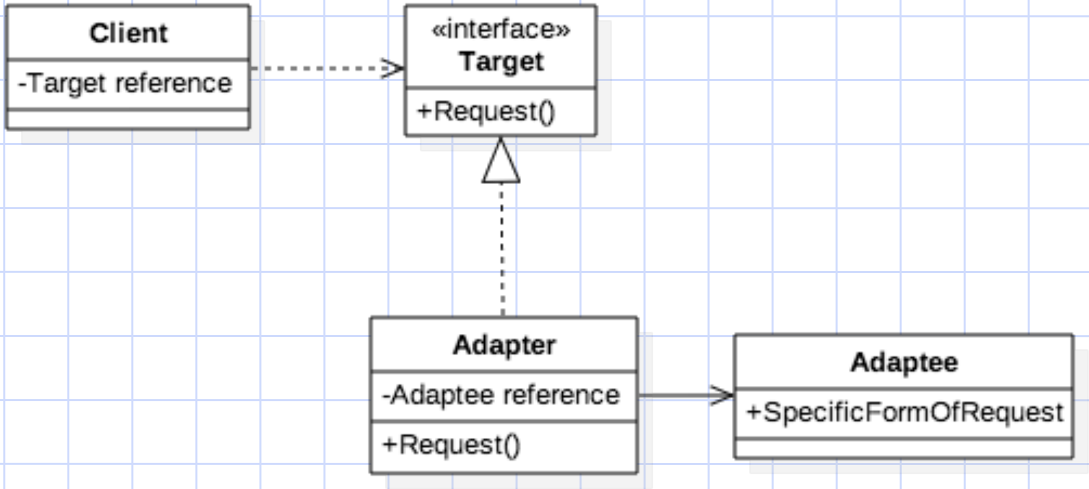
Design Pattern: Adapter

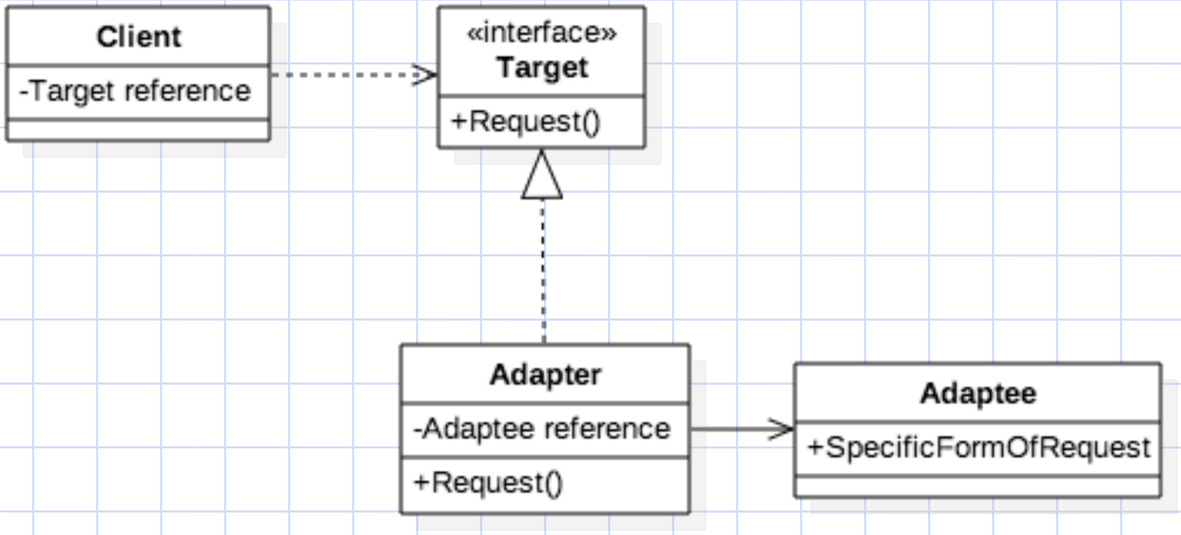
Design Pattern: Adapter

The Adapter Pattern from GoF

- **Intent**
  - Convert the interface of a class into another interface acceptable to the client.
    - Wrap an existing class with a new interface.
  - Allow incompatible classes work together



Participating Classes



Target: defines the domain-specific interface that Client likes.

