



Test-Driven Development with JUnit

Trainers: Yen Le

Title: Professional Software Engineer

Prerequisites

- Basic Java/.NET and OOP skills
- Familiar with Eclipse / MS Visual Studio 2012.

Course Objectives

- At the end of the course, you will have acquired sufficient knowledge to:
 - Understand Quality concepts and methodologies
 - Implement **Test-Driven Development** (TDD) with JUnit
 - Improve quality of deliverables.
 - Improve overall performance.
 - Improve client satisfaction.



Agenda

I.	Quality concepts	<u>10</u>
II.	From Quality to Testing	<u>14</u>
III.	Software Testing with Unit Test	<u>24</u>
IV.	Test-Driven Development - TDD	<u>31</u>
V.	Unit Testing with JUnit	<u>46</u>
VI.	Unit Test with Test Doubles Stub & Mock	<u>56</u>
VII.	Code Coverage	<u>61</u>

Assessment Disciplines

- Class Participation: 100%
- Final exam - Passing Scores: $\geq 70\%$

Duration and Course Timetable

- Course Duration: 4 hrs
- Course Timetable:
 - From 13:30 to 17:30

Further References

- Quality Concepts:
 - <http://asq.org/learn-about-quality/basic-concepts.html>
- TDD and Unit Test:
 - <http://www.agiledata.org/essays/tdd.html>
 - https://en.wikipedia.org/wiki/Test-driven_development
- JUnit
 - <http://www.junit.org>
 - <https://github.com/junit-team/junit>
- Stub & Mock
 - <http://spring.io/blog/2007/01/15/unit-testing-with-stubs-and-mocks/>
 - <http://xunitpatterns.com/Test%20Double.html>

Set Up Environment

- To complete the course, your PC must have:
 - Java
 - Eclipse
 - Maven

Course Administration

- In order to complete the course you must:
 - Sign in the Class Attendance List
 - Participate in the course
 - Provide your feedback in the End of Course Evaluation




Quality Concepts

Quality Concepts

Think about:

- Products you buy
- Services you use



How good are they?

What is Quality?

- Outcome of products / services
- Conform to requirements and satisfy customers

Quality =



Figure 1: What is Quality?

