



toad

Fall 2014



Principles of Software Construction: Objects, Design, and Concurrency

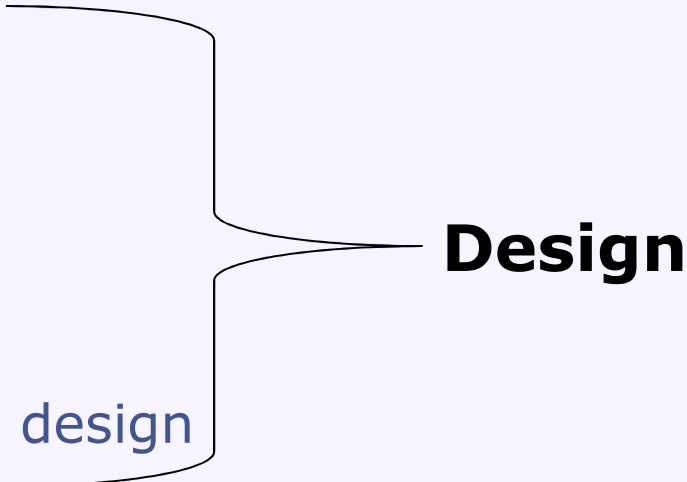
Specification, Testing, and Builds

Jonathan Aldrich Charlie Garrod

Administrivia

- Homework 0 due at 11:59pm tonight
- Document summarizing Java basics posted

Review: Steps in the Design Process

- Precondition: understand functional requirements
 - Pre- and post-condition specifications for IntSet
 - Precondition: understand quality attribute requirements
 - Design a logical architecture
 - Design a behavioral model
 - Responsibility assignment
 - Interface design
 - Algorithm and data structure design
 - Writing code
 - ...
- 
- Design**

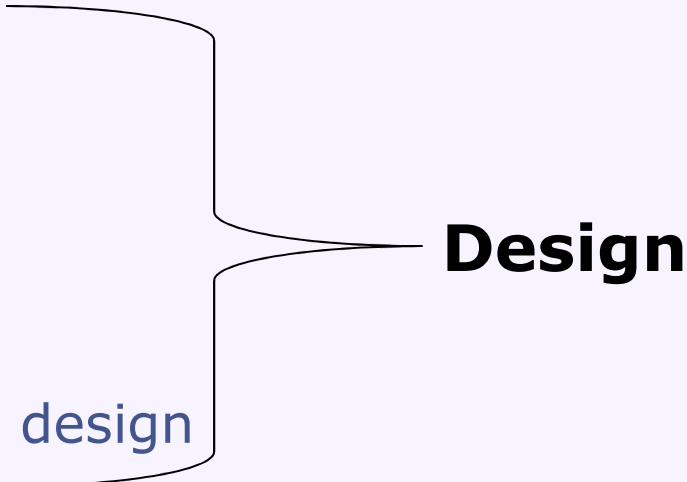
Review: In the Previous Lecture...

- We broke the Tree Simulation design process into steps
- We studied an example of the last step, detailed design
 - A data structure for mathematical sets
- We discussed a number of design principles
Can you name the benefits, and how achieved?
 - Representation hiding →
 - Object-oriented interfaces →
 - Abstract data types →

Review: In the Previous Lecture...

- We broke the Tree Simulation design process into steps
- We studied an example of the last step, detailed design
 - A data structure for mathematical sets
- We discussed a number of design principles
Can you name the benefits, and how achieved?
 - Representation hiding → ease of changing representation
 - Mechanisms: public vs. private, constructors
 - Object-oriented interfaces → interoperability of implementations
 - Use interface as type, especially for binary methods
 - Dispatch provides interoperability
 - Abstract data types → performance from shared representation
 - Using a class type instead of an interface type, esp. in binary methods
 - The type tells you the representation – can access private fields
 - Tradeoff: lose interoperability advantages

Steps in the Design Process

- Precondition: understand functional requirements
 - Pre- and post-condition specifications for IntSet
 - Precondition: understand quality attribute requirements
 - Design a logical architecture
 - Design a behavioral model
 - Responsibility assignment
 - Interface design
 - Algorithm and data structure design
 - Writing code
 - ...
- 
- Design**
- ← What goes here?**