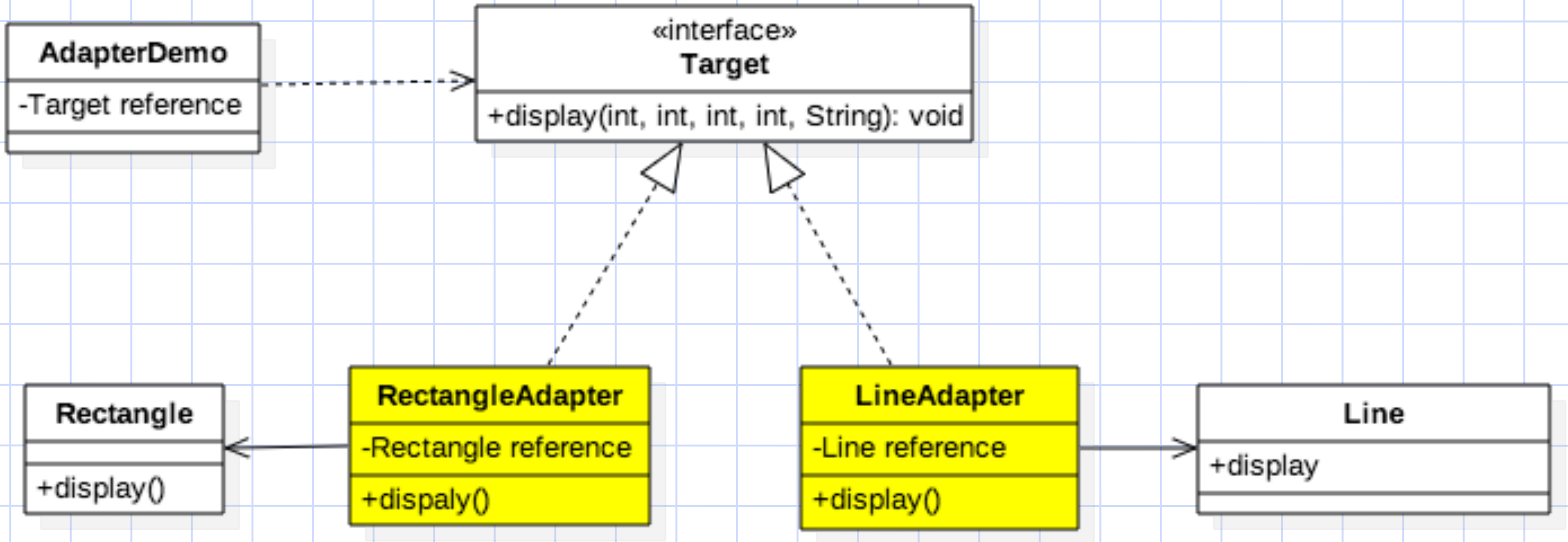
Adapter: adapts the interface Adaptee to the Target interface.

Adaptee: defines an existing interface that needs an adapter to become compatible to target.

Client: collaborates with objects conforming to the Target interface.

Now Lets Learn More By An Example

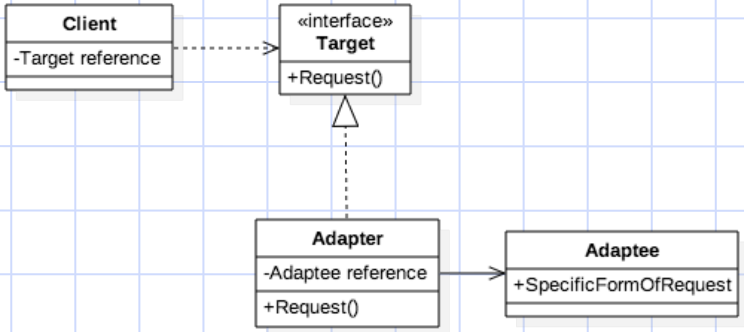
- Lets assume we would like to use a legacy code for a few geometric shapes (line, rectangle), and a client needs to use an adapter, as client's interface doesn't match with the legacy code.



General Template

```
class Adaptee {
   legacyMethod(…) {
   }

};

interface Target{
  clientMethod(…);
}

// a wrapper class
class Adapter implements Target {
   clientMethod(…) {
       adapteeMethod(…)
       // MORE
   }
}
```

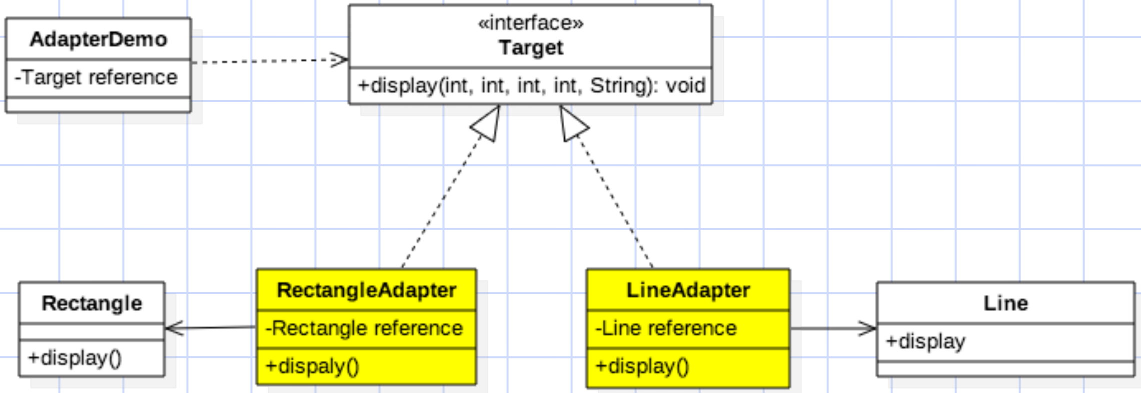
```
class Client{
   useAdapter() {
        Target x = new Adapter();
        x.clientMethod(…);
   }

};
```



