Observer Pattern

Lecture-8

Observer Pattern

- Observer is a behavioural design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

- Observer design pattern is also called as publish-subscribe pattern

- The observer pattern is generally implemented in a single-application scope. On the other hand, the publisher-subscriber pattern is mostly used as a cross-application pattern (such as how Kafka is used as the Heart of event-driven architecture) and is generally used to decouple data/event streams and systems

Problem

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

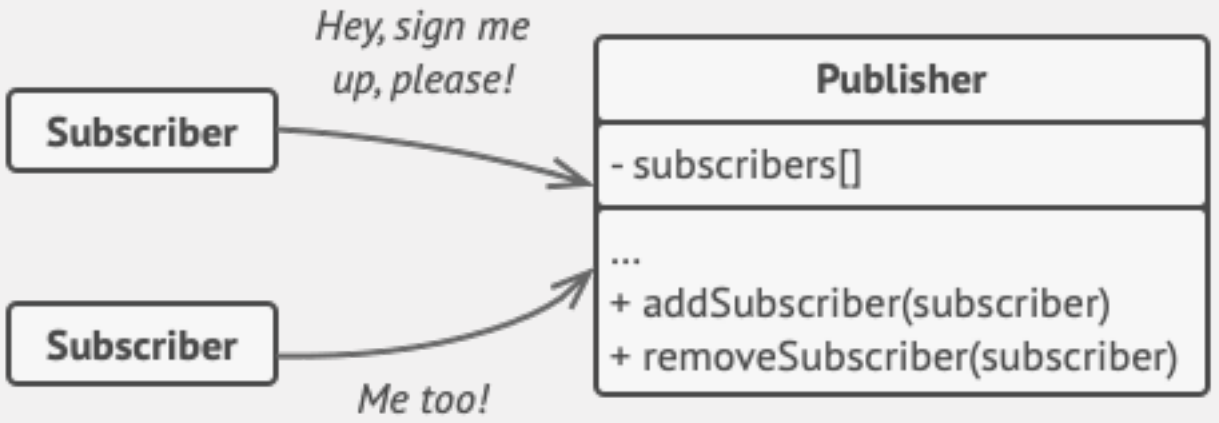
On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.

Solution

The object that has some interesting state is often called *subject*, but since it's also going to notify other objects about the changes to its state, we'll call it *publisher*. All other objects that want to track changes to the publisher's state are called *subscribers*.

