    void PassMsg_AsParameter(String message) {
        assertNotNull(message);
    }
}
```

Parameterized Tests

Using @CsvSource annotation.

- Configure the test data using an array of String objects following the rules below:
  - One String object must contain all method parameters of one method invocation.
  - The different method parameters comma separated.
  - The values found from each line must follow the same order as the method parameters in the test method.

Parameterized Tests – Using @CsvSource

Example:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

@DisplayName("pass the method parameters using the @CsvSource annotation")
class CsvSourceTest {

    @DisplayName("To calculate the correct total")
    @ParameterizedTest(name = "{index} => p={0}, q={1}, total={2}")
    @CsvSource({
            "2, 1, 3",
            "2, 5, 8"
    } )
    void total(int p, int q, int total) {
        assertEquals(total, p + q);
    }
}
```

Parameterized Tests – Using @CsvSource

Example -- load test data from CSV file instead:

Steps:

- create a CSV file that contains the test data. When adding test data to the created CSV file, the following rules must be used:
  - One line must contain all method parameters of one method invocation.
  - The different method parameters must be comma separated.
  - The values on each line must follow the same order as the method parameters of the test method.
- put this CSV file to the classpath.

Example entries in CSV file:

```
2,2,4
3,5,9
5,6,11
7,8,15
```

Parameterized Tests – Using @CsvSource

Example -- load test data from CSV file instead:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

@DisplayName("pass the method parameters using the @CsvSource annotation – via
CSVfile")
class CsvFileSourceTest {

    @DisplayName("To calculate the correct total")
    @ParameterizedTest(name = "{index} => p={0}, q={1}, total={2}")
    @CsvFileSource(resources = “/test-sumdata.csv”) /* info where the location /name of file)*/

     void total(int p, int q, int total) {
         assertEquals(total, p + q);
    }
}
```

Running JUnit

Running JUnit 5 tests in an IDE like Eclipse

Running JUnit5 tests with Maven or Gradle

Running JUnit 5 tests from Command Line

- Use the *ConsoleLauncher* - a command-line Java application that launch the JUnit Platform from the console. It can be used to execute JUnit Vintage and JUnit Jupiter tests and print test execution results to the console.
- Example Template

Test Runner Class - Example

TestRunner.java

```
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(TestRJunit.class);

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

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

TestRJunit.class – the file with the tests that you want to run. If you have another file – just replace with the name of that file

Executing a Test Suite

In Junit 5 – the only built-in support for test suite is via JUnitPlatform Runner – executed using Junit 4

Implication → you can technically employ the JUnitPlatform runner to execute a suite using JUnit 5, and the tests will execute. However, the reporting and display in an IDE will be suboptimal.

In Junit 5 the *@nested* annotation allows you to group related Junit tests together. while allowing you to initialize them differently to simulate different runtime conditions.

School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 3e: Test Automation In Detail

CSI3105:
Software Testing

Ending notes on automated testing

From the Ropota Andrei / Automated Testing slides

Don‘t worry too much about these slides

Refer to these when you are starting your report for the group assignment

Automated testing advantages

Costs and efficiency

- Detection of the errors that reached production phase (with regression tests)
- Multiple users simulation
- Reusable of the old scripts -> creation of the new scripts is reduce
- automatic execution of performance tests in the beginning of the production ->less costs for improving the performance

Time economy

- quick analysis in case of changing of environment parameters
- short duration of the testing cycles
- better estimation for test planning
- a large number of tests can be executed over night
- quick generation of testing preconditions

Quality increase

- automatic compare of results
- more consistent results due to repeating tests

Limitation of Automated Testing

Most of the times an Automated Testing system can't tell if something "looks good" on the screen or when a pictogram or a window is not displayed well

There are a bunch of problems that can appear when trying to automate the testing process:

- Unrealistic expectations (e.g. expectation that automated tests will find a lot of errors)
- Poor testing experience
- Maintenance of automated tests

Automated testing will never replace definitely the manual testing

Tests that should not be automated are:

- tests that are executed very rare
- where the system is very unstable
- tests that can be verified easily manually but hardly automated
- tests that need physical interaction

Automated testing vs. Manual testing

Pros of **Automated testing**

- If a set of tests must be ran repeatedly, automation is a huge win
- It offers the possibility to run automation against code that frequently change to catch regressions
- Offers the possibility to add a large test matrix (e.g. different languages on different OS platforms)
- Automated tests can be run the same time on different machines, whereas manual tests must be run sequentially
- It offers more time for the test engineer to invoke greater depth and breadth of testing, focus on problem analysis, and verify proper performance of software following modifications and fixes
- Combined with the opportunity to perform programming tasks, this flexibility promotes test engineer retention and improves his morale

Cons of **Automated testing**

- Costs - Writing the test cases and writing or configuring the automate framework that is used costs more initially than running the test manually.
- Some tests can't be automated – manual tests are needed

Automated testing vs. Manual testing (2)

Pros of **Manual testing**

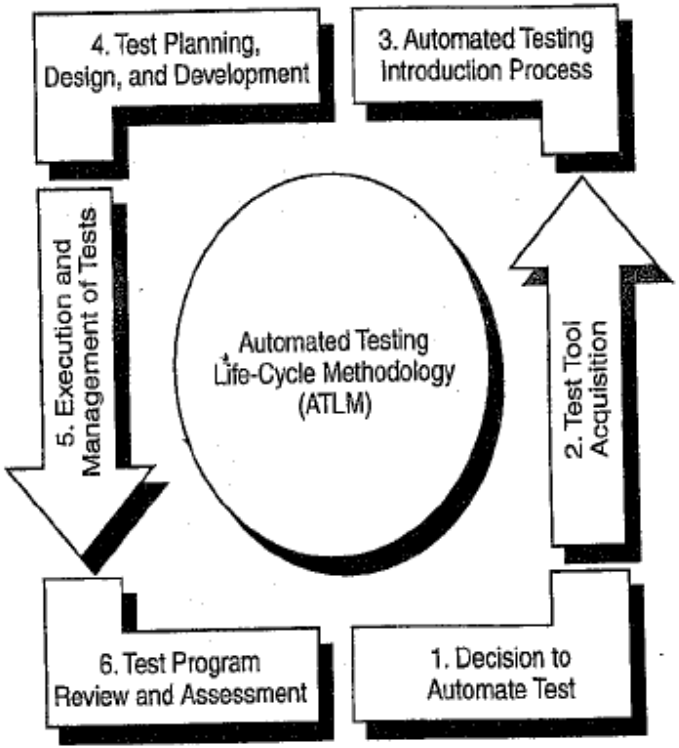
- If the test case runs once or twice most likely is a manual test. Less cost than automating it.
- It allows the tester to perform more ad-hoc (random tests). Experience has proven that more bugs are found via ad-hoc (Experience Based Testing) than via automation testing. And more time the testers spends playing with the feature, the greater are the chances of finding real user problems.

Cons of **Manual testing**

- Running tests manually can be very time consuming
- The manual tests are requiring more people and hardware
- Each time there is a new build, the tester must rerun all required tests - which after a while would become very boring and tiresome.

Automated Test Life-Cycle Methodology (ATLM)

1. Decision to Automate Test
2. Test Tool Acquisition
3. Automated Testing Introduction Process
4. Test Planning, Design and Development
5. Execution and Management of Tests
6. Test Program Review and Assessment



Decision to Automate Test: Overcoming False Expectations

EXPECTATION	REALITY
Automatic test plan generation	No tool can create automatically a comprehensive test plan
Test tool fits all	No single tool can support all operating systems environments and programming languages
Imminent test effort reduction	Initial use of an automated test tool can actually increase the test effort
Tool ease of use	Using an automated tool requires new skills, additional training is required
Universal application of test automation	Not all the tests required for a project can be automated (e.g. some test are physically impossible; time or cost limitation)
100% test coverage	It is impossible to perform an exhaustive testing of all the possible inputs (simple or combination) to a system

