Lecture 2
Software Testing
JUnit
Today’s class

- Software Testing
  - Concepts
  - Granularity
  - Unit Testing
- JUnit
Black-Box and White-Box Testing

- **Black Box** (aka Functional, aka Spec-Based)
  - Tests derived from functional requirements
  - Input/Output Driven
  - Internal source code of software is not relevant to design the tests

- **White Box** (aka Code-Based, aka Structural)
  - Tests derived from source code structure
  - Tests are evaluated in terms of coverage of the source code

Many others in between (Gray Box)
Manual and Automated Testing
Testing: Concepts

- Test case
- Test oracle
- Test fixture
- Test suite
- Test script
- Test driver
- Test adequacy
Testing: Concepts

• Test case (or, simply test)
  • An execution of the software with a given test input, including:
    • Input values
    • Sometimes include execution steps
    • Expected outputs

```java
int actual_output = sum(1, 2);
assert True(actual_output == 3);
```

Example JUnit test case for testing "sum(int a, int b)"
Testing: Concepts

- Test oracle
  - The expected outputs of software for given input
  - A part of test cases
  - Hardest problem in auto-testing: test oracle generation

Example JUnit test case for testing “sum(int a, int b)”

```java
int actual_output = sum(1, 2);
assert True(actual_output == 3);
```
Testing: Concepts

• Test fixture: a fixed state of the software under test used as a baseline for running tests; also known as the test context, e.g.,
  • Loading a database with a specific, known set of data
  • Preparation of input data and set-up/creation of fake or mock objects
Testing: Concepts

• Test suite
  • A collection of test cases
  • Usually these test cases share similar pre-requisites and configuration
  • Usually can be run together in sequence
  • Different test suites for different purposes
    • Certain platforms, Certain feature, performance, …

• Test Script
  • A script to run a sequence of test cases or a test suite automatically
Testing: Concepts

• Test driver
  • A software framework that can load a collection of test cases or a test suite
  • It can also handle the configuration and comparison between expected outputs and actual outputs
Testing: Concepts

• Test adequacy
  • We can’t always use all test inputs, so which do we use and when do we stop?
  • We need a strategy to determine when we have done enough
  • Adequacy criterion: A rule that lets us judge the sufficiency of a set of test data for a piece of software
Testing: Concepts

• Test adequacy example: test coverage
  • A measurement to evaluate the percentage of code tested
  • Statement coverage
  • Branch coverage, ...
Granularity of Testing

- **Unit Testing**
  - Test of each single module
- **Integration Testing**
  - Test the interaction between modules
- **System Testing**
  - Test the system as a whole, by developers
- **Acceptance Testing**
  - Validate the system against user requirements, by customers with no formal test cases
Unit testing

- Testing of an basic module of the software
  - A function, a class, a component
- Typical problems revealed
  - Local data structures
  - Algorithms
  - Boundary conditions
  - Error handling
Why Unit Testing?

- Divide-and-conquer approach
  - Split system into units
  - Debug unit individually
  - Narrow down places where bugs can be
  - Don’t want to chase down bugs in other units
How to Do Unit Testing

• Build systems in layers
  • Starts with classes that don’t depend on others.
  • Continue testing building on already tested classes.

• Benefits
  • Avoid having to write mock classes
  • When testing a module, ones it depends on are reliable.
How to Do Unit Testing

• Build systems in layers
  • Starts with classes that don’t depend on others.
  • Continue testing building on already tested classes.

• Benefits
  • Avoid having to write mock classes
  • When testing a module, ones it depends on are reliable.

Diagram:

- u1
- u2
- u3
- u4

Relationships:
- u1 → u3
- u2 → u3
- u3 → u4
- ?
How to Do Unit Testing

• Build systems in layers
  • Starts with classes that don’t depend on others.
  • Continue testing building on already tested classes.

• Benefits
  • Avoid having to write mock classes
  • When testing a module, ones it depends on are reliable.
How to Do Unit Testing

• Build systems in layers
  • Starts with classes that don’t depend on others.
  • Continue testing building on already tested classes.

