Design Patterns

Observer Pattern*

ebru@hacettepe.edu.tr

ebruakcapinarsezer@gmail.com

http://yunus.hacettepe.edu.tr/~ebru/

@ebru176

Kasım 2017

*revised from Observer Pattern, OOA&D, Rubal Gupta, CSPP, Winter '09

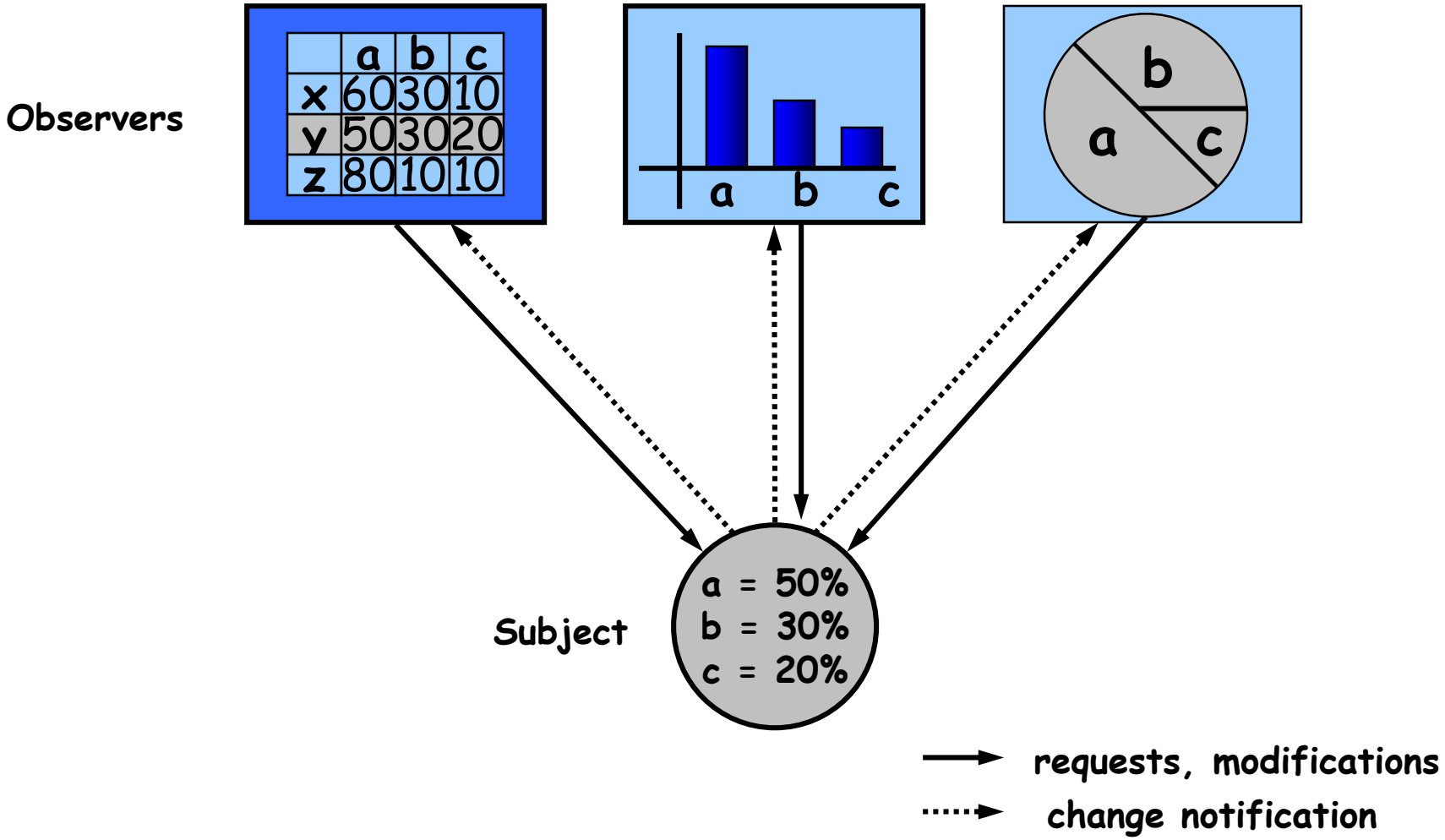
Observer Pattern

- Defines a “one-to-many” dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- a.k.a Dependence mechanism / publish-subscribe / broadcast / change-update

