Test-Driven Development
(a.k.a. Design to Test)

CSE219, Computer Science III
Stony Brook University
http://www.cs.stonybrook.edu/~cse219
Man-hours

- Labor is sometimes measured in man-hours, man-months, or man-years.
- Example: Doom3 took 5 years and more than 100 man-years of labor to develop
  - Company Spokesman: "It will be ready when it's done"
- Why not double the size of the team and halve the lead time (concept date to release date)?
Man-hours: The Mythical Man-Month

- Assume that a software program might take one expert programmer a year to develop = 12 man-months
- Market pressures might be such that we want to get the program finished in a month, rather than a year
- 1 programmer * 12 months = 12 programmers * 1 month?
  - When you throw additional programmers at a project that is late, you are likely to make it more late!
  - Remove promised-but-not-yet-completed features, rather than multiplying workers bees.
  - Also, at least one team member must have detailed knowledge of the entire system (all the modules).
Design to Implementation

• Assume a modular design has been completed
  • Can all the modules be developed in parallel?
    • most likely not - due to dependencies
  • division of work within a module may also be necessary
    • can classes within a module be developed in parallel?
      • most likely not - due to dependencies
    • division of work within a class may also be necessary
      • can methods within a class be developed in parallel?
        • Again most likely not - due to dependencies
Bottom-Up Development

- Traditional approach:
  - All modules used by module M are implemented and tested before M is implemented.
  - Requires the use of drivers (i.e., testers).
- Example of Module dependencies:

Bottom-up development can place less of a load on system resources.
Bottom-up development can lead to earlier completion of useful subsystems.
Top-Down Development

- All modules that use module M are implemented and tested before M is implemented.
  - Modules themselves will probably use bottom-up development
- Requires the use of stubs.
- Testing procedures are important
- Example of module dependencies:
  If the design contains a type hierarchy, top-down development is required.
The Development Strategy

- Should be defined explicitly before implementation begins
- Should be primarily top-down, with bottom-up used mainly for modules that are easier to implement than to simulate
- Advantages of top-down outweigh bottom-up
  - simplifies system integration & test
  - makes it possible to produce useful partial versions of the system
  - allows critical high-level design errors to be caught early
- Bottom-up development may be used for each module
  - we’ll see this with module testing as well
What is design to test?

- Approach to implementation
  - design modular classes and methods
  - before coding:
    - determine what needs to be tested
    - design test cases for those important methods
- test incrementally, as you implement your solution
Design to Test = Design to Fail

- Things to avoid:
  - coding without a design
  - not planning on how a design will be tested
  - creating large amounts of untested code
  - coding very large methods
  - lack of modularity can doom an implementation
Testing vs. Debugging

Coding

Testing

Does the code work properly

YES

NO

Debugging
Important Definitions

- **Testing**
  - a process of running a program on a set of test cases and comparing the actual results with expected results

- **Verification**
  - a formal or informal argument that a program works as intended for all possible inputs

- **Validation**
  - a process designed to increase confidence that a program works as intended
    - performed through verification or testing

- **Defensive Programming**
  - writing programs in a way designed to ease the process of validation and debugging
Kinds of Testing

- **Unit Testing**
  - Test each module in a program separately.

- **Integration Testing**
  - Test interfaces between modules.
  - Much more difficult than unit testing

- **Regression Testing**
  - Test programs after modifications to ensure correct behavior of the original program is preserved.

- **System Testing**
  - Test overall system behavior.
Aspects of Testing

• How do we generate test cases?
  • Exhaustive
    • Consider all possible combinations of inputs.
    • Often infeasible — why?
    • Is it feasible with your project?
  • Sampled
    • A small but representative subset of all input combinations.
      • Black-box testing - Test cases generated from program specifications and not dependent on the implementation
      • Glass-box testing - Test cases generated from program’s code
Black-box testing

- It is the best place to start when attempting to test a program thoroughly

- Test cases based on program’s specification, not on its implementation (see the homework grading sheets)

- Test cases are not affected by:
  - Invalid assumptions made by the programmer
  - Implementation changes
    - Use same test cases even after program structures has changed

- Test cases can be generated by an “independent” agent, unfamiliar with the implementation.

