CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 14 Assignment One Comments
Contents

References

Code Complete, McConnell, Microsoft Press, 1993

Refactoring: Improving the Design of Existing Code, Fowler, Addison Wesley Longman, 1999

Standard Issues
Consistency

There is a lot of flexibility in the coding styles

Be consistent

Example

In Java you can declare fields anywhere in the class

Most people put them either at the begging or end of a class

Pick one location and always do it.

Example

```
public void foo()
{
   code here
}

public void bar() { code here }

public void foobar() {
   code here
}
```

Code formatting

Indentation is important
Alignment is important

```
public abstract class HttpTransaction
 {
 HashMap headers = new HashMap();
 byte[] body;

     abstract void  parseFirstLine(ByteReader input ); q
 abstract String firstLineToString();
   public void fromStream( InputStream input) throws IOException
 {
  ByteReader bufferedInput = new ByteReader( input);

parseFirstLine(bufferedInput);
if ( hasHeaders() )
parseHeaders(bufferedInput);
if (hasBody() )
parseBody(bufferedInput);
}

     public byte[] toBytes()
    {
   if (!hasBody())
       return toString().getBytes();
      String headerString = toString();
     int length = headerString.length() + body.length;
      byte[]  transaction = new byte[length];
     System.arraycopy(headerString.getBytes(), 0, transaction, 0, headerString.length());
         System.arraycopy(body, 0, transaction, headerString.length(), body.length);
    return transaction;
         }

     public boolean hasContentLength()
         {
         return (headers.containsKey( "Content-length" ) ||
             headers.containsKey( "Content-Length" ));
         }
```

Line Wrap

Just say no to line wrap in source code

```
   public byte[] toBytes()
      {
      if (!hasBody())
         return toString().getBytes();         //get the string and
convert it to bytes then return it
      String headerString = toString();//convert to a string
      int length = headerString.length() +
 body.length;
      byte[]  transaction = new byte[length];//declare transaction
      System.arraycopy(headerString.getBytes(), 0,
transaction, 0, headerString.length());
      System.arraycopy(body, 0, transaction,
headerString.length(), body.length);
      return transaction;
      }
         {
         String requestLine = readLine(input);          // Read a
Line of input from the input
         while (requestLine.length() > 0 )
            {
            int separatorIndex = requestLine.indexOf(':');
            String name = requestLine.substring(0,
separatorIndex);
            String value =
requestLine.substring(separatorIndex + 1).trim();
            headers.put( name, value);
            requestLine = readLine(input);
```

What Causes Line wrap

End of line comments

```
String requestLine = readLine(input);          // Read a
Line of input from the input
```

Solution

- Do not use end of line comments

Long Variable Names

IfYouHaveReallyLongVariableNames they will cause line wrap

ReallyLongVariableNamesAreHardToRead

Solution

- Use short expressive name

Nesting and Line Wrap

Nesting code can cause line wrap

```
if ( blah ) {
   if (more blah) {
      foo;
   else {
      if ( bar ) {
         not bar
      else {
         if ( more bar ) {
            if (even more bar ) {
               if (still more bar ) {
                  if (even still more bar ) {
                     if ( bar until you can’t stand it ) {
                        if ( foo ) {
                           if ( more foo ) {
                              if ( still more foo ) {
                                    if ( you can understand the nesting level
here please do not work for me) {
                                    }
                              }
                           }
                        }
                     }
                  }
               }
            }
         }
      }
```

Nesting levels

Don't go deep.

Refactorings to reduce nesting levels
Replace Nested Conditional with Guard Clauses[1]

A method has conditional behavior that does not make clear the normal path of execution

Use guard clauses for all the special cases

```
double getPayAmount() {
   double result;
   if ( _isDead)
      result = deadAMount();
   else {
      if ( _isSeparated)
         result = separatedAmount();
      else {
         if (_isRetired)
            result = retiredAmount();
         else
            result = normalPayAmount();
      }
   }
}
```

```
double getPayAmount() {
   if ( _isDead)
      return deadAMount();
   if ( _isSeparated)
      return separatedAmount();
   if (_isRetired)
      return  retiredAmount();
   return normalPayAmount();
}
```

[1] Refactoring Text, pp. 250-254

Decompose Conditional[2]

You have a complicated conditional (if-then-else) statement

Extract methods from the condition, then part and else parts

Example[3]

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
   charge = quality * _winterRate + _winterServiceCharge;
else
   charge = quantity * _summerRate
```

becomes

```
if (notSummer( date))
   charge = winterCharge(quantity);
else
   charge = summerCharge(quantity);
```

[2] Refactoring Text, pp. 238-239
[3] Recall that "_" indicates an instance variable

Introduce Null Object[4]

You have repeated checks for a null value

Replace the null value with a null object

Example

