Software Engineering and Design

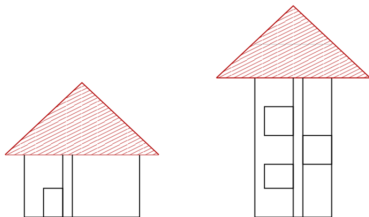
Master 2
Pôles Universitaires Français

Marie Beurton-Aimar
Université de Bordeaux

Design Patterns: The Tale

- An architect : Christopher Alexander [1] has developed the idea to identify objects given:

"A solution to a problem in a context"



- To define pattern, it is to give *why* and *how* building each solution.

[1] "Timeless way of building", 1979.

Design Patterns

- A book written by the Gang of Four (GoF) : Gamma Erich (PhD thesis), Richard Helm, Ralph Johnson and John Vlissides (1995).
- A new *"culture"* in programming community.
- Patterns are devices that allow programs to **share knowledge** about their design. In our daily programming, we encounter **many problems** that **have occurred**, and **will occur again**.
- Patterns design object which often **do not exist** as **entities into the human cognitive system**.

Design Patterns

- We need museum where we can look at “the best programming solutions”
- Doug Lea :
  Why bother writing patterns that just boil down to advice my grandmother would give me?
  Because some patterns are so good and useful that even your grandmother knows them. Writing them down makes the context, value and implications of the advice clearer than your grandmother probably did.

Bibliography

- *The Design Patterns Java Companion* James W. Cooper
  `www.patterndepot.com/put/8/JavaPatterns.htm`
- *Object-Oriented Software Development Using Java*
  Xiaoping Jia, Ph.D
  `se.cs.depaul.edu/Java/chap10.html`
- http://norvig.com/design-patterns/ppframe.htm
- http://users.csc.calpoly.edu/ dbut-
  ler/tutorials/winter96/patterns/

Just a break - Model View Controller

- G. Krasner et S. Pope (1988) - A cookbook for using the model-view controller user interface paradigm in SmallTalk-80 - J. of OOP
    - Model of data : the application
    - Representation of the model, ex : screen printing
    - Control : definition of the protocol. *subscribe/notify*. Each time data change, model notifies views.
- MVC separate view and model to improve flexibility and re-usability. One model can support several views.
- MVC allows to modify an object response without to modify its view with an encapsulation of the responses into a controller.

Design patterns: the catalog

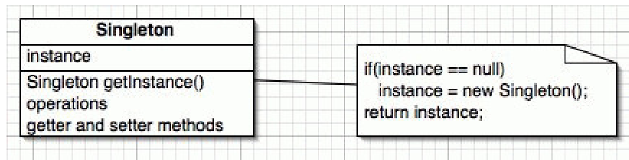
- 23 design patterns classified in 3 groups following their **role**

Creational	Structuration	Behavioral
AbstractFactory	Adapter, Bridge	Interpreter, Command
Factory	Composite,	Chain of Responsibility
Builder	Facade, Proxy	Iterator, Mediator,Template
Prototype	Flyweight	Memento, Observer
Singleton	Decorator	State, Strategy, Visitor

Creating object

- Example: **Singleton**
  - A singleton is a class which produce only one instance, and this class is accessible from everywhere.
  - For that, it is forbidden to build a new instance with the `new` operator.
  - Solution is to put the constructor `private`.
  - But now, how to get an instance of this class?

UML Schema for Singleton



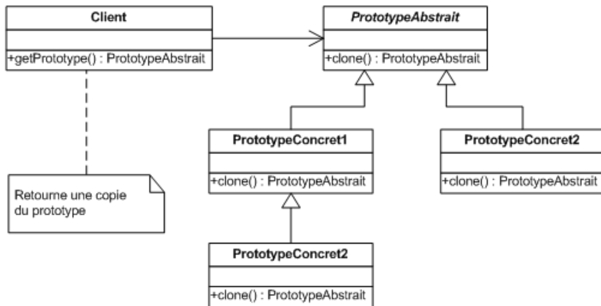
Java Code

```
  public class ClassicSingleton {
   private static ClassicSingleton instance = null;

   protected ClassicSingleton() {
      // Exists only to defeat instantiation.
   }
   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
}
```

Creating object

- Example: **Prototype**
  - Prototype pattern allows you to create exact copies without knowledge about their types.
  - It is an extension of the constructor by copy which permits to use polymorphism.
  - Using: automatic creation of classes and cloning objects.

