Software Engineering (Design Patterns)

Lina YE

CentraleSupélec

https://www.lri.fr/~linaye/GL.html

lina.ye@centralesupelec.fr

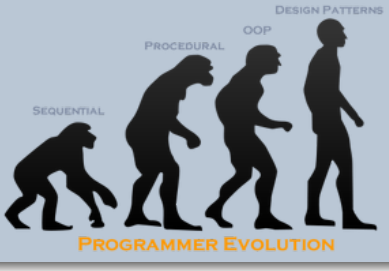
Sequence 3, 2017-2018

Plan

1. Introduction
2. Creational Patterns
3. Structural Patterns
4. Behavioral Patterns
5. Conclusion

Background

Evolution of Program



Definition

What is a design pattern

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [C. Alexander 77]”

Definition

What is a design pattern

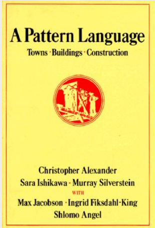
- ”Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [C. Alexander 77]”
- ”The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.[E. Gamma 97] ”

Definition

Essential elements

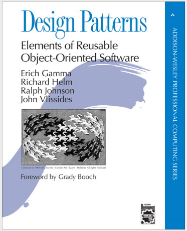
- Pattern name: describes a design problem, its solutions, and consequences in a word or two
- Problem: describes when to apply the pattern
- Solution: describes the elements that make up the design, their relationship, responsibilities and collaborations
- Consequences: the results and trade-offs of applying the pattern

Sources



- "A Pattern Language: Towns, Buildings, Construction": Christopher Alexander, Sara Ishikawa and Murray Silverstein

- "Design Patterns, Elements of Reusable Object-Oriented Software": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides



- wikipedia: design patterns

Categories

Pattern can be divided into three categories with respect to their purpose.

- Creational patterns: the process of object creation
- Structural patterns: the composition of classes or objects
- Behavioral patterns: the ways in which classes or objects interact and distribute responsibility

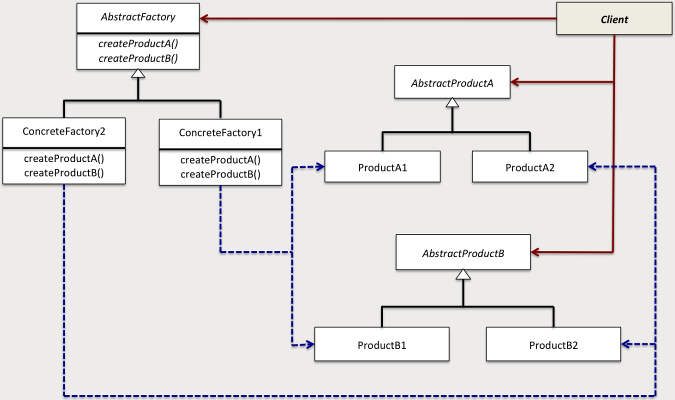
List

- Abstract Factory: provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder: separate the construction of a complex object from its representation.
- Prototype: specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton: ensure a class only has one instance, and provide a global point of access to it.

Abstract factory

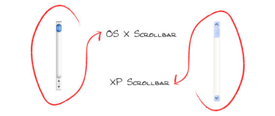
- Problem: create the group of objects without knowing their exact concrete class
- Example: program with a GUI Mac and Windows (manager the elements of graphic interface)
- Solution: the creation of the object is transferred to another specific class

Abstract factory: structure



User interface toolkit

Windows
Mac

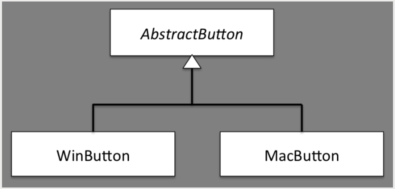


Factory code

```
public abstract class Button {
           public abstract void paint();
}

public class WinButton extends Button {
           public void paint(){ ... }
}

public class MacButton extends Button {
          public void paint(){ ... }
}
```

Factory code

```
public abstract class Button {

          public abstract void paint();
}

publi

}

publi

          public void paint(){ ... }
}
```

