CMPE 202

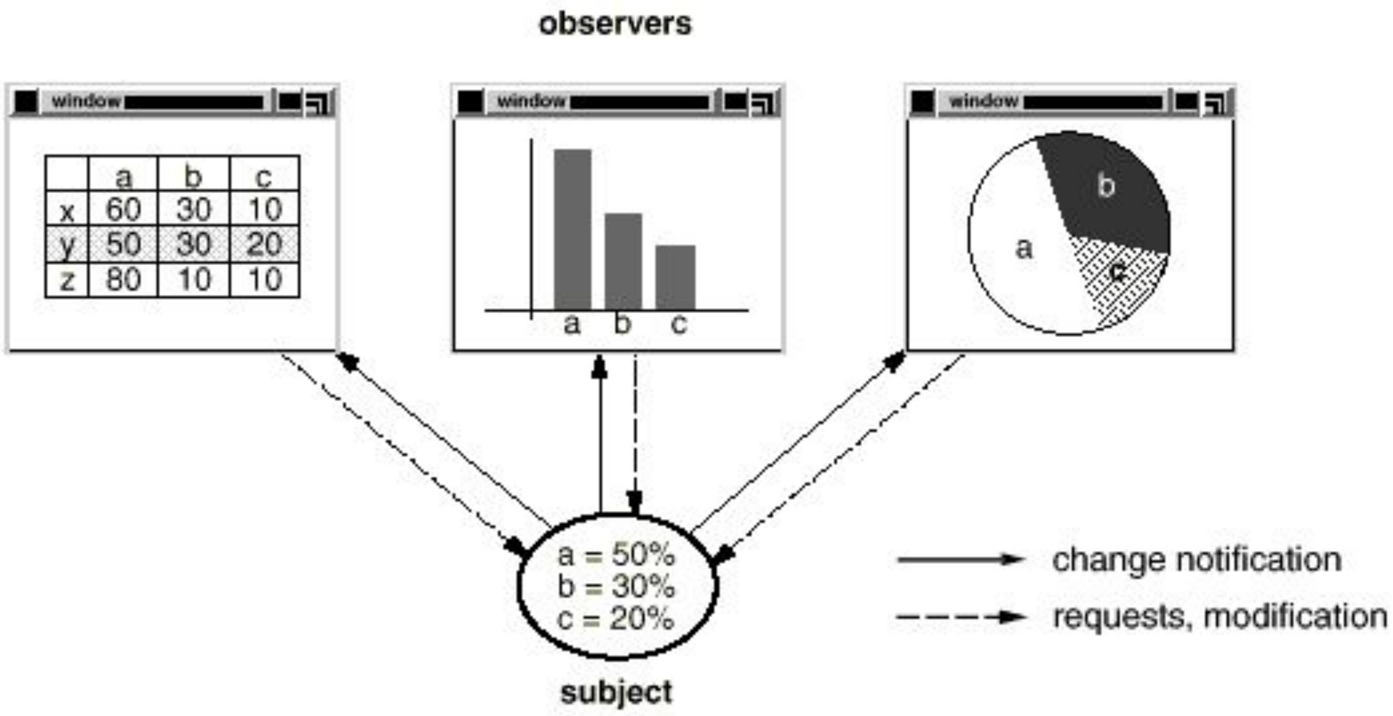
Gang of Four Design Patterns

Observer

Motivation

- Would like to decompose problem into classes, but doing so can sometimes introduce inconsistencies.

