Objec Oriented Design

Master 1 of Informatics I.E.I. - V.N.U.

2020-2021

Marie Beurton-Aimar

Université de Bordeaux

Reminders

- Create objects

```
class Person{
      private String name;
      private int age;
      public Person(String chain, int val){
             name = chain;
             age = val;
      }
  }
  class Game {
    public static void main(String argc[]){
            Person item=new Person("Linh", 20);
            Person item2= new Person("Tung", 25);
    }
  }
```

- Inheritance

```
class Player extends Person{
  private int [] scores;
  public Player(String name, int age){
      super(name,age);
      scores=new int[10];
  }
class Game {
  public static void main(String argc[]){
      Player item=new Player("Linh", 20);
      Player item2= new Player("Tung", 25);
   }
}
```

Reminders

Abstract class

- Calling `new` for a Person is forbidden.
- No instance of Person can be created.

```
abstract class Person{
    private String name;
    private int age;
    public Person(String chain, int val){
           name = chain;
           age = val;
    }
}
```

Reminders - Abstract class

```
class Game {
public static void main(String argc[]){
   Person newPerson; // declaration is authorized

   Player item=new Player("Linh", 20);
   Player item2= new Player("Tung", 25);
   newPerson=item;
```

Input/Output

Role

- To be useful a program has to communicate with user/another program.
- Data are managed through flux/stream.
- A lot of classes define different stream types.
- Most of the stream are sequential.

Input/Output

Description

- Keyboard and screen are standard input/output stream.
- Variables `in` and `out` belong to `InputStream` type and `PrintStream` type (which inherits of `OutputStream`.
- Standard error is coded by `err` variable which is belong to `PrintStream` type.

Stream Naming

- Stream direction : `Reader` et `Writer`
- Source or destination types :

Input/Output

Example of Class combination

- A way to collect input string from the keyboard

```
   public static String getString ()
    {
      try {
        BufferedReader buff = new BufferedReader
            (new InputStreamReader(System.in));
        String chain=buff.readLine();
        return chain;
      }
      catch(IOException e) {
System.out.println("no way to work" +e);
return null;
      }
    }
```

Input/Output

Example of Class combination

- A way to read string in a text file.

```
public static void read (Vector myTeam)throws IOException
{
 BufferedReader buff=new BufferedReader(new FileReader("fichier.txt"));
 try {
   Person item=null ;
   for(;;){ String name = buff.readLine();
      item = (Person) new Person(name);
      int num = Integer.valueOf(buff.readLine()).intValue();
      item.setAge(num);

      myTeam.addElement(courant);
     }
  }
 catch (InstantiationException e){
   System.out.println("End");
   buff.close();
  }
}
```

Example of Class combination

- A way to write in a text file.

```
public static void write (Vector myTeam)
       throws IOException
{
  BufferedWriter buff=new BufferedWriter
     (new FileWriter("file.txt"));
  for(Enumeration e = myTeam.elements();e.hasMoreElements();)
    {
     Person item = (Person)e.nextElement();
     item.save(buff);
    }
   buff.flush();
   buff.close();
}
```

Example of Class combination

- A way to write in a text file.

```
void save(BufferedWriter buff)throws IOException
 {
    buff.write(name);
    buff.newLine();
    buff.write((new Integer(age)).toString());
    buff.newLine();

 }
```

Interface

Role

- Define behavior to share between classes which have no inheritance link.
- Interface define a type.

Usage

- Classes which want to use interface declare the keyword `implements`. No limit of interface number to implement.
- Provide a solution to multiple inheritance ban in java.
- `Public` by default and `abstract`.

Interface

Usage

- `default` method can be defined - no necessity to implement it in the concrete class (only from Java 8).
- Functional interface - equivalent to `lambda` function (only from Java 8).
- Can be used to define `lambda`
- `Serializable` interface - use to tag classes that can be serializabel i.e. can be written in a text file or sent through a network, no method to implement.

Interface Example

Iterator

- A **Design Pattern** object.
- A tool for `collection` object.
- See `Vector` for a simple container :

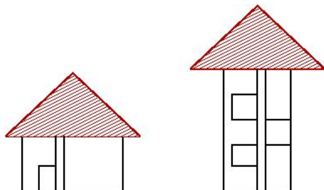
```
Vector <Person> myTeam = new Vector<Person>();
myTeam.addElement(item);
myTeam.addElement(item2);
for (Iterator it = myTeam.iterator(); it.hasNext();)
  Person myPerson = it.next();
  myPerson.display();
  }
```

A Small Break to present 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"



- Define pattern 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.

