Cats and Roses accept Visitors (such as Rain, Sunshine, and Grandma). Let’s start with Cats. A Cat is either Happy or Unhappy. Rain makes it Unhappy; Sunshine makes it Happy. When a Grandma visits a Cat, she prints the string returned by the Cat’s toString method (either a “purr” or “hiss”).

```java
Cat cat = new Cat();  // initially in the Happy state
Grandma granny = new Grandma();
Rain rainy = new Rain();
Sunshine sunny = new Sunshine();

cat.acceptVisitor(rainy);  // rain makes the cat unhappy
cat.acceptVisitor(granny);  // when grandma visits, the cat gives a “hiss”
cat.acceptVisitor(sunny);   // here comes the sun!
cat.acceptVisitor(granny);  // and now grandma receives a “purr”
```

In the next few questions, the class Cat will be written using the State Pattern and the pure Visitor pattern.

1. [15%] First, finish writing Cat. Recall that Grandma calls toString(), which is state-dependent. Since the visitors can change the cat’s state, public setters setHappy and setUnhappy are provided. You’ll write the states in question 2; for now just finish Cat.

```java
public class Cat {

    public Cat() {

        public void setHappy() {

            public void setUnhappy() {

                public String toString() {

                    public void acceptVisitor(Visitor v) {
```
2. [15%] Write the CatState hierarchy for the Cat class, starting with the base class. Recall that only the Cat’s toString method is state-dependent, returning “purr” or “hiss”. Use the pure State pattern (no instanceof or conditionals).

3. [10%] Write the abstract base Visitor as an interface (not as an abstract class). The only classes visited (i.e., “visitees” would be Cat and Rose).

Now for roses. While Rain and Sunshine just change a Cat’s mood, it is essential for Roses to bloom. A Rose is initially in a Growing state. Only after a Rose is visited by both Rain and Sunshine (in either order) is it in the state of Blooming. A Grandma System.out.println’s the string returned from the toString() of any Rose she visits. If the Rose is Blooming that string is ”I’m blooming!” otherwise ”I’m still growing”.

Grandma  granny = new Grandma();
Rain      rainy  = new Rain();
Sunshine  sunny  = new Sunshine();
Rose      rosy   = new Rose();

rosy.acceptVisitor(granny);       // results in printing ”I’m still growing”
rosy.acceptVisitor(rainy);
rosy.acceptVisitor(sunny);
rosy.acceptVisitor(granny);       // results in printing ”I’m blooming”
4. [15%] Next finish writing Rose. It starts in the Growing, and ends up in the Blooming. It enters the Blooming only if it has received both water and light, such as from Rain and Sunshine visitors. Visitors are not necessary, by the way; you could make a rose bloom by just two steps:

Rose rosy = new Rose();
rosy.receiveWater();
rosy.receiveLight();

Write Rose using the pure state pattern. There is an elegant solution using two additional RoseStates (call them GotEnoughWater and GotEnoughLight) such that no conditionals or ifs or booleans are needed.

public class Rose {
    private RoseState       currentState;
    private GotEnoughWater  enoughWater;
    private GotEnoughLight  enoughLight;
    private Blooming        blooming;

    public Rose() {
        currentState = new Growing(this);
        enoughWater  = new gotEnoughWater(this);
        enoughLight  = new gotEnoughLight(this);
        blooming     = new Blooming(this);
    }

    public void receiveWater() {
        // Implementation
    }

    public void receiveLight() {
        // Implementation
    }

    public String toString() {
        // Implementation
    }

    public void setState(RoseState s) { currentState = s; }

    public void acceptVisitor(Visitor v) {
        // Implementation
    }
}
5. [25%] Finish the RoseState hierarchy. There are three methods that are state dependent: `toString`, `receiveWater`, and `receiveLight()`.

```java
abstract public class RoseState {
    abstract public RoseState(Rose parent)
}

public class GrowingState {
    public GrowingState(Rose r) {
        public String toString() { return "I’m growing"; }
        public void receiveWater() {

            public void receiveLight() {

        }

    }
}

public class GotEnoughLight {
    public GotEnoughLight(Rose r) {
        public String toString() { return "I’m growing"; }
        public void receiveWater() {

            public void receiveLight() {

        }

    }
}

public class GotEnoughWater {
    public GotEnoughWater(Rose r) {
        public String toString() { return "I’m growing"; }
        public void receiveWater() {

            public void receiveLight() {

        }

    }
}

public class Blooming {
    public Blooming(Rose r) {
        public String toString() { return "I’m Blooming"; }
        public void receiveWater() {

            public void receiveLight() {

        }

    }
}
```
6. [10%] Write the Visitor **Rain** (no conditionals or booleans, or instanceof):

```java
public class Rain
```

7. [10%] Write the Visitor **Grandma** (no conditionals or booleans, or instanceof):

```java
public class Grandma
```
Extra credit or oops-I-understand-it-generally-but-haven’t-coded-yet credit]

8a) [10] Describe the State Pattern (and compare it to the Strategy Pattern). Perhaps make UML class diagrams for the example in this quiz and explain the benefits of the State Pattern for this problem.

8b) [10] Similarly, describe the Visitor Pattern in general, and specifically with regard to this quiz. What would some other applications of this pattern be for this problem area (Roses, Cats, Grandmas, Rain, etc.).