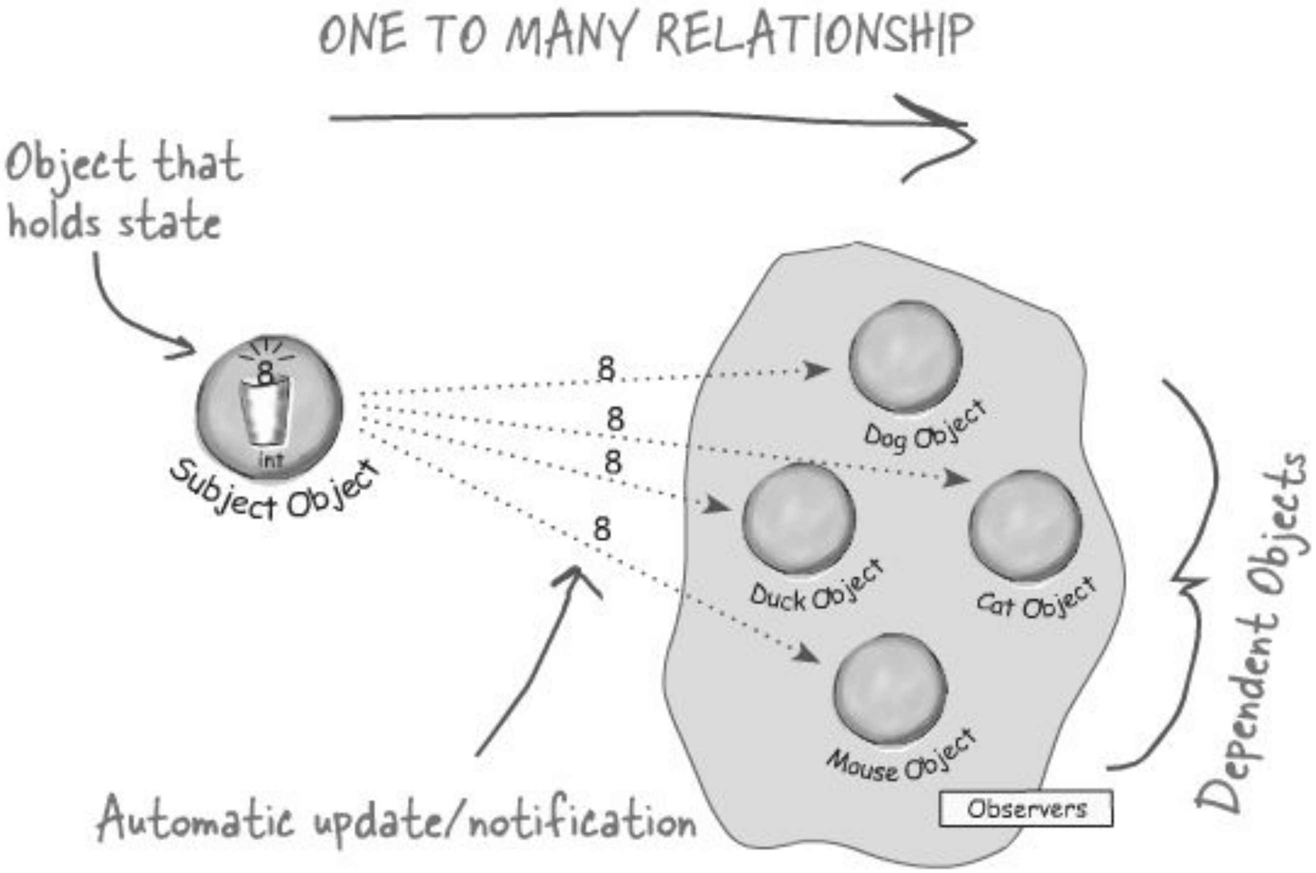
- Want to maintain consistency without tight coupling



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Let's relate this definition to how we've been talking about the pattern:



The Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As

Dependents, Publish-Subscribe

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Participants

- **Subject** (Interface)
    - knows its observers. Any number of Observer objects may observe a subject.
    - provides an interface for attaching and detaching Observer objects.

- **Observer** (Interface)
    - defines an updating interface for objects that should be notified of changes in a subject.

- **ConcreteSubject**
    - stores state of interest to ConcreteObserver objects.
    - sends a notification to its observers when its state changes.