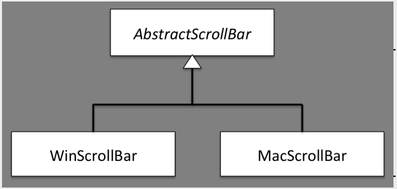


Factory code

```
public abstract class ScrollBar {

           public abstract void paint();
}

public class WinScrollBar extends ScrollBar {

           public void paint(){ ... }
}

public class MacScrollBar extends ScrollBar {

          public void paint(){ ... }
}
```

Factory code

```
public abstract class ScrollBar {
          public abstract void paint();
}

publi                                       Bar {


}

publi                                       Bar {

          public void paint(){ ... }
}
```

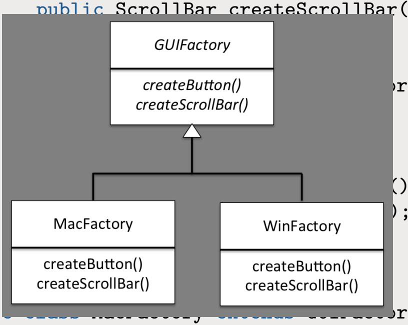


Factory code

```
public abstract class GUIFactory {
        public Button createButton();
        public ScrollBar createScrollBar();
}

public class WinFactory extends GUIFactory {
       public Button createButton(){
               return new WinButton();
       }
       public ScrollBar createScrollBar(){
               return new WinScrollBar();
       }
}

public class MacFactory extends GUIFactory {
         ...
}
```

Factory code

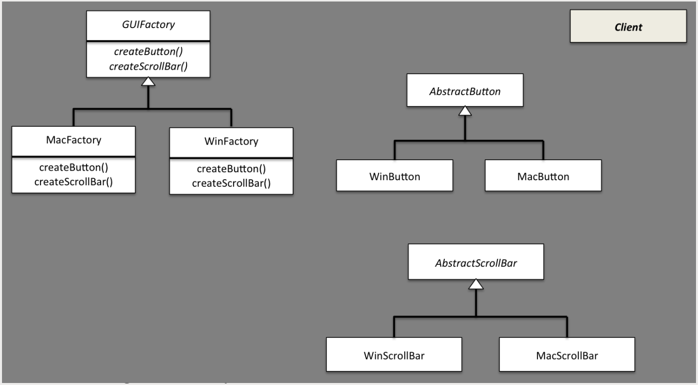
```
public abstract class GUIFactory {
        public Button createButton();
        public ScrollBar createScrollBar();
}

publi                                          ory {




                                               (){
                                               );


}

public class MacFactory extends GUIFactory {
         ...
}
```



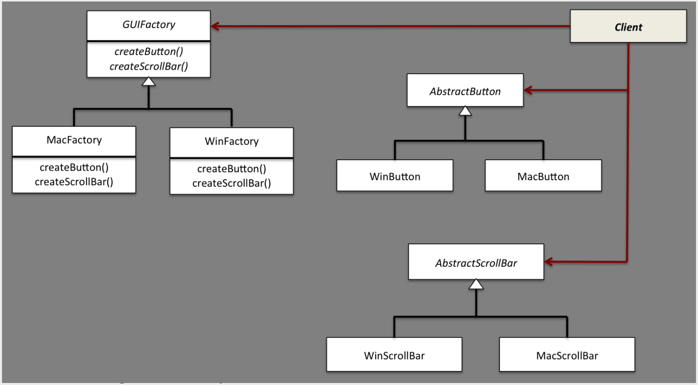
Factory code

```
GUIFactory guiFactory;
Button button;
ScrollBar scrollBar;

if (isMac()) {
          guiFactory=new MacFactory();
}

if (isWin()) {
          guiFactory=new WinFactory();
}

button=guiFactory.createButton();
scrollBar=guiFactory.createScrollBar();
```

