

CMSC 433 – Programming Language  
Technologies and Paradigms  
Spring 2003

Testing, Debugging, and Tools  
February 13, 2003

## The Testing Environment

- Want to create a scaffold for executing tests
  - Code infrastructure to run tests and check output
- Many benefits
  - Can automate testing process
  - Useful for regression testing
- But, can take some time to implement

## Testing Environment Components

- A *user* to generate input for tested component
- An *oracle* for verifying the results are correct
- These two may be combined into a single system

## Unit Testing with JUnit

- Testing environment for writing black-box tests
  - Write special **TestCase** classes to test other classes
  - Several ways to use/set up test cases
- Can be downloaded from
  - <http://www.junit.org>

## JUnit Philosophy

- Iterative, incremental process
  - Write small black-box test cases (as needed)
  - Test-as-you-go
    - i.e., after changes, when new method added, when bug identified
- **JUnit** test cases must be completely automated
  - No human judgment
  - Easy to run many of them at the same time
- Goal: lots of bang for the buck
  - Even simple tests can find many bugs quickly

## TestCase Example with Lists

```
public class ListTest extends TestCase {
    public void testAdd() {
        LinkedList l = new LinkedList();
        Object o = new Object();
        l.add(o);
        assertTrue(l.contains(o));
    }

    public void runTest() { testAdd(); }

    public static void main(String args[]) {
        TestResult result = (new ListTest()).run();
        if (!result.wasSuccessful()) {
            System.out.println("Doh!");
        }
    }
}
```

Create objects

Perform test/  
check result

## JUnit Components

- Test cases (class **TestCase**)
  - Individual tests
  - Can reuse test case setup (optional)
- Test suites (class **TestSuite**)
  - Test case container
- Test runner (various classes)
  - Executes test suites and presents results

## Each test has three 3 parts

- Code that creates test objects
  - Create a subclass of `junit.framework.TestCase`
- Code that executes the test
  - Override the method `runTest()` (which executes the test)
- Code that verifies the result
  - e.g., use `junit.framework.assertTrue()` to check results (throws exception is test fails)

## Setup/Teardown

- Creating objects for each test too simple
  - Setup overhead grows as number of tests grows
  - Instead, group setup (and teardown) code in one place and reuse
- junit.framework.TestCase.run() executes test case:
  - public void **run()** { setUp(); runTest(); tearDown(); }
  - Put setup code in setUp() method
  - Put cleanup code in tearDown() method

## TestCase Example, again

```
public class ListTest extends TestCase {
    private Object o;
    public void setUp() { o = new Object(); }
    public void testAdd() {
        LinkedList l = new LinkedList();
        l.add(o);
        assertTrue(l.contains(o));
    }
    public void testPushPop() {
        LinkedList l = new LinkedList();
        Object o2;
        l.addFirst(o);
        o2 = l.removeFirst();
        assertTrue(o==o2);
        assertTrue(l.size()==0);
    }
}
```

Create objects at outset

Perform test/check result

## More Asserts

- Junit has several different tests
  - assertTrue(b) -- asserts that b is true
  - assertFalse(b) -- asserts that b is false
  - assertEquals(o1, o2) -- assert that o1.equals(o2)
  - assertNotNull(o) -- assert o != null
  - assertNull(o) -- assert o == null
  - assertSame(o1, o2) -- assert o1==o2
  - assertNotSame(o1, o2) -- assert o1 != o2

## Running many tests with Test Suites

```
public class ListTest extends TestCase {
    ...
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new ListTest3() {
            protected void runTest() { testAdd(); }
        });
        suite.addTest(new ListTest3() {
            protected void runTest() { testPushPop(); }
        });
        return suite;
    }
}
```

## Test Suites (cont'd)

- You can also create test suites more easily:

```
public static Test suite() {  
    TestSuite suite = new TestSuite();  
    suite.addTest(new ListTest("testAdd"));  
    suite.addTest(new ListTest("testPushPop"));  
    return suite;  
}
```