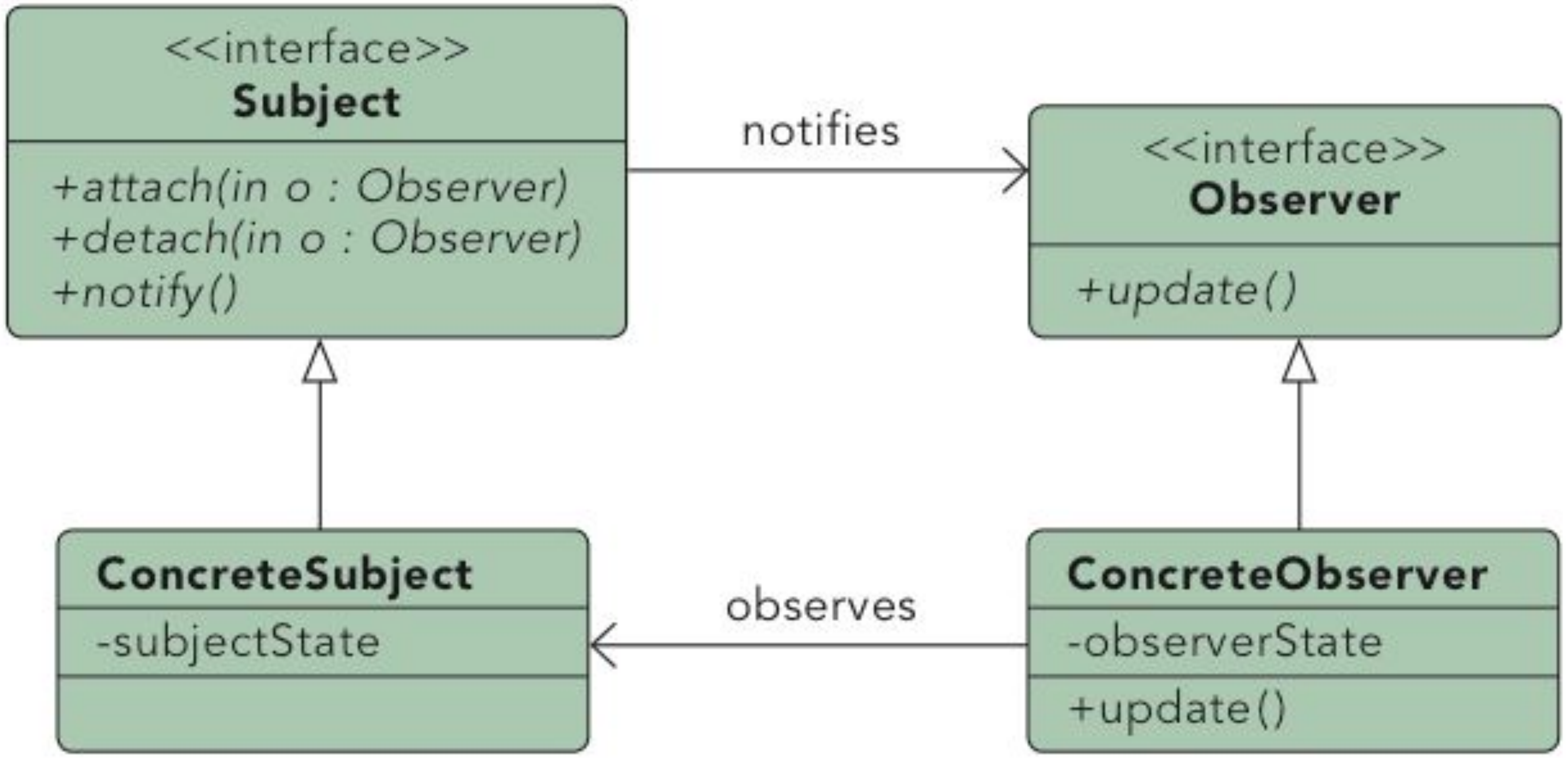
- **ConcreteObserver**
    - maintains a reference to a ConcreteSubject object.
    - stores state that should stay consistent with the subject's.
    - implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

