1. [30%] **Abstract class A** has only one method: `void m()`. There are four subclasses: **A1, A2, A3, and A4**. Regarding method `m`, A1 and A2 perform algorithm `mVersion1()` and A3 use `mVersion2()`, while A4 uses a third version, `mVersion3()`. Using the **Strategy Pattern**, first:

   1a [10%] Draw a complete UML class diagram showing all classes/interfaces and their relationships to one another.

   1b [20%] Write Java classes for A and A1 in their entirety plus whatever other classes/interfaces needed to make them compile and run (Don’t worry about A2, A3, and A4 and their associated versions of m).

```java
abstract public class A {
    protected MStrategy ms;

    public void m() { ms.m(); }
}

public interface MStrategy { public void m(); }

public class MStrategy1 implements MStrategy {
    public void m() { mVersion1(); }
}

public class A1 extends A {
    ms = new MStrategy1();
}
```
Bees observe Flowers, and when a Flower blooms, the Bee will want to visit it. Complete the driver, Bee and Flower code, using the Java-based Observer Pattern API. Remember to call setChanged before notifyObservers.

```java
Bee b = new Bee("Billy"); // each bee has a name
Flower f = new Flower();
// add any driver code involving b and f here:

f.addObserver(b); // 2
f.bloom(); // the flower blooms and the bee should consequently visit
```

```java
public class Flower extends Observable { [2]

    public void visit(Bee b) {
        System.out.println("I'm being visited by " + b.getName());
    }

    public void bloom() {
        setChanged();
        notifyObservers(); [4]
    }
}
```

```java
public class Bee extends Insect implements Observer { [2]

    String name;

    public Bee(String name) {
        this.name = name; [2 for the constructor and argument]
    }

    public String getName() { return name; }

    public void update(Observable o, Object obj) { [4]
        if (o instanceof Flower) { [2 for check and casting]
            Flower f = (Flower)o;
            f.visit(this); [2]
        }
    }
}
```
3. [25%] **Hummingbird** extends **Bird** (while **Bee** extends **Insect**; they have no common base class other than **Object**).

   Hummingbird h = new Hummingbird(“Hannah”);

3a. [10%] What changes would you make to the driver, and **Flower** code to allow a Hummingbird to visit a Flower promptly after it blooms?

   The driver makes a Hummingbird and tells the Flower to add it as an observer.

   Then either make a polymorphic visit method for Hummingbird:
   ```java
   public void visit(Hummingbird h) {
       System.out.println(“I’m being visited by ” + h.getName());
   }
   ```

   or create a new interface:
   ```java
   public interface FlowerVisitor {
       public String getName();
   }
   ```

   and then have only the one visit method for this new interface:
   ```java
   public void visit(FlowerVisitor v) {
       System.out.println(“I’m being visited by ” + v.getName());
   }
   ```

3b. [15%] Looking at the larger picture, how could the **Strategy Pattern** help? Presume for this simulation that Hummingbirds and Bees eat by **FlowerVisiting**, (while other kinds of Insect and Bird have other strategies for how they eat, such as **FruitEating**). Using UML and code snippets, describe how you would revise the design of **Flower**, **Bee**, and Hummmingbird to incorporate a common **FlowerVisiting** strategy (use a reverse page as necessary).

   A Bee or Hummingbird still needs to be an Observer, so it gets an update method call when the Flower blooms. Update needs to be in Hummingbird or Bee.

   The strategy of FlowerVisiting amounts to just calling the visit method in Flower, which is not much of a strategy. But you could at least bundle that into a single method eatByVisiting, and that could be the one method in this strategy.
4. [25%] Use the **Decorator Pattern** to create combinations of additional options for an abstract **Car** class. The standard subclasses of Car are **Sedan** ($15,000 base price) and **Convertible** ($20,000). For additional costs, you can then add **AirConditioned** ($2000 extra) or **SportsPackage** ($5000 extra), among others. A Car has an **int getPrice()** method that reflects the total price. Sketch the UML class diagram to show the pattern, then write **Car**, **Sedan**, the **Decorator** class/interface, and **AirConditioned**.

```java
public abstract class Car {
    protected int price;

    public int getPrice() { return price; }
}

public Sedan extends Car {
    public Sedan () { price = 15000; }
}

public abstract class Decorator extends Car {
    protected Car decorated;

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

public AirConditioned extends Decorator {
    public AirConditioned(Car decorated) {
        super(decorated);
        price = 2000;

        public int getPrice() {
            return price + decorated.getPrice(); // either access the price directly,
            // return Car.getPrice() + decorated.getPrice(); // or use the getter
        }
    }
}
```