Approach of Unit testing with the help of JUnit

Satish Mishra

mishra@informatik.hu-berlin.de

About me

- *Satish Mishra*
- *Master of Electronics Science from India*
- *Worked as Software Engineer,Project Manager,Quality Assurance Lead in Software development firm*
- *Presently Doing PhD in Computer Science*

This session

- Testing concepts
  - Unit testing
- Testing tools
  - JUnit
- Practical use of tools
  - Examples
- Discussions
  - Open to you all

Why?

- Why testing?
  - Improve software design
  - Make software easier to understand
  - Reduce debugging time
  - Catch integration errors
- In short, to Produce Better Code
- Preconditions
  - Working code
  - Good set of unit tests

What should be tested ?

- Test for boundary conditions
- Test for both success and failure
- Test for general functionality
- Etc..

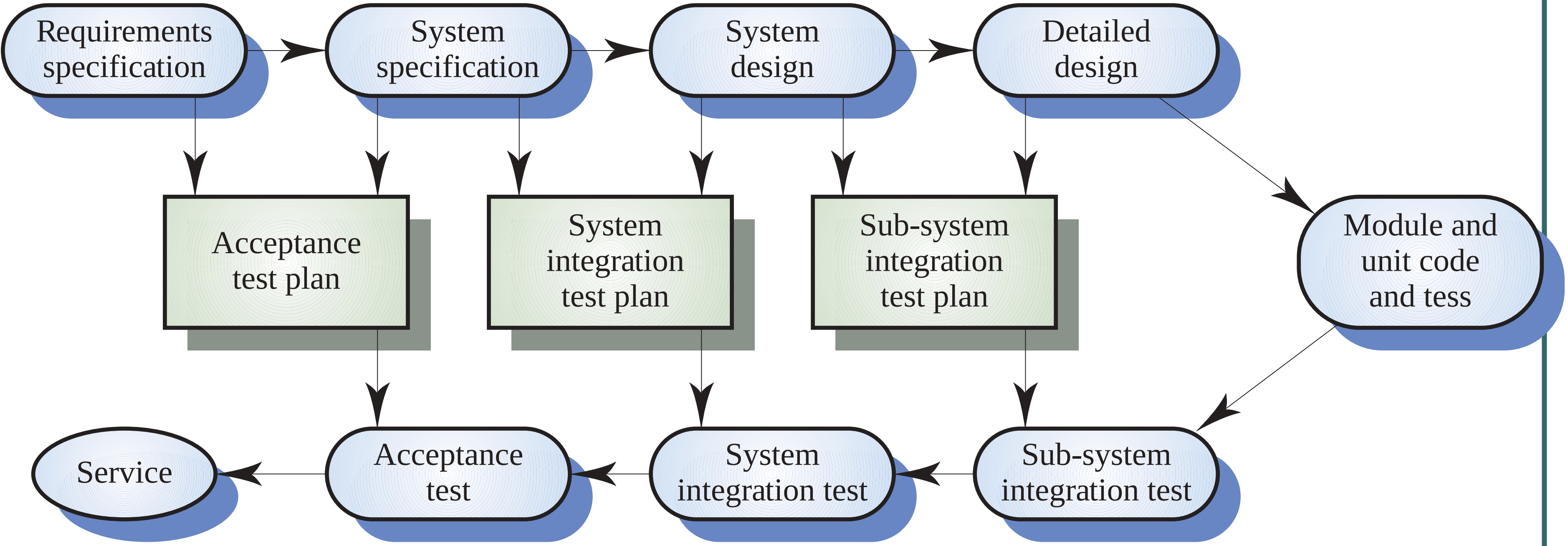
When to start testing

Software quality and testing is a life-cycle process

When to start testing...

- At the time of starting the projects
- How we start the projects ??
- Do we have any formal way ??

