Introduction to Design Patterns

Faculty of Computers and Information

Supervisor

Dr Mai Hamdallah

CS Dept., FCI, Helwan University

Presented by

Tamer Abd-Elaziz Yassen

Teaching Assistant CS Dept., FCI, Helwan University

Chapter 2
Observer Pattern

Keeping your Objects in the know

Hey Jerry, I'm notifying everyone that the Patterns Group meeting moved to Saturday night. We're going to be talking about the Observer Pattern. That pattern is the best! It's the BEST, Jerry!

Meet the Observer Pattern

You know how newspaper or magazine subscriptions work:

1. A newspaper publisher goes into business and begins publishing newspapers.

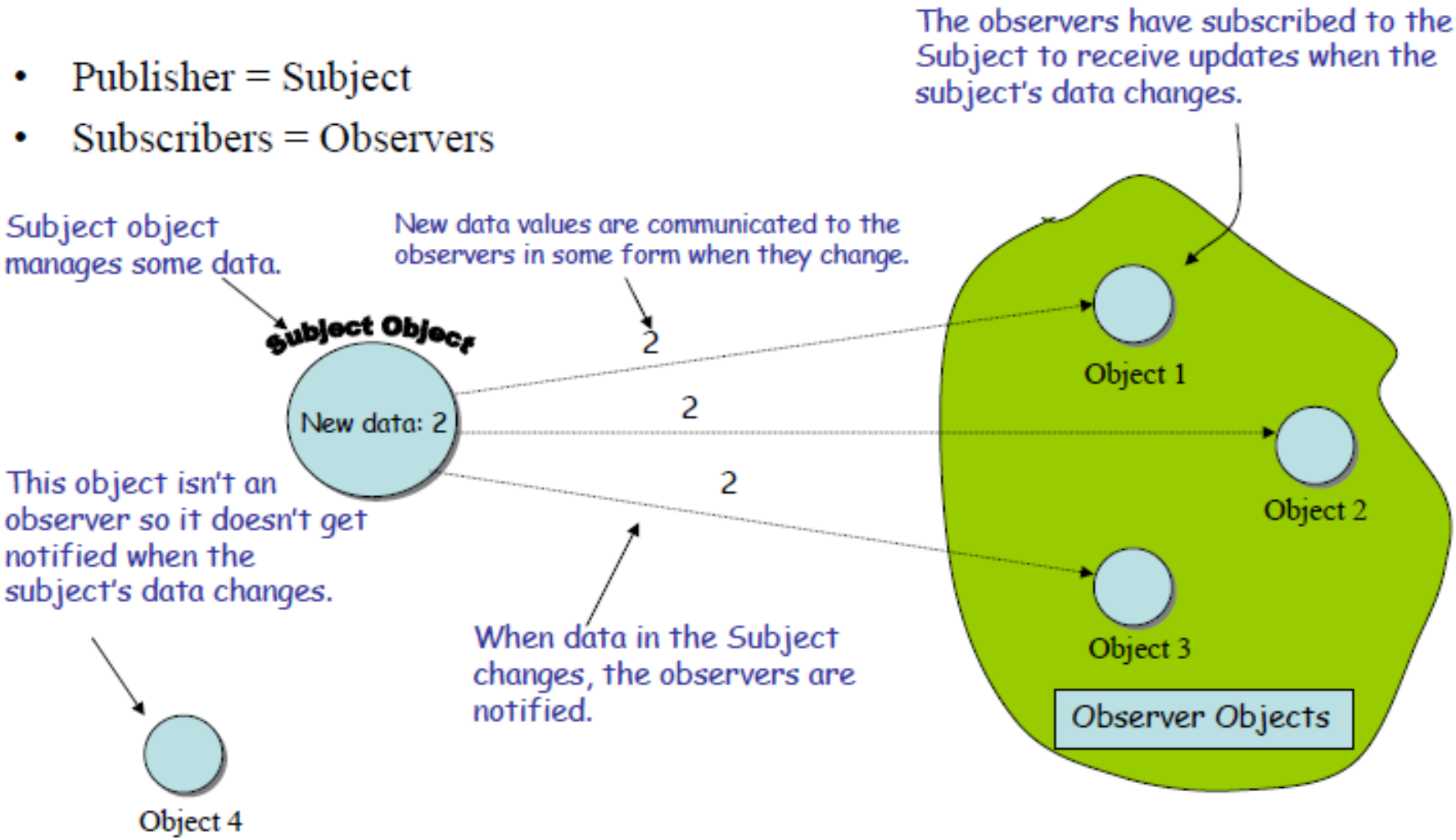
2. You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.

3. You unsubscribe when you don't want papers anymore, and they stop being delivered.

4. While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern

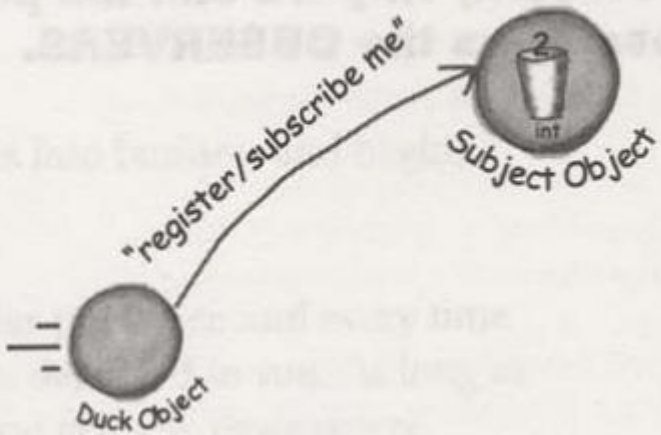
- Publisher = Subject
- Subscribers = Observers

