Functional Correctness

- Specification

15-214

- Formal Verification
- Unit Testing
- Type Checking
- Statistic Analysis

15-313

- Requirements definition
- Inspections, Reviews
- Integration/System/Acceptance/Regression/GUI/Blackbox/ Model-Based/Random Testing
- Change/Release Management

Testing

- Executing the program with selected inputs in a controlled environment

- Goals:
  - Reveal bugs (main goal)
  - Asses 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)
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point, extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

From problem to idea to correct program

- “While the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.”
  — Donald E. Knuth, Stanford

- “Given ample time, only about 10% of professional programmers were able to get this small program right”
  — Jon Bentley, AT&T Bell Labs

Writing Test Cases: Common Strategies

- Read specification
- Write tests for representative case
  - Small instances are usually sufficient
- Write tests for invalid cases
- Write tests to check boundary conditions
- Are there difficult cases? (error guessing)
  - Stress tests? Complex algorithms?
- Think like a user, not like a programmer
  - The tester's goal is to find bugs!
- 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);
```

Black box testing

Example

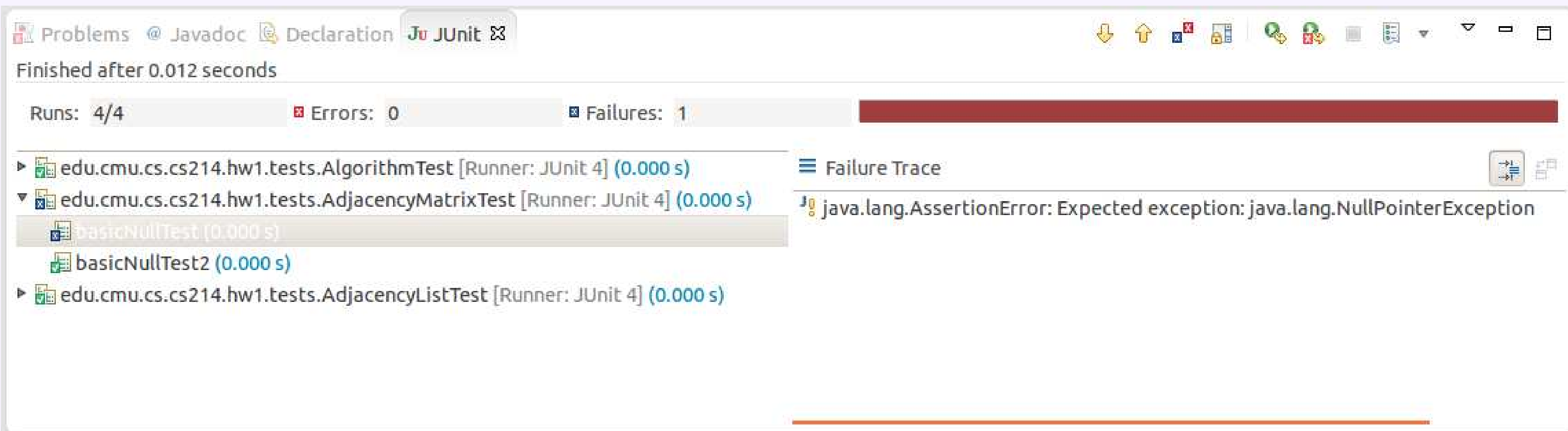
```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

- 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
- Test with a very long array

