Test Driven Development

Parag Shah

Introduction

- TDD – some history and background
- What exactly is Test Driven Development?
- How does it help us?
- How do we implement it?

Testing – What do we test ?

- Methods – unit testing
- Groups of software modules – integration testing
- Entire system – System testing
- Automated use cases – Functional testing

Unit Testing

We test code because we want to find out if it produces the desired behaviour... and we want to ensure this every single time a change is made in the software

Unit Testing

Unit testing increases quality of code and allows us to march ahead with new code and refactor old code with confidence

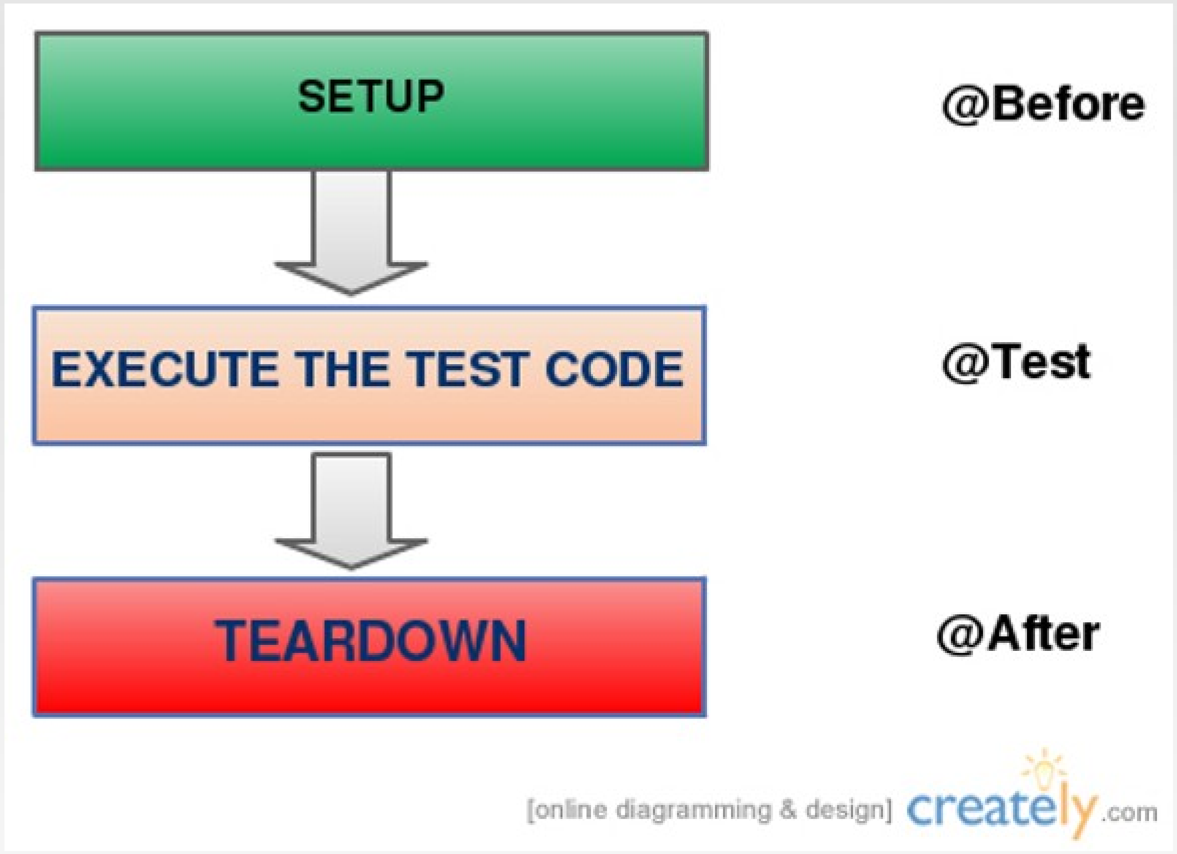
Writing testable code

Code that has hard wired dependencies with other objects and resources is difficult to test