Steps in the ~~Design~~ Development Process

- Precondition: understand functional requirements
 - Pre- and post-condition specifications for IntSet
- Precondition: understand quality attribute requirements
- Design a logical architecture
- Design a behavioral model
- Responsibility assignment
- Interface design
- Algorithm and data structure design
- Writing code
- Testing code ← **Note – possibly *before* writing code!**
 - Unit testing, JUnit, Coverage, EcJemma
- Automated builds and continuous integration
 - Ant and TravisCI

Design

This lecture



- 214: managing complexity, from programs to systems
 - **T**hreads and concurrency
 - **O**bject-oriented programming
 - **A**nalysis and modeling
 - **D**esign

Today's Lecture: Learning Goals



- Review principles for detailed design ✓
- Basic specification concepts (review from 15-122)
 - preconditions and postconditions
- Testing principles and practices
 - design effective unit test suites
 - write tests in JUnit
- Specification and code coverage
 - how they differ
 - what each is useful for
 - evaluate of code coverage using Eclemma
- Continuous integration
 - benefits in software development
 - use ant and TravisCI

This is a bug



A problem has been detected and windows has been shut down to prevent damage to your computer.

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

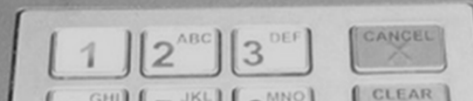
If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced startup options, and then select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0x800005F2, 0x00000000, 0x804E83C8, 0x00000000)

Beginning dump of physical memory
physical memory dump complete.

Contact your system administrator or technical support group for further assistance.



Is this a bug?

```
graph.getDistance(rachel, kramer);
```

```
> -1
```

Is this a bug? **NO!**

```
class Graph {  
    /** @return the distance between p1 and p2  
     * in the graph. Returns -1 if p1 and p2  
     * are unconnected. */  
    void getDistance(Person p1, Person p2);  
}
```

```
graph.getDistance(rachel, kramer);
```

```
> -1
```

We need ***specifications*** to determine whether or not code behaves correctly

Specifications

- Contain
 - Functional behavior
 - Erroneous behavior
 - Quality attributes
- Desirable attributes
 - Complete
 - Does not leave out any desired behavior
 - Minimal
 - Does not require anything that the user does not care about
 - Unambiguous
 - Fully specifies what the system should do in every case the user cares about
 - Consistent
 - Does not have internal contradictions
 - Testable
 - Feasible to objectively evaluate
 - Correct
 - Represents what the end-user(s) need

A Real Specification

```
/**
 * Returns the correctly rounded positive square root of a
 * double value.
 *
 * Special cases:
 * - If the argument is NaN or less than zero, then the
 *   result is NaN.
 * - If the argument is positive infinity, then the result
 *   is positive infinity.
 * - If the argument is positive zero or negative zero, then
 *   the result is the same as the argument.
 * Otherwise, the result is the double value closest to
 * the true mathematical square root of the argument value.
 *
 * @param a a value
 * @return the positive square root of a, or less than zero
 */
public static double sqrt(double a) { ...}
```

We need specifications to correctly
use code that we can't see, or don't
have time to study

Function Specifications

- A function's contract is a statement of the responsibilities of that function, and the responsibilities of the code that calls it.
 - Analogy: legal contracts
 - If you pay me \$30,000
 - I will build a new room on your house
 - Helps to pinpoint responsibility
- Contract structure
 - Precondition: condition the function relies on for correct operation
 - Postcondition: condition the function establishes after running
- (Functional) correctness with respect to the specification
 - If the client of a function fulfills the function's precondition, the function will execute to completion and when it terminates, the postcondition will be fulfilled
- What does the implementation have to fulfill if the client violates the precondition?
 - A: nothing at all!
 - In practice we often want to specify what happens on error inputs

Specifying IntSet (*in-class version*)

```
interface IntSet {  
    /** precondition:  
     * postcondition:  
     */  
    IntSet union(IntSet s);  
  
    /** precondition:  
     * postcondition:  
     */  
    boolean contains(int i);  
}
```


Specifying IntSet (*prepared version*)

interface IntSet {

*/** precondition: s is not null*

** postcondition: returns an IntSet that contains an*

** element i iff when this or s does*

**/*

IntSet union(IntSet s);

*/** precondition: none ("true" logically)*

** postcondition: returns true iff i is in the set*

**/*

boolean contains(**int** i);