Black box testing

JUnit

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



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…
}
```

Set up tests

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 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
- AssertError signals failed test (unchecked exception)

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

Common Setup

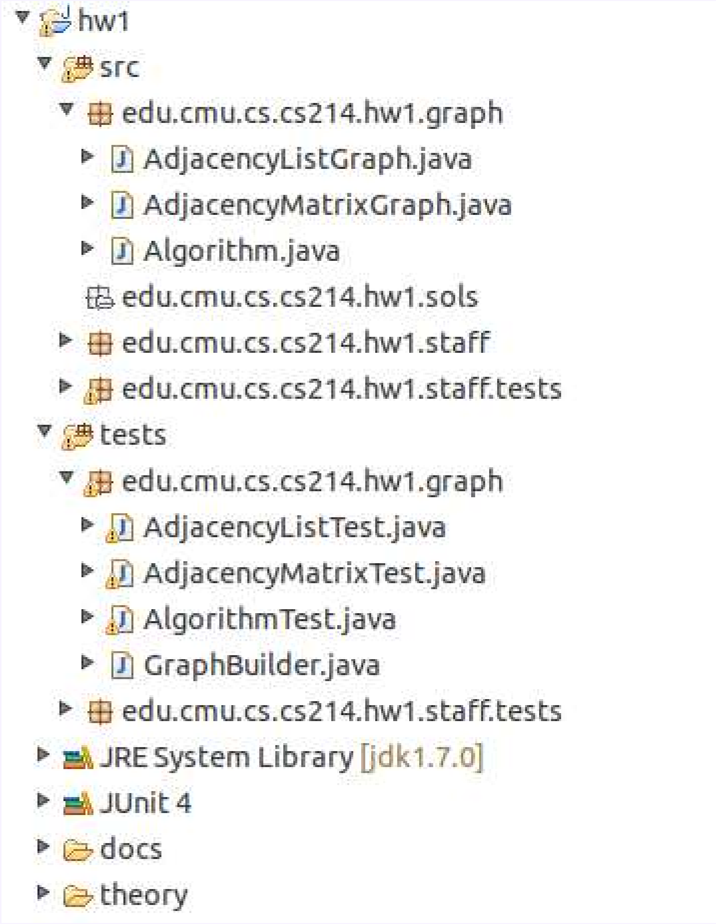
```
import org.junit.*;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

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));
      }
```

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 ATest for each class A
- Have a source directory and a test directory
  - Store ATest 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



Exercise (on paper!)

- Test a priority queue for Strings

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

- Write various kinds of test cases

JUnit Demo / Testing Practice

- Write some tests
- Write an invariant

Testing advice

Testable Code

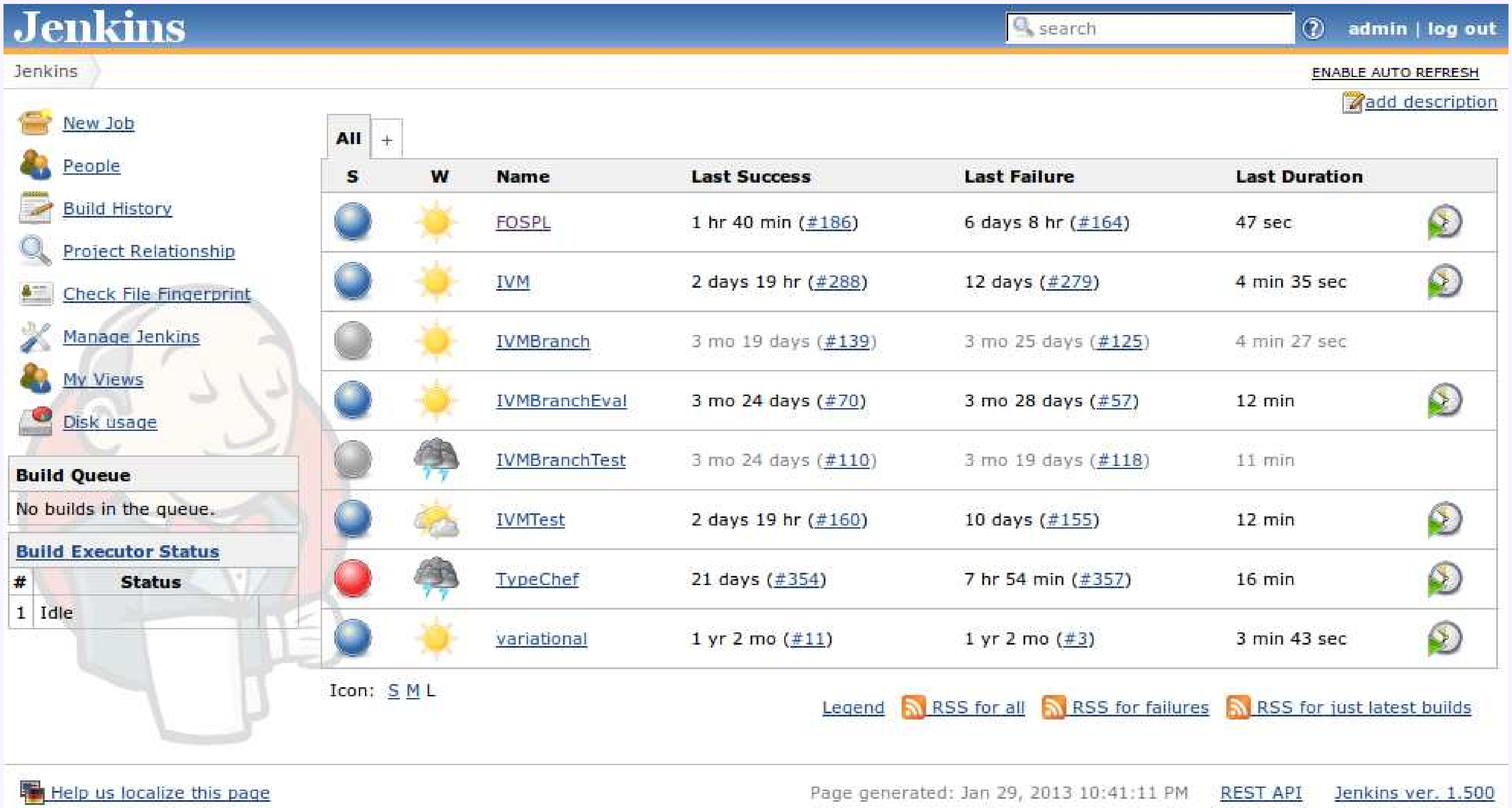
- Think about testing when writing code
- Unit testing encourages to write testable code
- Separate parts of the code to make them independently testable
- Abstract functionality behind interface, make it replaceable

- Test-Driven Development
  - A design and development method in which you write tests before you write the code!

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



See also travis-ci.org

Test Coverage

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?

Whitebox testing

Method Coverage

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

