CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2012
Doc 4 Assignment 1 Comments
Jan 31, 2012

Comments in Code

Goal

Make the easier to read the code

Duh

```
Node newNode = new Node(value); //create new node

//main function
public static void main(String[ ] args) {
```

Duh Comments

Comments that repeat the code

```
//Node class
public class Node {
    public Node next; // pointer to next node
    public Node previous; // pointer to previous node
    String data; // node data

    //Constructor
     public Node() {
        next = null;
        previous = null;
        data = '';
    }
}
```

Duh Comments

```
public class Node {
    public Node next;
    public Node previous;
    String data;

    public Node() {
        next = null;
        previous = null;
        data = '';
    }
}
```

Easier to read with out the comments

Duh Comments

Duh Comments make it harder to read the code

Duh Comments

Exercise:

How many of your comments just repeat the code?

Comments in Code

Comment the why not the what

If what you are doing is hard to understand

- Rewrite code to make it easier to understand
  - Use better names
  - Add methods
  - Simplify algorithm
  - Use guard statements

If still hard to understand add comments

Commenting on What

```
public String getItem((int index) throws Exception {
    //Check validity of index
    if( index >=0 || index < mlength) {
        //Traverse to find node in the index location
        Node current = null;
        int k = 0;
        for (current = first; k < index && current != null && current.next() != null;
                                current = current.next() {
            k++;
    }
    else {
        throw new Exception("out of bound index");
    }
    // just to help me debug
    if (k != index || current == null)
        throw new Exception("Linked List broken");
    return current.value();
}
```

Use methods to reduce comments

```
public String getItem((int index) throws Exception {
    Node nodeAtIndex;
    if( isIndexValid(index)) {
        nodeAtIndex = getNodeAtIndex(index);
    }
    else {
        throw new Exception("out of bound index");
    }
    // just to help me debug
    if (nodeAtIndex == null)
        throw new Exception("Linked List broken");
    return current.value();
}
```

Extract Method to replace comment

```
if (this.head == null && this.tail == null ) { //case 1 - empty list
    this.head = newNode;
    this.tail = newNode;
} else {
    etc.
```

```
if (isEmpty() ) {
    this.head = newNode;
    this.tail = newNode;
} else {
    etc.
```

```
public boolean isEmpty() {
    return (this.head == null && this.tail == null;
}
```


Read Extract method pp 110 -116. Page 110 is basically this example.

Fowler - Extract Method

"I look at a method that is too long or look at code that needs a comment to understand its purpose.

I then turn that fragment of code into its own method."

Use Guard to handle special cases

```
public String getItem((int index) throws Exception {
    Node nodeAtIndex;
    if( isIndexValid(index)) {
        nodeAtIndex = getNodeAtIndex(index);
    }
    else {
        throw new Exception("out of bound index");
    }
    // just to help me debug
    if (nodeAtIndex == null)
        throw new Exception("Linked List broken");
    return current.value();
}
```

With Guard

```
public String getItem((int index) throws Exception {
    if( isIndexInvalid(index)) {
        throw new IndexOutOfBoundsException("Index " + index +
                    " is not between 0 and " + this.size() );
    }
    Node nodeAtIndex = getNodeAtIndex(index);
    return current.value();
}
```

Let getNodeAtIndex throw Exception if node is null (for debuging)

Guard

Start method by handle special cases

Often can exit method when done with special case

Rest of body need not deal with special case

Complex

```
public void addElement(String itemAdded) {
        Node newNode = new Node(itemAdded);
        Node traverse = first;
        Node temp;
        if ( first == null ) { //new node is first node in list
                last = newNode;
                first = newNode;
                listCount++;
        } else {
                for (int i = 0; i < listCount; i++ ) {
                        if (traverse ! = null) {
                                temp = traverse;
                                traverse = traverse.next;
                        } else {
                                return;
                        }
                        if ((newNode.data).compareTo(temp.data)>0) {
                                if (temp.previious == null) {
                                        //adds node at beginning
                                        addsBeg((temp, newNode);
                                        traverse = first;
                                        return;
                                } else {
                                        newNode.previous = temp.previous;
                                        newNde.next = temp.previous.next;
                                        temp.previous = newNode;
                                        listCount++;
                                        return;
                                }
                        } else {
                                if(temp.next == null) {
                                        //add node at end
                                        addEnd(temp, newNode);
                                        traverse = last;
                                        return;
                                } else {
                                        if (temp.next != null)
                                        temp = temp.next;
                                        continue;
                                }
                        }
                }
        }
}
```

Complex - 1

Close up

```
public void addElement(String itemAdded) {
    Node newNode = new Node(itemAdded);
    Node traverse = first;
    Node temp;
    if ( first == null ) { //new node is first node in list
        last = newNode;
        first = newNode;
        listCount++;
    } else {
        for (int i = 0; i < listCount; i++ ) {
            if (traverse ! = null) {
                temp = traverse;
                traverse = traverse.next;
            } else {
                return;
            }
            if ((newNode.data).compareTo(temp.data)>0) {
                if (temp.previous == null) {
                    //adds node at beginning
                    addsBeg((temp, newNode);
                    traverse = first;
                    return;
  17        } else {
```