}

Specifying IntSet (*on your own – one solution at the end*)

```
interface IntSet {  
    IntSet union(IntSet s);  
    boolean contains(int i);  
  
    /** precondition:  
     *   postcondition:  
     *  
     */  
    boolean isSubsetOf(IntSet s);  
}
```

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

Testing Decisions

- What to test?
 - Functional correctness of each method? System behavior? UI?
- Who tests?
 - Developers? QA team?
- When to test?
 - Before coding? After writing each method? Before shipping?
- Manual or automated?
 - Ability to test anything? Ability to repeat tests?
- When to stop testing?
 - When all functionality is tested? When all code is tested? When we run out of time or money?

Guidelines for Designing Test Suites

- Write a test for each case in the specification
 - Representative classes of input
 - Invalid classes of input
 - Write tests for boundary conditions
 - Off-by-one errors are common
 - Write tests for difficult situations
 - Stress tests (extreme input)
 - Situations that require complex reasoning
- Black-box testing
- Write tests that exercise all of the code
 - Perhaps interesting paths through the code, too
- White-box testing
(*glass-box a better term?*)

Example (exercise on paper)

```
/**
 * 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);
```

Guideline Reminder

- Write a test for each case in the specification
 - Representative classes of input
 - Invalid classes of input
- Write tests for boundary conditions
 - Off-by-one errors are common
- Write tests for difficult situations
 - Stress tests (extreme input)
 - Situations that require complex reasoning

Black box testing

Example (possible solution)

```
/**  
 * 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
- Test with a very long array

Black box testing

Exercise (*on your own*)

- Test a priority queue for Strings

```
public interface Queue {  
    void add(String s);  
    String getFirstAlphabetically();  
}
```

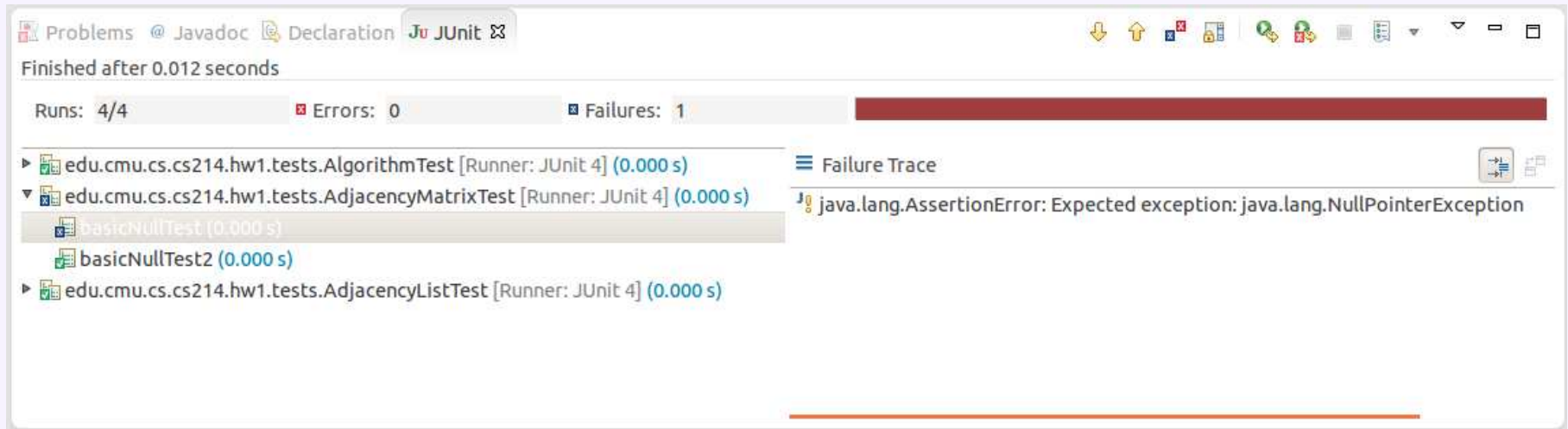
- Write various kinds of test cases

Unit Tests

- Unit tests for small units: functions, classes, subsystems
 - Smallest testable part of a system
 - Test parts before assembling them
 - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
 - Can run on every check-in, or every compile
- Little dependencies on other system parts or environment
- Insufficient but a good starting point, extra benefits:
 - Documentation (executable specification)
 - Design mechanism (design for testability)

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Integration into Eclipse, Ant, other tools
- Can be used to drive design
 - Testability, incrementally adding functionality



JUnit

