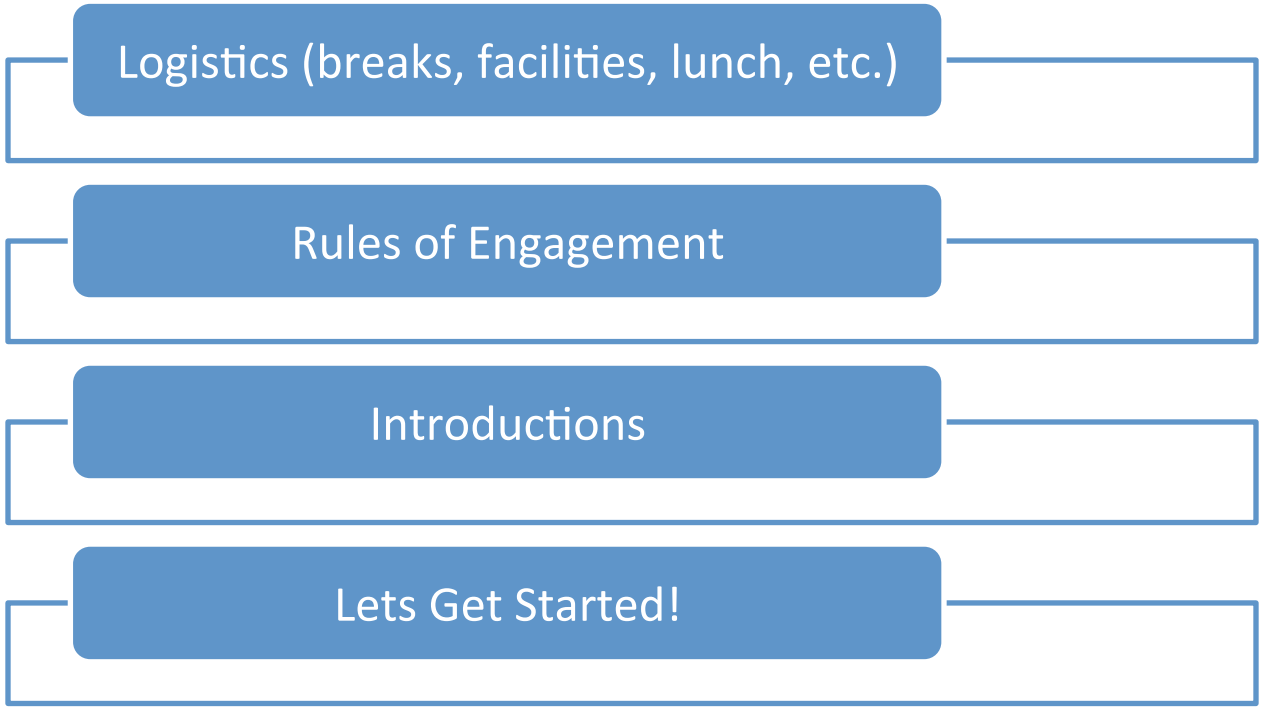
Test-Driven Development Workshop

60200M_2.0_2017

Welcome


Logistics (breaks, facilities, lunch, etc.)
Rules of Engagement
Introductions
Lets Get Started!

Instructor Introduction

Who is your instructor?

A little about me…

Introduction

- Your name
- Your role
- Your experience with TDD
- One interesting fact about yourself

Agenda

- Introduction
- Unit Testing
- Test Driven Development
- Test Doubles

- Refactoring
- Pair Programming
- ATDD and BDD
- Test Refactoring
- Design Principles

- Agile Overview
- DevOps Culture

INTRODUCTION AND OVERVIEW

What to Expect

- Flexibility
- Conversations
- A focus on human behavior and values
- Discussions and hands on labs about TDD

What Not to Expect

- Prescriptions and formulas, rigid processes, step-by-step instructions
- Big overnight transformations
- Perfect solutions that work for everyone

UNIT TESTING

Discussion: Unit Testing

10 Minutes

- What are some important facts about unit testing that you already know?
- What would you like to learn about unit testing?

What is Unit Testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing can be done manually but is often automated.

Manual Unit Testing Example

https://github.com/shekhar2010us/tdd/blob/master/src/main/java/shekhar/unit/example1/ManualUnitTest.java

```
//Given that we have a new shopping list
List listOfItemsForShopping = new ArrayList();

System.out.println("Initial size of list is "+listOfItemsForShopping.size());

//When we add an object in the shopping list

Object objANewGroceryItem= new Object(  );

listOfItemsForShopping.add(objANewGroceryItem);

// then the size of shopping list should be 1

System.out.println("After adding item, size of list "+listOfItemsForShopping.size());
```

Better Manual Unit Test - Example

```
private static void assertTrue( boolean condition) throws Exception {
    if (! condition)
        throw new Exception("Test failed");
}

public static void main(String[] args)
{
    //Given we have a List of size 0

    List listOfItemsForShopping = new ArrayList();
    try {
        assertTrue( condition: listOfItemsForShopping.size() == 0);
    //When we add an object in the shopping list

    Object objANewGroceryItem= new Object();

    listOfItemsForShopping.add(objANewGroceryItem);

    // then the size of shoppinglist should be 1

    assertTrue( condition: listOfItemsForShopping.size()== 1) ;



    /*
            Lab 1, excercise 2 - Implement  new test case-

            Add one object to listOfItemsForShopping
            And assertTrue that size of listOfItemsForShopping is 2
     */
```

https://github.com/shekhar2010us/tdd/blob/master/src/main/java/shekhar/unit/example1/AutoUnitTest.java

Lab 1: Setup Environment – clone the git project

20 Minutes

- Login to the provided AWS machine
- Update packages

  $$ sudo apt-get -y update
- Install necessary packages

  $$ sudo apt-get install -y maven git default-jdk tree

Check **java -version** **git --version** **mvn --version**

- Clone the git project

  $$ git clone https://github.com/shekhar2010us/tdd.git
- Build the project

  $$ cd tdd

  $$ mvn clean install -U

Lab 2: Run basic Manual & Auto Test

20 Minutes

- cd to the {tdd} project

  $$ cd ~/tdd
- Check code of the manual test (no framework used here)

  $$ vi src/main/java/shekhar/unit/example1/ManualUnitTest.java
- Run the class

  $$ mvn exec:java -q -Dexec.mainClass="shekhar.unit.example1.ManualUnitTest"

- Check code of the basic Junit example (just used assertion)

  $$ vi src/main/java/shekhar/unit/example1/AutoUnitTest.java
- Check code of the basic Junit example

  $$ mvn exec:java -q -Dexec.mainClass="shekhar.unit.example1.AutoUnitTest"

  If you don't get error, you are fine.

Unit Testing Frameworks

- JUnit
- NUnit
- xUnit

JUnit

In this module we would experience unit testing with some simple tests using JUnit.

JUnit:

- Runs tests automatically
- Runs many tests together and summarizes the results
- Provides a convenient place to collect the tests you've written
- Provides a convenient place for sharing the code used to create the objects you are going to test
- Compares actual results to expected results and reports differences

Using JUnit

- Use **junit.jar** and add to classpath
- The Git project is using **"maven" build automation tool**
- Check pom.xml in the cloned project, you will find "junit" and "mockito" as dependencies
- We have already cloned the project

https://github.com/shekhar2010us/tdd

JUnit API

Assert Methods

Javadoc:- http://junit.sourceforge.net/javadoc/org/junit/Assert.html

A collection of methods for checking actual values against expected values

- assertEquals
- assertNotNull
- assertTrue
- assertSame
- assertFalse

Question:- how [assertSame] different than [assertEquals]

assertSame and assertEquals

assertSame – the two objects refer to the same memory object

assertEquals – the two objects has same value

(i.e. the output of the method *.toString()* of the two objects are same)

```
String a = new String("TDD");
String b = a;
assertSame(a,b);
```

Pass

```
String a = new String("TDD");
String b = new String("TDD");
assertSame(a,b);
```

Fails !!

```
String a = new String("TDD");
String b = a;
assertEquals(a,b);
```

Pass

```
String a = new String("TDD");
String b = new String("TDD");
assertEquals(a,b);
```

Pass

JUnit API

@Test:

- The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
- **To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method.**
- Any exceptions thrown by the test will be reported by JUnit as a failure.
- If no exceptions are thrown, the test is assumed to have succeeded.

JUnit API

@Before:

- When writing tests, it is common to find that several tests need similar objects created before they can run.
- Annotating a public void method with @Before causes that method to be run before the Test method.
- The @Before methods of superclasses will be run before those of the current class.