Complex - 2

```
} else {
    newNode.previous = temp.previous;
    newNde.next = temp.previous.next;
    temp.previous = newNode;
    listCount++;
    return;
}
} else {
    if(temp.next == null) {
        //add node at end
        addEnd(temp, newNode);
        traverse = last;
        return;
    } else {
        if (temp.next != null)
        temp = temp.next;
        continue;
    }
}
}
}
}
```

Complex

Guards & methods simplify code

```
public void addElement(String itemAdded) {
    Node newNode = new Node(itemAdded);
    Node traverse = first;
    Node temp;
    if ( isEmpty() ) {
        return addToEmptyList(newNode);
    }
    for (int i = 0; i < listCount; i++ ) {
        if (traverse  = null) { return;}
        temp = traverse;
        traverse = traverse.next;
        if ((newNode.data).compareTo(temp.data)>0) {
            if (temp.previous == null) {
                return addsNodeAtBegining((temp, newNode);
            }
            temp.addAfter(newNode);
            listCount++;
            return;
            }
        } else {
            if(temp.next == null) {
                return addNodeAtEnd(temp, newNode);
            }
            if (temp.next != null)
                temp = temp.next;
            }
        }
    }
}
```

Complex

Rename

traverse = next

temp = current

```
public void addElement(String itemAdded) {
    Node newNode = new Node(itemAdded);
    Node next = first;
    Node current;
    if ( isEmpty() ) {
        return addToEmptyList(newNode);
    }
    for (int i = 0; i < listCount; i++ ) {
        if (next  == null) { return;}
        current = next;
        next = next.next;
        if ((newNode.data).compareTo(current.data)>0) {
            if (current.previous == null) {
                return addsNodeAtBegining((current, newNode);
            }
            current.append(newNode);
            listCount++;
            return;
            }
        } else {
            if(current.next == null) {
                return addNodeAtEnd(current, newNode);
            }
            if (current.next != null)
                current = current.next;
            }
        }
    }
}
```

Complex

More guards

```
public void addElement(String itemAdded) {
    if ( isEmpty() )
        return addToEmptyList(itemAdded);
    if (isFirstElement(itemAdded) )
        return addFirst(itemAdded);
    if (isLastElement(itemAdded) )
        return addLast(itemAdded);

    Node next = first;
    Node current;
    for (int i = 0; i < listCount; i++ ) {
        current = next;
        next = next.next;
        if (current.belongsBefore(itemAdded) {
            current.append(itemAdded);
            listCount++;
            return;
            }
        }
    }
}
```


The 3 guards now mean that we will know inside the loop that we will never add at the end of the list.

Complex

make loop understandable

```
public void addElement(String itemAdded) {
    if ( isEmpty() )
        return addToEmptyList(itemAdded);
    if (isFirstElement(itemAdded) )
        return addFirst(itemAdded);
    if (isLastElement(itemAdded) )
        return addLast(itemAdded);

    Node next = first;
    Node current;
    while (current != null ) {
        current = next;
        next = next.next;
        if (current.belongsBefore(itemAdded) {
            current.append(itemAdded);
            listCount++;
            return;
            }
        }
    }
}
```


The 3 guards now mean that we will know inside the loop that we will never add at the end of the list.

Are comments useful?

```
/**
 * @param filter: if the user wants to use some filters
 * @param direction: helps to identify which way in the list we are heading
 * @return: An array of matched strings
 * @throws Exception
 */

public ArrayList<String> getItems(String filter, int direction) throws Exception {
    bunch of code
}
```

Are comments useful?

```
/**
 *
 * @param filter: characters to search for
 * @param searchDirection: 0 = forward, 1 = reverse
 * @return: An array of strings that contain characters in filter
 * @throws Exception if direction invalid
 */

public ArrayList<String> getItems(String filter, int searchDirection) throws Exception {
    bunch of code
}
```

What should the name be in Java?

Why does it matter?

```
//Return a string representation of the linked list
pubic String print() {
    blah
    blah
    return result;
}
```

What should the name be in Java?

Why does it matter?

```
pubic String getItemAt(int index) throws OutOfBoundsException {
    blah
    blah
}
```

What should the name be in Java?

Why does it matter?

```
pubic void addToList(String item) {
    blah
    blah
}
```

What should the return type be in Java?

Why does it matter?

```
pubic void addToList(String item) {
    blah
    blah
}
```

What should the Parent class be?

