
Design for Change (class level)

Charlie Garrod       Bogdan Vasilescu
Administrivia

• Homework 1 due on Thursday 11:59pm
  – Everyone must read and sign our collaboration policy
• Reading assignment due today
• (Optional) reading due Thursday
• Next reading due next Tuesday
Part 1: Design at a Class Level
Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns

Part 2: Designing (Sub)systems
Understanding the Problem
Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies
Testing Subsystems
Design for Reuse at Scale: Frameworks and APIs

Part 3: Designing Concurrent Systems
Concurrency Primitives, Synchronization
Designing Abstractions for Concurrency
Design Goals for Today

• **Design for Change** (flexibility, extensibility, modifiability)

also

• Design for Division of Labor
• Design for Understandability
• Design for Reuse (more later)
Software Change

• ...accept the fact of change as a way of life, rather than an untoward and annoying exception.
  —Brooks, 1974

• Software that does not change becomes useless over time.
  —Belady and Lehman

• For successful software projects, most of the cost is spent evolving the system, not in initial development
  — Therefore, reducing the cost of change is one of the most important principles of software design
The limits of exponentials

Computing capability

Human capacity
Building Complex Systems

- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations
Fundamental Design Principle for Change: Information Hiding

• Expose as little implementation detail as necessary
• Allows to change hidden details later
Information hiding

• Visibility modifiers in Java ("encapsulation")
  – private
  – "package private"
  – protected
  – public

• Interface types vs. class types
Information hiding is more general than visibility

• Use interfaces to separate expectations from implementation
  – Create interfaces to define your API
  – Declare variables, arguments, and return values as interface type
    • Write API in terms of other interfaces, not implementations
• Do not publicly document implementation details
A more complex example

```java
public class Complex {
    private final double re;  // Real part
    private final double im;  // Imaginary part

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex minus(Complex c) { ... }
    public Complex times(Complex c) { ... }
    public Complex dividedBy(Complex c) { ... }
}
```
Using the Complex class

```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new Complex(-1, 0);
        Complex d = new Complex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");

        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it prints

-1.0 + 1.0i
-0.0 + -1.0i
Extracting an interface from our class

public interface Complex {
    // No constructors, fields, or implementations!
    double realPart();
    double imaginaryPart();
    double r();
    double theta();
    Complex plus(Complex c);
    Complex minus(Complex c);
    Complex times(Complex c);
    Complex dividedBy(Complex c);
}

An interface defines but does not implement API
Modifying our earlier class to use the interface

public class OrdinaryComplex implements Complex {
    private final double re; // Real part
    private final double im; // Imaginary part

    public OrdinaryComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex plus(Complex c) {
        return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
    }
    public Complex minus(Complex c) { ... }
    public Complex times(Complex c) { ... }
    public Complex dividedBy(Complex c) { ... }
}
Modifying our earlier client to use the interface

```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it still prints

-1.0 + 1.0i
-0.0 + -1.0i
Interfaces permit multiple implementations

public class PolarComplex implements Complex {
    private final double r; // Radius
    private final double theta; // Angle

    public PolarComplex(double r, double theta) {
        this.r = r;
        this.theta = theta;
    }

    public double realPart() { return r * Math.cos(theta); }
    public double imaginaryPart() { return r * Math.sin(theta); }
    public double r() { return r; }
    public double theta() { return theta; }

