

Objects Analysis

Threads



Design

15-214

15-214

toad

Spring 2013

Principles of Software Construction: Objects, Design and Concurrency

Testing

Christian Kaestner

Charlie Garrod

Learning Goals

- General considerations of testing
- Understand the possibilities and limitations of unit testing
- Ability to use JUnit to write unit tests
 - Reasonably sized unit tests
 - Whole suite
- Test suites as a design tool for testable code
- Understand test coverage goals: The good, the bad, and the ugly
 - Ability to use EclEmma for line coverage

Functional Correctness

- **Specification**

- **Formal Verification**

- **Unit Testing**

- **Type Checking**

- **Statistic Analysis**



- **Requirements definition**

- **Inspections, Reviews**

- **Integration / System / Acceptance / Regression / GUI / BI
ackbox / Model-Based / Random Testing**

- **Change / Release Management**

15-214

15-313

Testing

- Executing the program with selected inputs in a controlled environment
- Goals:
 - Reveal bugs (main goal)
 - Assess quality (hard to quantify)
 - Clarify the specification, documentation
 - Verify contracts

**"Testing shows the presence,
not the absence of bugs**

Edsger W. Dijkstra 1969

What to test?

- Functional correctness of a method (e.g., computations, contracts)
- Functional correctness of a class (e.g., class invariants)
- Behavior of a class in a subsystem/multiple subsystems/the entire system
- Behavior when interacting with the world
 - Interacting with files, networks, sensors, ...
 - Erroneous states
 - Nondeterminism, Parallelism
 - Interaction with users
- ...

Testing Decisions

Who tests?

- Developers
- Other Developers
- Separate Quality Assurance Team
- Customers

When to test?

- Before development
- During development
- After milestones
- Before shipping

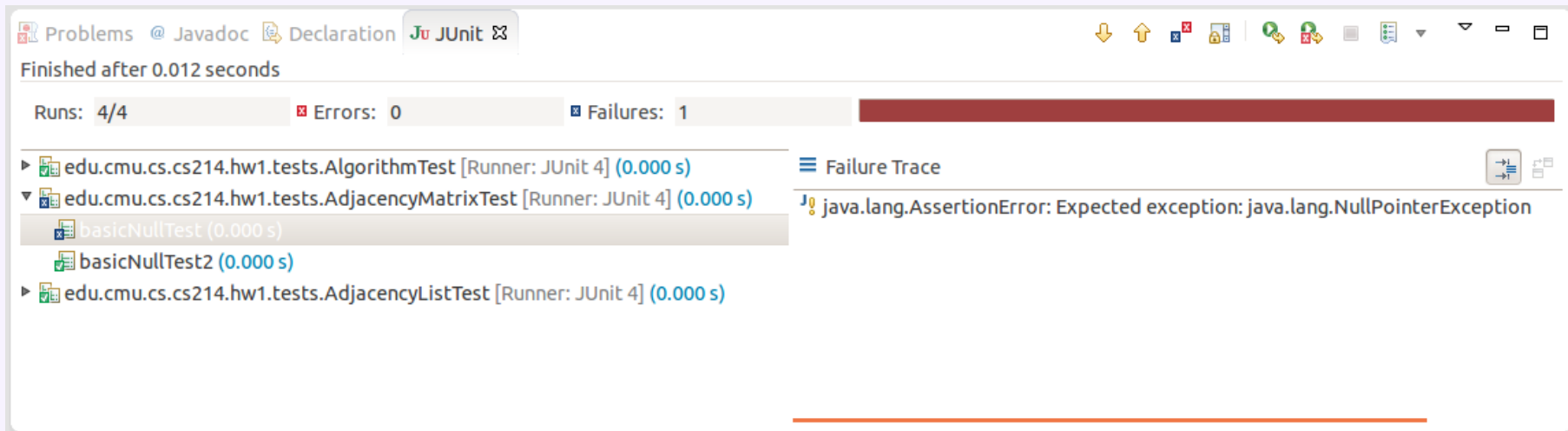
Discuss tradeoffs

Unit Tests

- Testing units of source code
 - Smallest testable part of a system
 - Test parts before assembling them
 - Typically small units (methods, interfaces), but later units are possible (packages, subsystems)
 - Supposed to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Little dependencies on other system parts or environment
- Insufficient but good starting point, extra benefits:
 - Documentation (executable specification)
 - Design mechanism (design for testability)