OBSERVER

Object Behavioral

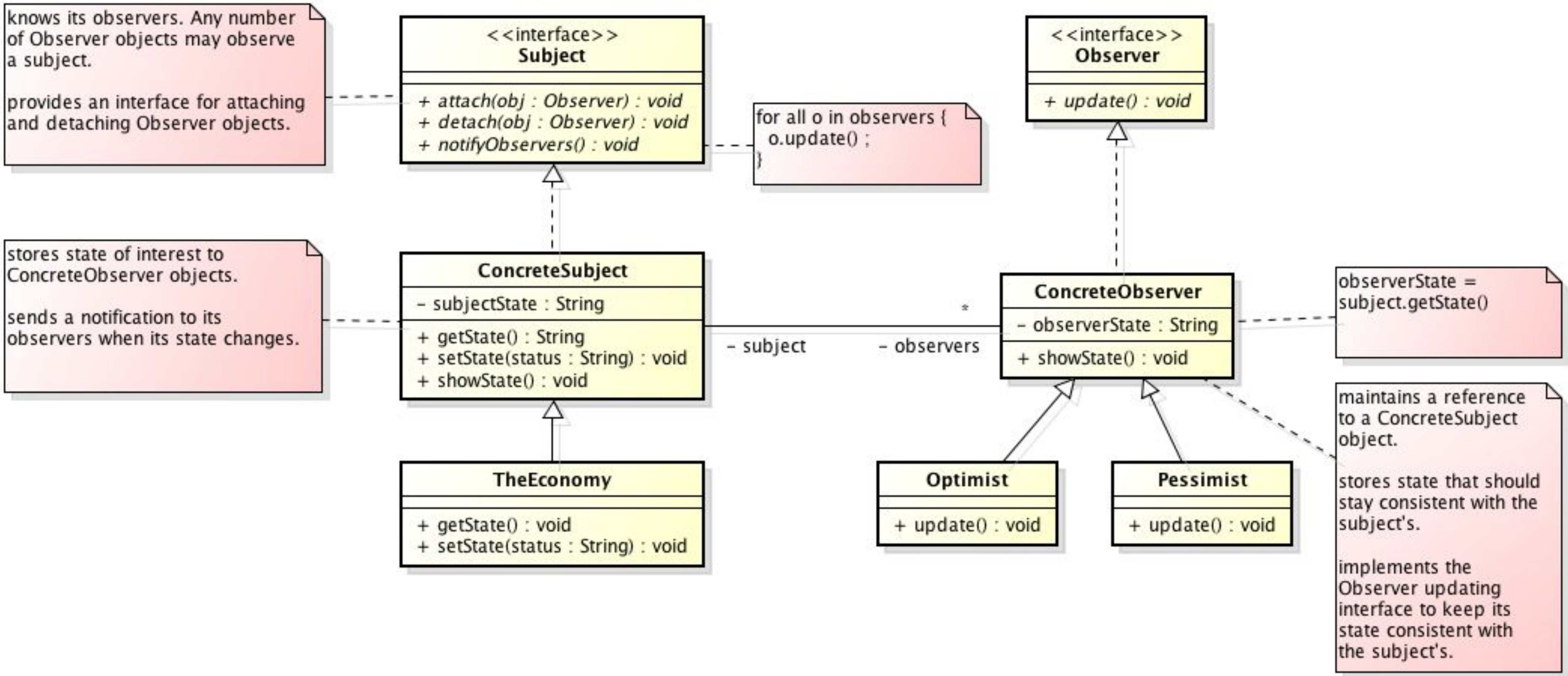


Purpose

Lets one or more objects be notified of state changes in other objects within the system.

Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.



knows its observers. Any number of Observer objects may observe a subject.

provides an interface for attaching and detaching Observer objects.

```
for all o in observers {
  o.update() ;
}
```

stores state of interest to ConcreteObserver objects.

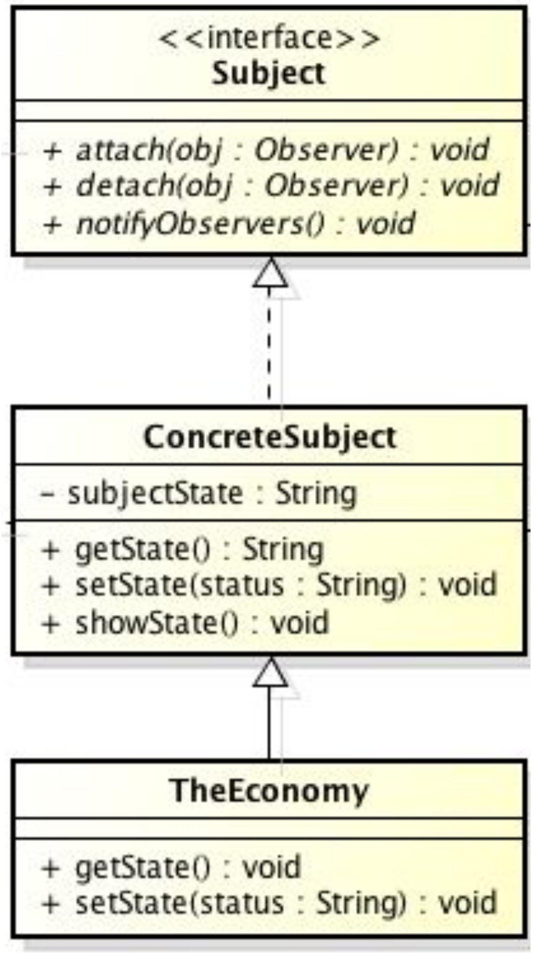
sends a notification to its observers when its state changes.