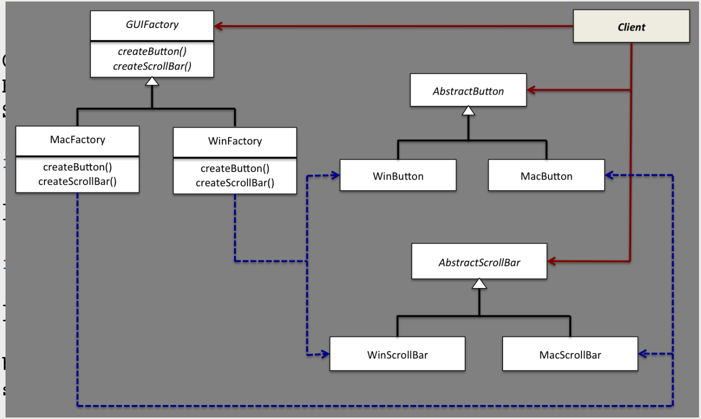
Factory code



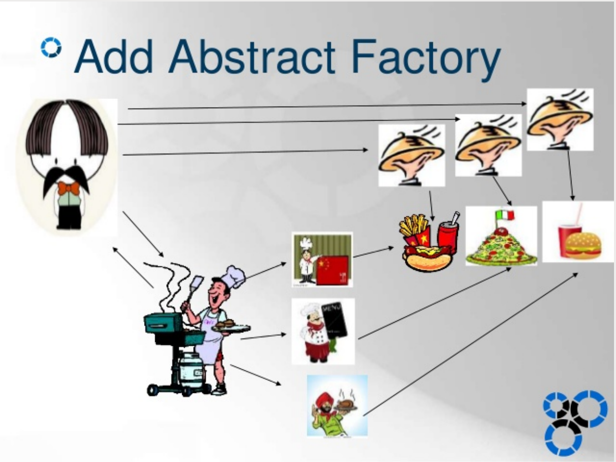
Factory code



Factory code



Abstract Factory



Add Abstract Factory

Singleton

I am the king of my kingdom , Only one king will manage the kingdom.

Only one instance will manage the application

Singleton

- Problem: guarantee that one class has only one instance; one unique and global access point to this instance
- Example: class represents the configuration of a system; class manipulates the window of one application
- Solution: intercept requests to create new objects; define a static method that returns the instance

Singleton code

```
public class Singleton {
         private static Singleton instance;

         private Singleton() {}

         public static Singleton getInstance() {
                if (Singleton.instance==null) {
                       Singleton.instance=new Singleton();
                }
             return Singleton.instance;
         }
}
```

```
Singleton s1=Singleton.getInstance();
Singleton s2=Singleton.getInstance();
System.out.println(s1==s2);
```

List

- Adapter: convert the interface of a class into another interface clients expect.
- Bridge: decouple an abstraction from its implementation so that the two can vary independently.
- Composite: compose objects into tree structures to represent part-whole hierarchies.
- Decorator: attach additional responsibilities to an object dynamically to extend functionality.
- Facade: provide a unified interface to a set of interfaces in a subsystem.

Composite

Problem

- Represent the set of objects by the tree structure
- Let clients treat individual objects and compositions of objects uniformly.

Composite

Problem

- Represent the set of objects by the tree structure
- Let clients treat individual objects and compositions of objects uniformly.

Example

- A draw application (a line and the set of lines should be treated in the same way)
- An arithmetic expression

Composite

Problem

- Represent the set of objects by the tree structure
- Let clients treat individual objects and compositions of objects uniformly.

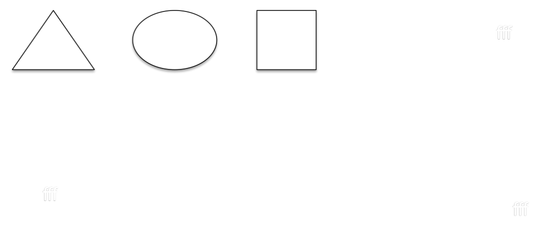
Example

- A draw application (a line and the set of lines should be treated in the same way)
- An arithmetic expression

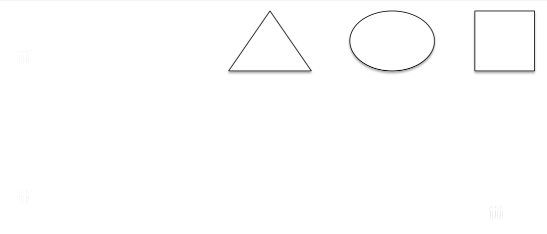
Solution

- an abstract class representing an object or a tree
- a child class representing the leafs
- a child class representing a node of the tree

