

# CSIT314 Software Development Methodologies



Verification & Validation and  
Test-driven development

# We have learned that ...

---

- ❑ Developing a software system involves a number of activities:
  - Planning
  - Requirements analysis
  - Design
  - Implementation
  - Verification and validation
  - Maintenance and evolution
  
- ❑ We now focus on one particular activity type:  
verification and validation

# This lecture

---

- ❑ Static and dynamic verification
- ❑ Software inspections
- ❑ Unit testing, Integration testing, and System testing
- ❑ White box vs. black box testing
- ❑ Test-driven development

# Verification vs. Validation

---

## □ Verification:

- "Are we building the product right?"
- The software should conform to its specification.

## □ Validation:

- "Are we building the right product?"
- The software should do what the user really requires.

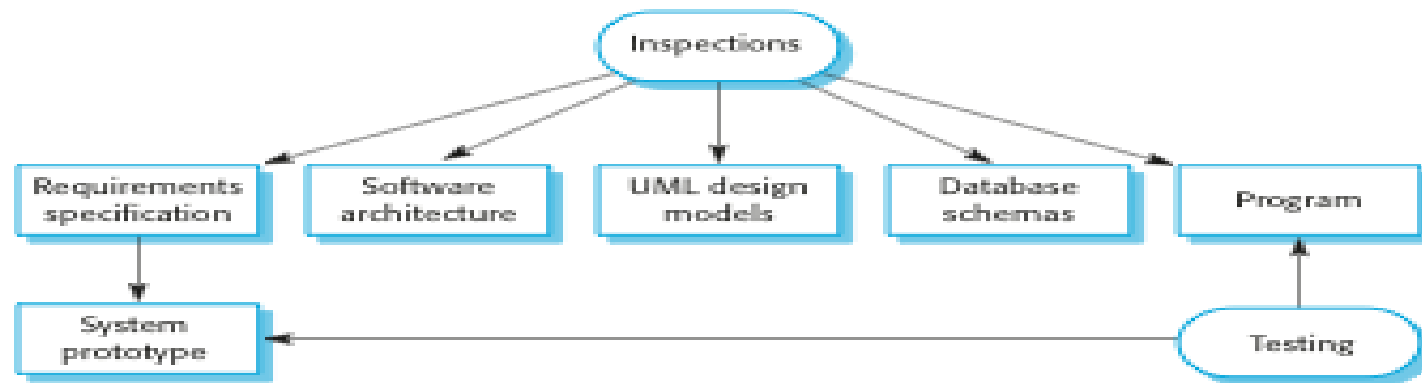
# The V & V process

---

- ❑ Is a whole life-cycle process - V & V must be applied during each phase of the software development process.
- ❑ Has two principal objectives
  - To discover and rectify defects in a system
  - To assess whether or not the system is usable in an operational situation.

# Static and Dynamic Verification

---



## ❑ *Software inspections*

- Concerned with **analysis of the static** system representation to discover problems (static verification)

## ❑ *Software testing*

- Concerned with **executing and observing** product behaviour (dynamic verification)
  - ❑ The system is executed with test data and its operational behaviour is observed

# Software inspections

---

- ✧ These involve people examining the source representation with the aim of discovering anomalies and defects.
- ✧ Inspections do 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.

# Software inspections

## Design review

Reviewer: Joe Bloggs  
Team: DT1  
Part/Level: WCMS-HLD-1-03  
Cycle: 1

Date: 10 March 2003  
Owner: M. Bean  
Moderator: M. Python

### Time Information

Start Time:	10:15
End Time:	11:40
Interruption:	25 min
Delta Review Time:	60 min

### Defect Information

[illegible]

## Summary

Number of major defects: 8  
Major / Total defects:  $8 / 10 = 80\%$

Number of minor defects: 2

Size of reviewed artefact: 15 pages

Rate of review: 15 pages / hour

Defect detection rate (maj/hr): 8 defects / hour



# Software inspections

## Code review

Fault class	Inspection check
Data faults	<p>Are all program variables initialised before their values are used?</p> <p>Have all constants been named?</p> <p>Should the lower bound of arrays be 0, 1, or something else?</p> <p>Should the upper bound of arrays be equal to the size of the array or Size -1?</p> <p>If character strings are used, is a delimiter explicitly assigned?</p>
Control faults	<p>For each conditional statement, is the condition correct?</p> <p>Is each loop certain to terminate?</p> <p>Are compound statements correctly bracketed?</p> <p>In case statements, are all possible cases accounted for?</p>
Input/output faults	<p>Are all input variables used?</p> <p>Are all output variables assigned a value before they are output?</p>
Interface faults	<p>Do all function and procedure calls have the correct number of parameters?</p> <p>Do formal and actual parameter types match?</p> <p>Are the parameters in the right order?</p> <p>If components access shared memory, do they have the same model of the shared memory structure?</p>
Storage management faults	<p>If a linked structure is modified, have all links been correctly reassigned?</p> <p>If dynamic storage is used, has space been allocated correctly?</p> <p>Is space explicitly de-allocated after it is no longer required?</p>
Exception management faults	<p>Have all possible error conditions been taken into account?</p>