Cost of Quality



Figure 2: Cost of Quality



From Quality To Testing

Roles in testing

- Quality Control (Testing)
- Quality Assurance
- Quality Management

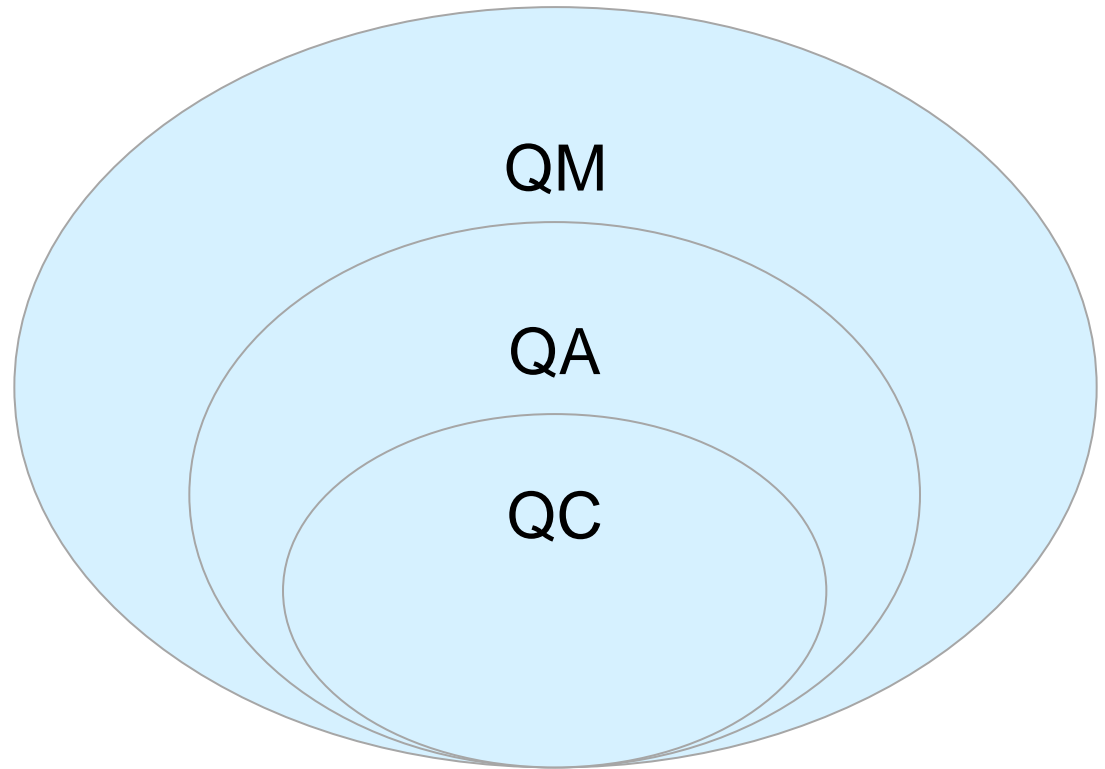


Figure 3: Roles in testing

Software Testing Methods

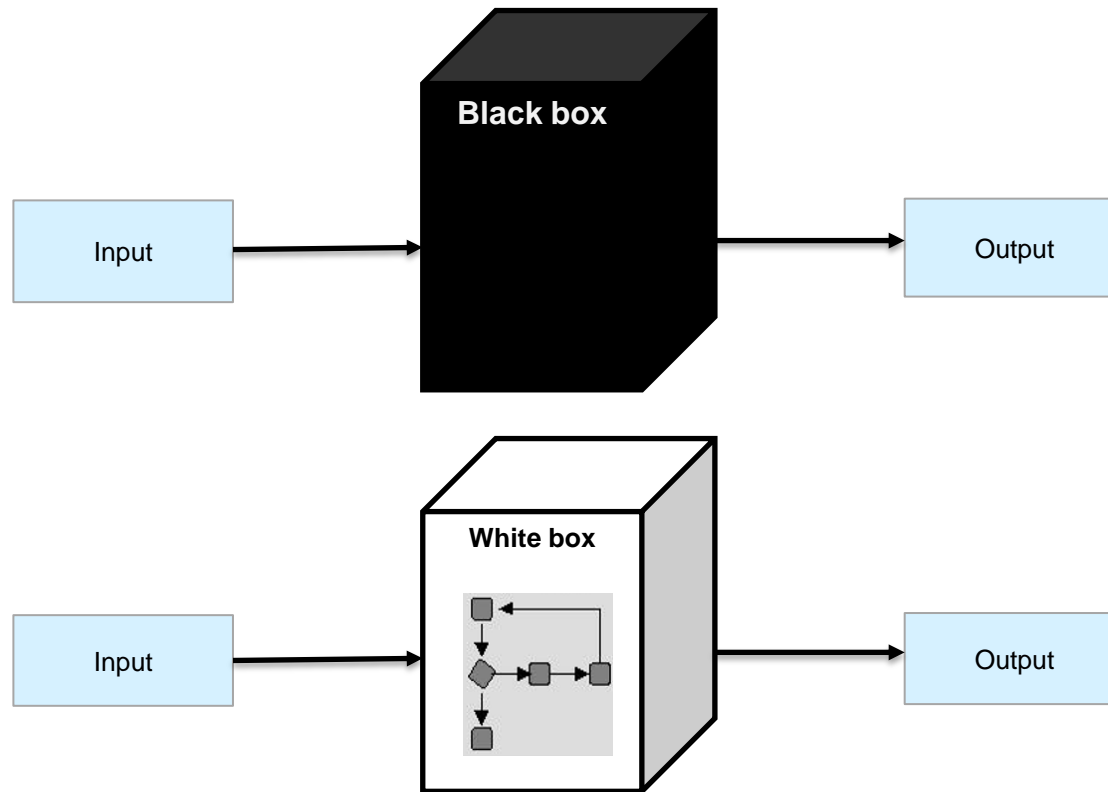


Figure 4: Comparison among Black-box & White-box Tests

Software Testing Methods – Black Box

- Advantages:
 - Efficient on large system or codes.
 - Balanced and unprejudiced since tester and developer are isolated.
 - Not requires code access, understand, and ability to code.
 - Helps to identify vagueness and contradictions in functional specs.
 - Test cases are designed as soon as functional specifications finished

Software Testing Methods – Black Box

- Disadvantages:
 - Hard to design test cases without having clear functional specs
 - Difficult to identify tricky inputs if the test cases aren't followed specs
 - Inefficient tests due to tester's lack of knowledge about software internals
 - Blind coverage since unidentified paths during the testing process
 - Duplicated tests already done by developers.

Software Testing Methods – White Box

- Advantages:
 - Efficiency in finding code errors or hidden defects.
 - Give developer introspection because they carefully describe any new implementation.
 - Help optimizing the code and maximum coverage
 - Improve knowledge of internals of the system.
 - Easy to create automation tests

Software Testing Methods – White Box

- Disadvantages:
 - Miss unimplemented cases or features omitted inside code.
 - Require knowledge of code and internal system which increases the cost.
 - Requires code access
 - Tests are often tightly coupled to the implementation details of the production code, causing unwanted test failures when the code is refactored