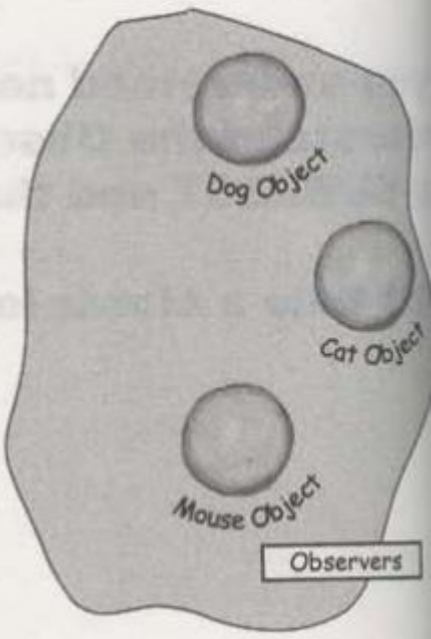


A day in the life of the Observer Pattern

A Duck object comes along and tells the Subject that it wants to become an observer.

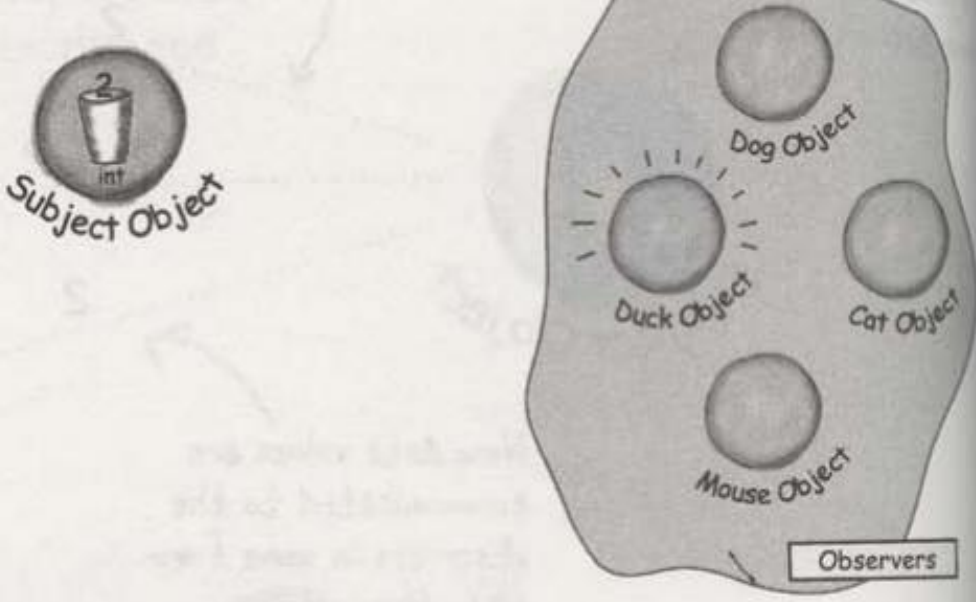
Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...





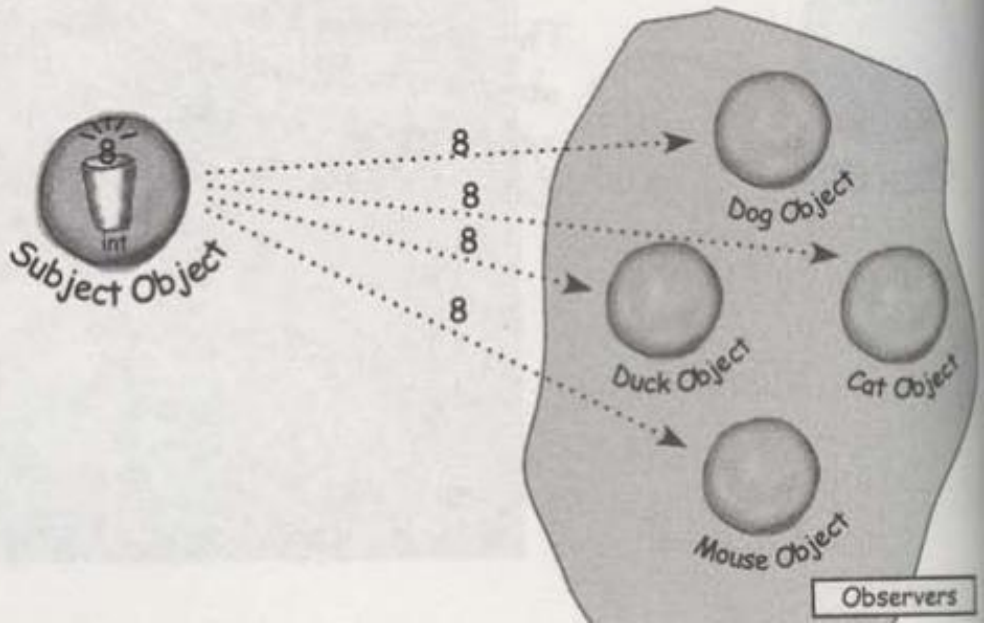
The Duck object is now an official observer.

Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.