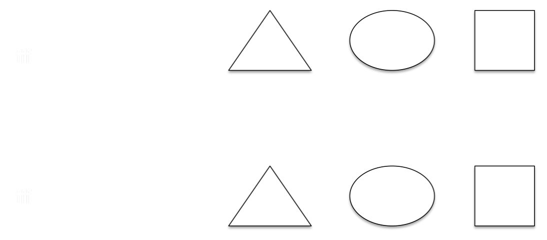
Composite: example



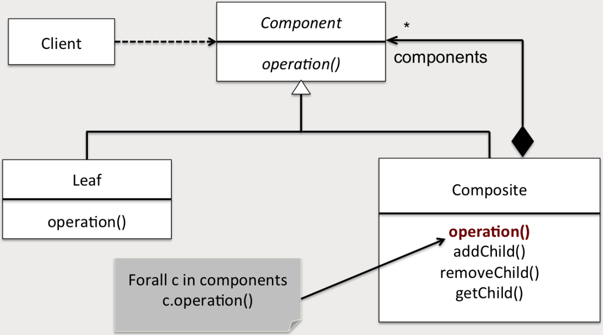
Composite: example



Composite: example



Composite: structure



Composite

Pattern objects

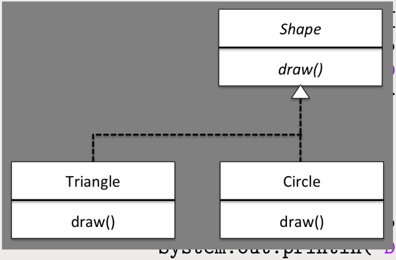
- Component: interface for all objects in this pattern (interface or an abstract class with some methods common to all objects).
- Leaf: the behaviors for the primitive elements. It is the building block and implements base component.
- Composite: consists of leaf or composite elements (children) and implements the operations by calling those of its children.

Composite code

```
public interface Shape {
          public void draw (String fillColor);
}

public class Triangle implements Shape {
          public void draw(String fillColor) {
                   System.out.println("Drawing Triangle
                   with color"+fillColor);
          }
}

public class Circle implements Shape {
          public void draw(String fillColor) {
                   System.out.println("Drawing Circle
                   with color"+fillColor);
          }
}
```

Composite code

```
public interface Shape {
        public void draw (String fillColor);
}

publi                                   {
                                        lor) {
                                        rawing Triangle
                                        );

}

publi
                                        lor) {
            ystem.out.println(" Drawing Circle
            with color"+fillColor);
        }
}
```

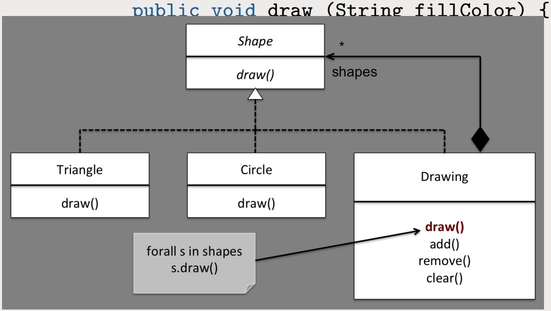


Composite code

```
public class Drawing implements Shape {
          private List<Shape> shapes = new ArrayList<Shape>();

          public void draw (String fillColor) {
               for(Shape sh : shapes)
                     sh.draw(fillColor);
          }

          public void add (Shape s) {
                this.shapes.add(s);
          }

          public void remove (Shape s) {
               shapes.remove(s);
          }

          public void clear () {
              shapes.clear();
              System.out.println("Clearing all shapes");
          }
}
```

Composite code

```
public class Drawing implements Shape {
        private List<Shape> shapes = new ArrayList<Shape>();

        public void draw (String fillColor) {
```



```
        public void clear () {
            shapes.clear();
            System.out.println("Clearing all shapes");
        }
}
```

Composite code

```
Shape tri = new Triangle();
Shape tri1 = new Triangle();
Shape cir = new Circle();
Drawing drawing = new Drawing();
drawing.add(tri);
drawing.add(tri1);
drawing.add(cir);
drawing.draw(“Red”);
drawing.clear();
drawing.add(tri);
drawing.add(cir);
drawing.draw(“Green”);
```