Software Testing Levels

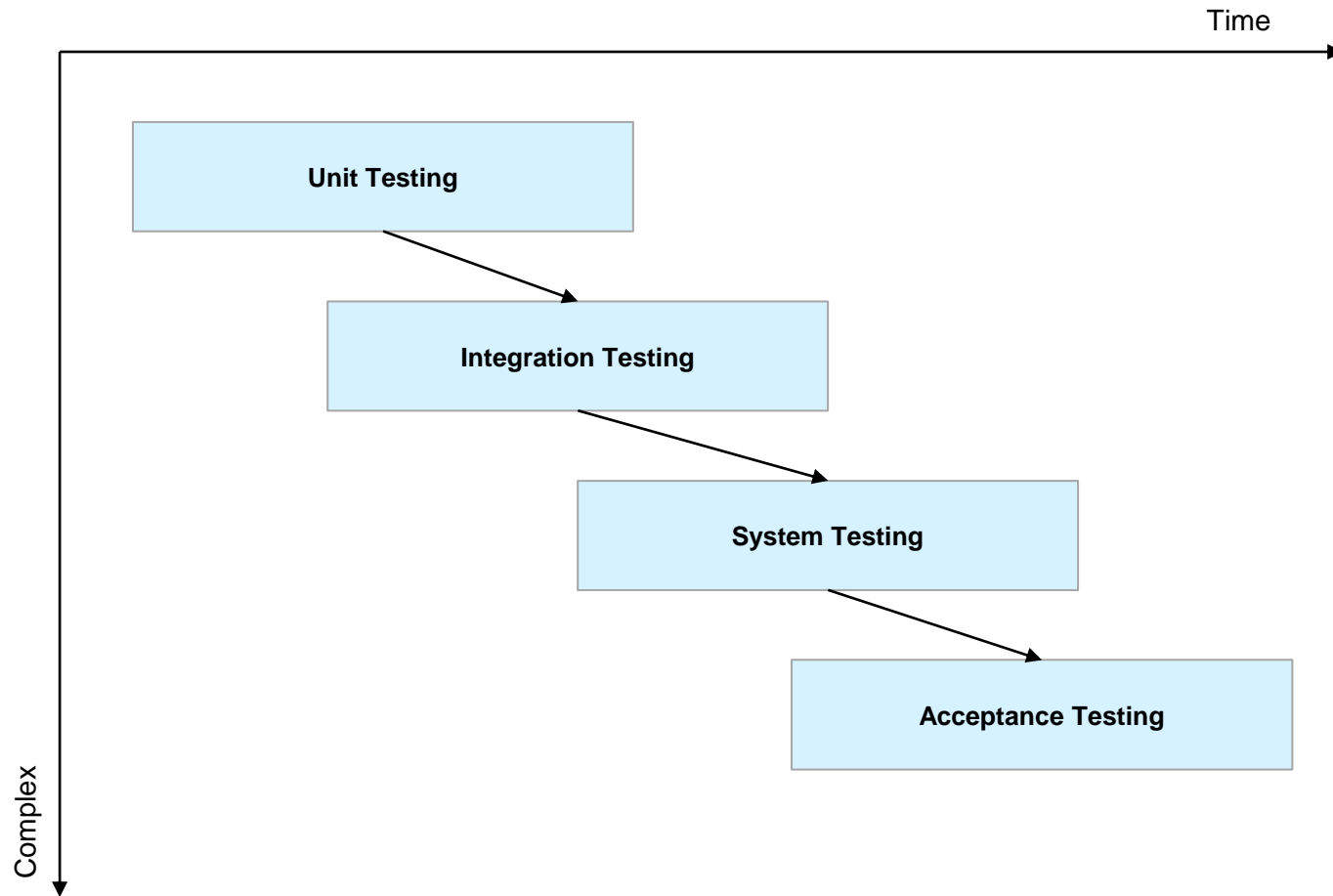


Figure 5: Levels of Software Testing

Testing in the software development life cycles

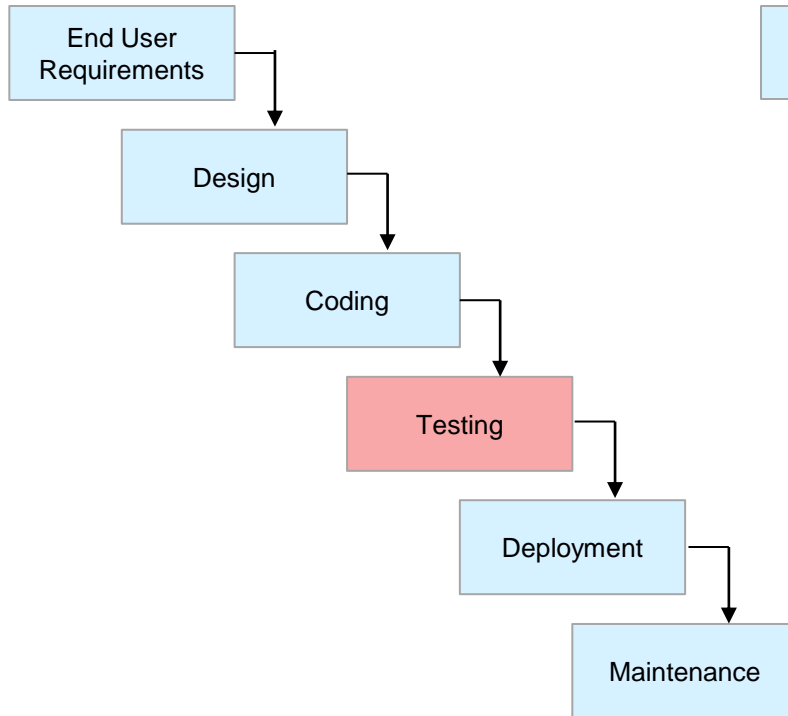


Figure 6: Waterfall model

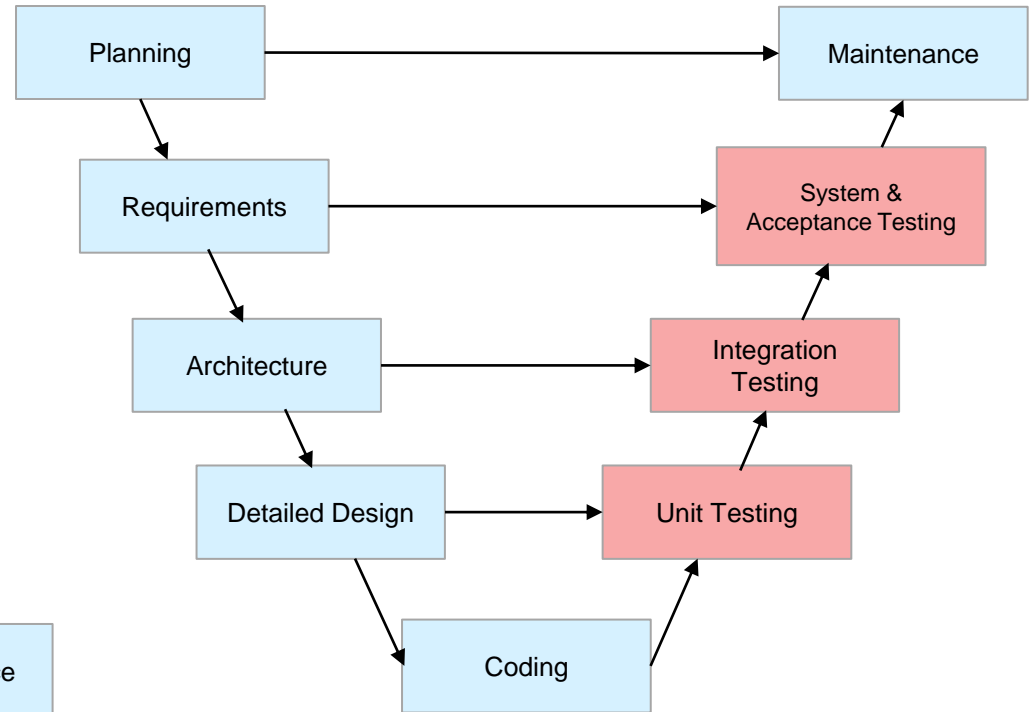


Figure 7: V model

Testing in Scrum

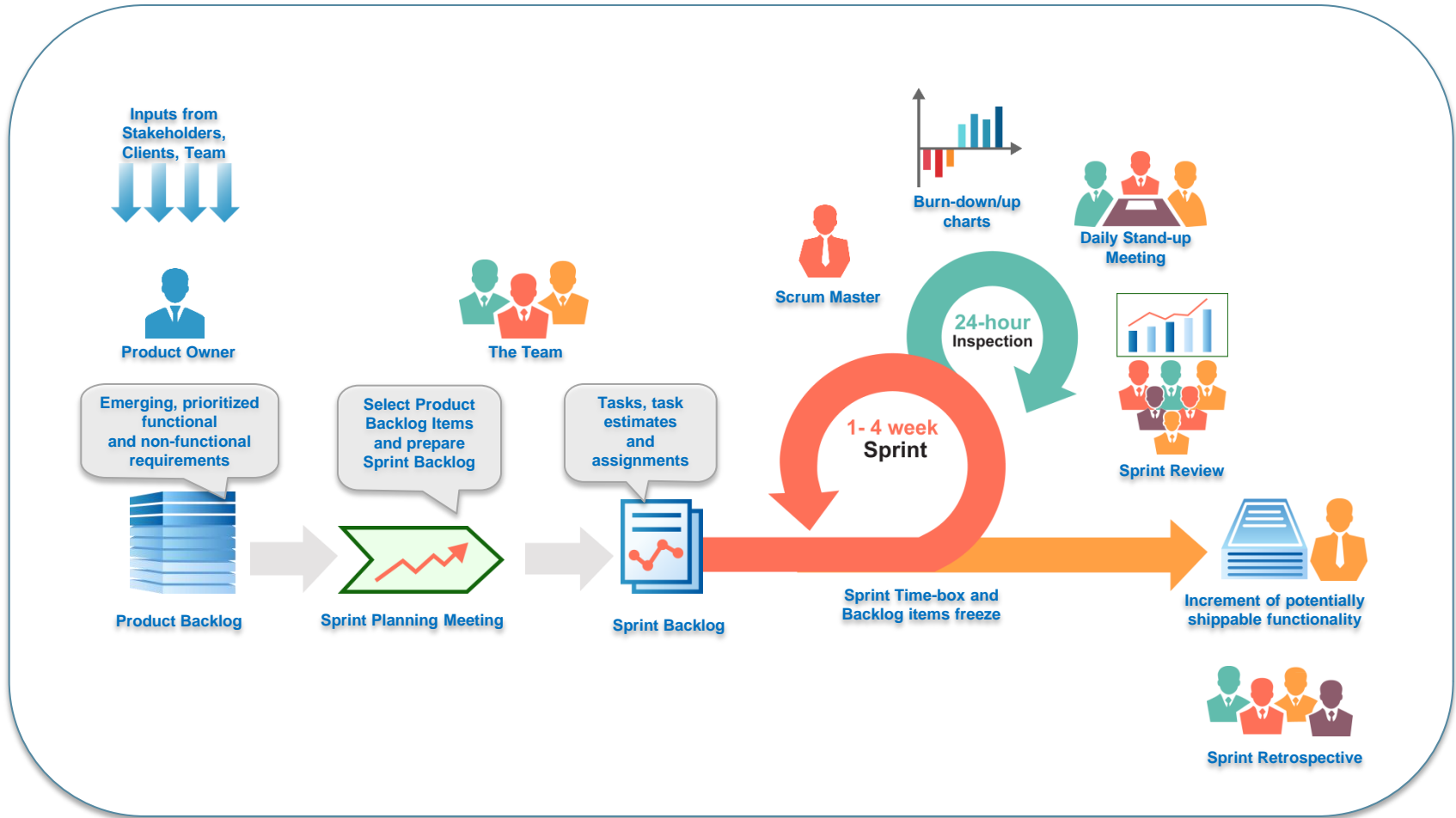


Figure 8: Scrum process

A grayscale background image showing a pair of hands holding a large, curved document or folder. The hands are positioned on the left and right sides of the document, with fingers visible. The document is held open, showing its inner pages. The background is slightly blurred, focusing attention on the hands and the document.

Software Testing with Unit Test

What is Unit Test?

- A unit test is code written by a developer that tests a small piece of functionality (the unit).
- Why we do Unit Test?
 - To know if code really works & when or where the code broken.
 - To help code refactoring.
 - To measure correctness by code coverage, reducing bugs...
 - To make deployments simpler and faster.



Example 1 - Create a Book

```
public class Book {  
    public String title = "";  
    Book(String title) {}  
}  
  
public abstract class UnitTest {  
    protected static int num_test_success = 0;  
    public static int getNumSuccess() {  
        return num_test_success;  
    }  
    public abstract void runTest() throws Exception;  
    protected void assertTrue(boolean condition, String msg)  
        if (!condition)  
            throw new Exception(msg);  
        num_test_success++;  
    }  
}
```

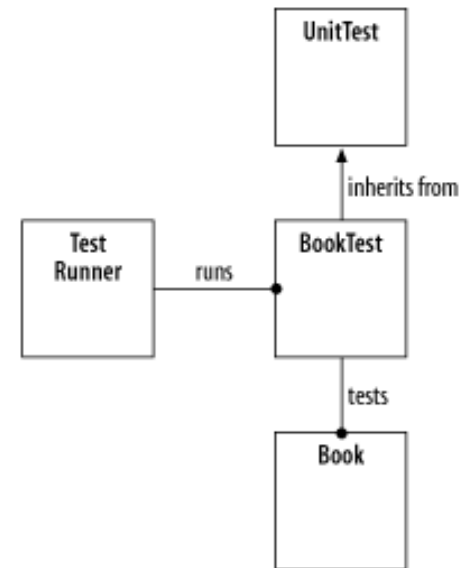


Figure 9: Class diagram for the basic unit test framework

Example 1 - Create a Book - Cont.

```
public class BookTest extends UnitTest {
    public void runTest() throws Exception {
        Book book = new Book("Tom&Jerry");
        assertTrue(book.title.equals("Tom&Jerry"), "checking title");
    }
}

public class TestRunner1 {

    public static void main(String[] args) {
        TestRunner1 tester = new TestRunner1();
    }

    public TestRunner1() {
        try {
            UnitTest test = new BookTest();
            test.runTest();
            System.out.println("Success");