UML Schema



Java Code

```
interface Person {
    Person clone();
}

class Tom implements Person {
    private final String NAME = "Tom";

    @Override
    public Person clone() {
        return new Tom();
    }

    @Override
    public String toString() {
        return NAME;
    }
}
```

Java Code

```
class Dick implements Person {
    private final String NAME = "Dick";

    @Override
    public Person clone() {
        return new Dick();
    }

    @Override
    public String toString() {
        return NAME;
    }
}

class Harry implements Person {
    private final String NAME = "Harry";

    @Override
    public Person clone() {
        return new Harry();
    }

    @Override
    public String toString() {
        return NAME;
    }
}
```

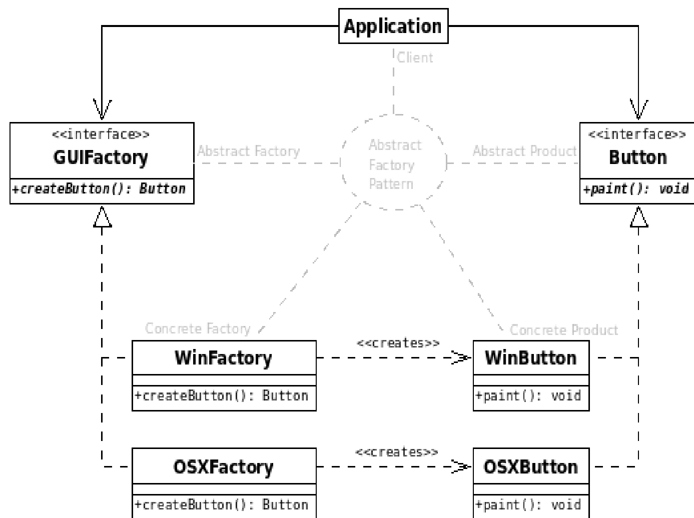
Java Code

```
class Factory {
    private static final Map<String, Person> prototypes = new HashMap<>();

    static {
        prototypes.put("tom", new Tom());
        prototypes.put("dick", new Dick());
        prototypes.put("harry", new Harry());
    }

    public static Person getPrototype(String type) {
        try {
            return prototypes.get(type).clone();
        } catch (NullPointerException ex) {
            System.out.println("Prototype with name: " + type + ", doesn't exist");
            return null;
        }
    }
}
public class PrototypeFactory {
    public static void main(String[] args) {
        if (args.length > 0) {
            for (String type : args) {
                Person prototype = Factory.getPrototype(type);
                if (prototype != null) {
                    System.out.println(prototype);
                }
            }
        } else {
            System.out.println("Run again with arguments of command string ");
        }
    }
}
```

Abstract Factory and Factory

- **Abstract factory**: creating families of linked or interacting objects without specifications of their concrete classes.
- **Factory**: allows to delegate the instantiation of objects to the sub-classes.

UML Schema



Java Code

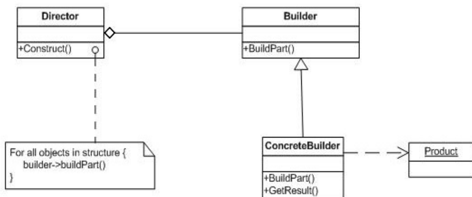
```
abstract class GUIFactory {
    public static GUIFactory getFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
    public abstract Button createButton();
}

class WinFactory extends GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}
class OSXFactory extends GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}
```

Java Code

```
abstract class Button {
    public abstract void paint();
}
class WinButton extends Button {
    public void paint() {
        System.out.println("I'm a WinButton: ");
    }
}
class OSXButton extends Button {
    public void paint() {
        System.out.println("I'm an OSXButton: ");
    }
}
public class Application {
    public static void main(String[] args) {
        GUIFactory factory = GUIFactory.getFactory();
        Button button = factory.createButton();
        button.paint();
    }
}
```

Builder

- This pattern is used when the creation of an object is complex and does not depend on the elements which compose it.



