Unit Testing and JUnit
Testing Objectives

- Tests intended to find errors
- Errors should be found quickly
- Good test cases have high $p$ for finding a yet undiscovered error
- Successful tests cause program failure, i.e. find an undiscovered error.
- Tests should be mutually exclusive and exhaustive
- Minimal set of test cases needs to be developed because exhaustive testing not possible
Unit Testing

• A unit is typically a method/class/a cluster
• Testing done to each unit, in isolation, to verify its behavior
• Typically the unit test will establish some sort of artificial environment and then invoke methods in the unit being tested
• It then checks the results returned against some known value
• When the units are assembled we can use the same tests to test the system as a whole
JUnit Testing Tool for Java: JUnit

JUnit is a framework for writing tests

- JUnit was written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
- JUnit uses Java’s reflection capabilities (Java programs can examine their own code)
- JUnit features includes:
  - Test fixtures for sharing common test data
  - Assertions for testing expected results
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners

- JUnit is not yet included in Sun’s SDK, but an increasing number of IDEs include it
- BlueJ, JBuilder, Eclipse, DrJava etc. now provide JUnit support

www.junit.org
Structure of a JUnit test class

• Suppose you want to test a class named Complex
• public class ComplexTest
  
  extends junit.framework.TestCase {
  – This is the unit test for the Complex class; it declares (and possibly defines) values used by one or more tests
• public ComplexTest() {}  
  – This is the default constructor
• protected void setUp()
  – Creates a test fixture by creating and initializing objects and values
• protected void tearDown()
  – Releases any system resources used by the test fixture
• public void testAdd(), public void testEquals(), etc.
  – These methods contain tests for the Complex class methods add(), Equals(), etc.
```java
public class Complex {
    int real_part; int imaginary_part;

    public Complex(int r, int i) {
        real_part=r;
        imaginary_part=i;
    }

    public Complex() {
        real_part=0;
        imaginary_part=0;
    }

    public boolean Equal(Complex c) {
        boolean result = false;
        if ((real_part==c.get_r()) && (imaginary_part==c.get_i())) result=true;
        return result;
    }

    public Complex Add(Complex c) {
        Complex result = new Complex(c.get_r()+real_part, c.get_i()+imaginary_part);
        return result;
    }

    public int get_r() { return real_part; }
    public int get_i() { return imaginary_part; }
}
```
public class ComplexTest extends TestCase {
    Complex c1;
    Complex c2;
    protected void setUp() {
        c1 = new Complex(7,3);
        c2 = new Complex(12,6);
    }
    protected void tearDown(){  }
}
Using JUnit (Add Test Cases)

```java
public void testAdd() {
    Complex result = c1.Add(new Complex(5,3));
    assertEquals(result.get_r(),c2.get_r());
    assertEquals(result.get_i(),c2.get_i());
}

public void testEqual(){
    assertTrue(!c2.Equal(c1));
    assertTrue(c1.Equal(new Complex(7,3)));
}
```

Note that each test begins with a *brand new* c1 and c2

This means you don’t have to worry about the order in which the tests are run
Two alternative to run Your tests

- Implicit
  - TestRunner uses reflection to determine the tests to be run i.e. methods start with "test" prefix
  
  ```java
  public static void main(String [] args) {
      junit.textui.TestRunner.run(ComplexTest.class);
  }
  ```

- Explicit
  - use of a `public static` method of TestCase class named `suite()`
  - explicitly naming the tests to be run
  
  ```java
  public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTest(new ComplexTest("testAdd"));
      suite.addTest(new ComplexTest("testEqual"));
      return suite;
  }
  public static void main(String[] argv){
      junit.textui.TestRunner.run(suite());
  }
  ```
Writing a Test Case

• To write a test case, follow these steps:
  1. Define a subclass of TestCase.
  2. Override the setUp() method to initialize object(s) under test.
  3. Override the tearDown() method to release object(s) under test.
  4. Define one or more testXXX() methods that exercise the object(s) under test.
  5. Define a suite() factory method that creates a TestSuite containing all the testXXX() methods of the TestCase.
  6. Define a main() method that runs the TestCase.
More Assert methods I

- **assertEquals**(`expected`, `actual`)
- **assertEquals**(String `message`, `expected`, `actual`)
  - This method is heavily overloaded: `arg1` and `arg2` must be both objects or both of the same primitive type
  - For objects, uses your equals method, *if* you have defined it properly, as `public boolean equals(Object o)`
  - --otherwise it uses `==`

- **assertSame**(Object `expected`, Object `actual`)
- **assertSame**(String `message`, Object `expected`, Object `actual`)
  - Asserts that two objects refer to the same object (using `==`)