The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

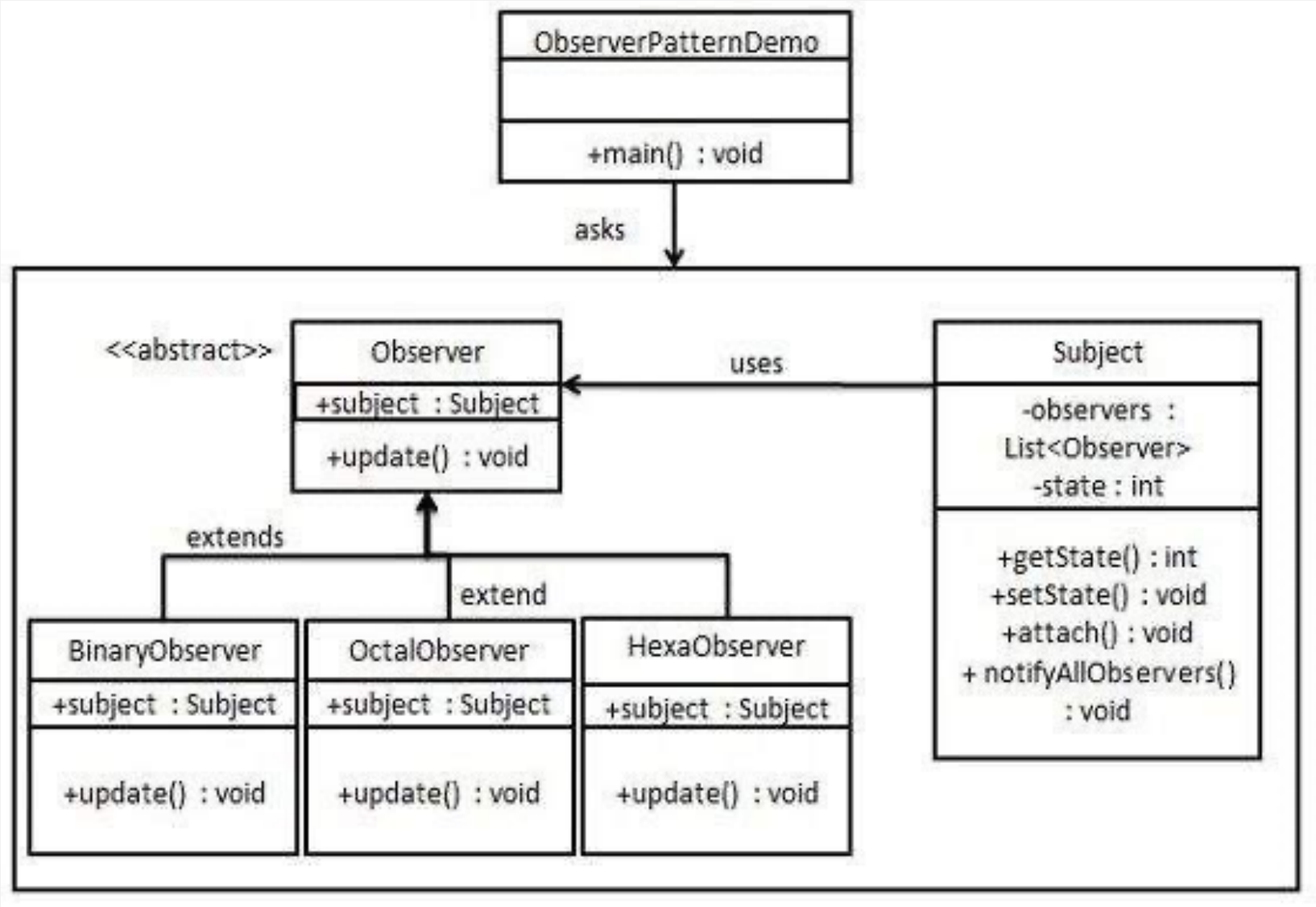
1. an array field for storing a list of references to subscriber objects
2. several public methods which allow adding subscribers to and removing them from that list.



Real World Analogy

- If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.
- The publisher maintains a list of subscribers and knows which magazines they're interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

Class Diagram



Implementation

Step 1
Create Subject class

```
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

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

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

Step 2
Create Observer class

```
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

Implementation

Step 3
Create Concrete Observer classes

```
public class OctalObserver extends Observer{

   public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
     System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
   }
}

public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
   }
}
```

```
public class HexaObserver extends Observer{

   public HexaObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );
   }
}
```