# Advantages of inspections

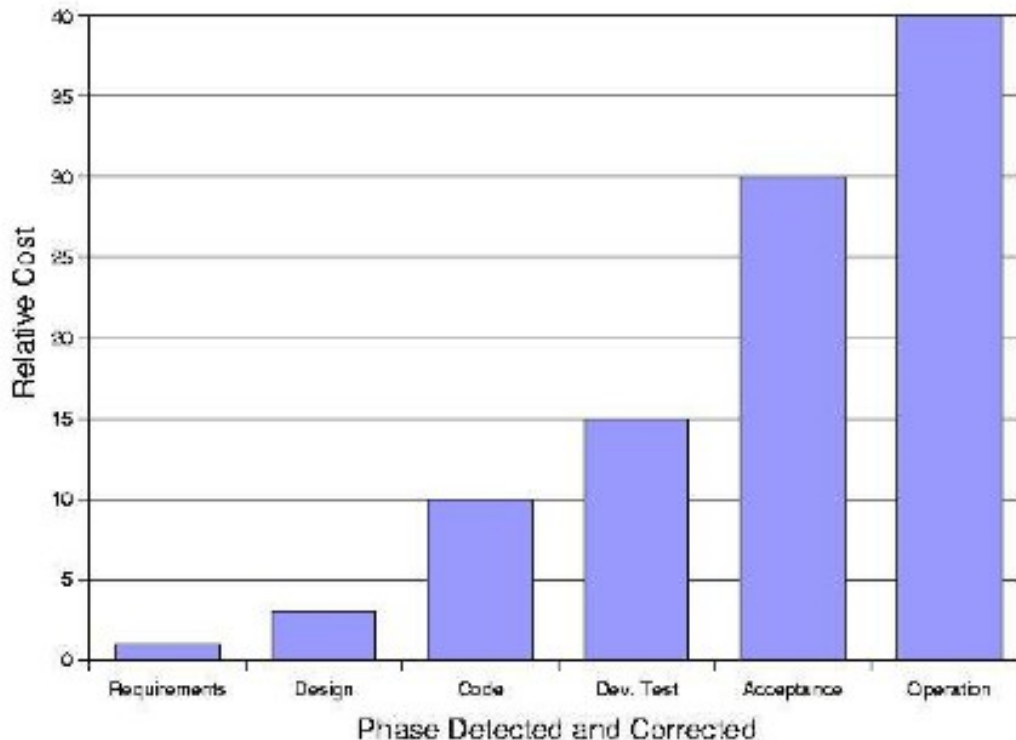
---

- ✧ 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.
- ✧ 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.
- ✧ With the reuse domain and programming knowledge, experienced reviewers are likely to have seen the types of error that commonly arise.

# Advantages of inspections (cont.)

---

- ❑ **Reviews provide a powerful way to improve quality by providing a means by which defects can be detected (and corrected) early in development.**



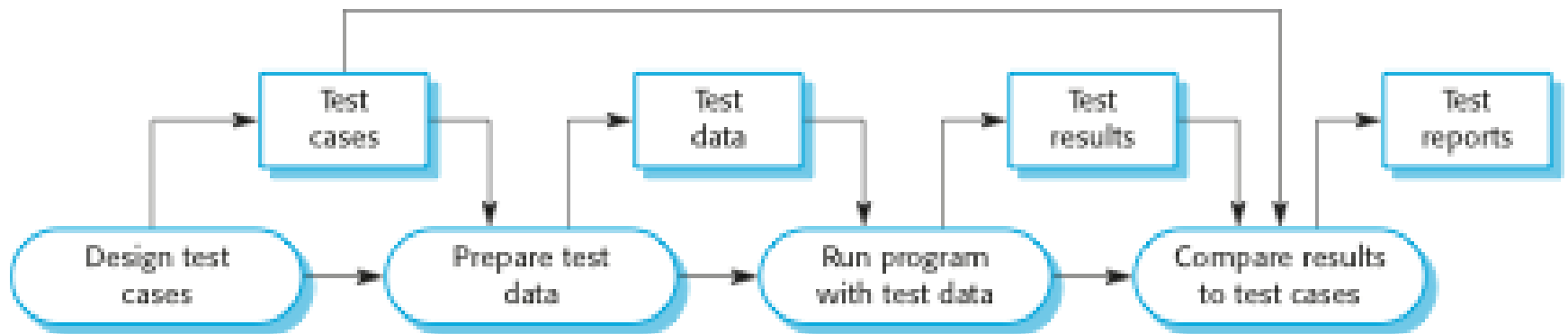
# Software Testing

---

- ❑ Testing is the process of executing a program with one of the following goals:
  - to force a program to work incorrectly
    - ❑ A good test is one that has a high probability of finding an, as yet, undiscovered error.
  - to demonstrate that the program works correctly
    - ❑ E.g. fault-based testing: to show that certain types of faults do not exist.
    - ❑ For example, to demonstrate that the software handles or avoids divide by zero correctly, the test data would include zero.
- ❑ Typically, testing takes about 30-40% of budget for system development

# A model of the software testing process

---



# Test Case Design

---

*Testing consists of a set of test cases which are systematically planned and executed*

- Each test case is uniquely identified
- Purpose of each test case is stated
- Expected results are listed in the test case and should include all outputs, resultant changes and error messages

# A Standard Test Matrix

---

ID	Test Case	Purpose	Expected Results	Actual Results

# Test case example

---

7. Test for call status	
<b>Actions</b>	a. Before placing a call b. On placing a call c. On the user ending the testing session by hanging-up the call d. On occurrence of any exception during the call
<b>Expected Result</b>	a. The call status must be "Normal Close" b. The call status must be "In Session" c. The call status must be "Normal Close" d. The call status must be "Exception Close"
<b>Pass/Fail</b>	
<b>Observations (if any)</b>	



# Who performs testing?

---

- Testing is conducted by two (or three) groups:
  - the software developer
  - (for large projects) an independent test group
  - the customer

# Development testing

---

- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
  - **Unit testing**, where individual program units or object classes are tested.
    - Unit testing should focus on testing the functionality of objects or methods.
  - **System testing**, where 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.

