Patrons de création

Un patron de création permet de résoudre les problèmes liés à la création et la configuration d'objets.

Par exemple, une classe nommée `RessourcesApplication` gérant toutes les ressources de l'application ne doit être instanciée qu'une seule et unique fois. Il faut donc empêcher la création intentionnelle ou accidentelle d'une autre instance de la classe. Ce type de problème est résolu par le patron de conception "Singleton".

Les différents patrons de création sont les suivants :

Singleton
: Il est utilisé quand une classe ne peut être instanciée qu'une seule fois.

Prototype
: Plutôt que de créer un objet de A à Z c'est à dire en appelant un constructeur, puis en configurant la valeur de ses attributs, ce patron permet de créer un nouvel objet par recopie d'un objet existant.

Fabrique
: Ce patron permet la création d'un objet dont la classe dépend des paramètres de construction (un nom de classe par exemple).

Fabrique abstraite
: Ce patron permet de gérer différentes fabriques concrètes à travers l'interface d'une fabrique abstraite.

Monteur
: Ce patron permet la construction d'objets complexes en construisant chacune de ses parties sans dépendre de la représentation concrète de celles-ci.

Singleton

Le **singleton** est un patron de conception dont l'objet est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement). Il est utilisé lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.

On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà. Dans beaucoup de langages de type objet, il faudra veiller à ce que le constructeur de la classe soit *privé* ou bien *protégé*, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.

Le singleton doit être implémenté avec précaution dans les applications multi-thread. Si deux processus légers exécutent *en même temps* la méthode de création alors que l'objet unique n'existe pas encore, il faut absolument s'assurer qu'un seul créera l'objet, et que l'autre obtiendra une référence vers ce nouvel objet.

La solution classique à ce problème consiste à utiliser l'exclusion mutuelle pour indiquer que l'objet est en cours d'instanciation.

Dans un langage à base de prototypes, où sont utilisés des objets mais pas des classes, un *singleton* désigne seulement un objet qui n'a pas de copies, et qui n'est pas utilisé comme prototype pour d'autres objets.

Diagramme de classes UML

La figure ci-dessous donne le diagramme de classes UML du patron de conception Singleton.

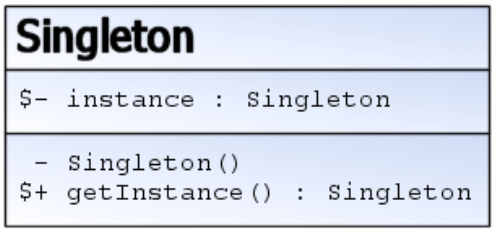

Diagramme de classes UML du patron de conception Singleton

Implémentations

Java

Voici une solution écrite en Java (il faut écrire un code similaire pour chaque classe-singleton) :

```
public class Singleton
{
    private static Singleton INSTANCE = null;

    /*
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {}

    /*
     * Le mot-clé synchronized sur la méthode de création
     * empêche toute instanciation multiple même par
     * différents threads.
     * Retourne l'instance du singleton.
     */
    public synchronized static Singleton getInstance()
    {
        if (INSTANCE == null)
            INSTANCE = new Singleton();
        return INSTANCE;
    }
}
```

Une solution variante existe cependant. Elle consiste à alléger le travail de la méthode *getInstance* en déplaçant la création de l'instance unique au niveau de la déclaration de la variable référant l'instance unique :

```
public class Singleton
{
    /*
     * Création de l'instance au niveau de la variable.
     */
    private static final Singleton INSTANCE = new Singleton();

    /*
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {}

    /*
     * Dans ce cas présent, le mot-clé synchronized n'est pas utile.
     * L'unique instanciation du singleton se fait avant
     * l'appel de la méthode getInstance(). Donc aucun risque d'accès concurrents.
     * Retourne l'instance du singleton.
     */
    public static Singleton getInstance()
    {
        return INSTANCE;
    }
}
```

À noter que la première implémentation est plus lente, étant donné que la méthode getInstance est synchronisée, les processus doivent faire la queue alors que le synchronized n'est utile qu'au premier appel, après l'instanciation, il n'y a pas d'erreur possible, toutefois il n'y pas d'autres possibilités : faire un verrouillage au niveau de l'objet dans la méthode ne suffit pas[1].

La dernière implémentation a une faille, il est possible, en sérialisant puis en désérialisant la classe d'obtenir une seconde instance[2]. Une troisième implémentation permet de pallier ce problème, le singleton est l'unique élément d'une énumération.

```
public enum Singleton implements Serializable
{
    INSTANCE;
}
```

C++

Voici une implémentation possible en C++, connue sous le nom de "singleton de Meyers". Le singleton est un objet *static* et *local*. Attention : cette solution n'est pas sûre dans un contexte multi-thread ; elle sert plutôt à donner une idée du fonctionnement d'un singleton qu'à être réellement utilisée dans un grand projet logiciel. Aucun constructeur ou destructeur ne doit être public dans les classes qui héritent du singleton.

```
template<typename T> class Singleton
{
  public:
    static T& Instance()
    {
        static T theSingleInstance; // suppose que T a un constructeur par défaut
        return theSingleInstance;
    }// ceci n'est pas valide dans un contexte multi-thread
};

class OnlyOne : public Singleton<OnlyOne>
{
    // constructeurs/destructeur de OnlyOne accessibles au Singleton
    friend class Singleton<OnlyOne>;
    //...définir ici le reste de l'interface
};
```

VB.Net

```
Public Class Singleton

    ' Variable locale pour stocker une référence vers l'instance
    Private Shared instance As Singleton = Nothing
    Private Shared ReadOnly mylock As New Object()


    ' Le constructeur est Private
    Private Sub New()
        '
```

Prototype

Le patron de conception **prototype** est utilisé lorsque la création d'une instance est complexe ou consommatrice en temps. Plutôt que créer plusieurs instances de la classe, on copie la première instance et on modifie la copie de façon appropriée.

Pour implanter ce patron il faut déclarer une classe abstraite spécifiant une méthode abstraite (virtuelle pure en C++) appelée *clone()*. Toute classe nécessitant un constructeur polymorphique dérivera de cette classe abstraite et implantera la méthode *clone()*.