Decision to Automate Test: Benefits of Automated Testing

Production of a reliable system	Improved requirements definition Improved performance/load/stress testing Improved partnership with development team Improved system development life cycle
Improvement of the quality of the test effort	Improved build verification testing (smoke testing) Improved regression testing Improved multiplatform/software compatibility testing Improved execution of the repetitive tests Improved focus on advanced test issues Execution of tests that manual testing can't accomplish Ability to reproduce software defects After-hours testing
Reduction of test effort and minimization of schedule	

Test Tool Acquisition

Business analysis phase	business modeling tools configuration management tools defect tracking tools
Requirements definition phase	requirements management tools requirements verifiers tools
Analysis and design phase	database design tools application design tools
Programming phase	syntax checkers/debuggers memory leak and run-time error detection tools source code or unit testing tools static and dynamic analyzers
Testing phase	test management tools network testing tools GUI testing tools (capture/playback) non-GUI test drivers load/performance testing tools environment testing tools

Identify which of the various tool types suit the organization system environment, considering:

- the group/department that will use the tool;
- the budget allocated for the tool acquisition;
- the most/least important functions of the tool etc.

Choose the tool type according to the stage of the software testing life cycle

Evaluate different tools from the selected tool category

Hands-on tool estimation – request product demonstration (evaluation copy)

Following the conclusion of the evaluation process, an evaluation report should be prepared

Automated Testing Introduction Process – Test Process Analysis

Process review	• Test process characteristics (goals, strategies, methodologies) have been defined and they are compatible with automated testing • Schedule and budget allows process implementation • The test team is involved from the beginning of SDLC
Test goals	• Increase the probability that application under test will behave correctly under all circumstances • Increase the probability that application meets all the defined requirements • Execute a complete test of the application within a short time frame
Test objectives	• Ensure that the system complies with defined client and server response times • Ensure that the most critical end-user paths through the system perform correctly • Incorporate the use of automated test tools whenever feasible • Perform test activities that support both defect prevention and defect detection • Incorporate the use of automated test design and development standards to create reusable and maintanable scripts
Test strategies	• Defect prevention (early test involvement, use of process standards, inspection and walkthroughs) • Defect detection (use of automated test tools, unit/integration/system/acceptance test phase)

Test Planning

The test planning element of the ATLM incorporates the review of all activities required in the test program

It ensures that testing processes, methodologies, techniques, people, tools, schedule and equipment are organized and applied in an efficient way

Key elements: planning associated with project milestone events, test program activities and test program-related documentation.

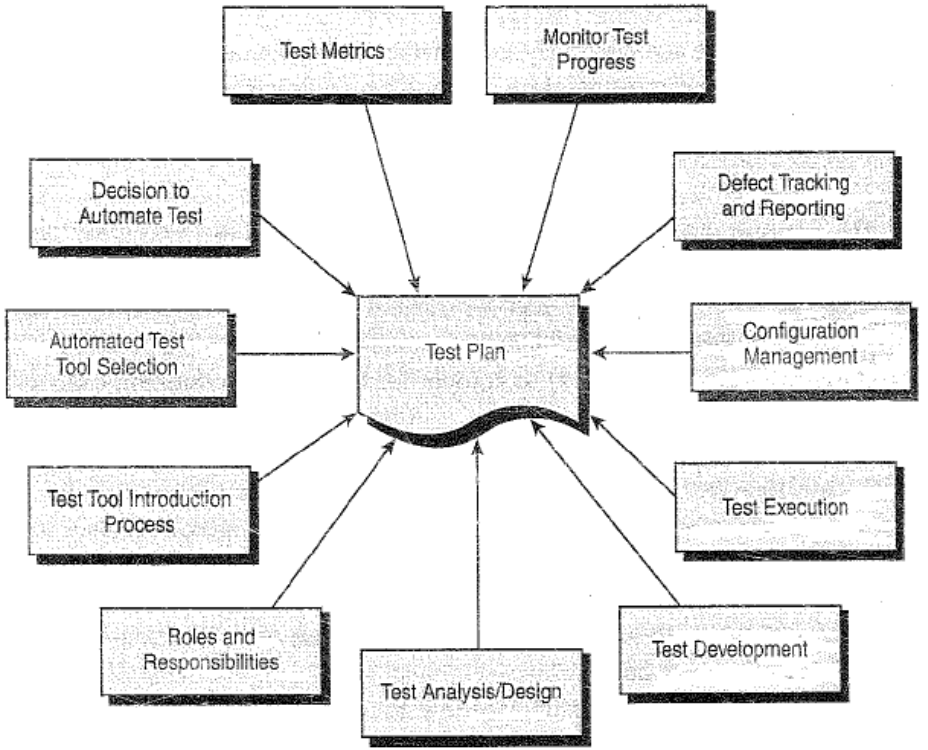
The following must be accomplished:

- the technical approach for these elements is developed;

- personnel are assigned

- performance timelines are specified in the test program schedule.

Test planning is not a single event, but rather a process. It is the document that guides test execution through to a conclusion, and it needs to be updated frequently to reflect any changes.

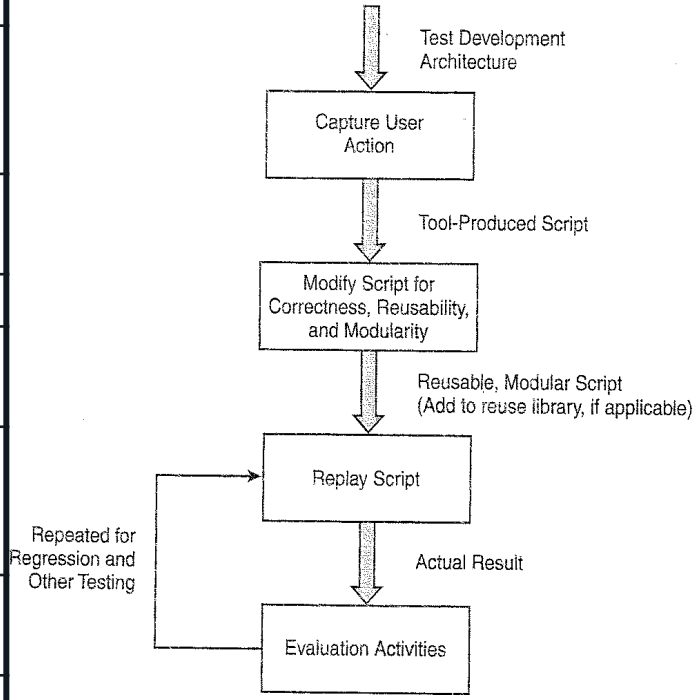


Test Design

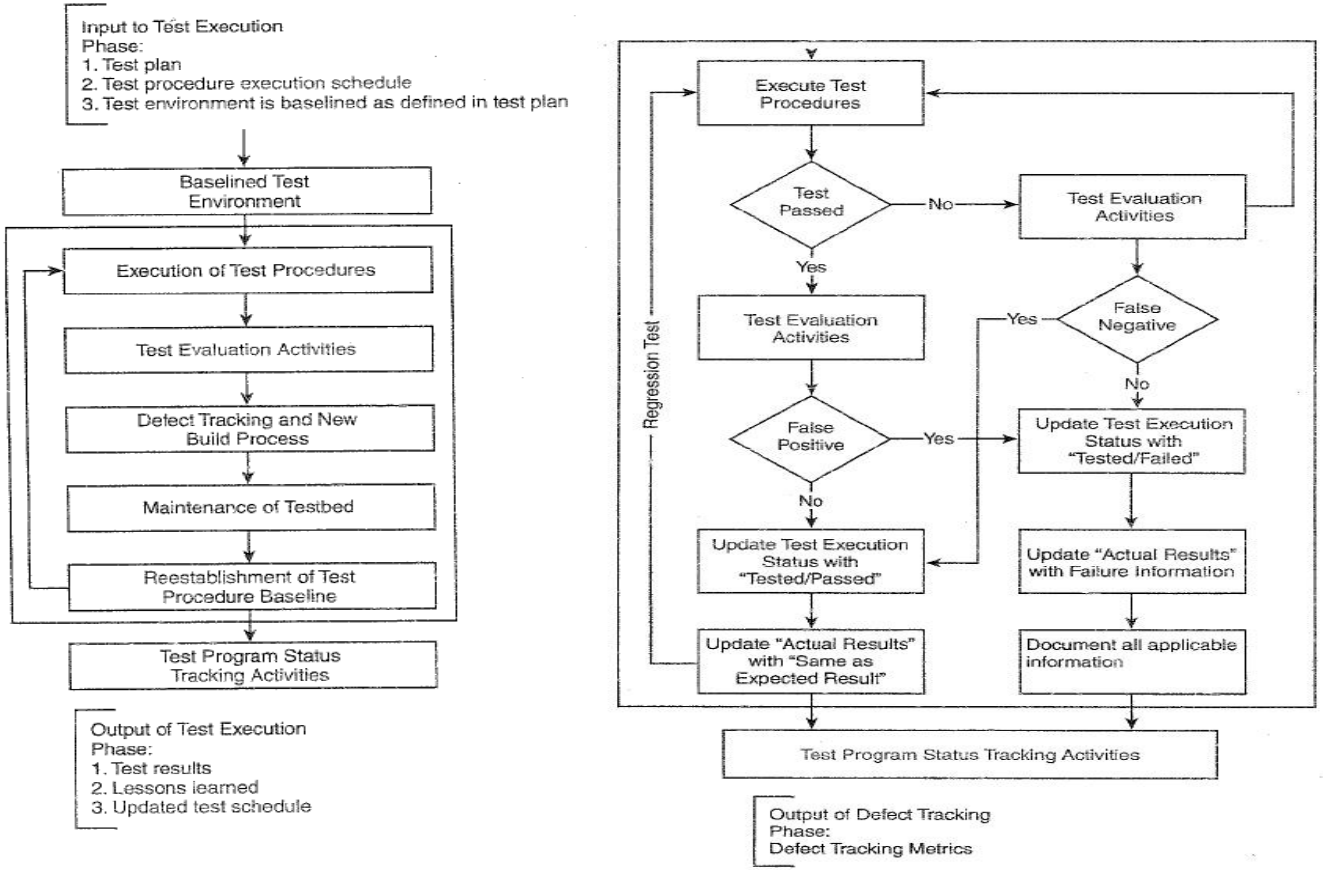
Black-Box Test Techniques	Automated Test Tools
Random Testing	GUI Test Tools
Regression Testing	GUI/Server Test Tools
Stress/Performance Testing	Load Test Tools
Security Testing	Security Test Tools
Data Integrity Testing	Data Analysis Tools
Configuration Testing	Multiplatform Test Tools
Functional Testing	Load/GUI/Server Test Tools
User Acceptance Testing	GUI Test Tools
Usability Testing	Usability Measurement Tools
Alpha/Beta Testing	Load/GUI/Server Test Tools
Boundary Value Analysis	Develop program code to perform tests
Backup and Recoverability Testing	Load/GUI/Server Test Tools

Test Development

Development Guideline Topics	Description
Design-to-Development Transition	Specify how design and setup activities will be translated into test development action
Reusable Test Procedures	Test procedures need to be reusable for highest test program return on investment
Data	Avoid hard-coding data values into scripts
Capture/Playback	Outlines on how to apply the use of capture/playback recording
Maintainable Test Procedures	A test procedure whose defects are easy to remove and can easily be adapted to meet new requirements
Test Script Documentation	Test script documentation is important for test procedure maintainability
Naming Standards	Defines the standard naming convention for test procedures
Modularity	Guidelines for creating modular test scripts



Test Execution



References

- A. P. Mathur, *Foundations of Software Testing, 2nd Edition*, Pearson Education, 2014.

P. Ammann & J. Offutt, *Introduction to Software Testing*, 2nd Edition, Cambridge University Press, 2008.

User Guide for Junit 5 : *https://junit.org/junit5/docs/current/user-guide/*