# Object-Oriented Design And Software Testing

In this section of notes you will learn about principles of good design as well how testing is an important part of good design



## Some Principles Of Good Design

1. Avoid going "method mad"
2. Keep an eye on your parameter lists
3. Minimize modifying immutable objects
4. Be cautious in the use of references
5. Be cautious when writing accessor and mutator methods

This list was partially derived from "Effective Java" by Joshua Bloch and is by no means complete. It is meant only as a starting point to get students thinking more about why a practice may be regarded as "good" or "bad" style.

## 1. Avoid Going Method Mad

- There should be a reason for each method
- Creating too many methods makes a class difficult to understand, use and maintain
- A good approach is to check for redundancies that exist between different methods



## 2. Keep An Eye On Your Parameter Lists

- Avoid long parameter lists
  - Rule of thumb: Three parameters is the maximum
- Avoid distinguishing overloaded methods solely by the order of the parameters

## 3. Minimize Modifying Immutable Objects

- Immutable objects
- Once instantiated they cannot change (all or nothing)

e.g., String s = "hello";
s = s + " there";



## 3. Minimize Modifying Immutable Objects (2)

- If you must make many changes consider substituting immutable objects with mutable ones

e.g.,

```
public class StringBuffer
{
  public StringBuffer (String str);
  public StringBuffer append (String str);
    :        :              :          :

}
```

For more information about this class

- http://java.sun.com/j2se/1.5.0/docs/api/java/lang/StringBuffer.html

## 3. Minimize Modifying Immutable Objects (3)

```
public class StringExample
{
    public static void main (String []
   args)
   {
      String s = "0";
      for (int i = 1; i < 100000; i++)
         s = s + i;
   }
}
```

```
public class StringBufferExample
{
    public static void main (String [] args)
    {
       StringBuffer s = new   StringBuffer("0");
       for (int i = 1; i < 100000; i++)
          s = s.append(i);
    }
}
```



## 4. Be Cautious In The Use Of References

- Similar pitfall to using global variables:

```
int i;

fun ()
{
   for (i = 0; i < 100; i++) { printf("foo"); }
}

main ():
{
   i = 10;
   fun ();
}
```

**With many programming languages (e.g., 'C' / 'C++') global variables can be accidentally changed anywhere after their declaration.**

## 4. Be Cautious In The Use Of References (2)

```
public class Foo
{
   private int num;
   public int getNum () { return num; }
   public void setNum (int newValue) { num = newValue; }
}
```



## 4. Be Cautious In The Use Of References (3)

```
public class Driver
{
   public static void main (String [] argv)
   {
      Foo f1, f2;
      f1 = new Foo ();
      f1.setNum(1);

      f2 = f1;
      f2.setNum(2);

      System.out.println(f1.getNum());
      System.out.println(f2.getNum());

   }
}
```

# 5. Be Cautious When Writing Accessor And Mutator Methods: First Version

```
public class Driver
{
   public static void main (String [] args)
   {
      CreditInfo newAccount = new CreditInfo (10, "James Tam");
      newAccount.setRating(0);
      System.out.println(newAccount);
   }
}
```



# 5. Be Cautious When Writing Accessor And Mutator Methods: First Version (2)

```
public class CreditInfo
{
   public static final int MIN = 0;
   public static final int MAX = 10;
   private int rating;
   private StringBuffer name;
   public CreditInfo ()
   {
      rating = 5;
      name = new StringBuffer("No name");
   }
   public CreditInfo (int newRating, String newName)
   {
      rating = newRating;
      name = new StringBuffer(newName);
   }

   public int getRating () { return rating;}
```

# 5. Be Cautious When Writing Accessor And Mutator Methods: First Version (3)

```
public void setRating (int newRating)
{
   if ((newRating >=  MIN) && (newRating <= MAX))
      rating = newRating;
}

public StringBuffer getName ()
{
   return name;
}

public void setName (String newName)
{
   name = new StringBuffer(newName);
}
```

•



# 5. Be Cautious When Writing Accessor And Mutator Methods: First Version (4)

```
 public String toString ()
{
   String s = new String ();
   s = s + "Name: ";
   if (name != null)
   {
      s = s + name.toString();
   }
   s = s + "\n";
   s = s + "Credit rating: " + rating + "\n";
   return s;
}
} // End of class CreditInfo
```

## 5. Be Cautious When Writing Accessor And Mutator Methods: Second Version

(All mutator methods now have private access).

```
public class Driver
{
   public static void main (String [] args)
   {
      CreditInfo newAccount = new CreditInfo (10, "James Tam");

      StringBuffer badGuyName;
      badGuyName = newAccount.getName();

      badGuyName.delete(0, badGuyName.length());
      badGuyName.append("Bad guy on the Internet");

      System.out.println(newAccount);
   }
}
```



## 5. Be Cautious When Writing Accessor And Mutator Methods: Second Version (2)

```
public class CreditInfo
{
   private int rating;
   private StringBuffer name;

   public CreditInfo ()
   {
      rating = 5;
      name = new StringBuffer("No name");
   }

   public CreditInfo (int newRating, String newName)
   {
      rating = newRating;
      name = new StringBuffer(newName);
   }
```

## 5. Be Cautious When Writing Accessor And Mutator Methods: Second Version (3)

```
 public int getRating ()
{
   return rating;
}
 private void setRating (int newRating)
{
   if ((newRating >= 0) && (newRating <= 10))
      rating = newRating;
}
public StringBuffer getName ()
{
   return name;
}
 private void setName (String newName)
{
   name = new StringBuffer(newName);
}
```