JUnit API

@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.
- **All @After methods are guaranteed to run even if a Before or Test method throws an exception.**
- The @After methods declared in superclasses will be run after those of the current class.

Sequence of @Before and @After

```
public class IntegrationTestBase {
    @Before
    public final void setUp() {
        System.out.println("before in base class");
    }
    @After
    public final void tearDown() {
        System.out.println("after in base class");
    }
}

public class MyTest extends IntegrationTestBase {
    @Before
    public final void before() {
        System.out.println("before in implemented
class");
    }
    @Test
    public void test() {
        System.out.println("test in implemented
class");
    }
    @After
    public final void after() {
        System.out.println("after in implemented
class");
    }
}
```

Base class
https://github.com/shekhar2010us/tdd/blob/master/src/test/java/shekhar/tdd/IntegrationTestBase.java

Implemented class
https://github.com/shekhar2010us/tdd/blob/master/src/test/java/shekhar/tdd/MyTest.java

<< OUTPUT >>
before in base class
before in implemented class
test in implemented class
after in implemented class
after in base class

Sequence of @Before and @After – Contd

```
public class BasicAnnotationTest {

    @After
    public void tearDownfirst() throws Exception {
        System.out.println("AFTER first IN BASE CLASS");
    }

    @After
    public void tearDownsecond() throws Exception {
        System.out.println("AFTER second IN BASE CLASS");
    }

    @Test
    public void test() throws Exception {
        System.out.println("TEST IN BASE CLASS");
    }

    @Test
    public void test2() throws Exception {
        System.out.println("TEST 2 IN BASE CLASS");
    }

    @Before
    public void setUpfirst() throws Exception {
        System.out.println("BEFORE first IN BASE CLASS");
    }

    @Before
    public void setUpsecond() throws Exception {
        System.out.println("BEFORE second IN BASE CLASS");
    }

}
```

Class
https://github.com/shekhar2010us/tdd/blob/master/src/test/java/shekhar/tdd/BasicAnnotationTest.java

<< OUTPUT >>
BEFORE second IN BASE CLASS
BEFORE first IN BASE CLASS
TEST 1 IN BASE CLASS
AFTER second IN BASE CLASS
AFTER first IN BASE CLASS

BEFORE second IN BASE CLASS
BEFORE first IN BASE CLASS
TEST 2 IN BASE CLASS
AFTER second IN BASE CLASS
AFTER first IN BASE CLASS

** Note: having multiple @Before and @After doesn't guarantee the sequence

JUnit API

@BeforeClass:

- Sometimes several tests need to share computationally expensive setup (like logging into a database).
- While this can compromise the independence of tests, sometimes it is a necessary optimization.
- Annotating a public static void no-arg method with @BeforeClass causes it to be **run once** before any of the test methods in the class.
- The @BeforeClass methods of superclasses will be run before those the current class.

JUnit API

@AfterClass:

- If you allocate expensive external resources in a BeforeClass method you need to release them after all the tests in the class have run.
- Annotating a public static void method with @AfterClass causes that method to be run after all the tests in the class have been run.
- All @AfterClass methods are guaranteed to run even if a BeforeClass method throws an exception.
- The @AfterClass methods declared in superclasses will be run after those of the current class.

Sequence of @BeforeClass and @AfterClass

```
public class BasicAnnotationTestClass {

    // Run once, e.g. Database connection, connection pool
    @BeforeClass
    public static void runOnceBeforeClass() {
        System.out.println("@BeforeClass –
runOnceBeforeClass");
    }

    // Run once, e.g close connection, cleanup
    @AfterClass
    public static void runOnceAfterClass() {
        System.out.println("@AfterClass – runOnceAfterClass");
    }

@Test
    @Before
    public void runBeforeTestMethod() {
        System.out.println("@Before – runBeforeTestMethod");
    }
    // Should rename to @AfterTestMethod
    @After
    public void runAfterTestMethod() {
        System.out.println("@After – runAfterTestMethod");
    }

    @Test
    public void test_method_1() {
        System.out.println("@Test – test_method_1");
    }
    @Test
    public void test_method_2() {
        System.out.println("@Test – test_method_2");
    }

}
```

Output:

```
@BeforeClass – runOnceBeforeClass

@Before - runBeforeTestMethod
@Test - test_method_1
@After - runAfterTestMethod

@Before - runBeforeTestMethod
@Test - test_method_2
@After - runAfterTestMethod

@AfterClass - runOnceAfterClass
```

https://github.com/shekhar2010us/tdd/blob/master/src/test/java/shekhar/tdd/BasicAnnotationTestClass.java

Lab 3: Run your first Junit test

40 Minutes

- cd to the {tdd} project $$ cd ~/tdd
- Check code of the base class

    $$ vi src/test/java/shekhar/tdd/IntegrationTestBase.java

This is just a base class defining @After and @Before, we do not have to run this

- Check code of the first test {MyTest}, that extends the base class

    $$ vi src/test/java/shekhar/tdd/MyTest.java

This class extends the previous base class and defines a @Test block, let's run it

- Run the test {MyTest}

    $$ mvn surefire:test -q -Dtest=shekhar.tdd.MyTest

Check the sequence of print statements and match with what we discussed

Lab 3: Run your first Junit test **Contd.**

- Check code of the class with multiple @Before and @After

  $$ vi src/test/java/shekhar/tdd/BasicAnnotationTest.java

This class defines multiple @After and @Before, let's run it

- Run the test {BasicAnnotationTest}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.BasicAnnotationTest

Check the sequence of print statements and match with what we discussed.

Confirm that in case you have multiple @Before and @After, the sequence of running these are not guaranteed.

Lab 3: Run your first Junit test **Contd.**

- Check code of the class defining multiple @Before and @After but better

  $$ vi src/test/java/shekhar/tdd/BasicAnnotationTestBetter.java

This class defines multiple @After and @Before, but wrapped in a method

- Run the test {BasicAnnotationTestBetter}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.BasicAnnotationTestBetter

Check the sequence of print statements and match with what we discussed.

Lab 3: Run your first Junit test **Contd.**

- Check code of the class that defines @AfterClass and @BeforeClass

  $$ vi src/test/java/shekhar/tdd/BasicAnnotationTestClass.java

This class defines @After, @AfterClass, @Before, @BeforeClass, let's run it

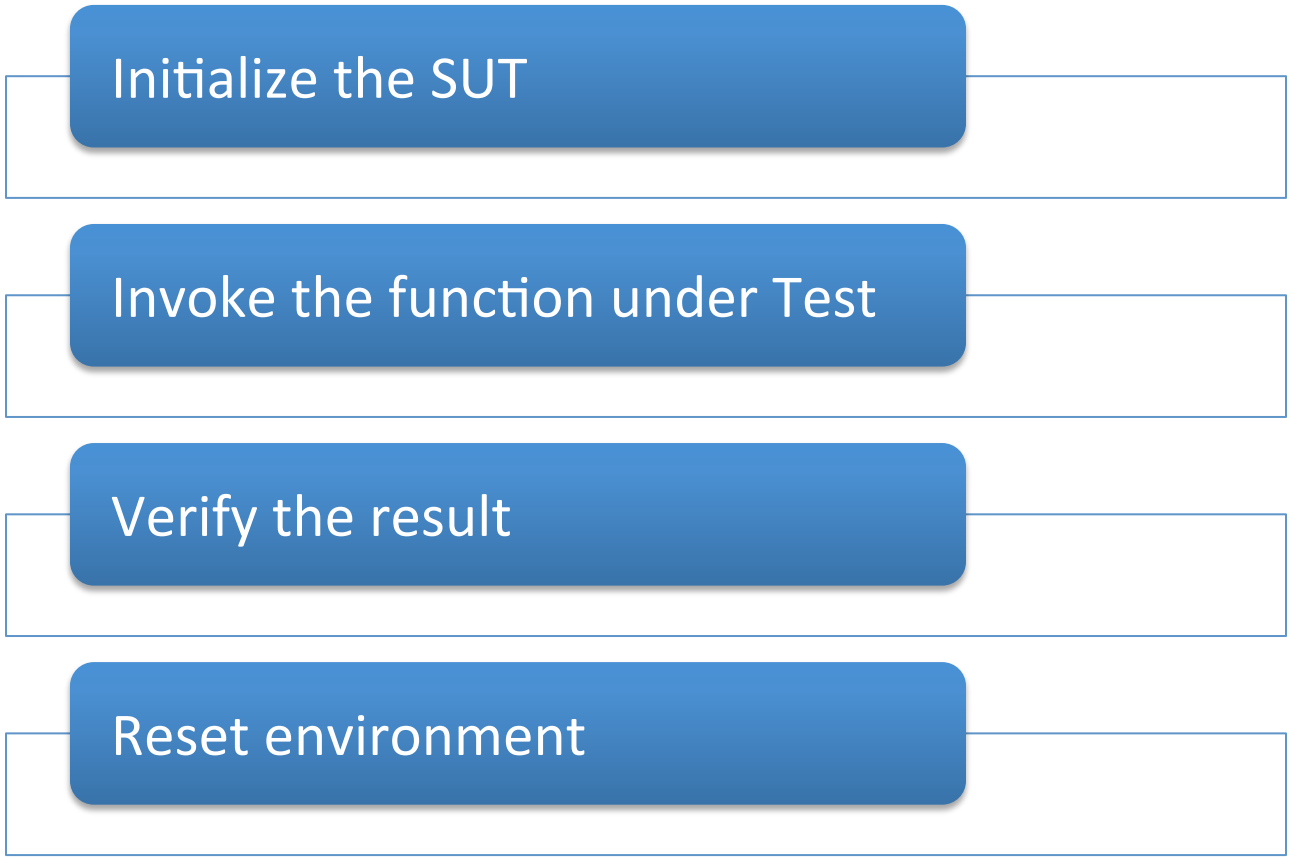
- Run the test {BasicAnnotationTestClass}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.BasicAnnotationTestClass