```java
    }
    public boolean equals(Object anObject) {
        if (isZero())
            if (anObject instanceof IMoney)
                return ((IMoney)anObject).isZero();
        if (anObject instanceof Money) {
            Money aMoney= (Money)anObject;
            return aMoney.currency().equals(currency())
                                && amount() == aMoney.amount();
        }
        return false;
    }
    public int hashCode() {
```

- Does this guarantee correctness?

Statement Coverage

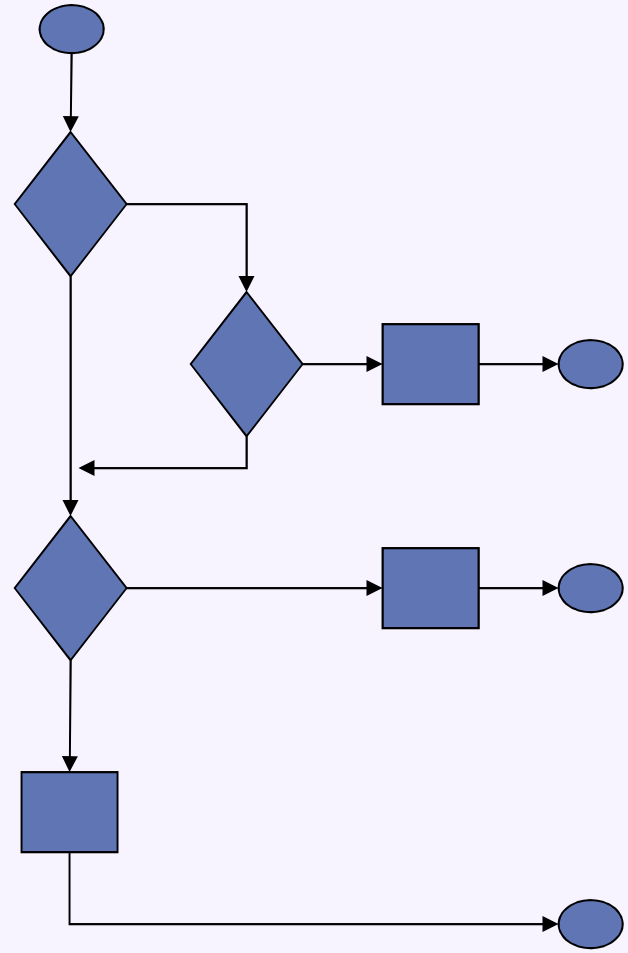
- Trying to test all parts of the implementation
- Execute every statement in at least one test

