Guidelines for Class Design & Testing

Horstmann Chapter 3.4-5 & 3.7

Unit 4
CS 3354
Spring 2017

Jill Seaman

Object-Oriented Design Continued:

- The previous unit was concerned with how to find classes for solving a practical programming problem
  - ✦Focused on classes and relationships to each other
- In this chapter, we explore how to write a single class well, and then test it.
- Note:
  - ✦Bad example shows how NOT to do it.
  - ✦Good example shows how to do it right.

3.4 The importance of Encapsulation

Or . . .The importance of Information Hiding

- Assume we have implemented a Day class as follows:

```
public class Day {
  public int year, month, date;
 ...
}
```

- But now we want to represent the day by an integer recording the number of days since Jan 1, 1970.
  - ✦We remove the year, month, and date fields and supply an `int julian` field, and add getYear(); getMonth(); getDate(); functions.
  - ✦Replace d.year with d.getYear()
  - ✦Replace d.year++ with d.setYear(d.getYear()+1);
  - ✦Etc.
- This is too much trouble! Better to have had made fields private and had a good public interface from the start.

3.4.1 Accessors and Mutators

- Mutator: method that changes object state (field values).
- Accessor: method that reads object state without changing it.
- Class without mutators is called immutable
- String is immutable
- java.util.Date, Calendar, and GregorianCalendar are mutable
- immutable objects are good, no one else can change them.
- immutable objects are easy to reason about, do not need to understand how they might be changed by other objects.

Don't supply set methods for every instance field.

- Our Day class has getYear, getMonth, getDate accessors
- Should we add setYear, setMonth,setDate mutators, with input validation?
- Example:

```
Day deadline = new Day(2001, 1, 31);
deadline.setMonth(2); // ERROR
deadline.setDate(28);
```

- Maybe we should call setDate first?

```
Day deadline = new Day(2001, 2, 28);
deadline.setDate(31); // ERROR
deadline.setMonth(3);
```

- Not all mutators are bad, maybe have a function to add a specific number of days (it knows how many days in each month):

```
Day deadline = new Day(2001, 2, 28);
deadline.addDays(31);
```

Sharing Mutable References unintentionally

- Pitfall:

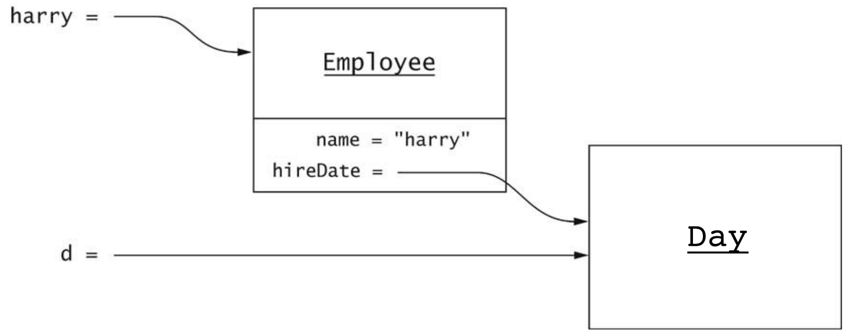
```
class Employee {
  private String name;
  private double salary;
  private Day hireDate;
  . . .
  public String getName() {return name;}
  public double getSalary() {return salary;}
  public Day getHireDate() {return hireDate;}
}
```

We want this class to be immutable

- No mutators, but if Day is mutable:

```
Employee harry = . . .;
Day d = harry.getHireDate();
d.setMonth(12); // changes Harry's state!!!
```

Sharing Mutable References



Sharing Mutable References unintentionally: solution

- Solution 1: If the class implements clone, you can use clone:

The clone method of the Object class makes a copy of the object with the same fields as the original.

```
public Calendar getHireDate() {
   return (Calendar)hireDate.clone();
}
```

- Solution 2: If the class has a copy constructor, use it:

```
public Day getHireDate() {
   return new Day(hireDate);
}
```

- If the class has neither, consider adding a copy constructor to the class, if possible. Or use another constructor.

Sharing Mutable References unintentionally

- Pitfall:

```
class Employee {
  public Employee(String aName, Day aHireDate {
    name = aName;
    hireDate = aHireDate;
  }
}
```

- No mutators, but Day is still mutable:

```
Day d = new Day();
Employee e = new Employee("Harry Hacker", d);
d.setMonth(12);  // changes Harry's state!!!
```

- Remedy: use clone in the constructor:

```
public Employee(String aName, Day aHireDate) {
    name = aName;
    hireDate = new Day(aHireDate);
}
```

3.4.3 Separating Accessors and Mutators

- A method that returns information about an object should ideally not change the object state.
- A method that changes the object state should ideally have return type void.
- Apparent example of violation:

```
java.util.Queue:
void add(E x)  //enqueue (at rear)
E remove()     //dequeue (remove from front, and returns it)
```

- remove yields front value AND removes it (is it an accessor or mutator?)
- What if I want to view the front element without removing it?