## 5. Be Cautious When Writing Accessor And Mutator Methods: Second Version (4)

```
 public String toString ()
{
   String s = new String ();
   s = s + "Name: ";
   if (name != null)
   {
      s = s + name.toString();
   }
   s = s + "\n";
   s = s + "Credit rating: " + rating + "\n";
   return s;
}
}
```

## 5. Be Cautious When Writing Accessor And Mutator Methods: Third Version

```
public class Driver
{
   public static void main (String [] args)
   {
      CreditInfo newAccount = new CreditInfo (10, "James Tam");
      String badGuyName;
      badGuyName = newAccount.getName();

      badGuyName = badGuyName.replaceAll("James Tam", "Bad guy on
                   the Internet");
      System.out.println(badGuyName + "\n");
      System.out.println(newAccount);
   }
}
```



## 5. Be Cautious When Writing Accessor And Mutator Methods: Third Version (2)

```
public class CreditInfo
{
   private int rating;
   private String name;
   public CreditInfo ()
   {
      rating = 5;
      name = "No name";
   }
   public CreditInfo (int newRating, String newName)
   {
      rating = newRating;
      name = newName;
   }
   public int getRating ()
   {
      return rating;
   }
```

## 5. Be Cautious When Writing Accessor And Mutator Methods: Third Version (3)

```
 private void setRating (int newRating)
{
   if ((newRating >= 0) && (newRating <= 10))
      rating = newRating;
}

public String getName ()
{
   return name;
}

private void setName (String newName)
{
   name = newName;
}
```



## 5. Be Cautious When Writing Accessor And Mutator Methods: Third Version (4)

```
 public String toString ()
{
   String s = new String ();
   s = s + "Name: ";
   if (name != null)
   {
      s = s + name;
   }
   s = s + "\n";
   s = s + "Credit rating: " + rating + "\n";
   return s;
}
}
```

## 5. Be Cautious When Writing Accessor And Mutator Methods

- When choosing a type for an attribute it comes down to tradeoffs, what are the advantages and disadvantages of using a particular type.
- In the previous examples:
  - –Using mutable types (e.g., StringBuffer) provides a speed advantage.
  - –Using immutable types (e.g., String) provides additional security



## Unit Testing

- Place objects in a know state
- Send a message to the object
- Compare the resulting state vs. the expected stated

http://www.junit.org/

http://junit.sourceforge.net/

## Testing Programs

- There are many available, some of them for free (and many commercial ones are available for free on a trial basis):
  - –JUnit: http://www.junit.org/ or http://junit.sourceforge.net/
  - –JTest: http://www.parasoft.com/



## How The Test Programs Work

- Some specialized testing classes must be downloaded.
- The location of the downloaded test classes must be either permanantly set for the operating system (the 'classpath' variable) or at runtime:

```
>javac -classpath . ; c:/junit-4.8.1.jar SubscriptionTest.java
```

Must explicitly include the current working directory

The compressed jar file containing the testing classes is located in the C drive. The name of the jar file is "junit-4.8.1.jar".

The name of the Driver class used to run the tests (the name of class being tested is "Subscription".

- Typically they test if a method's operations are correct via an assertion (assert if a method is correct: true/false).

## Example Program To Be Tested

- A program that calculates the monthly cost of a subscription.
- Given information: the total price of the submission and the number of months that the subscription will last.

Example from: http://code.google.com



## Class Subscription

```
public class Subscription
{
   private int price ; // subscription total price in euro-cent
   private int length ; // length of subscription in months
   public Subscription(int p, int n)
   {
      price = p ;
      length = n ;
   }
  public int pricePerMonth()
  {
      double r = price / (double) length ;
      return r ;
   }
   public void cancel()
  {
       length = 0 ;
  }
}
```

## Testing A Class

- A ‘Driver’ class must be written.
- The purpose of this class is to test the methods of another class.
- The testing will occur by creating an instance of the class to be tested and putting it into some known state.
- A message will be sent to the object to put it into another state.
- An assert statement is used to determine if the new state is correct.



## Setting Up The Driver Class

```
import org.junit.* ;
@ <name of the test case>
{
    <code for the test case, including an assert statement>
}
```

## Class SubscriptionTest

```
import org.junit.* ;
import static org.junit.Assert.* ;
public class SubscriptionTest
{
   @Test public void test_returnEuro()
   {
      System.out.println("Test if pricePerMonth returns Euro...") ;
      Subscription S = new Subscription(200,2) ;
      assertTrue(S.pricePerMonth() == 2.0) ;
   }

  @Test public void test_roundUp()
  {
     System.out.println("Test if pricePerMonth rounds up correctly...") ;
     Subscription S = new Subscription(200,3) ;
     assertTrue(S.pricePerMonth() == 0.67) ;
  }
}
```



## Compiling The Program

- Compile the Subscription class as you normally would.
- Compiling the Driver class:

  >javac -classpath . ; c:/junit-4.8.1.jar SubscriptionTest.java

- Running the test

  java -classpath .;c:/junit-4.8.1.jar org.junit.runner.JUnitCore Subscription

  Running the Junit test classes

  Name of the class to be tested

- Test results

  FAILURES!!!
  Tests run: 2,  Failures: 2

  Both assertions failed (evaluated to false)

## You Should Now Know

- Some general design principles
  - What constitutes a good or a bad design.
- How to automated testing programs such as JUnit can be used to test the methods in your classes.