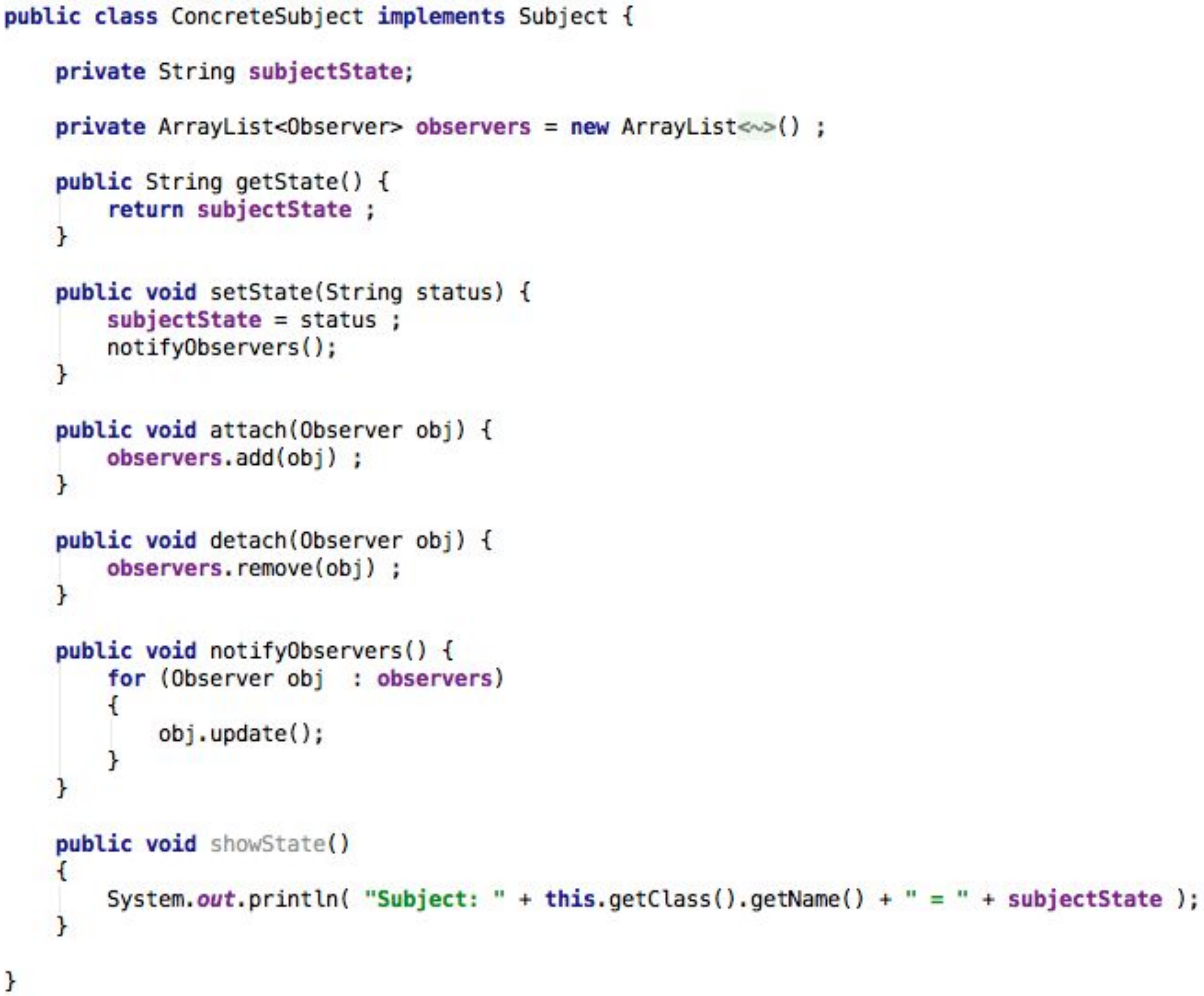
```
observerState =
subject.getState()
```

maintains a reference to a ConcreteSubject object.

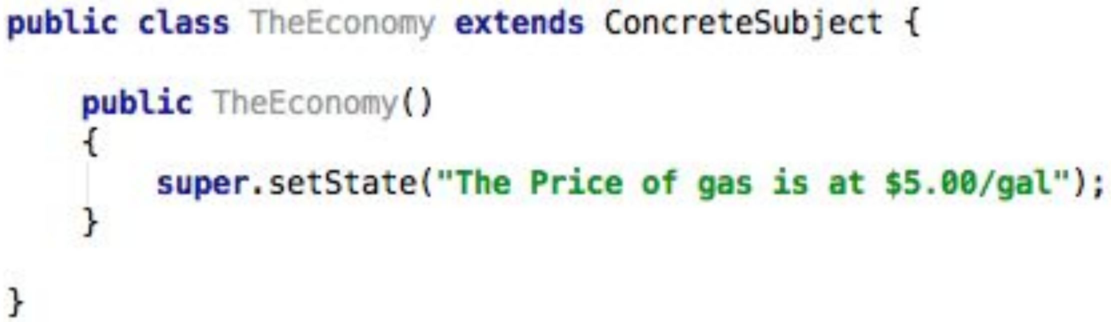
stores state that should stay consistent with the subject's.

implements the Observer updating interface to keep its state consistent with the subject's.