# Unit testing

---

- ✧ Unit testing is the process of testing individual components in isolation.
- ✧ Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods.

# Unit testing

## 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.

# Weather station testing

- ✧ Need to define test cases for **reportWeather**, **reportStatus**, **powerSave**, **remoteControl**, **reconfigure**, **restart** and **shutdown**.
- ✧ Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- ✧ For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

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

# Automated testing

---

- ✧ Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- ✧ In automated unit testing, you make use of a test automation framework (such as JUnit or cppUnit in this subject) to write and run your program tests.
- ✧ Unit testing frameworks provide generic test classes (e.g. `cppUnit::TestFixture`) that you extend to create specific test cases.
  - ✧ They can then run all of the tests that you have implemented and report, often through some GUI, on the success or otherwise of the tests.

# Unit test effectiveness

---

- ✧ 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.

# Quiz: true or false?

---

- ❑ Testing **cannot** show the absence of defects, it **can** only show that software defects are present.



# Testing Techniques

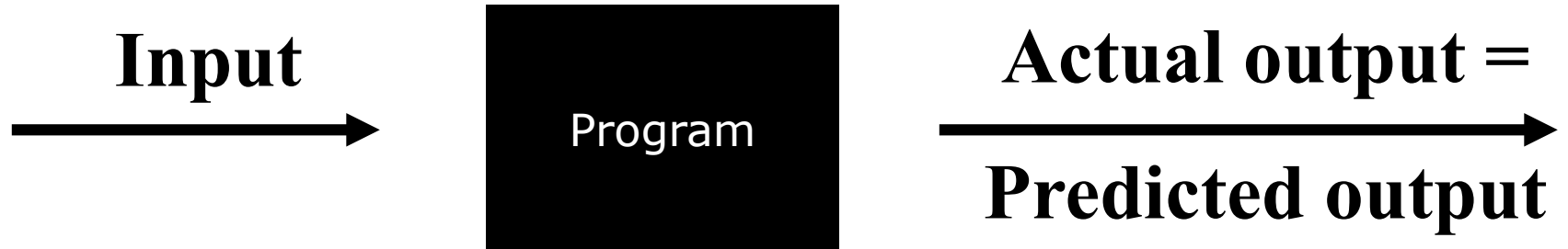
---

- Two different categories of test case design techniques are used:
  - ***White box Testing*** - examining the internal workings (i.e. source code) of each module.
  - ***Black box Testing*** - testing the product to see if it performs as specified

# Black Box Testing

---

**Black box tests** are designed to validate functional requirements without regard to the internal workings of the programs. Generally, a set of results is derived to ensure that modules produce correct results.



# Black Box Testing

## Systematic vs Random Testing

---

### □ Random (uniform):

- Pick possible inputs uniformly
- Avoids designer bias
  - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
- But treats all inputs as equally valuable

### □ Systematic (non-uniform):

- Try to select inputs that are especially valuable
- Usually by choosing representatives of classes that are apt to fail *often* or *not at all*

# Why not random?

- Compute the probability of selecting a test case that reveals the fault in line 19 by randomly sampling the input domain, assuming that type double has range  $-2^{31} \dots 2^{31}-1$ .

```
1  /** Find the two roots of  $ax^2 + bx + c$ ,  
2  * that is, the values of  $x$  for which the result is 0.  
3  */  
4  class Roots {  
5      double root_one, root_two;  
6      int num_roots;  
7      public roots(double a, double b, double c) {  
8          double q;  
9          double r;  
10         // Apply the textbook quadratic formula:  
11         // Roots =  $-b \pm \sqrt{b^2 - 4ac} / 2a$   
12         q = b*b - 4*a*c;  
13         if (q > 0 && a != 0) {  
14             // If  $b^2 > 4ac$ , there are two distinct roots  
15             num_roots = 2;  
16             r = (double) Math.sqrt(q) ;  
17             root_one = ((0-b) + r)/(2*a);  
18             root_two = ((0-b) - r)/(2*a);  
19         } else if (q==0) { // (BUG HERE)  
20             // The equation has exactly one root  
21             num_roots = 1;  
22             root_one = (0-b)/(2*a);  
23             root_two = root_one;  
24         } else {  
25             // The equation has no roots if  $b^2 < 4ac$   
26             num_roots = 0;  
27             root_one = -1;  
28             root_two = -1;  
29         }  
30     }  
31     public int num_roots() { return num_roots; }  
32     public double first_root() { return root_one; }  
33     public double second_root() { return root_two; }  
34 }
```

# Black Box Testing strategies

---

- ❑ There are 2 typical strategies of black box tests:
  - Equivalence Partitioning
  - Boundary Value Analysis

# Equivalence Partitioning

---

- ❑ A testing method that divides the input domain of a program into sets of data from which test cases can be derived.
- ❑ Equivalence partitioning strives to define a test case that uncovers a class of errors, thereby reducing the total number of test cases that must be developed.