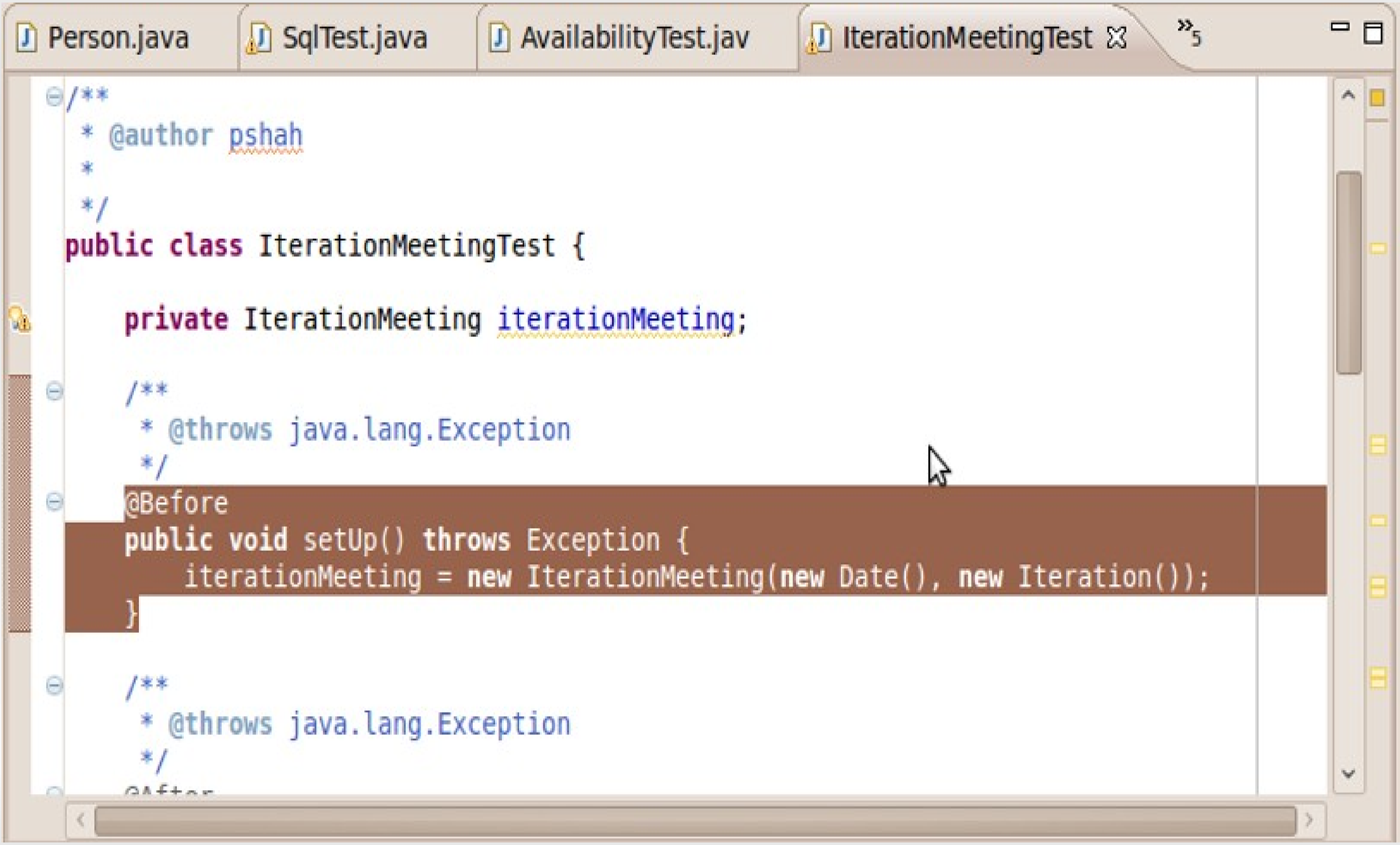
Junit

Junit is an open source software that makes it easy for us to write unit tests by providing us a framework that we can use for creating our unit tests

