CS342: Software Design

September 21, 2017
Agenda

Service oriented architecture

Introduction to JUnit

● Test Driven Development
● Manual testing vs. automated testing
● JUnit Setup
● TestCase class
● setup and teardown
● TestResult class
● Annotations
● Testing code coverage
Software design where services are provided by application components, through network protocol

- XML/JSON
- Represent business activity
- Self contained, black box,
- May contain other services

Examples of SOA benefits

- Re-useable
- Loosely coupled
- Cloud support by nature
Case study: 3PL system, traditional architecture

Optimzr: Admin facing app
- ASP.NET App
- Optimir.Business.dll: customer, shipment, carrier, rate...
- Optimizr.Data.dll

Trak: Client facing app
- ASP.NET App
- Trak.Business.dll: shipment, carrier, rate...
- Trak.Data.dll

Common.Business.dll: customer, shipment, carrier, rate...
Common.Data.dll: customer, shipment, carrier, rate...

Database: shipments, customers, users, carriers...
Case study: 3PL system, SOA architecture

Flex Admin facing app

- MVC Web App

Flex Client facing app

- Mobile App
- MVC Web App

Security framework

WCF service cloud, Enterprise Service Bus, WIF-based security

- Shipment Svc
- Finance Svc
- Rating Svc
- Carrier Svc
- Notification Svc
- Reporting Svc
- Routing Svc
- ... Svc

Database: shipments, customers, users, carriers...
Test Driven Development

Iterative, short software development cycle: Turn Requirements into very specific test cases

- Write testing cases that will fail -- BEFORE writing any production code
- Write minimum code to pass the testing
- Refactor and optimize
- Repeat

Not all software companies do pure TDD... However, always keep testing in mind

- More robust code, detect “side effect” when touching shared components
- Modularized, easier to read code. Each test case/suite is a “story”
Testing Staff

Traditional Quality Assurance (QA)
- Read SRS
- Create test case and scripts for new feature
- Test “happy path”
- Try to break it
- Log defects, assign to developers
- Repeat
- Regression test

What’s the problem with traditional QA?

Automated QA
- Read SRS
- Write testing suites to cover new feature
- Regression test is already covered

Why is it so difficult?
- Skill set
- Keep up with changes
- Bigger initial investment
- Buy-in from management and engineering
## Manual testing vs. Automated testing

<table>
<thead>
<tr>
<th>Manual Testing</th>
<th>Automated Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time consuming</td>
<td>Right the opposite</td>
</tr>
<tr>
<td>Prone to mistakes - QA need to be highly organized people</td>
<td>Auto serve as knowledge transfer</td>
</tr>
<tr>
<td>High human resource requirement</td>
<td>Longer term investment</td>
</tr>
<tr>
<td>Not reusable</td>
<td>Developers should be engaged in writing testings</td>
</tr>
<tr>
<td>Vulnerary to employee turnover</td>
<td>Shift from pure traditional QA to automated QA</td>
</tr>
<tr>
<td></td>
<td>Product owner: need to be able to read test cases</td>
</tr>
</tbody>
</table>
JUnit overview

Unit testing framework for Java

- Unit testing: smallest testable parts of an application, are individually and independently scrutinized for proper operation
- Provide a set of tools and classes to facilitate unit testing
  - Setup data and variables
  - Trigger tests
  - Check expected and actual behaviors
  - Collect and report results
  - Clear up data
Set up JUnit on your machine

Download and install

- [http://www.junit.org](http://www.junit.org)
- Put it in /Library/JUNIT (mac) or c:\junit (windows)

Environment variable

- Add /Library/JUNIT/junit-4.10.jar in CLASSPATH
- CLASSPATH=$CLASSPATH:/Library/JUNIT/junit-4.10.jar:
- export CLASSPATH

Most of the time, it doesn't work because of CLASSPATH

- printenv
Make sure it works

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJUnit {
    @Test
    public void testAdd() {
        String str = "JUnit is working fine";
        assertEquals("JUnit is working fine",str);
    }
}
```

```java
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(TestJUnit.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

javac TestJUnit.java TestRunner.java
java TestRunner
JUnit offers powerful features to facilitate testing

