Name: Aldrick Gardiner

Course: COP 4331 003

Date: November 15th 2015

Homework 4

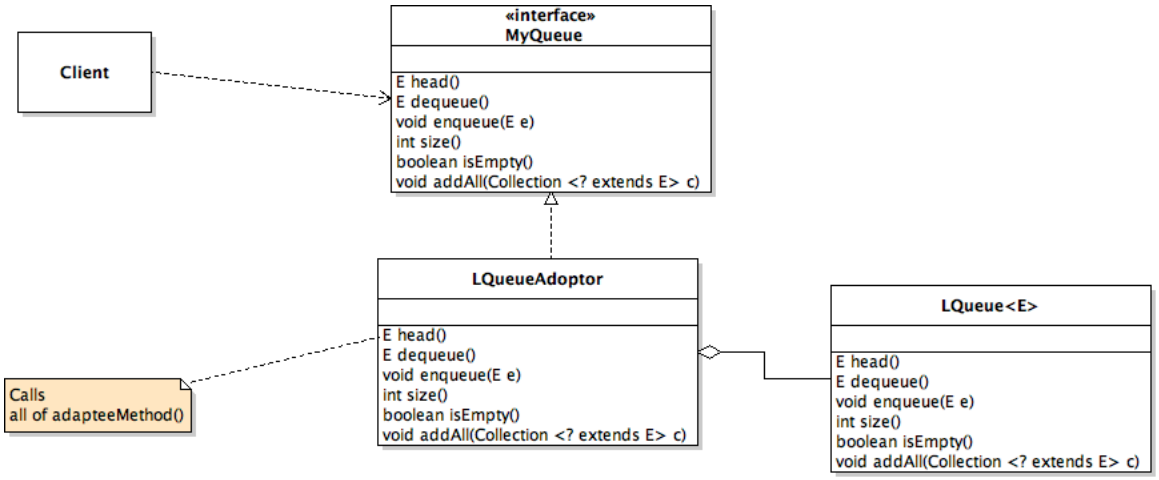
Question 5.1

Consider the following generic queue interface:

```
interface MyQueue <E> {
   // return the top of the queue element or throw
NoSuchElementException if empty
   E head();

   // remove and return the top of the queue element or
throw NoSuchElementException if empty
   E dequeue();

   // add an element to the queue
   void enqueue(E e);

   // returns the size of the queue
   int size();

   // returns true if the queue is empty
   boolean isEmpty();

   // add elements to this queue from a collection c of
E references:
   void addAll(Collection <? extends E> c);
}
```

a) Use the Adapter pattern to design a generic queue class called LQueue<E> that implements interface MyQueue<E> and that uses a LinkedList<E> object to stores the queue elements.
Write the UML class diagram for the pattern.

Identify the pattern roles (e.g. "adapter", "adaptee") with UML notes on the class diagram.



Implement class LQueue<E> in Java.
Write correct contracts for each method. (Notice that NoSuchElementException is not a checked exception, but you could put it in the function signature.)

filename: LQueue<E>.java

```
import java.util.Collection;
import java.util.*;

/**
 *
 * @author Ace
 * @param <E>
 */
public class LQueue<E> implements MyQueue<E>
{
    private Node<E> front;
    private Node<E> tail;
    private int size;

    public LQueue()
    {
        front = null;
        tail = null;
    }
```

```java
@Override
public E head() {
    return (E) front.data;
}

@Override
public E dequeue() {
    E element = front.data;
    front = front.next;
    if (front == null)
        tail = null;
    size--;
    return element;
}

@Override
public void enqueue(E e) {
    Node<E> newNode= new Node<E>(e, null);
    if (front == null)
    {
        front = newNode;
    }
    else {
        tail.next = newNode;
    }
        tail = newNode;
        size++;

}

@Override
public int size() {
    return size;
}

@Override
public boolean isEmpty() {
    return (front == null);
}

@Override
public void addAll(Collection<? extends E> c) {

    for (int i = 0; i <= c.size()-1; i++) {
        enqueue((E) c);
    }

}
```

```
    private static class Node<E>
    {
        private E data;
        private Node<E> next;
        public Node(E element, Node<E> nextNode)
        {
            data = element;
            next = nextNode;
            //size =
        }

    }

}
```