```
void add(int key, String value)
   if (key < _key ) {
      if (right == null)
         left == new BinaryNode(key, value);
      else
         left .add( key, value);
      etc.
```

Use a NullNode object that has a pointer to its parent

The NullNode add(key,value) method adds a new subtree to its parent

Then we get:

```
void add(int key, String value)
   if (key < _key ) {
      left .add( key, value);
      etc.
```

[4] Refactoring Text, pp. 260-266

An Example

```
void add(int key, String value)
   if (key < _key ) {
      if (left == null)
         left = new BinaryNode(key, value);
      else
         left.add( key, value);
   else {
      if (key == _key )
         value = _value;
      else {
         if (key > _key ) {
            if ( right == null ) {
               right = new BinaryNode(key, value);
            else
               left.add( key, value);
         }
      }
   }
}
```

Applying the above ideas this becomes

```
void add(int key, String value)
   if (key < _key )
      retun left.add( key, value);
   if (key == _key )
      return value = _value;
   if (key > _key )
      return left.add( key, value);
}
```

Comments

"Comments are easier to write poorly than well, and comments can be more damaging than helpful"

What does this do?

```
for i := 1 to Num do
 MeetsCriteria[ i ] := True;
for  i := 1 to Num / 2  do begin
 j := i + i;
 while ( j <= Num ) do begin
  MeetsCriteria[ j ] := False;
  j := j + i;
 end;
for i := 1 to Mun do
 if MeetsCriteria[ i ] then
  writeln( i, ' meets criteria ' );
```

How many comments does this need?

```
for PrimeCandidate:= 1 to Num do
   IsPrime[ PrimeCandidate] := True;

for  Factor:= 1 to Num / 2  do begin
   FactorableNumber := Factor + Factor ;
   while ( FactorableNumber <= Num ) do begin
      IsPrime[ FactorableNumber ] := False;
      FactorableNumber := FactorableNumber + Factor ;
   end;
end;

for PrimeCandidate:= 1 to Num do
   if IsPrime[ PrimeCandidate] then
      writeln( PrimeCandidate, ' is Prime ' );
```

Good Programming Style is the Foundation of Well Commented Program

Kinds of Comments

- Repeat of the code

```
X := X + 1  /* add one to X

/** Integer value */
private int key;

/** String value */
private String value;
```

- Explanation of code
  Used to explain complicated or tricky code

```
*p++->*c = a

/* first we need to increase p by one, then ..
```

Make code simpler before commenting

```
(*(p++))->*c = a

ObjectPointerPointer++;
ObjectPointer = *ObjectPointerPointer;
ObjectPointer ->*DataMemberPointer = a;
```

- Marker in the code

```
/*  **** Need to add error checking here  **** */
```

- Summary of the code
  Distills a few lines of code into one or two sentences

- Description of the code's intent

  Explains the purpose of a section of code

```
{ get current employee information }   intent

{ update EmpRec structure }     what
```

Commenting Efficiently

- Use styles that are easy to maintain

```
/*************************************
 * module: Print                    *
 *                                  *
 * author: Roger Whitney            *
 * date:   Sept. 10, 1995           *
 *                                  *
 * blah blah blah                   *
 *                                  *
 *************************************/
```

```
/*************************************
  module: Print

  author: Roger Whitney
  date:   Sept. 10, 1995

  blah blah blah

 *************************************/
```

- Comment as you go along

Commenting Techniques
Commenting Individual Lines

Avoid self-indulgent comments

```
MOV AX,  723h        ;    R. I. P. L. V. B.
```

Endline comments have problems

```
MemToInit := MemoryAvailable(); { get memory available }
```

- Not much room for comment
- Must work to format the comment

Use endline comments on

- Data declarations
- Maintenance notes
- Mark ends of blocks

Commenting Paragraphs of Code

Write comments at the level of the code's intent

Comment the why rather than the how

Make every comment count

Document surprises

Avoid abbreviations

How verses Why

How

```
/* if allocation flag is zero */

if ( AllocFlag == 0 ) ...
```

Why

```
/* if allocating a new member */

if ( AllocFlag == 0 ) ...
```

Even Better

```
/* if allocating a new member */

if ( AllocFlag == NEW_MEMBER ) ...
```

Summary comment on How

```
{ check each character in "InputStr" until a
  dollar sign is found or all characters have
  been checked }

Done   := false;
MaxPos := Length( InputStr );
i      := 1;
while ( (not Done) and (i <= MaxLen) ) begin
   if ( InputStr[ i ] = '$' ) then
      Done := True
   else
      i := i + 1
end;
```

Summary comment on Intent