- Test cases should cover all paths (not all cases) through the specification, including exceptions.
Boundary Conditions

- A boundary condition is an input that is “one away” from producing a different behavior in the program code.

- Such checks catch 2 common types of errors:
  - Logical errors, in which a path to handle a special case presented by a boundary condition is omitted.
  - Failure to check for conditionals that may cause the underlying language or hardware system to raise an exception (ex: arithmetic overflow).
Glass-box testing

- Black-box testing is generally not enough.
- For Glass-box testing, the code of a program being tested is taken into account.
- Path-completeness:
  - Test cases are generated to exercise each path through a program.
  - May be insufficient to catch all errors.
  - Can be used effectively only for a program fragment that contains a reasonable number of paths to test.
Testing paths through specification

• Examine the method specifications (preconditions) & all paths through method to generate unique test cases for testing.

/* REQUIRES: x >= 0 && y >= 10 */

public static int calc(int x, int y) { ... }

• Translate paths to test cases:

  x =  0, y = 10 (x == 0 && y == 10)
  x =  5, y = 10 (x > 0 && y == 10)
  x =  0, y = 15 (x == 0 && y > 10)
  x =  5, y = 15 (x > 0 && y > 10)
  x = -1, y = 10 (x < 0 && y == 10)
  x = -1, y = 15 (x < 0 && y > 10)
  x = -1, y =  9 (x < 0 && y < 10)
  x =  0, y =  9 (x == 0 && y < 10)
  x =  1, y =  9 (x > 0 && y < 10)
JUnit

- Unit-test framework for Java programs
- open source software
- hosted on SourceForge: http://junit.sourceforge.net/javadoc
  - Moved to http://junit.org (for JUnit 4 and later)
- not in the standard JDK:

  import junit.framework.*;
  // for JUnit 3.8 and earlier

  import org.junit.*; // for JUnit 4 and later

- Associate a Test class with each unit
- one or more classes

JUnit

- The test class has a set of test methods

  public void testX()

  where \( X \) is the method to be tested

- The test methods use “assertions” to perform the tests, ex:

  Assert.assertTrue(c)
  Assert.assertEquals(x, y)
  Assert.assertSame(obj1, obj2)
<table>
<thead>
<tr>
<th>method name / parameters</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertTrue(test)</code></td>
<td>Causes this test method to fail if the given boolean test is not true.</td>
</tr>
<tr>
<td><code>assertTrue(&quot;message&quot;, test)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertFalse(test)</code></td>
<td>Causes this test method to fail if the given boolean test is not false.</td>
</tr>
<tr>
<td><code>assertFalse(&quot;message&quot;, test)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertEquals(expectedValue, value)</code></td>
<td>Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the <code>equals</code> method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.</td>
</tr>
<tr>
<td><code>assertEquals(&quot;message&quot;, expectedValue, value)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNotEquals(value1, value2)</code></td>
<td>Causes this test method to fail if the given two values are equal to each other. (For objects, it uses the <code>equals</code> method to compare them.)</td>
</tr>
<tr>
<td><code>assertNotEquals(&quot;message&quot;, value1, value2)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNull(value)</code></td>
<td>Causes this test method to fail if the given value is not null.</td>
</tr>
<tr>
<td><code>assertNull(&quot;message&quot;, value)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNotNull(value)</code></td>
<td>Causes this test method to fail if the given value is null.</td>
</tr>
<tr>
<td><code>assertNotNull(&quot;message&quot;, value)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertSame(expectedValue, value)</code></td>
<td>Identical to <code>assertEquals</code> and <code>assertNotEquals</code> respectively, except that for objects, it uses the <code>==</code> operator rather than the <code>equals</code> method to compare them. (The difference is that two objects that have the same state might be <code>equals</code> to each other, but not <code>==</code> to each other. An object is only <code>==</code> to itself.)</td>
</tr>
<tr>
<td><code>assertSame(&quot;message&quot;, expectedValue, value)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNotSame(value1, value2)</code></td>
<td></td>
</tr>
<tr>
<td><code>assertNotSame(&quot;message&quot;, value1, value2)</code></td>
<td></td>
</tr>
<tr>
<td><code>fail()</code></td>
<td>Causes this test method to fail.</td>
</tr>
<tr>
<td><code>fail(&quot;message&quot;)</code></td>
<td></td>
</tr>
</tbody>
</table>
public class Calculator {
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
}
JUnit

