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).
2. [20%] 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 setChange 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.bloom(); // the flower blooms and the bee should consequently visit
```

```java
public class Flower

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

    public void bloom() {
    }

public class Bee extends Insect
```
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?

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 **Hummingbird** to incorporate a common **FlowerVisiting** strategy (use a reverse page as necessary).
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**.