            System.out.println("Number of Success " + UnitTest.getNumSuccess());
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Failure");
        }
    }
}
```

Unit Test - Purpose

- To confirm that methods meet business requirements.
 - Thus the test should verify that the function does what it is supposed to do.
- To confirm the expected behavior for boundary and special values.
- To confirm that exceptions are thrown when expected.

Unit Test – How to write good unit tests?

- Unit test should be written to verify single unit of code and not the integration.
- Small and isolated unit tests with clear naming would make it very easy to write and maintain.
- Changing other part of the software should not affect on unit test, if those are isolated and written for a specific unit of code.
- It should run quickly.
- Unit test should be reusable.

Common Unit Test Frameworks for Java

Name	Description
EasyMock	Mock framework
JUnit	Testing tool default for Java
TestNG	A multi-purpose testing framework, which include unit tests, functional tests, and integration tests, even no-functional tests (as loading tests, timed tests). Easier to use than JUnit
DbUnit	A JUnit extension to perform unit testing with database-driven programs
JMockit	Open source framework and tests can easily be written that will mock final classes, static methods, constructors, and so on. No limitations



Test-Driven Development - TDD

What is Test-Driven Development - TDD?

- Is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies
- Relate to the test-first programming concepts of Extreme Programming (write unit test code before business code, then refactoring)
- Think about “How to use it” first then “how to implement it”
- Also called test-driven design

Why TDD?

- Automated testing
- Better to test New/Modified Functionality
- Developer's confidence
- Manual testing stage is shortened
- Alternative Documentation
- Better way to fix bugs
- Repetition of the same bug reduced

TDD Cycle

- The motto of TDD is “Red, Green and Refactor”
- Red: Create a test and make it fail.
- Green: Make the test pass by any means necessary.
- Refactor: Making the code readable and eliminating duplication

Note: The Red/Green/Refactor cycle is repeated very quickly for each new unit of code.

