Workshop TDD

Workshop Overview

- ➔ Unit Tests
- ➔ Introduction to TDD
- ➔ Overview of Mocks and Stubs
- ➔ Deep Dive into Mocks
- ➔ Testing Legacy Code
- ➔ Coverage Metrics and Tools

Workshop Overview

Overview

- ➔ **Testing Database SP/Triggers**
- ➔ **Testing Controllers/Rest/SOAP**
- ➔ **Integration Tests**
- ➔ **Complete Use Case**

Machine & Code Setup

- ➔ **Clone :- https://github.com/PratikGarg/tdd-workshop.git**
  OR Download Zip :- https://github.com/PratikGarg/tdd-workshop.git

- ➔ **Run : mvn clean install**
- ➔ **Import Project in Eclipse**
- ➔ **Workspace walkthrough**

Chapter 1

Introduction to Unit Tests

What is a Test or Unit Test?

- Test is a means to determine the presence, quality or truth of something
- Definition of Unit is a situational thing. For object oriented design tends to treat a class as a Unit, procedural or functional approaches consider a single function as a unit.
- So Unit Test essentially means testing a unit of functionality.

How to write a unit Test

Demo.

Guidelines for Unit Tests

- Unit tests should run fast.
- Unit tests should be environment independent.
- Unit test should not have any side effects
  - Execution order of tests should not matter.
- Name your unit tests clearly and consistently.
  - They should also act as documentation to your code.
- Make each test orthogonal (i.e., independent) to all the others.
  - Test only one code unit at a time
  - Don’t make unnecessary assertions
  - Mock out all external services and state
  - Avoid unnecessary preconditions

Boundary Conditions

- **C**onformance — Does the value conform to an expected format?
- **O**rdering — Is the set of values ordered or unordered as appropriate?
- **R**ange — Is the value within reasonable minimum and maximum values?
- **R**eference — Does the code reference anything external that isn't under direct control of the code itself?
- **E**xistence — Does the value exist? (e.g., is non-null, non-zero, present in a set, etc.)
- **C**ardinality — Are there exactly enough values?
- **T**ime (absolute and relative) — Is everything happening in order? At the right time? In time?

source : Pragmatic Unit Testing

Benefits of Unit Tests

- Unit tests prove that your code actually works.
- Probability of finding early bugs is significantly high.
- Refactor the design without breaking it.
- They act as a sample code and documentation for your code.
- Increase confidence in code.

Junit & Annotations

Hamchrest Matchers

Hamcrest is a framework for writing matcher objects allowing 'match' rules to be defined declaratively

Theories and DataPoints

Parameterized Tests

Exercise.

