Guidelines for Class Design

Horstmann Chapter 3.4 and 3.5

CS 4354
Summer II 2016

Jill Seaman

Object-Oriented Design Continued:

- The previous chapter 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.
- 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 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

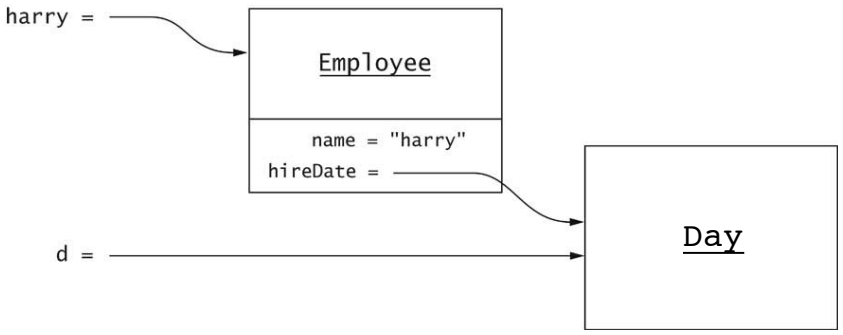
- No mutators, but Day is mutable:

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

- Remedy: use clone:

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


Sharing Mutable References



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 = (Day)hireDate.clone();
}
```

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 returning 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 returning 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)

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 abstraction
- 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 fixed this inconvenience

Clarity

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

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

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

```
// This isn't how it works, both calls are illegal
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 that was returned by next 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)
```