JUnit

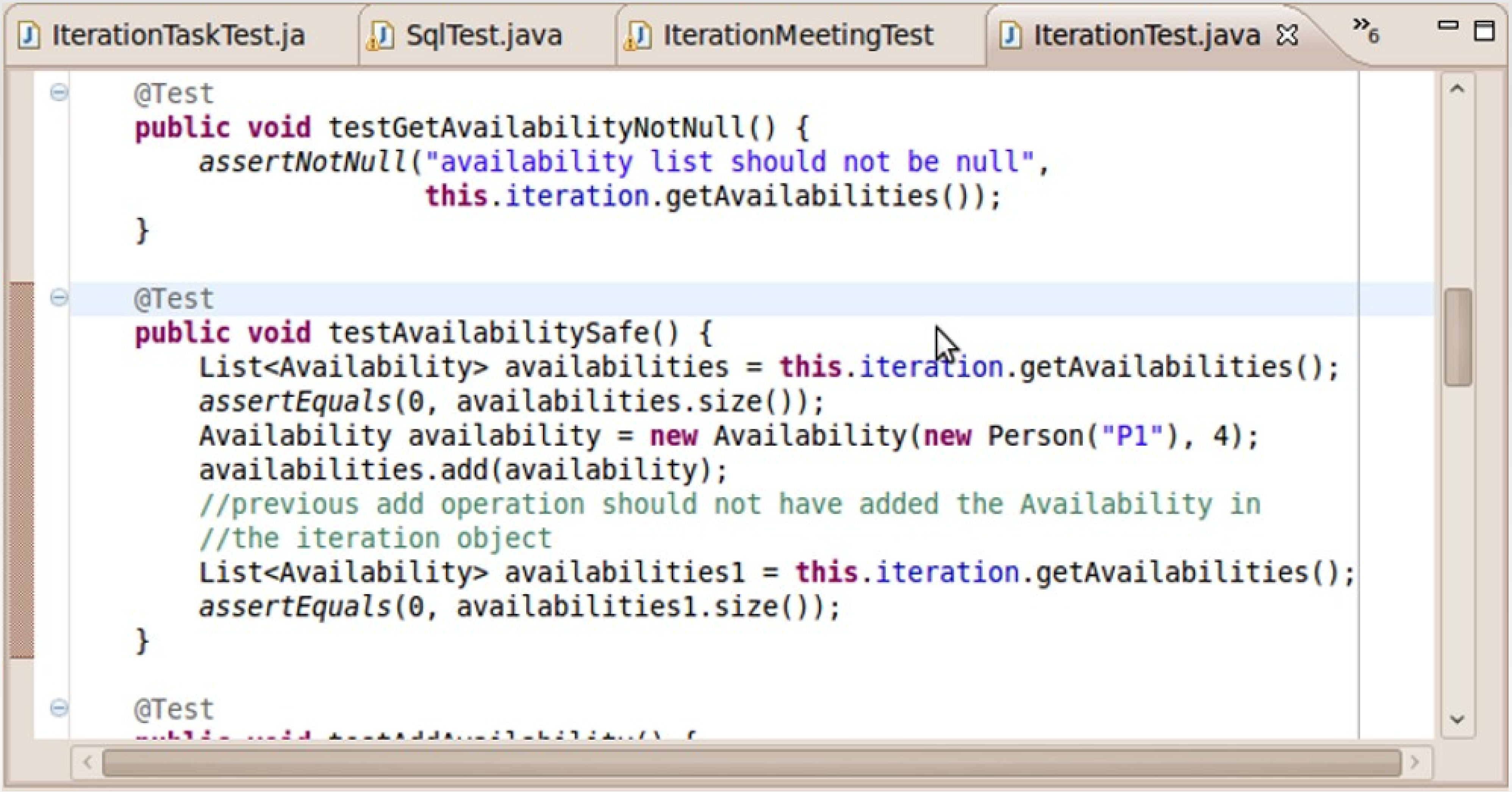

SETUP
@Before
EXECUTE THE TEST CODE
@Test
TEARDOWN
@After
[online diagramming & design] creately.com

Setup with @Before



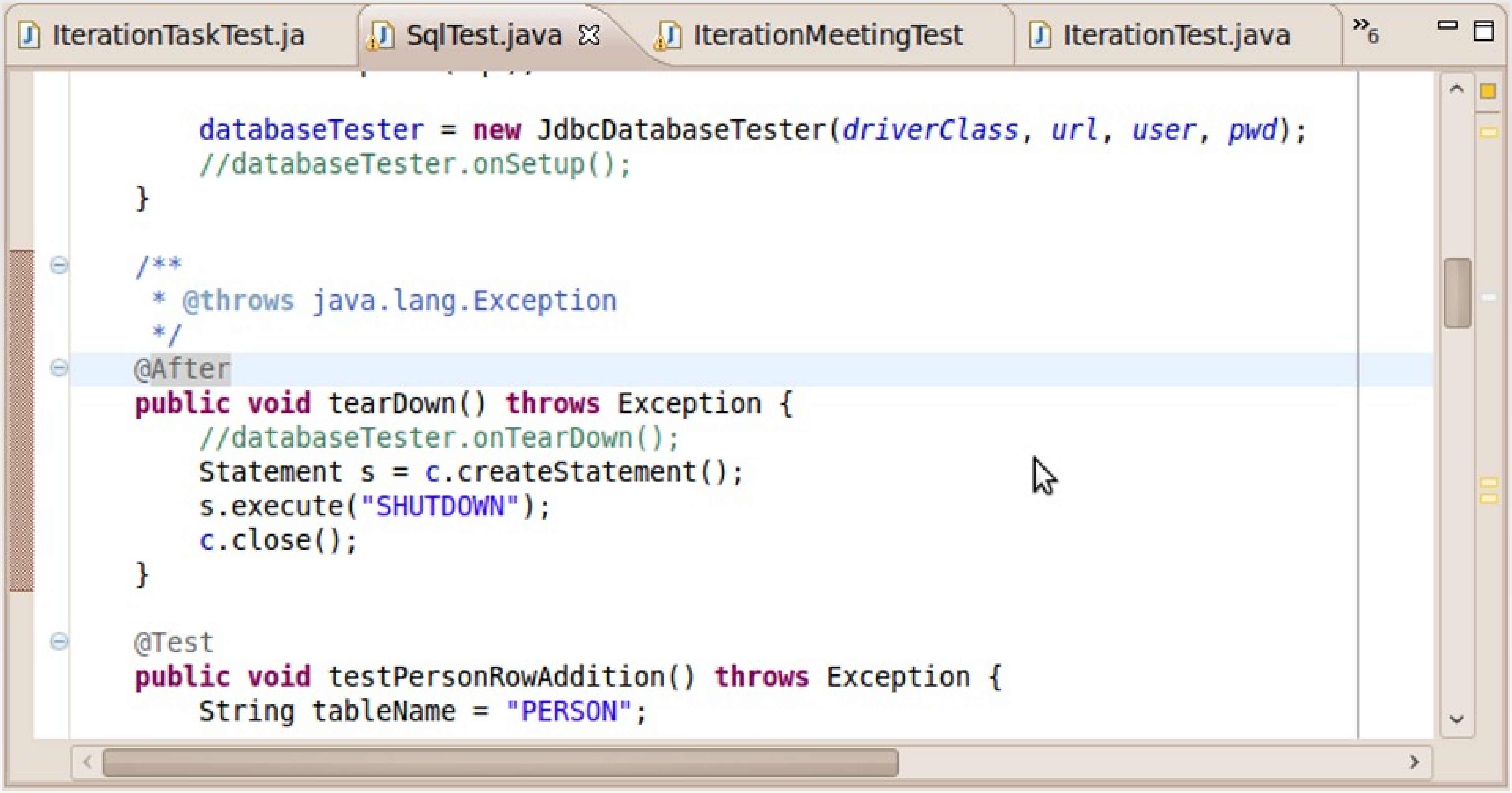
```
/**
 * @author pshah
 *
 */
public class IterationMeetingTest {

    private IterationMeeting iterationMeeting;

    /**
     * @throws java.lang.Exception
     */
    @Before
    public void setUp() throws Exception {
        iterationMeeting = new IterationMeeting(new Date(), new Iteration());
    }

    /**
     * @throws java.lang.Exception
     */
```