End Chapter 1

Chapter 2

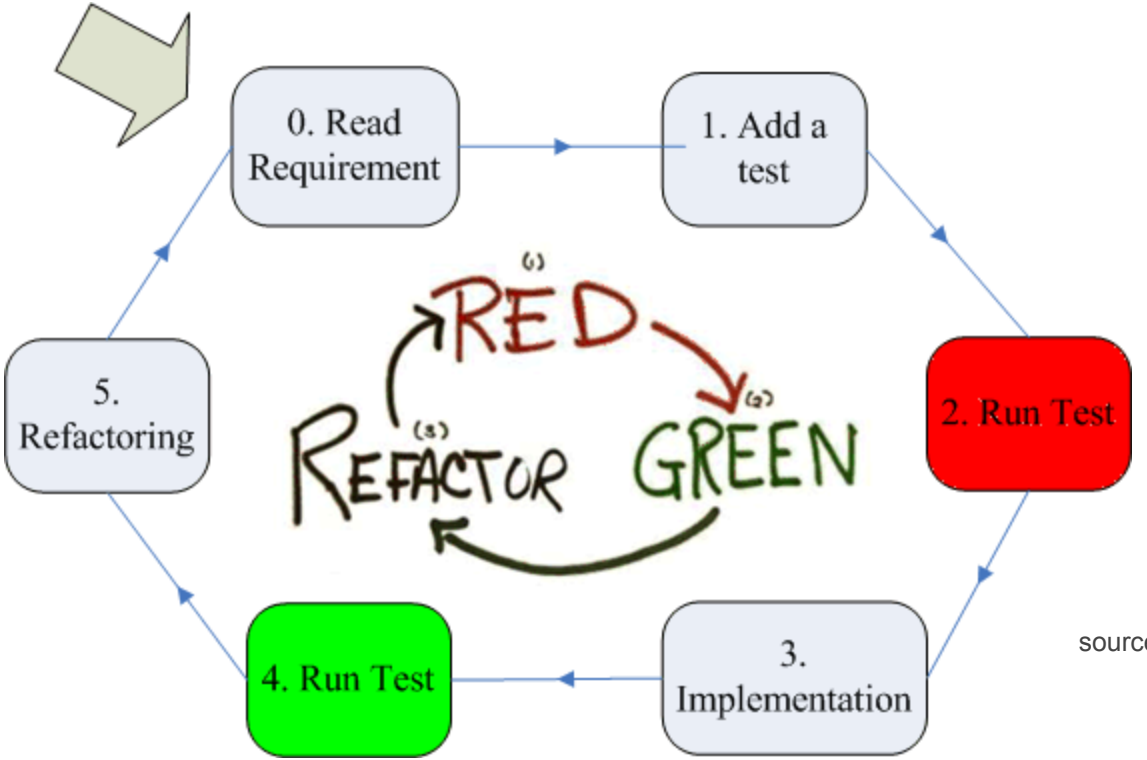
Introduction to TDD

What is TDD ?

- TDD is an evolutionary approach to development which combines **test-first development** where you write a test before you write just enough production code to fulfill that test and **refactoring**.

- Test-driven development (TDD) is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies. The result of using this practice is a comprehensive suite of unit tests that can be run at any time to provide feedback that the software is still working. This technique is heavily emphasized by those using Agile development methodologies

Doing TDD



source : javornikolov.github.io

Demo

Ping Pong Concept

A technique called “ping-pong programming” when one person writes a test, and another implements it. Then they switch.

Benefits of TDD

- Forces us to make our code testable.
- You will write more meaningful test cases covering more critical paths and not just for the code coverage.
- Better design of application.
- Speeds up development.

Exercise.

End Chapter 2

Chapter 3

Mocks & Stubs

Mocks & Stubs

Stubs : Stubs are fake objects generally used to contain data required for a test scenario.

Mock : Mock objects are used to define expectations i.e: In this scenario I expect method A() to be called with such and such parameters. Mocks record and verify such expectations.

What to Mock

- Objects or pieces of code you don’t need to test
- A good rule of thumb is to mock as close to the source as possible
- If you have a function call that calls an external API, mock out the external API, not the whole function
- Similarly, don’t mock return values when you could construct a real return value of the correct type with the correct attributes

What to Mock

- If an actual object has any of the following characteristics, it may be useful to use a mock object in its place:
  - the object supplies non-deterministic results (e.g., the current time or the current temperature)
  - it has states that are difficult to create or reproduce (e.g., a network error)
  - it is slow (e.g., a complete database, which would have to be initialized before the test)
  - it does not yet exist or may change behavior

Overusing Mocks

Problems with overusing mocks

- Tests can be harder to understand
  - Need to include extra code to tell the mocks how to behave
  - Having this extra code detracts from the actual intent of what you’re trying to test
- Tests can be harder to maintain
  - When you tell a mock how to behave, you're leaking implementation details of your code into your test. When implementation details in your production code change, you'll need to update your tests to reflect these changes

How much to Mock

- Some signs that you're overusing mocks
  - Mocking out more than two classes
  - Mocks specifies how more than two methods should behave

Best Practices

- Don’t mock your model
- Don’t mock servlet api
  - org.springframework.mock.web
- Don’t replace asserts with verify
  - If it doesn’t contain asserts, and instead contains a long list of verify() calls, then you’re relying on side effects
  - This is a unit-test bad smell, especially if there are several objects that need to be verified

Best Practices

- Mock across architecturally significant boundaries. For example, mock out the database, web server, and any external service

Common Frameworks

mockito

jmockit

A developer testing toolkit for Java

EASYMOCK

mockachino

A lightweight mocking framework for Java

Demo

End Chapter 3

Chapter 4

Deep Dive into Mocking

State Vs Interaction

Argument Matchers

Mock Exceptions

Execution Order

End Chapter 4

Chapter 5

Coverage Metrics and Tools

Code Coverage

- Code coverage consists of a set of software metrics that can tell you how much of the production code is covered by a given test suite.

- It's purely quantitative, and does not say anything about the quality of either the production code or the test code. That said, the examination of code coverage reports will sometimes lead to the discovery of unreachable code which can be eliminated. But more importantly, such reports can be used as a guide for the discovery of missing tests.

- This is not only useful when creating tests for existing production code, but also when writing tests first, such as in the practice of TDD (Test Driven Development).

source : http://jmockit.googlecode.com

Cobertura

Demo.