CalculatorTest.java

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

public class CalculatorTest {
    @Test
    public void evaluatesExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
    }
}
```
JUnit

java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest

JUnit version 4.12

Time: 0,006

OK (1 test)
public class Calculator {
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\\+"))
            sum -= Integer.valueOf(summand);
        return sum;
    }
}

JUnit

java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest

JUnit version 4.12
.E
Time: 0.007
There was 1 failure:
1) evaluatesExpression(CalculatorTest)
java.lang.AssertionError: expected:<6> but was:<-6>
   at org.junit.Assert.fail(Assert.java:88)
   ...
FAILURES!!!
Tests run: 1, Failures: 1
Right-click Calculator.java and choose Tools > Create Tests.

In the project Properties -> Add Library JUnit
Eclipse IDE

Open the New wizard (File > New > JUnit Test Case).
Building unit tests with JUnit

- Initialize any instance variables necessary for testing in the test object
- Define tests for emptiness, equality, boundary conditions, ...
- Define test suites, if necessary, to group tests.
- Use Assert methods to perform tests
JUnit 3.8 vs. 4

• JUnit 4: all test methods are annotated with @Test.
  • Unlike JUnit3 tests, you do not need to prefix the method name with "test".

• JUnit 4 does not have the test classes extend junit.framework.TestCase (directly or indirectly).
  • Usually, tests with JUnit4 do not need to extend anything (which is good, since Java does not support multiple inheritance).
public class StatCompiler {

    /**
     * a, b, & c must all be positive
     **/
    public static int averageOfPosInts(int a, int b, int c) throws IllegalArgumentException{
        if ((a < 0) || (b <0) || (c < 0))
            throw new IllegalArgumentException("No neg values");
        int sum = a + b + c;
        return sum/3;
    }

    public static int median(int a, int b, int c){
        if ( (a >=b) && (a <=c))    return a;
        else if ((a >= b) && (a >=c))    return b;
        else    return c;
    }
}
import junit.framework.*;
    // JUnit 3.8
public class StatCompilerTest extends TestCase {

    public StatCompilerTest(java.lang.String testName) {
        super(testName);
    }

    public void testAverageOfPosInts() {
        System.out.println("testAverageOfPosInts");
        Assert.assertEquals(StatCompiler.averageOfPosInts(1, 2, 3), 2);
        try{
            StatCompiler.averageOfPosInts(-1, 2, 3);
            fail("Exception should have been thrown");
        } catch (IllegalArgumentException iae) {}  
    }

    public void testMedian() {
        System.out.println("testMedian");
        Assert.assertEquals(2, StatCompiler.median(1, 2, 3));
        Assert.assertEquals(2, StatCompiler.median(3, 2, 1));
    }
}
JUnit version 3.8

testAverageOfPosInts
testMedian

===================================================================
Errors logged for the StatCompilerTest test:
    No errors.
===================================================================
Failures logged for the StatCompilerTest test:
    Total failures: 1
Test case testMedian(StatCompilerTest) failed with "expected:<2>
but was:<3>" at
StatCompilerTest.testMedian(StatCompilerTest.java:42)
===================================================================

Summary of StatCompilerTest test:
    Result: Failed

Run: 2
Failures: 1
Errors: 0
Elapsed time: 0.01
import org.junit.Test;
import static org.junit.Assert.*;
public class StatCompilerTest {
    @Test
    public void testAverageOfPosInts() {
        System.out.println("averageOfPosInts");
        int a = 1;
        int b = 2;
        int c = 3;
        int expResult = 2;
        int result = StatCompiler.averageOfPosInts(a, b, c);
        assertEquals(expResult, result);
    }
    @Test
    public void testMedian() {
        System.out.println("median");
        int a = 3;
        int b = 2;
        int c = 1;
        int expResult = 2;
        int result = StatCompiler.median(a, b, c);
        assertEquals(expResult, result);
    }
}
NetBeans and JUnit: Download the JUnit library and add it in the path. The JUnit plugin is installed.
Run JUnit version 4

Run: java org.junit.runner.JUnitCore [test class name]

JUnit version 4.11
.testAverageOfPosInts
.testMedian
Time: 0.005
There was 1 failure:
1) testMedian(JUnit_test_01)
java.lang.AssertionError: expected:<2> but was:<3>

FAILURES!!!
Tests run: 2, Failures: 1
Notes on Static import

- Static import is a feature introduced in the Java programming language that allows members (fields and methods) defined in a class as public static to be used in Java code without specifying the class in which the field is defined.

- The mechanism can be used to reference individual members of a class:

  ```java
  import static java.lang.Math.PI;
  import static java.lang.Math.pow;
  ```

- Or all the static members of a class:

  ```java
  import static java.lang.Math.*;
  ```
import static java.lang.Math.*;
// OR
// import static java.lang.Math.PI;
// import static java.lang.Math.pow;

import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello World!");
        out.println("A circle with a diameter of 5 cm has:");
        out.println("A circumference of " + (PI * 5) + " cm");
        out.println("And an area of " + (PI * pow(2.5,2)) + " sq. cm");
    }
}
Notes on Assertions

• An assertion is a Java statement that enables you to assert an assumption about your program.

• An assertion contains a Boolean expression that should be true during program execution.

• Assertions can be used to assure program correctness and avoid logic errors.
An assertion is declared using the Java keyword `assert` in JDK 1.5 as follows:

```java
assert assertion;    // OR
assert assertion : detailMessage;
```

where assertion is a Boolean expression and detailMessage is a primitive-type or an Object value.
public class AssertionDemo {
    public static void main(String[] args) {
        int i; int sum = 0;
        for (i = 0; i < 10; i++) {
            sum += i;
        }
        assert i==10;
        assert sum>10 && sum<5*10 : "sum is " + sum;
    }
}
Executing Assertions

- When an assertion statement is executed, Java evaluates the assertion.
  - If it is false, an AssertionError will be thrown.
  - The AssertionError class has a no-arg constructor and seven overloaded single-argument constructors of type int, long, float, double, boolean, char, and Object.
  - For the first assert statement with no detail message, the no-arg constructor of AssertionError is used.
  - For the second assert statement with a detail message, an appropriate AssertionError constructor is used to match the data type of the message.
  - Since AssertionError is a subclass of Error, when an assertion becomes false, the program displays a message on the console and exits.
Running Programs with Assertions

- By default, the assertions are disabled at runtime. To enable it, use the switch `–enableassertions`, or `–ea` for short, as follows:

  ```java
  java –ea AssertionDemo
  ```

  ```java
  public class AssertionDemo {
      public static void main(String[] args) {
          int i; int sum = 0;
          for (i = 0; i < 10; i++) {
              sum += i;
          }
          assert i!=10;
      }
  }
  ```

  Exception in thread "main" java.lang.AssertionError
  at AssertionDemo.main(AssertionDemo.java:7)

(c) Paul Fodor
Running Programs with Assertions

• Assertions can be selectively enabled or disabled at class level or package level.
  • The disable switch is –disableassertions or –da for short.
  • For example, the following command enables assertions in package package1 and disables assertions in class Class1.

```
java -ea:package1 -da:Class1 AssertionDemo
```
Using Exception Handling or Assertions?

- Assertion should not be used to replace exception handling.
  - Exception handling deals with unusual circumstances during program execution.
  - Assertions are to assure the correctness of the program.
  - Exception handling addresses robustness and assertion addresses correctness.
  - Assertions are used for internal consistency and validity checks.
  - Assertions are checked at runtime and can be turned on or off at startup time.
Using Exception Handling or Assertions?

- Do not use assertions for argument checking in public methods:
  - Valid arguments that may be passed to a public method are considered to be part of the method’s contract.
  - The contract must always be obeyed whether assertions are enabled or disabled.
  - For example, the following code in the Circle class should be rewritten using exception handling:

```java
public void setRadius(double newRadius) {
    assert newRadius >= 0;
    radius = newRadius;
}
```
Using Exception Handling or Assertions?

- Use assertions to reaffirm assumptions.
  - This gives you more confidence to assure correctness of the program.
- A common use of assertions is to replace assumptions with assertions in the code.
- A good use of assertions is place assertions in a switch statement without a default case. For example:

```java
switch (month) {
    case 1: ... ; break;
    case 2: ... ; break;
    ...
    case 12: ... ; break;
    default: assert false : "Invalid month: " + month;
}
```