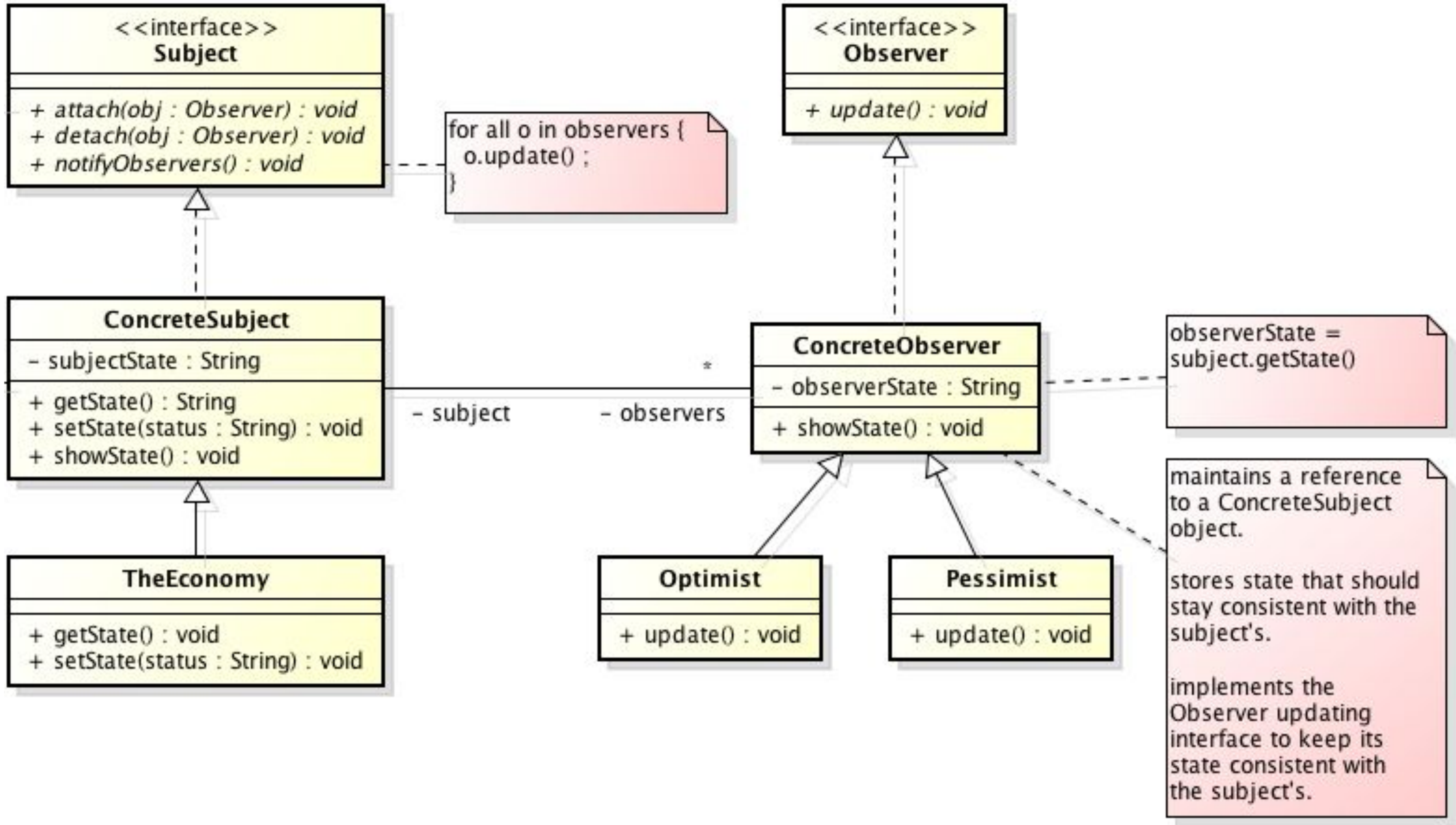

<<interface>>
Subject
+ attach(obj : Observer) : void
+ detach(obj : Observer) : void
+ notifyObservers() : void
ConcreteSubject
- subjectState : String
+ getState() : String
+ setState(status : String) : void
+ showState() : void
TheEconomy
+ getState() : void
+ setState(status : String) : void



```java
public class ConcreteSubject implements Subject {

    private String subjectState;

    private ArrayList<Observer> observers = new ArrayList<~>() ;

    public String getState() {
        return subjectState ;
    }

    public void setState(String status) {
        subjectState = status ;
        notifyObservers();
    }

    public void attach(Observer obj) {
        observers.add(obj) ;
    }

    public void detach(Observer obj) {
        observers.remove(obj) ;
    }

    public void notifyObservers() {
        for (Observer obj  : observers)
        {
            obj.update();
        }
    }

    public void showState()
    {
        System.out.println( "Subject: " + this.getClass().getName() + " = " + subjectState );
    }

}
```



```java
public class TheEconomy extends ConcreteSubject {

    public TheEconomy()
    {
        super.setState("The Price of gas is at $5.00/gal");
    }

}
```