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism



The screenshot shows an IDE's JUnit runner interface. At the top, it says "Finished after 0.012 seconds". Below that, a summary bar indicates "Runs: 4/4", "Errors: 0", and "Failures: 1". The test results list includes:

- edu.cmu.cs.cs214.hw1.tests.AlgorithmTest [Runner: JUnit 4] (0.000 s)
- edu.cmu.cs.cs214.hw1.tests.AdjacencyMatrixTest [Runner: JUnit 4] (0.000 s)
 - basicNullTest (0.000 s) - Failed
 - basicNullTest2 (0.000 s)
- edu.cmu.cs.cs214.hw1.tests.AdjacencyListTest [Runner: JUnit 4] (0.000 s)

The failure trace for the failed test is shown on the right:

```
java.lang.AssertionError: Expected exception: java.lang.NullPointerException
```



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

public class AdjacencyListTest {
    @Test
    public void testSanityTest(){
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g1.addVertex(s1));
        assertEquals(true, g1.addVertex(s2));
        assertEquals(true, g1.addEdge(s1, s2));
        assertEquals(s2, g1.getNeighbors(s1)[0]);
    }

    @Test
    public void test....

    private int helperMethod...
}
```

Small
setup

Check
expected
results

assert, Assert

- `assert` is a native Java statement throwing an `AssertionError` exception when failing
 - `assert` expression: "Error Message";
- `org.junit.Assert` is a library that provides many more specific methods
 - static void [assertTrue](#)(java.lang.String message, boolean condition) // Asserts that a condition is true.
 - static void [assertEquals](#)(java.lang.String message, long expected, long actual); // Asserts that two longs are equal.
 - static void [assertEquals](#)(double expected, double actual, double delta); // Asserts that two doubles or floats are equal to within a positive delta.
 - static void [assertNotNull](#)(java.lang.Object object) // Asserts that an object isn't null.
 - static void [fail](#)(java.lang.String message) //Fails a test with the given message.

JUnit Conventions

- TestCase collects multiple tests (one class)
- TestSuite collects test cases (typically package)
- Tests should run fast
- Test should be independent

- Tests are methods without parameter and return value
- AssertionError signals failed test (unchecked exception)

- Test Runner knows how to run JUnit tests
 - (uses reflection to find all methods with @Test annotat.)

Common Setup

```
import org.junit.*;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    Graph g;

    @Before
    public void setUp() throws Exception {
        graph = createTestGraph();
    }

    @Test
    public void testSanityTest(){
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g.addVertex(s1));
    }

    @BeforeClass ... //avoid that
```

Checking for presence of an exception

```
import org.junit.*;
import static org.junit.Assert.fail;

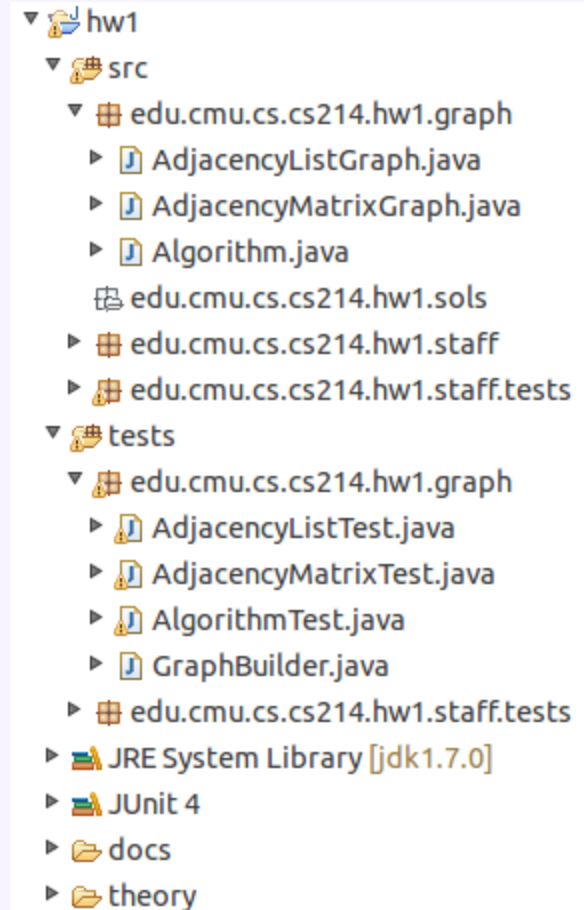
public class Tests {

    @Test
    public void testSanityTest(){
        try {
            openNonexistingFile();
            fail("Expected exception");
        } catch(IOException e) { }
    }