Step 1: Creating Legacy Classes (Adaptees)

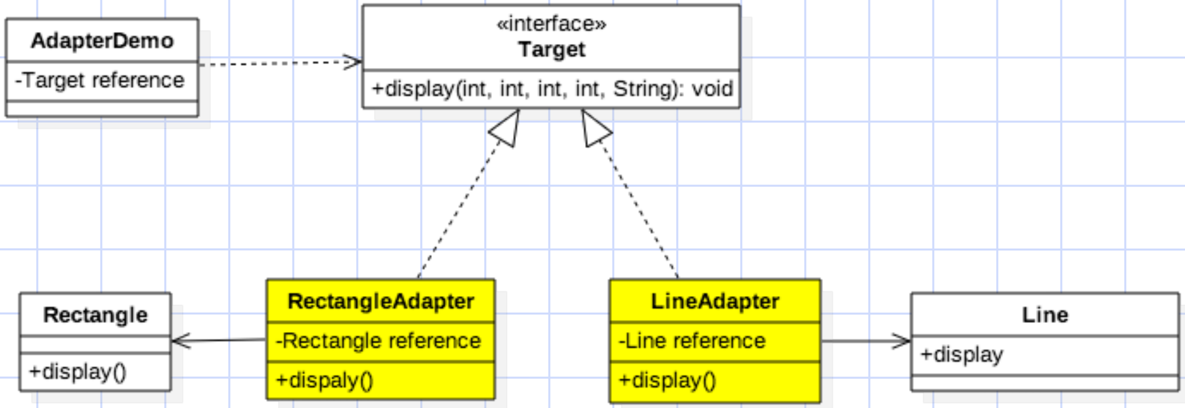
```
class Line {
   public void display(int x1, int y1, int x2, int y2)
  {
            System.out.print("Coordintes of Line are: (" + x1 + ","
                   + y1 + "), and (" + x2 + "," + y2 + ")");
    }
}
```

```
class Rectangle {
   public void display(int x, int y, int width, int height) {
      System.out.print("Coordinates of the Left-corner are (" + x + "," + y +
                              "), width: " + width   + ",  height:  "  +  height);
    }
}
```



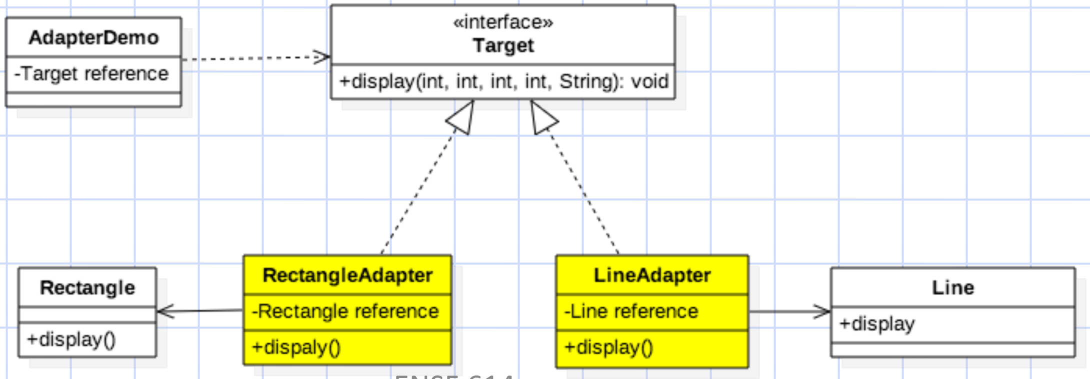
Step 2: Creating Target Interface

```
interface Target
{
   void display(int x, int y, int z, int w, String color);
}
```



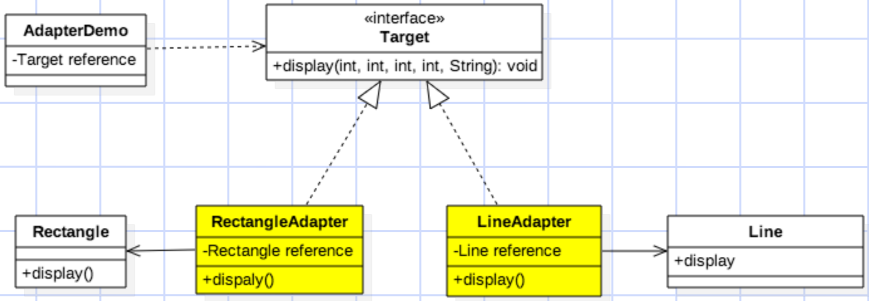
Step 3: Create An Adapter for class Line

```
class LineAdapter implements Target {
   private Line adaptee;

   public LineAdapter(Line line)
  {
      this.adaptee = line;
  }

   @Override
   public void display(int x1, int y1, int x2, int y2, String color)
  {
             adaptee.display(x1,  y1,  x2,  y2);
             System.out.println(" and its Color is: " + color);
  }
}
```