Check the sequence of print statements and match with what we discussed.

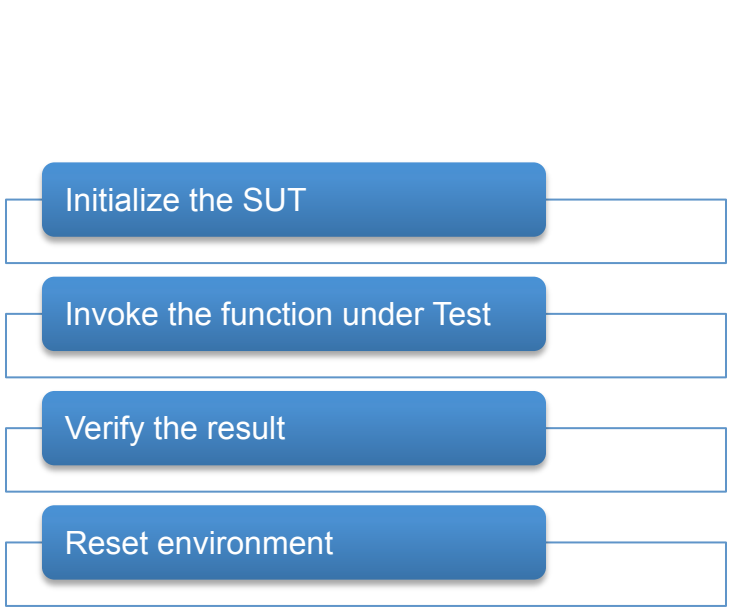
Confirm that in case you have multiple @Before and @After, the sequence of running these are not guaranteed.

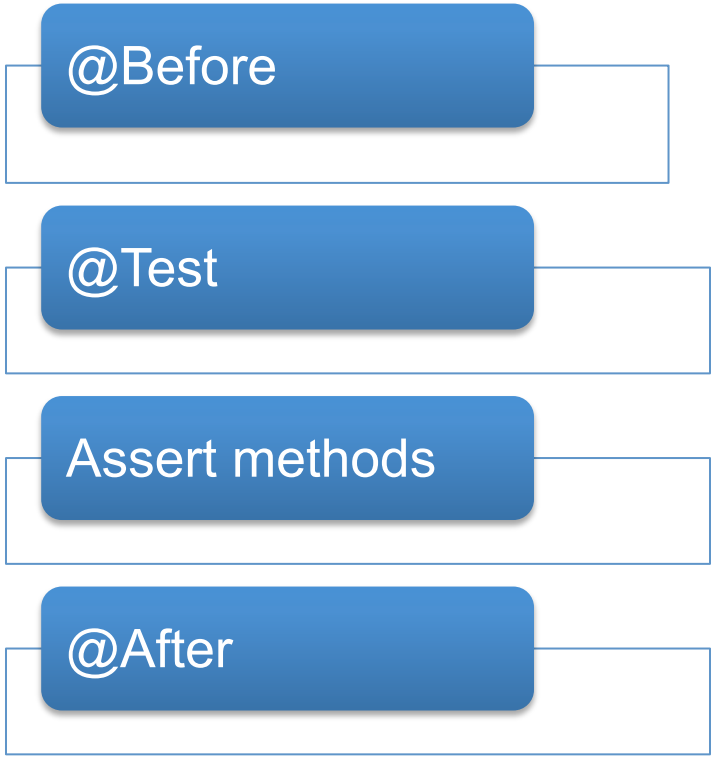
Four Phase Unit Tests Execution Strategy