filename: MyQueue<E>.java

```
import java.util.Collection;

/**
 *
 * @author Ace
 */
interface MyQueue <E> {

    // return the top of the queue element or throw
NoSuchElementException if empty
   E head();

   // remove and return the top of the queue element or throw
NoSuchElementException if empty
   E dequeue();

   // add an element to the queue
   void enqueue(E e);

   // returns the size of the queue
   int size();

   // returns true if the queue is empty
   boolean isEmpty();

   // add elements to this queue from a collection c of E
references:
   void addAll(Collection <? extends E> c);

}
```

c) Write a class QueueTest with a main(...) method that tests all 6 methods from class LQueue<E>.
There is no need of junit unit tests. Use hardcoded objects.

Filename: QueueTest.java

```
import java.util.ArrayList;

/**
 *
 * @author Ace
 */
public class QueueTest {

   public static void main( String args[] )
   {
      LQueue queue = new LQueue();
      // create an empty array list1 with an initial capacity
      ArrayList<Integer> array = new ArrayList<Integer>(5);
      array.add(12);
      array.add(20);
      array.add(45);
      array.add(1);
      array.add(2);



      // use enqueue method
      System.out.print("Is queue empty: " +queue.isEmpty() +
"\n");
      System.out.print("The size of queue: " +queue.size() +
"\n");
      queue.enqueue( -1 );
      queue.enqueue( 0 );
      queue.enqueue( 1 );
      queue.enqueue( 5 );
      System.out.print("The head of queue: " +queue.head() +
"\n");
      System.out.print("Is queue empty: " +queue.isEmpty() +
"\n");
      System.out.print("The size of queue: " +queue.size() +
"\n");
      System.out.print("Element deleted: " + queue.dequeue() +
"\n");
      System.out.print("The size of queue: " +queue.size() +
```

```
"\n");
      System.out.print("The head of queue: " +queue.head() +
"\n");
      queue.addAll(array);
      System.out.print("The size of queue: " +queue.size() +
"\n");


   }

}
```

Question 10.2
Use the Singleton pattern for a new Java class Stdout you need to write
that can have just one instance. A programmer can use the instance
to print text lines (String objects) to the terminal with the method printline():

```
    public class Stdout {
       public void printline(String s) {
          ...   // print s to System.out
       }
       ...      // fill in the dots
    }
```

Make sure the pattern is implemented correctly (e.g. just one instance can be created).
Notice that the printline() function is NOT static.

Write a main(....) method to test your code.

Filename:Stdout.java

```
import static java.lang.System.out;

/**
 *
 * @author Ace
 */
public class Stdout {

    private static Stdout instance = new Stdout( );

   /* A private Constructor prevents any other
```

```
     * class from instantiating.
     */
    private Stdout(){ }

    /* Static 'instance' method */
        public static Stdout getInstance()
     {
         if(instance == null)
         {
             instance = new Stdout();
         }
         return instance;
     }

     public void printline(String s) {
         // print s to System.out
         out.println(s);
        }

}
```

Filename:StdoutTester.java

```
public class StdoutTester {

     public static void main(String[] args) {
         Stdout tmp = Stdout.getInstance( );
         tmp.printline("Hello!!");
        }
}
```

10.3

a) The Decorator and Proxy patterns have an almost identical structure as defined by their class diagram. What are the differences between these two patterns ?

The difference between these two patterns are Decorator Pattern focuses on dynamically adding functions to an object, while Proxy Pattern focuses on controlling access to an object. With Proxy Pattern, the proxy class can hide the detail information of an object from its client. Therefore, when using Proxy Pattern, we usually create an instance of object inside the proxy class. And when using Decorator Pattern, we typically pass the original object as a parameter to the constructor of the decorator. With the Proxy pattern, the relationship between a proxy and the real subject is typically set at compile time, whereas decorators can be recursively constructed at runtime.

b) Explain why class MouseMotionAdapter from the Swing

library is not an adapter class in the sense of the Adapter design pattern.