    @Test(expected = IOException.class)
    public void testSanityTestAlternative() {
        openNonexistingFile();
    }
}
```

Test organization

- Conventions (not requirements)
- Have a test class `A`Test for each class `A`
- Have a source directory and a test directory
 - Store `A`Test and `A` in the same package
 - Tests can access members with default (package) visibility
- Alternatively store exceptions in the source directory but in a separate package



Run tests frequently

- You should only commit code that is passing all tests
- Run tests before every commit
- Run tests before trying to understand other developers' code
- If entire test suite becomes too large and slow for rapid feedback, run tests in package frequently, run all tests nightly
 - Medium sized projects easily have 1000s of test cases and run for minutes
- Continuous integration servers help to scale testing

Continuous Integration

The screenshot shows the Jenkins web interface. At the top, there's a blue header with the 'Jenkins' logo, a search bar, and user options like 'admin | log out'. Below the header, there's a navigation sidebar on the left with links for 'New Job', 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', 'My Views', and 'Disk usage'. The main content area displays a table of build jobs. The table has columns for 'S' (Status), 'W' (Weather icon), 'Name', 'Last Success', 'Last Failure', and 'Last Duration'. The jobs listed are FOSPL, IVM, IVMBranh, IVMBranhEval, IVMBranhTest, IVMTest, TypeChef, and variational. Below the table, there are links for 'Icon: S M L', 'Legend', and three RSS feeds: 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'. At the bottom, there's a footer with 'Help us localize this page', 'Page generated: Jan 29, 2013 10:41:11 PM', 'REST API', and 'Jenkins ver. 1.500'.

S	W	Name	Last Success	Last Failure	Last Duration
		FOSPL	1 hr 40 min (#186)	6 days 8 hr (#164)	47 sec
		IVM	2 days 19 hr (#288)	12 days (#279)	4 min 35 sec
		IVMBranh	3 mo 19 days (#139)	3 mo 25 days (#125)	4 min 27 sec
		IVMBranhEval	3 mo 24 days (#70)	3 mo 28 days (#57)	12 min
		IVMBranhTest	3 mo 24 days (#110)	3 mo 19 days (#118)	11 min
		IVMTest	2 days 19 hr (#160)	10 days (#155)	12 min
		TypeChef	21 days (#354)	7 hr 54 min (#357)	16 min
		variational	1 yr 2 mo (#11)	1 yr 2 mo (#3)	3 min 43 sec

See also travis-ci.org

What to test? How much to test?

Writing Test Cases: Common Strategies

- Read specification
- Write test case(s) for representative case
 - Small instances are usually sufficient
- Write test case for invalid case
- Write test case to check boundaries
- Are there difficult cases? (error guessing)
- Think like a user, not like a programmer

- Specification covered?
- Feel confident? Time/money left?

Example

```
/**  
 * computes the sum of the first len values of the array  
 *  
 * @param array array of integers of at least length len  
 * @param len number of elements to sum up  
 * @return sum of the array values  
 */  
int total(int array[], int len);
```

- Test empty array
- Test array of length 1 and 2
- Test negative numbers
- Test invalid length (negative or longer than array.length)
- Test null as array
- Others?

Blackbox testing

Testable Code

- Think about testing when writing code
- Unit testing encourages to write testable code
- Separate parts of the code to make them independently testable
- Abstract functionality behind interface, make it replaceable
- Recommended as design method Test-Driven Development by some

Structural Analysis for Test Coverage

- Organized according to program decision structure
- Touching: statement, branch