Four Phase Unit Tests Execution Strategy

JUnit and Four Stage Execution Strategy

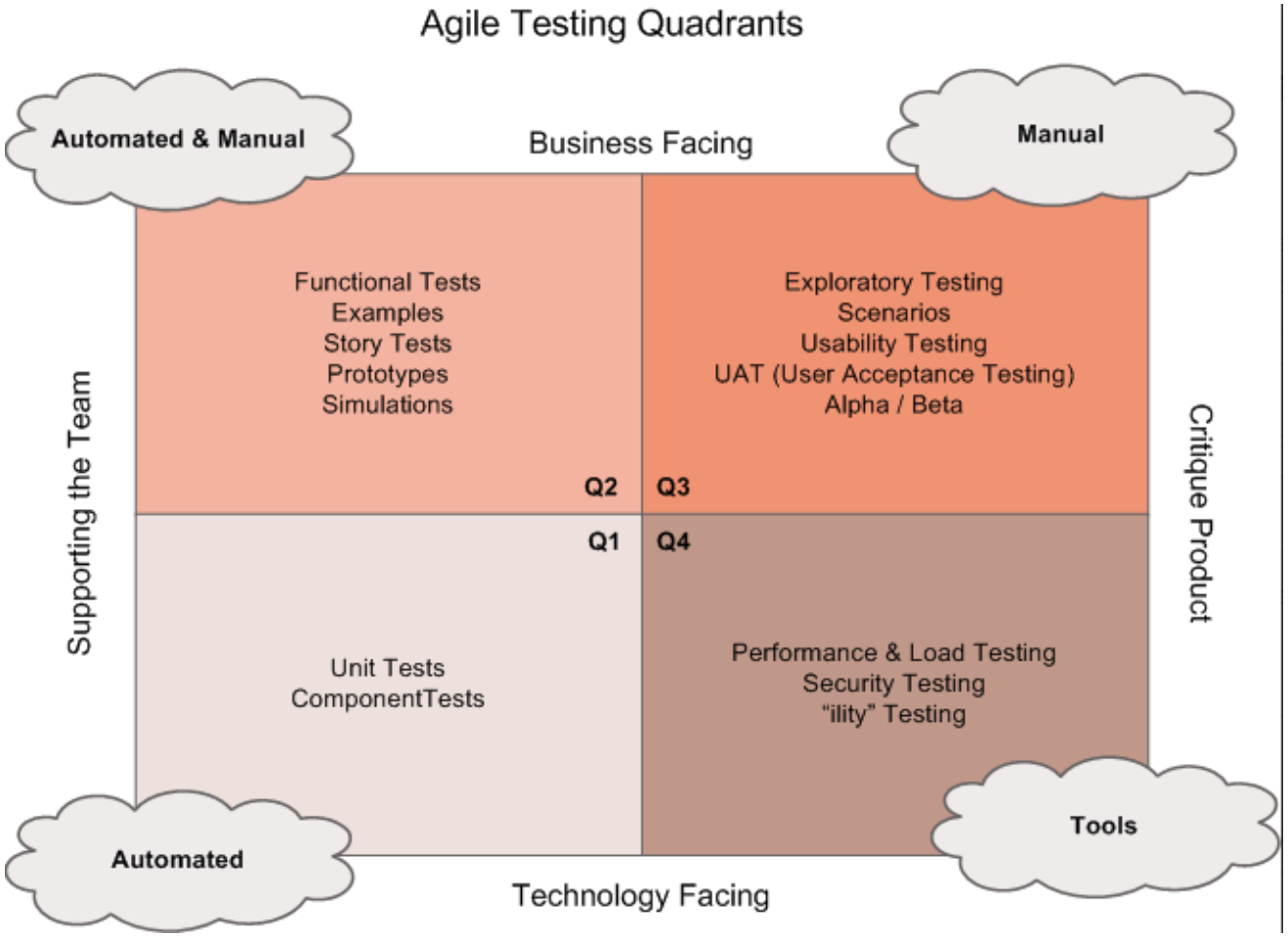




Qualities of Unit Testing

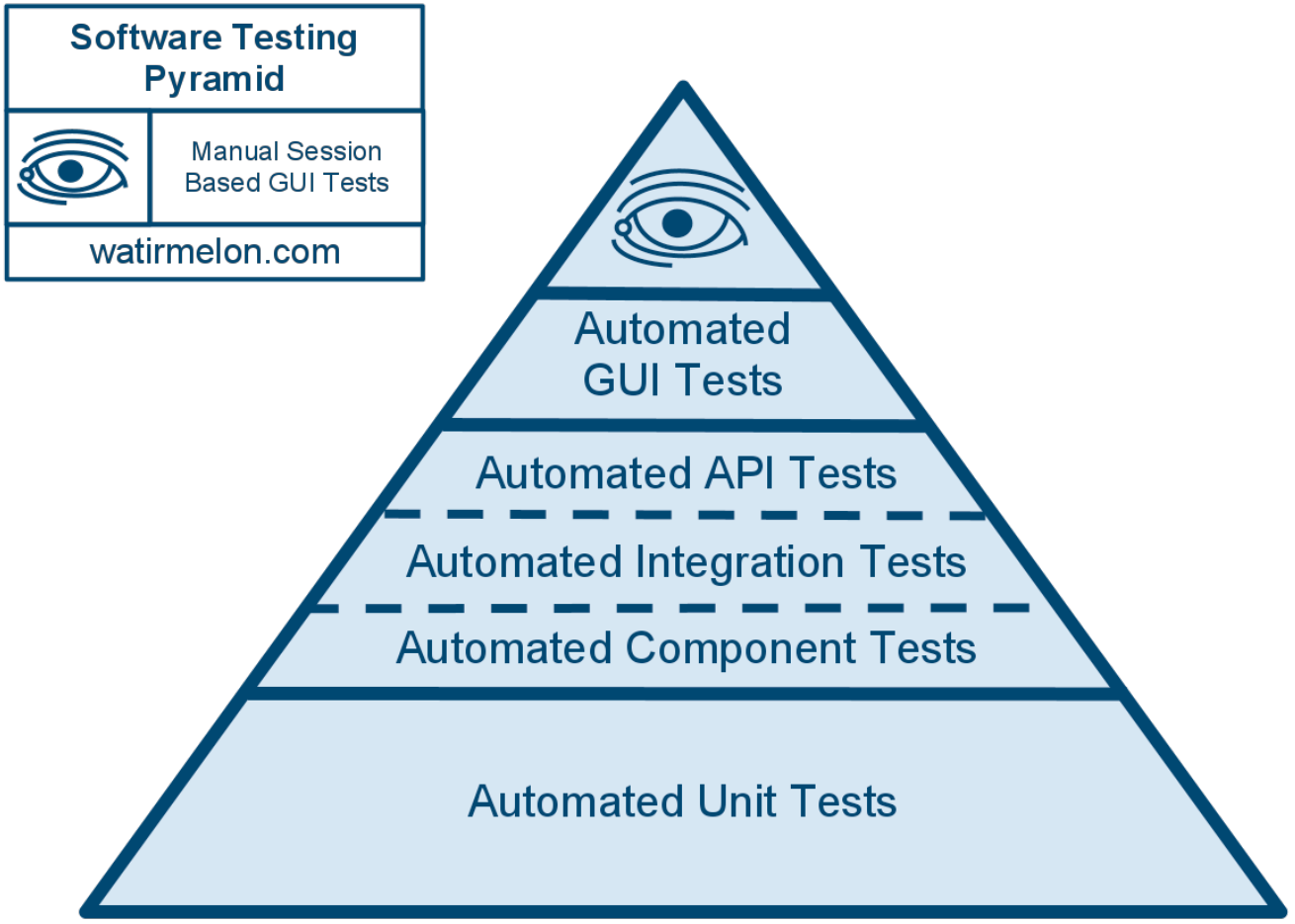
- **Order independent and isolated**: The test class ABCTest.java should not be dependent on the output of the test class XYZTest.java, or the test shouldn't fail if XYZTest.java is executed after ABCTest.java
- **Trouble-free setup and run**: Unit tests should not require DB connection or Internet connection or clean up temp directory
- **Effortless execution**: Unit tests should not be "It works fine on Server abc but doesn't run on my local"
- **Formula 1 execution**: A test should not take more than a second to finish the execution

Agile Testing Quadrant



http://lisacrispin.com/

Automation Pyramid



Tests as Documentation

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

-- Martin Fowler

"What makes a clean test? Three things. Readability, readability, and readability."

-- Uncle Bob

Tests as documentation

- Unit tests should provide lasting and trustworthy documentation on the capabilities of the classes.
- Tests provide opportunities to explain things that the code itself can't.
- Tests can supplant a lot of the comments you might otherwise feel compelled to write.
- Instead of suggesting what context you're going to test, you can suggest what happens as a result of invoking some behavior against a certain context.

Tests as documentation Example

Scope – Banking System

Not So Good Test Name	More descriptive name
makeWithdrawal	withdrawalReducesBalanceByWithdrawalAmount
overDraft	AttemptToWithdrawMoreThanAvailableBalanceGeneratesError
deposit	depositIncreasesBalanceByDepositAmount

Exercise : Culture and Testing Practices

15 Minutes

- Review the Agile testing Quadrant and note down practices that are applicable to your team or organization.
- What are different testing practices in your organization?
- Share your experience with others.

Exercise : Automated Test Exposure

15 Minutes

- Think about your deployment process and identify
  - Where and what kind of Automated Tests are being used
  - What are the shortcomings and room for improvements
  - Pain areas where auto tests are not used?
- What are different testing practices in your organization?
- Share your experience with others.

Reference


Copyrighted Material
Community Experience Distilled
Testing with JUnit
Master high-quality software development driven by unit tests

Reference

https://github.com/fappel/Testing-with-JUnit

Credit-

"Testing with Junit" book from Frank Appel

TEST DRIVEN DEVELOPMENT

Discussion: Test Driven Development

10 Minutes

- What are some important facts about Test Driven Development that you already know?
- What would you like to learn about test driven development?