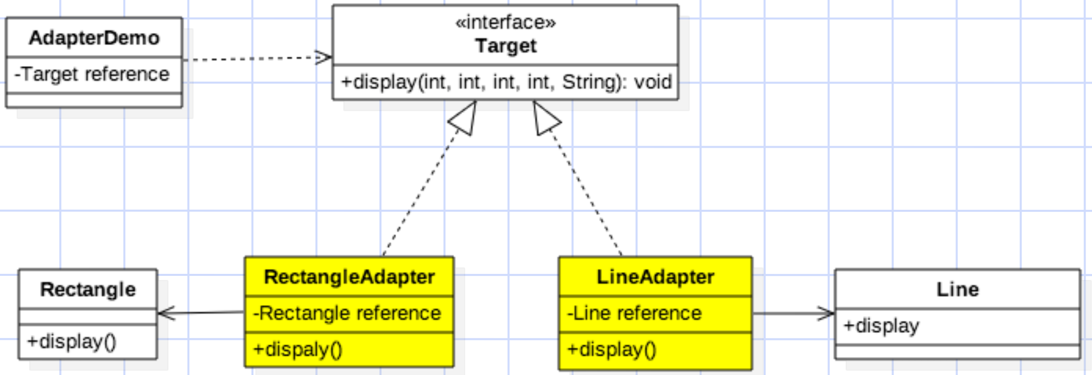
Step 4: Create Another Adapter for class Rectangle

```
class RectangleAdapter implements Target {
   private Rectangle adaptee;

   public RectangleAdapter(Rectangle rectangle) {
            this.adaptee = rectangle;
    }

   @Override
   public void display(int x, int y, int z, int w, String color) {
            adaptee.display(x,  y,  z,  w);
            System.out.println(" and its color is: " + color);
    }
}
```



Step 5: Lets See if it Works

```
public class AdapterDemo {
    public static void main(String[] args)
    {
        Target[] shapes = {new RectangleAdapter(new Rectangle()),
                                                            new LineAdapter(new Line())};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (Target shape : shapes) {
            shape.display(x1, y1, x2, y2, "Red");
        }
    }
}
```