Why does it matter?

```
public class LinkedList extends(or implements) XXX {

}
```

Polymorphism

If your linked list class has correct parent

- Method names become standard
- Can interchange with other collection classes

Issue?

```
public Node getItemAt(int index) { blah }
```

Information Hiding

Physical

Give access to internal data to outside world

```
public Node getItemAt(int index) { blah }

public void addNode(Node item) { blah}
```

Logical

Provide information about internal workings to outside world

```
public void addNode(String item ) { blah }

public void bubbleSort() { blah }
```

Where are the operations?

```
public class Node {
    private String value;
    private Node next, previous;

    public Node() {
        value = next = previous =  null;
    }

    public String getValue() { etc}
    public Node getNext() { etc}
    public Node getNext() { etc}
    public setValue(String aValue) {etc}
    public setNext(Node aNode) {etc}
    public setPrevious(Node aNode) {etc}
```

Heuristics

Keep related data and behavior in one place

A class should capture one and only one key abstraction

Heuristics

Beware of classes that have many accessor methods defined in their public interface

Do not create god classes/objects in your system

Beware of classes that have too much noncommunicating behavior

Where are the operations?

```
public class Node {
    private Node next;
    private Node previous;
    private String data;

    public void append(Node nextNode) {
        nextNode.previous = this;
        nextNode.next = this.next;
        this.next = nextNode;
    }

    public void append(String item) {
        append(new Node(item));
    }
}
```


Yes there is something missing here

Node & LinkedList Work together

Let Node handle links

Let LinkedList handle finding correct location

Helper methods

Methods that only act on the arguments

```
public class LinkedList {

    private void addAfter(Node a, Node b) {
        b.previous = a;
        b.next = a.next;
        a.next = b;
        b.next.previous = b;
    }

    private boolean containsAOrO(String value) {
        blah
    }
```

Heuristics

Keep related data and behavior in one place

Helper methods are a sign that you are not keeping data and behavoir in one place

Java, Helper Methods & OO

Can't add methods to classes in Java API

So can't put methods with operations for those classes (String)

So don't think about it in easy cases

So don't do in in hard cases either

So we are used to separating operations from data

But since it is Java must be OO right?

```
private boolean containsAOrO(String value) {
    blah
}
```

What is the abstraction?

```
public class LinkedList {

    public ArrayList getStringsWithAOrO() { blah }

    public ArrayList getEvenLenghtStrings() { blah }

    public boolean add(String item) { blah }
```

Class Name

public class LL { stuff removed}

public class Asgnt1StringLinkedList<String { stuff removed}

Java Names

```
public void Add(String item) { }

public void Insert(String item) { }
```

I am lazy and don't care who knows

```
public void remove(String item ) {
    // TODO Auto-generated method stub
    code to remove item
```

System.out.println

```
public void getEvenStrings() {
    blah
    blah
    while (nextNode != startNode) {
        if (blah blah) {
            System.out.println(nextNode.getData());
```

System.out

Good for debugging

When you don't have a good debugger

Good for Unix/Linux commands

Java is too big to use In this case

Returning strings

```
public String getEvenStrings() {
    String result;
    blah
    while (nextNode != startNode) {
        if (blah blah) {
            blah
    return result;
}
```

Returning Collection

```
public ArrayList<String> getEvenStrings() {
    ArrayList<String> result;
    blah
    while (nextNode != startNode) {
        if (blah blah) {
            blah
    return result;
}
```

Exceptions

What exception is thrown in get() method in Java's

- Vector
- ArrayList
- LinkedList
- etc.

Why is it important to do the same?

Long Names

```
public void insertStringInLexicographicalOrderInList(String value) {
    etc.
}

public void itTurnsOutThatNamesCanBeTooLongUseAShorterNameWhenItExists()



public class OrderedLinkedList extends LinkedList {

    public boolean add(String value) {
    etc.
```

Temporary Field

```
public class LinkedList {
    private Node head;
    private Node tail;
    private Node current; //when inserting a node

    public Node add(String value) {
        current = head;
        while (current != tail) {
            etc.
    }
```

Why waste time with Menu systems?

```
public void Menu() {
    while(true) {
        System.out.println( "Please select your choice.");
        System.out.println( "1. display all node in the list");
        System.out.println( "2. Display strings starting with vowels");
        System.out.println( "3. Display odd length strings back to front");
        etc
```

list1, list2

```
public void evenString() {
    Node evenNode = tail;
    List<String> list1 = new ArrayList<String>();
    //Used for traversing and putting all the data from linked list to list1
    while(evenNode != null)  {
        list1.add(evenNode.data);
        evenNode = evenNode.prev;
    }
    Iterator it = list1.iterator():
    List<String> list2 = new ArrayList<String>();
    //This checks if the string is of even lenght
    while(it.hasNext()) {
        String value = (String) it.next();
        if (value.length() % 2 == 0) list2.add(value);
    }
    Iterator it2 = list2.iterator():
    while (it2.hasNext()) {
        System.out.println("Even " + it2.next());
    }
}
```