```
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...
}
```

@Test annotation
signals a test case

Set up
tests

Check
expected
results

JUnit Demo

assert, Assert

- **assert** is a Java statement form that verifies a condition (if checking is turned on)
 - **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 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.

Common Setup

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

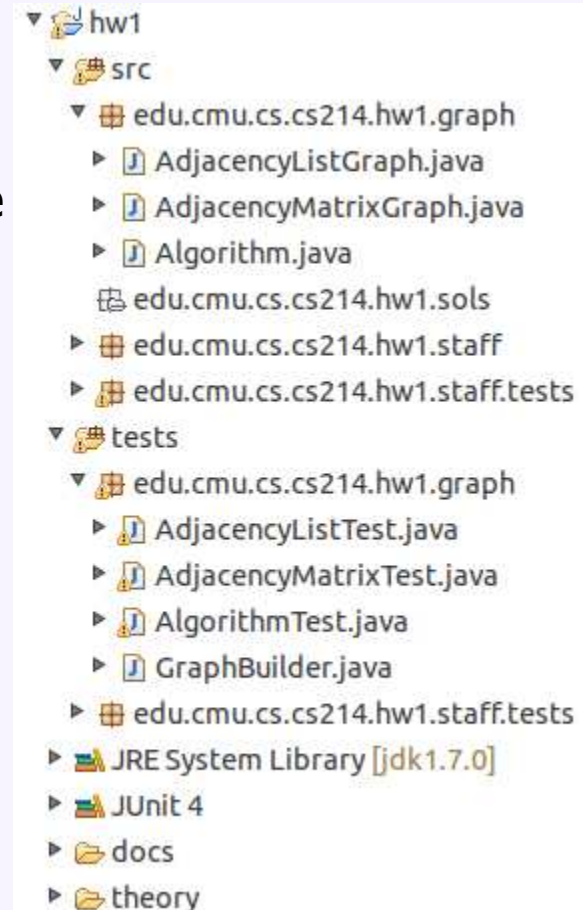
public class AdjacencyListTest {
    Graph g;

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

    @Test
    public void testSanityTest(){
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(3, g.getDistance(s1, s2));
    }
}
```

Test organization conventions

- A test class CTest for each class C
- Two directories: source and test
 - Store CTest and C in the same package
 - Tests can access members with default (package) visibility
- Alternative: (not in 15-214)
 - Store tests in the source directory but in a separate package



JUnit Operation

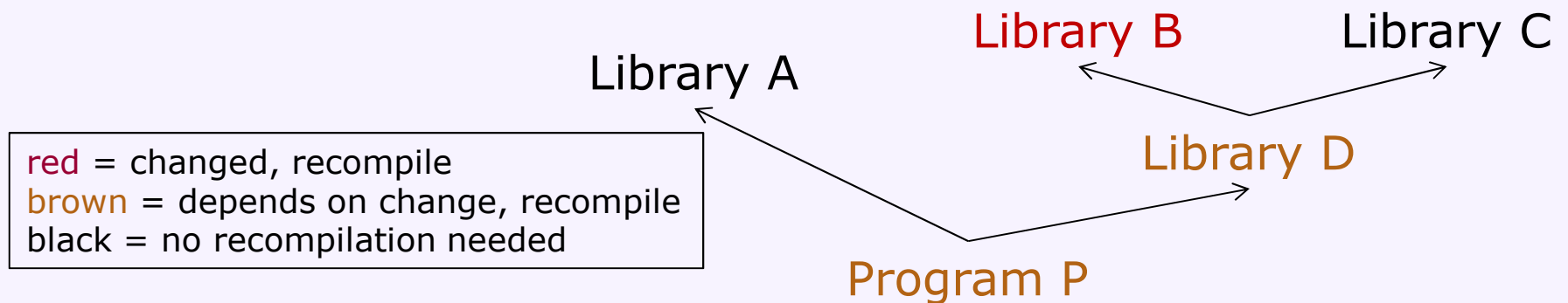
- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests are methods without parameters or return values
- Test runner knows how to run JUnit tests
 - (uses reflection to find all methods with @Test annotat.)