Test with @Test



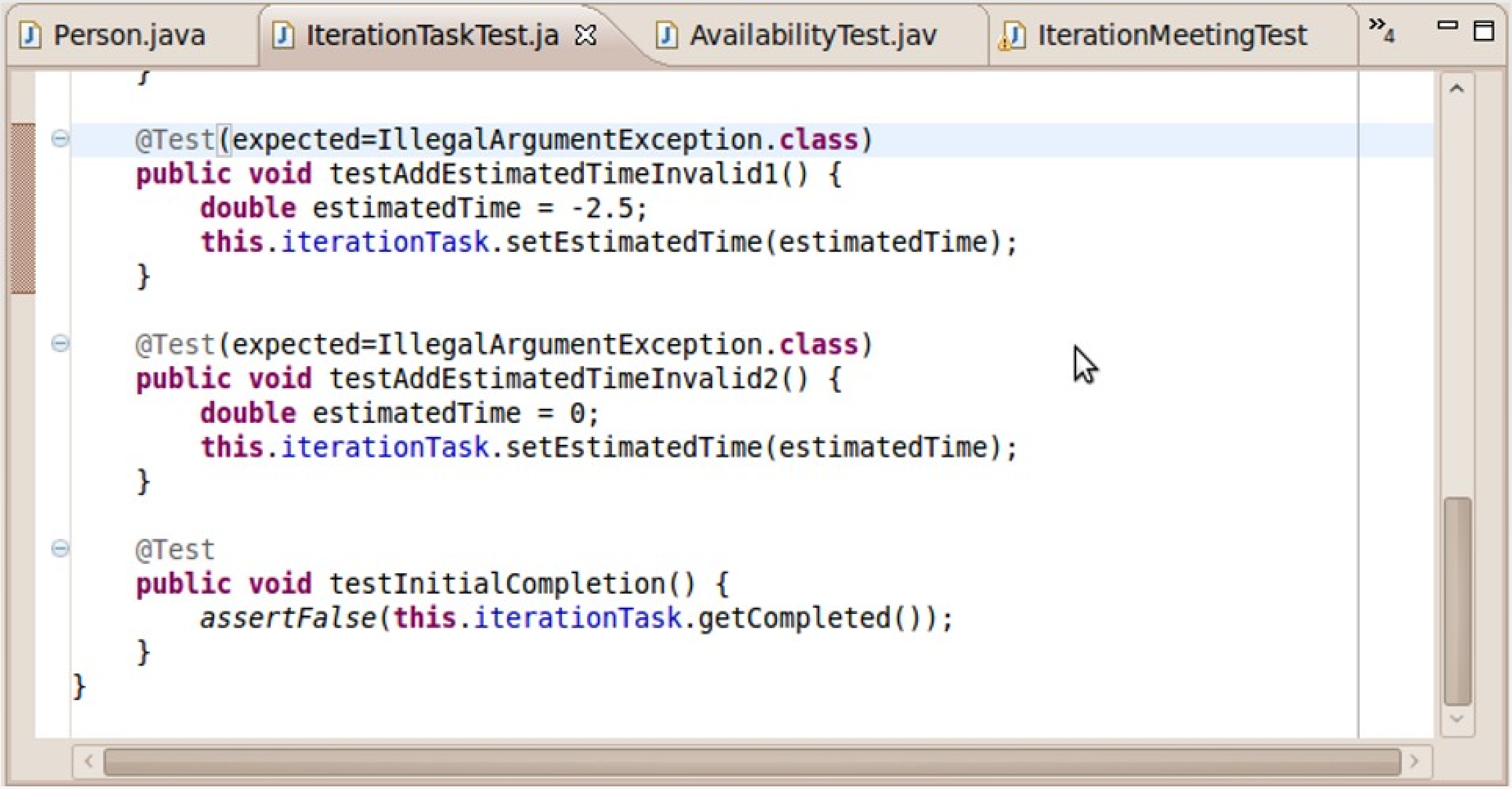
```
    @Test
    public void testGetAvailabilityNotNull() {
        assertNotNull("availability list should not be null",
                    this.iteration.getAvailabilities());
    }

    @Test
    public void testAvailabilitySafe() {
        List<Availability> availabilities = this.iteration.getAvailabilities();
        assertEquals(0, availabilities.size());
        Availability availability = new Availability(new Person("P1"), 4);
        availabilities.add(availability);
        //previous add operation should not have added the Availability in
        //the iteration object
        List<Availability> availabilities1 = this.iteration.getAvailabilities();
        assertEquals(0, availabilities1.size());
    }

    @Test
```