- **assertNotSame**(Object `expected`, Object `actual`)
- **assertNotSame**(String `message`, Object `expected`, Object `actual`)
  - Asserts that two objects do not refer to the same object
More Assert methods II

• `assertNull(Object object)`
  `assertNull(String message, Object object)`
  – Asserts that the object is null

• `assertNotNull(Object object)`
  `assertNotNull(String message, Object object)`
  – Asserts that the object is null

• `fail()`
  `fail(String message)`
  – Causes the test to fail and throw an `AssertionFailedError`
  – Useful for testing Exceptions
Using JUnit

• Download latest version (3.8.1) from: www.junit.org
• Setup classpath to junit.jar
• Write TestCase classes and organize them (Import junit.framework.*)
• Include appropriate JUnit TestRunner in main()
JUnit GUI Test Runners

**Figure 1**: A Successful Run (awtui)  
**Figure 2**: An Unsuccessful Run (swingui)
public class Complex {
    int real_part; int imaginary_part;
    public Complex(int r, int i) {
        real_part=r; imaginary_part=i;
    }
    public Complex() {
        real_part=0; imaginary_part=0;
    }
    public boolean Equal(Complex c) {
        boolean result = false;
        if ((real_part==c.get_r()) &&
            (imaginary_part==c.get_i())) result=true;
        return result;
    }
    public Complex Add(Complex c) {
        Complex result = new Complex(c.get_r()+real_part,c.
                                      get_i()+imaginary_part);
        return result;
    }
    public int get_r() { return real_part; }
    public int get_i() { return imaginary_part; }
}

public static void main(String[] argv) {
    Complex c1 = new Complex(7,3);
    Complex c2 = new Complex(12,6);
    Complex result = c1.Add(new Complex(5,3));
    if (result.get_r()==c2.get_r() &&
        result.get_i()==c2.get_i()) System.out.println("Addition test Passed");
    else System.out.println("Addition test Failed");
    if (!c2.Equal(c1)) System.out.println("Equality test I Passed");
    else System.out.println("Equality test I Failed");
}
Organizing and Running Tests

- To run your tests, you need:
- An instance of a TestRunner class.
- An instance of your test class (named `MyTestClass` for the purposes of this example) containing the tests you want to run. `MyTestClass` must extend `junit.framework.TestCase`.
- Some way of telling the chosen TestRunner instance which tests on your `MyTestClass` instance to run.
Organizing Tests Hierarchies:
An Example:

```java
import junit.framework.TestCase;
public class MyTestClass extends TestCase {
    public MyTestClass(String name) {
        super(name);
    }
    public void runTest() {
        testTurnLeft();
    }
    public void testTurnLeft() { ... code here ... }
}
```

```java
import junit.framework.TestCase;
public class CalculatorTest extends TestCase {
    public CalculatorTest(String name) {
        super(name);
    }
    public void runTest() {
        testIsDivisor();
    }
    public void testIsDivisor() { ... code here ... }
}
```

```java
import junit.framework.TestCase;
public class MyThirdTestClass extends TestCase {
    public MyThirdTestClass(String name) {
        super(name);
    }
    public void test1() { ... code here ... }
    public void test2() { ... code here ... }
}
```
public static suite() {
    TestSuite testSuite = new TestSuite();
    testSuite.addTest(new MyTestClass("testTurnLeft");
    testSuite.addTest(new CalculatorTest("testIsDivisor");
    testSuite.addTest(new TestSuite(MyThirdTest.class));
    return testSuite;
}
Problems with unit testing

JUnit is designed to call methods and compare the results they return against expected results

- This works great for methods that \textit{just} return results, but some methods have side effects
  - To test methods that do output, you have to capture the output
    - It’s possible to capture output, but it’s an unpleasant coding chore
  - To test methods that change the state of the object, you have to have code that checks the state
    - It’s a good idea in any case to write self-tests for object validity
- It isn’t easy to see how to unit test GUI code
Conclusions

• Unit Testing is the place to identify and remove errors at the earliest
• Effective Automated tests can be written to test most of the units, if not all
• Unit Testing Frameworks (UTF), like JUnit are quite useful in writing, executing and organizing unit tests for incremental code development and change management.
JUnit in Eclipse

• To create a test class, select File → New → Other... → Java, JUnit, TestCase and enter the name of the class you will test.

Fill this in

This will be filled in automatically
Running JUnit

First, select a Test class

Second, use this pulldown menu

Third, Run As → JUnit Test
Results

Your results are here

JUnit (TokenizerTest)