Subject & Observer

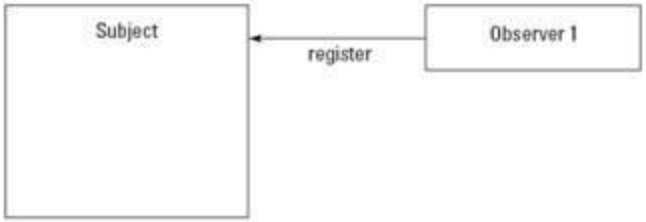
- Subject
  - the object which will frequently change its state and upon which other objects depend
- Observer
  - the object which depends on a subject and updates according to its subject's state.

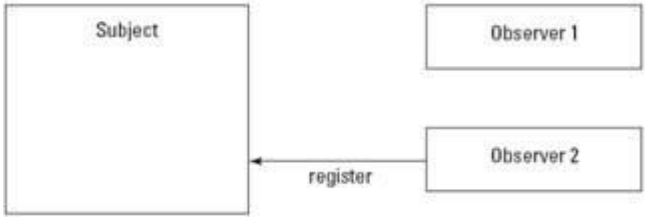
Observer Pattern - Example



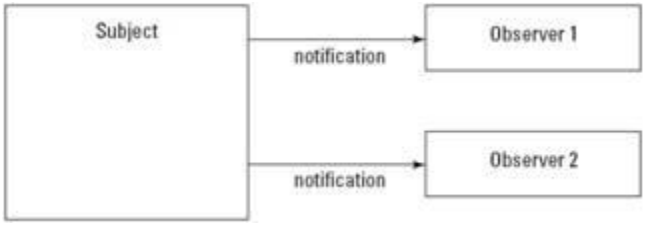
Observer Pattern - Working

A number of Observers “register” to receive notifications of changes to the Subject. Observers are not aware of the presence of each other.