Teardown with @After



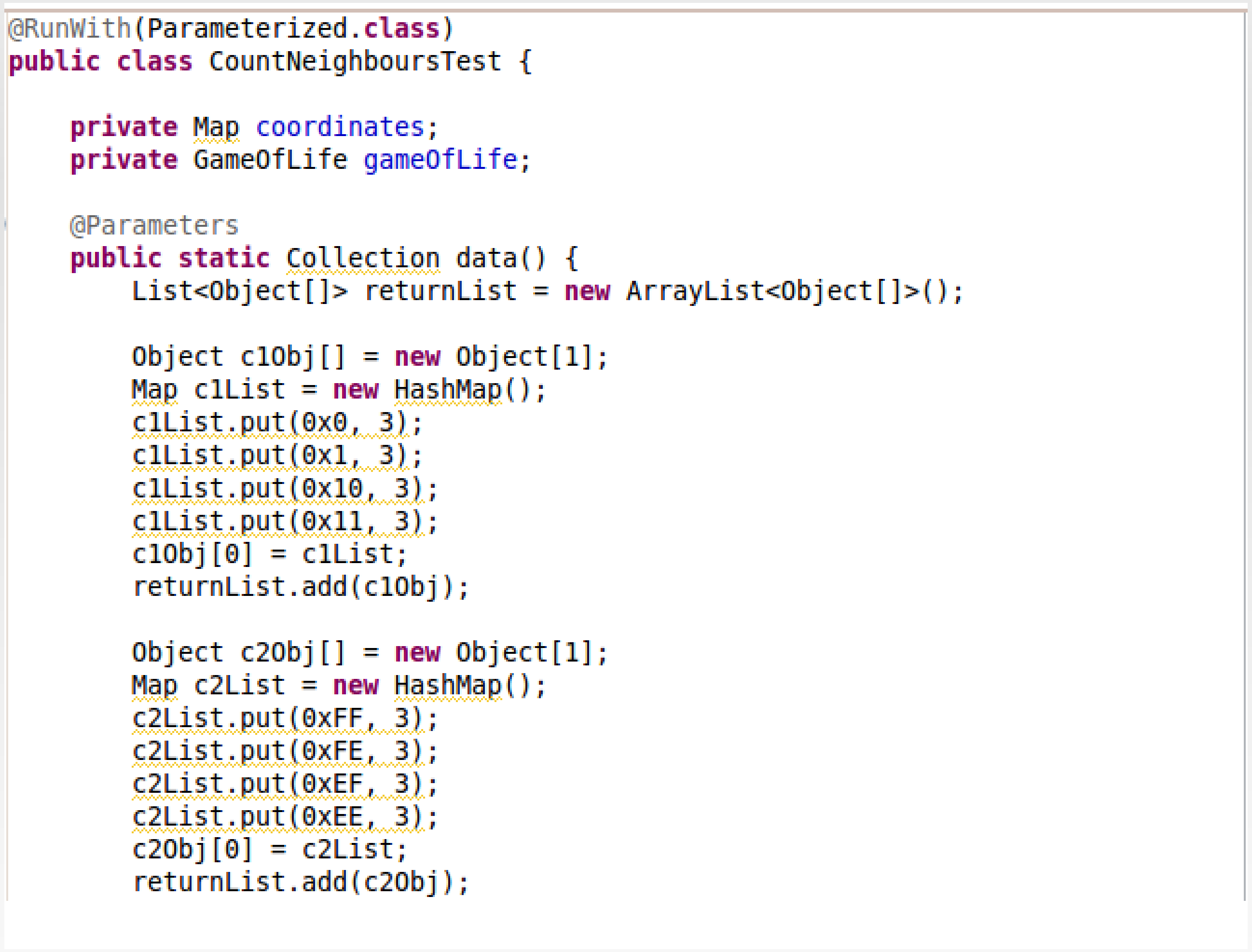
```
        databaseTester = new JdbcDatabaseTester(driverClass, url, user, pwd);
        //databaseTester.onSetup();
    }

    /**
     * @throws java.lang.Exception
     */
    @After
    public void tearDown() throws Exception {
        //databaseTester.onTearDown();
        Statement s = c.createStatement();
        s.execute("SHUTDOWN");
        c.close();
    }

    @Test
    public void testPersonRowAddition() throws Exception {
        String tableName = "PERSON";
```

