6.005 Software Construction
Fall 2011
Prof. Rob Miller

L2: Test-First Programming

Today

- Testing
- Choosing test cases
- Blackbox vs. whitebox testing
- Coverage
- Stubs, drivers, oracles
- Regression testing

Real Programmers Don't Test (?)

Here are the top-5 reasons why Louis Reasoner doesn't want to test his code:

5) I want to get this done fast – testing is going to slow me down.

4) I started programming when I was 2. Don't insult me by testing my perfect code!

3) Testing is for incompetent programmers who cannot hack.

2) We're not Harvard students – our code actually works!

1) "Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them." – an excerpt from a 6.170 student's e-mail

The biggest problem for Louis is optimism. Change that perspective – look for things that can go wrong, rather than assuming Pollyannishly that all will go right.

Testing isn't rocket science. Except when it is: a famous case was the Ariane 5 launch vehicle, designed and built for the European Space Agency in the 1990s. It self-destructed 37 seconds after its first launch. The reason was a control software bug that went undetected. Ariane 5's guidance software was reused from the Ariane 4, which was a slower rocket. As a result, when the velocity calculation converted from a 64-bit floating point number (a double in Java terminology, though this software wasn't written in Java) to a 16-bit signed integer (a short), it overflowed the small integer and caused an exception to be thrown. The exception handler had been disabled for efficiency reasons, so the guidance software crashed... and without guidance, the rocket did too. The cost of the failure was $1 billion... a cost that might have been prevented with better testing.

In PS1, we have a sequence of steps feeding each other; a stack of methods building on each other. The optimistic hacker's approach to PS1 might build the whole thing before testing any part of it. Contrast with the engineer's approach: isolate each component, build it and test it soundly in isolation, then plug together.

Basic Principles

Testing is only one part of a more general process called *validation*. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness. Validation includes:

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors.

- Formal reasoning about a program, usually called *verification*. Verification constructs a formal proof that a program is correct, by showing that whenever the preconditions are satisfied, the program always produces a state in which the postconditions are true. Verification is tedious to do by hand, and automated tool support for verification is still an active area of research. Nevertheless, small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system or the bytecode interpreter in a virtual machine.
- Informal reasoning. Having somebody else carefully read your code can be a good way to uncover bugs, much like having somebody else proofread an essay you have written. In industry, this practice goes by various names (with various degrees of formality): code reviews, code inspection, walkthroughs. *Pair programming* is an extreme form of this idea, where two programmers work together on a single computer, with one programmer typing and the other reading and thinking about the code being typed. You can also read your own code, although it is harder to catch your own mistakes.
- Testing — running the program on carefully selected inputs and checking the results.

Validation requires having the right attitude. Your goal is not to see the program work, but to make it fail. There's a subtle difference. It is all too tempting to treat code you've just written as a precious thing, a fragile eggshell, and test it very lightly just to see it work. You have to be brutal. A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable, so that those vulnerabilities can be eliminated.

There are three basic principles of testing:

1. **Be systematic.** Haphazard testing is less likely to find bugs (unless the program is so buggy that a randomly- chosen input is more likely to fail than to succeed), and it doesn't increase our confidence in program correctness. On the other hand, exhaustive testing — running the program on all possible inputs — is usually impossible. Instead, test cases must be chosen carefully and systematically. Some approaches to choosing test cases are discussed below.

2. **Do it early and often.** Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it. The technique we use in this class, *test-first programming*, carries this idea to its logical conclusion: you write tests before you even write any code. In other words, the development of a single module might proceed in this order:

(a) write a specification for the module;

(b) write tests that exercise the specification;

(c) write the actual code.

3. **Automate it.** Nothing makes tests easier to run, and more likely to be run, than complete automation. A test driver should not be an interactive program that prompts you for inputs and prints out results for you to manually check. Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct. The result of the test driver should be either "all tests OK" or "these tests failed: ..." A good testing framework, like JUnit, helps you build automated test suites. You can find links to more information about JUnit on the course web page.