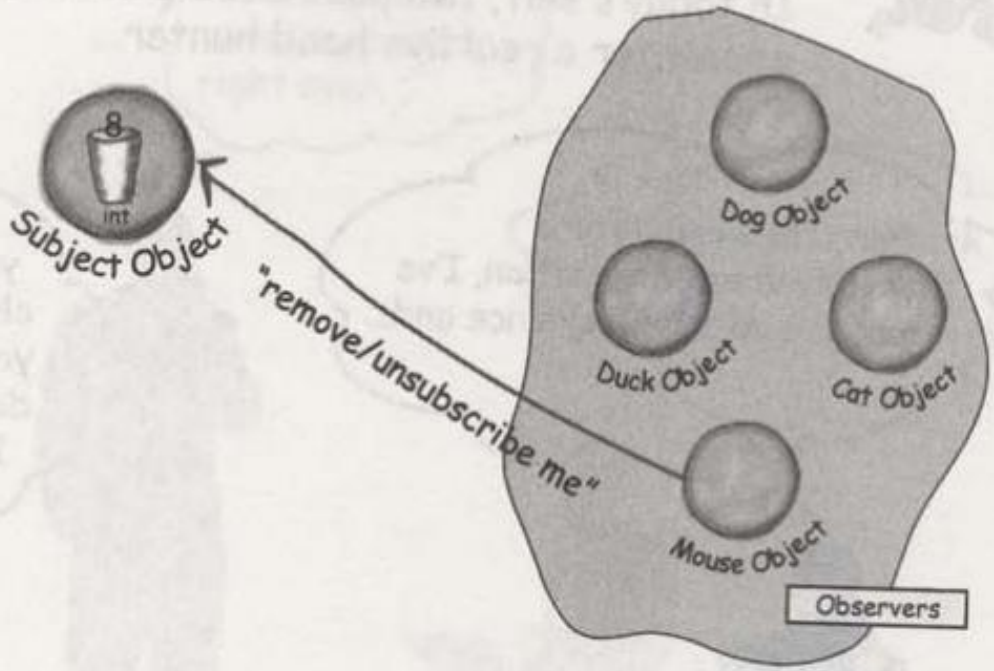
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



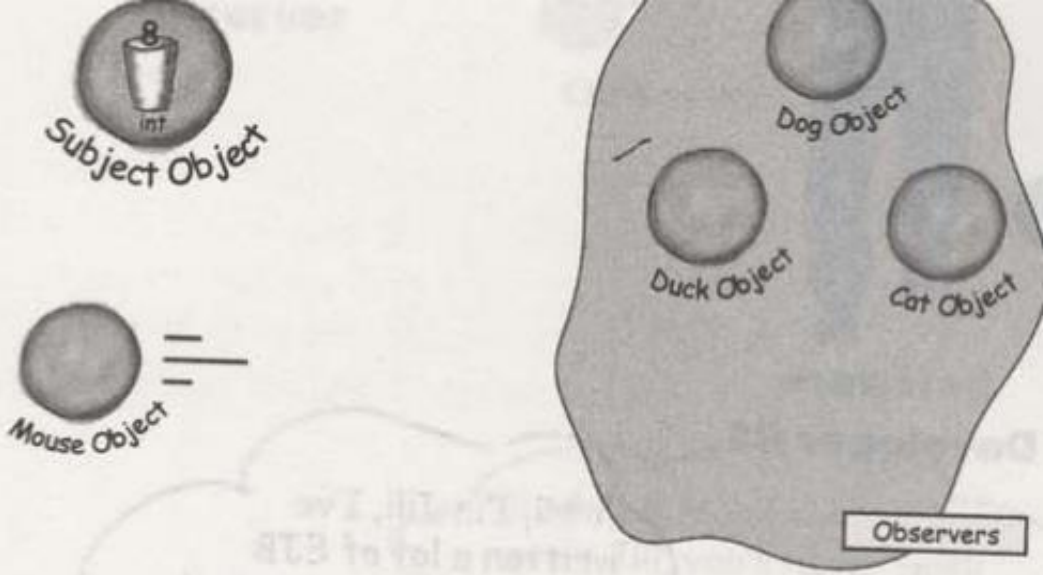
The Mouse object asks to be removed as an observer.

The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.



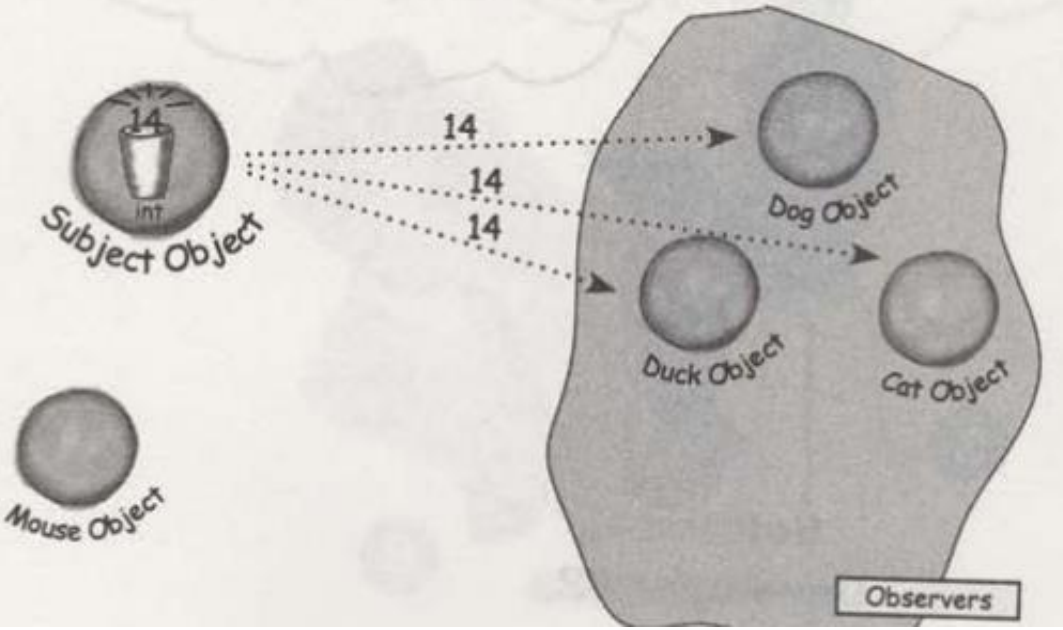
Mouse is outta here!