Le client de cette classe, au lieu d'écrire du code invoquant directement l'opérateur "new" sur une classe explicitement connue, appellera la méthode *clone()* sur le prototype ou passera par un mécanisme fourni par un autre patron de conception (par exemple une méthode de fabrique avec un paramètre désignant la classe concrète à instancier).

Structure

Le diagramme UML de classes est le suivant :

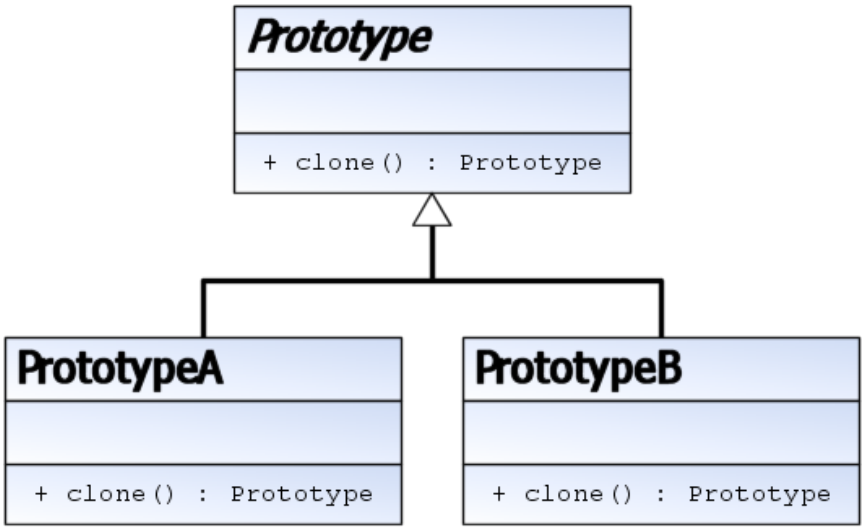

Diagramme UML des classes du patron de conception prototype

La classe *Prototype* sert de modèle principal pour la création de nouvelles copies. Les classes *PrototypeA* et *PrototypeB* viennent spécialiser la classe *Prototype* en venant par exemple modifier certains attributs. La méthode *clone()* doit retourner une copie de l'objet concerné. Les sous-classes peuvent hériter ou surcharger la méthode *clone()*. La classe utilisatrice va se charger d'appeler les méthodes de clonage de la classe *Prototype*.

Exemple de code en C#

```
public enum RecordType
{
    Car,
    Person
}

/// <summary>
/// Record est le Prototype
/// </summary>
public abstract class Record
{
    public abstract Record Clone();
}

/// <summary>
/// PersonRecord est un Prototype concret
/// </summary>
public class PersonRecord : Record
{
    string name;
    int age;

    public override Record Clone()
    {
        return (Record)this.MemberwiseClone(); // copie membre à membre par défaut
    }
}
```

```
/// <summary>
/// CarRecord est un autre Prototype concret
/// </summary>
public class CarRecord : Record
{
    string carname;
    Guid id;

    public override Record Clone()
    {
        CarRecord clone = (CarRecord)this.MemberwiseClone(); // copie membre à membre par défaut
        clone.id = Guid.NewGuid(); // générer un nouvel identifiant unique pour la copie
        return clone;
    }
}

/// <summary>
/// RecordFactory est la classe utilisatrice
/// </summary>
public class RecordFactory
{
    private static Dictionary<RecordType, Record> _prototypes = new Dictionary<RecordType, Record>();

    /// <summary>
    /// Constructeur
    /// </summary>
    public RecordFactory()
    {
        _prototypes.Add(RecordType.Car, new CarRecord());
        _prototypes.Add(RecordType.Person, new PersonRecord());
    }

    /// <summary>
    /// Méthode de fabrication
    /// </summary>
    public Record CreateRecord(RecordType type)
    {
        return _prototypes[type].Clone();
    }
}
```

Exemple de code en JAVA

```
/* Classe Prototype */
public class Cookie implements Cloneable
{
    public Cookie clone()
    {
        try {
            Cookie copy = (Cookie)super.clone();
            // Dans une implémentation réelle de ce patron de conception, il faudrait
            // créer la copie en dupliquant les objets contenus et en attribuants des
            // valeurs valides (exemple : un nouvel identificateur unique pour la copie).
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}

/* Prototype concrets à copier */
public class CoconutCookie extends Cookie { }

/* Classe utilisatrice */
public class CookieMachine
{
    private Cookie cookie; // peut aussi être déclaré comme : private Cloneable cookie;

    public CookieMachine(Cookie cookie)
    {
        this.cookie = cookie;
    }

    public Cookie makeCookie()
    {
        return cookie.clone();
    }

    public static void main(String args[])
    {
        Cookie        tempCookie =  null;
        Cookie        prot       = new CoconutCookie();
```

```
        CookieMachine cm         = new CookieMachine(prot);

        for (int i=0; i<100; i++)
            tempCookie = cm.makeCookie();
    }
}
```

Exemples

Exemple où **prototype** s'applique : supposons une classe pour interroger une base de données. À l'instanciation de cette classe on se connecte et on récupère les données de la base avant d'effectuer tous types de manipulation. Par la suite, il sera plus performant pour les futures instances de cette classe de continuer à manipuler ces données que de réinterroger la base. Le premier objet de connexion à la base de données aura été créé directement puis initialisé. Les objets suivants seront une copie de celui-ci et donc ne nécessiteront pas de phase d'initialisation.

Fabrique

La **fabrique** (*factory*) est un patron de conception de création utilisé en programmation orientée objet. Comme les autres modèles de création, la fabrique a pour rôle l'instanciation d'objets divers dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique.

Comme en général, les fabriques sont uniques dans un programme, on utilise souvent le patron de conception singleton pour gérer leur création.