Run tests frequently

- Run tests before every commit
 - Committing broken code makes teammates unhappy
 - You will be unhappy too if they call your cell on your vacation
- Run tests before trying to understand unfamiliar code
 - If it's broken, get someone to fix it first
- What if the test suite takes too long to run?
 - Medium sized projects easily have 1000s of test cases and run for minutes
 - Run a subset ("smoke tests") on every commit
 - Run all tests nightly/weekly/whatever ("Nightly build")
- Build tools and continuous integration servers make this convenient

Build and Test Automation

- Build automation: automatically compile a program
 - May be a multi-step process with dependencies
 - Program A uses library B → compile B first
 - Avoid unnecessary recompilation



- Test automation: run tests automatically after a build
- Many tools
 - make
 - ant
 - gradle
 - maven
 - sbt
 - ...

Ant build file example

XML format

```
<project name="MyProject" default="dist" basedir=". ">  
  <property name="src" location="src"/>  
  <property name="build" location="build"/>
```

Constants
used later

```
  <target name="compile"  
    description="compile the source">  
    <javac srcdir="${src}" destdir="${build}"/>  
  </target>
```

compile
target runs
javac

```
  <target name="test" depends="compile"  
    description="run tests">  
    <junit printsummary="on" haltonfailure="yes">  
      ...  
    </junit>  
  </target>  
</project>
```

test target
depends on
compile,
runs javac

Ant demo

Continuous Integration

- Automation server responds on every commit
 - Compiles code
 - Runs tests
 - Reports errors (web page, email, etc.)
- Benefits
 - Immediate feedback about problems
 - Allows developers to make frequent check-ins
 - Keeps code synchronized
- Example tool: TravisCI

TravisCI Demo

Whitebox Testing: Code 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     public int hashCode() {
```

- Does this guarantee correctness?

Statement Coverage

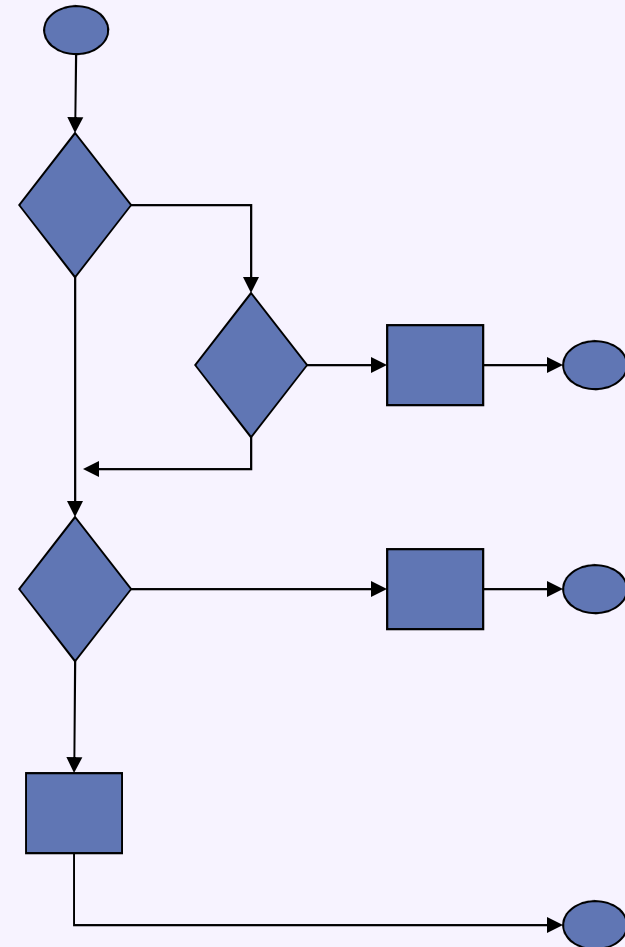
- Trying to execute **every statement** in at least one test

```
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     }
50     public int hashCode() {
```

- Does this guarantee correctness?

Structure of a Method Under 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 public int hashCode() {
```