The Subject acknowledges the Mouse's request and removes it from the set of observers.



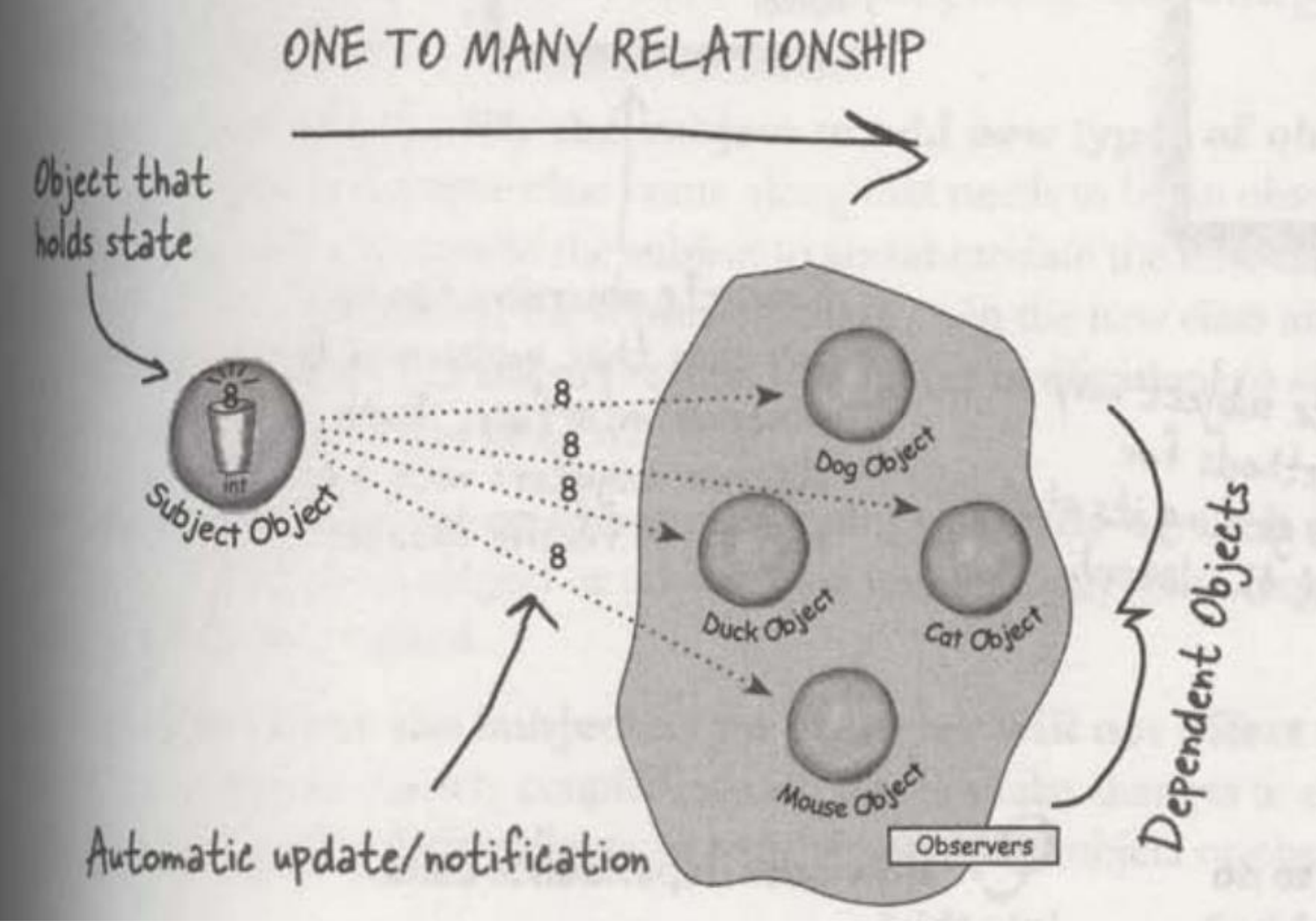
The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe it'll ask to be an observer again some day.