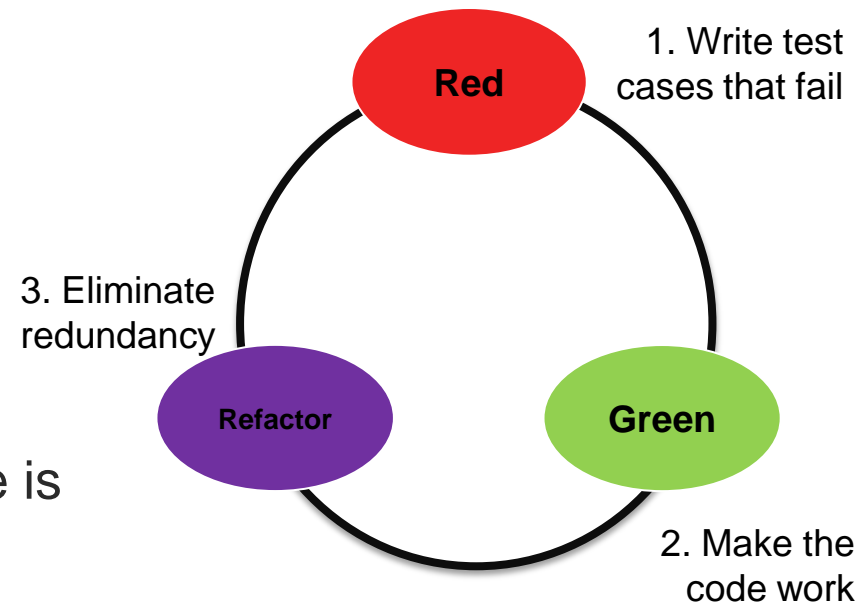


Figure 10: TDD Cycle

TDD in real practice

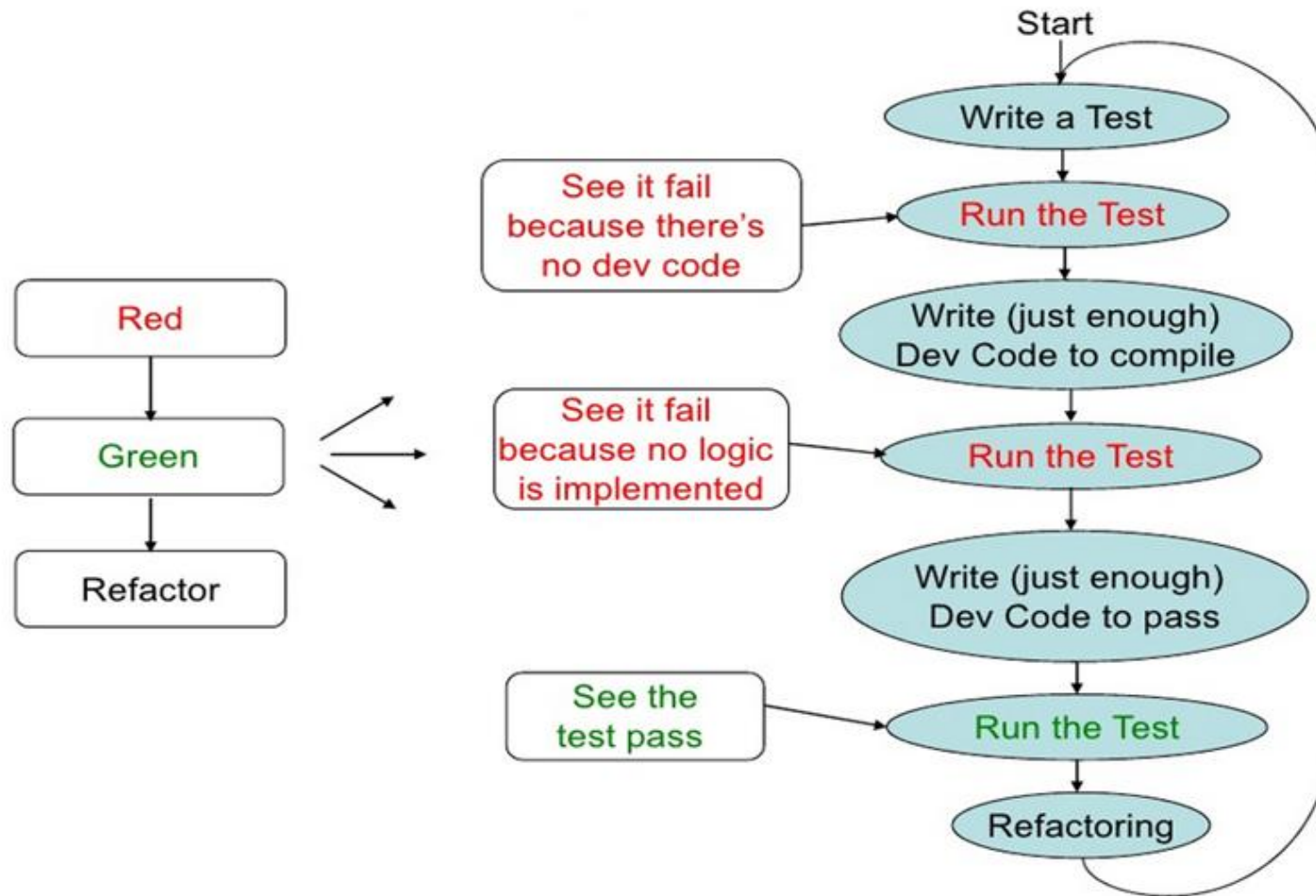



Figure 11: TDD Cycle Detail

Example 1 - Create a Book

Step 1: Create a Unit Test

```
public class Book {  
    public String title = "";  
    Book(String title) {}  
}  
  
public class BookTest extends UnitTest {  
    public void runTest() throws Exception {  
        Book book = new Book("Tom&Jerry");  
        assertTrue(book.title.equals("Tom&Jerry"), "checking title");  
    }  
}  
  
3 public class TestRunner1 {  
4  
5 public static void main(String[] args) {  
6     TestRunner1 tester = new TestRunner1();  
7 }  
8  
9 public TestRunner1() {  
10     try {  
11         UnitTest test = new BookTest();  
12         test.runTest();  
13         System.out.println("Success");  
14     } catch (Exception e) {  
15         System.out.println("Failure");  
16         e.printStackTrace();  
17     }  
18 }  
19 }  
20 }  
21 }  
--
```



java.lang.Exception: checking title
at tdd.UnitTest.assertTrue(UnitTest.java:14)
at tdd.BookTest.runTest(BookTest.java:10)
at tdd.TestRunner1.<init>(TestRunner1.java:12)
at tdd.TestRunner1.main(TestRunner1.java:6)

Example 1 - Create a Book - Cont.