Composite code

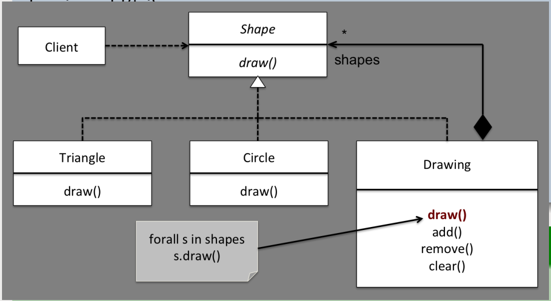
```
Shape tri = new Triangle();
Shape tri1 = new Triangle();
Shape cir = new Circle();
Drawing drawing = new Drawing();
drawing.add(tri);
drawing.add(tri1);
drawing.add(cir);
drawing.draw(“Red”);
drawing.clear();
drawing.add(tri);
drawing.add(cir);
drawing.draw(“Green”);
```

Results

```
Drawing Triangle with color Red
Drawing Triangle with color Red
Drawing Circle with color Red
Clearing all the shapes from drawing
Drawing Triangle with color Green
Drawing Circle with color Green
```

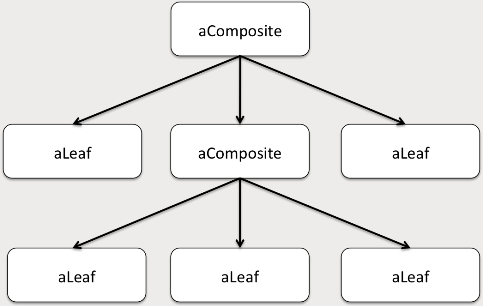
Composite code

```
Shape tri = new Triangle();
Shape tri1 = new Triangle();
Shape cir = new Circle();
Drawing drawing = new Drawing();
drawing.add(tri);
```



```
Drawing Triangle with color Red
Drawing Circle with color Red
Clearing all the shapes from drawing
Drawing Triangle with color Green
Drawing Circle with color Green
```

Composite hierarchy



Facade

Problem

- unified interface by defining a higher-level interface that makes the subsystem easier to use.

Facade

Problem

- unified interface by defining a higher-level interface that makes the subsystem easier to use.

Example

- recuperate old codes that are hard to be refactored
- several complex interfaces

Facade

Problem

- unified interface by defining a higher-level interface that makes the subsystem easier to use.

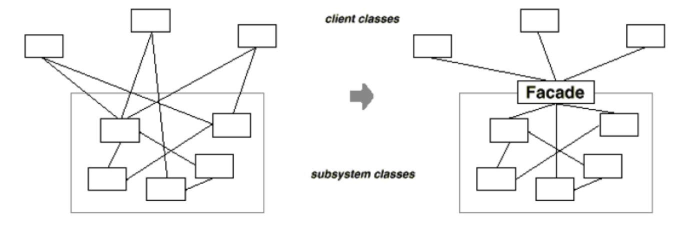
Example

- recuperate old codes that are hard to be refactored
- several complex interfaces

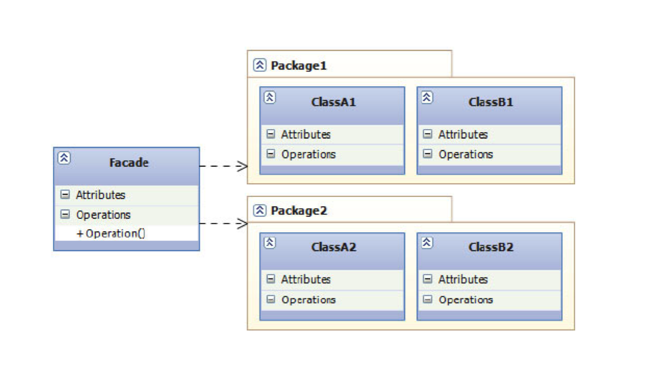
Solution

- one high-level class that reuses the useful functionalities

Facade: goal



Facade: structure



Facade code

```
class ComplexeA1{
        public void actionA11 () { ... }
        public void actionA12 () { ... }
}

class ComplexeA2{
        public void actionA21 () { ... }
        public void actionA22 () { ... }
}

class ComplexeB1 {
        public void actionB11 () { ... }
        public void actionB12 () { ... }
}

class ComplexeB2 {
        public void actionB21 () { ... }
        public void actionB22 () { ... }
}
```