Expecting Exceptions



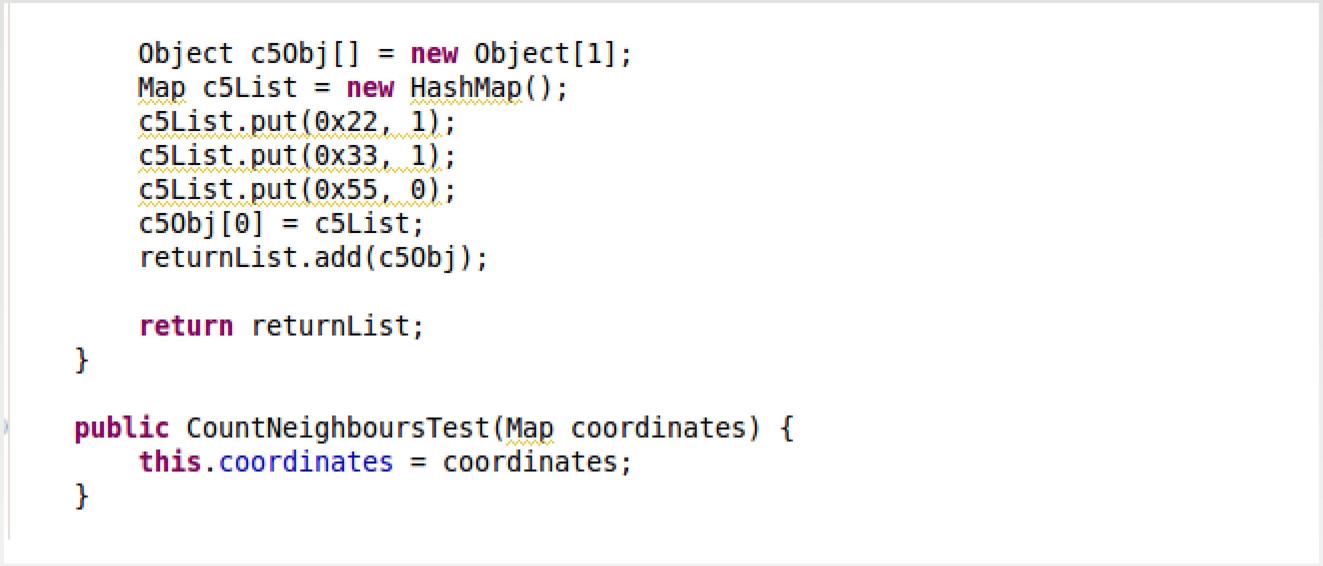
```java
    @Test(expected=IllegalArgumentException.class)
    public void testAddEstimatedTimeInvalid1() {
        double estimatedTime = -2.5;
        this.iterationTask.setEstimatedTime(estimatedTime);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testAddEstimatedTimeInvalid2() {
        double estimatedTime = 0;
        this.iterationTask.setEstimatedTime(estimatedTime);
    }

    @Test
    public void testInitialCompletion() {
        assertFalse(this.iterationTask.getCompleted());
    }
}
```