```java
    }
    public boolean equals(Object anObject) {
        if (isZero())
            if (anObject instanceof IMoney)
                return ((IMoney)anObject).isZero();
        if (anObject instanceof Money) {
            Money aMoney= (Money)anObject;
            return aMoney.currency().equals(currency())
                                && amount() == aMoney.amount();
        }
        return false;
    }
    public int hashCode() {
```

- Does this guarantee correctness?

Structure of Code Fragment to Test

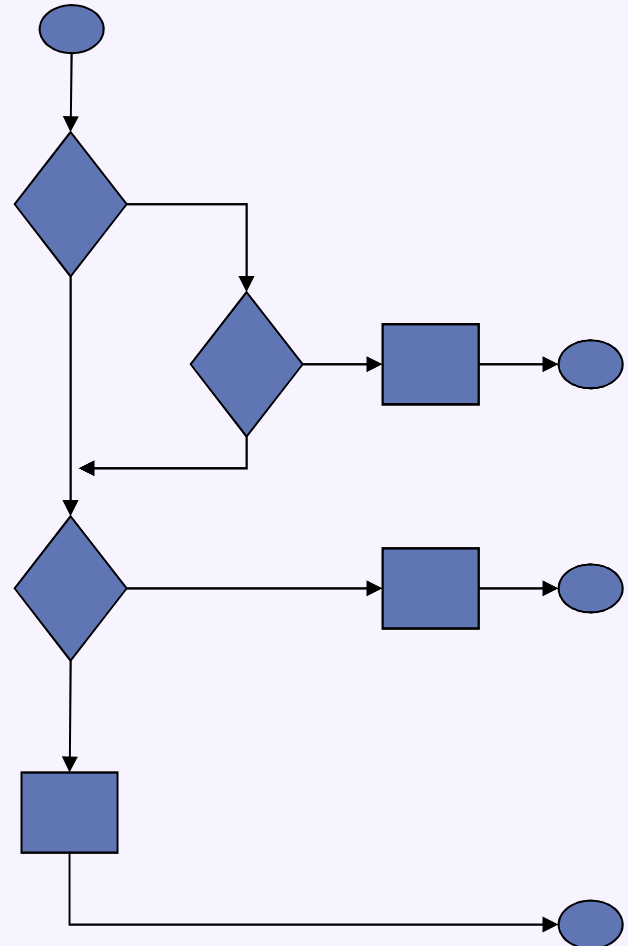
```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                    && amount() == aMoney.amount();
    }
    return false;
}
```

