

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

James Tam

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
6. Consider where you declare local variables

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.

James Tam

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

James Tam

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

James Tam

3. Minimize Modifying Immutable Objects

- Immutable objects
- Once instantiated they cannot change (all or nothing)
e.g., `String s = "hello";`
`s = s + " there";`

James Tam

3. Minimize Modifying Immutable Objects (2)

- If you must make changes substitute 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/1.4/docs/api/java/lang/StringBuffer.html>

James Tam

3. Minimize Modifying Immutable Objects (3)

```
class StringExample
{
    public static void main (String [] args)
    {
        String s = "0";
        for (int i = 1; i < 10000; i++)
            s = s + i;
    }
}
```

```
class StringBufferExample
{
    public static void main (String [] args)
    {
        StringBuffer s = new StringBuffer("0");
        for (int i = 1; i < 10000; i++)
            s = s.append(i);
    }
}
```

James Tam

4. Be Cautious In The Use Of References

Similar to global variables:

```
program globalExample (output);
```

```
var
```

```
    i : integer;
```

```
procedure proc;
```

```
begin
```

```
    for i:= 1 to 100 do;
```

```
end;
```

```
begin
```

```
    i := 10;
```

```
    proc;
```

```
end.
```

James Tam

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

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

James Tam

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

```
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());
    }
}
```

James Tam

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

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

James Tam

5. Be Cautious When Writing Accessor And Mutator Methods: First 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);
    }
    public int getRating ()
    {
        return rating;
    }
}
```

James Tam

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

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

public StringBuffer getName ()
{
    return name;
}

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

James Tam

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
```

James Tam

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

- (All mutator methods now have private access).

```
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);
    }
}
```

James Tam

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);
    }
}
```

James Tam

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);
}
```

James Tam

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;
}
}
```

James Tam

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

```
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);
    }
}
```

James Tam

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;
    }
}
```

James Tam

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;
}
```

James Tam

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;
}
}
```

James Tam

6. Consider Where You Declare Local Variables

- First Approach: Declare all local variables at the beginning of a method:

```
void methodName (..)
{
    int num;
    char ch;
    :
}
```

Advantage:

- Putting all variable declarations in one place makes them easy to find

James Tam

6. Consider Where You Declare Local Variables (2)

- Second Approach: declare local variables only as they are needed

```
void methodName (..)
{
    int num;
    num = 10;
    :
    for (int i = 0; i < 10; i++)
}
```

Advantage:

- For long methods it can be hard to remember the declaration if all variables are declared at the beginning
- Reducing the scope of a variable may reduce logic errors

James Tam

Object-Oriented Design And Testing

- Start by employing a top-down approach to design
 - Start by determining the candidate classes in the system
 - Outline a skeleton for candidate classes (methods are stubs)
- Implement each method one-at-a-time.
- Create test drivers for methods.
- Fix any bugs in these methods
- Add the working methods to the code for the class.

James Tam

Determine The Candidate Classes

Example:

A utility company provides three types of utilities:

1. Electricity:

Bill = No. of kilowatt hours used * \$0.01

2. Gas:

Bill = No. of gigajoules used * \$7.50

3. Water

a) Flat rate: \$10.00 + (square footage of dwelling * \$0.01)

b) Metered rate: \$1.00 * No. cubic of meters used

James Tam

Determine The Candidate Classes (2)

Some candidate classes

- ElectricityBill
- WaterBill
- GasBill

James Tam

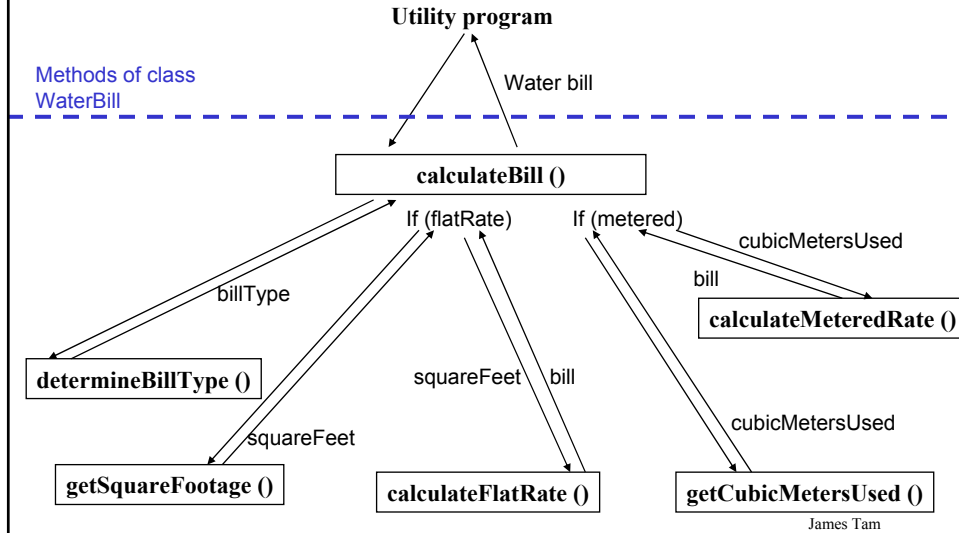
Skeleton For Class WaterBill

```
public class WaterBill
{
    private char billType;
    private double bill;
    public static final double RATE_PER_SQUARE_FOOT = 0.01;
    public static final double BASE_FLAT_RATE_VALUE = 10.0;
    public static final double RATE_PER_CUBIC_METER = 1.0;

    public WaterBill ()
    {
    }
    :      :      :
```