```
public static int binsrch (int[] a, int key) {
```

```
    int low = 0;  
    int high = a.length - 1;
```

```
    while (true) {
```

```
        if ( low > high ) return -(low+1);
```

```
        int mid = (low+high) / 2;
```

```
        if ( a[mid] < key ) low = mid + 1;
```

```
        else if ( a[mid] > key ) high = mid - 1;
```

```
        else return mid;
```

```
    }  
}
```

- Will this statement get executed in a test?
- Does it return the correct result?

- Could this array index be out of bounds?

- Does this return statement ever get reached?

Method Coverage

- Trying to execute each method as part of at least one test

```
38     }
39     public boolean equals(Object anObject) {
40         if (isZero())
41             if (anObject instanceof IMoney)
42                 return ((IMoney)anObject).isZero();
43         if (anObject instanceof Money) {
44             Money aMoney= (Money)anObject;
45             return aMoney.currency().equals(currency())
46                 && amount() == aMoney.amount();
47         }
48         return false;
49     }
50 }
```

- Does this guarantee correctness?

Statement Coverage

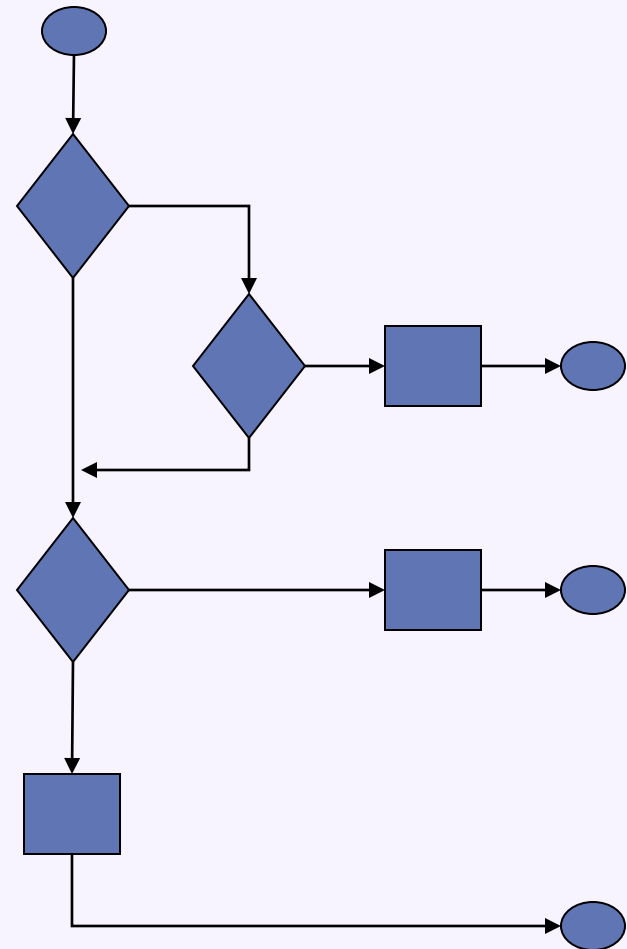
- Trying to test all parts of the implementation
- Execute every statement in at least one test

```
38 }
39 public boolean equals(Object anObject) {
40     if (isZero())
41         if (anObject instanceof IMoney)
42             return ((IMoney)anObject).isZero();
43     if (anObject instanceof Money) {
44         Money aMoney= (Money)anObject;
45         return aMoney.currency().equals(currency())
46             && amount() == aMoney.amount();
47     }
48     return false;
49 }
```

- Does this guarantee correctness?

Structure of Code Fragment to Test