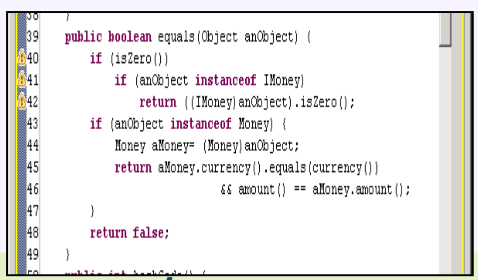


Flow chart diagram for junit.samples.money.Money.equals

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!

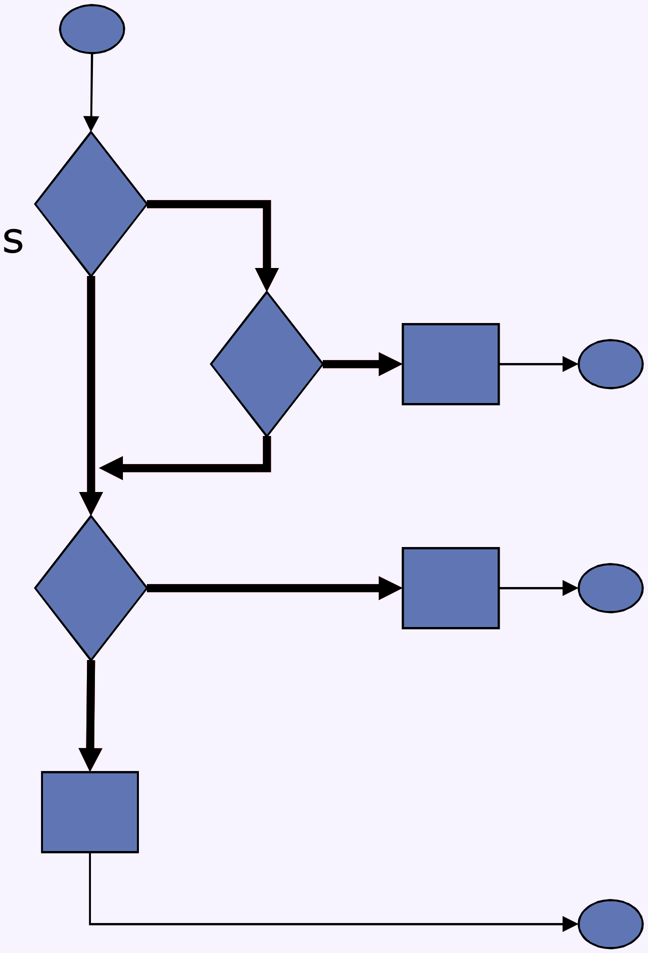




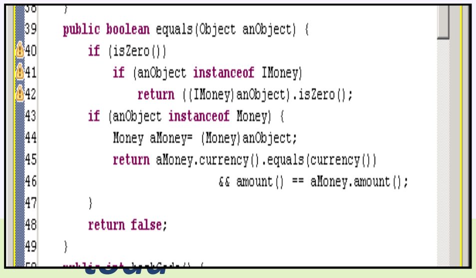
```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                        && amount() == aMoney.amount();
    }
    return false;
}
```

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



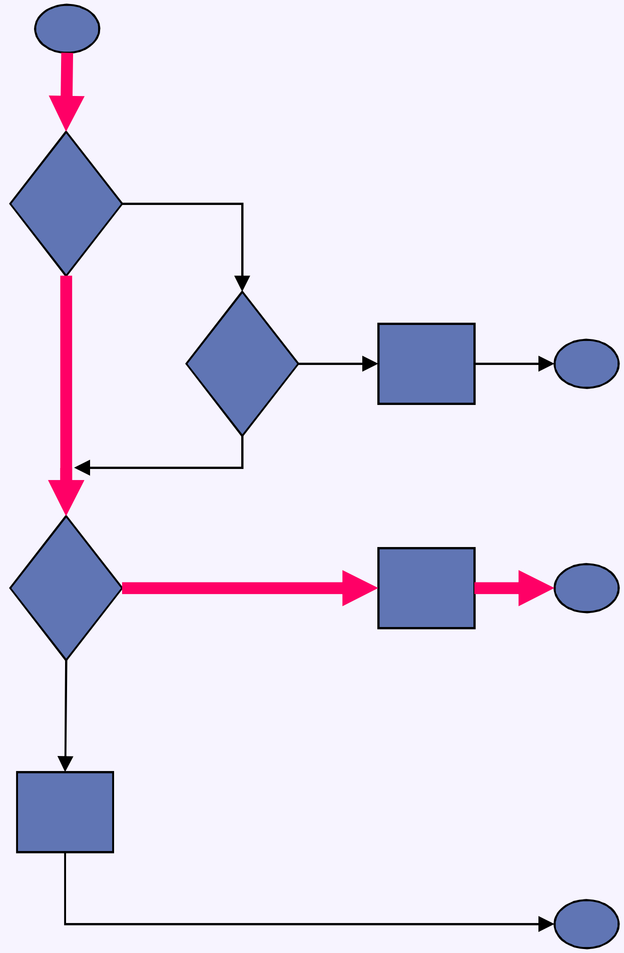
- Advantages
  - Test suite size and content derived from structure of boolean expressions
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - Fault-tolerant error-handling code may be difficult to "touch"

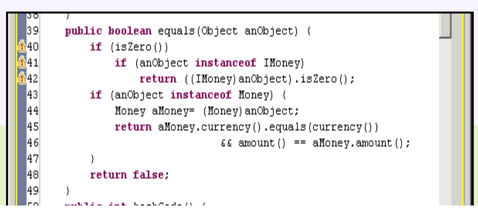