# Equivalence Partitioning

---

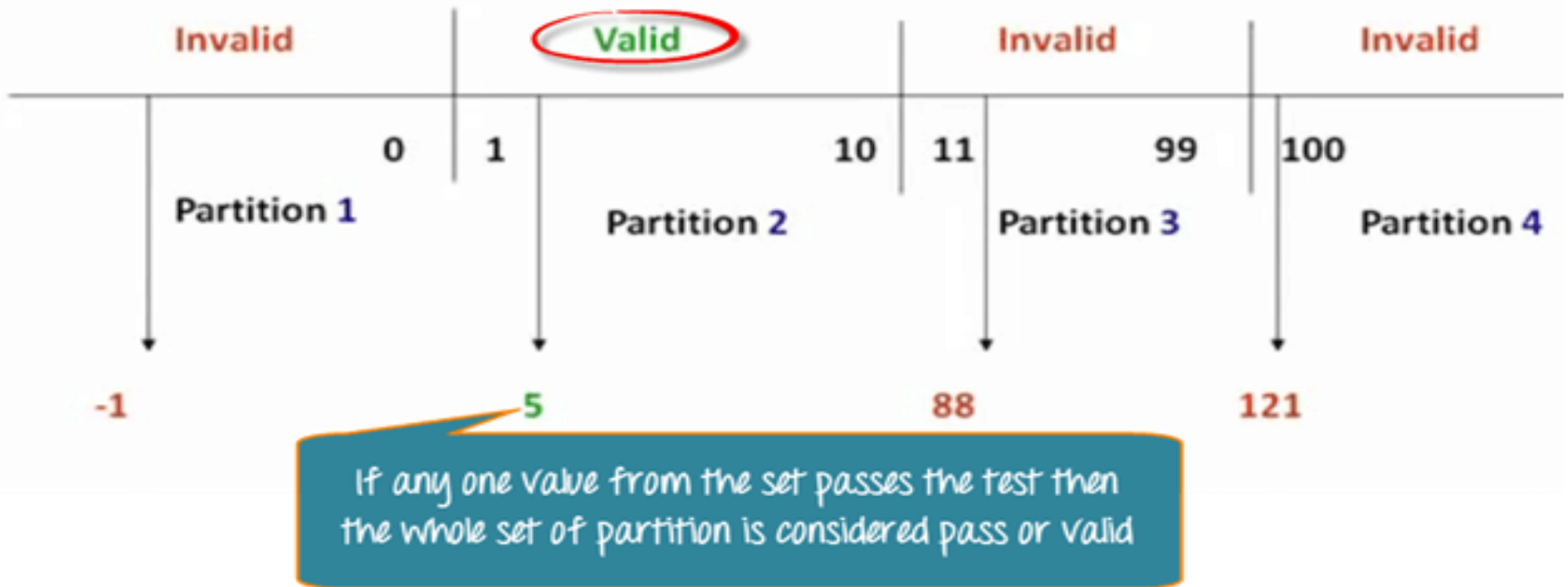
- ❑ Pizza values 1 to 10 is considered valid. A success message is shown.
- ❑ While value 11 to 99 are considered invalid for order and an error message will appear, "Only 10 Pizza can be ordered"



Order Pizza:

Source: <https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html>

# Equivalence Partitioning





# Boundary Value Analysis

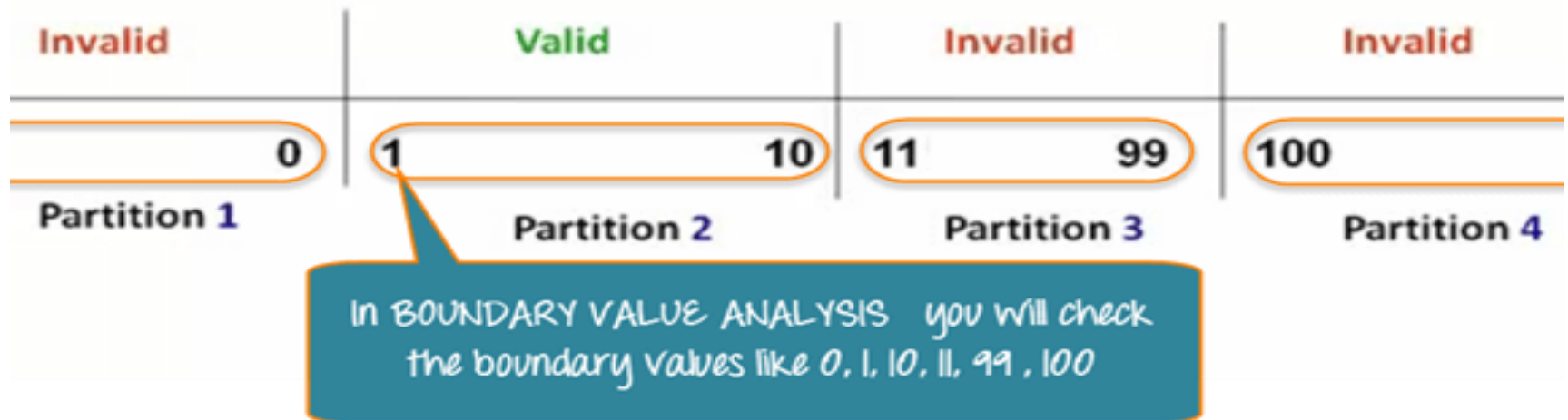
---

A test case design technique that complements equivalence partitioning.