Java Code

```java
/** "Product" */
class Pizza
{
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough)
    { this.dough = dough; }
    public void setSauce(String sauce)
    { this.sauce = sauce; }
    public void setTopping(String topping)
    { this.topping = topping; }
}
/** "Abstract Builder" */
abstract class PizzaBuilder
{
    protected Pizza pizza;

    public Pizza getPizza()
    {
        return pizza;
    }
    public void createNewPizzaProduct()
    {
        pizza = new Pizza();
    }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Java Code

```
 /** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder
{
    public void buildDough()
    {
        pizza.setDough("cross");
    }
    public void buildSauce()
    {
        pizza.setSauce("mild");
    }
    public void buildTopping()
    {
        pizza.setTopping("ham+pineapple");
    }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder
{
    public void buildDough()
    {
        pizza.setDough("pan baked");
    }
    public void buildSauce()
    {
        pizza.setSauce("hot");
    }
```

Java Code

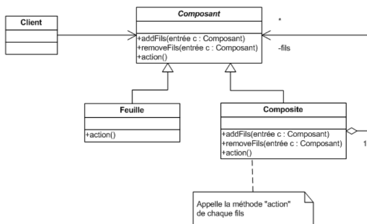
```
 /** "Director" */
class Cook
{
    private PizzaBuilder pizzaBuilder;
     public void setPizzaBuilder(PizzaBuilder pb)
    {
        pizzaBuilder = pb;
    }
    public Pizza getPizza()
    {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza()
    {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
 /** A given type of pizza being constructed. */
class BuilderExample
{
    public static void main(String[] args)
    {
        Cook cook = new Cook();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();
        cook.setPizzaBuilder( hawaiianPizzaBuilder );
        cook.constructPizza();
        Pizza pizza = cook.getPizza();
    }
}
```

Structural Design Patterns

- They are dedicated to problems for coupling objects and code structure.
- They prefer to create new way to compose objects rather than to multiply interfaces.

Composite

- **Goal** : representation of tree structures, or recursive composition.
- **Advantages** : to be able to treat in the same way one object and a collection of objects.



Java Code

```
/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();
}
/** "Composite" */
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> childGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }
    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }
}
```

Java Code

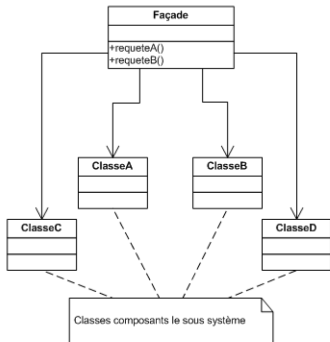
```
/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
/** Client */
public class Program {
    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);
        graphic2.add(ellipse4);
        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (Four times the string "Ellipse").
        graphic.print();
    }
}
```

Java Code

Facade

- The goal is to give the same interface to a sub-system composed of several interfaces and objects with complexes interactions.
- Typically, `facade` can response to the user who calls methods belonging to several objects of the sub-system.
- In this case he user calls `facade` methods without any attention for the work makes by the sub-system.

UML Schema

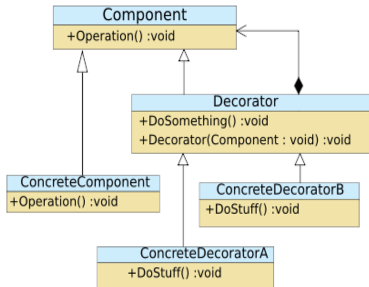


Java Code

```
/** Facade **/
class UserfriendlyDate
{
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

    public UserfriendlyDate (String isodate_ymd) throws ParseException {
        Date date = sdf.parse(isodate_ymd);
        cal.setTime(date);
    }

    public void addDays (int days) {
        cal.add (Calendar. DAY_OF_MONTH, days);
    }

    public String toString() {
        return sdf.format(cal.getTime());
    }
}
/** "Client" **/
class FacadePattern
{
    public static void main(String[] args) throws ParseException
    {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println ("Date: " + d.toString());
        d.addDays(20);
        System.out.println ("20 days after: " + d.toString());
    }
}
```

Decorator

- **Decorator** : dynamically add characteristics (or behaviors) to an object (a class). Example: window manager attributes.



Java Code