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

Statement Coverage

- **Statement coverage**

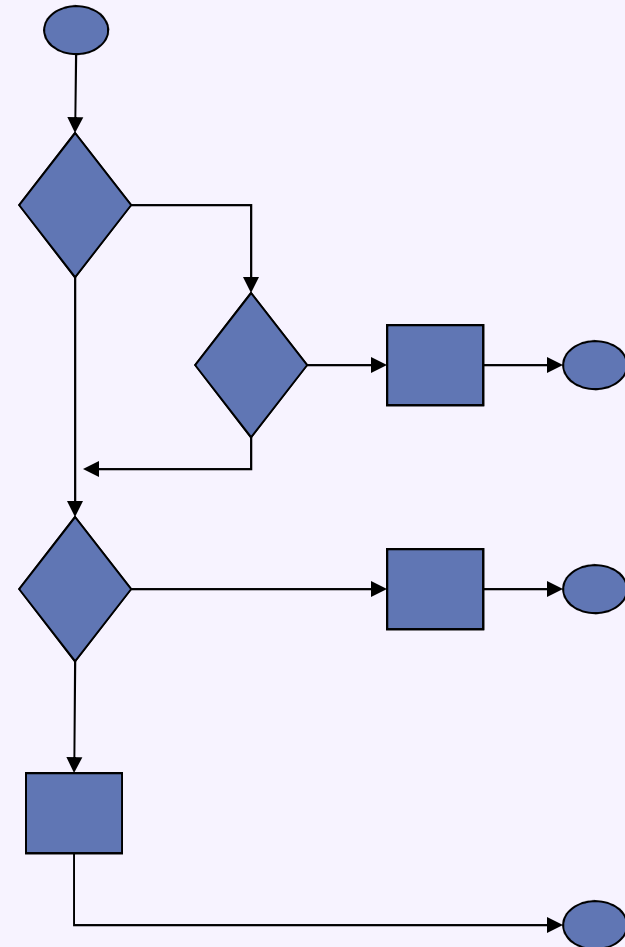
- What portion of program statements (nodes) are touched by test cases

- **Advantages**

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

- **Issues**

- May not exercise code in enough interesting situations



```
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 }
```

Branch Coverage

- **Branch coverage**

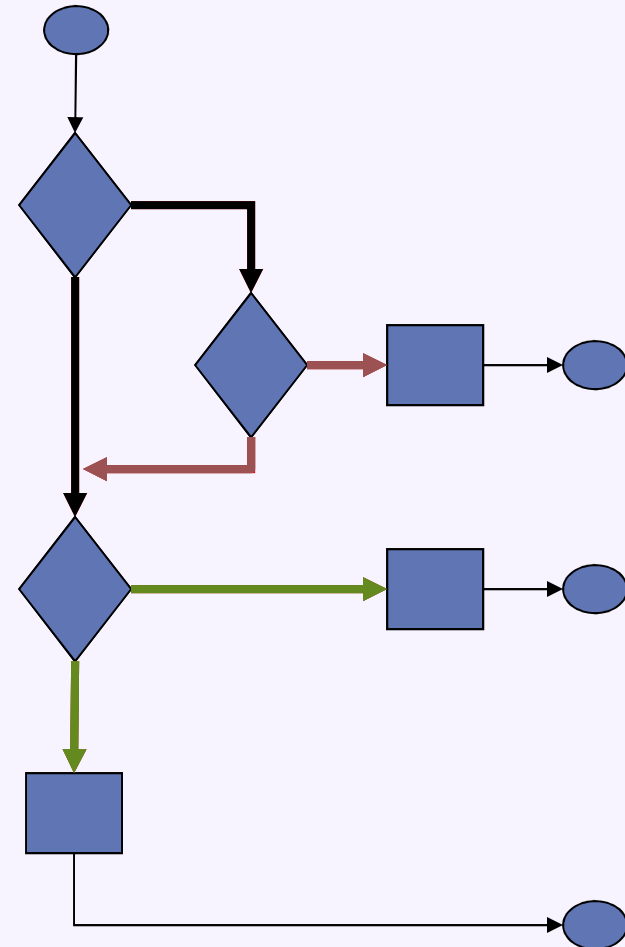
- What portion of **condition branches** are covered by test cases?
 - Consider true and false branches at each choice point

- **Advantages**

- Coverage easily assessed
- Test suite size and content derived from structure of control flow