The V-model of development


Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and tess
Service
Acceptance test
System integration test
Sub-system integration test

Fact of testing

Testing does not guarantee the absence of defects

What is test case

- A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly

Good test case design

- An good test case satisfies the following criteria:
  - Reasonable probability of catching an error
  - Does interesting things
  - Doesn’t do unnecessary things
  - Neither too simple nor too complex
  - Not redundant with other tests
  - Makes failures obvious
  - Mutually Exclusive, Collectively Exhaustive

Test case design technique

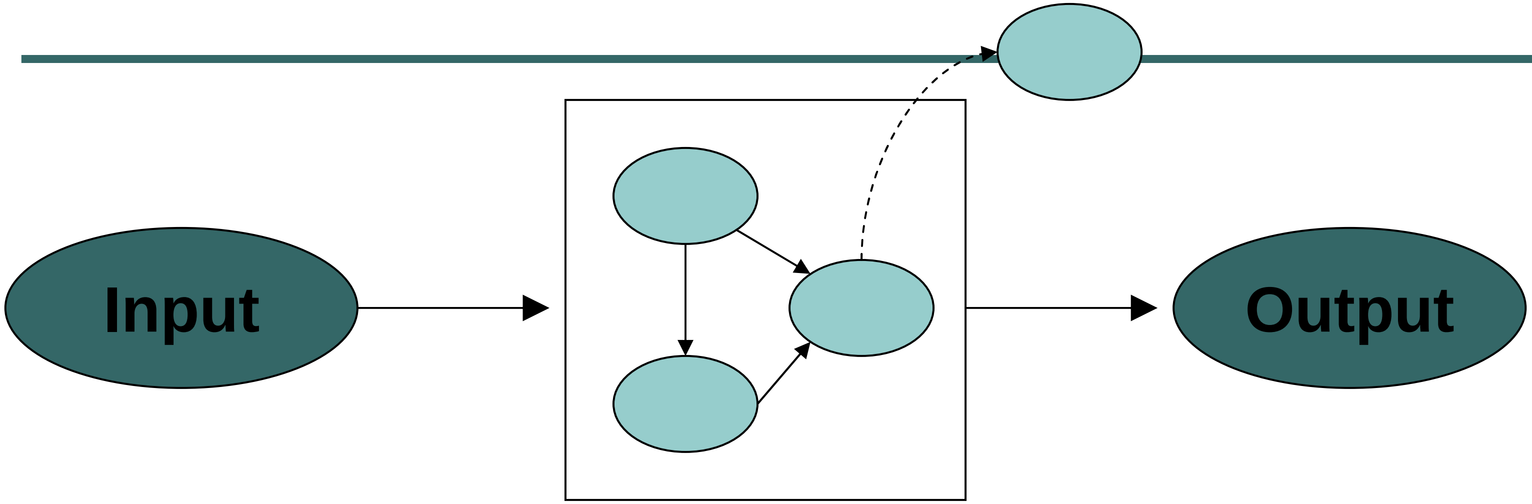
- Test case design techniques can be broadly split into two main categories
  - **Black box (functional)**
  - **White box (structural)**

Black Box tests

Input
Output

- Targeted at the apparent simplicity of the software
  - Makes assumptions about implementation
  - Good for testing component interactions
- Tests the interfaces and behavior

White Box tests



- Targeted at the <u>underlying complexity</u> of the software
  - Intimate knowledge of implementation
  - Good for testing individual functions
- Tests the implementation and design

Test case writing example

- Suppose we have two parameters we want to cover in a set of tests. Parameters are as follows..

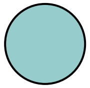
- Operating system
  - Win98
  - Win2k
  - Winxp

- Printers
  - HP 4100
  - HP 4200

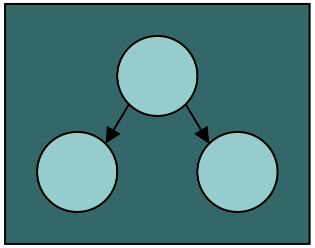
How We should write test case for this ??

Types of Tests

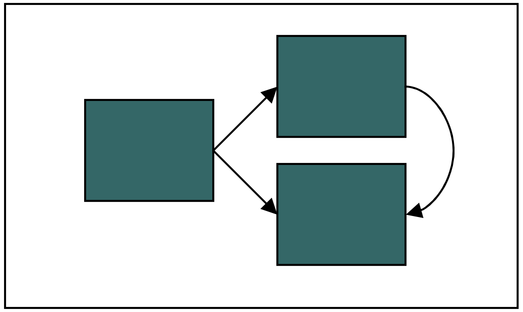
- Unit
  - Individual classes or types



- Component
  - Group of related classes or types



- Integration
  - Interaction between classes



What is a testing framework?

- A test framework provides reusable test functionality which:
  - Is easier to use (e.g. don’t have to write the same code for each class)
  - Is standardized and reusable
  - Provides a base for regression tests

Why use a testing framework?