Diagramme de classes UML

Le patron de conception Fabrique peut être représenté par le diagramme de classes UML suivant :

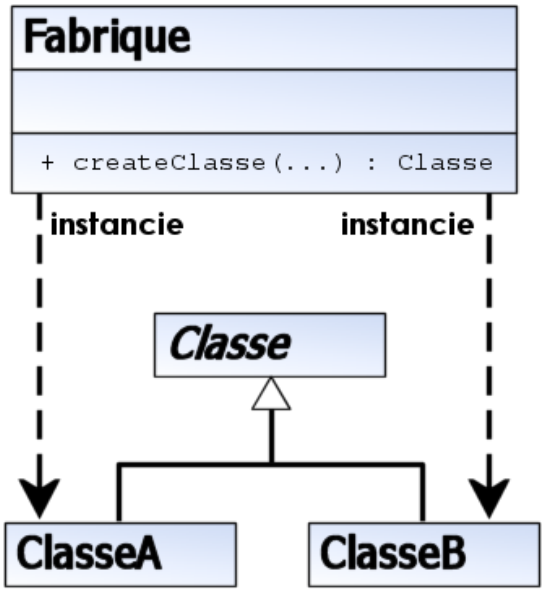

Diagramme de classe pour le patron de conception Fabrique

Exemples

Base de données

Considérons une interface de base de données qui supporte de nombreux types de champs. Les champs d'une table sont représentés par une classe abstraite appelée *Champ*. Chaque type de champ est associé à une sous-classe de *Champ*, donnant par exemple : *ChampTexte*, *ChampNumerique*, *ChampDate*, ou *ChampBooleen*.

La classe *Champ* possède une méthode *display()* permettant d'afficher le contenu d'un champ dans une interface utilisateur. Un objet de contrôle est créé pour chaque champ, la nature de chaque contrôle dépendant du type du champ associé : le contenu de *ChampTexte* sera affiché dans un champ de saisie texte, celui de *ChampBooleen* sera représenté par une case à cocher.

Pour résoudre ce problème, *Champ* contient une méthode de fabrique appelée *createControl()* et appelée depuis *display()* pour créer l'objet adéquat.

Animaux

Dans l'exemple suivant, en Java, une classe « fabrique » des objets dérivés de la classe Animal en fonction du nom de l'animal passé en paramètre. Il est également possible d'utiliser une interface comme type de retour de la fonction.

```
public class FabriqueAnimal
{
    private static FabriqueAnimal instance = new FabriqueAnimal();

    private FabriqueAnimal() {}

    public static FabriqueAnimal getFabriqueAnimalInstance() {
        return instance;
    }
```

```
    public Animal getAnimal(String typeAnimal) throws ExceptionCreation
    {
        if (typeAnimal.equals("chat"))
            return new Chat();
        else if (typeAnimal.equals("chien"))
            return new Chien();
        else
            throw new ExceptionCreation("Impossible de créer un " + typeAnimal);
    }
}

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

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

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

public class FabriqueExemple{
   public static void main(String [] args){
     FabriqueAnimal fabrique =  FabriqueAnimal.getFabriqueAnimalInstance();
     try {
       Animal animal = fabrique.getAnimal("chien");
       animal.myName();
     } catch(ExceptionCreation e) {
       e.printStackTrace();
     }
   }
}
```

Utilisation

- Les fabriques sont utilisées dans les toolkits ou les frameworks, car leurs classes sont souvent dérivées par les applications qui les utilisent.
- Des hiérarchies de classes parallèles peuvent avoir besoin d'instancier des classes les une des autres.

Autres avantages et variantes

Bien que la principale utilisation de la Fabrique soit d'instancier dynamiquement des sous-classes, elle possède d'autres avantages qui ne sont pas liés à l'héritage des classes. On peut donc écrire des fabriques qui ne font pas appel au polymorphisme pour créer plusieurs types d'objets (on fait alors appel à des méthodes statiques).

Noms descriptifs

Les langages orientés objet doivent généralement avoir un nom de constructeur identique au nom de la classe, ce qui peut être ambigu s'il existe plusieurs constructeurs (par surcharge). Les méthodes de fabrication n'ont pas cette obligation et peuvent avoir un nom qui décrit mieux leur fonction. Dans l'exemple suivant, les nombres complexes sont créés à partir de deux nombres réels qui peuvent être interprétés soit comme coordonnées polaires, soit comme coordonnées cartésiennes ; l'utilisation de méthodes de fabrication ne laisse aucune ambiguïté :

```
public class Complex
{
    public static Complex fromCartesian(double real, double imag)
    {
        return new Complex(real, imag);
    }

    public static Complex fromPolar(double rho, double theta)
    {
        return new Complex(rho * cos(theta), rho * sin(theta));
    }

    private Complex(double a, double b)
    {
        //...
```

```
    }
}

Complex c = Complex.fromPolar(1, pi); // Identique à fromCartesian(-1, 0)
```

Le constructeur de la classe est ici privé, ce qui oblige à utiliser les méthodes de fabrication qui ne prêtent pas à confusion.

Encapsulation

Les méthodes de fabrication permettent d'encapsuler la création des objets. Ce qui peut être utile lorsque le processus de création est très complexe, s'il dépend par exemple de fichiers de configuration ou d'entrées utilisateur.

L'exemple ci-dessous présente un programme qui crée des icônes à partir de fichiers d'images. Ce programme sait traiter plusieurs formats d'images représentés chacun par une classe :

```
public interface ImageReader
{
    public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader
{
    public GifReader( InputStream in )
    {
        // Vérifier qu'il s'agit d'une image GIF,
        // lancer une exception si ce n'est pas le cas,
        // décoder l'image sinon.
    }

    public DecodedImage getDecodedImage()
    {
        return decodedImage;
    }
}

public class JpegReader implements ImageReader
{
    //... même principe
}
```

Chaque fois que le programme lit une image, il doit créer le lecteur adapté à partir d'informations trouvées dans le fichier. Cette partie peut être encapsulée dans une méthode de fabrication :

```
public class ImageReaderFactory
{
    public static ImageReader getImageReader( InputStream is )
    {
        int imageType = figureOutImageType( is );

        switch( imageType )
        {
            case ImageReaderFactory.GIF:
                return new GifReader( is );
            case ImageReaderFactory.JPEG:
                return new JpegReader( is );
        // etc.
        }
    }
}
```

Le type d'image et le lecteur correspondant peuvent ici être stockés dans un tableau associatif, ce qui évite la structure *switch* et donne une fabrique facilement extensible.

