1. A **Dog** has two public methods **bark()** and **wagTail()**:

   ```java
   Dog fido = new Dog();
fido.bark(); // results in "woof" being printed
   ```

There is also a state-dependent **behave()** method which simulates a single step in a Dog’s life. In the **Calm** state, behave() calls wagTail() but in the **Excited** state it calls bark(). A Dog is initially in a Calm state.

When one Dog **visits** another, they begin to **observe each other**, and the actions of one might affect the state of the other. For instance:

```java
Dog fifi = new Dog();
fifi.acceptVisitor(fido); // could have had fido.acceptVisitor(fifi)
fifi.behave(); // fifi just wags her tail
fido.behave(); // and fido does too. They are calm dogs.
fifi.bark(); // But just cause one to bark ...
fido.behave(); // and it excites the other
fifi.behave(); // and now both are excited.
```

A dog observes its **Owner**:

```java
Owner ollie = new Owner(); // ollie will own both fifi and fido
ollie.owns(fifi); // and fifi now observes ollie
ollie.owns(fido); // and fido now observes ollie as well
```

The **Owner** can calm down the two dogs either by **visiting** them individually:

```java
fido.acceptVisitor(ollie); // now fido is calm
fifi.acceptVisitor(ollie); // now fifi is calm
```

or the Owner can just **shout**, which causes both observant dogs to calm down

```java
ollie.shout(); // both dogs calm down immediately
```

A **Cat** is another **Visitor** to a Dog. When a Cat visits a Dog, the Dog begins to observe it. Only if the Cat then **runs()** will the Dog get excited.

```java
Cat carl = new Cat;
fido.acceptVisitor(carl);
fido.behave(); // fido is calm, so it prints “tail wagging”
carl.run(); // but when the cat runs, fido gets excited
fido.behave(); // and so fido’s behavior changes to barking
fifi.behave(); // and that gets fifi excited and barking
```
A visit from a **Mirror** tricks a Dog into observing itself:

```java
Mirror m = new Mirror();
fido.acceptVisitor(m);
fido.bark(); // get calm Fido to bark just once, ...
fido.behave(); // and it gets excited
```

1) [30%] Finish the **Visitor hierarchy**, using an **interface** for the base **Visitor**. The only visitee is a Dog (Cat, Mirror, and Owner are the Visitors).

```java
public interface Visitor {
    public void visit(Dog d);
}

public class Cat implements Visitor extends Observable {
    public void visit(Dog d) { addObserver(d); }

    public void run() {
        setChanged();
        notifyObserver();
    }
}

public class Owner implements Visitor extends Observable {
    public void owns(Dog d) { addObserver(d); }

    public void visit(Dog d) { d.setCalm(); }

    public void shout() {
        setChanged();
        notifyObserver();
    }
}

public class Mirror implements Visitor {
    public void visit(Dog d) { d.addObserver(d); }
}

public class Dog implements Visitor {
    public void visit(Dog d) { d.addObserver(d); }
}
```
2) [30%] Finish writing Dog

```java
public class Dog extends Observable implements Observer {
    private DogState currentState;
    private CalmDogState calmDog;
    private ExcitedDogState exitedDog;

    public Dog() {
        calmDog = new CalmDogState(this);
        exitedDog = new ExcitedDogState(this);
        setCalm();
    }

    public void bark() {
        System.out.println("woof");
        setChanged();
        notifyObservers();
    }

    public void wagTail() { System.out.println("tail wagging"); }

    public void behave() {
        currentState.behave();
    }

    public void setCalm() {
        currentState = calmDog;
    }

    public void setExcited() {
        currentState = exitedDog;
    }

    public void acceptVisitor(Visitor v) {
        v.visit(this);
    }

    public void update(Observable o, Object obj) {
        if (o instanceof Owner)
            setCalm();
        else if (o instanceof Cat)
            setExcited();
        else if (o instanceof Dog)
            setExcited();
    }
}
```
3) [30%] Finish writing DogState and the extensions CalmDogState and ExcitedDogState.

abstract public class DogState {
    protected Dog parent;

    abstract public DogState(Dog parent) {this.parent = parent; }
    abstract public void behave();
}

public class CalmDogState extends DogState {
    public CalmDog(Dog parent) { super(parent); }
    public void behave() { parent.wagTail(); }
}

public class ExcitedDogState extends DogState {
    public ExcitedDog(Dog parent) { super(parent); }
    public void behave() { parent.bark(); }
}
4) [10%] How would you decorate a Dog with a Muzzle so that it would not bark? Create a UML class diagram, and describe (or simply provide code snippets) regarding the changes. What would it take to un-decorate the Dog (take off the muzzle)?