    public Complex plus(Complex c) { ... } // Completely new impls
    public Complex minus(Complex c) { ... }
    public Complex times(Complex c) { ... }
    public Complex dividedBy(Complex c) { ... }
}
Interface decouples client from implementation

```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(Math.PI, 1); // -1
        Complex d = new PolarComplex(Math.PI/2, 1); // i

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + " + e.imaginaryPart() + "i");

        e = c.times(d);
        System.out.println(e.realPart() + " + " + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it **STILL** prints

-1.0 + 1.0i
-0.0 + -1.0i
Information hiding facilitates change, promotes reuse

• Think in term of abstractions, not implementations
  – Abstractions are more likely to be reused
• Can change implementations more easily
  – Different performance
  – Different behavior
• Prevents bad programmer behavior, unnecessary dependencies
Other benefits of information hiding

- Decoupled subsystems are easier to understand in isolation
- Speeds up system development
- Reduces cost of maintenance
- Improves effectiveness of performance tuning

But:
- Requires anticipation of change (judgment)
- Not all change can be anticipated
Best practices for information hiding

• Carefully design your API
• Provide *only* functionality required by clients
  – *All* other members should be private
• You can always make a private member public later without breaking clients
  – But not vice-versa!
CONTRACTS
(BEYOND TYPE SIGNATURES)
Contracts and Clients

- Hidden from service* client
- Service* interface
- Hidden from service* provider

* service = object, subsystem, ...
Contracts

• Agreement between provider and users of an object

• Includes
  – Interface specification (types)
  – Functionality and correctness expectations
  – Performance expectations

• What the method does, not how it does it
  – Interface (API), not implementation
Who’s to blame?

Algorithms.shortestDistance(g, “Tom”, “Anne”);

> ArrayOutOfBoundsException
Who’s to blame?

Algorithms.shortestDistance(g, “Tom”, “Anne”);

> -1
Who’s to blame?

Algorithms.shortestDistance(g, "Tom", "Anne");

> 0
Who’s to blame?

class Algorithms {

    /**
     * This method finds the
     * shortest distance between to
     * vertices. It returns -1 if
     * the two nodes are not
     * connected. */

    int shortestDistance(...) {...}
}

Who’s to blame?

Math.sqrt(-5);

> 0
Who’s to blame?

/**
 * Returns the correctly rounded positive square root of a
 * `double` value.
 * Special cases:
 * <ul>
 * <li>If the argument is NaN or less than zero, then the result is NaN.
 * <li>If the argument is positive infinity, then the result is positive infinity.
 * <li>If the argument is positive zero or negative zero, then the result is the same as the argument.</ul>
 * 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`.
 * If the argument is NaN or less than zero, the result is NaN.
 */

class sqrt {
  public static double sqrt(double a) { ...}
}
public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let k be the number of bytes actually read; these bytes will be stored in elements b[off] through b[off+k-1], leaving elements b[off+k] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off
public int read(byte[] b, int off, int len) throws IOException

- Specifies the read method for reading up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read.

- The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.

- If len is zero, then no bytes are read, and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned into b.

- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes actually read, these bytes will be stored in elements b[off] through b[off+len-1], leaving elements b[off+len] through b[b.length-1] unaffected.

- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

- Multiple error cases, each with a precondition
  - Includes “runtime exceptions” not in throws clause

- Specification of return
- Timing behavior (blocks)
- Case-by-case spec
  - len=0 \(\rightarrow\) return 0
  - len>0 && eof \(\rightarrow\) return -1
  - len>0 && !eof \(\rightarrow\) return >0

- Exactly where the data is stored
Specifications

• Contains
  – Functional behavior
  – Erroneous behavior
  – Quality attributes (performance, scalability, security, …)

• 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
Functional Specification

• States method’s and caller’s responsibilities

• Analogy: legal contract
  – If you pay me this amount on this schedule...
  – I will build a with the following detailed specification
  – Some contracts have remedies for nonperformance

• Method contract structure
  – **Preconditions**: what method requires for correct operation
  – **Postconditions**: what method establishes on completion
  – **Exceptional behavior**: what it does if precondition violated

• Defines what it means for impl to be correct
Functional Specification

• States method's and caller's responsibilities
• Analogy: legal contract
  – If you
  – I will
  – Some contracts have remedies for nonperformance
• Method contract structure
  – **Preconditions**: what method requires for correct operation
  – **Postconditions**: what method establishes on completion
  – **Exceptional behavior**: what it does if precondition violated
• Defines what it means for impl to be correct

**What does the implementation have to fulfill if the client violates the precondition?**
Formal Specifications

```java
/*@ requires len >= 0 && array != null && array.length == len; 
@ ensures \result == \sum int j; 0 <= j && j < len; array[j]);
@*/
int total(int array[], int len);
```

**Advantage of formal specifications:**

* runtime checks (almost) for free
* basis for formal verification
* assisting automatic analysis tools

JML (Java Modelling Language) as specifications language in Java (inside comments)

Disadvantages?
Runtime Checking of Specifications with Assertions

/*@ requires len >= 0 && array.length == len
   @ ensures \result ==
   @       (\sum int j; 0 <= j && j < len; array[j])
   @*/
float sum(int array[], int len) {
    assert len >= 0;
    assert array.length == len;
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    assert sum …;
    return sum;
}
Specifications in the real world

**Javadoc**

/**
 * Returns the element at the specified position of this list.
 * <p>This method is <i>not</i> guaranteed to run in constant time. In some implementations, it may run in time proportional to the element position.
 * @param index position of element to return; must be non-negative and less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range ({@code index < 0 || index >= this.size()})
 */
E get(int index);
Javadoc contents

• Document
  – Every parameter
  – Return value
  – Every exception (checked and unchecked)
  – What the method does, including
    • Purpose
    • Side effects
    • Any thread safety issues
    • Any performance issues

• Do **not** document implementation details
Write a Specification

- Write
  - a type signature,
  - a textual (Javadoc) specification, and
  - a formal specification

for a function `slice(list, from, until)` that returns all values of a list between positions `<from>` and `<until>` as a new list

Reminder: Formal specification