Step 2: Create a Book

```
public class Book {  
    public String title = "";  
    Book(String title) {  
        this.title = title;  
    }  
}
```

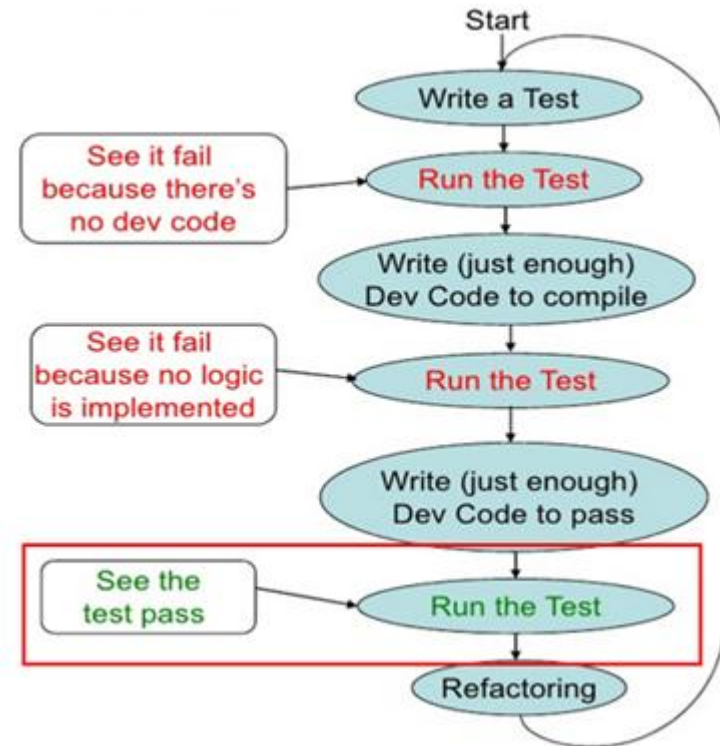


Figure 12: TDD Cycle Detail

Example 2 - Create a Library

Step 1: Test adding a Book to a Library

```
public class Library {  
  
    Library() {  
    }  
  
    public void addBook(Book book) {}  
  
    public Book getBook() {  
        return new Book("");  
    }  
}  
  
public class LibraryTest extends UnitTest {  
    public void runTest() throws Exception {  
        Library library = new Library();  
        Book expectedBook = new Book("Badman");  
        library.addBook(expectedBook);  
  
        Book actualBook = library.getBook();  
  
        assertTrue(actualBook.title.equals("Badman"), "got book!");  
    }  
}
```

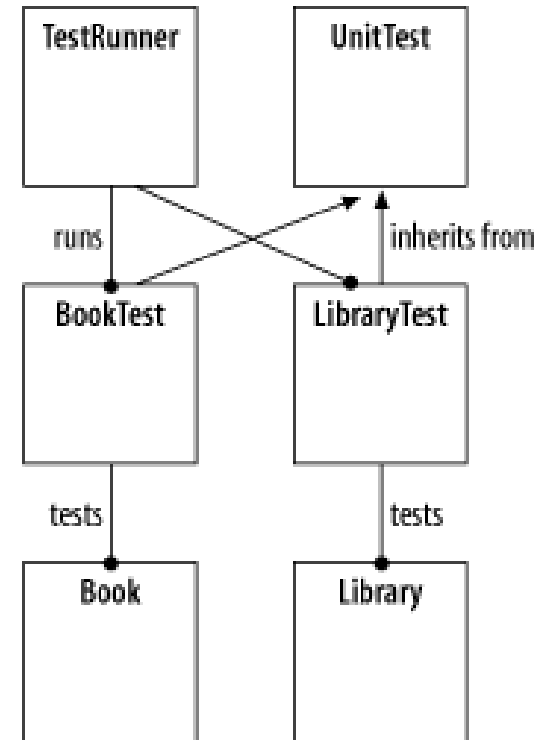


Figure 13: Class diagram for the basic unit test framework

Example 2 - Create a Library – Cont.

Step 1: Test adding a Book to a Library

```
3 public class TestRunner2 {
4
5     public static void main(String[] args) {
6         TestRunner2 tester = new TestRunner2();
7     }
8
9     public TestRunner2() {
10         try {
11             UnitTest bookTest = new BookTest();
12             bookTest.runTest();
13             UnitTest libTest = new LibraryTest();
14             libTest.runTest();
15             System.out.println("Success");
16         } catch (Exception e) {
17             e.printStackTrace();
18             System.out.println("Failure");
19         }
20         System.out.println(UnitTest.getNumSuccess()
21             + " tests completed successfully ");
22     }
23 }
24 }
```

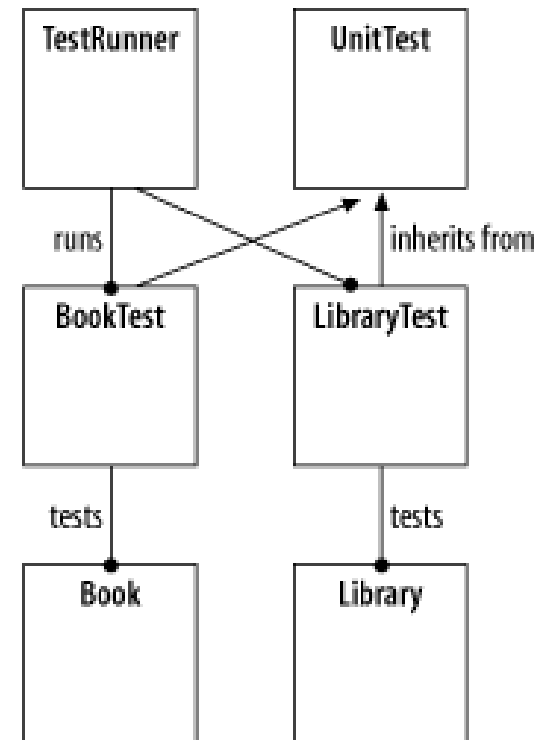


Figure 13: Class diagram for the basic unit test framework

Example 2 - Create a Library - Cont.