```
38 }
39 public boolean equals(Object anObject) {
40     if (isZero())
41         if (anObject instanceof IMoney)
42             return ((IMoney)anObject).isZero();
43     if (anObject instanceof Money) {
44         Money aMoney= (Money)anObject;
45         return aMoney.currency().equals(currency())
46             && amount() == aMoney.amount();
47     }
48     return false;
49 }
```



**Flow chart diagram for
junit.samples.money.Money.equals**

Statement Coverage

- **Statement coverage**

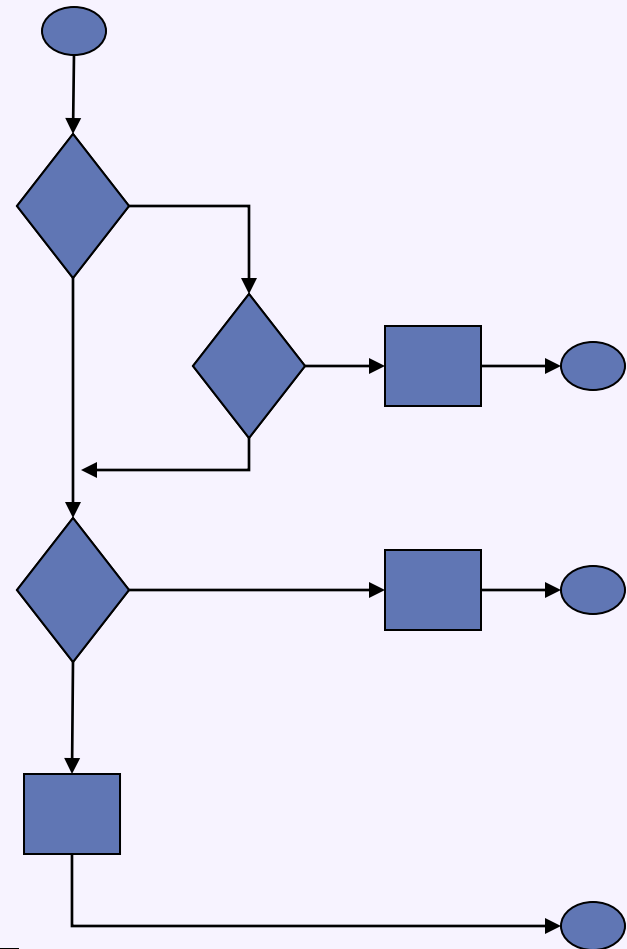
- What portion of program statements (nodes) are touched by test cases

- **Advantages**

- Test suite size linear in size of code
- Coverage easily assessed

- **Issues**

- Dead code is not reached
- May require some sophistication to select input sets
- Fault-tolerant error-handling code may be difficult to “touch”
- Metric: Could create incentive to *remove* error handlers!