Separating Accessors and Mutators

- Better interface, peek is accessor, remove is mutator:

```
java.util.Queue:
void add(E x)  //enqueue (at rear)
E peek()       //returns front element without removing it.
void remove()  //removes front element (no return value)
```

- But less convenient, programmer has to do two operations in order to dequeue: peek then remove.
- The actual interface is more convenient:

```
java.util.Queue:
void add(E x)  //enqueue (at rear)
E peek()       //returns front element without removing it.
E remove()     //dequeue (remove from front, and returns it)
```

- Refine rule of thumb: Mutators can return a convenience value, provided there is also an accessor to get the same value.

3.4.4 Side Effects

- A **side effect** of a method is any data modification that is observable when the method is called.
- A method can change:
  - ✦fields of its class (then it’s a mutator)
  - ✦its arguments
  - ✦accessible static fields of other objects
- Changing the arguments or other objects is unexpected.
- Good example:

```
a.addAll(b)  //adds all elements of collection b to collection a
```

- Changes a, but not b.

Side Effects

- Bad example:
- System.out is a public static object (a PrintStream)
- System.out.println(x); changes System.out

```
if (newMessages.isFull())
   System.out.println("Sorry--no space");
```

- Your classes may need to run in an environment without System.out
- Instead throw an exception to report an error condition!
- “Printing error messages to System.out is reprehensible:”
- Try to access System.out from the driver only.

3.4.5 The Law of Demeter

“Don’t talk to strangers”

- An object should call methods on (or use) only the following
  - the object itself (self call)
  - the objects attributes (instance variables)
  - the parameters of methods of the object
  - Any object created by this object
- It should NOT call methods on an object **returned** from a method call.
  - Specifically: An object should not ask another object to give it a part of its internal state to manipulate.
- This is a good guideline, not a law.

The Law of Demeter:

- Bad Example: Mail system in chapter 2: Connection object changes contents of Mailbox returned by MailSystem.findMailbox:

```
Mailbox currentMailbox = mailSystem.findMailbox(…);
currentMailbox.setPasscode(accumulatedKeys);
```

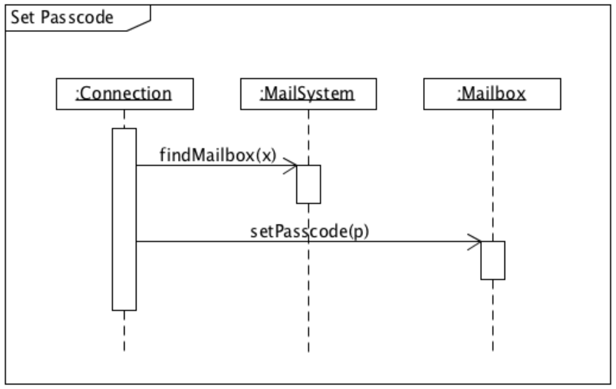
- I call this: Sharing Mutable References **intentionally**
- Breaks encapsulation (information hiding) of the MailSystem class
- A future version of the MailSystem might not use Mailbox objects.
- Remedy in mail system: Delegate mailbox methods to mail system:

```
mailSystem.setPasscode(mailboxNumber, accumulatedKeys);
```

- Reduces the dependencies in the system (coupling)

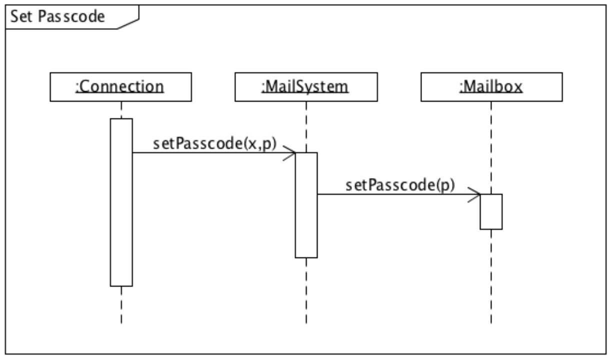
Law of Demeter: before

- Connection depends on both MailSystem and Mailbox.
- Connection class is coupled with the Mailbox class.