- Rather than selecting any element of an equivalence set, it selects test cases at the edges of the set.
- Boundary value analysis leads to a selection of test cases that exercise boundary values.

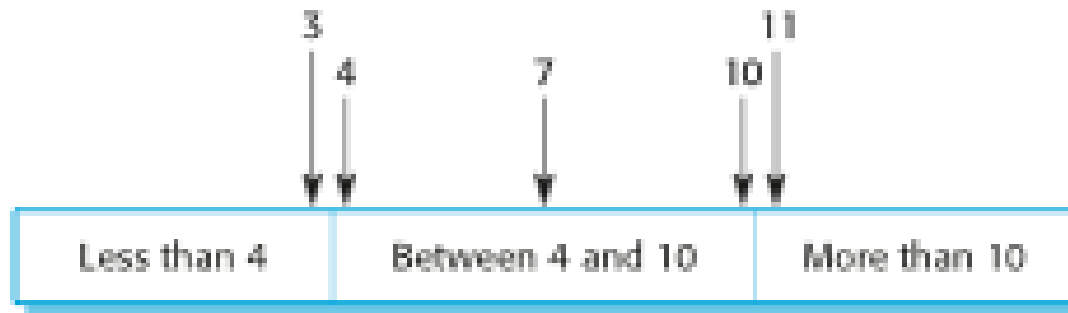
# Boundary Value Analysis

---

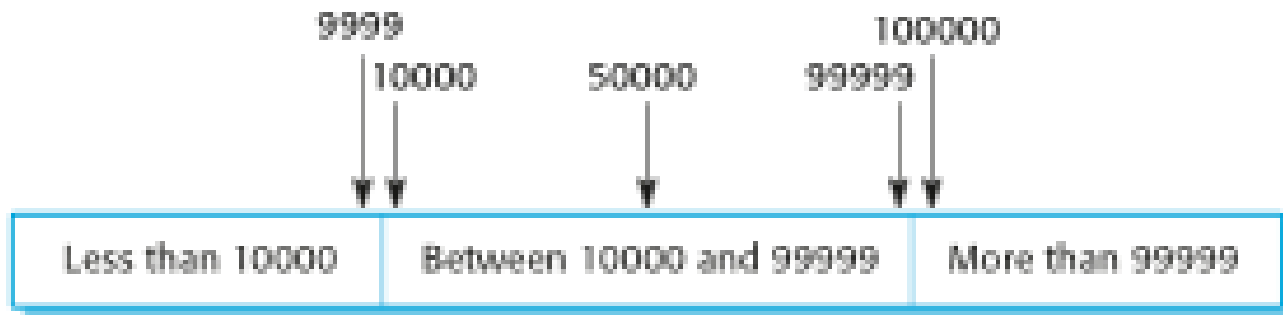


# Equivalence partitions

---



Number of input values

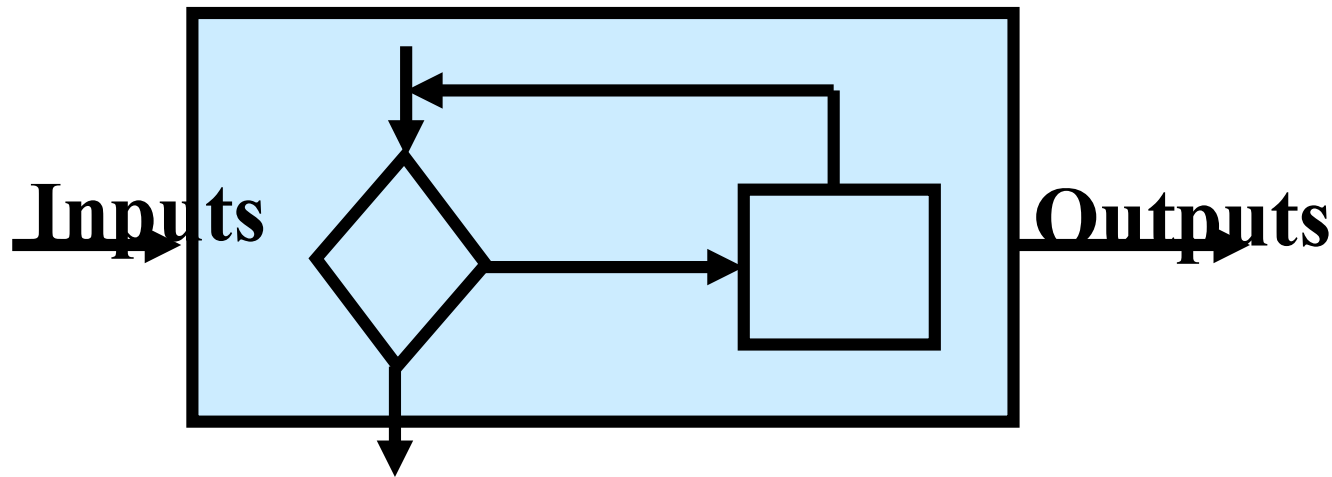


Input values

# Unit Testing - White Box Testing

---

- **White Box tests** focus on the program control structure.