```
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the Window
}

class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }
    public String getDescription() {
        return "simple window";
    }
}

abstract class WindowDecorator implements Window {
    protected Window decoratedWindow;
    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
}
```

Java Code

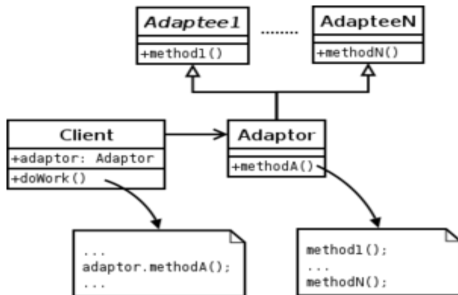
```
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawVerticalScrollBar();
        decoratedWindow.draw();
    }
    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical scrollbars";
    }
}

class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawHorizontalScrollBar();
        decoratedWindow.draw();
    }
    private void drawHorizontalScrollBar() {
        // draw the horizontal scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including horizontal scrollbars";
    }
}
```

Java Code

```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

Adapter

- **Adapter**: convert interface of a class to another one to response to the client expectations, by composition or by multiple inheritance.
- the adapter contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object



Java Code

```
public class Battery
{
    private IChargeable phone;
    private const int voltage = 10;
    public void branchCellular(IChargeable cellular)
    {
        Console.WriteLine("Cellular branch ");
        this.phone = cellular;
        this.phone.load(voltage);
    }
}

public interface IChargeable
{
    /* methode called to charg cellular
     param name="volts" battery voltage */
    void load(int volts);
}
```

Java Code

```
public class TestCellular implements IChargeable
{
    public void load(int volts)
    {
        Console.WriteLine("test Portable loading");
        Console.WriteLine("voltage: {0}", volts.ToString());
    }
}
public class SonneEricSonneCellular
{
    // only 10 volts
    public void loadCellular(int volts)
    {
        Console.WriteLine("Cellular SonneEricSonne loading");
        Console.WriteLine("voltage: {0}", volts.ToString());
    }
}
public class SamSaouleCellular
{
    // only 5 volts
    public void loadCellular(int volts)
    {
        Console.WriteLine("Cellular SamSaoule in charge");
        Console.WriteLine("voltage: {0}", volts.ToString());
    }
}
```

Java Code

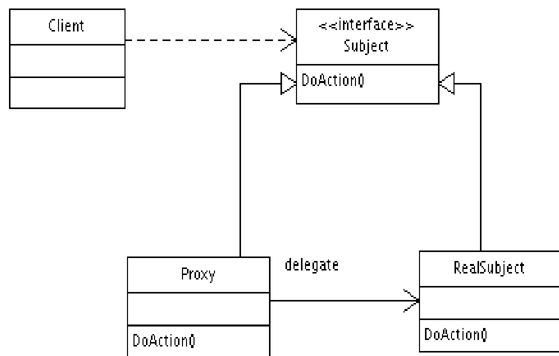
```
public class AdapterSonneEricSonne implements IChargeable
{
    private PortableSonneEricSonne phone;
    public AdapterSonneEricSonne(PortableSonneEricSonne cellular)
    {
        this.phone = cellular;
    }

    public void load(int volts)
    {
        this.phone.loadCellular(volts);
    }
}
public class AdapterSamSaoule implements IChargeable
{
    private CellularSamSaoule phone;
    public AdapterSamSaoule(CellularSamSaoule cellular)
    {
        this.phone = cellular;
    }
    public void load(int volts)
    {
        int newVoltage = volts > 5 ? 5 : volts ;

        this.phone.loadCellular(newVoltage);
    }
}
```

```
static void Main(string[] args)
{
    Battery battery = new Battery();

    /******************* Cellular SonneEricSonne***********************/

    CellularSonneEricSonne cellularSonne = new CellularSonneEricSonne();
    AdapterSonneEricSonne adapaterSonne = new AdapterSonneEricSonne(cellularSonne);
    battery.branchCellular(adapaterSonne);
    Console.WriteLine();

    /******************** Cellular SamSaoule**************************/

    CellularSamSaoule cellularSam = new CellularSamSaoule();
    AdapterSamSaoule adapaterSam = new AdapterSamSaoule(cellularSam);
    battery.branchCellular(adapaterSam);

    Console.ReadLine();
}
```

Proxy

- A `Proxy` is a class substituted to another one, it is an interface to this class.
- A specific `Proxy` is dedicated to only one class.
- Using : use of multiple copies of a complex object. Only one complex object is created and multiple instances of `Proxy` objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object.

