1. Ferrari and Audi extend the abstract notion of a Car. For a Ferrari, the method int getMaxSpeed() returns 175 and for Audi it returns 150. Different options are available which affect the maximum speed. For instance the Turbo option adds 10 and the Spyder (convertible) option subtracts 10. The following uses the Decorator Pattern:

```java
Car f1 = new Ferrari();
Car f2 = new Turbo(f1);
Car f3 = new Spyder(f2);  // now we have a Turbo Spyder Ferrari
```

1a) [5%] what does f3.getMaxSpeed() return? Explain your answer.

It would return 175, basic speed of 175 plus 10 (Turbo wrapper) but then minus 10 (due to Spyder wrapper).

1b) [5%] After the above three lines of java are executed, is the Ferrari instance that f2 references now a convertible (instanceof Spyder)? Explain.

No, because it’s been wrapped by a Spyder, but it still points to the Turbo.

1c) [15%] Draw a UML class diagram of the above (careful: the Decorator Pattern design requires at least six classes/interfaces) and make sure you show how the parts interrelate.
1d) [10%] Finish the implementation using the **Decorator Pattern**:

```java
abstract public class Car extends Observable {
    abstract public int getMaxSpeed();
}

public class Ferrari extends Car {
    public int getMaxSpeed() { return 175; }
}

abstract public class Decorator extends Car {
    protected Car decorated;

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

public class Turbo extends Decorator {
    public Turbo(Car c) { super(c); }

    public int getMaxSpeed() { return decorated.getMaxSpeed() + 10; }
}

public class Spyder extends Decorator {
    public Spyder(Car c) { super(c); }

    public int getMaxSpeed() { return decorated.getMaxSpeed() - 10; }
}
```

1e) [10%] What’s an alternative to using the Decorator Pattern that would allow creating all these combinations of variations (e.g., a Ferrari with both Turbo and Spyder options, or an Audi with only Turbo)? What if we also wanted to add some additional options in the future that further affected the maximum speed (e.g., Nitro or, Hybrid) and apply that to all the various types of cars? Is there any advantage or disadvantage to this method?
2) Cars use three forms of **FuelStrategy**: Electric, Gas, and Hybrid. The **Strategy Pattern** will be used to implement `public String getEngineType()`, which returns “electricity”, “petrol”, or “hybrid”. For example,

```java
Car c = new Ferrari();
System.err.println(c.getEngineType()); // prints “petrol”.
```

2a) [10%] While the entire hierarchy consists of **FuelStrategy**, Electric, Gas, and Hybrid, just write **FuelStrategy** and **Gas**:

```java
public interface FuelStrategy {
    public String getEngineType();
}

class Gas implements FuelStrategy {
    public String getEngineType() { return “petrol”; }
}
```

2b) [15%] While adding this method to the hierarchy affects many parts of the code, only indicate only the additional lines of code needed for **Car** and **Ferrari** to use the Strategy Pattern so that the above code would work. Don’t copy the other aspects such as getMaxSpeed().

```java
abstract public class Car {
    protected FuelStrategy fuelStrategy;

    public String getEngineType() {
        return fuelStrategy.getEngineType();
    }
}

public class Ferrari extends Car {
    public Ferrari() {
        fuelStrategy = new Gas();
    }
}
```
3.) For the Observer Pattern, we’ll use Java’s Observer and Observable.java. Presume that Car implements Vehicle and extends Observable:

```java
public interface Vehicle {
    public void setSpeed(int speed);  // adjust this Car’s speed
    public int getSpeed();            // returns its current speed
    public void receiveSpeedingTicket(); // bad news
}
```

Here is the following driver code (the observe method is discussed later):

```java
PoliceCar p = new PoliceCar();  // PoliceCar extends Car.
Ferrari   f = new Ferrari();

p.observe(f);   // this is what Pooooolice do, they observe others
```

3a) [5%] Whenever the Ferrari’s setSpeed method is called, any observers (including the police or those gawking kids in the van in the slow lane) are notified. Finish setSpeed (just deal with the Observer Pattern aspects):

```java
public void setSpeed(int speed) {
    currentSpeed = speed;  // store the new speed
    setChanged();
    notifyObservers();
}
```

3b) [5%] PoliceCar is a subclass of Car. Finish writing observe (hint: remember all Cars are Observable):

```java
public void observe(Car c) {
    c.addObserver(this);
}
```

3c) [10%] For any Car c observed by PoliceCar p, whenever c changes speed (by setSpeed), p will check, using c.getSpeed(), and if it is faster than 65 (MPH), c will receiveSpeedingTicket. Add this functionality to PoliceCar (first decide what method you need to write):

```java
public void update(Observable o, Object obj) {
    if (o instanceof Car) {
        Car c = (Car)o;
        if (c.getSpeed() > 65)
            c.receiveSpeedingTicket();
    }
}
```
3d) [10%] Complete the **UML Sequence Diagram** for the following, assuming p is a PoliceCar and f is a Ferrari.

```java
p.observe(f);
f.setSpeed(100);
```