Facade code

```
class Facade {
         complexeA1 a1;
         complexeB2 b2;
         Facade(){
              a1=new ComplexeA1();
              b2=new ComplexeB2();
         }
         void performActionsAB1 () {
              a1.actionA11();
              b2.actionB22();
         }
          void performActionsAB2 () {
              a1.actionA11();
              a1.actionA12();
              b2.actionB21();
         }
}
```

Introduction

Creational Patterns

Structural Patterns

Behavioral Patterns

Conclusion

Facade code

```
class Facade {
         complexeA1 a1;
         complexeB2 b2;
         Facade(){
              a1=new ComplexeA1();
              b2=new ComplexeB2();
         }
         void performActionsAB1 () {
              a1.actionA11();
              b2.actionB22();
         }
          void performActionsAB2 () {
              a1.actionA11();
              a1.actionA12();
              b2.actionB21();
         }
}
```

```
Facade facade=new Facade();
facade.performActionAB1();
```

List

- Command: an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.
- Interpreter: define a representation for the grammar of a given language with an interpreter to interpret sentences in this language.
- Iterator: provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

List

- Observer: define a one-to-many dependency between objects so that one object changes state, all its dependents are notified and updated automatically.
- State: allow an object to alter its behavior when its internal state changes.
- Strategy: define a family of algorithms such that they are interchangeable.
- Visitor: separate an algorithm from an object by defining a new operation without changing the classes of the elements on which it operates.

Strategy

Problem

- Allow different algorithms for the same object

Strategy

Problem

- Allow different algorithms for the same object

Example

- Robots with different behaviors
- Different sorting algorithms

Strategy

Problem

- Allow different algorithms for the same object

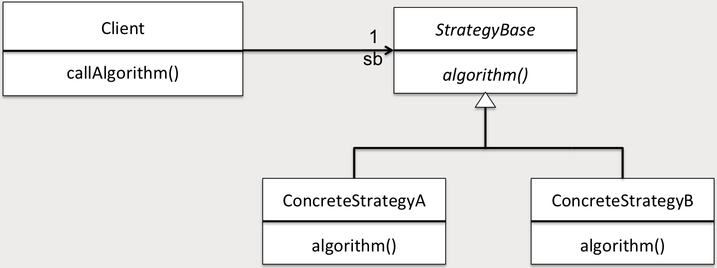
Example

- Robots with different behaviors
- Different sorting algorithms

Solution

- The algorithms are encapsulated by classes.
- The class using the algorithm will have an instance of this class as attribute.

Strategy: structure

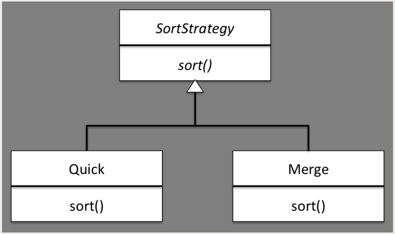


Strategy code

```
abstract class SortStrategy {
          abstract void sort (List list);
}

class Quick extends SortStrategy {
          void sort (List list) { ... }
}

class Merge extends SortStrategy {
          void sort (List list) { ... }
}
```

Strategy code

```
abstract class SortStrategy {
                                      ;
}

class

}

class

}
```

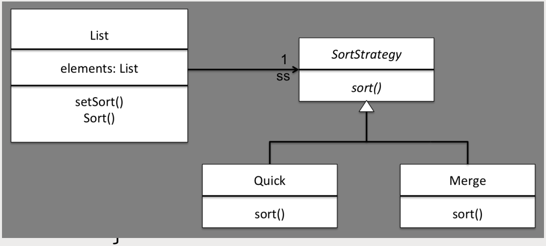


Strategy code

```
class List {

        private List elements;
        private SortStrategy ss;

        public void setSort (SortStrategy s) {
                 this.ss=s;
        }

        public void sort () {
                 this.ss.sort(this.elements);
        }

}
```