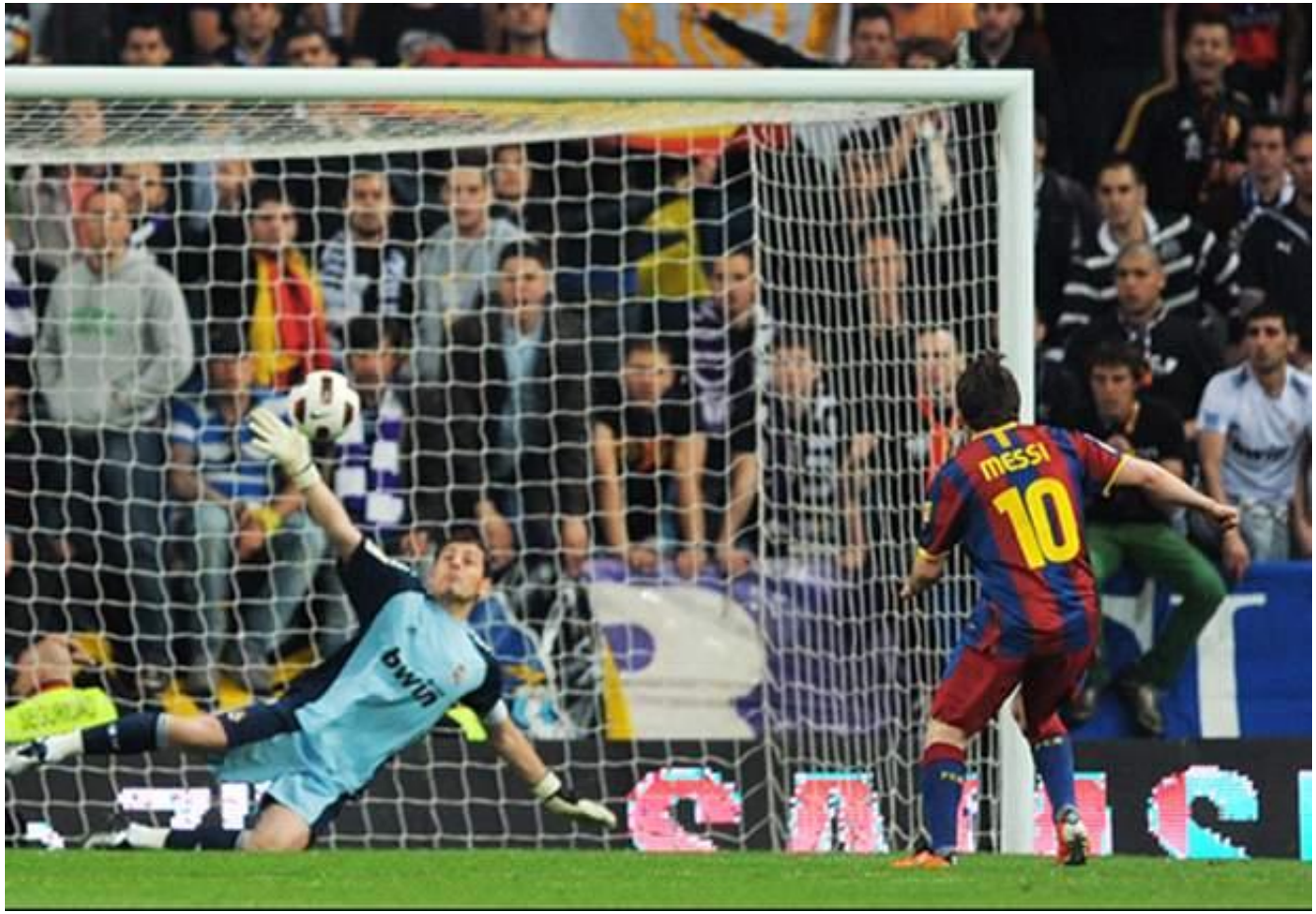


Observer Pattern

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.



Sports Goal is a fantastic sports site for sport lovers. Now, they are planning to provide live commentary or scores of matches as an SMS service, As a user, you need to subscribe to the package and when there is a live match you will get an SMS to the live commentary. The site also provides an option to unsubscribe from the package whenever you want to.