Understanding TDD

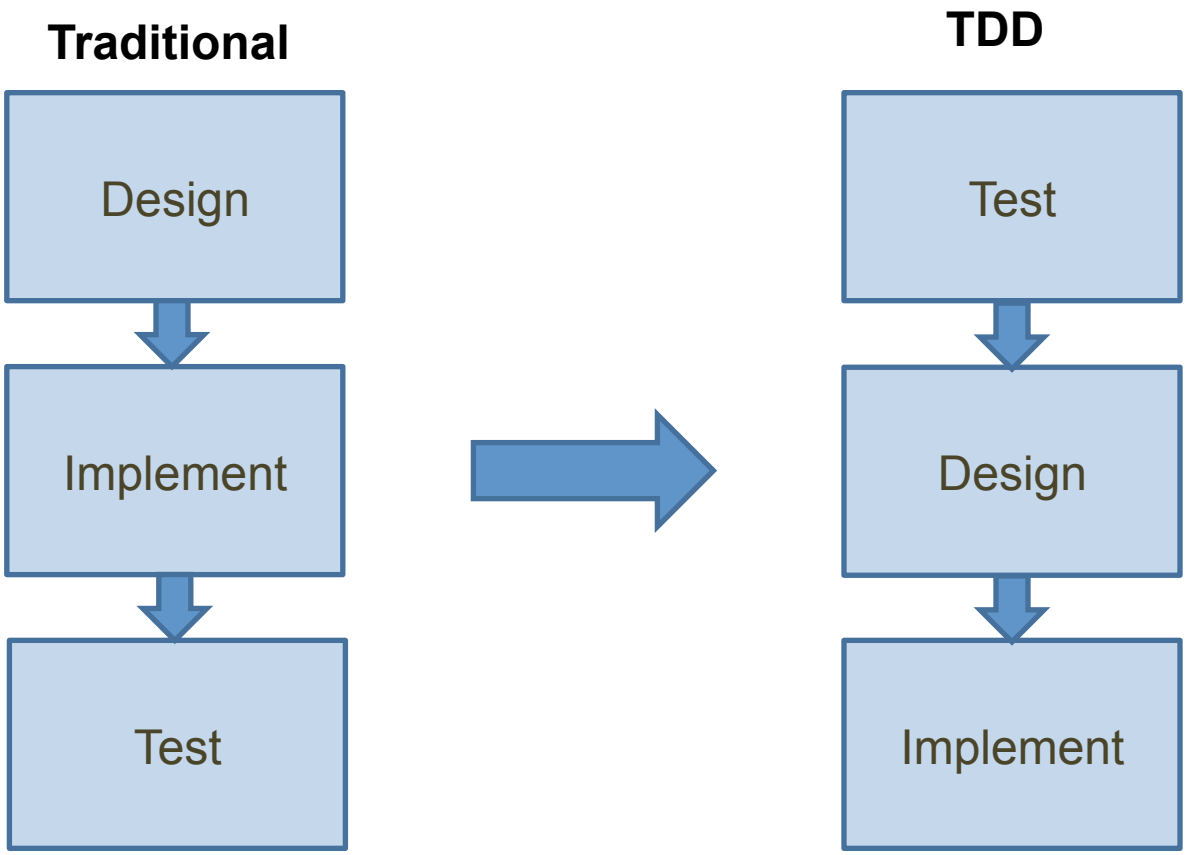
Many Names

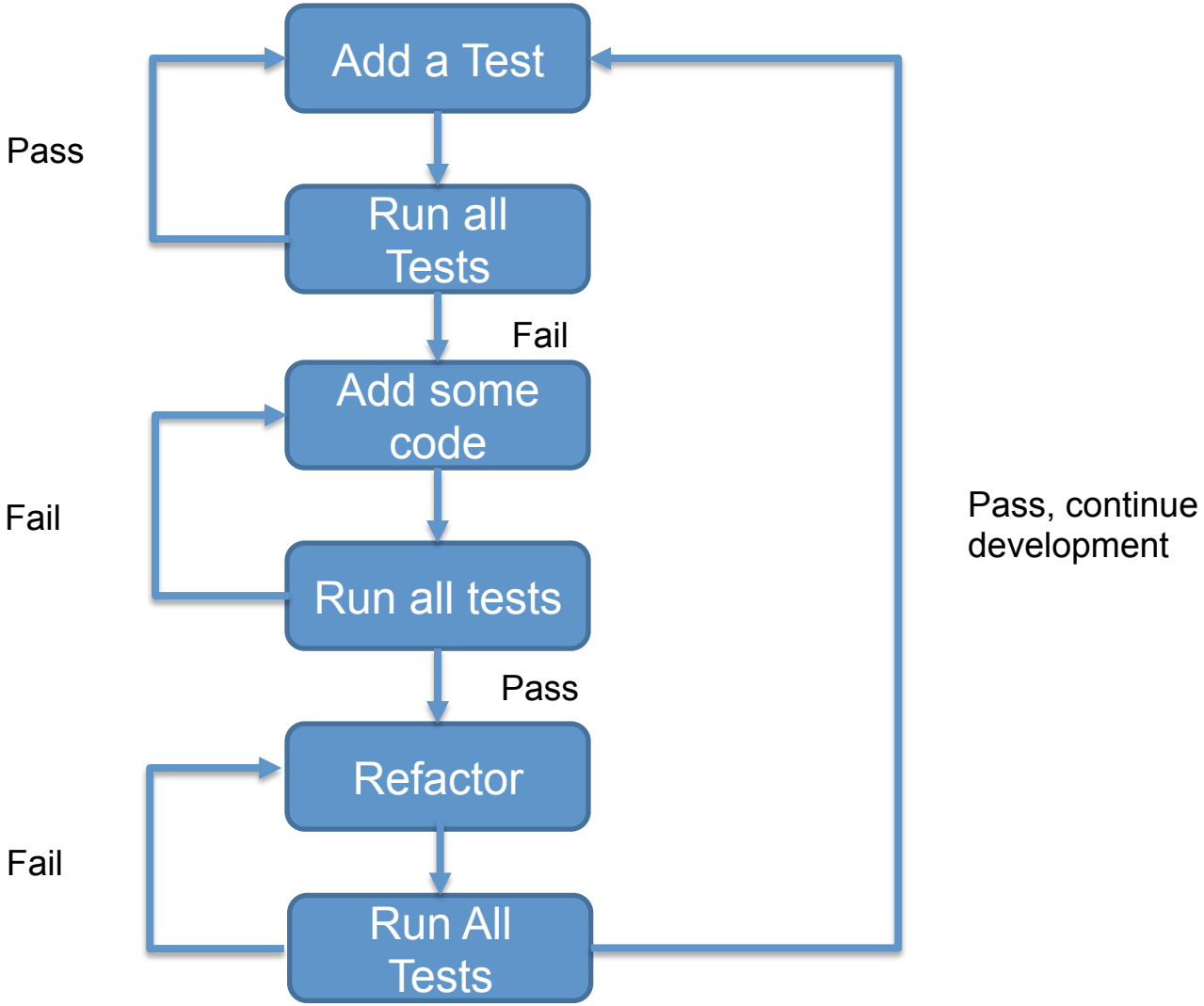
- Test Driven Development
- Test Driven Design
- Emergent Design
- Test First Development

Do you think about these !!!!

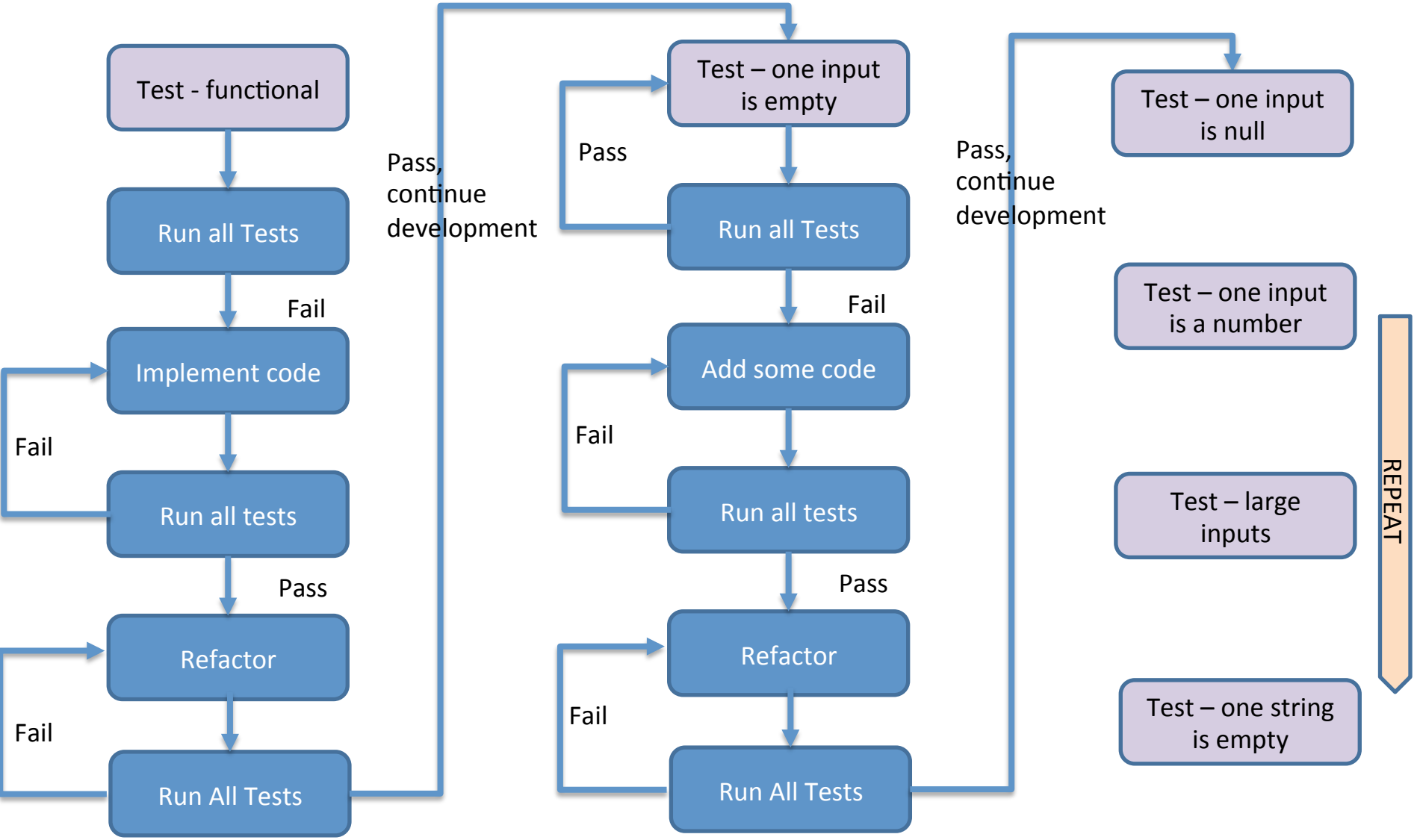
- How do you know your code works
- If code works, does it do what it should do
- Do you manually check it
- Step through the debugger to verify code works
- How often do you check output of the program (webpage or database or backend logic)
- **How do you check the output is correct**
- How much time do you spend in manual test
- How good are you in testing your own code
- **How confident are you that you covered everything**
- **How do you know the recent code didn't break code from last month**
- Cost of change grows exponentially with size of code

