Running many tests with Test Suites

```java
public class LogRecordTest extends TestCase {
    ...
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new LogRecordTest("equals1") {
            protected void runTest() { testEquals1();}
        });
        suite.addTest(new LogRecordTest("equals2") {
            protected void runTest() { testEquals2();}
        });
        return suite;
    }
}
```

Test Suites (cont’d)

- If you follow certain constraints (discussed later), you can create test suites more easily:
  ```java
  public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTest(new LogRecordTest("testEquals1"));
      suite.addTest(new LogRecordTest("testEquals2"));
      return suite; }
  ```
- Or simply:
  ```java
  public static Test suite() { return new TestSuite(LogRecordTest.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:
  ```java
  public static void main(String args[]) {
      junit.textui.TestRunner.run(suite());
  }
  ```

Using JUnit with DrJava

- At the top of the file, include:
  - import junit.framework.TestCase;
- The main class of the file:
  - must be public, extend TestCase, and have a constructor of the form:
    - public classname(String name) { super(name);}
- Tests run automatically
  - must be public and not static, return void, take no arguments, and have a name beginning with test
  - can use suite() as well
- Verify results using
  - void assertTrue(String, boolean), void assertEquals(String, int, int), and void fail(String)
- Set up tests using
  - protected void setUp()
import junit.framework.*;
import java.io.*;

public class LogRecordTest extends TestCase {
    protected String event1, event2;
    LogRecord tmp1, tmp2, tmp3;

    public LogRecordTest(String name) { super (name); }

    protected void setUp() {
        event1 = "event string1"; event2 = "event string2";
        tmp1 = new LogRecord(event1);
        tmp2 = new LogRecord(event2);
        tmp3 = new LogRecord(event2);
    }

    public void testEquals1() {
        assertTrue(tmp1.equals(tmp1));
    }

    public void testEquals2() {
        assertTrue(!tmp1.equals(tmp2));
    }

    public void testEquals3() {
        assertTrue(!tmp3.equals(tmp2));
    }

    public void testPubConstructor1() {
        assertTrue(tmp1.getEvent() == event1);
        assertTrue(tmp1.getTimestamp().compareTo(new java.util.Date()) <= 0);
    }

    public void testCompareTo1() {
        assertTrue(tmp1.compareTo(tmp1) == 0);
        assertTrue(tmp1.compareTo(tmp2) < 0);
        assertTrue(tmp2.compareTo(tmp1) > 0);
        assertTrue(tmp2.compareTo(tmp3) < 0);
        assertTrue(tmp3.compareTo(tmp2) > 0);
        assertTrue(tmp1.compareTo(tmp3) < 0);
        assertTrue(tmp3.compareTo(tmp1) > 0);
    }
}

Example, cont’d

public void testEquals3() {
    assertTrue(tmp3.equals(tmp2));
}

public void testPubConstructor1() {
    assertTrue(tmp1.getEvent() == event1);
    assertTrue(tmp1.getTimestamp().compareTo(new java.util.Date()) <= 0);
}

public void testCompareTo1() {
    assertTrue(tmp1.compareTo(tmp1) == 0);
    assertTrue(tmp1.compareTo(tmp2) < 0);
    assertTrue(tmp2.compareTo(tmp1) > 0);
    assertTrue(tmp2.compareTo(tmp3) < 0);
    assertTrue(tmp3.compareTo(tmp2) > 0);
    assertTrue(tmp1.compareTo(tmp3) < 0);
    assertTrue(tmp3.compareTo(tmp1) > 0);
    }
Example, cont’d

```java
public void testFormatFromFormat1() {
    StringWriter s = new StringWriter();
    tmp1.format(new PrintWriter(s));
    LogRecord tmp4 = tmp1.fromFormat(new BufferedReader(new StringReader(s.toString())));
    assertTrue(tmp1.toString().equals(tmp4.toString()));
}
public static Test suite() {
    return new TestSuite(LogRecordTest.class);
}
public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
}
```
Structural Coverage Testing

- Adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
- Attempted compromise between the impossible and the inadequate

Statement Coverage

One test datum \((N=1, A[0]=-7, X=9)\) is enough to guarantee statement coverage of function select
Faults in handling positive values of \(A[i]\) would not be revealed
We must add a test datum (N=1, A[0]=7, X=9) to cover branch False of the if statement. Faults in handling positive values of A[i] would be revealed. Faults in exiting the loop with condition A[i] < X would not be revealed.

Both conditions (i<N), (A[i]<X) must be false and true for different tests. In this case, we must add tests that cause the while loop to exit for a value greater than X. Faults that arise after several iterations of the loop would not be revealed.
Debugging

- My program doesn’t work: why?
- Some part of the program has a “bug;” narrow down the possible locations of the bug
  - Figure out which parts of the program work
  - Test the rest
  - Iterate
- How to figure out which parts work?
  - Testing!
Starting to Debug

- What are the symptoms of the misbehavior?
  - Input/output
  - Stack trace (from thrown exception)
- At what point did the program fail?
- Reason backwards: what could have led to this failure?
- What invariants should have been preserved?
- Test the invariants, narrow down the problem

Checking that Invariants Hold

- Print statements
  - Print out expected invariants
- Automatic debugger
  - Allows you to step through the program interactively
  - Verify expected invariants
  - 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

• Start Dr. Java with the debug libraries:
  – java -classpath /usr/local/drjava/drjava-20020814.jar:/usr/local/j2sdk1.4.0/lib/tools.jar edu.rice.cs.drjava.DrJava

• Creates debugging menu
  – Select debug mode to on
    • Turns on debug panel with state information

• Set break point(s) in Java source
• Run the program

Tips

• Make the bug reproducible
  – If it’s not reproducible, what does that imply?

• Boil it down to the smallest program that reproduces the bug
  – Reveals the core problem

• Explain the problem to someone else (i.e. the instructor or TA)
  – Explaining may reveal the flaw in your logic

• Keep notes: don’t make the same mistake twice
Avoiding Errors

- Test as you go
  - Using JUnit
  - Using the on-line debugger
- Do not ignore possible error states
  - Deal with exceptions appropriately
- Codify your invariants
  - Include assertions in the code when entering/exiting functions, iterating on loops