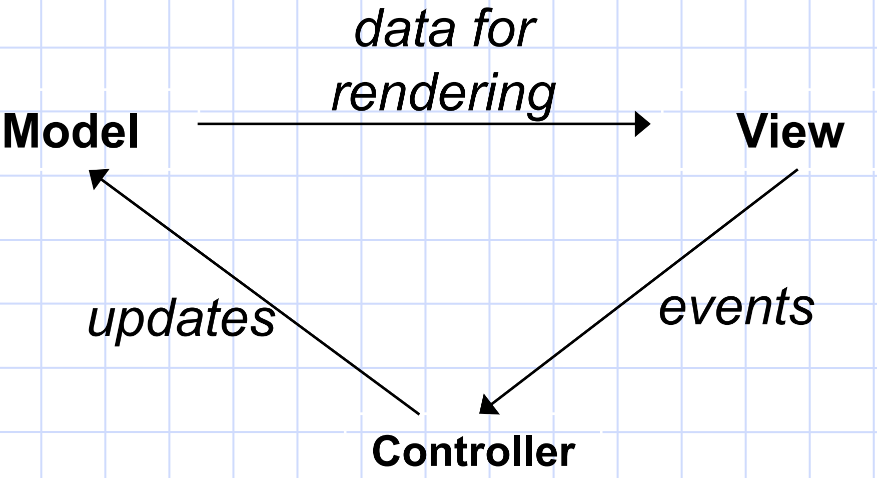


Design Pattern: Observer

objects whose state can be watched

Model-View-Controller

- **model-view-controller (MVC):** common design paradigm for graphical systems



Model

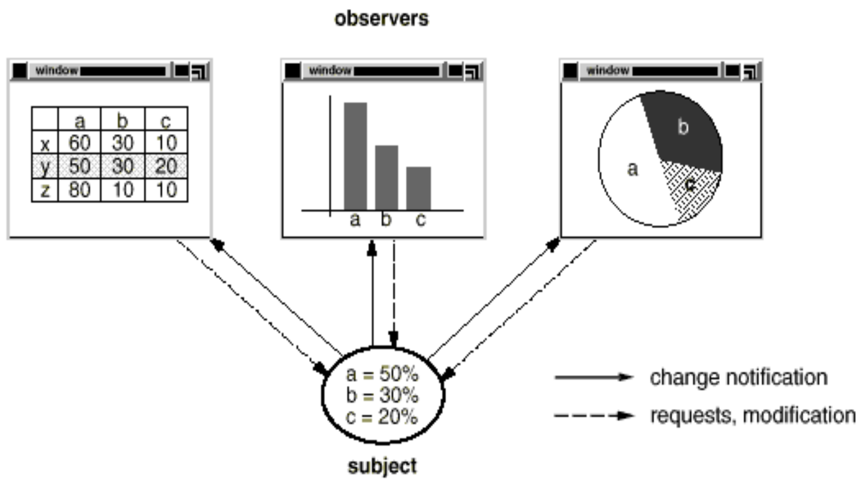
data for rendering

View

updates

events

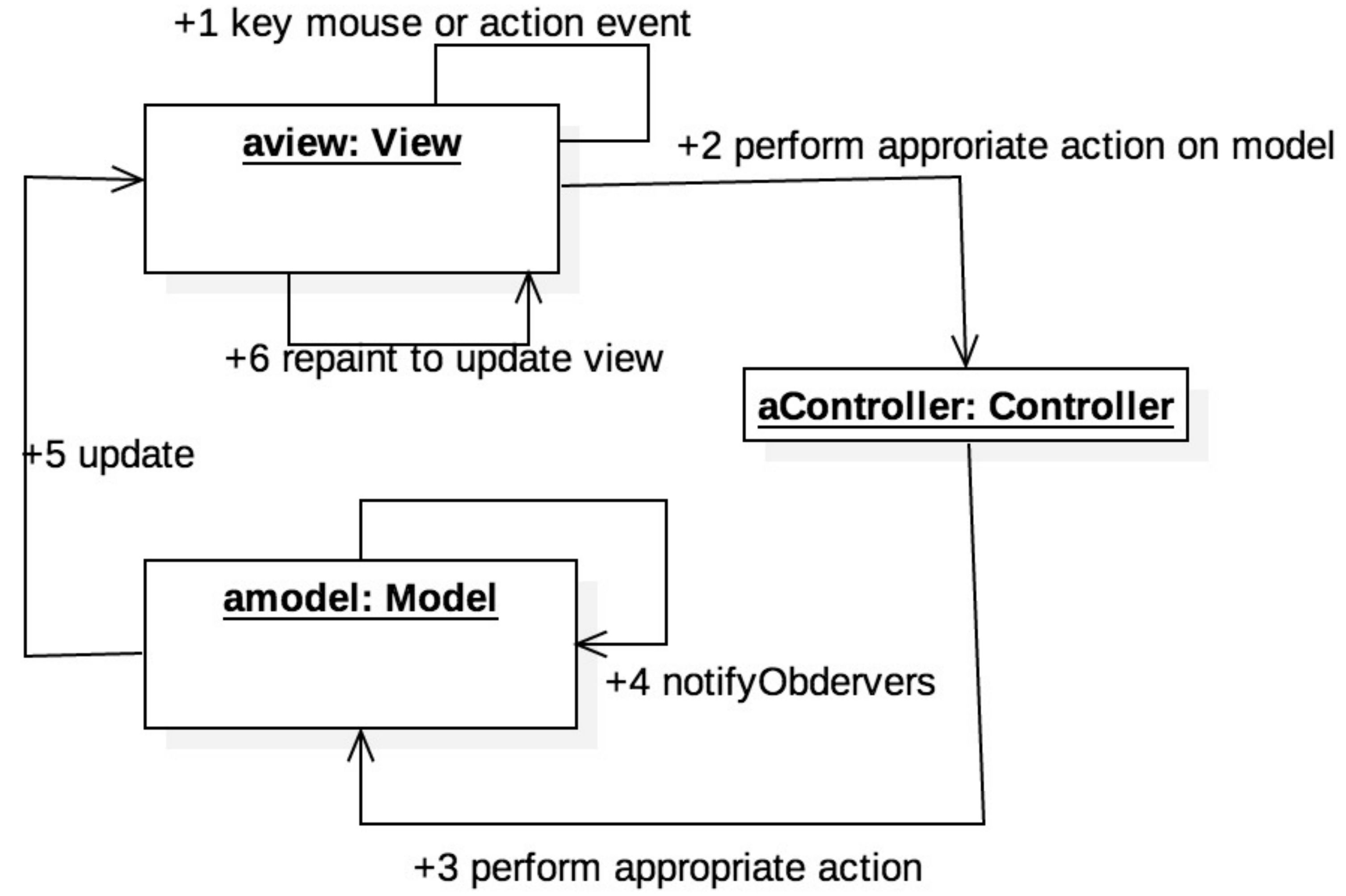
Controller



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

https://www.gofpatterns.com/design-patterns/module6/tradeoffs-implementing-observerPattern.php

Observer Pattern Object Diagram


+1 key mouse or action event
aview: View
+2 perform approriate action on model
+6 repaint to update view
aController: Controller
+5 update
amodel: Model
+4 notifyObdervers
+3 perform appropriate action

MVC Pattren

- **model**: classes in your system that are related to the internal representation of the state of the system
- **view**: classes in your system that display the state of the model to the user
- **controller**: classes that connect model and view