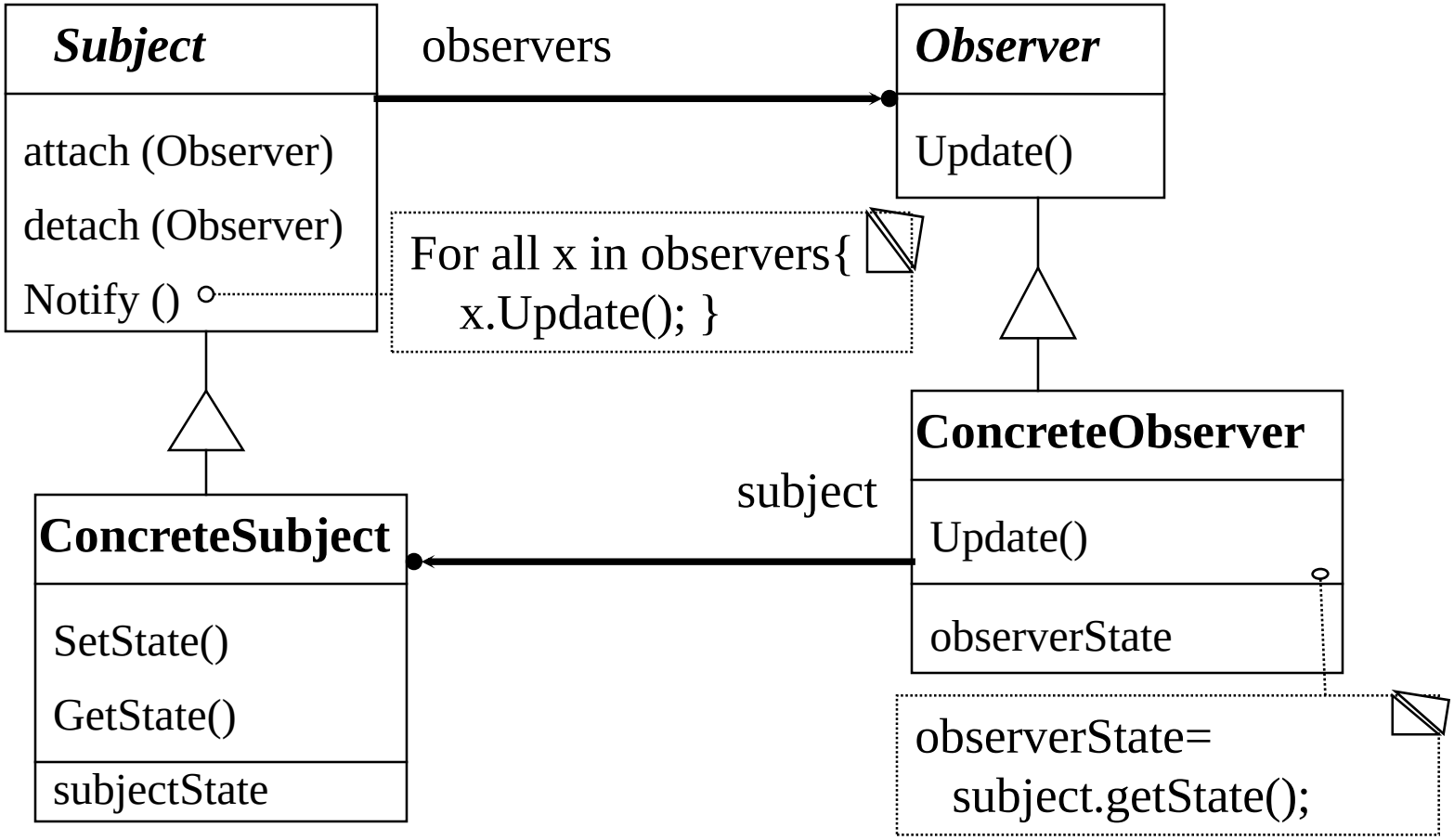
When a certain event or “change” in Subject occurs, all Observers are “notified’.