Strategy code

```
class List {
```



```
          }

}
```

Observer

Problem

- Allow following the state modification of an object

Observer

Problem

- Allow following the state modification of an object

Example

- Graphic interface depending on the application engine.
- Model-View-Controller

Observer

Problem

- Allow following the state modification of an object

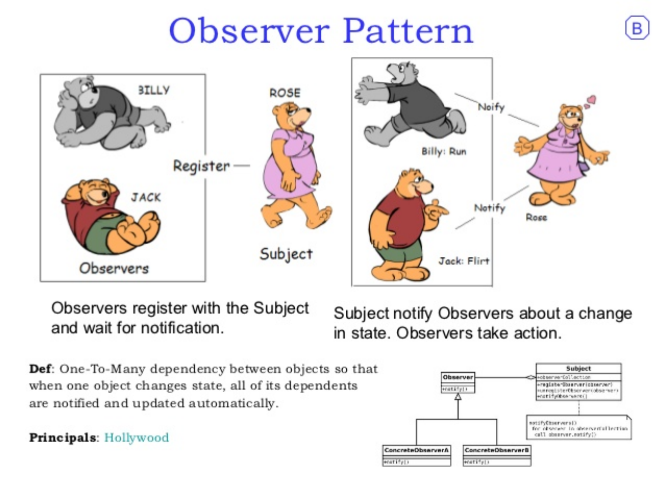
Example

- Graphic interface depending on the application engine.
- Model-View-Controller

Solution

- The observable is linked to all its observers
- The observable should notify all its observers when its state is changed
- Each observer requires the necessary information

Observer: example



Observer Pattern

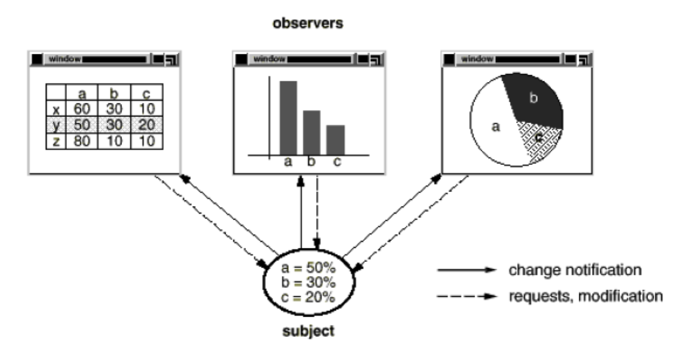
Observers register with the Subject and wait for notification.

Subject notify Observers about a change in state. Observers take action.

Def: One-To-Many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

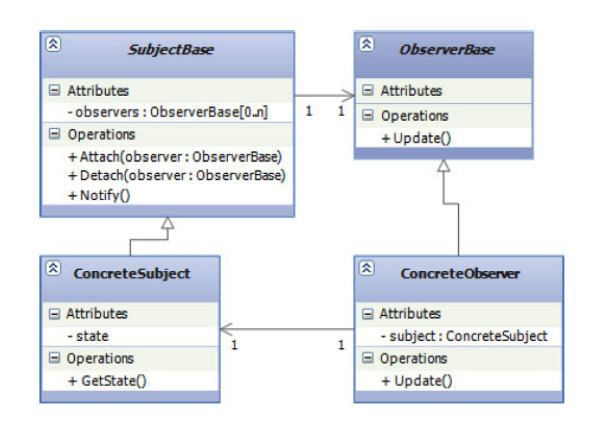
Principals: Hollywood

Observer: example



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

Observer: structure

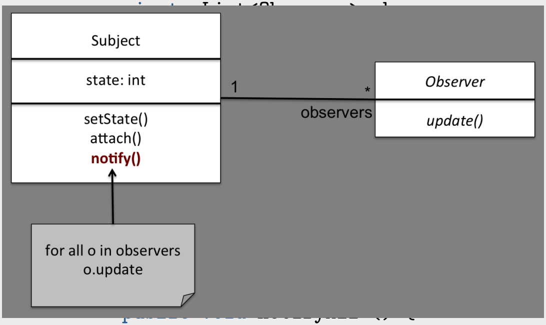


Observer code

```
class Subject {

        private List<Observer> observers;
        private int state;

        public void setState (int val) {
                 state=val;
                 notifyAll();
         }

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

         public void notifyAll () {
                 for (Observer obs: observers)
                        obs.update();
         }
}
```