```
{ find the command-word terminator }

Done   := false;
MaxPos := Length( InputStr );
i      := 1;

while ( (not Done) and (i <= MaxPos ) ) begin
   if ( InputStr[ i ] = '$' ) then
      Done := True
   else
      i := i + 1
end;
```

Summary comment on Intent with Better Style

```
{ find the command-word terminator }

FoundTheEnd      := false;
MaxCommandLength := Length( InputStr );
Index            := 1;

while ((not FoundTheEnd) and
      (Index <= MaxCommandLength)) begin

   if ( InputStr[ Index ] = '$' ) then
      FoundTheEnd := True;
   else
      Index := Index + 1;
end;
```

Commenting Data Declarations

Comment the units of numeric data

Comment the range of allowable numeric values

Comment coded meanings

```
var
   CursorX:   1..MaxCols;       { horizontal screen position of cursor }
   CursorY:   1..MaxRows;       { vertical position of cursor on screen }

   AntennaLength:   Real;       { length of antenna in meters: >= 2 }
   SignalStrength: Integer;     { strength of signal in kilowatts: >= 1 }

   CharCode:  0..255;       { ASCII character code }
   CharAttib:  Integer;     { 0=Plain; 1=Italic; 2=Bold  }
   CharSize:   4..127;      { size of character in points }
```

Comment limitations on input data

Document flags to the bit level

Commenting Routines

Avoid Kitchen-Sink Routine Prologs

Keep comments close to the code they describe

Describe each routine in one or two sentences at the top of the routine

Document input and output variables where they are declared

Differentiate between input and output data

Document interface assumptions

Keep track of the routine's change history

Comment on the routine's limitation

Document the routine's global effects

Document the source of algorithms that are used

```
procedure InsertionSort
   {
   VarData:    SortArray;    { sort array elements }
      FirstElement: Integer          {index of first element to sort}
      LastElement: Integer           {index of last element to sort}
   }
```

Comments From Hell

```
//*****************************************************
//*****************************************************
// Class: Proxy
//*****************************************************
//*****************************************************
public class Proxy {
   //*****************************************************
   //Default Constructor
   //*****************************************************
   public Proxy() { blah }
   //*****************************************************
   //Constructor
   //*****************************************************
   public Proxy(int size) {
   _size = size;}
   //*****************************************************
   //Main
   //*****************************************************
   public static void public static void main( String[] args ) {
      //Create a new socket
      ServerSocket acceptor = new ServerSocket(0);
      //Print out the port number
      System.out.println("On port " + acceptor.getLocalPort());
      while (true) {//runs forever, until program is manually killed
         Socket client = acceptor.accept();//Accept a new connection
         processRequest(//Process the client request
            client.getInputStream(),//get the input
            client.getOutputStream());//get the ouput
         client.close();//Close the connection
```

Less is Better

```
public class Proxy
   {

   public Proxy()
      {
      blah
      }

   public Proxy(int size)
      {
      _size = size;
      }

   public static void public static void main( String[] args )
      {
      ServerSocket acceptor = new ServerSocket(0);
      System.out.println("On port " + acceptor.getLocalPort());

      //Must kill process to terminate
      while (true)
         {
         Socket client = acceptor.accept();
         processRequest(
            client.getInputStream(),
            client.getOutputStream());
         client.close();
         }
      }
   }
```

If you really need to separate Sections

```
//-------------------------------------------------------------

public class Proxy
   {

   public Proxy()
      {
      blah
      }

   public Proxy(int size)
      {
      _size = size;
      }

   public static void public static void main( String[] args )
      {
      ServerSocket acceptor = new ServerSocket(0);
      System.out.println("On port " + acceptor.getLocalPort());

      //Must kill process to terminate
      while (true) {
         Socket client = acceptor.accept();
         processRequest(
            client.getInputStream(),
            client.getOutputStream());
         client.close();
```

Names

Use the Java standard

thisIsTheJavaStandardNameStyleForMethodNamesAndVariables

ClassName

STATIC_FINAL

"Finding good names is the hardest part of OO Programming"

"Names should fully and accurately describe the entity the variable represents"

What role does the variable play in the program?

Examples in Assignment

s
r
l
temp
BSIterator
BST
stk

Coupling

```
public Interator interator(int order)
   {
   switch(order)
      {
      case PREORDER:
         return new PreorderInterator(this);
      case INORDER:
         return new InorderInterator(this);
      default:
         throw new IllegalArgumentExeption("No such order");
      }
   }
```

Verses:

```
public Interator preorderInterator()
   {
   return new PreorderInterator(this);
   }

public Interator inorderInterator()
   {
   return new InorderInterator(this);
   }
```

Repeated Fields