```
38 public boolean equals(Object anObject) {  
39     if (isZero())  
40         if (anObject instanceof IMoney)  
41             return ((IMoney)anObject).isZero();  
42     if (anObject instanceof Money) {  
43         Money aMoney= (Money)anObject;  
44         return aMoney.currency().equals(currency())  
45             && amount() == aMoney.amount();  
46     }  
47     return false;  
48 }  
49 }
```

toad

Branch Coverage

- **Branch coverage**

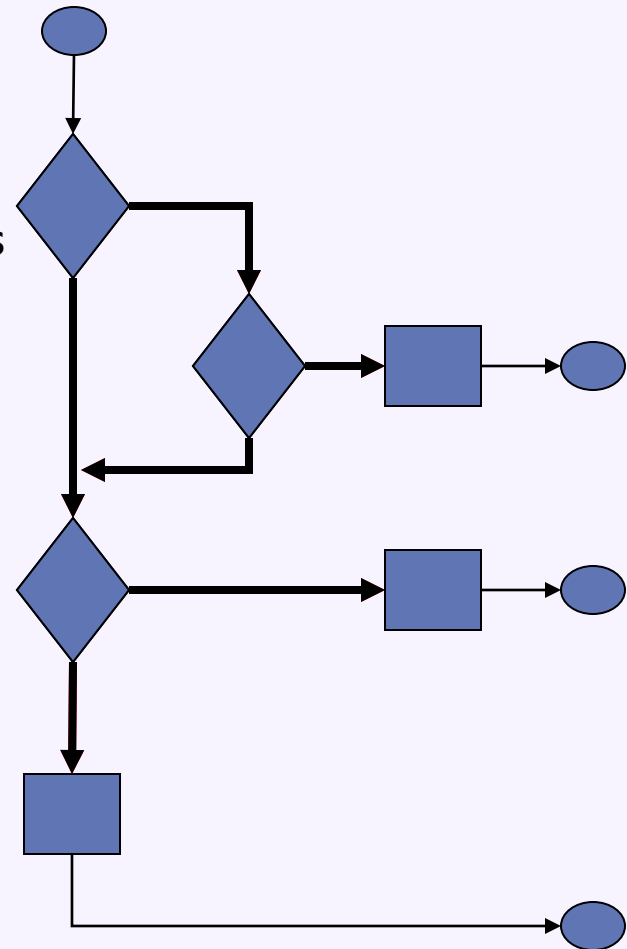
- What portion of condition branches are covered by test cases?
- *Or:* What portion of relational expressions and values are covered by test cases?
 - Condition testing (Tai)
- **Multicondition coverage** – all boolean combinations of tests are covered

- **Advantages**

- Test suite size and content derived from structure of boolean expressions
- Coverage easily assessed

- **Issues**

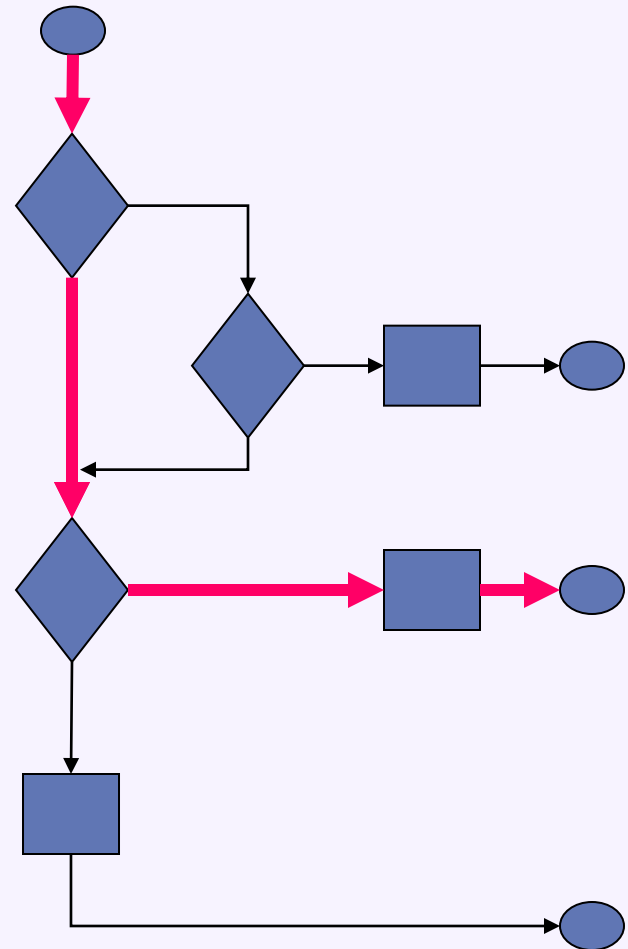
- Dead code is not reached
- Fault-tolerant error-handling code may be difficult to “touch”



```
37  
38  
39 public boolean equals(Object anObject) {  
40     if (isZero())  
41         if (anObject instanceof IMoney)  
42             return ((IMoney)anObject).isZero();  
43     if (anObject instanceof Money) {  
44         Money aMoney= (Money)anObject;  
45         return aMoney.currency().equals(currency())  
46             && amount() == aMoney.amount();  
47     }  
48     return false;  
49 }
```

Path Coverage

- Path coverage
 - What portion of all possible paths through the program are covered by tests?
 - Loop testing: Consider representative and edge cases:
 - Zero, one, two iterations
 - If there is a bound n : $n-1$, n , $n+1$ iterations
 - Nested loops/conditionals from inside out
- Advantages
 - Better coverage of logical flows
- Disadvantages
 - Not all paths are possible, or necessary
 - What are the *significant* paths?
 - Combinatorial explosion in cases unless careful choices are made
 - E.g., sequence of n if tests can yield up to 2^n possible paths
 - Assumption that program structure is basically sound



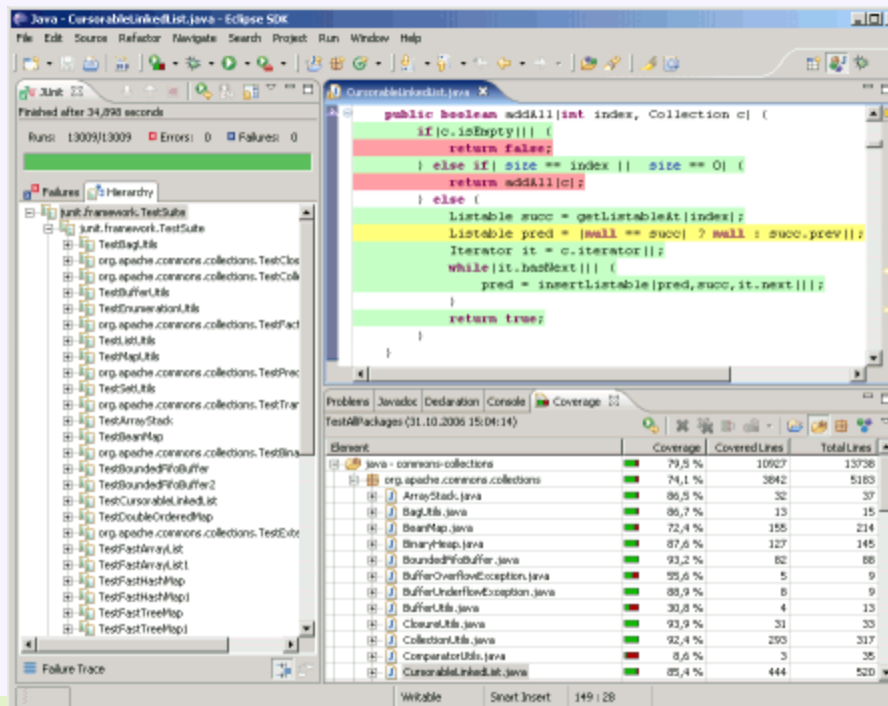
```
38  
39 public boolean equals(Object anObject) {  
40     if (isZero())  
41         if (anObject instanceof IMoney)  
42             return ((IMoney) anObject).isZero();  
43     if (anObject instanceof Money) {  
44         Money aMoney = (Money) anObject;  
45         return aMoney.currency().equals(currency())  
46             && amount() == aMoney.amount();  
47     }  
48     return false;  
49 }
```