- Each class must be tested when it is developed
- Each class needs a regression test
- Regression tests need to have standard interfaces
- Thus, we can build the regression test when building the class and have a better, more stable product for less work

Regression testing

- New code and changes to old code can affect the rest of the code base
  - ‘Affect’ sometimes means ‘break’
- We need to run tests on the old code, to verify it works – these are regression tests
- Regression testing is required for a stable, maintainable code base

Testing tools

Tools are part of the quality equation, but not the entire equation

JUnit

- JUnit is a framework for writing unit tests
  - A unit test is a test of a *single* class
    - A test case is a single test of a single method
    - A test suite is a collection of test cases
- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

JUnit..

- JUnit helps the programmer:
  - Define and execute tests and test suites
  - Formalize requirements and clarify architecture
  - Write and debug code
  - Integrate code and always be ready to release a working version

What JUnit does

- JUnit runs a suite of tests and reports results
- For *each* test in the test suite:
  - JUnit calls setUp()
    - This method should create any objects you may need for testing

What JUnit does...

- JUnit calls *one* test method
  - The test method may comprise multiple test cases; that is, it may make multiple calls to the method you are testing
  - In fact, since it’s your code, the test method can do anything you want
  - The setUp() method ensures you *entered* the test method with a virgin set of objects; what you do with them is up to you
- JUnit calls tearDown()
  - This method should remove any objects you created

Creating a test class in JUnit

- Define a subclass of TestCase
- Override the setUp() method to initialize object(s) under test.
- Override the tearDown() method to release object(s) under test.
- Define one or more public testXXX() methods that exercise the object(s) under test and assert expected results.
- Define a static suite() factory method that creates a TestSuite containing all the testXXX() methods of the TestCase.
- Optionally define a main() method that runs the TestCase in batch mode.

Fixtures

- A fixture is just a some code you want run before every test
- You get a fixture by overriding the method
  - protected void setUp() { ...}
- The general rule for running a test is:
  - protected void runTest() {
     setUp(); <run the test> tearDown();
    }
  - so we can override setUp and/or tearDown, and that code will be run prior to or after every test case

Implementing setUp() method

- Override setUp() to initialize the variables, and objects
- Since setUp() is your code, you can modify it any way you like (such as creating new objects in it)
- Reduces the duplication of code

Implementing the **tearDown()** method

- In most cases, the tearDown() method doesn't need to do anything
  - The next time you run setUp(), your objects will be replaced, and the old objects will be available for garbage collection
  - Like the finally clause in a try-catch-finally statement, tearDown() is where you would release system resources (such as streams)

The structure of a test method

- A test method doesn't return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an AssertionFailedError
- The JUnit framework catches the error and deals with it; you don't have to do anything

Test suites

- In practice, you want to run a group of related tests (e.g. all the tests for a class)
- To do so, group your test methods in a class which extends TestCase
- Running suites we will see in examples

assert*X* methods

- static void assertTrue(boolean ***test***)
- static void assertFalse(boolean ***test***)
- assertEquals(***expected***, ***actual***)
- assertSame(Object ***expected***, Object ***actual***)
- assertNotSame(Object ***expected***, Object ***actual***)
- assertNull(Object ***object***)