# White Box Testing (cont.)

---

- ❑ White-box testing is applied to increase logic coverage
- ❑ Basic forms
  - Statement coverage
  - Decision (branch) coverage
  - Condition coverage
  - Path coverage

# White Box Testing (cont.)

---

```
premium = 500;
if (age < 25) && (sex == male) && !married
{
    premium += 500;
}
else
{
    if (married || (sex == female))
        premium -= 200;
    if (age > 45) && (age < 65)
        premium -= 100;
}
```

# White-box methods for internals-based tests

---

- ❑ Items to keep in mind during white box testing
  - Statement coverage: each statement is executed at least once
  - Decision (branch) coverage: each statement ...; each decision takes on all possible outcome at least once
  - Condition coverage: each statement...; each decision ...; each condition in a decision takes on all possible outputs at least once
  - Path coverage: each statement ...; all possible combinations of condition outcomes in each decision occur at least once

# Integration testing

---

- ❑ Tests complete systems or subsystems composed of integrated components.
- ❑ If **unit** testing has been effective, then faults that remain to be found in **integration** testing will be primarily **interface faults**.
  - Testing effort should focus on interfaces between units rather than their internal details.



# Systems Testing

---

- ❑ System tests encapsulate the environment as well as the product and include:
  - Function Testing
    - ❑ tests functions of the system, i.e. the functional requirements
  - Performance Testing
    - ❑ tests some of the non-functional requirements, e.g. reliability, availability, etc.
  - Acceptance Testing
    - ❑ performs validation testing on the system prior to handover to the customers. Most usual test phases here are *Alpha Testing* and *Beta Testing*

# Automated testing demo

---

- ▣ A demo of automated regression testing

<https://www.youtube.com/watch?v=wDpBDqOXRHc&feature=youtu.be>

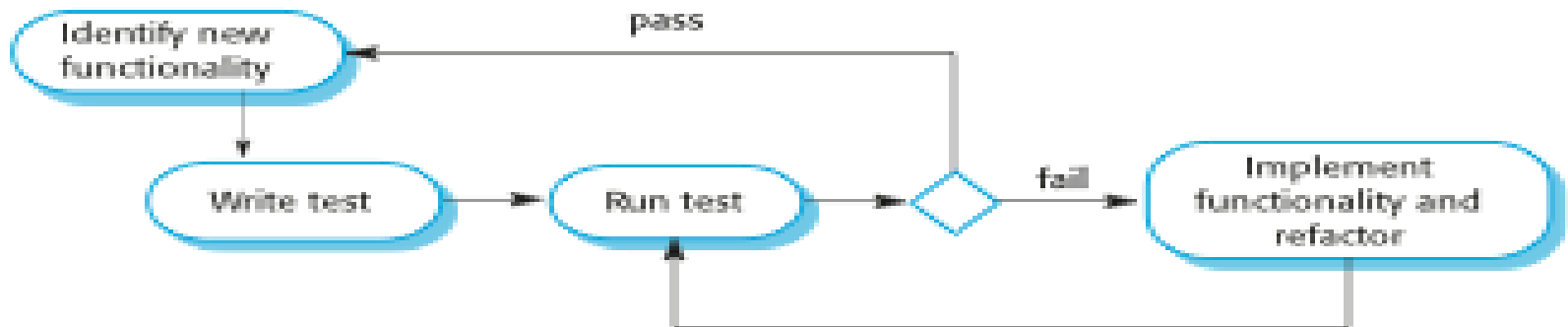
# Test-driven development

---

- ✧ Test-driven development (TDD) is an approach to program development in which you **interleave 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 other development processes.

# Test-driven development

---



- ✧ Start by identifying the new functionality that is required. This should normally be **small** and implementable in a few lines of code.
- ✧ Write a test for this functionality and implement this as an automated test.
- ✧ Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- ✧ Implement the functionality and re-run the test.
- ✧ 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 has at least one associated test so all code written has at least one test.

## ✧ Regression testing

- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- A regression test suite is developed incrementally as a program is developed.

## ✧ Simplified debugging

- When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

## ✧ System documentation

- The tests themselves are a form of documentation that describe what the code should be doing.

# Test-driven development with JUnit

---

- ❑ JUnit (<https://junit.org>) is a widely-used unit testing framework for Java
- ❑ Junit provide 4 important features:
  - Test fixtures
  - Test suites
  - Test runners
  - JUnit classes

# TestFixture

---

- ❑ Each test should be derived from the abstract class TestFixture
- ❑ Override two functions:
  - setup (using the annotation *@Before*)
    - ❑ operations that would be invoked prior to each individual test, e.g. create new clean instances of class ready for next test
  - teardown (using the annotation *@After*)
    - ❑ operations that would be invoked after each individual test, e.g. deletes instances

# TestSuite, TestRunner

---

## □ TestSuite

- Groups a number of tests (using TestFixture) so that they can all be run

## □ TestRunner

- Driver program



# JUnit Assert

---

- Assert:  
provide a set  
of assertion  
methods  
useful for  
writing tests.

**void assertEquals(boolean expected, boolean actual)**

Checks that two primitives/objects are equal.

**void assertTrue(boolean condition)**

Checks that a condition is true.