Voir aussi

Patrons associés

- Fabrique abstraite
- Monteur
- Patron de méthode

Fabrique abstraite

Une **fabrique abstraite** encapsule un groupe de fabriques ayant une thématique commune. Le code client crée une implémentation concrète de la fabrique abstraite, puis utilise les interfaces génériques pour créer des objets concrets de la thématique. Le client ne se préoccupe pas de savoir laquelle de ces fabriques a donné un objet concret, car il n'utilise que les interfaces génériques des objets produits. Ce patron de conception sépare les détails d'implémentation d'un ensemble d'objets de leur usage générique.

Un exemple de fabrique abstraite : la classe *documentCreator* fournit une interface permettant de créer différents produits (e.g. *createLetter()* et *createResume()*). Le système a, à sa disposition, des versions concrètes dérivées de la classe *documentCreator*, comme par exemple *fancyDocumentCreator* et *modernDocumentCreator*, qui possèdent chacune leur propre implémentation de *createLetter()* et *createResume()* pouvant créer des objets tels que *fancyLetter* ou *modernResume*. Chacun de ces produits dérive d'une classe abstraite simple comme *Letter* ou *Resume*, connues du client. Le code client obtient une instance de *documentCreator* qui correspond à sa demande, puis appelle ses méthodes de fabrication. Tous les objets sont créés par une implémentation de la classe commune *documentCreator* et ont donc la même thématique (ici, ils seront tous *fancy* ou *modern*). Le client a seulement besoin de savoir manipuler les classes abstraites *Letter* ou *Resume*, et non chaque version particulière obtenue de la fabrique concrète.

Une **fabrique** est un endroit du code où sont construits des objets. Le but de ce patron de conception est 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.

Avec ce patron de conception, on peut interchanger des classes concrètes sans changer le code qui les utilise, même à l'exécution. Toutefois, ce patron de conception exige un travail supplémentaire lors du développement initial, et apporte une certaine complexité qui n'est pas forcément souhaitable.

Utilisation

La *fabrique* détermine le type de l'objet *concret* qu'il faut créer, et c'est ici que l'objet est effectivement créé (dans le cas de C++, Java et C#, c'est l'instruction **new**). Cependant, la fabrique retourne un pointeur *abstrait* ou une référence *abstraite* sur l'objet concret créé.

Le code client est ainsi isolé de la création de l'objet en l'obligeant à demander à une fabrique de créer l'objet du type abstrait désiré et de lui en retourner le pointeur.

Comme la fabrique retourne uniquement un pointeur abstrait, le code client qui sollicite la fabrique ne connaît pas et n'a pas besoin de connaître le type concret précis de l'objet qui vient d'être créé. Cela signifie en particulier que :

- Le code client n'a aucune connaissance du type concret, et ne nécessite donc aucun fichier d'en-tête ou déclaration de classe requis par le type concret. Le code client n'interagit qu'avec la classe abstraite. Les objets concrets sont en effet créés par la fabrique, et le code client ne les manipule qu'avec leur interface abstraite.
- L'ajout de nouveaux types concrets dans le code client se fait en spécifiant l'utilisation d'une fabrique différente, modification qui concerne typiquement une seule ligne de code (une nouvelle fabrique crée des objets de types concrets *différents*, mais renvoie un pointeur du *même* type abstrait, évitant ainsi de modifier le code client). C'est beaucoup plus simple que de modifier chaque création de l'objet dans le code client. Si toutes les fabriques sont stockées de manière globale dans un singleton et que tout le code client utilise ce singleton pour accéder aux fabriques pour la création d'objets, alors modifier les fabriques revient simplement à modifier l'objet singleton.

Diagramme de classes UML

Le patron de conception Fabrique Abstraite peut être représenté par le diagramme UML de classes suivant :

Monteur

Le **monteur** (*builder*) est un patron de conception utilisé pour la création d'une variété d'objets complexes à partir d'un objet source. L'objet source peut consister en une variété de parties contribuant individuellement à la création de chaque objet complet grâce à un ensemble d'appels à l'interface commune de la classe abstraite *Monteur*.

Un exemple d'objet source est une liste de caractères ou d'images dans un message devant être codé. Un objet directeur est nécessaire pour fournir les informations à propos de l'objet source vers la classe *Monteur*. La classe *Monteur* abstraite pourrait être une liste d'appel de l'interface que la classe directeur utilise comme par exemple *handleCharacter()* ou *handleImage()*. Chaque version concrète de la classe *Monteur* pourrait implémenter une méthode pour ces appels ou bien simplement ignorer l'information si appelée. Un exemple de monteur concret serait *enigmaBuilder* qui chiffrerait le texte, mais ignorerait les images.

Dans l'exemple précédent, le logiciel va créer une classe *Monteur* spécifique, *enigmaBuilder*. Cet objet est passé à un objet directeur simple qui effectue une itération à travers chaque donnée du message principal de l'objet source. La classe monteur crée, incrémentalement, son projet final. Finalement, le code principal va demander l'objet final depuis le *Monteur* et ensuite détruire celui-ci et l'objet directeur. Par la suite, si jamais un remplacement de la technique de cryptage de *enigmaBuilder* par une autre se faisait sentir, une nouvelle classe Monteur pourrait être substituée avec peu de changements pour la classe directeur et le code principal. En effet, le seul changement serait la classe *Monteur* actuelle passée en paramètre au directeur.

But : Séparer la construction d'un objet complexe de la représentation afin que le même processus de construction puisse créer différentes représentations.

Diagramme de classes

La structure des classes du patron de conception Monteur peut être représenté par le diagramme de classes UML suivant :