```java
/*@ requires len >= 0 && array != null &&
    array.length == len;
@ ensures \result ==
    \sum int j;  0 <= j &&
    j < len;  array[j];
@*/
int total(int array[], int len);
```

Reminder: Javadoc specification

```java
/**
 * Returns ...
 * @param index position of element ...
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the ...
 *     (@code index < 0 || index >= this.size)
 */
E get(int index);
```
Contracts and Interfaces

• All objects implementing an interface must adhere to the interface’s contracts
  – Objects may provide different implementations for the same specification
  – Subtype polymorphism: Client only cares about interface, not about the implementation

\[ p.getX() \quad s.read() \]

=> Design for Change
FUNCTIONAL CORRECTNESS
(UNIT TESTING AGAINST INTERFACES)
Context

- **Design for Change** as goal
- **Encapsulation** provides technical means
- **Information Hiding** as design strategy
- **Contracts** describe behavior of hidden details
- **Testing** helps gaining confidence in functional correctness (w.r.t. contracts)
Functional correctness

• Compiler ensures types are correct (type-checking)
  – Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”
Functional correctness

• Compiler ensures **types** are correct (**type-checking**)
  – Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”

• **Static analysis** tools (e.g., FindBugs) recognize many common problems (**bug patterns**)
  – Warns on possible NullPointerExceptions or forgetting to close files
FindBugs

```java
@Override
public void run() {
    Lock localLock = new ReentrantLock();
    localLock.lock();
    int a = 1;
    localLock.lock();
    if (a == 2) {
        localLock.unlock();
    } else {
        // do nothing
    }
    return;
}
```
public final class CartesianPoint {

    private int X, Y;

    CartesianPoint(int x, int y) {
        this.X = x;
        this.Y = y;
    }

    public int getY() {
        return Y;
    }

    public int getX() {
        return X;
    }
}

CheckStyle Problem (9 items)

- ',' is not followed by whitespace.
- '=' is not followed by whitespace.
- '=' is not preceded with whitespace.
- File contains tab characters (this is the first instance).
- Name 'GetY' must match pattern '^\[a-zA-Z0-9]*$'.
- Name 'X' must match pattern '^\[a-zA-Z0-9]*$'.
- Name 'Y' must match pattern '^\[a-zA-Z0-9]*$'.

Description:

Description:

Resou
Functional correctness

• Compiler ensures types are correct (type-checking)
  – Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”

• Static analysis tools (e.g., FindBugs) recognize many common problems (bug patterns)
  – Warns on possible NullPointerExceptions or forgetting to close files

• How to ensure functional correctness of contracts beyond type correctness and bug patterns?
Formal verification

• Use mathematical methods to prove correctness with respect to the formal specification
• Formally prove that all possible executions of an implementation fulfill the specification
• Manual effort; partial automation; not automatically decidable
Testing

• Executing the program with selected inputs in a controlled environment

• Goals
  – Reveal bugs, so they can be fixed (main goal)
  – Assess quality
  – Clarify the specification, documentation
Re: Formal verification, Testing

“Beware of bugs in the above code; I have only proved it correct, not tried it.”
Donald Knuth, 1977

"Testing shows the presence, not the absence of bugs.”
Edsger W. Dijkstra, 1969
Q: Who’s right, Dijkstra or Knuth?

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }

    return -(low + 1); // key not found.
}
```

Binary search from java.util.Arrays
Q: Who’s right, Dijkstra or Knuth?

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }

    return -(low + 1); // key not found.
}
```

Spec: sets mid to the average of low and high, truncated down to the nearest integer.

Fails if low + high > MAXINT (2^{31} - 1) Sum overflows to negative value
A: They’re both right

• There is no silver bullet!
• Use all the tools at your disposal
  – Careful design
  – Testing
  – Formal methods (where appropriate)
  – Code reviews
  – …
• You’ll still have bugs, but hopefully fewer.
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
- Other qualities (performance, robustness, usability, security, ...)

Our focus now
Manual testing

**Generic test case: user sends MMS with picture attached.**

<table>
<thead>
<tr>
<th>Step ID</th>
<th>User Action</th>
<th>System Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Go to Main Menu</td>
<td>Main Menu appears</td>
</tr>
<tr>
<td>2</td>
<td>Go to Messages Menu</td>
<td>Message Menu appears</td>
</tr>
<tr>
<td>3</td>
<td>Select “Create new Message”</td>
<td>Message Editor screen opens</td>
</tr>
<tr>
<td>4</td>
<td>Add Recipient</td>
<td>Recipient is added</td>
</tr>
<tr>
<td>5</td>
<td>Select “Insert Picture”</td>
<td>Insert Picture Menu opens</td>
</tr>
<tr>
<td>6</td>
<td>Select Picture</td>
<td>Picture is Selected</td>
</tr>
<tr>
<td>7</td>
<td>Select “Send Message”</td>
<td>Message is correctly sent</td>
</tr>
</tbody>
</table>

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?
Automated testing

- Execute a program with specific inputs, check output for expected values
- Easier to test small pieces than testing user interactions
- Set up testing infrastructure
- **Execute tests regularly**
  - After *every* change
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);
Example