Line Coverage

- The *line coverage* metric tells us how much of the *executable code* in a source file has been exercised by tests. Each executable line of code can be *uncovered*, *covered*, or *partially covered*.

- In the first case, *none* of the executable code in it was executed at all. In the second, *all* of the code was fully executed at least once. In the third case, only *part* of the executable code in the line was executed. This can happen, for example, with lines of code containing multiple logical conditions in a complex boolean expression.

source : http://jmockit.googlecode.com

Branch Coverage

- A *branching point* exists wherever the program makes a decision between two possible execution paths to follow. Any line of code containing a logical condition will be divided in at least two executable *segments*, each belonging to a separate *branch*.

- An executable line of source code with no branching points contains a single segment. Lines with one or more branching points contain two or more executable segments, separated by consecutive branching points in the line.

source : http://jmockit.googlecode.com

Line Vs Branch

```
public int updateUser(boolean isNewUser) {
     User user = null;
     if (isNewUser) {
       user = new John();
      }
  return user.getName().length();
}
```

source : http://jmockit.googlecode.com

End Chapter 5!

Chapter 6

Testing Legacy Code

What do you mean by Legacy Code?

When do you encounter it ?

- Bug Fixes
- New Features
- Re Factoring

Scenario’s

Pre DI Era

Mock Public Methods

Mock Private Methods

Mock Static Methods

Mock Constructors

Exercise.

End Chapter 6

Chapter 7

DAOTest

H2 Database

- In Memory Database
- Very fast, open source
- Small footprint : ~1.5MB jar
- Compatibility Modes
  - DB2
  - Derby
  - HSQLDB
  - Oracle
  - PostgreSQL
  - MS SQL
  - MySQL
- Supported Platforms
  - Windows, Mac OS X and Linux

H2 Database

- In some cases, only one connection to a in-memory database is required; database URL is “jdbc:h2:mem”
  - Opening two connections within the same virtual machine means opening two different (private) databases
- Sometimes multiple connections to the same in-memory database are required; database URL is “jdbc:h2:mem:db1”
  - Accessing the same database using this URL only works within the same virtual machine
- To access an in-memory database from another process or from another computer, you need to start a TCP server in the same process as the in-memory database was created; “jdbc:h2:tcp://localhost/mem:db1”

H2 Database

- By default, closing the last connection to a database closes the database
- For an in-memory database, this means the content is lost. To keep the database open, add ;DB_CLOSE_DELAY=-1 to the database URL
- To keep the content of an in-memory database as long as the virtual machine is alive, use “jdbc:h2:mem:test;DB_CLOSE_DELAY=-1”

Demo

Function & SPs

- H2 database supports user-defined Java functions. Java functions can be used as stored procedures as well
- A function must be declared (registered) before it can be used
- A function can be defined using source code, or as a reference to a compiled class that is available in the classpath
- Only static Java methods are supported; both the class and the method must be public