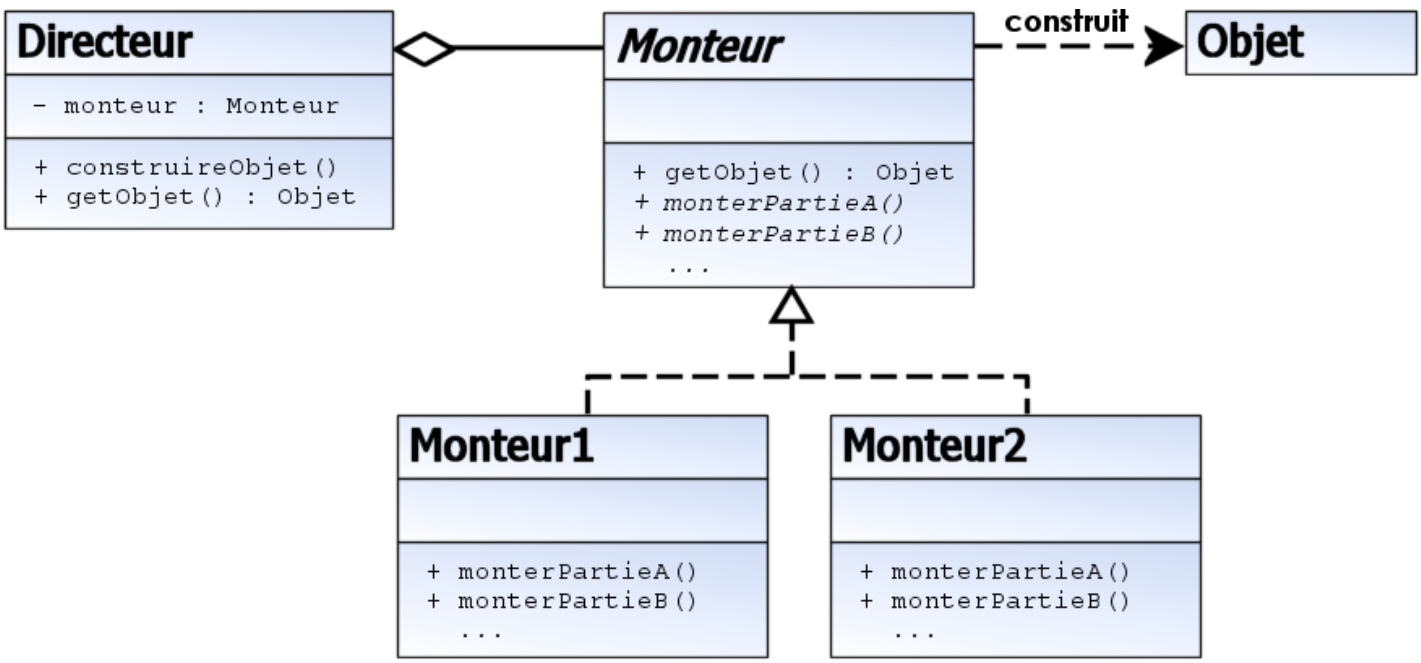

Diagramme UML des classes du patron de conception Monteur

- **Monteur**
  - interface abstraite pour construire des objets
- **Monteur1** et **Monteur2**
  - fournissent une implémentation de Monteur
  - construisent et assemblent les différentes parties des objets
- **Directeur**
  - construit un objet en appelant les différentes méthodes afin de construire chaque partie de l'objet complexe
- **Objet**
  - l'objet complexe en cours de construction

Exemples

```
struct WinGUIFactory : public GUIFactory
{
    Button* createButton()
    {
        return new WinButton();
    }
};

struct OSXGUIFactory : public GUIFactory
{
    Button* createButton()
    {
        return new OSXButton();
    }
};

struct Application
{
    Application(GUIFactory* factory)
    {
        Button* button = factory->createButton();
        button->paint();
    }
};

/* application : */
int main()
{
    GUIFactory* factory1 = new WinGUIFactory();
    GUIFactory* factory2 = new OSXGUIFactory();

    Application* winApp  = new Application (factory1);
    Application* osxApp  = new Application (factory2);

    delete factory1, factory2;

    return 0;
}
```

C#

```
/*
 * Exemple : GUIFactory
 */
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:GUIFactory
{
    public override Button createButton()
    {
        return(new WinButton());
    }
}

class OSXFactory:GUIFactory
{
    public override Button createButton()
    {
        return(new OSXButton());
    }
}

abstract class Button
{
    public string caption;
    public abstract void paint();
}

class WinButton:Button
```

```
{
    public override void paint()
    {
        Console.WriteLine("I'm a WinButton: "+caption);
    }
}

class OSXButton:Button
{
    public override void paint()
    {
        Console.WriteLine("I'm a OSXButton: "+caption);
    }
}

class Application
{
    static void Main(string[] args)
    {
        GUIFactory aFactory = GUIFactory.getFactory();
        Button aButton = aFactory.createButton();
        aButton.caption = "Play";
        aButton.paint();
    }
    // affiche :
    //   I'm a WinButton: Play
    // ou :
    //   I'm a OSXButton: Play
}
```

Java

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

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
{
```

Java

```java
/* Produit */
class Pizza
{
    private String pate = "";
    private String sauce = "";
    private String garniture = "";

    public void setPate(String pate)          { this.pate = pate; }
    public void setSauce(String sauce)         { this.sauce = sauce; }
    public void setGarniture(String garniture) { this.garniture = garniture; }
}

/* Monteur */
abstract class MonteurPizza
{
    protected Pizza pizza;

    public Pizza getPizza() { return pizza; }
    public void creerNouvellePizza() { pizza = new Pizza(); }

    public abstract void monterPate();
    public abstract void monterSauce();
    public abstract void monterGarniture();
}

/* MonteurConcret */
class MonteurPizzaHawaii extends MonteurPizza
{
    public void monterPate()      { pizza.setPate("croisée"); }
    public void monterSauce()     { pizza.setSauce("douce"); }
    public void monterGarniture() { pizza.setGarniture("jambon+ananas"); }
}

/* MonteurConcret */
class MonteurPizzaPiquante extends MonteurPizza
{
    public void monterPate()      { pizza.setPate("feuilletée"); }
    public void monterSauce()     { pizza.setSauce("piquante"); }
    public void monterGarniture() { pizza.setGarniture("pepperoni+salami"); }
}

/* Directeur */
class Serveur
{
    private MonteurPizza monteurPizza;

    public void setMonteurPizza(MonteurPizza mp) { monteurPizza = mp; }
    public Pizza getPizza() { return monteurPizza.getPizza(); }

    public void construirePizza()
    {
        monteurPizza.creerNouvellePizza();
        monteurPizza.monterPate();
        monteurPizza.monterSauce();
        monteurPizza.monterGarniture();
    }
}

/* Un client commandant une pizza. */
class ExempleMonteur
{
    public static void main(String[] args)
    {
        Serveur serveur = new Serveur();
        MonteurPizza monteurPizzaHawaii  = new MonteurPizzaHawaii();
        MonteurPizza monteurPizzaPiquante = new MonteurPizzaPiquante();

        serveur.setMonteurPizza(monteurPizzaHawaii);
        serveur.construirePizza();