A part - 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 Factory Builder Prototype Singleton	Adapter, Bridge Composite, Facade, Proxy Flyweight Decorator	Interpreter, Command Chain of Responsibility Iterator, Mediator,Template Memento, Observer State, Strategy, Visitor

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

- Some of them exist as class/interface in java language.

Object Oriented Programming

Implement the concepts

- Design patterns allow to implement concepts as : modularity, abstraction, delegation, encapsulation.

Delegation

- Delegate to another object a piece of the task to assume.
- Go back to `Iterator`

```
Vector <Person> myTeam = new Vector<Person>();
myTeam.addElement(item);
myTeam.addElement(item2);
for (Iterator it = myTeam.iterator(); it.hasNext();)
 {
  Person myPerson = it.next();
  myPerson.display();
  }
```

Delegation

Genericiy

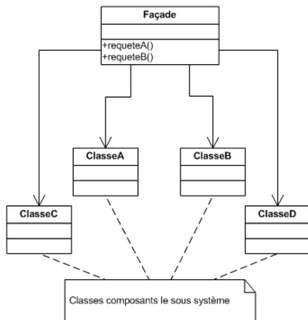
- Possibility to change the container but keeping the same algorithm by calling `iterator`.
- Just need to have the way to extract data in a right format.

```
ArrayList <Person> myTeam = new ArrayList<Person>();
  myTeam.add(item);
  myTeam.add(item2);
  for (Iterator it = myTeam.iterator(); it.hasNext();)
    {
    Person myPerson = it.next();
    myPerson.display();
    }
```

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

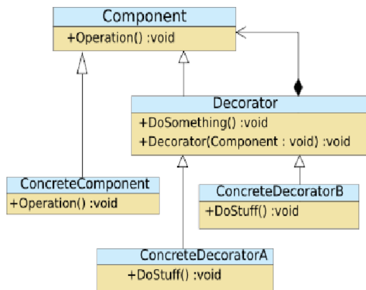


Java Code

```
/** 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());
    }
}
```

Creational Patterns

Role

- Provide creation mechanisms to increase flexibility and reusing code.
- A very simple 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?
- **Factory** and **Builder** belong to this group.

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;
   }
}
```

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.

Java Code

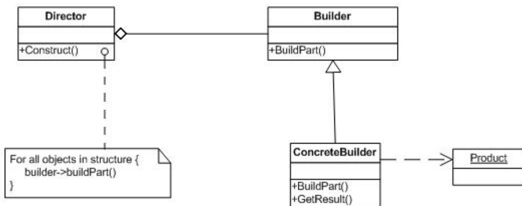
```
abstract class GUIFactory {
    public static GUIFactory getFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
    public abstract Button createButton();
}

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

```
/** "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

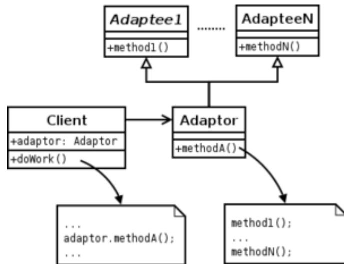
```
 /** "Director" */
class Cook
{
    private PizzaBuilder pizzaBuilder;
     public void setPizzaBuilder(PizzaBuilder pb)
    {
        pizzaBuilder = pb;
    }
    public Pizza getPizza()
    {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza()
    {
        pizzaBuilder.createNewPizzaProduct();
        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.
- **Facade** and **Decorator** belongs to this group.

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.