Why Testing is Hard

we want to

- know when product is stable enough to launch
- deliver product with known failure rate (preferably low)
- offer warranty?

but

- it's very hard to measure or ensure quality in software
- residual defect rate after shipping:
    - 1 - 10 defects/kloc (typical)
    - 0.1 - 1 defects/kloc (high quality: Java libraries?)
    - 0.01 - 0.1 defects/kloc (very best: Praxis, NASA)

  example: 1Mloc with 1 defect/kloc means you missed 1000 bugs!

exhaustive testing is infeasible

- space is generally too big to cover exhaustively
- imagine exhaustively testing a 32-bit floating-point multiply operation, a*b
    - there are 2^64 test cases!

statistical testing doesn't work for software

- other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer defect rate for whole lot
- many tricks to speed up time (e.g. opening a refrigerator 1000 times in 24 hours instead of 10 years)
- gives known failure rates (e.g. mean lifetime of a hard drive)
- but assumes continuity or uniformity across the space of defects, which is true for physical artifacts
- **this is not true** for software
    - overflow bugs (like Ariane 5) happen abruptly
    - Pentium division bug affected approximately 1 in 9 billion divisions

often confused, but very different

(a) problem of **finding** bugs in defective code

(b) problem of showing **absence** of bugs in good code

approaches

- testing: good for (a), occasionally (b)
- reasoning: good for (a), also (b)

theory and practice

- for both, you need grasp of basic theory
- good engineering judgment essential too

what are we trying to do?

- find bugs as cheaply and quickly as possible

reality vs. ideal

- ideally, choose one test case that exposes a bug and run it
- in practice, have to run many test cases that "fail" (because they don't expose any bugs)

in practice, conflicting desiderata

- increase chance of finding bug
- decrease cost of test suite (cost to generate, cost to run)

design testing strategy carefully

- know what it's good for (finding egregious bugs) and not good for (security)
- complement with other methods: code review, reasoning, static analysis

- exploit automation (e.g. JUnit) to increase coverage and frequency of testing
- do it early and often: **test-first programming**

Basic Notions

what's being tested?

- unit testing: individual module (method, class, interface)
- subsystem testing: entire subsystems
- integration, system, acceptance testing: whole system

how are inputs chosen?

- random: surprisingly effective (in defects found per test case), but not much use when most inputs are invalid (e.g. URLs)
- systematic: partitioning large input space into a few representatives
- arbitrary: *not* a good idea, and not the same as random!

how are outputs checked?

- automatic checking is preferable, but sometimes hard (how to check the display of a graphical user interface?)

how good is the test suite?

- coverage: how much of the specification or code is exercised by tests?

when is testing done?

- test-first programming: tests are written first, before the code
- regression testing: a new test is added for every discovered bug, and tests are run after every change to the code

essential characteristics of tests

- modularity: no dependence of test driver on internals of unit being tested
- automation: must be able to run (and check results) without manual effort

Choosing Test Cases

Arbitrary testing is not convincing – "just try it and see if it works" won't fly as a convincing validation. Exhaustive testing is clearly infeasible — even a simple int → int function requires billions of runs to test all inputs. Haphazard or random testing, on the other hand, is less likely to discover bugs. We want to pick a set of test cases that is small enough to run quickly, yet large enough to validate the program.

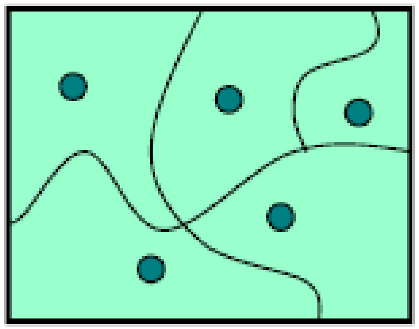
To do this, we divide the input space into *subdomains*, each consisting of a set of inputs. The subdomains completely cover the input space, so that every input lies in at least one subdomain. Then we choose one test case from each subdomain.