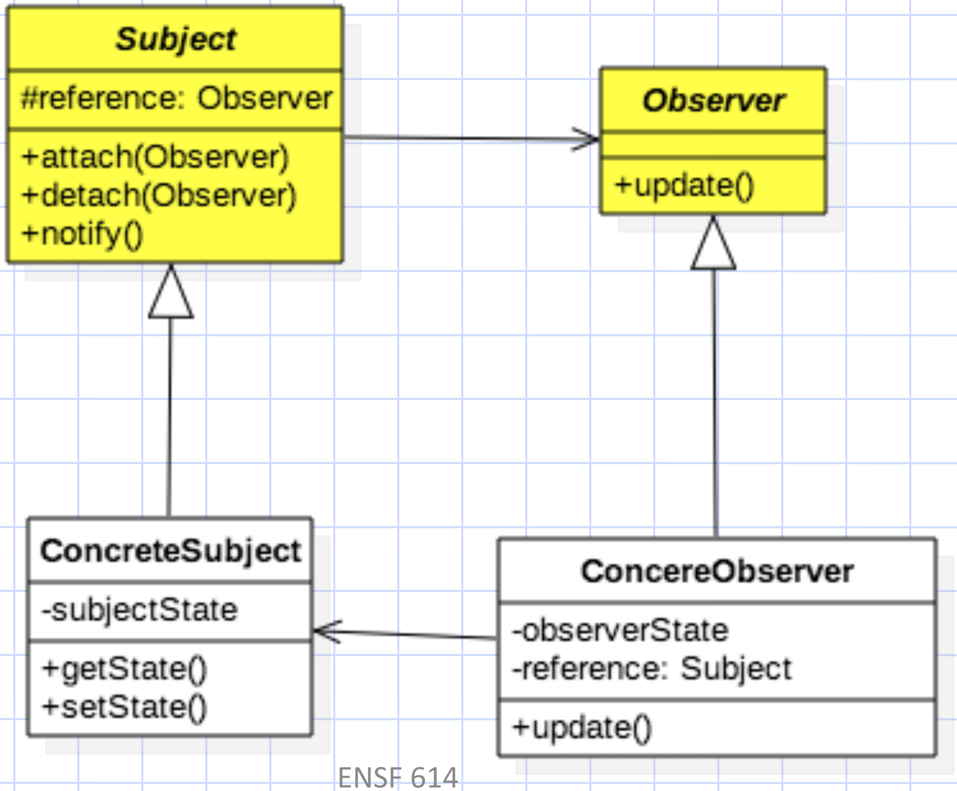
Observer pattern

- **observer**: an object that "watches" the state of another object and takes action when the state changes in some way

- **observable object**: an object that allows observers to examine it (often the observable object notifies the observers when it changes)