Step 2: Add a Book to a Library

```
public class Library {  
    private Book book;  
  
    Library() {  
    }  
  
    public void addBook(Book book) {  
        this.book = book;  
    }  
  
    public Book getBook(String title) {  
        return book;  
    }  
}
```

[InitTest.java:14](#))
[LibraryTest.java:11](#))
[TestRunner2.java:14](#))
[TestRunner2.java:6](#))

Tools and Frameworks

TDD (Unit Testing)

JUnit (Java)	NUnit (.Net)	GUnit (C++)	PyUnit (Python)	Test::Unit (Ruby)	PHPUnit (PHP)
-----------------	-----------------	----------------	--------------------	----------------------	------------------

TDD Best Practices

- Have separate source and test folders. Test code should follow the structure of source.
- Test should fail the first time it's written/run
- Test names should reflect intent, and names should be expressive
- Refactor to remove duplicate code after passing test
- Re-run tests after every refactoring
- Only write new code when a test is failing. Each test should test new/different behavior.
- Write the assertion first
- Minimize the assertions in each test
- All tests should pass before writing the next test

TDD Best Practices – Cont.

- Only refactor when all tests are passing
- Write the simplest code to pass the test
- Don't introduce dependencies between tests. Test should pass when run in any order.
- Tests should run fast. A slow test is a test that won't get run.
- Use mock objects to test code at system boundaries (e.g. database, container, file system) so that tests run fast.
- ...

In conclusion, TDD ensures

- Code Coverage - every line is executed and tested
- Test repeatability
- Documentation – tests helps to catch up code behavior and form document of API code.
- API design
- System design
- Reduce debugging

Practice TDD

Write Unit Test for Book & Library objects



Unit testing with JUnit

Overview of JUnit

- JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.
- Most JDK versions or IDEs supported
- It has been developed by Erich Gamma and Kent Beck



JUnit API

- The most important package in JUnit is `junit.framework` which contain all the core classes.

#	Class Name	Functionality
1	Assert	A set of assert methods.
2	TestCase	A test case defines the fixture to run multiple tests.
3	TestResult	A TestResult collects the results of executing a test case.
4	TestSuite	A TestSuite is a Composite of Tests.

Junit API – Assert

#	Methods & Description
1	Void assertEquals(boolean expected, boolean actual) Check that two primitives/Objects are equal
2	void assertFalse(boolean condition) Check that a condition is false
3	void assertNotNull(Object object) Check that an object isn't null.
4	void assertNull(Object object) Check that an object is null
5	void assertTrue(boolean condition) Check that a condition is true.
6	void fail() Fails a test with no message.

Junit API – TestCase

#	Methods & Description
1	int countTestCases() Counts the number of test cases executed by run(TestResult result).
2	TestResult createResult() Creates a default TestResult object.
3	String getName() Gets the name of a TestCase.
4	TestResult run() A convenience method to run this test, collecting the results with a default TestResult object.
5	void run(TestResult result) Runs the test case and collects the results in TestResult.
6	void setName(String name) Sets the name of a TestCase.
7	void setUp() Sets up the fixture, for example, open a network connection.
8	void tearDown() Tears down the fixture, for example, close a network connection.
9	String toString() Returns a string representation of the test case.

Junit API – TestResult

#	Methods & Description
1	void addError (Test test, Throwable t) Adds an error to the list of errors.
2	void addFailure (Test test, AssertionError t) Adds a failure to the list of failures.
3	void endTest (Test test) Informs the result that a test was completed.
4	int errorCount () Gets the number of detected errors.
5	Enumeration<TestFailure> errors () Returns an Enumeration for the errors.
6	int failureCount () Gets the number of detected failures.
7	void run (TestCase test) Runs a TestCase.
8	int runCount () Gets the number of run tests.
9	void startTest (Test test) Informs the result that a test will be started.
10	void stop () Marks that the test run should stop.

Junit API – TestSuite

#	Methods & Description
1	void addTest (Test test) Adds a test to the suite.
2	void addTestSuite (Class<? extends TestCase> testClass) Adds the tests from the given class to the suite.
3	int countTestCases () Counts the number of test cases that will be run by this test.
4	String getName () Returns the name of the suite.
5	void run (TestResult result) Runs the tests and collects their result in a TestResult.
6	void setName (String name) Sets the name of the suite.
7	Test testAt (int index) Returns the test at the given index.
8	int testCount () Returns the number of tests in this suite.
9	static Test warning (String message) Returns a test which will fail and log a warning message.

JUnit API – Basic Annotations

# Annotation & Description	
1	@Test The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
2	@Before Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method.
3	@After If you allocate external resources in a Before method you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method.
4	@BeforeClass Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
5	@AfterClass This will perform the method after all tests have finished. This can be used to perform clean-up activities.
6	@Ignore The Ignore annotation is used to ignore the test and that test will not be executed.

JUnit complete example using Eclipse

- Will see in detail how to create and run tests and we will show how to use specific annotations and assertions of JUnit

Advantages of JUnit

- Quick to write / reuse test cases & test data
- Easy manage tests of whole project
- Asserts for contrast between expected and real output
- Integrate with tools – Ant, Maven, IDEs



Disadvantages of JUnit

- Unable to do dependency tests.
- Not suitable for larger test suite.



A grayscale background image showing a person's hands holding an open book. The person is wearing a ring on their left ring finger. The book is open, and the pages are visible. The image is slightly blurred, focusing on the hands and the book.

Unit Test with Test Doubles Stub & Mock

Test Double

- Test Double is an Object that can stand in for a real object in a test.
- Test Double helps to verify logic independently when code it depends on is not available or unusable
- Test Double helps to avoid Slow Test.

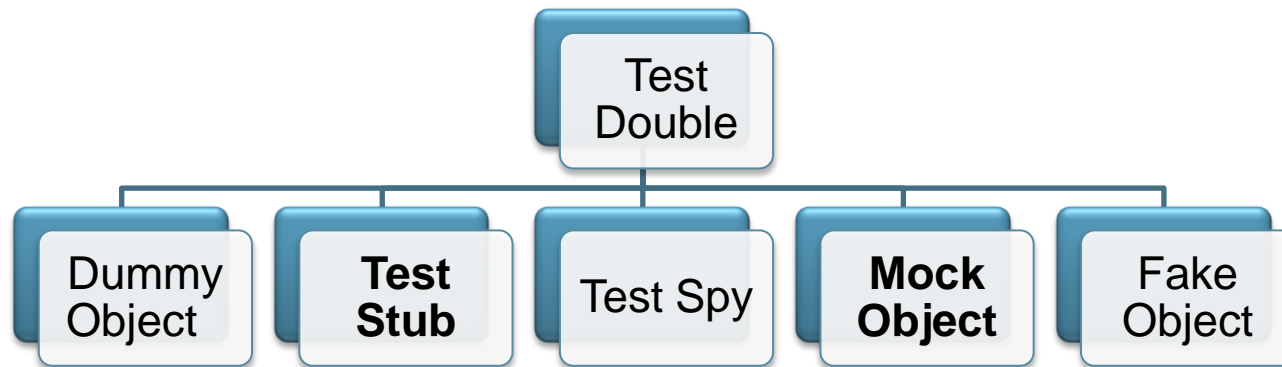


Figure 14: Test Double variations

Test Stub

- A variation of Test Double.
- Test Stub helps to verify logic independently when it depends on indirect inputs from other components.

```
public Student findStudentByStudentID(String studentID)
{
    if (studentID == null || "".equals(studentID))
    {
        throw new IllegalArgumentException("StudentID is not valid");
    }
}
```

How to create a Test Stub ?