Law of Demeter: after

- Connection depends on only MailSystem now.
- No coupling between Connection and Mailbox.



3.5 Analyzing the Quality of an Interface

- The design of classes must be approached from two points of view simultaneously.
- But the two have different priorities:
  - class designer: efficient algorithms, convenient coding
  - class user (another programmer): ability to use operations without reading code, just the right operations provided.

- Use the following criteria to evaluate your class interfaces:
  - **cohesion, completeness, convenience, clarity, consistency.**
- Note: these criteria sometimes conflict with each other.
  Use your judgment to balance these conflicts.

Cohesion

- Class should describe a a single concept
- Methods should be related to support a single purpose
- Bad example:

```
public class Mailbox
{
   public addMessage(Message aMessage) { ... }
   public Message getCurrentMessage() { ... }
   public Message removeCurrentMessage() { ... }
   public void processCommand(String command) { ... }
   ...
}
```

- Why is processCommand there? It’s not related to Messages. Should probably go elsewhere.

Completeness

- Support ALL operations that are well-defined (or make sense) for the abstract data type
- Potentially bad example: java.util.Date
  We want to count how many milliseconds elapse between two statements:

```
Date start = new Date();
// do some work
Date end = new Date();
// How many milliseconds between start and stop?
```

- No such operation in Date class
- Does it fall outside the responsibility?
- It provides a way to check ordering between Dates, and get an absolute number of milliseconds, but not the difference.

Convenience

- A good interface makes all tasks possible . . . and common tasks simple
- Bad example: Reading from System.in before Java 5.0

```
BufferedReader in = new BufferedReader(
                    new InputStreamReader(System.in));
String line = in.readLine();
```

- Why doesn't System.in have a method to read a line of text?
- After all, System.out has println.
- Scanner class finally helped alleviate this inconvenience

Clarity

- The interface of a class should be clear to programmers, without generating confusion.
- ListIterator.add(T) makes sense, adds before the cursor:

```
LinkedList<String> list = new LinkedList<String>();
list.add(“A"); list.add(“B"); list.add(“C");
ListIterator<String> iterator = list.listIterator(); // |ABC
iterator.next(); // A|BC
iterator.add("X"); // AX|BC
```

- API documentation for add(): Inserts the specified element into the list. The element is inserted immediately before the element that would be returned by next(), if any, and after the element that would be returned by previous(), if any. (If the list contains no elements, the new element becomes the sole element on the list.) The new element is inserted before the implicit cursor.

Clarity continued

- ListIterator.remove() does NOT always remove the element before the cursor:

```
// This isn’t how it works, both calls are illegal
//   because we haven’t called next() (or previous())

//continued from previous, AX|BC
iterator.remove(); // A|BC (should remove the X)
iterator.remove(); // |BC  (should remove the A)
```

- API documentation for remove(): Removes from the list the last element <u>that was returned by next</u> or previous. This call can only be made once per call to next or previous. It can be made only if add has not been called after the last call to next or previous. (confusing!!)

Consistency

- The operations in a class should be consistent with each other with respect to names, parameters and return values, and behavior.
- To specify a day in the Gregorian-Calendar class call:

```
new GregorianCalendar(year, month - 1, day)
```

- because the month should be between 0 and 11, but the day is between 1 and 31. Why is only the month 0-based?
- To check if two strings are equal you call s.equals(t); or s.equalsIgnoreCase(t); to do case-insensitive comparison.
- Theres also compareTo/compareToIgnoreCase.
- But then there’s this (why break the pattern with a flag?):

```
boolean regionMatches(boolean ignoreCase, int toffset, String other,
int ooffset, int len)
```

3.7 Unit Testing with JUnit

Much of the material in this section of the lecture is from the online tutorials listed on the Readings page (accessible from the class website). The section from the Horstmann book on JUnit is out of date.

Testing

Test case: set of inputs and expected results that exercises (part of) a system with the purpose of detecting faults/bugs (coding mistakes).

Test cases should contain the following:

- **Name**: Explains what is being tested
- **Input**: Set of input data and/or commands and/or actions
- **Expected results:** Output or state or behavior that is correct for the given input.

Unit testing: individual program units (i.e. classes) are tested by the developers.

JUnit

- Open source framework for the automation of unit testing in Java.
- Test cases are written in Java, and are executable with no input.
- Test code is separate from application code.
- Tests can run independently of each other (one failure does not cause others to fail).
- Tests can be run automatically, overnight or on demand.
- The success or failure of a test is visible at a glance.
- It is used widely in the industry.
- It can be downloaded from junit.org.
- I will be using version 4.12.