```
abstract class Visitor
   {
   Node root;
   abstract void visitTree(BinaryTree visitee);
   abstract void visitNode(Node visitee);
   }

class PreorderVisitor extends Visitor
   {
   Node root;
   blah
   etc.
   }
```

This is asking for trouble

Just say no to repeated fields

Indentation and Names Again

```
abstract class Visitor
   {
   abstract void visitTree(Node visitee);
   abstract void visitNode(Node visitee);
   }

class PreorderVisitor extends Visitor
{
   Node root;
   blah
   etc.
}
```

What are the issues?

More Flags

```
public class BinaryTreeIterator implements Iterator{
   public static final int PREORDER = 1;
   public static final int INORDER = 2;
   int type;

   public boolean hasNext() {
      blah
      if (PREORDER  == type)
         do the preorder thing
      else if (INORDER == type)
         do the inorder thing
      else
         now what?
      more blah
   }

   public Object next() {
      blah
      if (PREORDER  == type)
         do the preorder thing
      else if (INORDER == type)
         do the inorder thing
      else
         now what?
      more blah
   }
}
```

Not how every method becomes two methods
Use two classes and each method becomes simpler

Outline of Solution

Problem 1. A Binary Search Tree

Nearly everyone got this. So will not provide a solution.

Will assume three classes and one interface:

- BinarySearchTree
- Node (interface)
- BinaryNode
- NullNode

Problem 2. Iterators

Nearly everyone got this. So will not provide a solution.

Will assume two classes:

- PreorderIterator
- InorderIterator

Problem 3. Visitor
With traversal in Elements

```
class BinarySearchTree {
   public accept(Visitor visitor) {
      visitor.visitBinarySerachTree( this);
      root.accept( visitor);
   }
}

class BinaryNode{
   public accept(Visitor visitor) {
      visitor.visitBinaryNode( this);
      left.accept( visitor);
      right.accept( visitor);
   }
}

class NullNode{
   public accept(Visitor visitor) {
      visitor.visitNullNode( this);
   }
}
```

The Visitor

```
class Visitor {
   integer count = 0;

   public visitBinarySerachTree(BinarySerachTree tree) {
   }

   public visitBinaryNode(BinaryNode node) {
      count++;
      reverseValue( node);
   }

   public visitNullNode(BinaryNode node) {
      count++;
   }

   private void reverseValue(BinaryNode node) {
      put your code here
   }
}
```

With traversal in the Visitor

```
class BinarySearchTree {
   public accept(Visitor visitor) {
      visitor.visitBinarySerachTree( this);
   }
}

class BinaryNode{
   public accept(Visitor visitor) {
      visitor.visitBinaryNode( this);
   }
}

class NullNode{
   public accept(Visitor visitor) {
      visitor.visitNullNode( this);
   }
}
```

Performing the Traversal

```
BinarySearchTree studentNames;

Code here to populate the tree

Visitor reverseNames = new Visitor();
studentNames.accept( reverseNames  );
```

The Visitor

```
class Visitor {
   integer count = 0;

   public visitBinarySerachTree(BinarySerachTree tree) {
      Node root = tree.root();
      root.accept( this );
   }

   public visitBinaryNode(BinaryNode node) {
      count++;
      reverseValue( node);
      Node left = node.left();
      left.accept( this );
      Node right = node.right();
      right.accept( this );
   }

   public visitNullNode(BinaryNode node) {
      count++;
   }

   private void reverseValue(BinaryNode node) {
      put your code here
   }
}
```

Why accept?

Many of you did:

```
public visitBinaryNode(BinaryNode node) {
   count++;
   reverseValue( node);
   if ( node.left() != null );
      visitBinaryNode( node.left() );
   if ( node.right() != null );
      visitBinaryNode( node.right() );
}
```

The visitor pattern replaces case statements with polymorphism

```
public visitBinaryNode(BinaryNode node) {
   count++;
   reverseValue( node);
   Node left = node.left();
   left.accept( this );
   Node right = node.right();
   right.accept( this );
}
```

Problem 4 Strategy

```
class Visitor {
   integer count = 0;
   Iterator traversal;

   public Visitor(Interator order) {
      traversal = order;
   }

   public visitBinarySerachTree(BinarySerachTree tree) {
      order.on( tree );
      while ( order.hasNext() ) {
         Node next = (Node) order.next();
         next.accept( this );
      }
   }

   public visitBinaryNode(BinaryNode node) {
      count++;
      reverseValue( node);
   }

   public visitNullNode(BinaryNode node) {
      count++;
   }

   private void reverseValue(BinaryNode node) {
      blah
   }
}
```

Performing the Traversal

```
BinarySearchTree studentNames;

Code here to populate the tree

Visitor reverseNames = new Visitor( new InorderIterator() );
studentNames.accept( reverseNames  );
```