1. [20%] A **Counter** is set to a given int value, then used to count up (or down) by one (1) each time **step** is called. Counting up versus down is selected by **setCountingUp()** versus **setCountingDown()**. Consider for example:

   ```java
   Counter c = new Counter();
   c.set(10);           // set the counter to start at 10
   c.setCountingDown(); // step will count down from the value 10
   c.step();            // now the value of the counter is 9
   ```

   You’ll implement this over the next two questions. First consider whether to use the Strategy Pattern or the State Pattern.

   1a. [5%] Of the two patterns, which is the better suited for the Counter class? Why?

   **The State Pattern is a better match because the Counter can be in one or the other state and shift according to which setter is called. Both could work, and other than identifiers they are virtually identical.**

   1b. [5%] Explain what the other pattern (of the two under discussion) would be better suited for.

   **Strategy Pattern is better for method re-use across a hierarchy. It is not primarily intended for the strategies (algorithms) to be swapped around during runtime.**

   1c. [10%] With your choice of pattern for Counter, create a UML class diagram that provides names of classes and methods that will be used (whether public, private or protected, and references among the classes/interfaces.
2a. [20%] Recall the intended usage of the **Counter** class:

```
counter c = new counter();
c.set(10);           // set the counter to start at some value, e.g. 10
  c.setCountingDown(); // step will count down from the value 10
  c.step();            // now the value of the counter is 9
```

Complete **Counter** consistent with your design for question 1c.

```java
public class Counter {
    private int n;
    private State currentState;
    private State ascendingState;
    private State descendingState;

    public Counter() {
        ascendingState = new AscendingState();
        descendingState = new DescendingState();
        setCountingUp();
    }

    public void set(int n) { this.n = n; } // start off at the count at n
    public int get() { return n; } // returns the current counter value

    public void setCountingUp() { currentState = ascendingState; }

    public void setCountingDown() { currentState = descendingState; }

    public void step() { n = currentState.step(n); }
}

public interface State {
    public int step(int n);
}

public class AscendingState implements State {
    public int step(int n) { return n + 1; }
}

public class DescendingState implements State {
    public int step(int n) { return n - 1; }
}
```
3. [25%] Now design and build an **Alarm** that is triggered when a Counter reaches a set number. It simply prints “ALARM:” plus that number. For example, in the following, two Alarms are set, one if and when Counter c (from before), reaches the value 5 and another is triggered if it reaches 50:

```
Alarm a1 = new Alarm(c); // the same Counter is passed as an argument
Alarm a2 = new Alarm(c); // to two instances of Alarm
a1.set(5);               // a1 set to trigger if Counter c reaches 5
a2.set(50);              // while a2 set to trigger if c reaches 50
```

```
c.set(0);                 // set up the Counter to increase from 0
c.setCountingUp():
for (int i = 0; i < 100; i++)
  c.step();
```

This would result in the following two distinct printouts:

```
ALARM: 5
ALARM: 50
```

3a [10%] Describe **all changes** to Counter to accommodate this design, including replacement code for the method step() from question 1.

**Counter should extend Observable.**

```
public void step() {
    currentState.step();
    setChanged();
    notifyObservers();
}
```

3b [15%] Finish writing Alarm below:

```
public class Alarm implements Observer {
    private int v;

    public Alarm(Counter c) {
        c.addObserver(this);
    }

    public void set(v) { this.v = v; }

    public void update(Observable o, Object obj) {
        Counter c = (Counter)o; // presume o is guaranteed to be a Counter
        if (c.get() == v)
            System.out.println("ALARM " + v);
    }
}
```
4a. [10%] Write `Visitor`, the base class (or interface) for a hierarchy of
visitors to `Alarm` and `Counter` (only those two classes).

```java
abstract public class Visitor {
    abstract public void visit(Alarm a);
    abstract public void visit(Counter c);
}
```
or

```java
public interface Visitor {
    public void visit(Alarm a);
    public void visit(Counter c);
}
```

4b. [5%] Write the method to be added to `Counter` regarding the Visitor Pattern. How does it differ from the one added to `Alarm`? Be specific.

```java
public void acceptVisitor(Visitor v) { v.visit(this); } // Counter
public void acceptVisitor(Visitor v) { v.visit(this); } // Alarm
```

Specific difference: NOTHING, ABSOLUTELY NOTHING!!!

4c. [10%] Given (non-null) instances of a `Visitor` in `v`, and a `Counter` in `c`, provide the Java code to get one to visit the other according to the Visitor Pattern:

```java
c.acceptVisitor(v);
```

Complete the following (UML-style) “sequence diagram” to show the steps involved in the visitation:

```
driver | c:Counter | v:Visitor
<--------------------------|-- acceptVisitor(v)------>
| | | -------visit(this)-------->
| | | something happens
| | |<---------------------------|
| |<--------------------------|
```
5a. [5%] The Decorator Pattern usually has a reference to the base class. For instance, if A1 is a subclass of Decorator for class A:

```java
public class A {
    ...
}

public class Decorator extends A {
    protected A decorated;

    public Decorator(A decorated) { this.decorated = decorated; }
    ...
}

public class A1 extends Decorator {
    ...
}
```

Now, if one wants to decorate an instance of an A to become an A1, what would the code look like? Start with:

```java
A a = new A();
a = new A1(a);
```

5b. [5%] What is the primary limitation of the use of wrappers (such as you can see in the code above) in trying to dynamically decorate an object?

You need to reference the wrapper, not the original. The stuffing idea gets around that, as mentioned in class. Also, if you add methods in the decorator, or its subclasses, which are not in the base class, you need to explicitly cast to get access to them.