
Design for Change (class level)

Christian Kästner  Bogdan Vasilescu
Administrivia

• Homework 1 due today
• Homework 2:
  – out tonight
  – due next Thursday (Feb 2)
• Reading assignment due next Tuesday (Jan 31)
The Strategy Design Pattern

Context
algorithm()

Strategy
execute()

ConcreteStrA
execute()

ConcreteStrB
execute()
The Composite Design Pattern

```java
operation() {
    for (c in children)
        c.operation();
}
```
Design Exercise (on paper)

• You are designing software for a shipping company.
• There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
• Two important considerations are the weight of an item and its insurance cost.
  – Fragile items cost more to insure.
  – All letters are assumed to weigh an ounce
  – We must keep track of the weight of other packages.
• The company sells boxes and customers can put several items into them.
  – The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  – However, most of the software should treat a box holding several items just like a single item.
• Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)
interface Item {
    double getWeight();
}

class Letter implements Item {
    double weight;
    double getWeight() {...}
}

class Box implements Item {
    ArrayList<Item> items=new ArrayList<>();
    double getWeight() {
        double weight = 0.0
        for(Item item : items) {
            weight += item.getWeight();
        }
    }
    void add(Item item){
        items.add(item);
    }
}
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

- Service* interface
- Service* implementation
- Client environment

*service = object, subsystem, ...
What is a contract?

• Agreement between an object and its user
• Includes
  – Method signature (type specifications)
  – Functionality and correctness expectations
  – Performance expectations

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

/**
 * Returns the correctly rounded positive square root of a
 * {@code 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 {@code double} value closest to
 * the true mathematical square root of the argument value.
 *
 * @param a a value.
 * @return the positive square root of {@code a}.
 * @return the positive square root of {@code a}.
 * If the argument is NaN or less than zero, the result is NaN.
 */

public static double sqrt(double a) { …}
Method contract details

• States method’s and caller’s responsibilities

• Analogy: legal contract
  – If you pay me this amount on this schedule I will build you a house 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 implementation to be correct
Formal contract specification

Java Modelling Language (JML)

```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);
```

- **Theoretical approach**
  - **Advantages**
    - Runtime checks generated automatically
    - Basis for formal verification
    - Automatic analysis tools
  - **Disadvantages**
    - Requires a lot of work
    - Impractical in the large
    - Some aspects of behavior not amenable to formal specification
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;
}
Runtime Checking of Specifications with Exceptions

/*@ requires len >= 0 && array.length == len
@ ensures \result ==
  @               (\sum int j; 0 <= j && j < len; array[j])
@*/
float sum(int array[], int len) {
    if (len < 0 || array.length != len)
        throw IllegalArgumentException(...);
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    return sum;
}
Textual contract specification - Javadoc

• Practical approach
  – Writing specifications is good practice
  – Especially necessary when reusing code and integrating code
  – Writing fully formal specifications is often unrealistic

• 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
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);
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 ==
  \( \sum \text{int } j; \ 0 \leq j && j < \text{len}; \ array[j] \);
@*/
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
 * */
 int 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
ASIDE:
THE EQUALS CONTRACT
The class hierarchy

• All Java objects inherit from `java.lang.Object`

- **Instrument**
  - **Guitar**
- **Toy**
  - **Yoyo**

• Commonly-used/overridden public methods:
  - `equals` - returns true if the two objects are “equal”
  - `hashCode` - returns an `int` that must be equal for equal objects, and is likely to differ on unequal objects
  - `toString` - returns a printable string representation
The `.equals(Object obj)` contract

- **Reflexive** – every object is equal to itself
- **Symmetric** – if `a.equals(b)` then `b.equals(a)`
- **Transitive** – if `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
- **Consistent** – Invoking `a.equals(b)` repeatedly returns the same value unless `a` or `b` is modified; implemented by `.hashCode()`
- **“Non-null”** – `a.equals(null)` returns false
- Taken together these ensure that equals is a global equivalence relation over all objects
The == operator vs. the equals() method

• The == operator determines if two references are identical to each other
• The equals method determines if objects are equal
• User classes can override the equals method to implement a domain-specific test for equality
What’s the output?