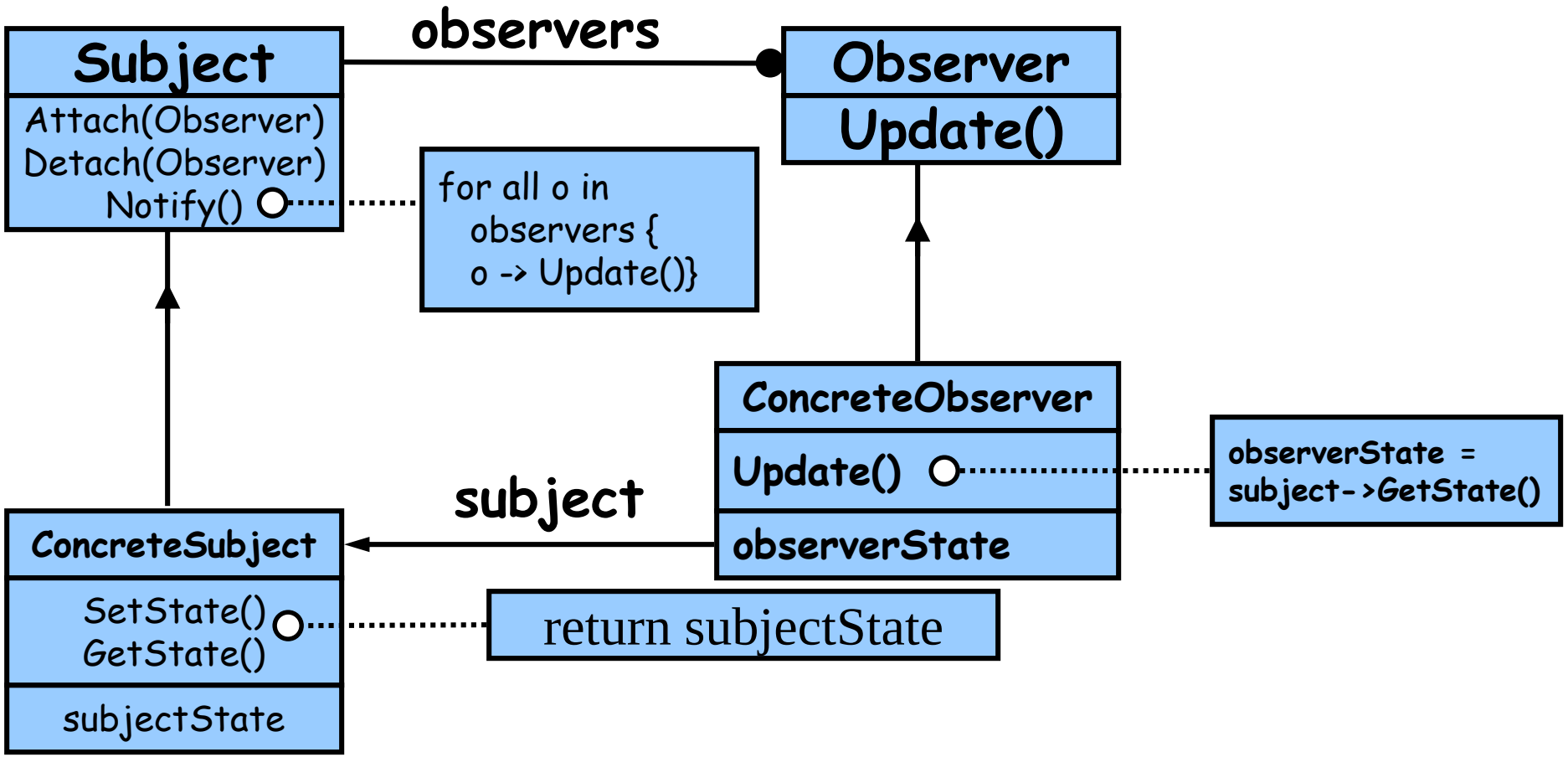
Observer Pattern - Key Players

- Subject
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that gets notified of changes in a subject
- ConcreteSubject
  - Stores “state of interest” to observers
  - Sends notification when state changes
- ConcreteObserver
  - Implements updating interface

