1. [25%] **Guy** is a subclass of Person. Person has `String` method `talk`. A **Surfer** adds “, dude” after any phrase, and a **Californian** adds “like,” before any phrase. The following uses the Decorator Pattern:

```java
Person g = new Guy();
Person s = new Surfer(g);
Person c = new Californian(g);
Person cs = new Californian(s);
Person sc = new Surfer(c);

System.out.println(g.talk("hey"));   // prints "hey"
System.out.println(s.talk("hey"));   // prints "hey, dude"
System.out.println(c.talk("hey"));   // prints "like, hey"
System.out.println(cs.talk("hey"));  // prints "like, hey, dude"
```

1a [5%] what does `System.out.println(sc.talk("hey"));` print?
// prints "like, hey, dude"

1b [20%] finish the following using the Decorator Pattern:

```java
abstract public class Person {
    public String talk(String phrase) { return phrase; }
}

public class Guy extends Person {}

abstract public class Decorator{
    protected Person decorated;

    public Decorator(Person decorated) {
        this.decorated = decorated;
    }
    public abstract String talk(String phrase);
}

public class Surfer extends Decorator{
    public Surfer(Person p) {
        super(p);
    }

    public String talk(String p) {
        return decorated.talk(p) + ", dude";
    }
}

public class Californian extends Decorator{
    public Californian(Person p) {
        super(p);
    }

    public String talk(String p) {
        return "like, " + decorated.talk(p);
    }
}
```

---

1 For international students: a ‘surfer’ is a person who uses a surfboard for recreation at the beach. Like many sub-sets of society, surfers have their own vocabulary and call people “dude” (at least, according to the stereotype).
2. [25%] Instead, let’s use the **Strategy Pattern** and **not** the decorator, so that the usage would be as follows:

```java
Person g  = new Guy();
Person s  = new Surfer();
Person c  = new Californian();

System.out.println(g.talk("hey"));  // prints "hey"
System.out.println(s.talk("hey"));  // prints "hey, dude"
System.out.println(c.talk("hey"));  // prints "like, hey"
```

2a) [10%] And **without** using the Decorator pattern, how would you add a Californian Surfer (that would say both “hey” and “dude”)? Or in the future, add a Young Republican, or (gasp) a Young Republican Californian Surfer, etc.? Use a simple UML class diagram to explain.

Each kind of Person would be a concrete subclass. A CalifornianSurfer, a YoungRepublican, or (gasp) a YoungRepublicanCalifornianSurfer would each be a subclass. There could be some careful reuse where some TalkingStrategies call other TalkingStrategies (and using string concatenation), but it strongly violates preferring “programming to interfaces”.

For instance, the CalifornianSurferTalkStrategy could be defined to have method `talk`

```java
public String talk(String phrase) {
    return "like, " + SurferTalkStrategy.talk(phrase);
}
```

2b) [15%] Finish writing **Person** and **Surfer** using the Strategy Pattern. Be sure to add the specific strategy class needed for Surfer, dude.

```java
public interface TalkingStrategy { public String talk(String phrase); }  

abstract public class Person {
    protected TalkingStrategy ts;

    public String talk(String phrase) { return ts.talk(phrase); }  
}

public class Surfer extends Person {
    public Surfer() { ts = new SurferTalkingStrategy(); }  
}

public class SurferTalkingStrategy implements TalkingStrategy {
    public String talk(String phrase) { return phrase + ", dude"; }  
}  
```
3. [30%] Now for the **Observer Pattern**. Recall that the method `talk()` returns a `String`; it does not call `System.out.println` directly. But now, modify the design so that when any `Person` talks, *anyone observing* will directly comment on it (using `System.err.println`). Specifically, a surfer will print “totally, dude” and a Californian will print out “OMG!”.

So the following driver code:

```java
    g.addObserver(s);  // g is just some Guy, s is a Surfer
    g.addObserver(c);  // c is a Californian
    System.out.println(g.talk("CIS 211 makes my brain hurt"));
```

results in the following printouts:

```
CIS 211 makes my brain hurt
"totally, dude"
"OMG!"
```

and that’s the end of that conversation. **Write (and describe/UML diagram) all changes to the code of question 1 or 2 to create this behavior using the Observer Pattern.** Recall the `Observable` class has methods: public void `addObserver(Observer o)`, protected void `setChanged()` and void `notifyObservers(Object arg)` and the `Observer` interface has only public void `update(Observable o, Object obj)`.

1) *Person* would extend `Observable`
2) *Person* would implement `Observer`

*Person’s* `talk` method would be modified to `setChanged` and `notifyObservers`. This is not a perfect solution, for `talk` actually only returns a `String`; it does not call `System.out.println()`. This weakness in the design could have been fixed by having `talk` actually print, of course. In the Decorator version:

```java
    public String talk(String phrase) {
        return phrase;
    }
```

would become:

```java
    public String talk(String phrase) {
        setChanged();
        notifyObservers();
        return phrase;
    }
```

(and similarly in the Strategy Pattern version, but one must be careful to only have one call to `setChanged()` and `notifyObservers()` otherwise, dude, you would have multiple “totally, dude” and “OMG!” interspersed, like, totally throughout the conversation, like, OMG!)

And for instance the update method for Surfer would be:

```java
    public void update(Observable o, Object obj) {
        System.err.println("totally, dude");
    }
```
4. [20%] The **List** class we have used has methods: **add**, **get**, **iterator** (which returns an instance of an iterator), two versions of **remove**, and **size**. The **Iterator** class has methods **hasNext**, **next** and **remove**.

4a [10%] What is the design purpose (advantages) of the **Iterator**? Is it largely redundant if you have a List?

*It separates the potentially error-prone act of enumerating the elements of a list, and removing elements, from the creating and adding to a list. You need not be the owner of the list to be able to enumerate its elements. It is a bit of a curiosity that Iterator can remove but not add, which is an asymmetry that is in the Java API itself. Since the List class does not have the hasNext and next operations, Iterator is not redundant, but it is not necessary either. It is a convenience class that helps distinguish between creating, adding to, and a few other operations (size, removing by index and item) on a list from its enumeration.*

4b [5%] If you have List l with size greater than zero, and you create

```java
Iterator i = l.iterator();
```

what method would you use to return the very first element of the list L, if, as mentioned, the **Iterator** class has only methods **hasNext**, **next** and **remove**?

```java
i.next(); // like, strange method name instead of just get(), dude
```

4c [5%] Finish writing a **while** loop that iterates the list l. Within the loop, just call the toString() method on each element of the list.

```java
Iterator iterator = l.iterator();
while (iterator.hasNext())
    iterator.next().toString();
```

*Obviously this has no externally observable effect without System.out.println, or passing the string to some variable, but I was just trying to see if people were comfortable with the basic mechanics of iteration.*