UML Schema



Java Code

```
interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading   "+filename);
    }

    public void displayImage()
    { System.out.println("Displaying "+filename); }
}

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}
```

Java Code

```
class ProxyExample {
    public static void main(String[] args) {
        ArrayList<Image> images = new ArrayList<Image>();
        images.add( new ProxyImage("HiRes_10MB_Photo1") );
        images.add( new ProxyImage("HiRes_10MB_Photo2") );
        images.add( new ProxyImage("HiRes_10MB_Photo3") );

        images.get(0).displayImage();
        images.get(1).displayImage();
        images.get(0).displayImage();
    }
}
```

Behavioral Patterns

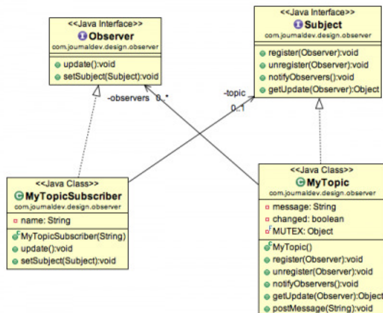
Two kinds of patterns

- Patterns oriented classes which use inheritance and polymorphism to describe algoritms and treatments flux.
- Patterns oriented objects which describe interactions in group of objects to make treatments cannot be made by only one. They use composition.

Observer

- Allows to define dependency between several objects, so when one of them (the subject) change its state the other ones (observers) can advertise immediately.
- **Using** : when an object must notify something to objects without knowing nothing about these objects.
- **Example** : data in a spreadsheet visualized with graphics, When the data change all graphics must be updated without any dependence been build between these objects.

Schema UML



Java Code

```
abstract class Observer {
    protected Subject subject;
    public abstract void update();
}
class HexObserver extends Observer {
    public HexObserver(Subject subject) {
        this.subject = subject;
        this.subject.add(this);
    }

    public void update() {
        System.out.print(" " + Integer.toHexString(subject.getState()));
    }
}
```

Java Code

```
class OctObserver extends Observer {
    public OctObserver(Subject subject) {
        this.subject = subject;
        this.subject.add( this );
    }

    public void update() {
        System.out.print(" " + Integer.toOctalString(subject.getState()));
    }
}

class BinObserver extends Observer {
    public BinObserver(Subject subject) {
        this.subject = subject;
        this.subject.add(this);
    }

    public void update() {
        System.out.print(" " + Integer.toBinaryString(subject.getState()));
    }
}
```

Java Code

```
class Subject {
    private List<Observer> observers = new ArrayList<>();
    private int state;

    public void add(Observer o) {
        observers.add(o);
    }

    public int getState() {
        return state;
    }

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

    private void execute() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

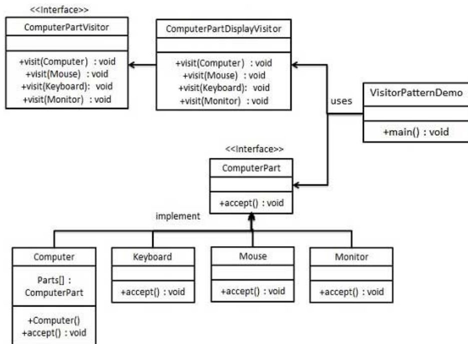
Java Code

```
public class ObserverDemo {
    public static void main( String[] args ) {
        Subject sub = new Subject();
        // Client configures the number and type of Observers
        new HexObserver(sub);
        new OctObserver(sub);
        new BinObserver(sub);
        Scanner scan = new Scanner(System.in);
        for (int i = 0; i < 5; i++) {
            System.out.print("\nEnter a number: ");
            sub.setState(scan.nextInt());
        }
    }
}
```

Visitor

- **Goal** : allows to add treatments to a class without modify it.
- **Using** :
  - ElementA and ElementB classes are subclasses of ElementAbstract, and we want to add specific functionalities to each other without lost advantage of a well-defined common interface and of the polymorphism.
  - No modification of ElementA and ElementB classes but creation of virtual method Accept(Visitor),
  - Visitor is the object that allows you to make the treatment.
  - Creation of a VisitorAbstract class with two methods visitElementA(ElementA) and visitElementB(ElementB). So it is possible to create Visitor subclasses which redefine these methods.

Schema UML



Java Code