Demo

End Chapter 7

Chapter 8

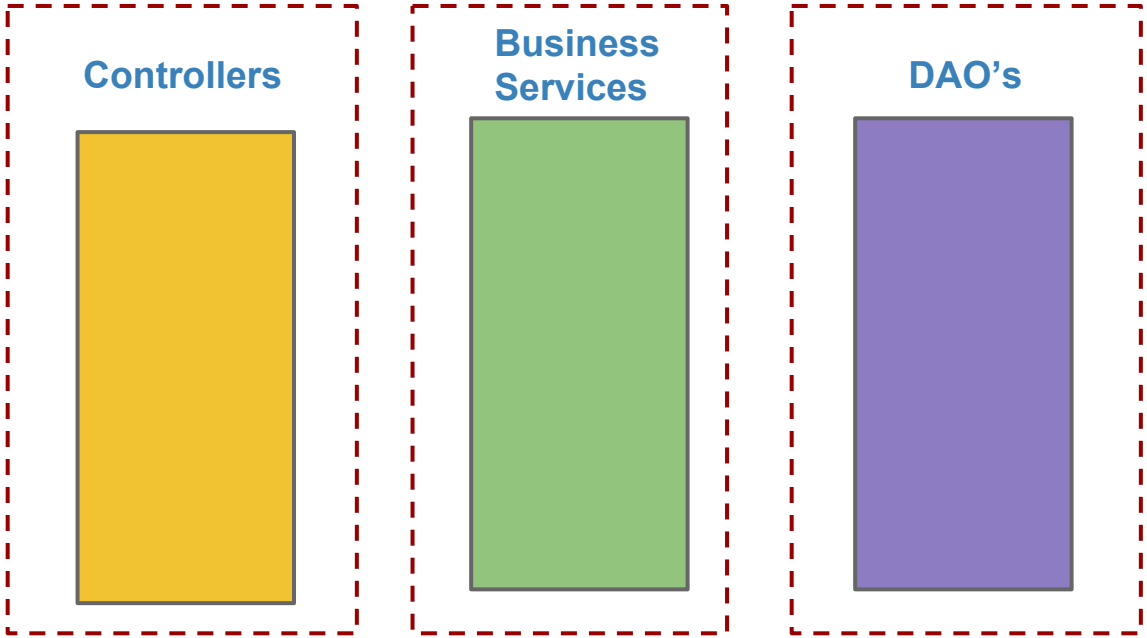
REST/SOAP/Controllers

End Chapter 8

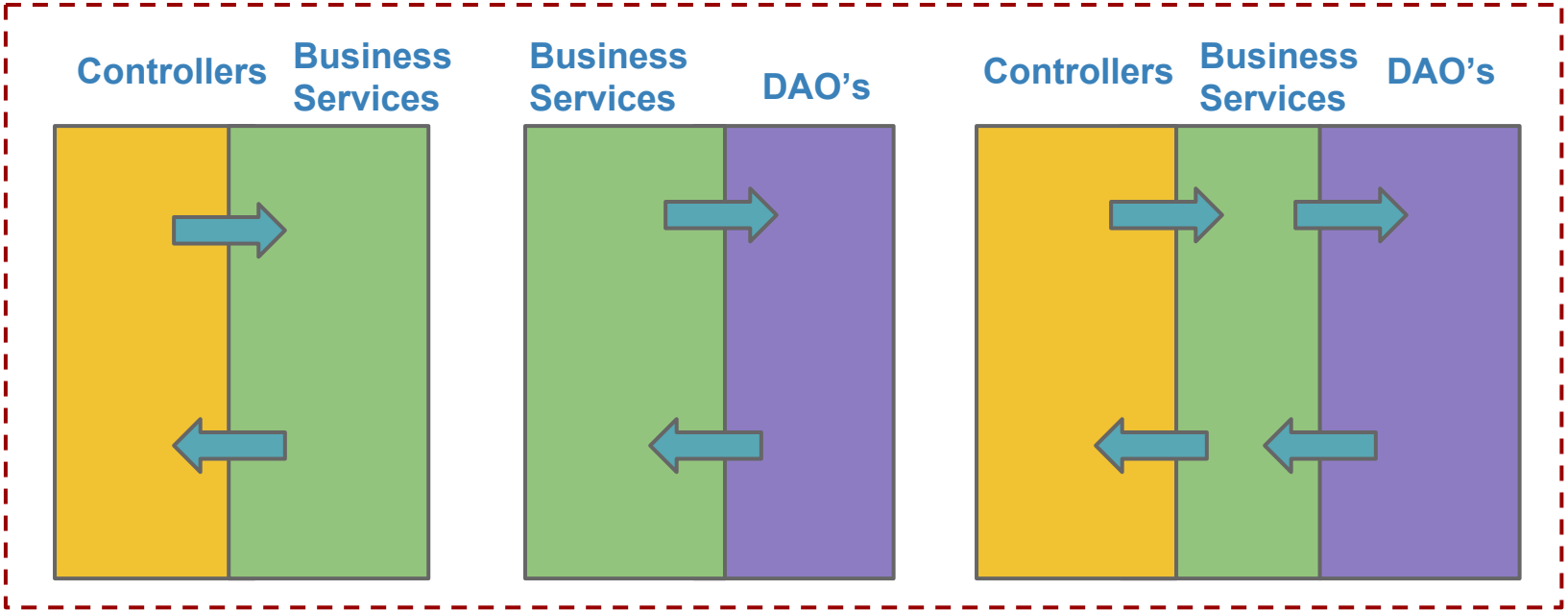
Chapter 9

Integration Tests

Unit Tests



Integration Tests



Note :- Integration test can be between two classes of the same layer also.

Integration Tests

Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.

- **Smoke test**: A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up. It is an analogy with electronics, where the first test occurs when powering up a circuit: if it smokes, it's bad.
- **Acceptance test**: Test that a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case to provide rather than on the components involved.
- **Regression test**: A test that was written when a bug was fixed. It ensure that this specific bug will not occur again. The full name is "non-regression test".

Integration Tests

- Embedded Web Server - Tomcat / Jetty
- Integration Tests Configuration in POM

Demo - SpringIT

Demo - RESTIT

Demo - SOAPIT

End Chapter 9

Chapter 10

Assignment

End Chapter 10