The idea behind subdomains is to partition the input space into sets of similar inputs. Then we use one representative of each set. This approach makes the best use of limited testing resources by choosing dissimilar test cases, and forcing the testing to explore parts of the input space that random testing might not reach.

key problem: choosing a test suite systematically

- small enough to run quickly
- large enough to validate the program convincingly

Key Idea #1: Partition the Input Space



input space is very large, but program is small

- so behavior must be the "same" for whole sets of inputs

ideal test suite

- identify sets of inputs with the same behavior
- try one input from each set

multiply : BigInteger × BigInteger → BigInteger

- partition BigInteger into:

  BigNeg, SmallNeg, -1, 0, 1, SmallPos, BigPos

- pick a value from each class

  -2^{65}, -9 -1, 0, 1, 9, 2^{65}

- test the 7 × 7 = 49 combinations

max : int × int → int

- partition into:

  a < b, a = b, a > b

- pick value from each class

  (1, 2), (1, 1), (2, 1)

intersect : Set × Set → Set

- partition Set into:

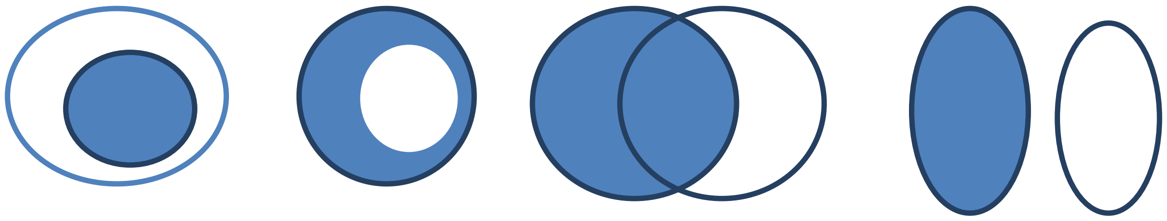
  ∅, singleton, many

- partition whole input space into:

  this = that, this ⊆ that, this ⊇ that, this ∩ that ≠ ∅, this ∩ that = ∅

- pick values that cover both partitions

{},{}	{},{2}	{},{2,3,4}
{5},{}	{5},{2}	{4},{2,3,4}
{2,3},{}	{2,3},{2}	{1,2},{2,3}



Key idea #2: Boundary testing

- include classes at **boundaries** of the input space
  - zero, min/max values, empty set, empty string, null
- why? because bugs often occur at boundaries
  - off-by-one errors
  - forget to handle empty container
  - overflow errors in arithmetic

Blackbox vs. whitebox

black box testing

- choosing test data only from spec, without looking at implementation

glass box (white box) testing

- choosing test data with knowledge of implementation
  - e.g. if implementation does caching, then should test repeated inputs
  - if implementation selects different algorithms depending on the input, should choose inputs that exercise all the algorithms
- must take care that tests don't *depend* on implementation details
  - e.g. if spec says "throws exception if the input is poorly formatted", your test shouldn't check specifically for a NullPtrException just because that's what the current implementation does
- good tests should be **modular** -- depending only on the spec, not on the implementation
  - **max : List<int> → int**

```
/**
 * Find the largest element in a list.
 * @param l list of elements.  Requires l to be nonempty
 * and all elements to be nonnegative.
 * @return the largest element in l
 */
public static int max(List<Integer> l) { ... }
```

- list length: ~~length 0~~, length 1, length 2+
- max position: start, middle, end of list
- value of max: ~~MIN_INT~~, ~~negative~~, 0, positive, MAX_INT