TDD – General Idea




Add a Test
Pass
Run all Tests
Fail
Add some code
Fail
Run all tests
Pass
Refactor
Fail
Run All Tests
Pass, continue development

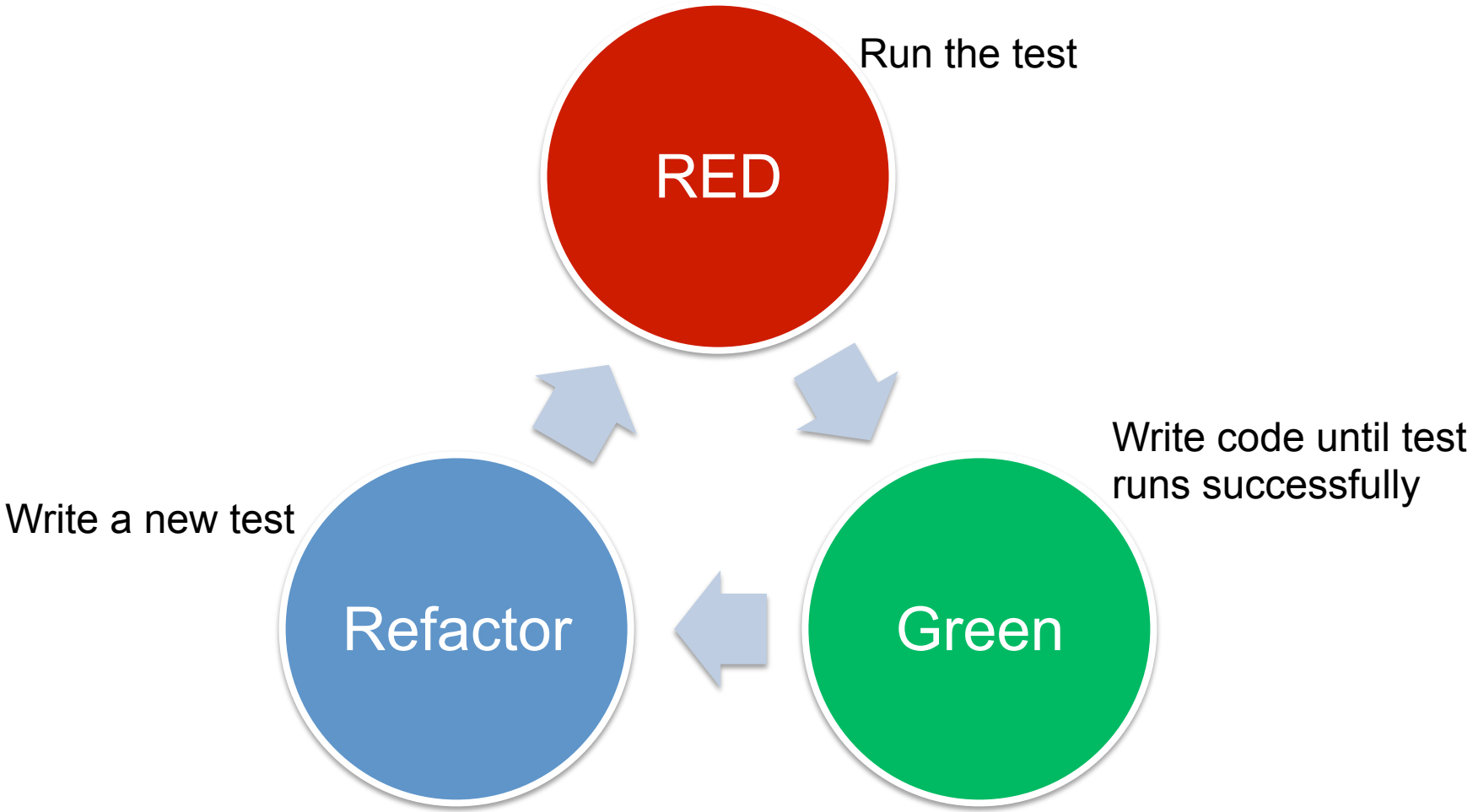
An Example – String Concatenate



TDD Cycle

- Think
- Write a test
  - Include all of the elements in the story that you imagine will be necessary to calculate the right answers
- Implement so that test passes (think about edge scenarios also)
- Run all unit tests and fix failures
- Refactor new code
- Run all unit tests and fix failures
- Check in your code

TDD CYCLE


RED
Run the test
Green
Write code until test runs successfully
Refactor
Write a new test

It's not about testing

- Test and design are One

- Test-driven development is the way we approach the design.
- It is the way to force us to think about the implementation and to what the code needs to do before writing it.
- It is the way to focus on requirements and implementation of just one thing at a time—organize your thoughts and better structure the code. This does not mean that tests resulting from TDD are useless—it is far from that.
- They are very useful and they allow us to develop with great speed without being afraid that something will be broken. **This is especially true when refactoring takes place.** Being able to reorganize the code while having the confidence that no functionality is broken is a huge boost to the quality.

Group Exercise: Test Driven Development

20 Minutes

- Is your complete team part of discussions about requirement, specification and product vision?
- Are you using manual testing?
- How much are you using automated tests?
  - Which tool and how much is the coverage?
- There is always something more urgent than dedicating time to testing. If this had happened did your team invested on testing later on?

- Do you feel the maintenance cost will be too high?
- Do you feel the time-to-market will be too big if you focus on testing?
- Is your documentation up to date ?

Why TDD

- Design the system by writing tests
- System is
  - Easy to use
  - Easy to test (after any new change)
- Resulting automated unit tests are easily run
  - Quickly and easily regression test
  - Fix bugs soon after they're introduced
- Confident Coders
  - If you break something, just fix or revert
- Less development time
- No bug is long lived
  - with every change, run all tests automated

Agony of Untestable Code

- One of the problems with unit tests especially when you're just getting started is that you might end up writing code that's difficult to test.
- For example, maybe you've got some internal state that you need to access, but you don't want to expose it. Or maybe your unit under test has a lot of complicated dependencies that are difficult to mock.
- Writing code that is testable requires experience, but how can you get that? Well, it turns out that *you don't need that experience if you start with TDD*. **When you write the tests first, you can't write untestable code.**

Rules for TDD

- Write new code only if an automated test has failed
- Eliminate duplication

Technical Implications of TDD

- Team must design organically, with running code providing feedback between decisions.
- We must write our own tests, because we can't wait 20 times per day for someone else to write a test.
- Our development environment must provide rapid response to small changes.
- Our designs must consist of many highly cohesive, loosely coupled components, just to make testing easy.

Lab 4: Let's design using TDD

90 Minutes

Let us implement java list using TDD

Coding exercise:
https://github.com/shekhar2010us/tdd/blob/master/list_impl.md

If someone do not want to code, you can just go through the code and run the test

$ cd ~/tdd

Implementation code:-
$ vi src/main/java/shekhar/tdd_list_impl/CustomList.java

Test Class:-
$ vi src/test/java/shekhar/tdd_list_impl/CustomListTest.java

Run the test:
$ mvn surefire:test -q -Dtest=shekhar.tdd_list_impl.CustomListTest

Inside-Out VS Outside-In

TEST DRIVEN DEVELOPMENT

Inside-Out

Inside Out TDD allows the developer to focus on one thing at a time.