        Pizza pizza = serveur.getPizza();
    }
}
```

PHP

```php
/* Produit */
class Pizza {
    private $_pate = "";
```

Adaptateur

Adaptateur est un patron de conception qui permet de convertir l'interface d'une classe en une autre interface que le client attend. **Adaptateur** fait fonctionner un ensemble de classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces.

Exemple

Vous voulez intégrer une classe que vous ne voulez/pouvez pas modifier.

Applicabilité

- Une API tiers convient à votre besoin fonctionnel, mais la signature de ses méthodes ne vous convient pas.
- Vous voulez normaliser l'utilisation d'anciennes classes sans pour autant en reprendre tout le code.

Diagramme de classes UML

Le patron de conception Adaptateur peut être représenté par le diagramme de classes UML suivant :

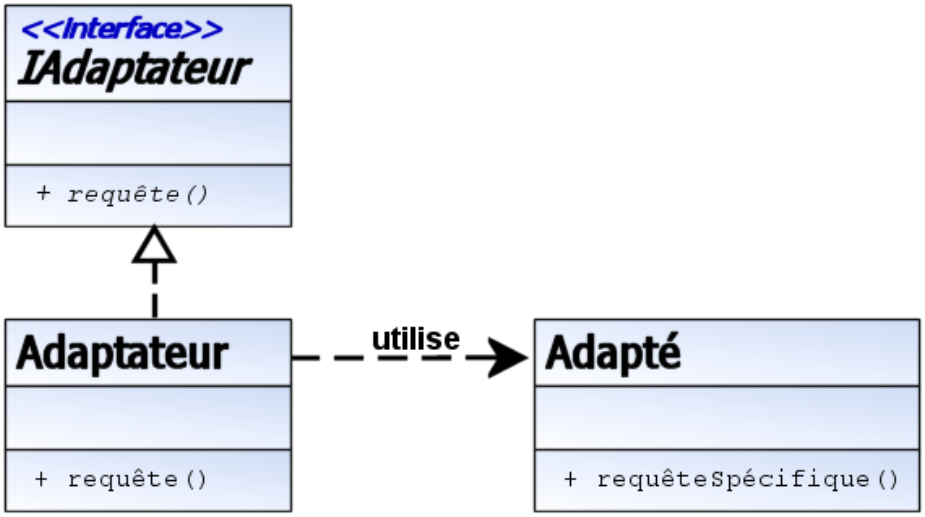

Diagramme de classes UML du patron de conception Adaptateur

- **IAdaptateur** : Définit l'interface métier utilisée par la classe cliente.
- **Adapté** : Définit une interface existante devant être adaptée.
- **Adaptateur** : Fait correspondre l'interface de **Adapté** à l'interface **IAdaptateur**, en convertissant l'appel aux méthodes de l'interface **IAdaptateur** en des appels aux méthodes de la classe **Adapté**.

Conséquences

Un objet **Adaptateur** sert de liaison entre les objets manipulés et un programme les utilisant, à simplifier la communication entre deux classes. Il est utilisé pour modifier l'interface d'un objet vers une autre interface.

Exemples

C++

Un adaptateur pour faire un carré aux coins ronds. Le code est en c++.

```
class Carre
{
  public:
    Carre();
    virtual DessineCarre();
    virtual coordonnees* GetQuatreCoins();
};

class Cercle
{
  public:
```

```
    Cercle();
    virtual DessineCercle();
    virtual void SetArc1(coordonnees* c1);
    virtual void SetArc2(coordonnees* c2);
    virtual void SetArc3(coordonnees* c3);
    virtual void SetArc4(coordonnees* c4);
    virtual coordonnees* GetCoordonneesArc();
};

class CarreCoinsRondAdapter: public Carre, private Cercle
{
  public:
    CarreCoinsRondAdapter();

    virtual void DessineCarre()
    {
        SetArc1(new coordonnees(0,0));
        SetArc2(new coordonnees(4,0));
        SetArc3(new coordonnees(4,4));
        SetArc4(new coordonnees(0,4));
        // Fonction qui dessine les lignes entre les arcs
        DessineCercle();
    }

    virtual coordonnees* GetQuatreCoins()
    {
       return GetCoordonneesArc();
    }
};
```

C#

```
/// <summary> la signature "IAdaptateur" utilisée par le client </summary>
public interface IDeveloppeur
{
    string EcrireCode();
}

/// <summary> concrétisation normale de "IAdaptateur" par une classe </summary>
class DeveloppeurLambda : IDeveloppeur
{
    public string EcrireCode()
    {
        return "main = putStrLn \"Algorithme codé\"";
    }
}

/// <summary> "Adapté" qui n'a pas la signature "IAdaptateur" </summary>
class Architecte
{
    public string EcrireAlgorithme()
    {
        return "Algorithme";
    }
}

/// <summary> "Adaptateur" qui encapsule un objet qui n'a pas la bonne signature</summary>
class Adaptateur : IDeveloppeur
{
    Architecte _architecte;
    public Adaptateur (Architecte archi)
    {
        _architecte = archi;
    }
    public string EcrireCode()
    {
        return string.Format(
            "let main() = printfn \"{0} codé\"",
            _architecte.EcrireAlgorithme());
    }
}

//_____________________________________________________________________
// Implémentation

/// <summary> "Client" qui n'utilise que les objets qui respectent la signature </summary>
class Client
{
    void Utiliser(IDeveloppeur developpeur)
    {
        Console.WriteLine(developpeur.EcrireCode());
    }
```

```
    static void Main()
    {
        var client = new Client();

        IDeveloppeur developpeur1 = new DeveloppeurLambda();
        client.Utiliser(developpeur1);

        var architecte = new Architecte();
        IDeveloppeur developpeur2 = new Adaptateur(architecte);
        client.Utiliser(developpeur2);
    }
}
```

Utilisations connues

On peut également utiliser un adaptateur lorsque l'on ne veut pas implémenter toutes les méthodes d'une certaine interface. Par exemple, si l'on doit implémenter l'interface MouseListener en Java, mais que l'on ne souhaite pas implémenter de comportement pour toutes les méthodes, on peut dériver la classe MouseAdapter. Celle-ci fournit en effet un comportement par défaut (vide) pour toutes les méthodes de MouseListener.

Exemple avec MouseAdapter :