```
public interface ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor);
}
public class Keyboard implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
public class Monitor implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}

public class Mouse implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Java Code

```
public interface ComputerPartVisitor {
   public void visit(Computer computer);
   public void visit(Mouse mouse);
   public void visit(Keyboard keyboard);
   public void visit(Monitor monitor);
}

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

   @Override
   public void visit(Computer computer) {
      System.out.println("Displaying Computer.");
   }

   @Override
   public void visit(Mouse mouse) {
      System.out.println("Displaying Mouse.");
   }

   @Override
   public void visit(Keyboard keyboard) {
      System.out.println("Displaying Keyboard.");
   }

   @Override
   public void visit(Monitor monitor) {
      System.out.println("Displaying Monitor.");
   }
}
```

Java Code

```
public class Computer implements ComputerPart {

   ComputerPart[] parts;

   public Computer(){
      parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
   }


   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      for (int i = 0; i < parts.length; i++) {
         parts[i].accept(computerPartVisitor);
      }
      computerPartVisitor.visit(this);
   }
}

public class VisitorPatternDemo {
   public static void main(String[] args) {

      ComputerPart computer = new Computer();
      computer.accept(new ComputerPartDisplayVisitor());
   }
}
```

Strategy and Bridge

- `Strategy` is used for dynamic swap between algorithms at runtime. With `Strategy` it is possible to define family of functions. This pattern is invisible in language with first-class functions like Python.
- `Bridge` separate the interface from the implementation. So implementation can vary independently. The two hierarchy can be updated separately.

Strategie - Java Code

```java
public interface PaymentStrategy {

   public void pay(int amount);
}

public class CreditCardStrategy implements PaymentStrategy {

   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;

   public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){
      this.name=nm;
      this.cardNumber=ccNum;
      this.cvv=cvv;
      this.dateOfExpiry=expiryDate;
   }
   @Override
   public void pay(int amount) {
      System.out.println(amount +" paid with credit/debit card");
   }

}
```

Java Code

```
public class PaypalStrategy implements PaymentStrategy {
   private String emailId;
   private String password;
   public PaypalStrategy(String email, String pwd){
      this.emailId=email;
      this.password=pwd;
   }
   public void pay(int amount) {
      System.out.println(amount + " paid using Paypal.");
   }
}

public class Item {
   private String upcCode;
   private int price;
   public Item(String upc, int cost){
      this.upcCode=upc;
      this.price=cost;
   }

   public String getUpcCode() {
      return upcCode;
   }

   public int getPrice() {
      return price;
   }
}
```

Java Code

```
public class ShoppingCart {
   List<Item> items;

   public ShoppingCart(){
      this.items=new ArrayList<Item>();
   }

   public void addItem(Item item){
      this.items.add(item);
   }

   public void removeItem(Item item){
      this.items.remove(item);
   }

   public int calculateTotal(){
      int sum = 0;
      for(Item item : items){
         sum += item.getPrice();
      }
      return sum;
   }

   public void pay(PaymentStrategy paymentMethod){
      int amount = calculateTotal();
      paymentMethod.pay(amount);
   }
}
```

Java Code

```
public class ShoppingCartTest {

   public static void main(String[] args) {
      ShoppingCart cart = new ShoppingCart();

      Item item1 = new Item("1234",10);
      Item item2 = new Item("5678",40);

      cart.addItem(item1);
      cart.addItem(item2);

      //pay by paypal
      cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

      //pay by credit card
      cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/15"));
   }
}
```

Bridge - Java Code

```
class Node {
    public int  value;
    public Node prev, next;
    public Node( int i ) { value = i; }
}


class StackArray {
    private int[] items = new int[12];
    private int   total = -1;
    public void push( int i ) { if ( ! isFull()) items[++total] = i; }
    public boolean isEmpty()  { return total == -1; }
    public boolean isFull()   { return total == 11; }
    public int top() {
        if (isEmpty()) return -1;
        return items[total];
    }
    public int pop() {
        if (isEmpty()) return -1;
        return items[total--];
}   }
```