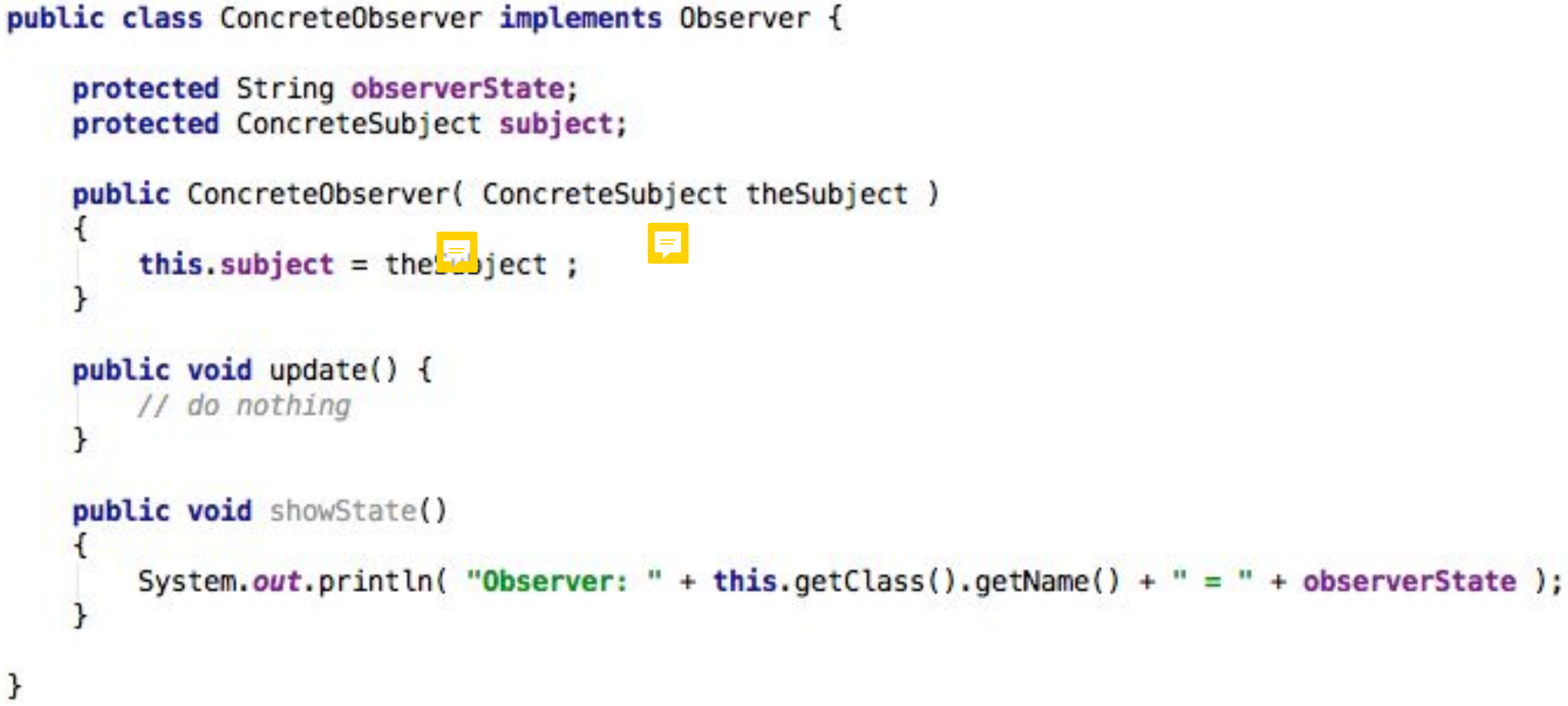
for all o in observers {
  o.update() ;
}

observerState =
subject.getState()