• Benefits
  • Avoid having to write mock classes
  • When testing a module, ones it depends on are reliable.

mock classes
Unit test framework

• xUnit
  • Created by Kent Beck in 1989
    • This is the same guy who invented XP and TDD
    • The first one was sUnit (for smalltalk)
  • JUnit
    • The most popular xUnit framework
    • There are about 70 xUnit frameworks for corresponding languages

Never in the annals of software engineering was so much owed by so many to so few lines of code

--Martin Fowler
Today’s class

• Software Testing
  • Concepts
  • Granularity
  • Unit Testing
• JUnit
Program to Test

class IMath {

    /**
     * Returns an integer to the square root of x (discarding the fractional parts)
     */

    public int isqrt(int x) {
        int guess = 1;
        while (guess * guess < x) {
            guess++;
        }
        return guess;
    }
}

Conventional Testing

/** A class to test the class IMath. */
public class IMathTestNoJUnit {
    /** Runs the tests. */
    public static void main(String[] args) {
        printTestResult(0);
        printTestResult(1);
        printTestResult(2);
        printTestResult(3);
        printTestResult(100);
    }
    private static void printTestResult(int arg) {
        IMath tester=new IMath();
        System.out.print("isqrt(" + arg + ") ==> ");
        System.out.println(tester.isqrt(arg));
    }
}
Conventional Test Output

• What does this say about the code? Is it right?
• What’s the problem with this kind of test output?

<table>
<thead>
<tr>
<th>Isqrt(0)</th>
<th>==&gt; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isqrt(1)</td>
<td>==&gt; 1</td>
</tr>
<tr>
<td>Isqrt(2)</td>
<td>==&gt; 2</td>
</tr>
<tr>
<td>Isqrt(3)</td>
<td>==&gt; 2</td>
</tr>
<tr>
<td>Isqrt(100)</td>
<td>==&gt; 10</td>
</tr>
</tbody>
</table>
Solution?

• Automatic verification by testing program
  • Can write such a test program by yourself, or
  • Use testing tool supports, such as JUnit
• JUnit
  • A simple, flexible, easy-to-use, open-source, and practical unit testing framework for Java.
  • Can deal with a large and extensive set of test cases.
• Refer to www.junit.org.
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathJUnit1 {

    /** A JUnit test method to test isqrt. */
    @Test
    public void testIsqrt() {
        IMath tester = new IMath();
        assertTrue(0 == tester.isqrt(0));
        assertTrue(1 == tester.isqrt(1));
        assertTrue(1 == tester.isqrt(2));
        assertTrue(10 == tester.isqrt(100));
    }

    /** Other JUnit test methods */
}

/** A JUnit test class to test isqrt. */

/** A JUnit test method to test isqrt. */

/** Other JUnit test methods */
JUnit Execution (1)

- Right click the JUnit class, and select “Run As” => “JUnit Test”

Not so good, why? 😞
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathJUnit2 {

    /** A JUnit test method to test isqrt. */
    @Test
    public void testIsqrt() {
        IMath tester = new IMath();
        assertEquals(0, tester.isqrt(0));
        assertEquals(1, tester.isqrt(1));
        assertEquals(1, tester.isqrt(2));
        assertEquals(1, tester.isqrt(3));
        assertEquals(10, tester.isqrt(100));
    }

    /** Other JUnit test methods */
}

assertTrue(0 == tester.isqrt(0));
assertTrue(1 == tester.isqrt(1));
assertTrue(1 == tester.isqrt(2));
assertTrue(1 == tester.isqrt(3));
assertTrue(10 == tester.isqrt(100));
JUnit Execution (2)

- Why now better error info?
  - `assertTrue(0==tester.isqrt(0))`
  - `assertEquals(0, tester.isqrt(0))`

Can we make it better?

detailed result is abstracted into boolean before passed to JUnit

the detailed result is passed to JUnit
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathJUnit3 {

/** A JUnit test method to test isqrt. */
@Test
public void testIsqrt() {
    IMath tester = new IMath();
    assertEquals("square root for 0 ", 0, tester.isqrt(0));
    assertEquals("square root for 1 ", 1, tester.isqrt(1));
    assertEquals("square root for 2 ", 1, tester.isqrt(2));
    assertEquals("square root for 3 ", 1, tester.isqrt(3));
    assertEquals("square root for 100 ", 10, tester.isqrt(100));
}

/** Other JUnit test methods*/
}
JUnit Execution (3)

Still have problems, why?

We only see the error info for the first input... 😞
public class IMathTestJUnit4 {
    private IMath tester;

    @Before /**< Setup method executed before each test */
    public void setup(){
        tester=new IMath();
    }

    @Test /**< JUnit test methods to test isqrt. */
    public void testIsqrt1() {
        assertEquals("square root for 0 ", 0, tester.isqrt(0));
    }

    @Test
    public void testIsqrt2() {
        assertEquals("square root for 1 ", 1, tester.isqrt(1));
    }

    @Test
    public void testIsqrt3() {
        assertEquals("square root for 2 ", 1, tester.isqrt(2));
    }
    ...
}
JUnit Execution (4)

Still may have trouble, why?