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();
    }
}
```

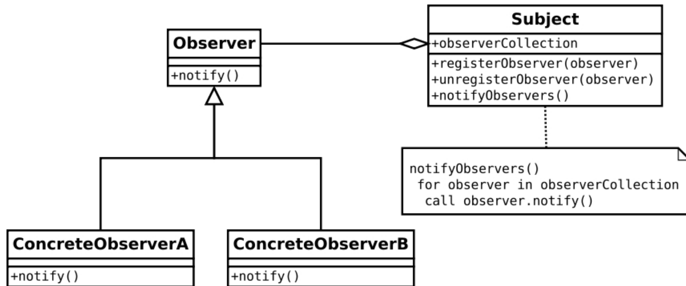
Java Code

```
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();
    }
}
```

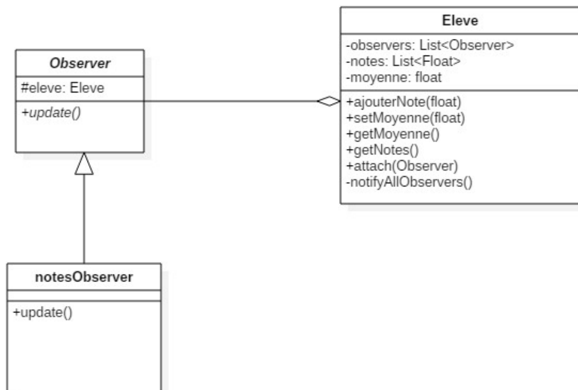
Observer

- 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



Schema UML



Observer

```
 import java.util.List;
 Import java.util.ArrayList;
 public class Student {

 private List<Observer> observers;
 private List<Float> marks;
 private float average;
 public Student() {
     observers = new ArrayList<Observer>();
     marks = new ArrayList<Float>();
 }
 public void ajouterMark(float mark) {
   marks.add(mark);
   notifyAllObservers();
 }
public void setAverage(float average) {
   this.average = average;
}
public float getAverage() {
   return average;
}
public List<Float> getMarks() {
   return marks;
}
public void attach(Observer observer){
   observers.add(observer);
}
private void notifyAllObservers() {
   for (Observer observer : observers) {
      observer.update();
   }
}
}
```

Observer

```
public abstract class Observer {
   protected Student student;
   public abstract void update();
}
public class marksObserver extends Observer{

public marksObserver(Student student){
   this.student = student;
   this.student.attach(this);
}
public void update() {

   float average = 0;

   for(float mark : student.getMarks()) {
      average += mark;
   }

   average /= student.getMarks().size();

   student.setAverage(average);
  }

}
```

Observer Test

```
public class Main {

 public static void main(String[] args) {

   Student student = new Student();

    new marksObserver(student);

    student.ajouterMark(15.0f);
    System.out.println(student.getAverage());
    student.ajouterMark(5.0f);
    System.out.println(student.getAverage());
    student.ajouterMark(13.0f);
    System.out.println(student.getAverage());

  }

}
```

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.

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

- **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.

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.

Visitor

```
interface CarElement {
  void accept(CarElementVisitor visitor);
  // Méthode à définir par les classes implémentant CarElements
}


class Wheel implements CarElement {
  private String name;

  Wheel(String name) {
      this.name = name;
  }

  String getName() {
      return this.name;
  }

  public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
  }
}
```

Visitor - Implementation

```
class Engine implements CarElement {
  public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
  }
}

class Body implements CarElement {
  public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
  }
}
class Car {
  CarElement[] elements;

  public CarElement[] getElements() {
      return elements.clone(); // Retourne une copie du tableau de références
  }

  public Car() {
      this.elements = new CarElement[] {
              new Wheel("front left"),
              new Wheel("front right"),
              new Wheel("back left"),
              new Wheel("back right"),
              new Body(),
              new Engine()
          };
  }
}
```

Visitor for the Car

```
  interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visitCar(Car car);
}

class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting "+ wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visitCar(Car car) {
        System.out.println("\nVisiting car");
        for(CarElement element : car.getElements()) {
            element.accept(this);
        }
        System.out.println("Visited car");
    }
}
```

Visitor for the Car

```
class CarElementDoVisitor implements CarElementVisitor {
   public void visit(Wheel wheel) {
       System.out.println("Kicking my "+ wheel.getName());
   }

   public void visit(Engine engine) {
       System.out.println("Starting my engine");
   }

   public void visit(Body body) {
       System.out.println("Moving my body");
   }

   public void visitCar(Car car) {
       System.out.println("\nStarting my car");
       for(CarElement carElement : car.getElements()) {
           carElement.accept(this);
       }
       System.out.println("Started car");
   }
}
```

Visitor test

```
public class TestVisitorDemo {
  static public void main(String[] args) {
      Car car = new Car();