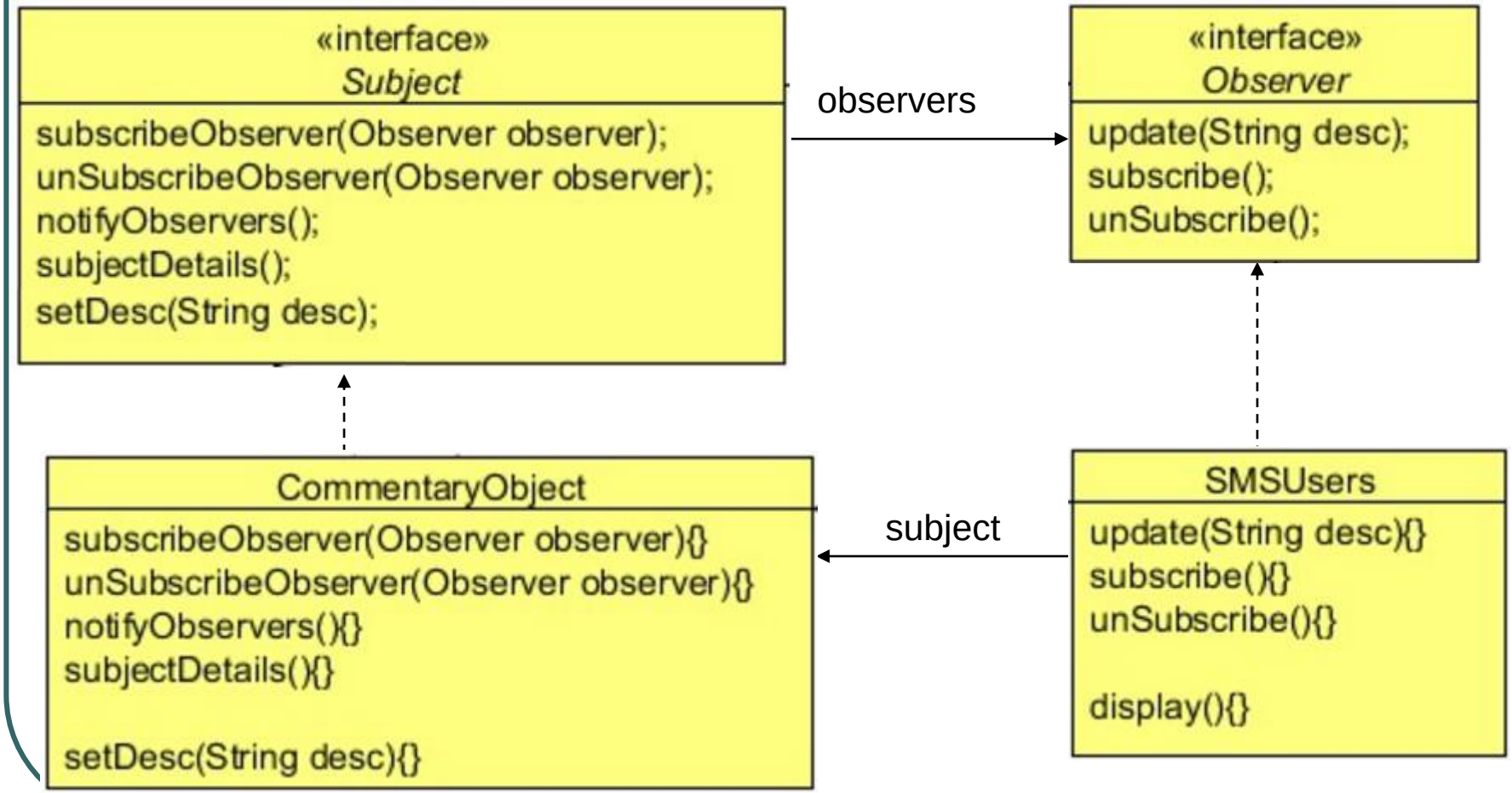
Your Job ?

As a developer your job is to provide the commentary to the registered users by fetching it from the commentary object when it's available. When there is an update, the system should update the subscribed users by sending them the SMS.

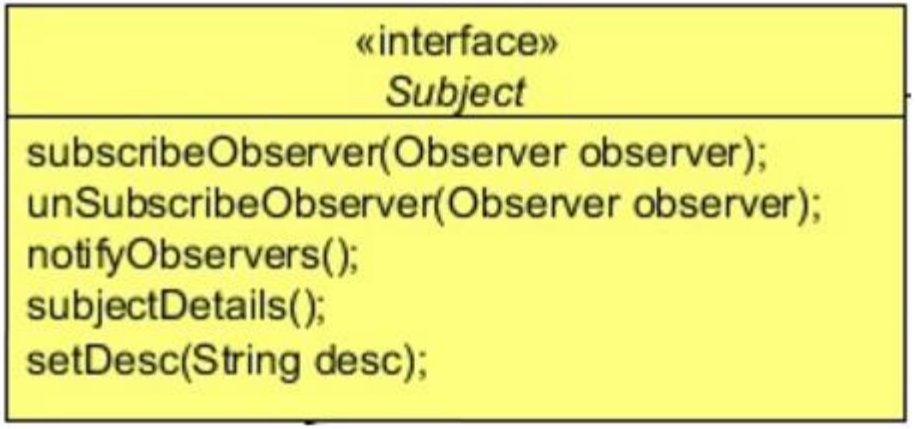
This situation clearly shows one-to-many mapping between the match and the users, as there could be many users to subscribe to a single match.

The Observer Design Pattern is best suited to this situation, let's see about this pattern and then create the feature for Sport Goal.

