Patrons de conception

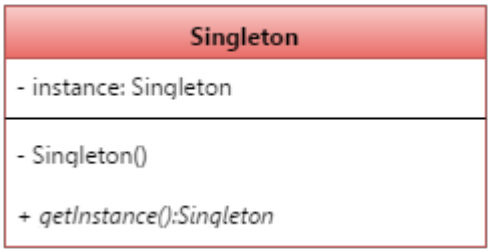
Les patrons de conception (design patterns) apportent des solutions algorithmiques et d'implémentation aux problèmes courants rencontrés dans le cadre de la conception orientée objet.

Le Singleton (Singleton)

Problématique : Créer et avoir toujours à disposition une unique instance d'une classe

Le pattern Singleton apporte une solution aux cas où une unique instance d'une classe doit exister dans un programme (pour une gestion d'un pool de connexions, une gestion du cache, une instance d'application...)

Diagramme de classes



Implémentation en java

```
public final class Singleton {

  private static Singleton instance;

  private Singleton() {
    // constructeur
  }

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

  @Override
  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

Utilisation

Il est ensuite possible de faire appel à la méthode getInstance qui retournera toujours l'instance unique de singleton.

```
Singleton.getInstance();
```

La Fabrique (Factory)

Problématique : Créer différents objets dont le type peut varier à l'exécution, en fonction du déroulement du programme

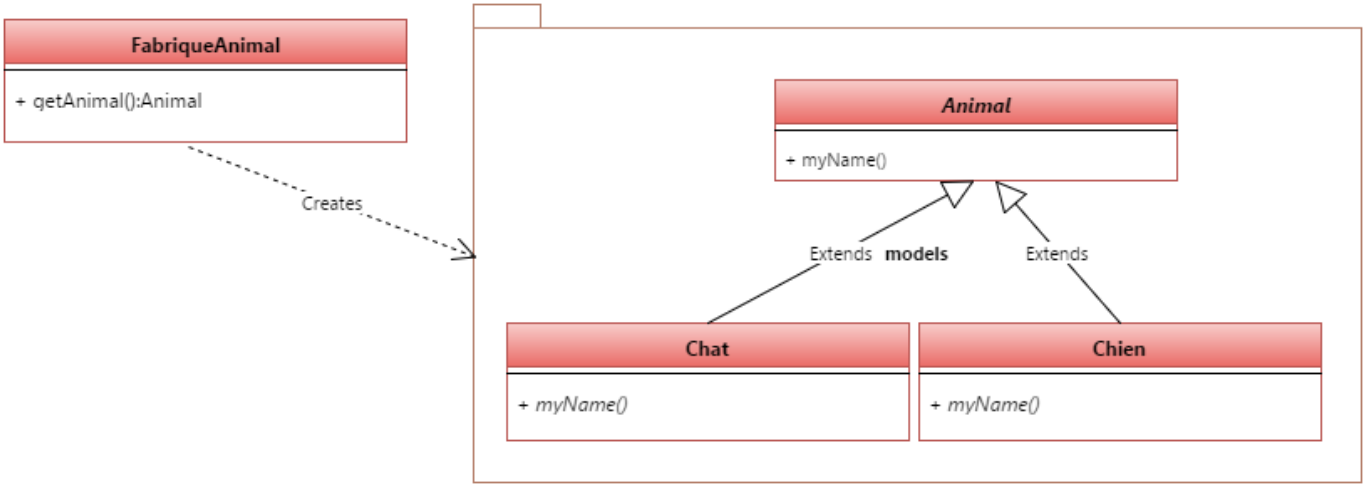
Objectifs

La fabrique est une classe de création d'objets, de différents types, héritant d'une classe de base, ou implémentant une interface. Elle permet la création d'objets dynamiquement en fonction de paramètres passés.

Caractéristiques

- La fabrique étant souvent unique dans le programme, elle peut utiliser le pattern Singleton
- Les paramètres passés à la fabrique déterminent le type d'instance à créer

Diagramme de classes



Implémentation en java

Les différents types à créer (héritant d'un type générique)

```
public abstract class Animal{
   public abstract void myName();
}
```

```
public class Chat extends Animal{
   public void myName(){
     System.out.println("Je suis un Chat");
   }
}

public class Chien extends Animal{
   public void myName(){
     System.out.println("Je suis un Chien");
   }
}
```

La fabrique :

```
public class FabriqueAnimal{

    private static FabriqueAnimal instance = new FabriqueAnimal();
    private FabriqueAnimal(){}

    public static FabriqueAnimal getFabriqueAnimalInstance(){
        return instance ;
    }

    public Animal getAnimal(String typeAnimal) throws ExceptionCreation{
        Animal result=null;
        switch(typeAnimal){
            case "chat":
                result= new Chat();
                break;
            case "chien":
                result=new Chien();
                break;
            default:
                throw new ExceptionCreation("Impossible de créer un " +
typeAnimal);
                break;
        }
        return result;
    }
}
```

Utilisation :