The MouseMotionAdapter is not an adapter class in the sense of the Adapter design pattern, because the Adapter design pattern that allows the interface of an existing class to be used from another interface.[1] It's used to make existing classes work with others without modifying their source code whereas the MouseMotionAdapter is an abstract adapter class for receiving mouse motion events. All methods of this class are empty. This class is convenience class for creating listener objects.

Question 7.1.

a. Implement a generic class Pair<K,V> that stores pairs of (key, value) pairs.
Usage example:

```
    Pair<Integer,String> p = new Pair<>(1, "one");
    System.out.println("The string value of " + p.k()
+ " is " + p.v());
```

A pair object has a constructor

```
    public Pair(K k, V v) {...}
```

has accessors:

```
    public K k() {...}
    public V v() {...}
```

and overrides these methods inherited from class Object:

```
    public boolean equals(Object obj) {...}
    public int hashCode() {...}
    public String toString() {...}
    public Object clone() {...}
```

Your class must be immutable, serializable, and cloneable. It is OK to return a shallow clone. Follow the guidelines from the textbook when implementing the above functions.

b. Write a method PairTest class with method main() where you test:

- equality with equals()
- cloning : make a clone and compare with the original
- serialization. Serialize to an ObjectStream backed by a file and then load the object
  back to memory and compare it to the original with equals().
- hashCode(). Must be compatible with equals (i.e. equal hashCode() for equal objects, and
  different values (with high probability) for different key or/and value objects

Pair.java

```
public class Pair<K,V> implements Cloneable {

    public Pair(K aKey,V aValue)
    {
        key   = aKey;
        value = aValue;

    }

    public K k()
    {
        return key;
    }
    public V v()
    {
        return value;
    }

    @Override
    public boolean equals(Object obj)
    {
        return this == obj;
    }
    @Override
    public int hashCode()
    {
        return(key == null ? 0 : key.hashCode()) ^ (value == null
? 0 : value.hashCode());
    }
    @Override
    public String toString()
```

```
    {
        return getClass().getName()
               + "[key=" + key
               + ",value=" + value
               + "]";
    }

    @Override
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }



    private final K key;
    private final V value;

}
```

PairTest.java

```
public class PairTest {

    public static void main(String[] args) throws
CloneNotSupportedException, IOException
    {

        Pair<Integer,String> p = new Pair<>(1, "one");
        Pair<Integer,String> cloned = (Pair) p.clone();
        System.out.println(cloned.toString());
        System.out.println("The string value of " + p.k() + " is
" + p.v());
        System.out.println(p.equals(p));
        System.out.println(p.toString());
        System.out.println(p.hashCode());

      }

    }
```

a) Write a generic method

```
      public static <...> Collection<Pair<...>>
sortPairCollection(Collection <Pair<....>> col)
```

in a Utils class that takes as parameter a collection of Pair<K,V> objects
and returns a new collection object (use ArrayList<...>) with the pair elements from collection col
sorted in ascending order. For comparing pairs, the K type must implement Comparable<....>
Use the proper type constraints.

b)
Write a main() function in Utils.java that tests the sortPairCollection() with K=String and V=Integer.

Utils.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import javafx.util.Pair;

/**
 *
 * @author Ace
 */
public class Utils {

    public static <K extends Comparable<K>,V>
Collection<Pair<K,V>> sortPairCollection(Collection <Pair<K,V>>
col)
    {
        ArrayList<Pair<K,V>> arr = new ArrayList<>();
        arr.addAll(col);
        Collections.sort(arr,new Comparator<Pair<K,V>>()
        {

            @Override
            public int compare(Pair<K,V> o1, Pair<K,V> o2) {
                return o1.getKey().compareTo(o2.getKey());
```

```
            }

        });
        return (Collection) arr;
    }


     public static void main(String[] args)
    {
        ArrayList<Pair<String,Integer>> arr = new ArrayList<>();
        arr.add(new Pair("Cars",20));
        arr.add(new Pair("Houses",10));
        arr.add(new Pair("Yacht",30));

    // Sort the Items with their Comparable interface methods.
    ArrayList<Pair<String,Integer>> arr2 = (ArrayList)
Utils.sortPairCollection(arr);

    // Display our results.
    for (Pair p2 : arr2)
        {
        System.out.println(p2.toString());
    }


    }

 }
```