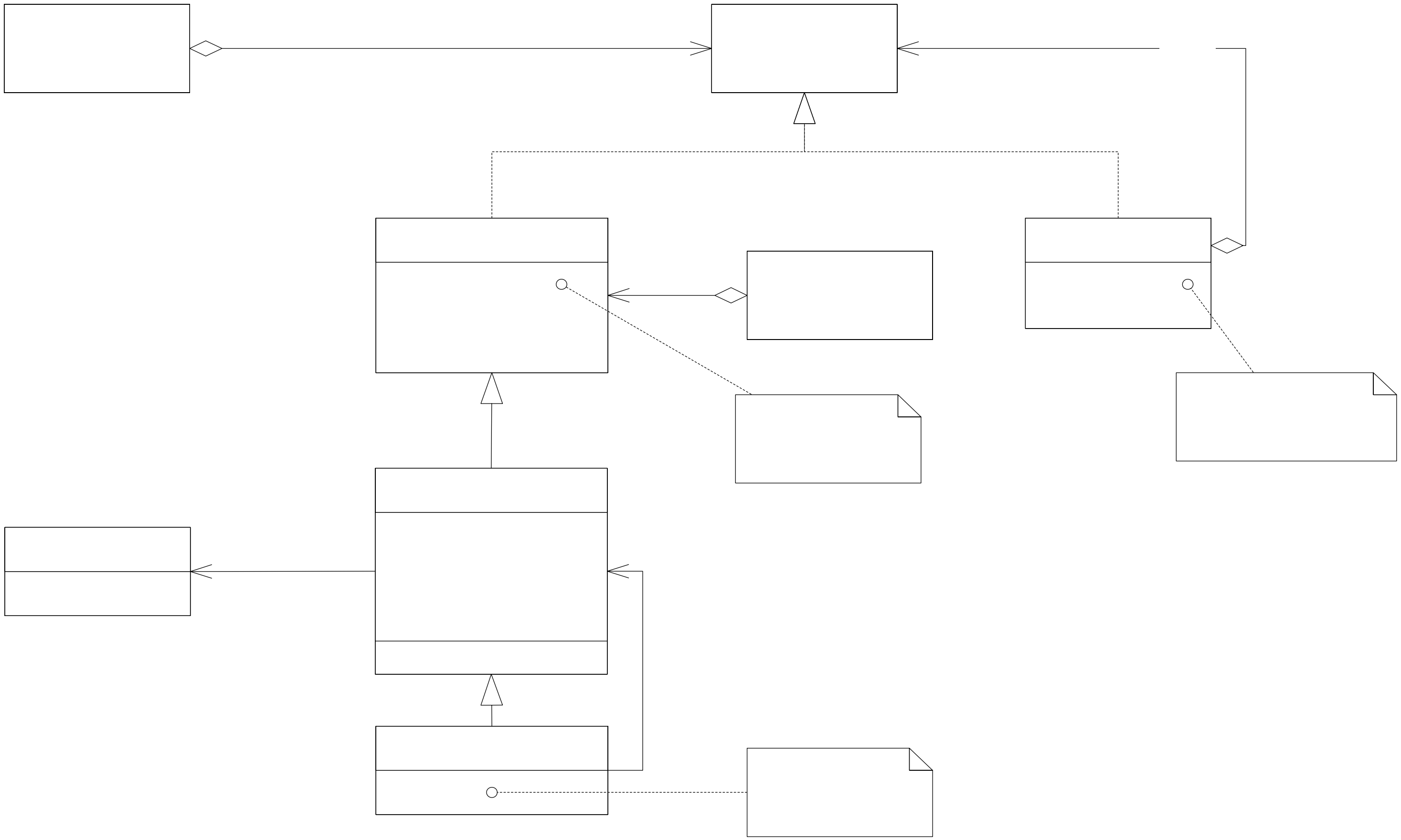
assert*X* methods

- assertNotNull(Object ***object***)
- fail()
- All the above may take an optional String message as the first argument, for example, static void assertTrue(String ***message***, boolean ***test***)

Organize The Tests

- Create test cases in the same package as the code under test
- For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package
- Define similar TestSuite classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application
- Make sure your build process includes the compilation of all tests

JUnit framework



Example: Counter class

- For the sake of example, we will create and test a trivial “counter” class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value

Example: Counter class

- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)
- Don’t be alarmed if, in this simple example, the JUnit tests are more code than the class itself

JUnit tests for **Counter**

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;

    public CounterTest() { }   // default constructor

    protected void setUp() {   // creates a (simple) test fixture
        counter1 = new Counter();
    }

    protected void tearDown() { } // no resources to release
```

JUnit tests for Counter…

```
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}       // End from last slide
```

The **Counter** class itself

```
public class Counter {
        int count = 0;
        public int increment() {
            return ++count;
        }
        public int decrement() {
            return --count;
        }
        public int getCount() {
            return count;
        }
}
```

Result

- We will see with the help of tool

When ??

!! Now !!

Why JUnit

- Allow you to write code faster while increasing quality
- Elegantly simple
- Check their own results and provide immediate feedback
- Tests is inexpensive
- Increase the stability of software
- Developer tests
- Written in Java
- Free
- Gives proper uniderstanding of unit testing

Problems with unit testing

- JUnit is designed to call methods and compare the results they return against expected results
  - This ignores:
    - Programs that do work in response to GUI commands
    - Methods that are used primary to produce output

Problems with unit testing…

- I think heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse

The End

Thank You