```
public class MouseBeeper extends MouseAdapter
{
    public void mouseClicked(MouseEvent e)
    {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Exemple avec MouseListener :

```
public class MouseBeeper implements MouseListener
{
    public void mouseClicked(MouseEvent e)
    {
        Toolkit.getDefaultToolkit().beep();
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

Voir aussi

Patron de conception connexes

- Pont
- Décorateur
- Proxy

Liens et documents externes

- Adaptateur sur DoFactory (http://www.dofactory.com/Patterns/PatternAdapter.aspx)

Objet composite

Dans ce patron de conception, un **objet composite** est constitué d'un ou de plusieurs objets similaires (ayant des fonctionnalités similaires). L'idée est de manipuler un groupe d'objets de la même façon que s'il s'agissait d'un seul objet. Les objets ainsi regroupés doivent posséder des opérations communes, c'est-à-dire un "dénominateur commun".

Quand l'utiliser

Vous avez l'impression d'utiliser de multiples objets de la même façon, souvent avec des lignes de code identiques ou presque. Par exemple, lorsque la seule et unique différence entre deux méthodes est que l'une manipule un objet de type `Carré`, et l'autre un objet `Cercle`. Lorsque, pour le traitement considéré, la différenciation n'a *pas besoin* d'exister, il serait plus simple de considérer l'ensemble de ces objets comme homogène.

Un exemple

Un exemple simple consiste à considérer l'affichage des noms de fichiers contenus dans des dossiers :

- Pour un fichier, on affiche ses informations.
- Pour un dossier, on affiche les informations des fichiers qu'il contient.

Dans ce cas, le patron composite est tout à fait adapté :

- L'Objet est de façon générale ce qui peut être contenu dans un dossier : un fichier ou un dossier,
- L'ObjetSimple est un fichier, sa méthode affiche() affiche simplement le nom du fichier,
- L'ObjetComposite est un dossier, il contient des objets (c'est à dire des fichiers et des dossiers). Sa méthode affiche() parcourt l'ensemble des objets qu'il contient (fichier ou dossier) en appelant leur méthode affiche().

Diagramme de classes UML

Le patron de conception Objet composite peut être représenté par le diagramme de classes UML suivant :

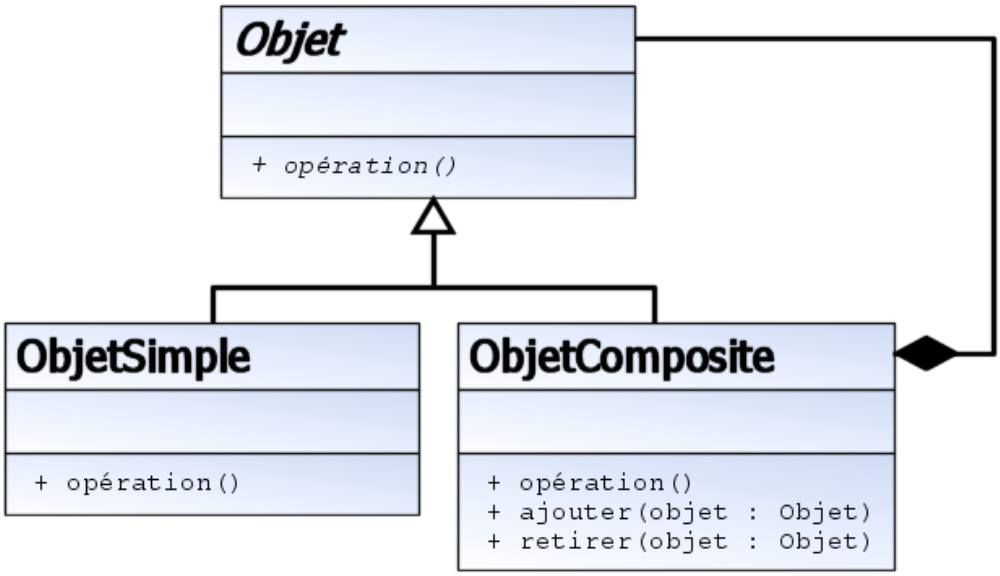

Diagramme des classes UML du patron de conception Objet composite

- **Objet**
  - déclare l'interface pour la composition d'objets
  - met en œuvre le comportement par défaut
- **ObjetSimple**
  - représente les objets manipulés, ayant une interface commune
- **ObjetComposite**
  - définit un comportement pour les composants ayant des enfants
  - stocke les composants enfants
  - met en œuvre la gestion des composants enfants

```
        }
    }

    //Ajoute le graphique à la composition.
    public void add(Graphic graphic)
    {
        mChildGraphics.add(graphic);
    }

    //Retire le graphique de la composition.
    public void remove(Graphic graphic)
    {
        mChildGraphics.remove(graphic);
    }

}

class Ellipse implements Graphic
{
    //Imprime le graphique.
    public void print()
    {
        System.out.println("Ellipse");
    }

}

public class Program
{
    public static void main(String[] args)
    {
        //Initialise quatre ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialise three graphiques composites
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes les graphiques
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Imprime le graphique complet (quatre fois la chaîne "Ellipse").
        graphic.print();
    }
}
```

PHP 5