We need to write so many similar test methods...
Parameterized Tests: Illustration

input x

\text{test m1()}

input y

\text{test m1()}

input z

\text{test m1()}

\text{test m1()}

input x  \quad \text{input y}  \quad \text{input z}

\text{test m1()}

\text{test m1()}

\text{test m1()}

\text{test m1()}

\text{test m1()}
@RunWith(Parameterized.class)
public class IMathJUnitParameterized {
    private IMath tester;
    private int input;
    private int expectedOutput;

    /** Constructor method to accept each input-output pair*/
    public IMathJUnitParameterized(int input, int expectedOutput) {
        this.input = input;
        this.expectedOutput = expectedOutput;
    }

    @Before
    /** Set up method to create the test fixture */
    public void initialize() {
        tester = new IMath();
    }

    @Parameterized.Parameters
    /** Store input-output pairs, i.e., the test data */
    public static Collection<Object[]> valuePairs() {
        return Arrays.asList(new Object[][]{{0, 0}, {1, 1}, {2, 1}, {3, 1}, {100, 10}});
    }

    @Test
    /** Parameterized JUnit test method*/
    public void testIsqrt() {
        assertEquals("square root for " + input + " ", expectedOutput, tester.isqrt(input));
    }
}
JUnit Execution: Parameterized Tests

Note that not all tests can be abstract into parameterized tests
A Counter Example

public class ListJUnit {
    List list;
    @Before /** Set up method to create the test fixture */
    public void initialize() {
        list = new ArrayList();
    }
    /** JUnit test methods*/
    @Test
    public void test1() {
        list.add(1);
        list.remove(0);
        assertEquals(0, list.size());
    }
    @Test
    public void test2() {
        list.add(1);
        list.add(2);
        list.add(3);
        assertEquals(3, list.size());
    }
    ...}
These tests cannot be abstract into parameterized tests, because the tests contains different method invocations
JUnit Test Suite

- Test Suite: a set of tests (or other test suites)
  - Organize tests into a larger test set.
  - Help with automation of testing
- Consider the following case, how can I organize all the tests to make testing easier?
  - I need to test the List data structure
  - I also need to test the Set data structure

```java
@RunWith(Suite.class)
@SuiteClasses({ ListTestJUnit.class, SetTestJUnit.class })
public class MyJUnitSuite {
}

@RunWith(Suite.class)
@SuiteClasses({ MyJUnit.class, ... })
public class MyMainJUnitSuite {
}
```
## JUnit: Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Test</td>
<td>Identify test methods</td>
</tr>
<tr>
<td>@Test (timeout=100)</td>
<td>Fail if the test takes more than 100ms</td>
</tr>
<tr>
<td>@Before</td>
<td>Execute before each test method</td>
</tr>
<tr>
<td>@After</td>
<td>Execute after each test method</td>
</tr>
<tr>
<td>@BeforeClass</td>
<td>Execute before each test class</td>
</tr>
<tr>
<td>@AfterClass</td>
<td>Execute after each test class</td>
</tr>
<tr>
<td>@Ignore</td>
<td>Ignore the test method</td>
</tr>
</tbody>
</table>
JUnit: Assertions

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail([msg])</td>
<td>Let the test method fail, optional msg</td>
</tr>
<tr>
<td>assertTrue([msg], bool)</td>
<td>Check that the boolean condition is true</td>
</tr>
<tr>
<td>assertFalse([msg], bool)</td>
<td>Check that the boolean condition is false</td>
</tr>
<tr>
<td>assertEquals([msg], expected, actual)</td>
<td>Check that the two values are equal</td>
</tr>
<tr>
<td>assertNull([msg], obj)</td>
<td>Check that the object is null</td>
</tr>
<tr>
<td>assertNotNull([msg], obj)</td>
<td>Check that the object is not null</td>
</tr>
<tr>
<td>assertSame([msg], expected, actual)</td>
<td>Check that both variables refer to the same object</td>
</tr>
<tr>
<td>assertNotSame([msg], expected, actual)</td>
<td>Check that variables refer to different objects</td>
</tr>
</tbody>
</table>
More on JUnit?

• Homepage:
  • www.junit.org

• Tutorials
  • http://www.vogella.com/tutorials/JUnit/article.html
  • http://www.tutorialspoint.com/junit/
  • https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml
Today’s class

• Software Testing
  • Concepts
  • Granularity
  • Unit Testing
• JUnit
Next class

• Test Coverage
Thanks!