Fixtures
- Baseline data

Test suites
- Run multiple test cases in one shot

Test runners
- Trigger tests

JUnit classes
- Asserts, TestCase, TestResult
Code example: POJO and business logic classes

```java
public class EmployeeDetails {
    private String name;
    private double monthlySalary;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getMonthlySalary() {
        return monthlySalary;
    }

    public void setMonthlySalary(double monthlySalary) {
        this.monthlySalary = monthlySalary;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```java
public class EmpBusinessLogic {
    // Calculate the yearly salary of employee
    public double calculateYearlySalary(EmployeeDetails employeeDetails) {
        double yearlySalary = 0;
        yearlySalary = employeeDetails.getMonthlySalary() * 12;
        return yearlySalary;
    }

    // Calculate the appraisal amount of employee
    public double calculateAppraisal(EmployeeDetails employeeDetails) {
        double appraisal = 0;

        if(employeeDetails.getMonthlySalary() < 10000){
            appraisal = 500;
        }else{
            appraisal = 1000;
        }

        return appraisal;
    }
}
```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestEmployeeDetails {
    EmpBusinessLogic empBusinessLogic = new EmpBusinessLogic();
    EmployeeDetails employee = new EmployeeDetails();

    @Test
    public void testCalculateAppraisal() {
        employee.setName("Rajeev");
        employee.setAge(25);
        employee.setMonthlySalary(8000);

        double appraisal = empBusinessLogic.calculateAppraisal(employee);
        assertEquals(500, appraisal, 0.0);
    }

    @Test
    public void testCalculateYearlySalary() {
        employee.setName("Rajeev");
        employee.setAge(25);
        employee.setMonthlySalary(8000);

        double salary = empBusinessLogic.calculateYearlySalary(employee);
        assertEquals(96000, salary, 0.0);
    }
}
Testing runner

```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunnerEmployee {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestEmployeeDetails.class);

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

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

javal EmployeeDetails.java EmpBusinessLogic.java TestEmployeeDetails.java TestRunnerEmployee.java
java TestRunnerEmployee
Assert Class

- assertEquals
- assertFalse
- assertNotNull
- assertNull
- assertTrue
- fail - fail the case unconditionally. Useful for marking in progress work
**TestCase Class**

- countTestCases()
- createResult
- getName
- run
- setUp(), tearDown()
- toString

```java
import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;

public class TestJUnit2 extends TestCase {
    protected double fValue1;
    protected double fValue2;

    @Before
    public void setUp() {
        fValue1 = 2.0;
        fValue2 = 3.0;
    }

    @Test
    public void testAdd() {
        System.out.println("Num"+ this.countTestCases());
        String name = this.getName();
        System.out.println("Test Case Name = " + name);
        this.setName("testNewAdd");
        String newName = this.getName();
        System.out.println("Updated Test Case Name = " + newName);
    }

    // tearDown used to close the connection
    public void tearDown() {
    }
}
```
TestResult Class

- addError()
- addFailure()
- failure vs. error
Run multiple suites
- addTest(Test test)
- addTestSuite()
@RunWith and @Suite
Annotuations

@Test Tells JUnit that the public void method can be run as a test case

- 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

@Before, @After

@BeforeClass, @AfterClass

@Ignore: be careful with this one!
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class JunitAnnotation {

    //execute before class
    @BeforeClass
    public static void beforeClass() {
        System.out.println("in before class");
    }

    //execute after class
    @AfterClass
    public static void afterClass() {
        System.out.println("in after class");
    }

    //execute before test
    @Before
    public void before() {
        System.out.println("in before");
    }

    //execute after test
    @After
    public void after() {
        System.out.println("in after");
    }

    //test case
    @Test
    public void test() {
        System.out.println("in test");
    }

    //test case ignore and will not execute
    @Ignore
    public void ignoreTest() {
        System.out.println("in ignore test");
    }
}
Execution order

First of all, the beforeClass() method executes only once.

The afterClass() method executes only once.

The before() method executes for each test case, but before executing the test case.

The after() method executes for each test case, but after the execution of test case.

In between before() and after(), each test case executes.
public class MessageUtil {
    private String message;

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

    public void printMessage() {
        System.out.println(message);
        int a = 0;
        int b = 1/a;
    }

    public String salutationMessage() {
        message = "Hi!" + message;
        System.out.println(message);
        return message;
    }
}

import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class ExceptionTestJUnit {

    String message = "Robert";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test(expected = ArithmeticException.class)
    public void testPrintMessage() {
        System.out.println("Inside testPrintMessage()");
        messageUtil.printMessage();
    }

    @Test
    public void testSalutationMessage() {
        System.out.println("Inside testSalutationMessage()");
        message = "Hi!" + "Robert";
        assertEquals(message, messageUtil.salutationMessage());
    }
}
Speed Test: timeout=...
**Conclusion**

TDD or not, writing tests pays off in long term

But don’t over test. You know you are doing so when you end up fixing tests more often than code