As code evolves through refactoring - new collaborators, interactions and other components appear. TDD guides the design completely.

Outside-In

Outside-In involves using acceptance test and upfront design to drive the implementation details you need to deliver business features.

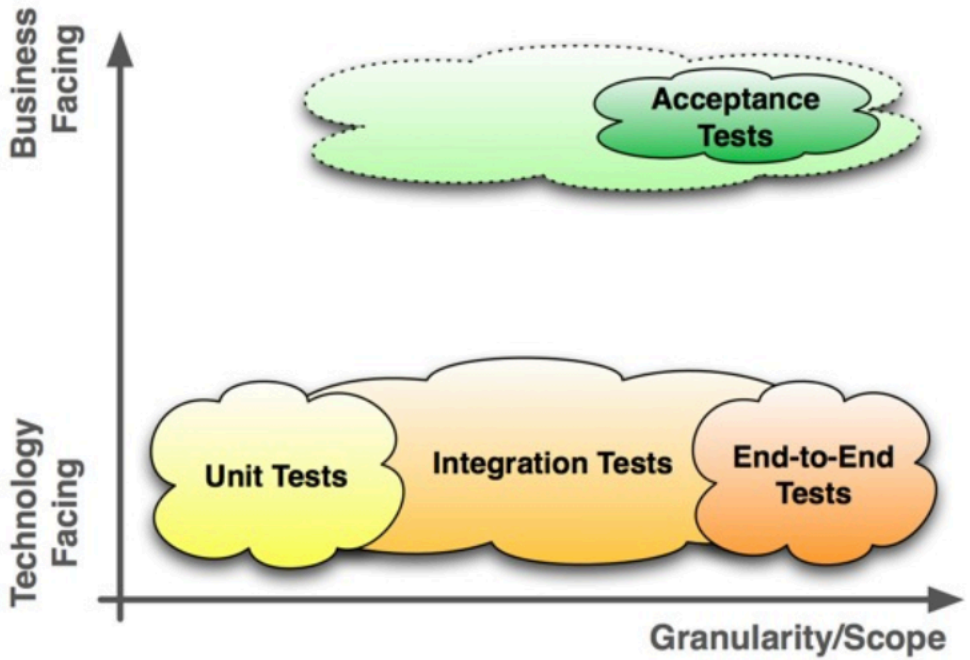
The process typically iterates over the following steps:

- Start with a high-level acceptance criterion and Automate the acceptance criterion .
- Imagining the code you'd like to have to make each step work, make some lean design decision.
- Use these step definitions to flesh out unit tests that specify how the application code will behave.
- Implement the application code, and refactor as required.

Inside-Out VS Outside-In

Both approaches are being used by different organizations and teams depending on what they need. In large, enterprise solutions, where parts of the design come from architects (or exists upfront) one might start with ”Outside-In" approach.

When you're not sure how your code should look, it might be easier to start with some low-end component using Inside-Out and let it evolve as more tests, refactoring and requirements are introduced.


Business Facing
Acceptance Tests
Technology Facing
Unit Tests
Integration Tests
End-to-End Tests
Granularity/Scope

Acceptance Tests

```
Feature: Timeline UI Display

  Scenario: Mike opens Timeline UI
    Given class has posted atleast 10 items
    When Mike opens Timeline UI
    Then Mike Should see 10 items on timeline

    Scenario: Mike clicks on Load More Updates button
      Given Mike is on Timeline UI
      When Mike clicks on Load More Updates button
      Then Mike should see 20 items on Timeline
```

Acceptance tests can be set on different levels of granularity.
This means an acceptance test can be a unit-, integration or end-to-end test.

Integration Test

```
Feature: Timeline display and ordering of updates


  Scenario: Timeline Display order
    Given Timeline Database has atleast 10 items
    When Timeline is initialized by consumer
    Then  timeline should be loaded with 10 items from database
        And items should be sorted in descending order of date-time when items were created

  Scenario: Consumer requests to fetch more items
    Given Database has more then 20 items and timeline has been initialized
    When consumer requests more items
    Then Timeline should be loaded with 10 more items
      And total item count should be 20
```

So far we have discussed

- Manual unit testing
- Automated unit testing using Junit
- Test annotation and hierarchy
- TDD and TDD cycle
- Java Implementation of List using TDD
- Inside-out Vs Outside-in

Next we will see

- Test doubles
  - Dummy
  - Fake
  - Mock
  - Stub

Developing Independently Testable Units

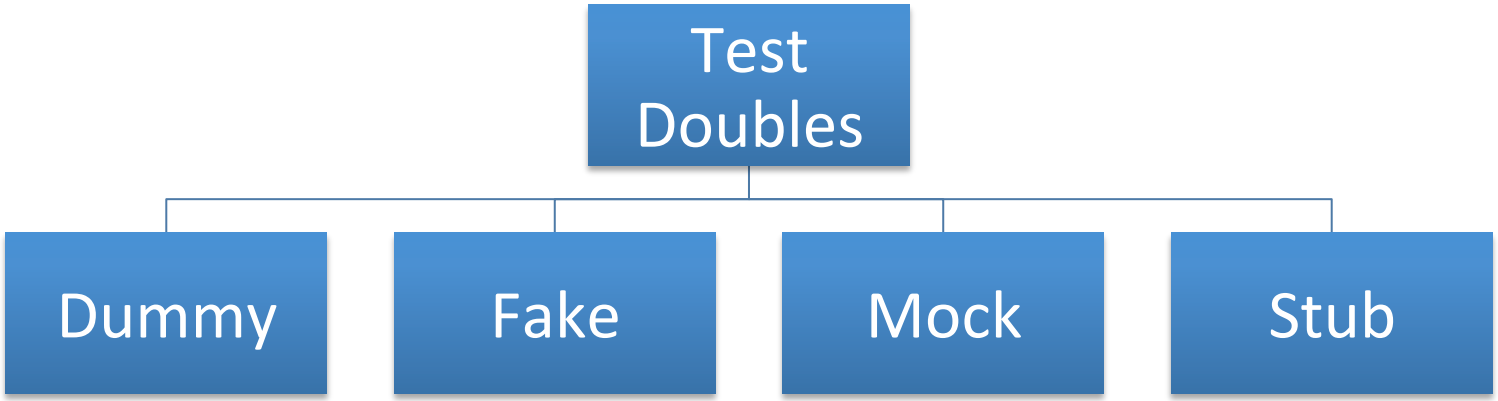
TEST DRIVEN DEVELOPMENT

Test Doubles

Sometimes it is not possible to unit test a code because of the unavailability of collaborator objects or the cost of instantiation for the collaborator.

Test Double is a generic term for any case where you **replace a production object for unit testing purposes.**

Test Doubles



Dummy

Dummy is used in unit testing when their presence is required, like a mandatory parameter for method. These objects are passed around but never actually used. Usually they are just used to fill parameter lists

Dummies are generally programmed to throw error if their behavior/method is called.

Dummy Example

```java
public interface UserLocation {

    String getCurrentLocation(  );

}
```

```java
public class UserLocationDummy implements UserLocation {

    private static String MESSAGE = "Dummy method must never be called.";
    @Override
    public String getCurrentLocation( ) {
        System.out.println(MESSAGE);
        throw new UnsupportedOperationException( MESSAGE );
    }
}
```

```java
import org.junit.Test;
import shekhar.tdd.dummies.UserLocationDummy;

public class Dummy_TimelineTest {
    @Test
    public void initializeWithDummyUserLocation() {
        UserLocationDummy userLocationDummy = new UserLocationDummy();
        Timeline timeline = new Timeline(userLocationDummy);
        userLocationDummy.getCurrentLocation();
    }
}
```

Fake

Fake objects are working implementations; the class generally extends the original class or implements an interface, but usually performs a hack, which makes it unsuitable for production.