**void assertFalse(boolean condition)**

Checks that a condition is false.

**void assertNotNull(Object object)**

Checks that an object isn't null.

**void assertNull(Object object)**

Checks that an object is null.

**void assertSame(object1, object2)**

The assertSame() method tests if two object references point to the same object.

**void assertNotSame(object1, object2)**

The assertNotSame() method tests if two object references do not point to the same object.

**void assertEquals(expectedArray, resultArray);**

The assertEquals() method will test whether two arrays are equal to each other.

# Test driven development – step 1

---

- ❑ Define class Calculator
  - Performs simple calculations
- ❑ Method specification:
  - add(int x, int y) returns a sum of x and y

```
public class Calculator {  
  
    public int add(int x, int y) {  
        // no implementation yet  
        return 0;  
    }  
}
```

See the demo at

<https://documents.uow.edu.au/~hoa/teaching/SIM/TDD-demo.mp4>

# Test driven development – step 2

---

- Write a (JUnit) test

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

public class CalculatorTest {

    private Calculator testSubject;

    @Before
    public void setUp() throws Exception {
        this.testSubject = new Calculator();
    }

    @After
    public void tearDown() throws Exception {
        testSubject = null;
    }

    @Test
    public void testAdd() {
        assertEquals("Adding", 32, testSubject.add(12, 20));
    }
}
```

# Test driven development – step 2

---

- ▣ Write a (JUnit) test (cont.)

```
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(CalculatorTest.class);

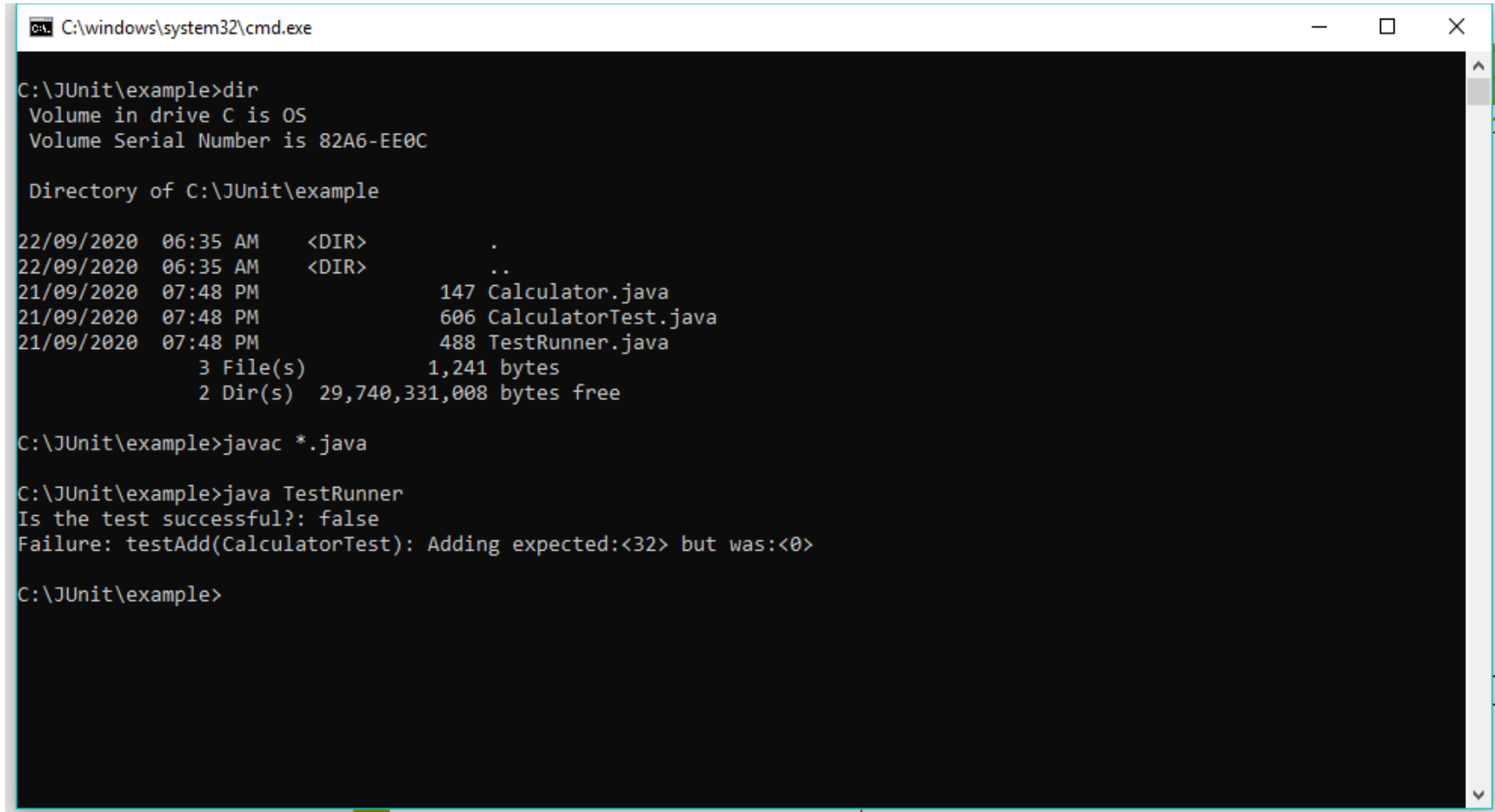
        System.out.println("Is the test successful?: " + result.wasSuccessful());