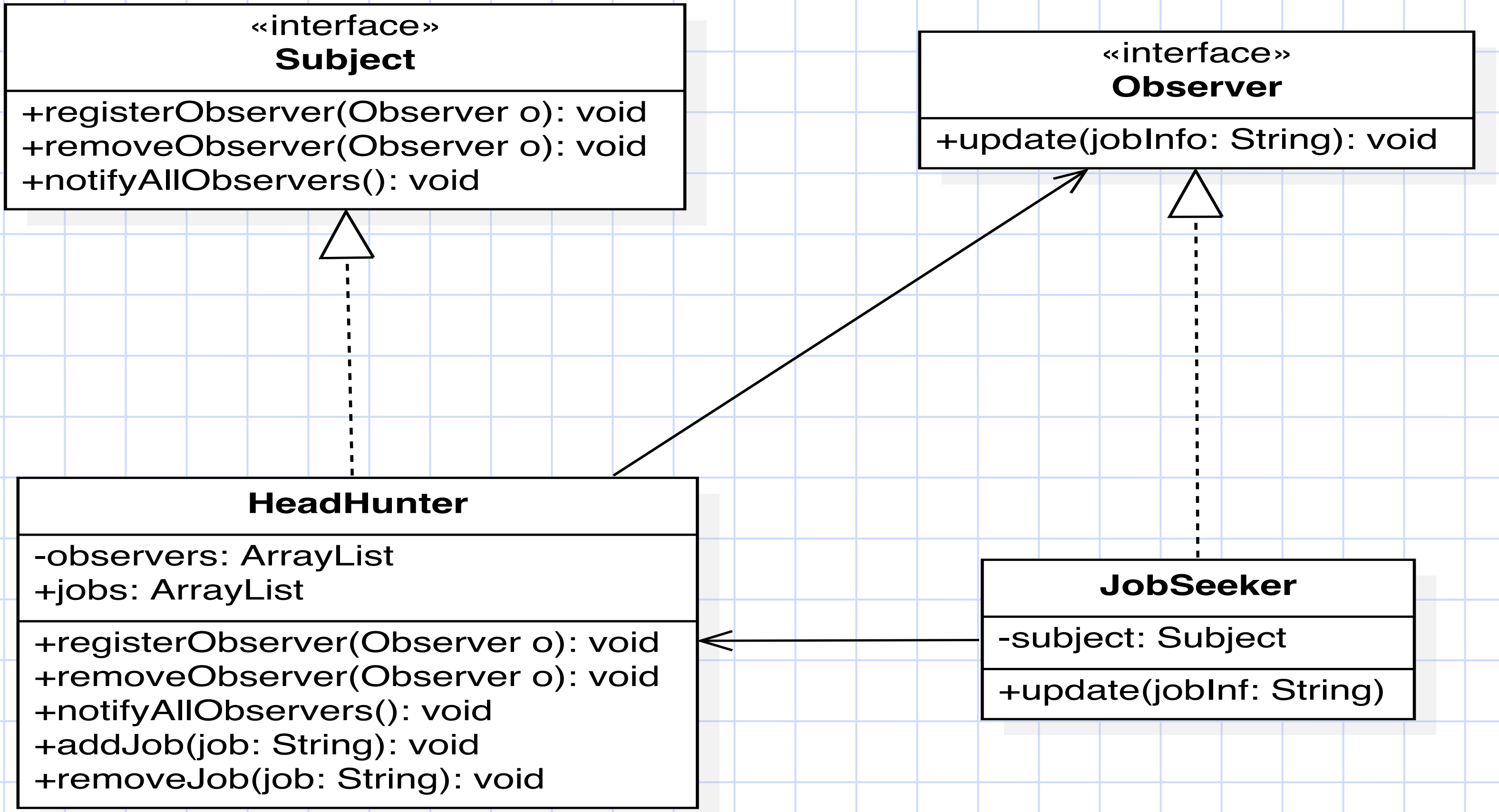
Observer Pattern Model

- The Observer pattern is one of the **behavioural** patterns.
- It's again used to form relationships between objects at runtime.
- This diagram shows Observer Pattern in C++ format using abstract classes instead of interfaces for



Other Applications of Observer Pattern

- Using observer pattern is not limited to GUI presentation; it can be used for any notification system. Here is an example:


«interface»
Subject
+registerObserver(Observer o): void
+removeObserver(Observer o): void
+notifyAllObservers(): void
«interface»
Observer
+update(jobInfo: String): void
HeadHunter
-observers: ArrayList
+jobs: ArrayList
+registerObserver(Observer o): void
+removeObserver(Observer o): void
+notifyAllObservers(): void
+addJob(job: String): void
+removeJob(job: String): void
JobSeeker
-subject: Subject
+update(jobInf: String)