Parameterized Tests



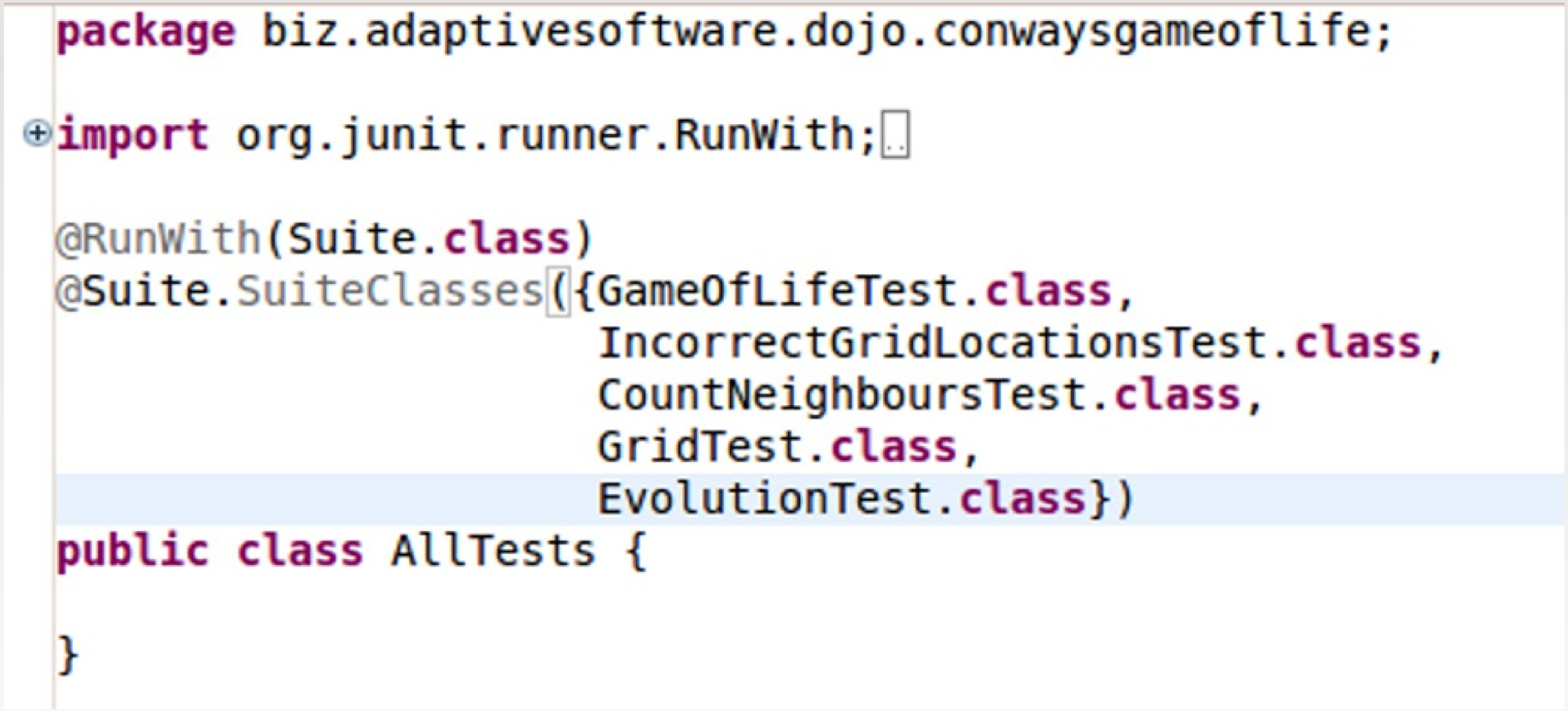
```
@RunWith(Parameterized.class)
public class CountNeighboursTest {

    private Map coordinates;
    private GameOfLife gameOfLife;

    @Parameters
    public static Collection data() {
        List<Object[]> returnList = new ArrayList<Object[]>();

        Object c1Obj[] = new Object[1];
        Map c1List = new HashMap();
        c1List.put(0x0, 3);
        c1List.put(0x1, 3);
        c1List.put(0x10, 3);
        c1List.put(0x11, 3);
        c1Obj[0] = c1List;
        returnList.add(c1Obj);

        Object c2Obj[] = new Object[1];
        Map c2List = new HashMap();
        c2List.put(0xFF, 3);
        c2List.put(0xFE, 3);
        c2List.put(0xEF, 3);
        c2List.put(0xEE, 3);
        c2Obj[0] = c2List;
        returnList.add(c2Obj);
```

Parameterized Tests



```
        Object c5Obj[] = new Object[1];
        Map c5List = new HashMap();
        c5List.put(0x22, 1);
        c5List.put(0x33, 1);
        c5List.put(0x55, 0);
        c5Obj[0] = c5List;
        returnList.add(c5Obj);

        return returnList;
    }

    public CountNeighboursTest(Map coordinates) {
        this.coordinates = coordinates;
    }
```

AllTests



```java
package biz.adaptivesoftware.dojo.conwaysgameoflife;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@Suite.SuiteClasses({GameOfLifeTest.class,
                     IncorrectGridLocationsTest.class,
                     CountNeighboursTest.class,
                     GridTest.class,
                     EvolutionTest.class})
public class AllTests {

}
```

Test First Development

First write a test and let it fail, then write the code to make the test pass

A Simple TDD Example

- Let's write a simple sorting program using TDD principles

Some Testing Tips

Martin Fowler says

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test case instead.

Some Testing Tips

Does it seem like a lot of work?

At first you may have to add a lot of fixtures, but over time you will realize that all the infrastructure to create tests is in place, and new tests will become easier to write

Some Testing Tips

When should we write tests ?