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

# Test driven development – step 3

---

## □ Run the test: fail or pass?



```
C:\windows\system32\cmd.exe

C:\JUnit\example>dir
Volume in drive C is OS
Volume Serial Number is 82A6-EE0C

Directory of C:\JUnit\example

22/09/2020  06:35 AM    <DIR>          .
22/09/2020  06:35 AM    <DIR>          ..
21/09/2020  07:48 PM                147 Calculator.java
21/09/2020  07:48 PM                606 CalculatorTest.java
21/09/2020  07:48 PM                488 TestRunner.java
               3 File(s)                1,241 bytes
               2 Dir(s)  29,740,331,008 bytes free

C:\JUnit\example>javac *.java

C:\JUnit\example>java TestRunner
Is the test successful?: false
Failure: testAdd(CalculatorTest): Adding expected:<32> but was:<0>

C:\JUnit\example>
```

# Test driven development – step 4

---

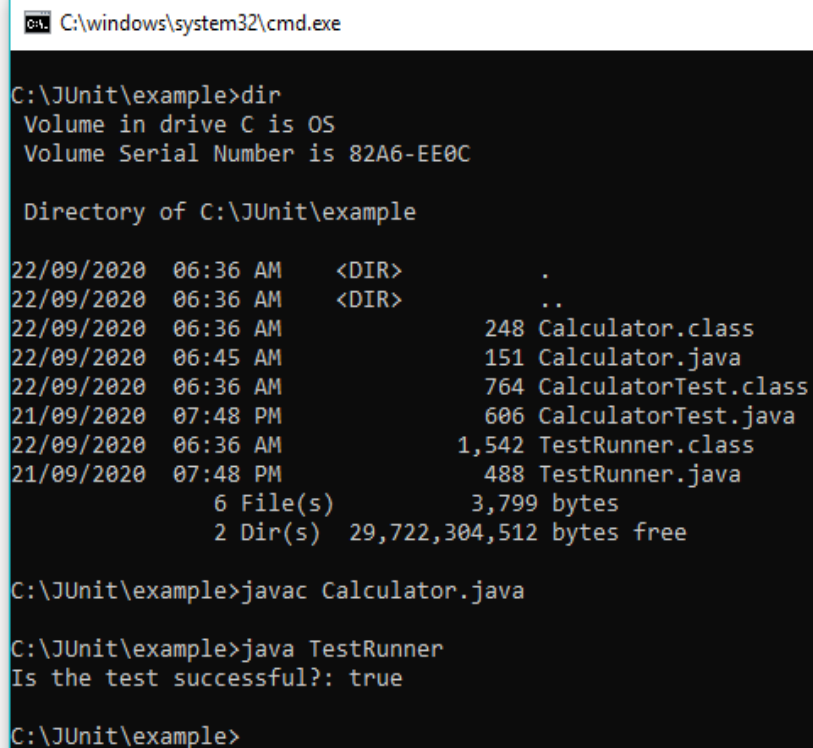
## □ Implement the function

```
public class Calculator {  
    public int add(int x, int y) {  
        return x + y;  
    }  
}
```

# Test driven development – step 5

---

## □ Run the test again: fail or pass?



A screenshot of a Windows command prompt window titled "C:\windows\system32\cmd.exe". The window shows the following commands and output:

```
C:\JUnit\example>dir
Volume in drive C is OS
Volume Serial Number is 82A6-EE0C

Directory of C:\JUnit\example

22/09/2020  06:36 AM    <DIR>          .
22/09/2020  06:36 AM    <DIR>          ..
22/09/2020  06:36 AM                248 Calculator.class
22/09/2020  06:45 AM                151 Calculator.java
22/09/2020  06:36 AM                764 CalculatorTest.class
21/09/2020  07:48 PM                606 CalculatorTest.java
22/09/2020  06:36 AM            1,542 TestRunner.class
21/09/2020  07:48 PM                488 TestRunner.java
               6 File(s)              3,799 bytes
               2 Dir(s) 29,722,304,512 bytes free

C:\JUnit\example>javac Calculator.java

C:\JUnit\example>java TestRunner
Is the test successful?: true

C:\JUnit\example>
```

# Test-driven development support

---

- ❑ JUnit is supported in many popular IDEs, e.g.:
  - IntelliJ IDEA  
<https://blog.jetbrains.com/idea/2016/08/using-junit-5-in-intellij-idea/>
  - Eclipse  
[https://www.tutorialspoint.com/junit/junit\\_plugin\\_with\\_eclipse.htm](https://www.tutorialspoint.com/junit/junit_plugin_with_eclipse.htm)
  - NetBeans  
<https://netbeans.org/kb/docs/java/junit-intro.html>
  - Visual Studio Code  
<https://code.visualstudio.com/docs/java/java-testing>



# Test-driven development support

---

- ❑ Automated unit testing frameworks for other languages:
  - NUnit is widely used unit-testing framework use for .NET languages <https://nunit.org/>
  - PHPUnit: unit testing tool for PHP <https://phpunit.de/>
  - cppUnit: unit testing for C++
  - unittest: Python unit test

See a comprehensive list here:

[https://en.wikipedia.org/wiki/List\\_of\\_unit\\_testing\\_frameworks](https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)