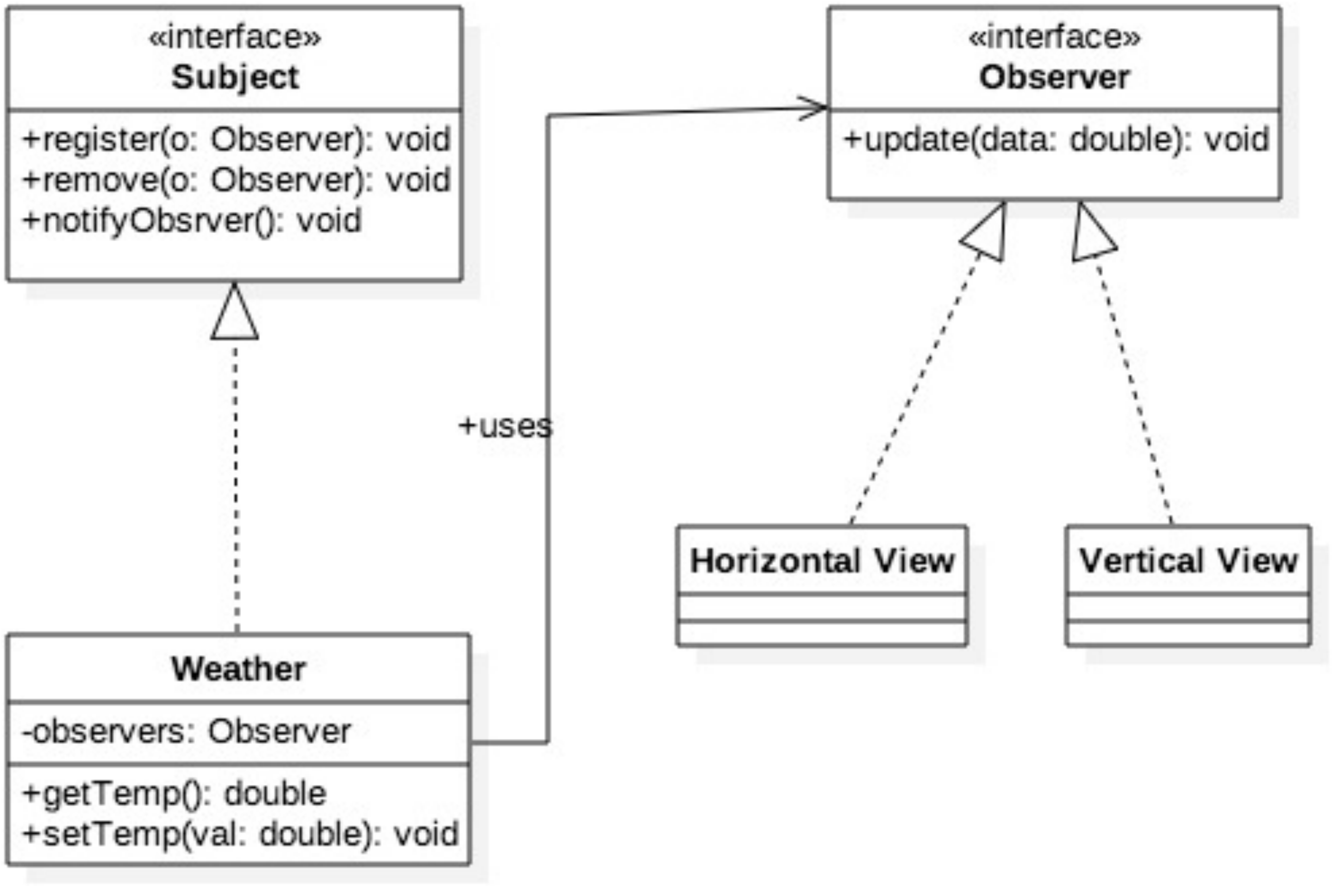
Implementation Steps

1. Create an Observer Interface with a n update method.
2. Create either an interface or abstract class for Subject that contains methods to add or remove an observer object.
3. Create a class that implements Subject
4. Create one or more class that that implements Observer:

Observer Pattern Example

A Class Exercise

Let's try the following model as an example:



A Five-Step Instruction

Implementation Step 1

Create an Observer Interface:

```
public interface Observer {
    public void update(double data);
}
```

Implementation Step 2

- Create either an interface or abstract class Subject:

```
interface Subject {
    public void register(Observer o);
    public void remove(Observer o);
    public void notifyObserver();

}
```

Implementation Step 3 (cont'd)

```
class Weather implements Subject {
   private double temp;
   private ArrayList <Observer> observers;

   public Weather(double t) {
       observers = new ArrayList<Observer>();
       temp  =  t;
    }

   public void register(Observer o) {
             observers.add(o);
             o.update(temp);
   }
    public void remove(Observer o) {

    }
```

```
 public void notifyObserver() {
   for(int i = 0; i < observers.size(); i++)
  {
       Observer o = observers.get(i);
       o.update(temp);
    }
 }

 public double getTemp(){
       return temp;
  }

   public void setTemp(double t){
            temp = t;
            notifyObserver();
      }
} // END OF CLASS WEATHER
```

Implementation Step 4

- Create a set of view classes that implement observer:

```
class HorizontalDisplay implements Observer {
        double temp;
        Subject weather;

        public HorizontalDisplay(Subject w) {
                weather = w;
                weather.register(this);
        }

        @Override
        public void update(double temp) {
                this.temp = temp;
                display();
        }

        public void display(){
                // code to display horizontally
        }
}
```

```
class VerticalDisplay implements Observer, {
        double temp;
        Subject weather;

        public VerticalDisplay(Subject w) {
                weather = w;
                weather.register(this);
        }

        @Override
        public void update(double temp) {
                this.temp = temp;
                display();
        }

        public void display(){
                // code to display vertically

        }
}
```

You can assume there are more View classes that implement Observer

Implementation Step 5

- Create a client class the uses the observers:

```
public class Cient {
   public static void main(String []s) {
      Weather w = new Weather(34.5);
      HorizontalDisplay h = new HorizontalDisplay(w);
      VerticalDisplay v = new VerticalDisplay(w);
      w.setTemp(55);
      h.display();   // displays horizontally
      v.display();   // displays vertically
   }
}
```

How Easy is to Add New Observer?

```
class DiagonalDisplay implements Observer {
        double temp;
        Subject weather;

        public DiagonalDisplay(Subject w) {
                weather = w;
                weather.register(this);
        }

        @Override
        public void update(double temp) {
                this.temp = temp;
                display();
        }

        public void display(){
                // code to display temp diagonally
        }
}
```

```
public class Cient {
   public static void main(String []s) {
     Weather w = new Weather(34.5);
     HorizontalDisplay h =
                 new HorizontalDisplay(w);
     VerticalDisplay v = new VerticalDisplay(w);
    DiagonalDisplay d = new DiagonalDisplay(w);
      w.setTemp(55);
      h.display();   // displays horizontally
      v.display();   // displays vertically
      d.display();   // displays diagonally
   }
}
```

Benefits of Observer Pattern

- Supports loose coupling between objects that interact with each other.
  - abstract coupling between subject and observer;
- Allows sending data to other objects without any change to the Subject or Observer classes. Observers need only to register with the Subject
- dynamic relationship between subject and observer:
  - Relationship can be established at run time
  - Observers can be added/removed at anytime
  - Observers can be extended and reused individually
- Automatic Broadcast:
  - notification is broadcasted automatically to all interested objects that subscribed to it.