- **Issues**

- More tests than statement coverage
- Still may not exercise enough interesting situations



```
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 }
```

toad

Path Coverage

- **Path coverage**

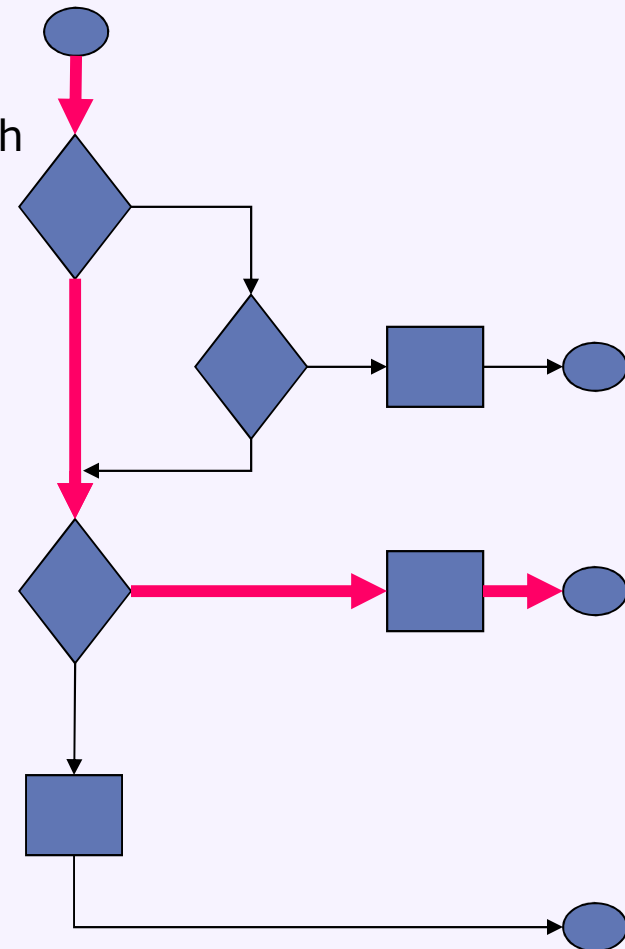
- What portion of all possible **paths** through the program are covered by tests?
 - For loops: must limit to a finite set of iterations (e.g. 0, 1, 2, N)

- **Advantages**

- Considers all logical combinations

- **Issues**

- Combinatorial explosion of paths
- Not necessarily worth the extra tests



```
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 }
```

toad

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: EcEmma tool for JUnit tests

The screenshot shows the Eclipse IDE interface. On the left, the 'JUnit' window displays test results for 'TestCursorableLinkedList', showing 'Runs: 13009/13009', 'Errors: 0', and 'Failures: 0'. Below this is a 'Failure Trace' section. The main editor window shows the source code of 'CursorableLinkedList.java'. The 'Coverage' window at the bottom right displays a table of coverage data for various packages and classes.

Package	Coverage	Covered Lines	Total Lines
java - 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
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtil.java	30,8 %	4	13
CloseUtil.java	93,9 %	31	33
CollectionUtil.java	92,4 %	293	317
ComparatorUtil.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

EclEmma Demo

Exercise (on your own)

- Write test cases to achieve 100% line coverage but **not** 100% branch coverage

```
void foo(int a, int b) {  
    if (a == b)  
        a = a * 2;  
    if (a + b > 10)  
        return a - b;  
    return a + b;  
}
```


“Coverage” is useful but also dangerous

- Examples of what coverage analysis could miss
 - Unusual paths
 - 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*

Toad's Take-Home Messages



- Specifications
 - Defined in terms of preconditions and postconditions
 - Necessary for determining correctness, and reusing code
- Testing
 - Finds bugs, but cannot prove their absence
 - Useful as documentation
 - JUnit is a practical tool
- Build and Test Automation
 - Runs tests often and automatically
 - Enables finding bugs more quickly
- Black box: coverage of the specification
 - The core of good testing practice – but must be done well
 - Representative, boundary, error, and extreme cases
- White box: coverage of code
 - Good for finding untested functionality
 - Not an indication that a test suite is adequate

Specifying IntSet (exercise solution)

```
interface IntSet {  
    IntSet union(IntSet s);  
    boolean contains(int i);  
  
    /** precondition: s is not null  
     *   postcondition: returns true iff for every integer i  
     *       such that this.contains(i), we have s.contains(i)  
     */  
    boolean isSubsetOf(IntSet s);  
}
```