Implementation

Step 4

Use *Subject* and concrete observer objects.

```
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

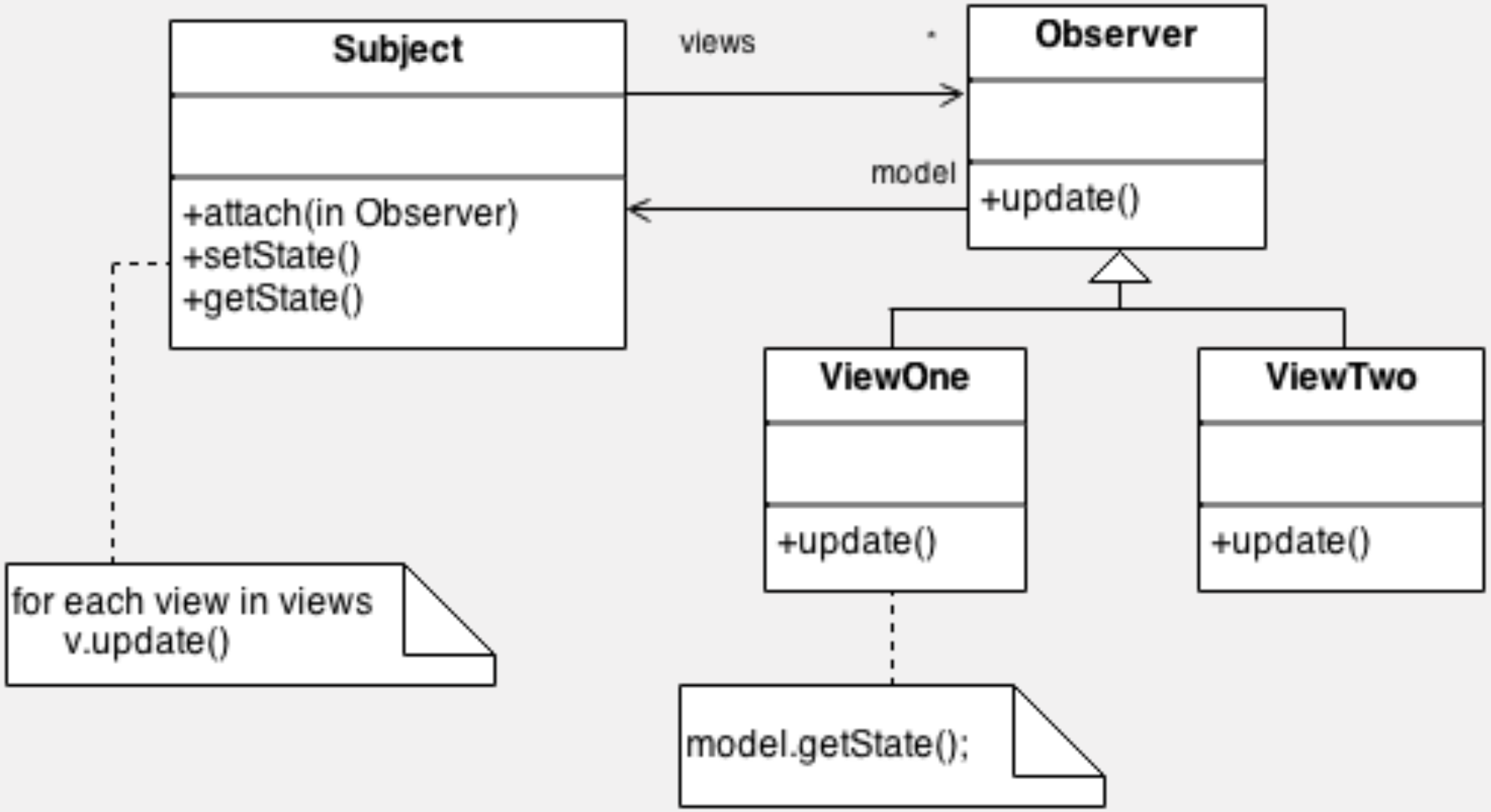
Output

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

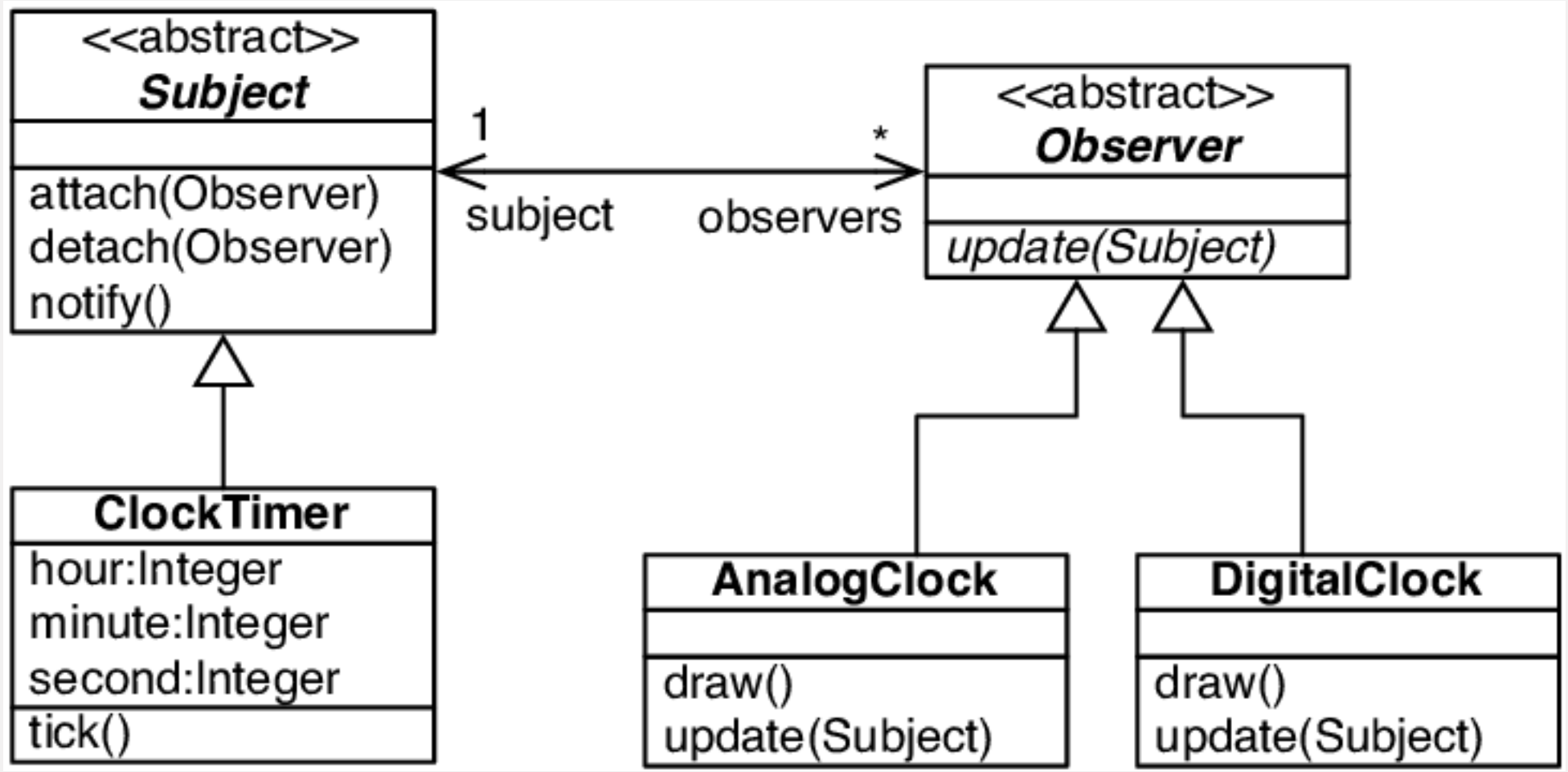
Advantages

- It describes the coupling between the objects and the observer.
- It provides the support for broadcast-type communication.
- Open/Closed Principle. You can introduce new subscriber classes without having to change the publisher’s code (and vice versa if there’s a publisher interface)