```cpp
/**
 * 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 / longer than array.length)
- Test null as array
- Test with a very long array
Unit Tests

- 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
- 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
• Tool support available
• Can be used as design mechanism
JUnit

```java
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
JUnit conventions

- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests should run fast
- Tests 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.)
Test organization

• Conventions (not requirements)
• Have a test class FooTest for each public class Foo
• Have a source directory and a test directory
  – Store FooTest and Foo in the same package
  – Tests can access members with default (package) visibility
Selecting test cases: common strategies

• Read specification
• Write tests for
  – Representative case
  – Invalid cases
  – Boundary conditions
• Are there difficult cases? (error guessing)
  – Stress tests?
  – Complex algorithms?
• Think like an attacker
  – The tester’s goal is to find bugs!
• How many test should you write?
  – Aim to cover the specification
  – Work within time/money constraints
Testable code

• Think about testing when writing code
• Unit testing encourages you 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
Write testable code

Unit testing as design mechanism

- Code with low complexity
- Clear interfaces and specifications

Source:
http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx
When should you stop writing tests?

• When you run out of money...
• When your homework is due...
• When you can't think of any new test cases...
• The *coverage* of a test suite
  – Trying to test all parts of the implementation
  – Statement coverage
    • Execute every statement, ideally
    • Compare to: method coverage, branch coverage, path coverage
When to stop writing tests?

• Outlook: statement coverage
  – Trying to test all parts of the implementation
  – Execute every statement, ideally

Does 100% coverage guarantee correctness?
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found.
When to stop writing tests?

• Outlook: statement coverage
  – Trying to test all parts of the implementation
  – Execute every statement, ideally

Does less than 100% coverage guarantee incorrectness?
public int subtract(int a, int b) {
    int x = a - b;
    return x;
}

public boolean conditional(int a, int b) {
    return a == b;
}

public void uncoveredMethod() {
    String line = "not covered";
}

public String coveredMethod() {
    String a = "hello"; String b = "world"; return a.concat(b);
}
# Coverage Report - All Packages

<table>
<thead>
<tr>
<th>Package</th>
<th># Classes</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Packages</td>
<td>55</td>
<td>75%</td>
<td>64%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.ant</td>
<td>11</td>
<td>52%</td>
<td>43%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.check</td>
<td>3</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.coveragedata</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.instrument</td>
<td>10</td>
<td>90%</td>
<td>75%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.merge</td>
<td>1</td>
<td>86%</td>
<td>78%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html</td>
<td>3</td>
<td>87%</td>
<td>80%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html.files</td>
<td>4</td>
<td>91%</td>
<td>77%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.xml</td>
<td>1</td>
<td>87%</td>
<td>62%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.util</td>
<td>9</td>
<td>60%</td>
<td>69%</td>
</tr>
<tr>
<td>someotherpackage</td>
<td>1</td>
<td>83%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Report generated by [Cobertura](http://cobertura.sourceforge.net) 1.9 on 6/9/07 12:37 AM.
Run tests frequently

• You should only commit code that is passing all tests
• Run tests before every commit
• If entire test suite becomes too large and slow for rapid feedback:
  – Run local tests ("smoke tests", e.g. all 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 - Travis CI

Automatically builds, tests, and displays the result
Continuous integration - Travis CI

You can see the results of builds over time
Testing, Static Analysis, and Proofs

• **Testing**
  – Observable properties
  – Verify program for one execution
  – Manual development with automated regression
  – Most practical approach now
  – Does not find all problems (unsound)

• **Static Analysis**
  – Analysis of all possible executions
  – Specific issues only with conservative approx. and bug patterns
  – Tools available, useful for bug finding
  – Automated, but unsound and/or incomplete

• **Proofs (Formal Verification)**
  – Any program property
  – Verify program for all executions
  – Manual development with automated proof checkers
  – Practical for small programs, may scale up in the future
  – Sound and complete, but not automatically decidable

**What strategy to use in your project?**
SUMMARY: DESIGN FOR CHANGE/
DIVISION OF LABOR
Design Goals

• Design for Change such that
  – Classes are *open for extension* and modification without invasive changes
  – Classes encapsulate details likely to change behind (small) stable interfaces

• Design for Division of Labor such that
  – Internal parts can be *developed* independently
  – Internal details of other classes do not need to be *understood*, contract is sufficient
  – Test classes and their contracts separately (unit testing)