Replace the component with lighter weight implementations

Example:- In-memory database

Fake Example

```java
public interface LocationDatabase {
    void save(String key, String loc);
    String get(String key);
}
```

```java
public class LocationDatabaseFake implements LocationDatabase {
    public Map<String, String> map = new HashMap<>();
    @Override
    public void save(String key, String loc) { map.put(key, loc); }
    @Override
    public String get(String key) { return map.get(key); }
}
```

```java
public class Fake_TimelineTest {
    protected LocationDatabaseFake locationDatabaseFake = null;
    @Before
    public void setup() throws Exception {
        locationDatabaseFake = new LocationDatabaseFake();
        locationDatabaseFake.save( key: "home",  loc: "tv room");
    }
    @Test
    public void initializeWithFakeUserLocation() throws Exception {
        assertEquals( expected: "tv room", locationDatabaseFake.get("home") );
    }
    @After
    public void tearDown() throws Exception {
        locationDatabaseFake = null;
    }
}
```

Fake

```
public class FakeItem implements Item {
    long timeStamp;
    String message;
```

```
FakeItem FAKE_ITEM_ONE=new FakeItem( timeStamp: 10);
FakeItem FAKE_ITEM_TWO=new FakeItem( timeStamp: 20);
FakeItem FAKE_ITEM_THREE=new FakeItem( timeStamp: 30);
FakeItem FAKE_ITEM_FOUR=new FakeItem( timeStamp: 40);
FakeItem FAKE_ITEM_FIVE=new FakeItem( timeStamp: 50);
FakeItem FAKE_ITEM_SIX=new FakeItem( timeStamp: 60);
FakeItem FAKE_ITEM_SEVEN=new FakeItem( timeStamp: 70);
```

Stub

A stub delivers indirect inputs to the caller when stub's methods are called. Stubs are programmed only for the test scope.

A Fake is closer to a real-world implementation than a stub. Stubs contain basically hard-coded responses to an expected request; they are commonly used in unit tests, but they are incapable of handling input other than what was pre-programmed.

Fakes have a more real implementation, like some kind of state that may be kept for example. They can be useful for system tests as well as for unit testing purposes, but they aren't intended for production use because of some limitation or quality requirement.

Stub Example

```
public interface DeviceLocationInfo {
    public boolean isLocationEnabled();
}
```

```
public class DeviceLocationInfoStub implements DeviceLocationInfo {
    @Override
    public boolean isLocationEnabled() {
        return true;
    }
}
```

```
public class Stub_TimelineTest {
    protected DeviceLocationInfoStub deviceLocationInfoStub = null;

    @Before
    public void setup() throws Exception {
        deviceLocationInfoStub = new DeviceLocationInfoStub(); }
    @Test
    public void initializeWithStubUserLocation() throws Exception {
        assertTrue(deviceLocationInfoStub.isLocationEnabled() ); }
    @After
    public void tearDown() throws Exception {
        deviceLocationInfoStub = null; }
}
```

Mock

Mock objects have expectations, a test **expects** a value from a mock object and during execution, the mock object returns the expected result.

Good for verifying exact behavior

Good for strict TDD

When to Mock

- The object generates deterministic results.
- The object is slow and requires time to process.
- The most common example would be databases interaction.

Mockito

Mockito is a mocking framework with a clean and simple API. Tests produced with Mockito are readable, easy-to-write, and intuitive. It contains three major static methods:

- **mock**(): This is used to create mocks. Optionally, we can specify how those mocks behave with when() and given().
- **verify():** This is used to check whether methods were called with given arguments. It is a form of assert.
- **spy():** This can be used for partial mocking. Spied objects invoke real methods unless we specify otherwise. As which mock(), behavior can be set for every public or protected method (excluding static). The major difference is that mock() creates a fake of the whole object, while spy() uses the *real* object

Mockito Example

Interface :- This doesn't have any implementation

```
public interface ClosestUsers {
    public double distanceBetweenTwoUsers(double lat1, double lng1, double lat2, double lng2);
}
```

Service:- This use the interface that doesn't have any implementation

```
public class ClosestUserService {
    ClosestUsers closestUsers;

    // Note that "ClosestUsers" interface has no implementation;
        // the mock of the same will be created in the test class
    public double closenessOfTwoUsers(double lat1, double lng1, double lat2, double lng2) {
        return closestUsers.distanceBetweenTwoUsers(lat1, lng1, lat2, lng2);
    }

    public ClosestUsers getClosestUsers() { return closestUsers; }
    public void setClosestUsers(ClosestUsers _closestUsers) { this.closestUsers = _closestUsers; }

}
```

Mockito Example Contd.

Test:- Creates a mock of the interface that did not have an implementation

```
public class Mock_TimelineTest {

    ClosestUserService closestUserService;

    @Before
    public void setup() {
        closestUserService = new ClosestUserService();

        ClosestUsers closestUsers = Mockito.mock(ClosestUsers.class);
        when(closestUsers.distanceBetweenTwoUsers( lat1: 40.109166,
                lng1: -75.392951,  lat2: 40.083297,  lng2: -75.404281)).thenReturn(5.5);
        closestUserService.setClosestUsers(closestUsers);
    }

    // Note that "ClosestUsers" interface has no implementation;
    // the mock of the same will be created in the test class
    @Test
    public void testDistanceBetweenTwoUsers() {
        assertEquals( expected: 5.5, closestUserService.closenessOfTwoUsers( lat1: 40.109166,
                lng1: -75.392951,  lat2: 40.083297,  lng2: -75.404281),  delta: 0.5);
    }

}
```

Lab 5: Test Doubles

30 Minutes

DUMMY

- cd to the {tdd} project $$ cd ~/tdd
- Check code of the interface

  $$ vi src/main/java/shekhar/tdd/dummies/UserLocation.java
- Check code of the implemented class

  $$ vi src/test/java/shekhar/tdd/dummies/UserLocationDummy.java
- Check code of the Test class

  $$ vi src/test/java/shekhar/tdd/Dummy_TimelineTest.java

Uncomment the last line of the test class and the run the test. It will return you the dummy message

- Run the test {Dummy_TimelineTest}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.Dummy_TimelineTest

*** Note: make sure you comment the last line again

Lab 5: Test Doubles Contd.

FAKE

- Check code of the interface

  $$ vi src/main/java/shekhar/tdd/fakes/LocationDatabase.java

- Check code of the implemented class

  $$ vi src/test/java/shekhar/tdd/fakes/LocationDatabaseFake.java

- Check code of the Test class

  $$ vi src/test/java/shekhar/tdd/Fake_TimelineTest.java

- Run the test {Fake_TimelineTest}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.Fake_TimelineTest

Lab 5: Test Doubles Contd.

STUB

- Check code of the interface

  $$ vi src/main/java/shekhar/tdd/stubs/DeviceLocationInfo.java

- Check code of the implemented class

  $$ vi src/test/java/shekhar/tdd/stubs/DeviceLocationInfoStub.java

- Check code of the Test class

  $$ vi src/test/java/shekhar/tdd/Stub_TimelineTest.java

- Run the test {Stub_TimelineTest}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.Stub_TimelineTest

Lab 5: Test Doubles Contd.

MOCK

- Check code of the interface

  $$ vi src/main/java/shekhar/tdd/mocks/ClosestUsers.java

- Check code of the implemented class

  $$ vi src/test/java/shekhar/tdd/mocks/ClosestUserService.java

- Check code of the Test class

  $$ vi src/test/java/shekhar/tdd/Mock_TimelineTest.java

- Run the test {Mock_TimelineTest}

  $$ mvn surefire:test -q -Dtest=shekhar.tdd.Mock_TimelineTest

Test Suite

- Used to bundle few unit test cases and run them together

```
public class MessageUtil {

   private String message;
   public MessageUtil(String message) {
      this.message = message; }

public String printMessage() {
      System.out.println(message);
      return message;
   }
}
```

```
public class TestJunit1 {

   String message = ”TDD-1";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      assertEquals(message, messageUtil.printMessage());
   }
}
```

```
public class TestJunit2 {

   String message = ”TDD-2";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      assertEquals(message, messageUtil.printMessage());
   }
}
```

Test Suite

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
   TestJunit1.class,
   TestJunit2.class
})

public class JunitTestSuite {
}
```

```
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(JunitTestSuite.class);

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

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

JUnit Integration Test

- In JUnit - **@category** – to label and group test

```
// Create a marker interface
public interface FastTest{}
```

```
// Create a marker interface
public interface SlowTest{}
```

```
import org.junit.experimental.categories.Category;

// Use the marked interface
@Category(FastTest.class)
public class A1{

 @Test
 public void longRunningServiceTest() throws
Exception {}
}
```

```
import org.junit.experimental.categories.Category;

// Use the marked interface
public class A2{

@Category(SlowTest.class)
 @Test
 public void longRunningServiceTest() throws
Exception {}
}
```

Test Suite with @Category

```
@RunWith(Categories.class)
@IncludeCategory(SlowTest.class)
@ExcludeCategory(FastTest.class)
@SuiteClasses({ A.class, B.class })
public class SlowTestSuite {}
```

```
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(SlowTestSuite.class);

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

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