```
public class DemoFabrique{
   public static void main(String [] args){
     FabriqueAnimal fabrique =  FabriqueAnimal.getFabriqueAnimalInstance();
     Animal animal = FabriqueAnimal.getAnimal("chat");
     animal.myName();
   }
}
```

La Fabrique abstraite (Abstract Factory)

Problématique : Créer différents objets dont le type peut varier à l'exécution, en fonction du déroulement du programme, dissocier création et utilisation pour permettre l'ajout de nouveaux objets, sans modifier le code du programme utilisateur.

Objectifs

La fabrique est un créateur d'objets, de différents types, héritant d'une classe de base, ou implémentant une interface. Elle permet d'isoler la création des objets de leur utilisation.
On peut ainsi ajouter de nouveaux objets dérivés sans modifier le code qui utilise l'objet de base.

Implémentation en java

Les différents types d'objet à créer et leur classe de base :

```
public abstract class Button{
    private String caption;
    public abstract void paint();
    public String getCaption(){
        return caption;
    }

    public void setCaption(String caption){
        this.caption = caption;
    }
}

class WinButton extends Button{
    public void paint(){
        System.out.println("I'm a WinButton: "+ getCaption());
    }
}

class OSXButton extends Button{
    public void paint(){
        System.out.println("I'm a OSXButton: "+ getCaption());
    }
}
```

Les factories concrètes et leur classe abstraite :

```
/*
 * GUIFactory example
 */
public 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());
    }
}
```

Utilisation :

```
public class DemoAbstractFactory{
    public static void main(String[] args){
        GUIFactory aFactory = GUIFactory.getFactory();
        Button aButton = aFactory.createButton();
        aButton.setCaption("Jouer");
        aButton.paint();
    }
}
```

Limites

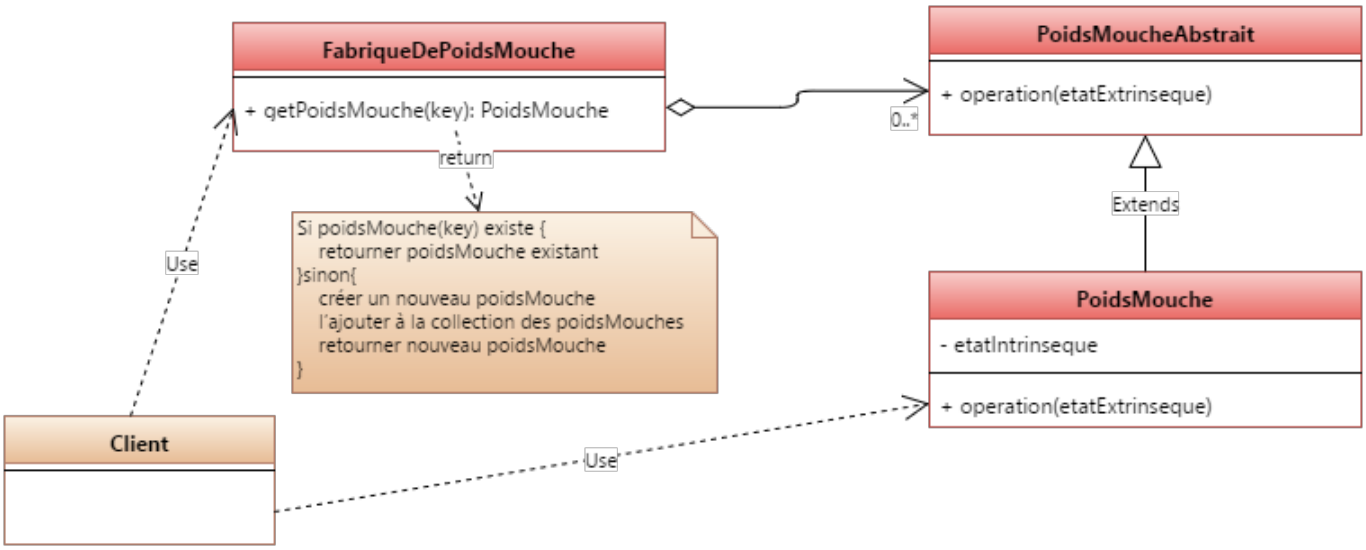
- Introduit une certaine complexité dans le développement, parfois à éviter

Le poids mouche (Flyweight)

Problématique : Eviter de créer un trop grand nombre d'instances d'une classe ayant des propriétés intrinsèques

Le poids mouche permet de factoriser le nombre d'instances à créer d'une classe, en permettant la réutilisation d'instance existantes.

Diagramme de classes



Implémentation en java

Source : http://www.journaldev.com/1562/flyweight-pattern-in-java-example-tutorial

Création de formes (Shape), de types concrets (Line, oval...)

Models

