1) [15%] Recall the simple *singly-linked List* convention:

interface ListI {
    abstract public void setItem(Item c);
    abstract public void setNext(List l);
    abstract public Item getItem();
    abstract public List getNext();
    abstract public int getLength();
}

Abstract public class List implements ListI, and EmptyList and Node extend List. Node has only one constructor: Node(Item i, List next) while EmptyList has only the default constructor. Start with:

```java
List l = new EmptyList();
Item i1, i2; // assume each is assigned a non-null instance of Item
```

and write code to add nodes to l, such that the following are true:

```java
l.getLength() == 2
l.getItem().equals(item1)
l.getNext().getItem().equals(item2)
```
2) [35%] Given that same ListI interface:

```java
interface ListI {
    abstract public void setItem(Item c);
    abstract public void setNext(List l);
    abstract public Item getItem();
    abstract public List getNext();
    abstract public int  getLength();
}
```

Use the **State Pattern** to create a single concrete class **List** that is user friendly when the list is empty. Example code would be:

```java
Item i = new Thing(); // some Item to later be added to the list
List l = new List();   // it starts off acting like an empty list
l.getItem();          // prints "empty!" and returns null
l.getNext();          // prints "empty!" and returns null
l.getLength();        // returns zero.
```

But now, note that this same List instance `l` can become **non-empty**:

```java
l.setItem(item1);      // wow! it now acts like a node holding an Item
l.getItem();           // and it can now return item1;
l.getLength();         // now returns 1.
l.setItem(null);       // now acts like an empty list again, and ...
l.getLength();         // returns 0 again.
```

There are two kinds of **ListState**, either an **EmptyState** or a **NodeState**.

2a) [10%] Create a **UML class diagram** of **List**, the **ListState** hierarchy, and show the “is-a” and “has-a” relationships.
2b) [10%] Write the List class (later you will write the associated ListState classes; but you need to use instances of them here):

final public class List implements ListI {

2c) [15%] Finally, write the classes of the **ListState** hierarchy, starting with **ListState**:
3) [25%]

a) [15%] Add the **Observer-Observable pattern** to the State Pattern version of List, so that when a List changes between EmptyState and NodeState (in either direction) all observers will be signaled. Provide the necessary changes below.

b) [10%] Finish the code for ListChangeDetector so it prints out “list is non-empty” when a given List changes from EmptyState to NodeState.

```java
public class ListChangeDetector implements Observer {
```
4) [25%] Using the usual Visitor Pattern.

a) [5%] First write the additions to List to permit a ListVisitor to visit with a List.

b) [10%] Then write the abstract class ListVisitor (note that List will be the only visitee, but there would be any number of different subclasses of ListVisitor).

c) [10%] Finally, write PrintV, which extends ListVisitor, such that:

```java
ListVisitor v = new PrintV();
List l = new List();
l.acceptVisitor(v);

then for every Item i in l, v prints out i.toString().
```