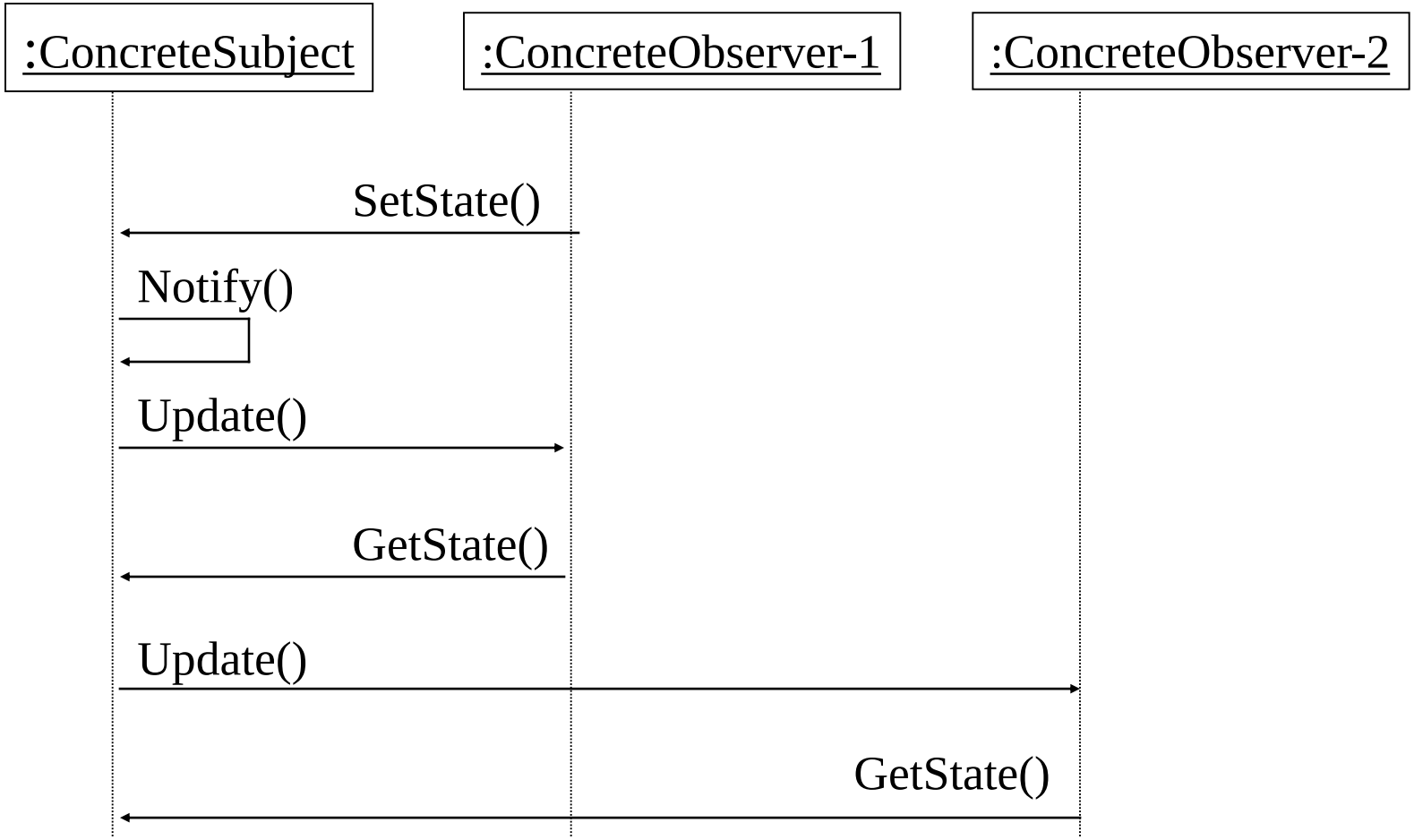
Observer Pattern - UML



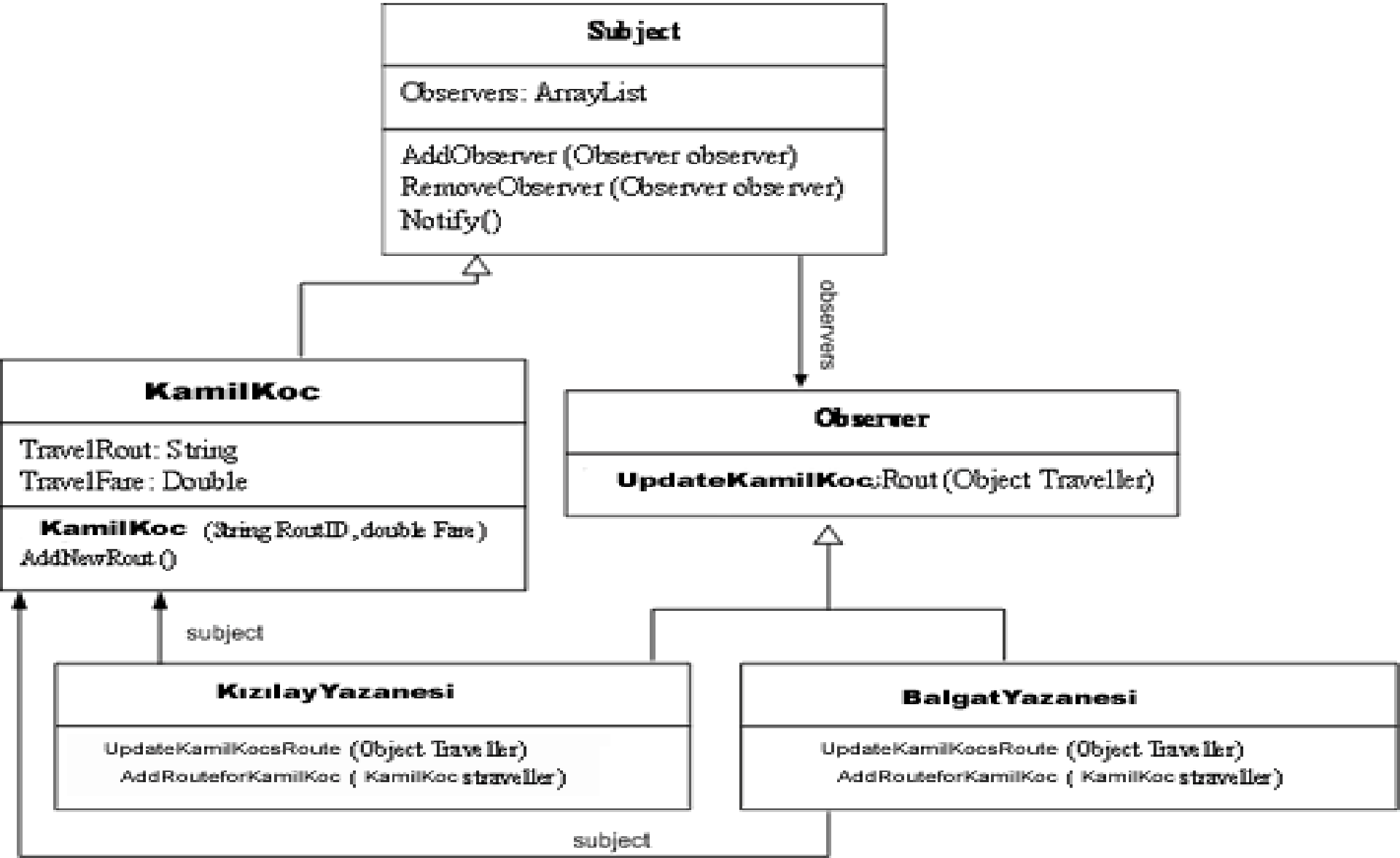
Observer Pattern - UML



Observer Pattern - Collaborations



Observer Pattern based design



Code based example

SUBJECT

```
public abstract class Subject
{
    private ArrayList observers=new ArrayList();
    public void AddObservers(Observer observer)
    {
            observers.Add(observer);
    }
    public void RemoveObserver(Observer observer)
    {
            observers.Remove(observer);
    }
    public void Notify()
    {
            foreach(Observer observer in observers)
            {
                    observer.UpdateKamilKocsRout(this);
            }
    }
}
```

Observer

```
public interface Observer
{
    void UpdateKamilKocsRout(object Traveller);
}
```

Code based example

ConcreteObservers

```
public class KızılayYazanesi:Observer
{
   public void UpdateKamilKocsRout(Object subject)
   {
            if( subject is KamilKoc)
            {
              AddRoutforKamilKoc((KamilKoc)subject);
            }
   }
   private void AddRoutforKamilKoc(KamilKoc
   traveller)
   {
            Console.WriteLine("new rout No. " +
            traveller.TravelRout + " Veri Tabanina
   Eklendi");
   }
}
```

ConcreteObservers