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
...
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
```

```java
System.out.println(p1 == p2);
```

False
What’s the output?

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
    ...
    Point p1 = new Point(1, 2);
    Point p2 = new Point(1, 2);
    System.out.println(p1.equals(p2));
    False
```
What’s the output?

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
...
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
System.out.println(p1.equals(p2));
```

```
public boolean equals(Object obj) {
    return this == obj;
}
```

Default object impl
What’s the output?

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
...
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

@Override
public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Point) {
        Point that = (Point) obj;
        result =
            (this.getX() == that.getX() && this.getY() == that.getY());
    }
    return result;
}
System.out.println(p1.equals(p2));
```

True
What’s the output?

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
...
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

@override
public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Point) {
        Point that = (Point) obj;
        result =
                (this.getX() == that.getX() && this.getY() == that.getY());
    }
    return result;
}
...
HashSet<Point> coll = new HashSet<Point>();
coll.add(p1);
System.out.println(coll.contains(p2));
```

False
The `.hashCode()` contract

- **Consistent**
  - Invoking `x.hashCode()` repeatedly returns same value unless `x` is modified

- **`x.equals(y)` implies `x.hashCode() == y.hashCode()`**
  - The reverse implication is not necessarily true:
    - `x.hashCode() == y.hashCode()` does not imply `x.equals(y)`

- **Advice**: Override `.equals()` if and only if you override `.hashCode()`
What’s the output?

```java
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
...
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
HashSet<Point> coll = new HashSet<Point>();
coll.add(p1);
System.out.println(coll.contains(p2));
```

True
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
...
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

@Override
public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Point) {
        Point that = (Point) obj;
        result =
            (this.getX() == that.getX() && this.getY() == that.getY());
    }
    return result;
}

@Override public int hashCode() {
    return (41*(41 + getX()) + getY());
}

But it’s not over; see Effective Java #8
The lesson: Conforming to contracts can be difficult!
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”
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); }
}
class Cow implements Animal {
    public void makeSound() { mew(); }
    public void mew() {System.out.println("Mew!"); }
}

1 Animal a = new Animal();
2 a.makeSound();
3 Dog d = new Dog();
4 d.makeSound();
5 Animal b = new Cow();
6 b.mew();
7 b.jump();

• What happens?
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
```java
@Override
public void run() {
    Lock localLock = new ReentrantLock();
    l.lock();
    int a = 1;
    localLock.lock();
    if (a == 2) {
        l.unlock();
    } else {
        // do nothing
    }
    return;
}
```

0 errors, 12 warnings, 0 others

- Description
  - Iterator is a raw type. References to generic type Iterator<E> should be parameterized
  - No required execution environment has been set
  - plugin.ProgramPoint defines equals and uses Object.hashCode() [Troubleshooting(14), High confidence]
  - tests.NoUnlock$T3.run() does not release lock on all paths [Troubleshooting(12), High confidence]
  - tests.NoUnlock$T4.run() might ignore java.lang.Exception [Troubleshooting(14), High confidence]
  - Type safety: Unchecked cast from Object to Map.Entry<String,ProgramPoint.LockState>
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;
    }

    File contains tab characters (this is the first instance).
    Name 'GetY' must match pattern '^([a-z][a-zA-Z0-9]+)$'.
    Name 'X' must match pattern '^([a-z][a-zA-Z0-9]+)$'.
    Name 'Y' must match pattern '^([a-z][a-zA-Z0-9]+)$'.
}
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.
}
```
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.
}
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

```java
/**
 * 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
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...
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 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.
  - static void assertEquals(double expected, double actual, double delta); // Asserts that two doubles are equal to within a positive delta
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
- AssertionError 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

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

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 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?
A: No
## 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>88%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting</td>
<td>3</td>
<td>87%</td>
<td>80%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html</td>
<td>4</td>
<td>91%</td>
<td>77%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html.files</td>
<td>1</td>
<td>87%</td>
<td>62%</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.xml</td>
<td>1</td>
<td>100%</td>
<td>95%</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
  – Subtype polymorphism enables changes behind interface
  – 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)