```
package models;

import java.awt.Color;
import java.awt.Graphics;

public interface Shape {
    public void draw(Graphics g, int x, int y, int width, int height, Color color);
}
```

```
package models;

import java.awt.Color;
import java.awt.Graphics;

public class Line implements Shape {
    @Override
    public void draw(Graphics g, int x, int y, int width, int height, Color color)
{
        g.setColor(color);
        g.drawLine(x, y, width, height);
    }
}
```

```
package models;

import java.awt.Color;
import java.awt.Graphics;

public class Oval implements Shape {

    private boolean fill;

    public Oval(boolean fill) {
        this.fill = fill;
    }

    @Override
    public void draw(Graphics g, int x, int y, int width, int height, Color color)
{
        g.setColor(color);
        g.drawOval(x, y, width, height);
        if (fill) {
            g.fillOval(x, y, width, height);
        }
    }
}
```

Factory

```
package models;

import java.util.HashMap;

public class ShapeFactory {

    private static final HashMap<ShapeType, Shape> shapes = new HashMap<ShapeType,
Shape>();

    public static Shape getShape(ShapeType type) {
        Shape shapeImpl = shapes.get(type);

        if (shapeImpl == null) {
            switch (type) {
            case OVAL_FILL:
                shapeImpl = new Oval(true);
                break;
            case OVAL:
                shapeImpl = new Oval(false);
                break;
            case LINE:
                shapeImpl = new Line();
                break;
            }
            shapes.put(type, shapeImpl);
        }
        return shapeImpl;
```

```
    }

    public static enum ShapeType {
        OVAL_FILL, OVAL, LINE;
    }
}
```

Utilisation

```
public class DemoFlyweight extends JFrame {

    private static final long serialVersionUID = -1350200437285282550L;
    private final int WIDTH;
    private final int HEIGHT;

    private static final ShapeType shapes[] = { ShapeType.LINE,
ShapeType.OVAL_FILL, ShapeType.OVAL };
    private static final Color colors[] = { Color.RED, Color.GREEN, Color.YELLOW };

    public DemoFlyweight(int width, int height) {
        this.WIDTH = width;
        this.HEIGHT = height;
        Container contentPane = getContentPane();

        JButton startButton = new JButton("Draw");
        final JPanel panel = new JPanel();

        contentPane.add(panel, BorderLayout.CENTER);
        contentPane.add(startButton, BorderLayout.SOUTH);
        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                Graphics g = panel.getGraphics();
                for (int i = 0; i < 40; ++i) {
                    Shape shape = ShapeFactory.getShape(getRandomShape());
                    shape.draw(g, getRandomX(), getRandomY(), getRandomWidth(),
getRandomHeight(), getRandomColor());
                }
            }
        });
    }

    private ShapeType getRandomShape() {
        return shapes[(int) (Math.random() * shapes.length)];
    }

    private int getRandomX() {
        return (int) (Math.random() * WIDTH);
    }
```

```
    private int getRandomY() {
        return (int) (Math.random() * HEIGHT);
    }

    private int getRandomWidth() {
        return (int) (Math.random() * (WIDTH / 10));
    }

    private int getRandomHeight() {
        return (int) (Math.random() * (HEIGHT / 10));
    }

    private Color getRandomColor() {
        return colors[(int) (Math.random() * colors.length)];
    }

    public static void main(String[] args) {
        new DemoFlyweight(500, 600);
    }
}
```

MVC 2

MVC2 est un patron de conception dont la volonté est de séparer les données, les traitements et la présentation. MVC2 permet de segmenter une application en trois composants essentiels :

- Le modèle
- Le contrôleur
- Les vues

MVC2 est une variante de MVC, dans laquelle le contrôleur est unique (contrairement à MVC où le rôle du contrôleur est assuré par plusieurs éléments).

Nombre de frameworks permettent une implémentation de MVC ou MVC2 facilitée.

Le Modèle

Le modèle correspond aux classes et objets métiers. Le modèle est responsable de la gestion de ces données, et en assure l'intégrité. Il ne doit faire aucune référence aux vues, pas plus qu'au contrôleur.

Le contrôleur

Le contrôleur synchronise (via évènements) les vues et le modèle. Il est responsable de la logique applicative. Le contrôleur n'effectue aucun traitement, mais se contente de solliciter le modèle adéquat, ou d'afficher la vue correspondante.

Les vues

Les vues correspondent à l'IHM, elles affichent les résultats fournis par le modèle, en réponse aux actions de l'utilisateur.

Voir MVC expliqué à mam

Injection de dépendance (dependency injection)

Problématique : Eviter une dépendance directe entre deux classes, en définissant dynamiquement la dépendance plutôt que statiquement.

Une classe A dépend d'une autre classe B quand la classe A possède un attribut de type B, ou possède une méthode utilisant la classe B (type de paramètre, valeur de retour, variable locale, appel de méthode de la classe B).

Pour mettre en œuvre l'injection de dépendance :

Créer une interface I déclarant les méthodes de la classe B utilisées par la classe A ;