Coverage

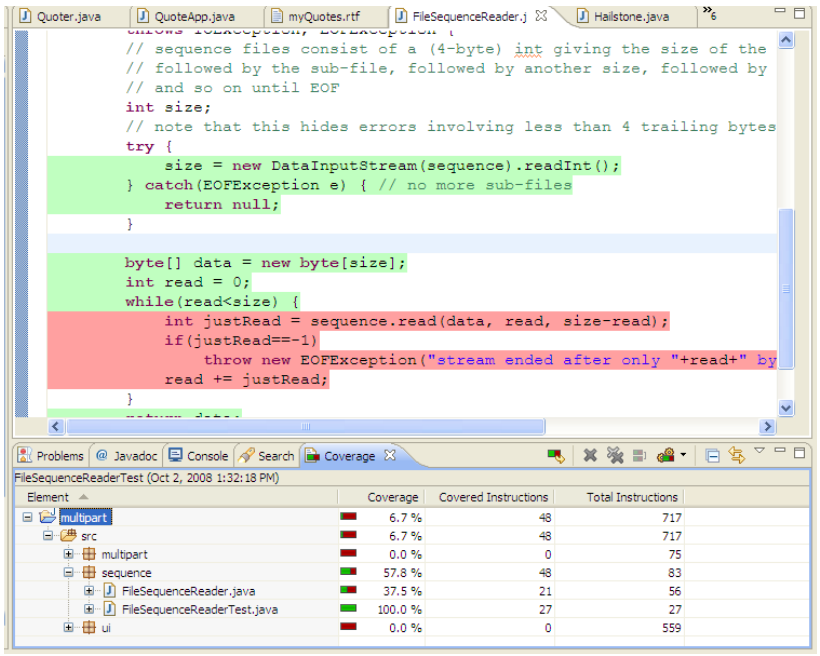
Three kinds of coverage:

- all-statements: is every statement run by some test case?
- all-branches: if every direction of an if or while statement (true or false) taken by some test case?
- all-paths: is every possible combination of branches – every path through the program – taken by some test case?

A standard approach to testing is to add tests until the test suite achieves adequate statement coverage: so that every statement in the program is executed by at least one test case.

In practice, statement coverage is usually measured by a *code coverage tool*, which instruments your program to count the number of times each statement is run by your test suite. With such a tool, glass box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed. (Note that you can't achieve 100% statement coverage when your program includes defensive code, like always-false assertions or exception handling, that is included for safety but should never be executed.)

a typical code coverage tool (EclEmma plugin for Eclipse):



```
        // sequence files consist of a (4-byte) int giving the size of the
        // followed by the sub-file, followed by another size, followed by
        // and so on until EOF
        int size;
        // note that this hides errors involving less than 4 trailing bytes
        try {
            size = new DataInputStream(sequence).readInt();
        } catch(EOFException e) { // no more sub-files
            return null;
        }

        byte[] data = new byte[size];
        int read = 0;
        while(read<size) {
            int justRead = sequence.read(data, read, size-read);
            if(justRead==-1)
                throw new EOFException("stream ended after only "+read+" by
            read += justRead;
        }
```

FileSequenceReaderTest (Oct 2, 2008 1:32:18 PM)

Element	Coverage	Covered Instructions	Total Instructions
multipart	6.7 %	48	717
src	6.7 %	48	717
multipart	0.0 %	0	75
sequence	57.8 %	48	83
FileSequenceReader.java	37.5 %	21	56
FileSequenceReaderTest.java	100.0 %	27	27
ui	0.0 %	0	559

Courtesy of The Eclipse Foundation. Used with permission.

industry practice

- all-statements is common goal, rarely achieved (due to unreachable code)
- all branches if possible: safety critical industry has more arduous criteria (eg, "MCDC", modified decision/condition coverage)
- all-paths is infeasible

Test frameworks

driver

- just runs the tests
- must design unit to be drivable!
- eg: program with GUI should have API

stub

- replaces other system components
- allows reproducible behaviours (esp. failures)

oracle

- determines if result meets spec
- preferably automatic and fast
- varieties: computable predicate (e.g. is the result odd?), comparison with literal (e.g. must be 5), manual examination (by a human)
- in regression testing, can use previous results as "gold standard"

Test-first programming

write tests before coding

- specifically, for every method or class:

  1) write specification

  2) write test cases that cover the spec

  3) implement the method or class

  4) once the tests pass (and code coverage is sufficient), you're done

writing tests first is a good way to understand the spec

- think about partitioning and boundary cases
- if the spec is confusing, write more tests
- spec can be buggy too
  - incorrect, incomplete, ambiguous, missing corner cases
  - trying to write tests can uncover these problems

Regression Testing

It's very important to rerun your tests when you modify your code. This prevents your program from regressing — introducing other bugs when you fix new bugs or add new features. Running all your tests after every change is called *regression testing*.