```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                    && amount() == aMoney.amount();
    }
    return false;
}
```

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





```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                    && amount() == aMoney.amount();
    }
    return false;
}
```

```
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.

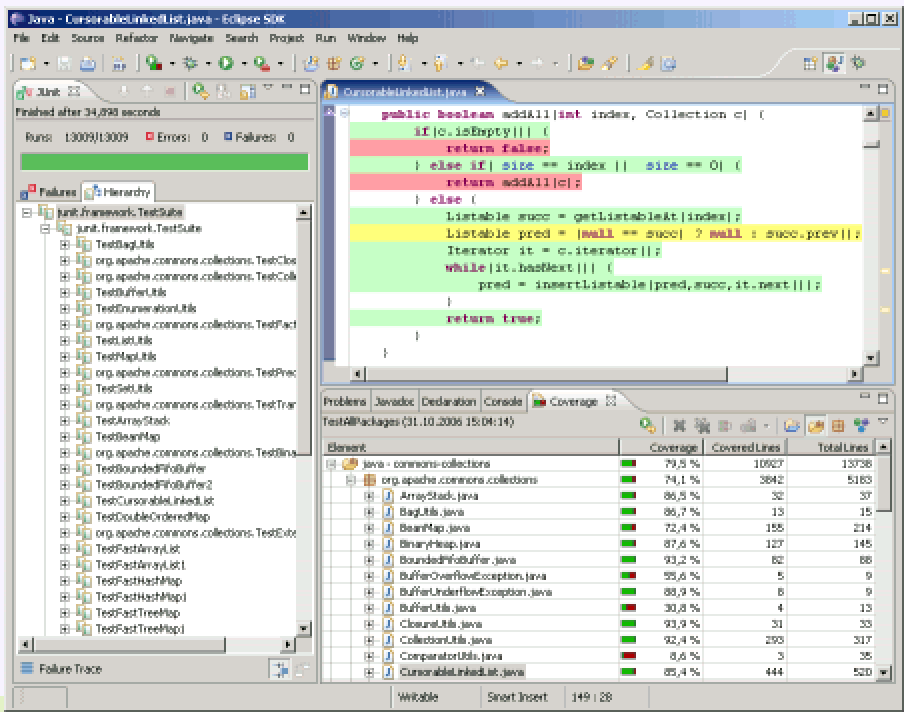
Write testable code

```
//700LOC
public boolean foo() {
   try {
      synchronized () {
         if () {
         } else {
         }
         for () {
            if () {
               if () {
                  if () {
                     if ()?
                     {
                        if () {
                           for () {
                           }
                        }
                     }
                  } else {
                     if () {
                        for () {
                           if () {
                           } else {
                           }
                           if () {
                           } else {
                              if () {
                              }
                           }
                           if () {
                              if () {
                                 if () {
                                    for () {
                                    }
                                 }
                              }
                           } else {
                           }
                        }
```