Observer code

```
class Subject {
```



Subject

state: int

setState()
attach()
notify()

1

*

observers

Observer

update()

for all o in observers
o.update

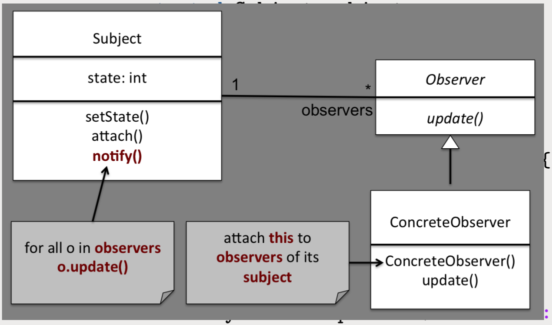
```
       public void notifyAll () {
              for (Observer obs: observers)
                    obs.update();
       }
}
```

Observer code

```
abstract class Observer {

         protected Subject subject;
         public abstract void update();
}

class ConcreteObserver {

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

         public void update () {
                 System.out.println("new value: "
                 +subject.getValue());
         }

}
```

Observer code

```
abstract class Observer {
```



```
                                                                      {

                                                                      :  "
                        +subject.getValue());
               }

}
```

MVC

- Pattern used to implement an user’s interface
- Widely adopted in web applications

MVC

- Pattern used to implement an user's interface
- Widely adopted in web applications

Separate the elements of an application into three groups

- View: output representation of information that allows the interaction with users
- Model: central component that expresses the application's behaviors, independent of the user interface
- Controller: accepts input and converts it to commands for the model or view

MVC

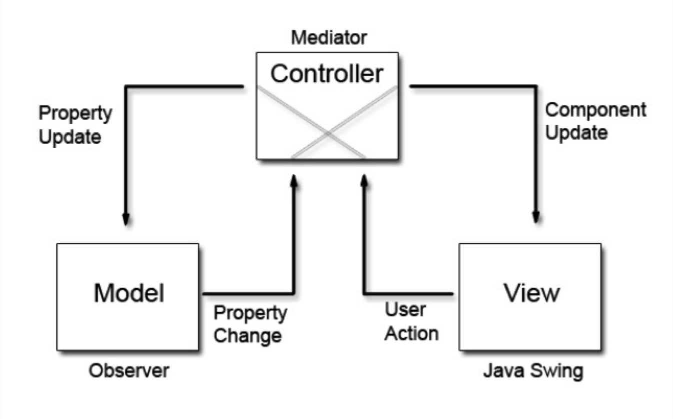
- Pattern used to implement an user's interface
- Widely adopted in web applications

Separate the elements of an application into three groups

- View: output representation of information that allows the interaction with users
- Model: central component that expresses the application's behaviors, independent of the user interface
- Controller: accepts input and converts it to commands for the model or view

- Simultaneous development with separate parts
- Ease of modification
- Multiple views or controls for a model

MVC


Mediator
Controller
Property Update
Component Update
Model
Property Change
User Action
View
Observer
Java Swing

MVC in Java

- Packages are used to represent model/view/controller
- The elements in each group are represented by classes

Example

Problem

- An application to draw geometrical forms with the library SWING

Example

Problem

- An application to draw geometrical forms with the library SWING

One possible solution

- a view package containing
  - one window class
  - one draw surface class
  - one class to manage mouse events

Example

Problem

- An application to draw geometrical forms with the library SWING

One possible solution

- a view package containing
  - one window class
  - one draw surface class
  - one class to manage mouse events
- a model package containing
  - a form class with its sub-classes like polygon, circle, etc.
  - an arrangement class (contains an array of forms)
  - ...

Example

Problem

- An application to draw geometrical forms with the library SWING

One possible solution

- a view package containing
  - one window class
  - one draw surface class
  - one class to manage mouse events
- a model package containing
  - a form class with its sub-classes like polygon, circle, etc.
  - an arrangement class (contains an array of forms)
  - ...
- a controller package containing
  - a controller class

Conclusion

- Simple and elegant solutions to specific problems
- Useful for designing reusable object-oriented software
- Several key steps:
  1. factor pertinent objects into classes at the right granularity
  2. define class interfaces and inheritance hierarchies
  3. establish key relationships among them