Bridge - Java Code

```
class StackList {
    private Node last;
    public void push( int i ) {
        if (last == null)
            last = new Node( i );
        else {
            last.next = new Node( i );
            last.next.prev = last;
            last = last.next;
    }   }
    public boolean isEmpty() { return last == null; }
    public boolean isFull()  { return false; }
    public int top() {
        if (isEmpty()) return -1;
        return last.value;
    }
    public int pop() {
        if (isEmpty()) return -1;
        int ret = last.value;
        last = last.prev;
        return ret;
}   }
```

Java Code

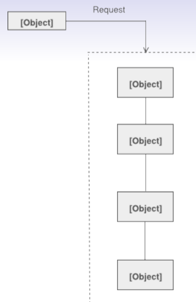
```
class StackFIFO extends StackArray {
    private StackArray temp = new StackArray();
    public int pop() {
        while ( ! isEmpty())
            temp.push( super.pop() );
        int ret =  temp.pop();
        while ( ! temp.isEmpty())
            push( temp.pop() );
        return ret;
}  }

class StackHanoi extends StackArray {
    private int totalRejected = 0;
    public int reportRejected()   { return totalRejected; }
    public void push( int in ) {
        if ( ! isEmpty()  &&  in > top())
            totalRejected++;
        else super.push( in );
}   }
```

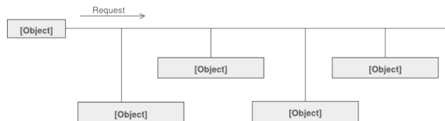
Java Code

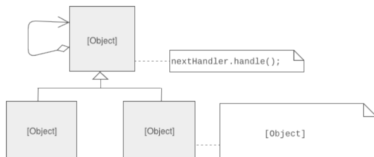
```
class BridgeDisc {
    public static void main( String[] args ) {
        StackArray[] stacks = { new StackArray(), new StackFIFO(), new StackHanoi() };
        StackList    stack2 = new StackList();
        for (int i=1, num; i < 15; i++) {
            stacks[0].push( i );
            stack2.push( i );
            stacks[1].push( i );
        }
        java.util.Random rn = new java.util.Random();
        for (int i=1, num; i < 15; i++)
            stacks[2].push( rn.nextInt(20) );
        while ( ! stacks[0].isEmpty())
            System.out.print( stacks[0].pop() + "  " );
        System.out.println();
        while ( ! stack2.isEmpty())
            System.out.print( stack2.pop() + "  " );
        System.out.println();
        while ( ! stacks[1].isEmpty())
            System.out.print( stacks[1].pop() + "  " );
        System.out.println();
        while ( ! stacks[2].isEmpty())
            System.out.print( stacks[2].pop() + "  " );
        System.out.println();
        System.out.println( "total rejected is "
            + ((StackHanoi)stacks[2]).reportRejected() );
}   }
```

Chain of Responsibility

- Build a chain of processing unit, each unit handle the request if threshold is satisfied.
- Since a chain is built, if one unit is not satisfied, it's next unit will be tested, and so on.
- Give to more than one object a chance to handle the request.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- Architecture : a linked list with recursive traversal.







Go to the example code !

AntiPatterns

- The massive using of Design patterns leads to new **misconception** phenomena.
- A book to explain that:
  AntiPatterns by William Brown, Raphael Malveau, Skip McCormick, Tom Mowbray and Scott Thomas.
- Definition: is a pattern that tells you how to go from problem to a bad solution.
- Jim Coplien: *"an anti-pattern is something that looks like a good idea, but which backfires badly when applied."*
- An AntiPattern can also explain to you why a bad solution looks like a good idea.

AntiPatterns

- Some examples:
  - Inverse abstraction - if an interface offers only complex behaviors whereas only simple behavior are required.
  - Reinventing the square wheel: creating a poor solution when a good one exists.
  -
- If a software application is garnished with the unfortunate combination or mix of too many anti-patterns, then it may be known under the expression **"full monty"**.