```
public interface I{
    public void doSomething();
}
```

Déclarer la classe B comme implémentation de cette interface I ;

```
public class B implements I{
    public void doSomething(){
        system.out.println("something done !");
}
```

Remplacer toute référence à la classe B par des références à l'interface I ; Si la classe A instancie des instances de B à son initialisation, alors remplacer l'instanciation par un passage d'une instance de l'interface I au(x) constructeur(s) de A ; Si besoin, ajouter une méthode pour spécifier l'instance de l'interface I à utiliser.

```
public class A{
    private I injected;
    public A(I injected){
        this.injected=injected;
    }
}
```

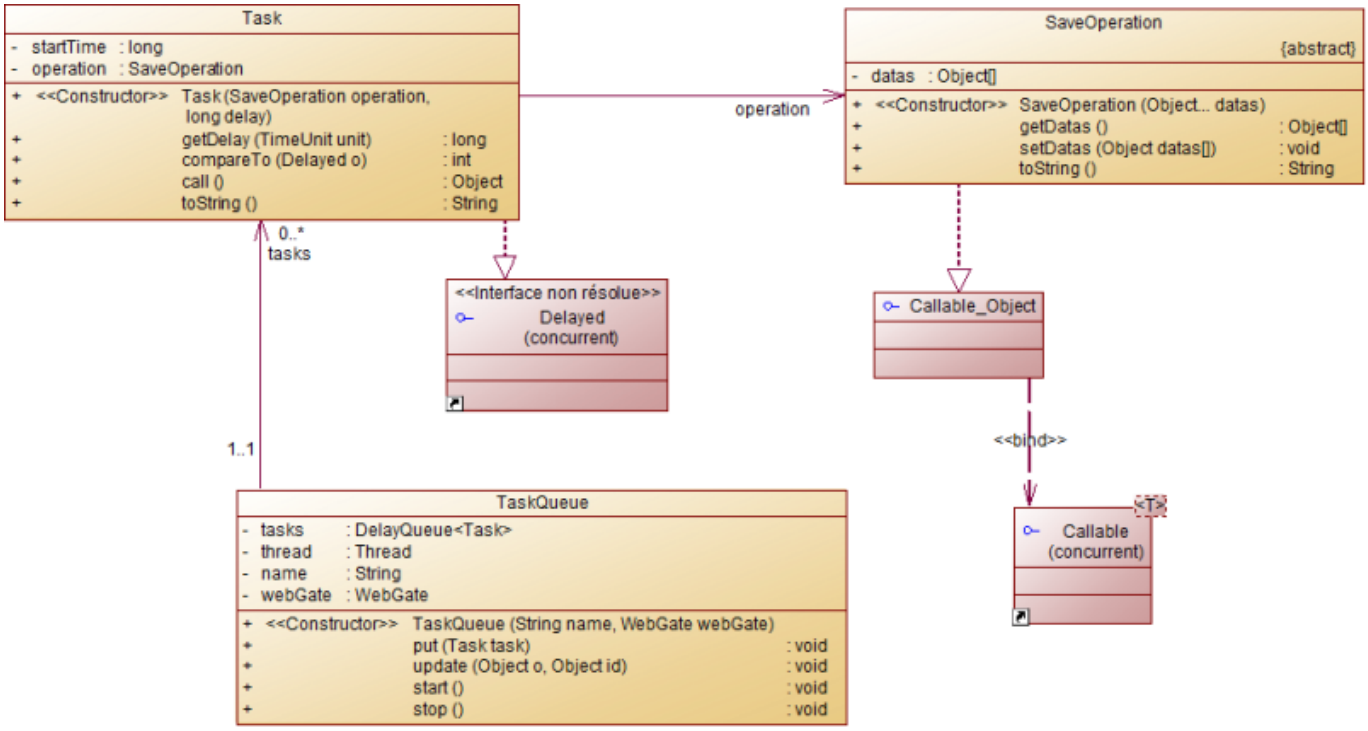
Observer/observable

Le design pattern Obserser en java

Delayed Queue

Permet de gérer une liste d'éléments en sortie FIFO avec délai.

- java DelayQueue API
- Code geeks example



Références

- Design patterns (TutorialsPoint)
- Patrons de conception (wikiBooks)