Implementing using Observer Pattern

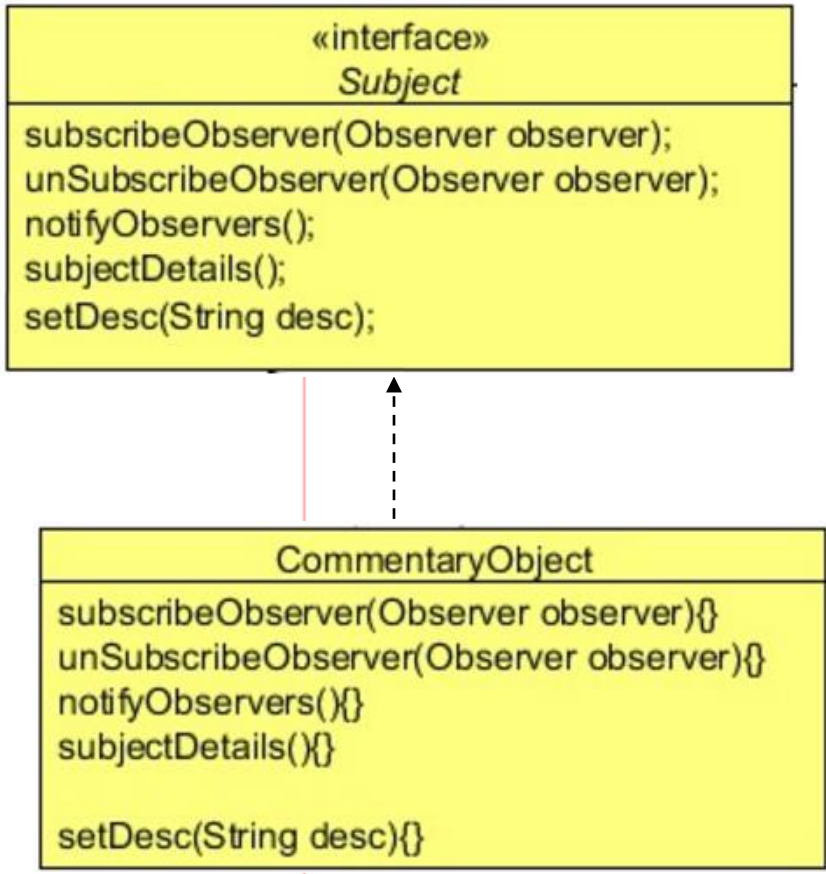


Implementing interface Subject

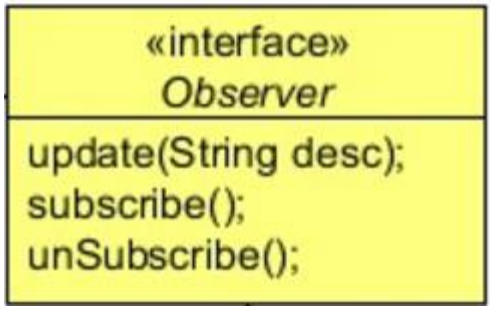



```
public interface Subject {

        public void subscribeObserver(Observer observer);
        public void unSubscribeObserver(Observer observer);
        public void notifyObservers();
        public void setDesc(String desc);
        public String subjectDetails();
}
```

```java
public class CommentaryObject implements Subject{
        private final List<Observer>observers;
        private String desc;
        private final String subjectDetails;
        public CommentaryObject(String subjectDetails){
                this.observers = new ArrayList<Observer>();
                this.subjectDetails = subjectDetails;
        }
        @Override
        public void subscribeObserver(Observer observer) {
                observers.add(observer);
        }
        @Override
        public void unSubscribeObserver(Observer observer) {
                int index = observers.indexOf(observer);
                observers.remove(index);
        }
        @Override
        public void notifyObservers() {
                System.out.println();
                for(Observer observer : observers){
                        observer.update(desc);
                }
        }
        @Override
        public void setDesc(String desc) {
                this.desc = desc;
                notifyObservers();
        }
        @Override
        public String subjectDetails() {
                return subjectDetails;
        }
}
```

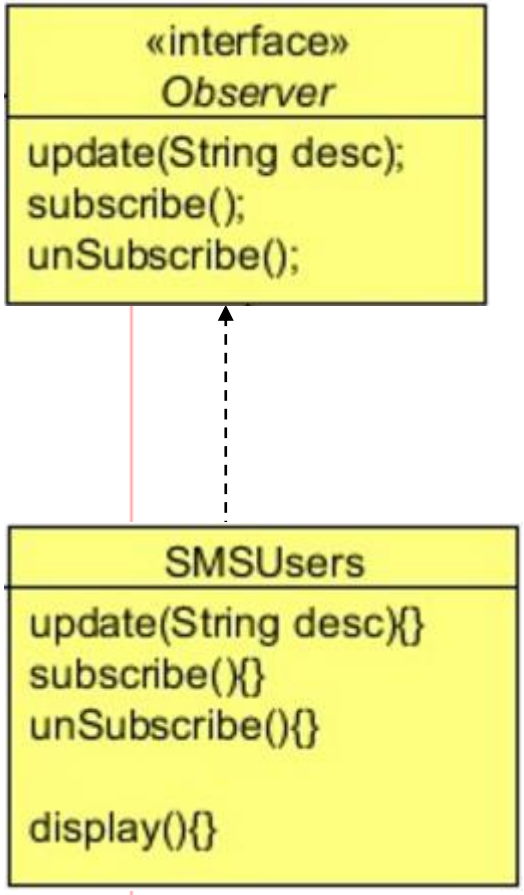


Implementing interface Observer




```
public interface Observer {

        public void update(String desc);
        public void subscribe();
        public void unSubscribe();
}
```