      CarElementVisitor printVisitor = new CarElementPrintVisitor();
      CarElementVisitor doVisitor = new CarElementDoVisitor();

      printVisitor.visitCar(car);
      doVisitor.visitCar(car);
  }
}
```

Prototype

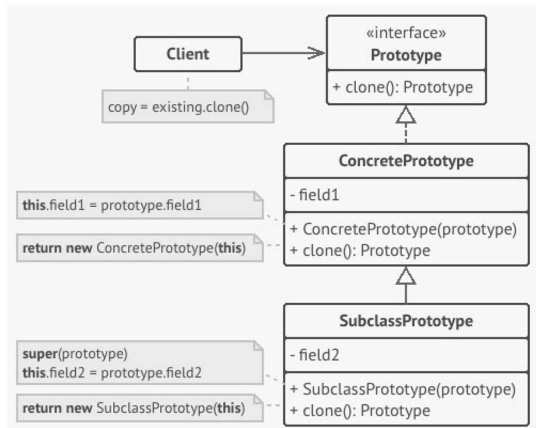
Goal

- Create new objects from existing objects without creating dependancies between code and classes.

Solution

- Delegation to the copied object the task to copy itself.
- Declare an interface for all objects to copy.
- This interface mostoften contains only a method `clone`

Schema UML



Prototype

```
public abstract class Tree {
 int height;
 int mass;
 Position pos;

 public abstract Tree copy();

}

public class PlasticTree extends Tree {

 @Override
 public Tree copy() {
     PlasticTree plasticTreeClone = new PlasticTree(this.getMass(), this.getHeight());
     plasticTreeClone.setPosition(this.getPosition());
     return plasticTreeClone;
 }

}

public class PineTree extends Tree {

 @Override
 public Tree copy() {
     PineTree pineTreeClone = new PineTree(this.getMass(), this.getHeight());
     pineTreeClone.setPosition(this.getPosition());
     return pineTreeClone;
 }
}
```

Prototype

```
public class TreePrototypesUnitTest {

  @Test
  public void givenAPlasticTreePrototypeWhenClonedThenCreateA_Clone() {


      PlasticTree plasticTree = new PlasticTree(mass, height);
      plasticTree.setPosition(position);
      PlasticTree anotherPlasticTree = (PlasticTree) plasticTree.copy();
      anotherPlasticTree.setPosition(otherPosition);

      assertEquals(position, plasticTree.getPosition());
      assertEquals(otherPosition, anotherPlasticTree.getPosition());
  }
}
@Test
public void givenA_ListOfTreesWhenClonedThenCreateListOfClones() {

  // create instances of PlasticTree and PineTree

  List<Tree> trees = Arrays.asList(plasticTree, pineTree);
  List<Tree> treeClones = trees.stream().map(Tree::copy).collect(toList());



  assertEquals(height, plasticTreeClone.getHeight());
  assertEquals(position, plasticTreeClone.getPosition());
}
```

Modules and Packages

Modularity in Java

- `package` keyword given at the first line of a file, attaches the class to a package.
  - For example : `package ao; class Person{ // some description of the class }`
- The class name is built by concatenation of the package name and the class name.
  - To use the class `Person` you now need to `import ao;` if the client does not belong to the `ao` package.
- Package/module can be a directory (folder) -
  - It is the case in java library.
- If no package is given, the current directory is the default package. It is why you have been able to ignore `package` instruction until now.

Modules and Packages

Java packages

- The package `java.lang` is accessible by default.
- Packages with a prefix `java` and `javax` are always deliver with the jdk repository.
- Using another suppose to `import` the corresponding package at the beginning of the file class.
- Depending on your coding environnement you could have or not some packages : `javafx, jdk.*...`
- The rule to create your own packages organisation is :
  - Build a composite name from the largest purpose to the inner one : `vn.iei.m1.s8.myapplication`
- The `PATH` and `CLASSPATH` have to be set to be able to join the right directory.

Modules and Packages

Example

- The `Main` class contains the instruction : `package vn.iei.m1.s8.myapplication;`
- You are in the directory src
- You have created a directory `classes` inside
- Instructions to compile and to run classes :

shell: `javac ./vn/iei/m1/s8/myapplication/*.java -d classes`
shell: `java -cp ./classes vn.iei.m1.s8.myapplication.Main`