- Implement the same interface as the production object, often use hard-code implementation (manually).

- Use mocking framework. Example: Mockito.

```
Student temp = new Student();
temp.setStudentID(studentID);
Map<String> students = new HashMap<>();
if (students == null || students.size() == 0)
{
    return null;
}
else
{
    if (students.size() > 1)
    {
        throw new RuntimeException("There are more than one students with given ID: " + studentID);
    }
    else
    {
        Student foundStudent = students.get(0);
        cachingService.putToCache(studentID, foundStudent);
        return foundStudent;
    }
}
```

Need a Stub here



Mock Object

- A variation of Test Double.
- Mock Object helps to verify logic independently when it depends on indirect inputs from other components.
- Mock Object implements Behavior Verification for indirect outputs of the SUT.

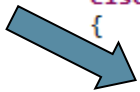
```
public Student findStudentByStudentID(String studentID)
{
    if (studentID == null || "".equals(studentID))
    {
        throw new IllegalArgumentException("StudentID is not valid");
    }
}
```

How to create a Mock Object ?

- Use mocking framework. Example. EasyMock, jMock and Mockito.

```
List<Student> students = repositoryService.searchStudent(temp);
if (students == null || students.size() == 0)
{
    return null;
}
else
{
    if (students.size() > 1)
    {
        throw new RuntimeException("There are more than one students with given ID: " + studentID);
    }
    else
    {
        Student foundStudent = students.get(0);
        cachingService.putToCache(studentID, foundStudent);
        return foundStudent;
    }
}
```

Need a
Mock
here



Mockito

- Mockito is a mocking framework and it is very simple to learn.
- Mockito helps to create Test Double such as Dummy, Stub, Mock Object easily

```
@Test
public void testFindStudentByStudentID_returnFoundStudent()
{
    //Setup
    IStudentRepositoryService repoService = Mockito.mock(IStudentRepositoryService.class);
    INotificationService notiService = Mockito.mock(INotificationService.class);
    IStudentCachingService cachingService = Mockito.mock(IStudentCachingService.class);

    List<Student> resultList = new ArrayList<Student>();
    Student dummy = new Student();
    dummy.setFirstName("Nguyen");
    dummy.setLastName("Van");
    resultList.add(dummy);

    Mockito.when(repoService.searchStudent(Mockito.any(Student.class))).thenReturn(resultList);

    sut = new StudentManagementServiceBackKhoaImpl(repoService, notiService, cachingService);

    //Exercise
    Student foundStudent = sut.findStudentByStudentID("50303416");

    //Verify
    Assert.assertEquals("Nguyen", foundStudent.getFirstName());
    Assert.assertEquals("Van", foundStudent.getLastName());

    Mockito.verify(cachingService).putToCache("50303416", foundStudent);
    //Tear down
}
```

Stub object

Dummy object

Mock object



Code Coverage






Code Coverage

- Code coverage is the degree to which the source code of a program is tested by a particular test suite.
- Coverage criteria:
 - Function coverage
 - Statement coverage
 - Branch coverage
 - Condition coverage




Benefit of Code Coverage

- It creates additional test cases to increase coverage.
- It helps in finding areas of a program not exercised by a set of test cases.
- It helps in determining a quantitative measure of code coverage, which indirectly measure the quality of the application or product.













Code Coverage – Example

Element	Missed Instructions	Cov.	Missed Branches	Cov.
 training.csc.com.testdouble.management		57%		50%
 training.csc.com.testdouble.model		39%		n/a
Total	94 of 191	51%	8 of 16	50%

training.csc.com.testdouble.management

Element	Missed Instructions	Cov.	Missed Branches	Cov.
 StudentManagementServiceBackKhoalImpl		57%		50%
Total	56 of 129	57%	8 of 16	50%


StudentManagementServiceBackKhoalImpl

Element	Missed Instructions	Cov.	Missed Branches	Cov.
 updateStudentProfile(Student)		0%		0%
 findStudentByStudentID(String)		91%		80%
 registerNewStudent(Student)		0%		n/a
 StudentManagementServiceBackKhoalImpl(IStudentRepositoryService, INotificationService, IStudentCachingService)		100%		n/a
 StudentManagementServiceBackKhoalImpl(IStudentRepositoryService, INotificationService)		100%		n/a
Total	56 of 129	57%	8 of 16	50%



Code Coverage – Example – Cont.

Not yet covered
this line.



```
35.     public Student findStudentByStudentID(String studentID)
36.     {
37.         if (studentID == null || "".equals(studentID))
38.         {
39.             throw new IllegalArgumentException("StudentID is not valid");
40.         }
41.
42.         Student temp = new Student();
43.         temp.setStudentID(studentID);
44.         List<Student> students = repositoryService.searchStudent(temp);
45.
46.         if (students == null || students.size() == 0)
47.         {
48.             return null;
49.         }
50.         else
51.         if (students.size() > 1)
52.         {
53.             throw new RuntimeException("There are more than one students with given ID: " + studentID);
54.         }
55.         else
56.         {
57.             Student foundStudent = students.get(0);
58.             cachingService.putToCache(studentID, foundStudent);
59.             return foundStudent;
60.         }
61.     }
```

Tools and Frameworks

Java Code Coverage Tools

JCov	JaCoCo	Clover	Cobertura	EMMA	Serenit
------	--------	--------	-----------	------	---------

Summary

- Software Quality and Testing
- TDD and Unit Test
- Junit, Stub, Mock and how to use
- Code Coverage



Q&A



Thank You

Revision History

Date	Version	Description	Updated by	Reviewed and Approved By
Nov 26, 2015	1.0	Create first version for Java fresher training with the contents I. Quality concepts II. From Quality to Testing III. Software Testing with Unit Test IV. Test-Driven Development - TDD V. Unit Testing with JUnit VI. Unit Test with Test Doubles Stub & Mock VII. Code Coverage	Duyen Dang Yen Huynh	Khanh Lam Quang Tran



BUSINESS SOLUTIONS
TECHNOLOGY
OUTSOURCING