- Or simply:

```
public static Test suite() {  
    return new TestSuite(ListTest.class);  
}
```

## Test Runner

- To execute test suite, pick a class:

- For graphical display

- `junit.awtui.TestRunner TestCaseClass` or
- `junit.swingui.TestRunner TestCaseClass`

- For textual display

- `junit.textui.TestRunner TestCaseClass`

- Or run from within your own code:

```
public static void main(String args[]) {  
    junit.textui.TestRunner.run(suite());  
}
```

## Running Tests Easily

- Put `import junit.framework.*;` at top of file
- Test runners will use static `suite()` method
- If no `suite()` method, suite selected automatically
  - Every method that is **public**, returns **void**, takes no arguments, and begins with "test"
- Then use `junit.*.TestRunner TestCase`
  - Or use DrJava

## IDEs

- IDE: Interactive Development Environment
  - Editor
    - Usually with some nice syntax coloring, indentation features
  - Compiler
    - Errors sorted, displayed nicely; easy to see corresponding code
  - Debugger
    - Closely watch/change execution of source code
  - Etc...
    - Testing, search, code transformations, ...
- Examples: DrJava, NetBeans, Eclipse, Visual Studio, emacs

## Dr. Java

- Light-weight IDE
- Editing
  - Syntax coloring, auto-indent, brace matching
- Testing
  - Integrates with Junit testing framework
    - Uses suite() or auto-generated suite
  - Interaction panel allows interactive method invocations
- Debugging
  - Integrates with Java debugger
  - Interactions panel also useful

## Debugging

- My program doesn't work: why?
- Use the scientific method:
  - Study the data
    - Some tests work, some don't
  - Hypothesize what could be wrong
  - Run experiments to check your hypotheses
    - Testing!
  - Iterate

## Starting to Debug

- What are the symptoms of the misbehavior?
  - Input/output
  - Stack trace (from thrown exception)
- Where did the program fail?
- What could have led to this failure?
- Test possible causes, narrow down the problem

## Checking that Properties Hold

- Print statements
  - Check whether values are correct
    - E.g., look at value of  $i$  to check if  $i > 0$
  - Check whether control-flow is correct
    - E.g., see if  $f()$  is called after  $g()$
- Automatic debugger
  - Allows you to step through the program interactively
  - Verify expected properties
    - Don't need to put in print statements and recompile
  - Use as part of testing

### Dr. Java Interactions Pane

- Can evaluate Java expressions interactively
  - Can bind variables, execute expressions/statements
- Benefits
  - Make sure that methods work as expected
  - Test invariants by constructing expressions not in program text
  - Combines with interactive debugger

### Dr. Java's Automatic Debugger

- Set execution breakpoints
- Step through execution
  - **into**, **over**, and **out** of method calls
- Examine the stack
- Examine variable contents
- Set watchpoints
  - Notified when variable contents change

### Using the Debugger

- Set debug mode to on
  - Turns on debug panel with state information
- Set break point(s) in Java source
- Run the program

### Tips

- Make bug reproducible
  - If it's not reproducible, what does that imply?
- Boil down to smallest program that reproduces bug
  - Reveals the core problem
- Explain problem to someone else (i.e., instructor or TA)
  - Explaining may reveal the flaw in your logic
- Keep notes: don't make the same mistake twice

## Defensive Programming

- Assume that other methods/classes are broken
  - They will mis-use your interface

```
public Vector(int initialCapacity, int
capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException(
            "Illegal Capacity: "+ initialCapacity);
    ... }
```

- Goal: Identify errors as soon as possible

## Avoiding Errors

- Codify your assumptions
  - Include checks when entering/exiting functions, iterating on loops
- Test as you go
  - Using Junit
  - Using the on-line debugger
- Re-test when you fix a bug
  - Be sure you didn't introduce a new bug
- Do not ignore possible error states
  - Deal with exceptions appropriately