JUnit Tutorial (based on vogella.com)

- First we will consider the code to be tested:

```
package mine;

class MyClass {
    public int multiply (int x, int y) {
        return x*y;
    }
}
```

- How can I use JUnit to test it?
  - ✦Create a Test class: a class which is used only for testing.
  - ✦Add a method that will implement the test case.
  - ✦Annotate the method with the @Test annotation.
  - ✦In this method, compare the expected result of the code execution to the actual result, using a method provided by the JUnit framework.

JUnit Tutorial: the test class

- The test class:

```
package mine;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MyClassTest {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {

        // MyClass is tested
        MyClass tester = new MyClass();

        // Tests
        // assertEquals(String message, int expected, int actual)
        assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
        assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
        assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
    }
}
```

JUnit Tutorial: How to compile and run the test? Part I: From the command line

- Download the jar files from junit.org:

```
junit.jar
hamcrest-core.jar
```

- The downloaded filenames may include version numbers.
- Put these in a directory.
- I use src as my root directory. I put these in src/lib.
- I also made a src/bin file to store my *.class files.
- The *.java files from the last slides go in src/mine.

JUnit Tutorial: How to compile and run the test? Part I: From the command line

- Now I need a driver class to execute the test(s):

```
package mine;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(MyClassTest.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

- I passed the name of my Test class to the runClasses method.

JUnit Tutorial: How to compile and run the test? Part I: From the command line

- Here is the compile and execute process ($ is the prompt):

```
$ javac -d bin -cp lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar
mine/*.java
$ java -cp bin:lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar
mine.MyTestRunner
```

- No output means the test(s) passed
- The -d bin option tells the compiler to store the *.class files in the bin directory.
- The -cp option tells the compiler and JVM where to look for the required class files. (“cp” stands for “classpath”).
- Note the **“:”** to separate the directory names and jar files in the -cp option.

JUnit Tutorial: How to compile and run the test? Part I: From the command line

- Now I will change the last test to expect 1 instead of 0, so that it fails:

```
// Tests
assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
assertEquals("0 x 0 must be 1", 1, tester.multiply(0, 0));
```

- Now I recompile and run again, and I get this (see below).
- Note the error message in red (not red on the computer):

```
$ javac -d bin -cp lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar
mine/*.java
$ java -cp bin:lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar
mine.MyTestRunner
multiplicationOfZeroIntegersShouldReturnZero(mine.MyClassTest):
0 x 0 must be 1 expected:<1> but was:<0>
```

JUnit Annotations (@tags)

Annotation	Description
@Test	identifies a public void method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named Exception
@Before	identifies a method that is to be executed before **each** test.
@BeforeClass	identifies a method that is to be executed once, before the start of **all** tests. It must be public **static** void.
@After @AfterClass	Analogous to Before/BeforeClass
@Ignore	identifies a method to be skipped (it's broken, or not ready)

JUnit Assert methods

- JUnit provides static methods in its Assert class to test for certain conditions.
- These throw an AssertionException if the comparison test fails.

Statement	Description
fail(string)	Let the method fail.
assertTrue(message, boolean)	Checks that the boolean condition is true.
assertFalse(message, boolean)	Checks that the boolean condition is false.
assertEquals(message, expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the
assertEquals(message, expected, actual, toler)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull(message, object)	Checks that the object is null.
assertNotNull(message, object)	Checks that the object is NOT null.

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- Eclipse has built-in support for creating and running JUnit tests.
  - ✦you do not need to download and install the junit.jar files, at least not for the more recent versions of eclipse.
- For example, to create a JUnit test or a test class for an existing class,
  - ✦select this class in the Package Explorer view,
  - ✦right-click on it and select New → JUnit Test Case.
- To run a test,
  - ✦select the class which contains the tests,
  - ✦ right-click on it and select Run-as → JUnit Test. This starts JUnit and executes all test methods in this class.

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- I will do the following demo in class.
- Make a project for Assignment2, put the classes in the src folder, inside the assign2 package (drag+drop the assign2 folder).
- Make a new src folder called test (right-click on the project, select New → Source Folder)
- Right-click on Movie.java and select New → JUnit Test Case. Call it MovieTest and put it in the test folder.
  - ✦if you get “Warning JUnit 4 is not on the BuildPath…” say yes to add it.

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- I added the method testShippingCreditMovie to test the shipping credit of a Movie:

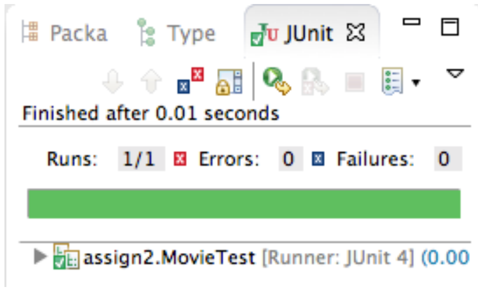
```
package assign2;
import static org.junit.Assert.*;
import org.junit.Test;