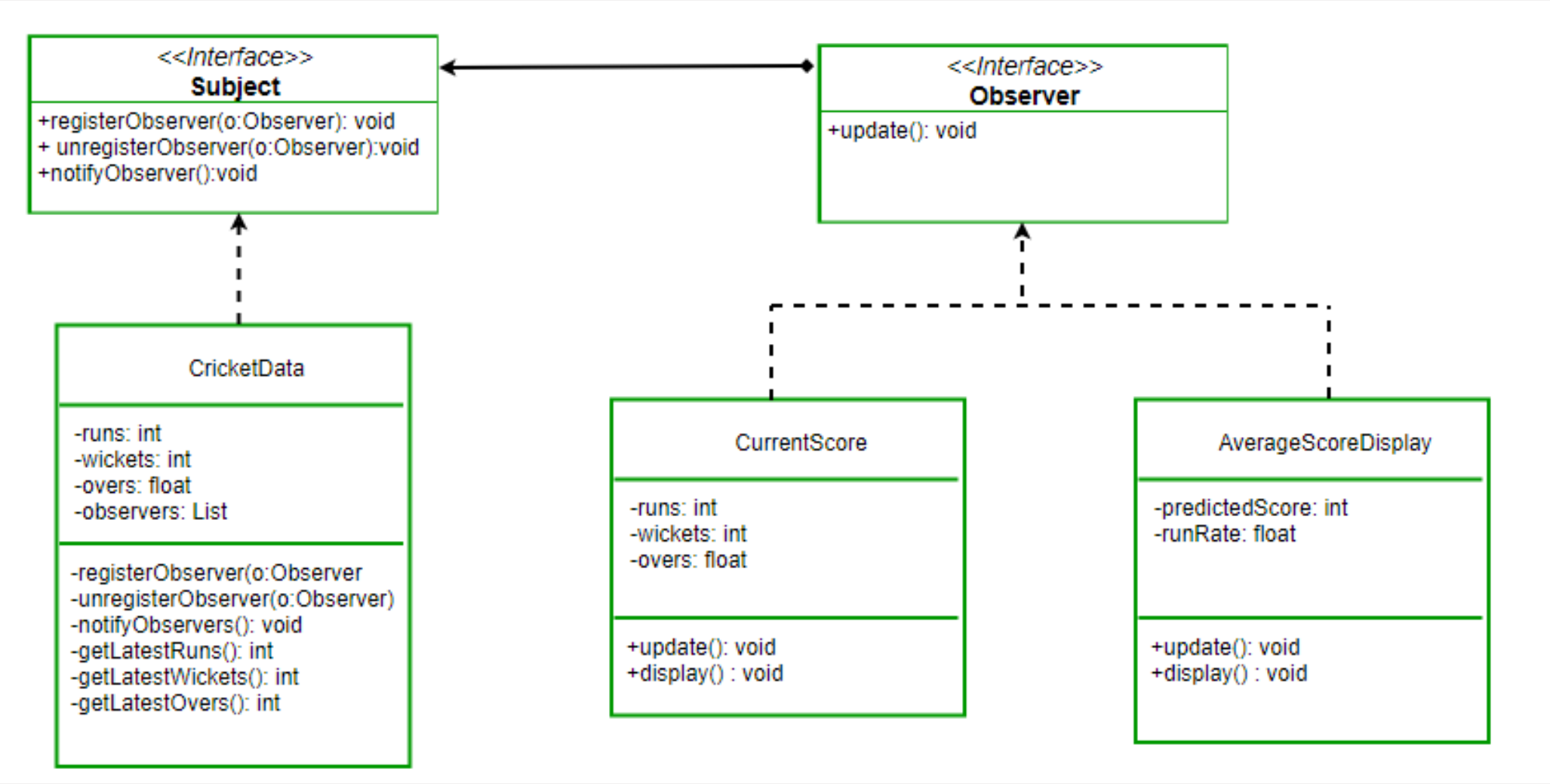
Example 2


Subject
views
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
ViewTwo
+update()
+update()
for each view in views
v.update()
model.getState();

Example 3


<<abstract>>
Subject
attach(Observer)
detach(Observer)
notify()
1
subject
*
observers
<<abstract>>
Observer
update(Subject)
ClockTimer
hour:Integer
minute:Integer
second:Integer
tick()
AnalogClock
draw()
update(Subject)
DigitalClock
draw()
update(Subject)

Example 4


<<Interface>>
Subject
+registerObserver(o:Observer): void
+ unregisterObserver(o:Observer):void
+notifyObserver():void
<<Interface>>
Observer
+update(): void
CricketData
-runs: int
-wickets: int
-overs: float
-observers: List
-registerObserver(o:Observer
-unregisterObserver(o:Observer)
-notifyObservers(): void
-getLatestRuns(): int
-getLatestWickets(): int
-getLatestOvers(): int
CurrentScore
-runs: int
-wickets: int
-overs: float
+update(): void
+display() : void
AverageScoreDisplay
-predictedScore: int
-runRate: float
+update(): void
+display() : void