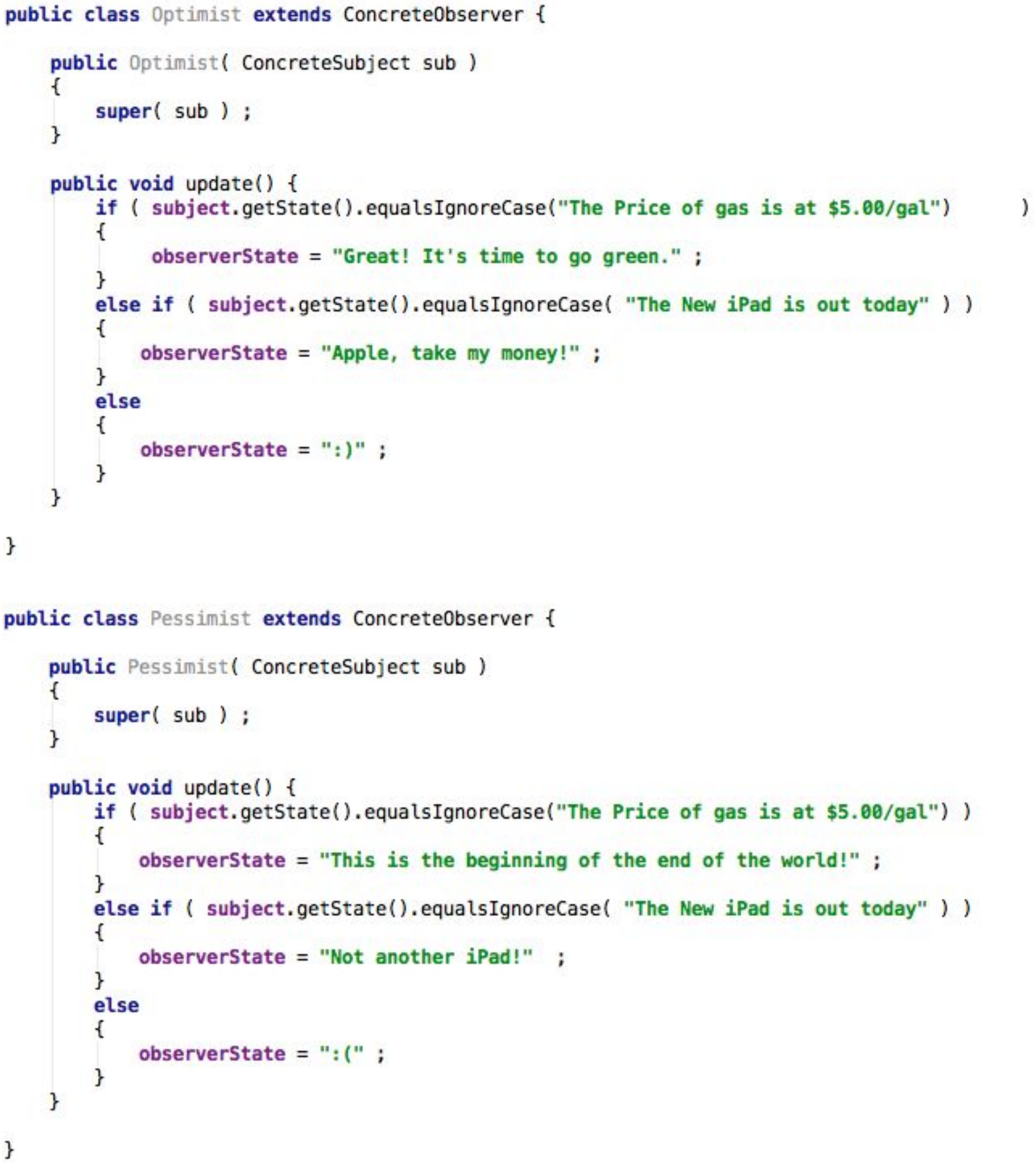
maintains a reference to a ConcreteSubject object.

stores state that should stay consistent with the subject's.

implements the Observer updating interface to keep its state consistent with the subject's.



```
public class ConcreteObserver implements Observer {

    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject ;
    }

    public void update() {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().getName() + " = " + observerState );
    }

}
```



```java
public class Optimist extends ConcreteObserver {

    public Optimist( ConcreteSubject sub )
    {
        super( sub ) ;
    }

    public void update() {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal")       )
        {
             observerState = "Great! It's time to go green." ;
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Apple, take my money!" ;
        }
        else
        {
            observerState = ":)" ;
        }
    }

}


public class Pessimist extends ConcreteObserver {

    public Pessimist( ConcreteSubject sub )
    {
        super( sub ) ;
    }

    public void update() {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal") )
        {
            observerState = "This is the beginning of the end of the world!" ;
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Not another iPad!"  ;
        }
        else
        {
            observerState = ":(" ;
        }
    }

}
```