whenever you find and fix a bug

- store the input that elicited the bug
- store the correct output
- add it to your test suite

why regression tests help

- helps to populate test suite with good test cases
  - remember that a test is good if it elicits a bug – and every regression test did in one version of your code
- protects against reversions that reintroduce bug
- the bug may be an easy error to make (since it happened once already)

test-first debugging

- when a bug arises, immediately write a test case for it that elicits it
- once you find and fix the bug, the test case will pass, and you'll be done

How to avoid debugging

first defense against bugs is to make them impossible

- Java makes buffer overflow bugs impossible
- static typing eliminates many runtime type errors
- immutable objects like Strings and URLs can be passed around and shared without fear that they will be modified
- immutable (final) references guarantee that you won't

if we can't prevent bugs, we can try to localize them to a small part of the program

- fail fast: the earlier a problem is observed, the easier it is to fix
- assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation

  in Java: **assert *boolean-expression***

  note that you must enable assertions with -ea
- unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (or in the test driver)
- regression testing: run tests as often as possible when changing code.

  if a test fails, the bug is probably in the code you just changed

when localized to a single method or small module, bugs may be found simply by studying the program text

Code review

other eyes looking at the code can find bugs

code review

careful, systematic study of source code by others (not original author)

analogous to proofreading an English paper

look for bugs, poor style, design problems, etc.

formal inspection: several people read code separately, then meet to discuss it

lightweight methods: over-the-shoulder walkthrough, or by email

many dev groups require a code review before commit

code review complements other techniques

code reviews can find many bugs cheaply

also test the understandability and maintainability of the code

three proven techniques for reducing bugs: static checking, code reviews, testing

An example of test-first programming

First, the spec:

```
    /**
     * Find the first occurrence of x in sorted array a.
     * @param x value to find
     * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-
1])
     * @return lowest i such that a[i]==x, or -1 if x not found in a.
     */
    public static int find(int x, int[] a) {...}
```

Then, the test cases:

// Testing strategy for i = search(x, a):
//   partition the space of (x, a, i) as follows

// x: neg, 0, pos

// a.length: 0, 1, 2+

// a.vals: neg, 0, pos; all same, increasing;

// i: 0, middle, n-1, -1

- // x=2, a=[-1, 1, 3], i=-1
- // x=0, a=[0], i=0
- // x=1, a=[-1, 1, 3], i=1
- // x=-1, a=[-1, 1, 3], i=0
- // x=3, a=[-1, 1, 3], i=n-1
- // x=2, a=[-1, 1, 3], i=-1
- // x=1, a=[1, 1, 1], i=0

Then, a simple implementation: (tests the tests! also gives us a slow oracle in case we need it later)

```
    /**
     * Find the first occurrence of x in sorted array a.
     * @param x value to find
     * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-
1])
     * @return lowest i such that a[i]==x, or -1 if x not found in a.
     */
    public static int find(int x, int[] a) {
        for (int i = 0; i < a.length; ++i) {
            if (x == a[i]) {
                return i;
            }
        }
        return -1;
    }
```

Now, some attempts at the real binary-search implementation. Let's do it recursively:

```
    public static int search(int x, int[] a) {
        int mid = a.length/2;
        if (x < a[mid]) {
            binarySearch(x, left half of a)
        } else if (x > a[mid]) {
            binarySearch(x, right half of a)
        } else {
            return mid; // because x == a[mid], i.e. we found it!
        }
    }
```

OK, we need to strengthen the induction hypothesis to avoid copying the array:

```
    /*
     * Find the first occurrence of x in sorted array a[first..last].
     * @param x value to find
     * @param a array sorted in increasing order
     *          (a[0] <= a[1] <= ... <= a[n-1])
     * @param first first index of range.
     *              Requires 0 <= first <= a.length.
     * @param last last index of range.
     *              Requires 0 <= last <= a.length, and first <= last.
     * @return lowest i such that first<=i<=last and a[i]==x,
     *         or -1 if there's no such i.
     */
    private static int binarySearchInRange(int x, int[] a, int first, int last)
{
```