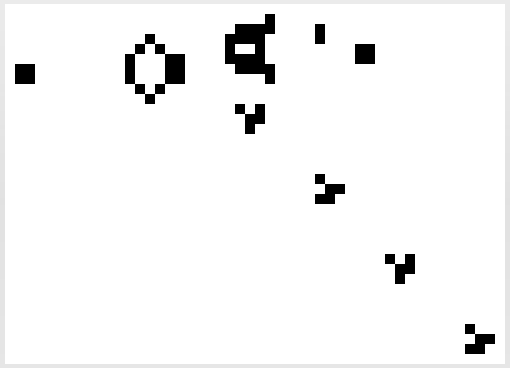
- Before writing the code, write a test to test what the intended code will do
- Whenever we discover a bug, write a test to reproduce it

Some Testing Tips

How often should we run tests ?

- Before writing new code and after writing the code
- Before checking code into your version control system
- Your test suite should always pass 100% - this will give you confidence in your code

Conways Game Of Life



Introduction:

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

Conways Game Of Life

Rules:

- Any live cell with fewer than 2 live neighbors dies as if by under population
- Any live cell with more than 2 live neighbors dies as if by over crowding
- Any live cell with 2 or 3 live neighbors lives on to the next generation
- Any dead cell with 3 live neighbors becomes a live cell as if by reproduction

Conways Game Of Life

Playing:

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

Conways Game Of Life

Introduction:

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

EasyMock

Mocking

- What is a mock and why do we need it?
- Difference between stub and mock
- REMEMBER: Mocking is only possible when the dependency can be injected

EasyMock

Benefits of using EasyMock

- We do not need to manually create mock classes
- Supports refactoring safe mock objects
- Supports return values and exceptions
- Supports checking the order of method calls for one or more mock objects

EasyMock

Simple Usage

- Create the mock
- Record expected behavior
- Replay
- Test code
- Verify expectations with the mock

EasyMock

Advanced Usage

- Expect an explicit number of calls
- Specify return values
- Expect Exceptions
- Test code
- Verify expectations with the mock

EasyMock

Type of Mocks

- Strict mocks
- Nice mocks

DBUnit

Makes it easier to test database code

- Allows us to setup a database with initial data
- Allows us to test the result of inserts and queries with known datasets
- Works seamlessly with Junit 4.0

DBUnit

DBUnit Workflow

- Initialize the database with initial data
- Run code which will change the state of the database
- Verify the new data
- Do any tearDown if required

TestNG

TestNG is a testing framework inspired by Junit and NUnit.

It adds several features which make testing easier.

It also increases the scope of itself from unit testing to functional, integration, as well as end-to-end testing

TestNG

- Features of TestNG
- Annotations
- ThreadPools
- Test for thread safety of code
- Flexible test configuration
- Support for data driven testing @DataProvider
- No need for TestSuite's

Continuous Integration

Martin Fowler says

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage.

Continuous Integration

Practices of continuous integration - Martin Fowler

- Maintain a single source repository
- Automate the build
- Make your build self testing
- Everybody commits to the mainline everyday (often...)
- Every commit should build the mainline on the build machine
- The build should happen fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable

Continuous Integration

Advantages of continuous integration

- Find bugs early
- Prevent cumulative bugs
- Instill confidence in the system

Continuous Integration

Practices of continuous integration - Martin Fowler

- Maintain a single source repository
- Automate the build
- Make your build self testing
- Everybody commits to the mainline everyday (often...)
- Every commit should build the mainline on the build machine
- The build should happen fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable

Practice Session – IPM Manager

Most Agile teams begin their iteration with an 'Iteration Planning Meeting'. This meeting begins with a retrospective of the previous iteration, including the team velocity. After this all developers announce their time availability in this iteration. Then the list of user stories is put upfront. Developers then select a story they want to work on and provide a time estimate. This goes on until all the developers have exhausted their available time. As developers work on tasks, they will mark a story as complete once it has been implemented, along with the actual time spent on it. This time log helps us calculate the team velocity in the next planning meeting, and also helps us determine the user stories which have yet to be done.

Practice Session – IPM Manager

This software will track the following:

- Complete list of user stories
- Stories taken by various developers in an iteration
- Estimated vs. actual time details
- Team velocity in each iteration
- A note of points bought up in retrospectives.

Summary

- Test your code at different levels
  - Unit tests
  - Integration Tests
  - Functional Tests
  - System tests
- First write unit tests, then write production code
- Testing ensures that your code does what it is supposed to do

Summary

- Testing ensures that new code does not break old code
- Testing helps us refactor fearlessly
- Testing forces us to think about the code and design before it is implemented

Summary

- Several frameworks and libraries for testing
  - Junit, TestNG
  - EasyMock
  - DBUnit
  - Several others...

Summary

- Continuous integration adds to the benefits of testing by identifying bugs early
  - Cruise Control & Hudson
- Pairing and Test Driven Development

Summary

- Several frameworks and libraries for testing
  - Junit, TestNG
  - EasyMock
  - DBUnit
  - Several others...