James Tam

Determining The Remaining Methods



Remaining Skeleton For Class WaterBill (2)

```
public double calculateBill () { return 1.0; }
public void determineBillType () { }
public int getSquareFootage () { return 1; }
public double calculateFlatRate (int squareFootage) { return 1.0; }
public double getCubicMetersUsed () { return 1.0; }
public double calculateMeteredRate (double cubicMetersUsed) { return 1.0; }

public char getBillType () { return billType; }
```

Implementing The Bodies For The Methods

1. calculateBill
2. determineBillType
3. getSquareFootage
4. calculateFlatRate
5. getCubicMetersUsed
6. calculateMeteredRate

James Tam

Body For Method CalculateBill

```
public double calculateBill ()
{
    int squareFootage;
    double cubicMetersUsed;
    determineBillType();
    if (billType == 'f')
    {
        squareFootage = getSquareFootage ();
        bill = calculateFlatRate (squareFootage);
    }
    else if (billType == 'm')
    {
        cubicMetersUsed = getCubicMetersUsed();
        bill = calculateMeteredRate (cubicMetersUsed);
    }
    else
        System.out.println("Bill must be either based on a flat rate or metered.");
    return bill;
}
```

James Tam

Body For DetermineBillType

```
public class WaterBill
{
    :      :      :      :
    public void determineBillType ()
    {
        System.out.println("Please indicate the method of billing.");
        System.out.println("(f)lat rate");
        System.out.println("(m)etered billing");
        billType = (char) Console.in.readChar();
        Console.in.readLine();
    }
}
```

James Tam

Creating A Driver To Test DetermineBillType

```
class Driver
{
    public static void main (String [] args)
    {
        WaterBill test = new WaterBill ();
        test.determineBillType ();
        bill = test.getBillType();
        System.out.println(bill);
    }
}
```

James Tam

Body For GetSquareFootage

```
public class WaterBill
{
    :    :    :    :
    public int getSquareFootage ()
    {
        int squareFootage;
        System.out.print("Enter square footage of dwelling: ");
        squareFootage = Console.in.readInt();
        Console.in.readChar();
        return squareFootage;
    }
}
```

James Tam

Creating A Driver To Test GetSquareFootage

```
class Driver
{
    public static void main (String [] args)
    {
        WaterBill test = new WaterBill ();
        int squareFootage = test.getSquareFootage ();
        System.out.println(squareFootage);
    }
}
```

James Tam

Body For CalculateFlatRate

```
public class WaterBill
{
    :    :    :    :
    public double calculateFlatRate (int squareFootage)
    {
        double total;
        total = BASE_FLAT_RATE_VALUE + (squareFootage *
            RATE_PER_SQUARE_FOOT);
        return total;
    }
}
```

James Tam

Creating A Driver For CalculateFlatRate

```
class DriverCalculateFlatRate
{
    public static void main (String [] args)
    {
        WaterBill test = new WaterBill ();
        double bill;
        int squareFootage;

        squareFootage = 0;
        bill = test.calculateFlatRate(squareFootage);
        if (bill != 10)
            System.out.println("Incorrect flat rate for 0 square feet");
        else
            System.out.println("Flat rate okay for 0 square feet");
    }
}
```

James Tam

Creating A Driver For CalculateFlatRate (2)

```
squareFootage = 1000;
bill = test.calculateFlatRate(squareFootage);
if (bill != 20)
    System.out.println("Incorrect flat rate for 1000 square feet");
else
    System.out.println("Flat rate okay for 1000 square feet");
}
} // End of Driver
```

James Tam

Body For GetCubicMetersUsed

```
public class WaterBill
{
    :   :   :   :
    public double getCubicMetersUsed ()
    {
        double cubicMetersUsed;
        System.out.print("Enter the number of cubic meters used: ");
        cubicMetersUsed = Console.in.readDouble();
        Console.in.readChar();
        return cubicMetersUsed;
    }
}
```

James Tam

Creating A Driver To Test GetCubicMetersUsed

```
class Driver
{
    public static void main (String [] args)
    {
        WaterBill test = new WaterBill ();
        double cubicMeters = test.getCubicMetersUsed ();
        System.out.println(cubicMeters);
    }
}
```

James Tam

Body For CalculateMeteredRate

```
public double calculateMeteredRate (double cubicMetersUsed)
{
    double total;
    total = cubicMetersUsed * RATE_PER_CUBIC_METER;
    return total;
}
```

James Tam

Driver For CalculateMeteredRate

```
class DriverCalculateMeteredRate
{
    public static void main (String [] args)
    {
        WaterBill water = new WaterBill ();
        double bill;
        double cubicMetersUsed;

        cubicMetersUsed = 0;
        bill = water.calculateMeteredRate(cubicMetersUsed);
        if (bill != 0 )
            System.out.println("Incorrect metered rate for 0 cubic meters consumed.");
        else
            System.out.println("Metered rate for 0 cubic meters consumed is okay.");
    }
}
```

James Tam

Driver For CalculateMeteredRate (2)

```
cubicMetersUsed = 100;
bill = water.calculateMeteredRate(cubicMetersUsed);
if (bill != 100 )
    System.out.println("Incorrect metered rate for 100 cubic meters
        consumed.");
else
    System.out.println("Metered rate for 100 cubic meters consumed is
        okay.");
}
```

James Tam

General Rule Of Thumb: Test Drivers

- Write a test driver class if you need to verify that a method does what it is supposed to do (determining if it is correct).
 - e.g., When a method performs a calculation, if a method is getting input
- Benefits of writing test drivers:
 1. Ensuring that you know precisely what your code is supposed to do.
 2. Making code more robust (test it before adding it a code library).

James Tam

You Should Now Know

- Some general design principles
 - What constitutes a good or a bad design.
- How to write test drives and what are the benefits of using test drivers in your programs

James Tam