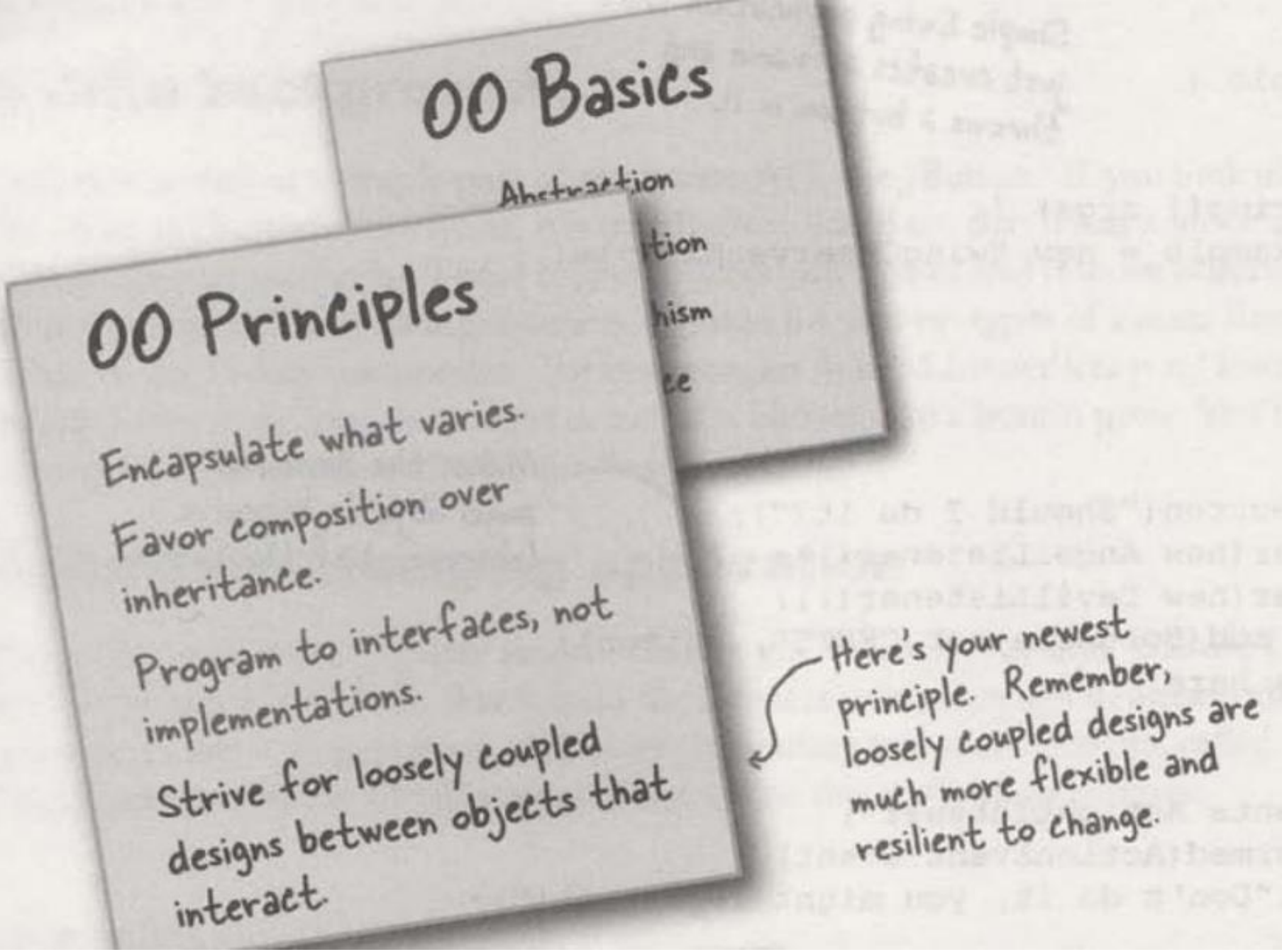
```java
public class SMSUsers implements Observer {

    private final Subject subject;
    private String desc;
    private String userInfo;

    public SMSUsers(Subject subject, String userInfo) {
        if (subject == null) {
            throw new IllegalArgumentException("No Publisher found.");
        }
        this.subject = subject;
        this.userInfo = userInfo;
    }

    @Override
    public void update(String desc) {
        this.desc = desc;
        display();
    }
    private void display() {
        System.out.println("[" + userInfo + "]: " + desc);
    }
    @Override
    public void subscribe() {
        System.out.println("Subscribing " + userInfo + " to " + subject.subjectDetails() + " ...");
        this.subject.subscribeObserver(this);
        System.out.println("Subscribed successfully.");
    }
    @Override
    public void unSubscribe() {
        System.out.println("Unsubscribing " + userInfo + " to " + subject.subjectDetails() + " ...");
        this.subject.unSubscribeObserver(this);
        System.out.println("Unsubscribed successfully.");
    }
}
```



```
public static void main(String[] args) {
    Subject subject = new CommentaryObject("Soccer Match [2016AUG24]");

    Observer observer = new SMSUsers(subject, "Adam Warner [New York]");
    observer.subscribe();

    System.out.println();

    Observer observer2 = new SMSUsers(subject, "Tim Ronney [London]");
    observer2.subscribe();


    subject.setDesc("Welcome to live Soccer match");
    subject.setDesc("Current score 0-0");

    System.out.println();

    observer2.unSubscribe();

    System.out.println();

    subject.setDesc("It's a goal!!");
    subject.setDesc("Current score 1-0");

    System.out.println();

    Observer observer3 = new SMSUsers(subject, "Marrie [Paris]");
    observer3.subscribe();

    System.out.println();

    subject.setDesc("It's another goal!!");
    subject.setDesc("Half-time score 2-0");


}
```

Output - CS_352 _ Sec 2 _ Spring 2016 SWE_Dept (run)

```
run:
Subscribing Adam Warner [New York] to Soccer Match [2016AUG24] ...
Subscribed successfully.

Subscribing Tim Ronney [London] to Soccer Match [2016AUG24] ...
Subscribed successfully.

[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Welcome to live Soccer match

[Adam Warner [New York]]: Current score 0-0
[Tim Ronney [London]]: Current score 0-0

Unsubscribing Tim Ronney [London] to Soccer Match [2016AUG24] ...
Unsubscribed successfully.


[Adam Warner [New York]]: It's a goal!!

[Adam Warner [New York]]: Current score 1-0

Subscribing Marrie [Paris] to Soccer Match [2016AUG24] ...
Subscribed successfully.


[Adam Warner [New York]]: It's another goal!!
[Marrie [Paris]]: It's another goal!!

[Adam Warner [New York]]: Half-time score 2-0
[Marrie [Paris]]: Half-time score 2-0
BUILD SUCCESSFUL (total time: 2 seconds)
```


OO Basics
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

The power of Loose Coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Why?

Thanks

Any questions contact with me via e-mail : tamer.a.yassen@gmail.com