Now:

```
    public static int find(int x, int[] a) {
        return binarySearchInRange(x, a, 0, a.length);
    }

    private static int binarySearchInRange(int x, int[] a, int first, int last)
{
       int mid = (first + last) / 2;
        if (x < a[mid]) {
            return binarySearchInRange(x, a, first, mid);
        } else if (x > a[mid]) {
            return binarySearchInRange(x, a, mid+1, last);
        } else {
            // x == a[mid]... we found it!
            return mid;
        }
    }
```

Broken! Let's adjust the spec of our private method:

```
    /*
     * Find the first occurrence of x in sorted array a[first..max-1].
     * ...
     * @param first first index of range.
     *              Requires 0 <= first <= a.length.
     * @param max one beyond the last index of range.
     *              Requires 0 <= max <= a.length, and first <= max.
     * @return lowest i such that first<=i<max and a[i]==x,
     *         or -1 if there's no such i.
     */
    private static int binarySearchInRange(int x, int[] a, int first, int max)
{...}
    public static int find(int x, int[] a) {
        return binarySearchInRange(x, a, 0, a.length-1);
    }
```

And now add a test for an empty range:

```
    private static int binarySearchInRange(int x, int[] a, int first, int max)
{
        if (first >= max) {
            return -1; // range has dwindled to nothingness
        }

       int mid = (first + max) / 2;
        if (x < a[mid]) {
            return binarySearchInRange(x, a, first, mid);
        } else if (x > a[mid]) {
            return binarySearchInRange(x, a, mid+1, max);
        } else {
            // x == a[mid]... we found it!
            return mid;
        }
```

Still broken! Not handling the case of multiple matches correctly. Let's fix:

```
        } else {
            // x == a[mid]... we found it, but check if it's the first
            if (x == a[mid-1]) {
                // not the first! search lower half
                return binarySearchInRange(x, a, first, mid);
            } else {
                return mid; // it's the first
            }
        }
```

Still broken! In fact, we've experienced a *regression* – tests that used to be passing are now failing because of our change. Regressions happen! Run all your test cases after every change you make!

Need to be careful about a[mid-1], because mid might be 0. Finally:

```
    } else {
            // x == a[mid]... we found it, but check if it's the first
            if (mid > 0 && x == a[mid-1]) {
                // not the first! search lower half
                return binarySearchInRange(x, a, first, mid);
            } else {
                return mid; // it's the first
            }
    }
```

Here's the final code. Left as an exercise to the reader to tidy up binarySearchInRange() and make it simpler and more compact. The fact that you have a working test suite makes this tidying far safer. Without a test suite, attempts to simplify logic can easily introduce bugs. Just run it after every little change you make.

```
    public static int search(int x, int[] a) {
        return binarySearchInRange(x, a, 0, a.length);
    }
    private static int binarySearchInRange(int x, int[] a, int first, int max)
{
        if (first >= max) {
            return -1; // range has dwindled to nothingness
        }

        int mid = (first + max) / 2;
        if (x < a[mid]) {
            return binarySearchInRange(x, a, first, mid);
        } else if (x > a[mid]) {
            return binarySearchInRange(x, a, mid+1, max);
        } else {
            // x == a[mid]... we found it, but check if it's the first
            if (mid > 0 && x == a[mid-1]) {
                // not the first! search lower half
                return binarySearchInRange(x, a, first, mid);
            } else {
                return mid; // it's the first
            }
        }
    }
```

Summary

Thinking about our three main measures of code quality (safe from bugs, easy to understand, ready for change), the techniques in this lecture have mainly addressed safety. Readiness for change was considered by writing tests that only depend on behavior in the spec. Testing doesn't measure or improve ease of understanding, but we talked about code review, which does.

testing matters
- you need to convince others (and yourself) that your code works
- testing generally can't prove absence of bugs, but can increase quality by reducing bugs

test early and often
- unit testing catches bugs before they have a chance to hide
- automate the process so you can run it frequently
- regression testing will save time in the long run

be systematic
- use input partitioning, boundary testing, and coverage
- regard testing as a creative design problem

use tools and build your own
- automated testing frameworks (JUnit) and coverage tools (EclEmma)
- design modules to be driven, and use stubs for repeatable behavior