```
int binarySearch(int[] a, int key) {
    int imin = 0;
    int imax = a.length-1;
    while (imax >= imin) {
        int imid = midpoint(imin, imax);
        if (a[imid] < key)
            imin = imid + 1;
        else if (a[imid] > key )
            imax = imid - 1;
        else
            return imid;
    }
    return -1;
}
```

Find test cases to maximize line, branch, and path coverage.

Test Coverage Tooling

- Coverage assessment tools
 - Track execution of code by test cases
- Count visits to statements
 - Develop reports with respect to specific coverage criteria
 - Instruction coverage, line coverage, branch coverage
- Example: EclEmma tool for JUnit tests



The screenshot shows the Eclipse IDE with the following components:

- Editor:** Displays the source code for `CursorableLinkedList.java`. The code is color-coded to show coverage: green for covered lines and yellow for uncovered lines. The code includes a `public boolean addAll(int index, Collection c)` method.
- Left Panel:** Shows the Project Hierarchy with a tree view of test classes under `org.apache.commons.collections`.
- Bottom Panel:** The Coverage tab is active, displaying a table of coverage data for the project.

Class	Coverage	Covered Lines	Total Lines
org.apache.commons.collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtil.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryTree.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	35,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtil.java	30,8 %	4	13
CloneUtil.java	93,9 %	31	33
CollectionUtil.java	92,4 %	293	317
ComparatorUtil.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

“Coverage” is useful but also dangerous

- Examples of what coverage analysis could miss
 - Missing code
 - Incorrect boundary values
 - Timing problems
 - Configuration issues
 - Data/memory corruption bugs
 - Usability problems
 - Customer requirements issues
- Coverage is not a good **adequacy** criterion
 - Instead, use to find places where testing is *inadequate*

Test coverage – Ideal and Real

- An Ideal Test Suite

- Uncovers all errors in code
- Uncovers all errors that requirements capture
 - All scenarios covered
 - Non-functional attributes: performance, code safety, security, etc.
- Minimum size and complexity
- Uncovers errors early in the process

- A Real Test Suite

- Uncovers some portion of errors in code
- Has errors of its own
- Assists in exploratory testing for validation
- Does not help very much with respect to non-functional attributes
- Includes many tests inserted after errors are repaired to ensure they won't reappear

Summary

- Unit testing is one of many testing approaches
- Unit testing to
 - discover bugs (not prove correctness)
 - document code
 - design testable code
- JUnit details (@Test, ...)
- Test coverage: The good, the bad, and the ugly
- You should be able to write unit tests for all your code now