```
public class BalgatYazanesi:Observer
{
   public void UpdateKamilKocsRout(Object subject)
   {
            if( Traveller is KamilKoc)
            {

            AddRoutforKamilKoc((KamilKoc)subject);
            }
   }
   private void AddRoutforKamilKoc(KamilKoc
   traveller)
   {
            Console.WriteLine("new rout No. " +
            traveller.TravelRout + " Veri Tabanina
   Eklendi");
   }
}
```

Code based example

```
class Client
{
  static void Main(string[] args)
  {
        KamilKoc KK= new KamilKoc("EC 2012",2230);
        KizilayYazanesi Kizilay=new KizilayYazanesi();
        BalgatYazanesi Balgat=new BalgatYazanesi();
        KK.AddObservers(Kizilay);
        KK.AddObservers(Balgat);
        KK.AddNewRout();
  }
}
```

Observer Pattern - Consequences

- Loosely Coupled
  - Reuse subjects without reusing their observers, and vice versa
  - Add observers without modifying the subject or other observers
- Abstract coupling between subject and observer
  - Concrete class of none of the observers is known
- Support for broadcast communication
  - Subject doesn’t need to know its receivers
- Unexpected updates
  - Can be blind to changes in the system if the subject is changed (i.e. doesn’t know “what” has changed in the subject)