public class MovieTest {
   @Test
   public void testShippingCreditMovie() {

      Movie m = new Movie(3333,"Star Wars",33.50,3,"1234567899");
      assertEquals("movie shipping credit should be 2.98”,
                   2.98, m.shippingCredit(), .01);
   }
}
```

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- To run the test,
  - ✦select the MovieTest class (in the package explorer)
  - ✦ right-click on it and select Run-as → JUnit Test. This starts JUnit and executes all test methods in this class.
  - ✦Eclipse uses the JUnit view which shows the results of the tests

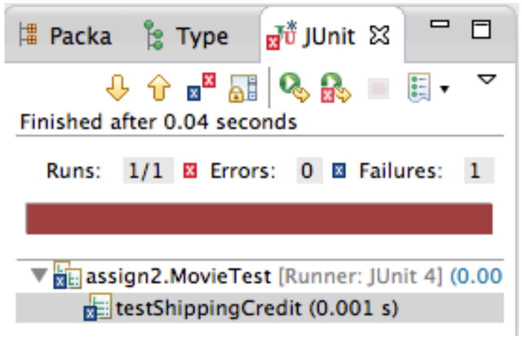


```
Finished after 0.01 seconds
Runs: 1/1   Errors: 0   Failures: 0
assign2.MovieTest [Runner: JUnit 4] (0.00
```

Green is good

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- To make the test fail,
  - ✦In the Movie.java class, change the value of the shipping credit to 2.88.
  - ✦Run the test again.



```
Finished after 0.04 seconds
Runs: 1/1   Errors: 0   Failures: 1
assign2.MovieTest [Runner: JUnit 4] (0.00
    testShippingCredit (0.001 s)
```

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- Add another class ToyTest.java with a method testShippingCreditToy:

```
@Test
public void testShippingCreditToy() {

    Toy t = new Toy(3344,"Monopoly",12.55,2,34);
    assertEquals("toy shipping credit should be ????”,
                    ????, t.shippingCredit(), .01);
}
```

JUnit Tutorial: How to compile and run the test? Part II: From within Eclipse

- Test the Inventory class (Inventory collaborates with Product):

```
public class InventoryTest {
    Inventory inv;    // member variable
    @Before
    public void setUp() throws Exception {    // occurs before each test
        inv = new Inventory();
        inv.addProduct(new Movie(5566,”Fargo",9.99,5,"1234567899"));
        inv.addProduct(new Movie(1122,”Jaws",5.99,17,"1112223334"));
        inv.addProduct(new Movie(8899,"Alien",6.50,12,"8888888888"));
    }
    @Test
    public void testRemove() {
        inv.removeProduct(5566);
        //now what????
    }
     @Test
    public void testProcessSale() {
        inv.processSale(5566, 3, 8.04);
       //now what????
    }
}
```

```
//Final definition of the two tests, and how I changed the Inventory:
    @Test
    public void testRemove() {
        inv.removeProduct(5566);
        //Note: Change searchList to be public.
        //if found, it returns the Product, if not it returns null
        Product p = inv.searchList(5566);
        assertNull("product was not removed:",p);
    }
    @Test
    public void testProcessSale() {
        //Note: Change process Sale to return an ArrayList of Double:
        //public ArrayList<Double> processSale(int sku, int quantitySold,
        //            double shippingCost) {
        //     ArrayList<Double> result = new ArrayList<>();
        //     .. inside of the else after the values are calculated:
        //     result.add(price);
        //     result.add(shippingCredit);
        //     result.add(commission);
        //     result.add(profit);
        //   and at the very end:
        //     return result;

        ArrayList<Double> result = inv.processSale(5566, 3, 8.04);
        assertEquals("Total Price: ",29.97,result.get(0),.01);
        assertEquals("Total Shipping Credit: ",8.94,result.get(1),.01);
        assertEquals("Total Commission: ",3.60,result.get(2),.01);
        assertEquals("Total Profit: ",27.27,result.get(3),.01);
    }
```