Source:
http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx

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



“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*

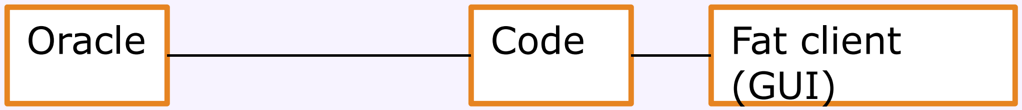
Test coverage – Ideal and Real

- 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

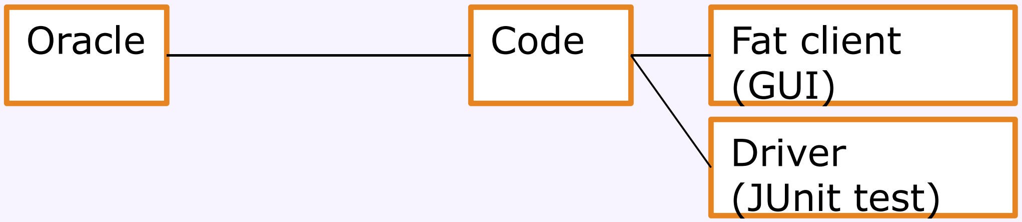
- Unit testing is one of many testing approaches
- Unit testing to
  - discover bugs (not prove correctness)
  - document code
  - design testable code
- JUnit details (@Test, …)
- Test coverage: The good, the bad, and the ugly
- You should be able to write unit tests for all your code now

Extra: Mock Objects



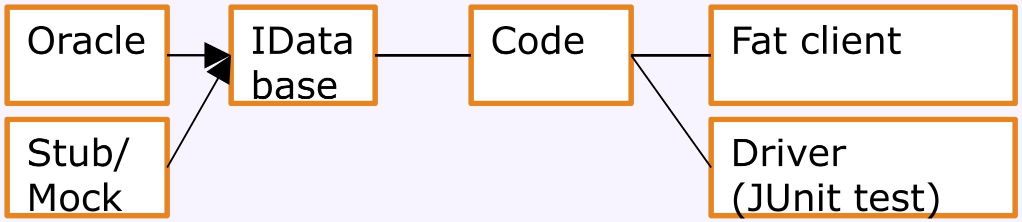

```
void buttonClicked() {
        render(getFriends());
}
Pair[] getFriends() {
        OracleDB database = oracle.getConnection();
        List<Node> persons = database.getTable("Persons");
        for (Node personA: persons) {
                for (Node personB: persons) {
                        ...
        }}
        return result;
}
```

Mock Objects




```
@Test void testGetFriends() {
        assert getFriends() == …;
}
Pair[] getFriends() {
        OracleDB database = oracle.getConnection();
        List<Node> persons = database.getTable("Persons");
        for (Node personA: persons) {
                for (Node personB: persons) {
                        …
        }}
        return result;
}
```

Mock Objects

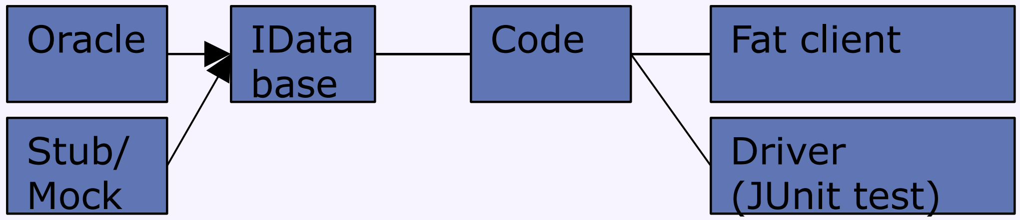



```
IDatabase database;
@Before void init() {database = new MockDatabase(); }
@Test void testGetFriends() {
        assert getFriends() == …;
}
Pair[] getFriends() {
        List<Nod
        for (Node
                f

        }}
        return re
}
```

```
class MockDatabase implements IDatabase {
        void open() {}
        List<Node> getTable(String n) {
                if ("Persons".equals(n)) {
                        List<Node> result=new List();
                        result.add(…);
                        return result;
                }
        }
```

Mock Objects

- Separate business logic and data representation from GUI for testing
- Test algorithms locally without large environment



Test Driven Development

Empirical Results – What works in practice?