```
<?php
class Component
{
    // Attributs
    private $basePath;
    private $name;
    private $parent;

    public function __construct($name, CDirectory $parent = null)
    {
        // Debug : echo "constructor Component";
        $this->name = $name;
        $this->parent = $parent;
        if($this->parent != null)
        {
            $this->parent->addChild($this);
            $this->basePath = $this->parent->getPath();
        }
        else
        {
            $this->basePath = '';
        }
    }
```

Décorateur

Un **décorateur** est le nom d'un patron de conception de structure.

Un décorateur permet d'attacher dynamiquement de nouveaux comportements ou responsabilités à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de nouvelles fonctionnalités.

Objectifs

Beaucoup de langages de programmation orientés objets ne permettent pas de créer dynamiquement des classes, et la conception ne permet pas de prévoir quelles combinaisons de fonctionnalités sont utilisées pour créer autant de classes.

Exemple : Supposons qu'une classe de fenêtre Window ne gère pas les barres de défilement. On créé une sous-classe ScrollingWindow. Maintenant, il faut également ajouter une bordure. Le nombre de classes croît rapidement si on utilise l'héritage : on créé les classes WindowWithBorder et ScrollingWindowWithBorder.

Par contre, les classes décoratrices sont allouées dynamiquement à l'utilisation, permettant toutes sortes de combinaisons. Par exemple, les classes d'entrées-sorties de Java permettent différentes combinaisons (FileInputStream + ZipInputStream, ...).

En reprenant l'exemple des fenêtres, on créé les classes ScrollingWindowDecorator et BorderedWindowDecorator sous-classes de Window stockant une référence à la fenêtre à « décorer ». Étant donné que ces classes décoratives dérivent de Window, une instance de ScrollingWindowDecorator peut agir sur une instance de Window ou une instance de BorderedWindowDecorator.

Exemples

Java

Ce programme illustre l'exemple des fenêtres ci-dessus.

```
// interface des fenêtres
interface Window
{
    public void draw(); // dessine la fenêtre
    public String getDescription(); // retourne une description de la fenêtre
}

// implémentation d'une fenêtre simple, sans barre de défilement
class SimpleWindow implements Window
{
    public void draw()
    {
        // dessiner la fenêtre
    }

    public String getDescription()
    {
        return "fenêtre simple";
    }
}
```

Les classes suivantes contiennent les décorateurs pour toutes les classes de fenêtres, y compris les décorateurs eux-mêmes.

```
// classe décorative abstraite, implémente Window
abstract class WindowDecorator implements Window
{
    protected Window decoratedWindow; // la fenêtre décorée

    public WindowDecorator (Window decoratedWindow)
    {
        this.decoratedWindow = decoratedWindow;
    }
}

// décorateur concret ajoutant une barre verticale de défilement
class VerticalScrollBarDecorator extends WindowDecorator
{
    public VerticalScrollBarDecorator (Window decoratedWindow)
    {
        super(decoratedWindow);
    }
```

```
    public void draw()
    {
        drawVerticalScrollBar();
        decoratedWindow.draw();
    }

    private void drawVerticalScrollBar()
    {
        // afficher la barre verticale de défilement
    }

    public String getDescription()
    {
        return decoratedWindow.getDescription() + ", avec une barre verticale de défilement";
    }
}


// décorateur concret ajoutant une barre horizontale de défilement
class HorizontalScrollBarDecorator extends WindowDecorator
{
    public HorizontalScrollBarDecorator (Window decoratedWindow)
    {
        super(decoratedWindow);
    }

    public void draw()
    {
        drawHorizontalScrollBar();
        decoratedWindow.draw();
    }

    private void drawHorizontalScrollBar()
    {
        // afficher la barre horizontale de défilement
    }

    public String getDescription()
    {
        return decoratedWindow.getDescription() + ", avec une barre horizontale de défilement";
    }
}
```

Voici un programme de test qui créé une fenêtre pleinement décorée (barres de défilement verticale et horizontale) et affiche sa description :

```
public class DecoratedWindowTest
{
    public static void main(String[] args)
    {
        Window decoratedWindow =
            new HorizontalScrollBarDecorator (
            new VerticalScrollBarDecorator(
            new SimpleWindow()
            )
            );

        // afficher la description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

Ce programme affiche :

```
fenêtre simple, avec une barre verticale de défilement, avec une barre horizontale de défilement
```

Les deux décorateurs utilisent la description de la fenêtre décorée et ajoute un suffixe.

C#

Ici l'héritage est utilisé.

```
//_____________________________________________________________________
// Déclarations

abstract class Voiture
```

Patrons de comportement

Un patron de comportement permet de résoudre les problèmes liés aux comportements, à l'interaction entre les classes.

Les différents patrons de comportement sont les suivants :

Chaîne de responsabilité
: Permet de construire une chaîne de traitement d'une même requête.

Commande
: Encapsule l'invocation d'une commande.

Interpréteur
: Interpréter un langage spécialisé.

Itérateur
: Parcourir un ensemble d'objets à l'aide d'un objet de contexte (curseur).

Médiateur
: Réduire les dépendances entre un groupe de classes en utilisant une classe Médiateur comme intermédiaire de communication.

Mémento
: Mémoriser l'état d'un objet pour pouvoir le restaurer ensuite.

Observateur
: Intercepter un évènement pour le traiter.

État
: Gérer différents états à l'aide de différentes classes.

Stratégie
: Changer dynamiquement de stratégie (algorithme) selon le contexte.

Patron de méthode
: Définir un modèle de méthode en utilisant des méthodes abstraites.

Visiteur
: Découpler classes et traitements, afin de pouvoir ajouter de nouveaux traitements sans ajouter de nouvelles méthodes aux classes existantes.

```
public:
    EmailLogger(int level)
    {
        this->level = level;
        this->next = NULL;
    }

    void writeMessage(string msg)
    {
        cout << "Notification par email : " << msg << endl;
    }
};

class ErrorLogger : public Logger
{
public:
    ErrorLogger(int level)
    {
        this->level = level;
        this->next = NULL;
    }

    void writeMessage(string msg)
    {
        cerr << "Erreur : " << msg << endl;
    }
};

int main()
{
     // Construction de la chaîne de responsabilité
    DebugLogger logger(Logger::DEBUG);
    EmailLogger logger2(Logger::NOTICE);
    ErrorLogger logger3(Logger::ERR);
    logger.setNext(&logger2);
    logger2.setNext(&logger3);

    logger.message("Entering function y.", Logger::DEBUG);    // Utilisé par DebugLogger
    logger.message("Step1 completed.", Logger::NOTICE);    // Utilisé par DebugLogger et EmailLogger
    logger.message("An error has occurred.", Logger::ERR);    // Utilisé par les trois Loggers

    return 0;
}
```

Java

Le même exemple que le précédent, en Java.

```
import java.util.*;

/**
    Classe de gestion de journalisation abstraite.
*/
abstract class Logger
{
    public static final int
        ERR = 0,
        NOTICE = 1,
        DEBUG = 2;

    protected int level;

    /** L'élément suivant dans la chaîne de responsabilité. */
    protected Logger next;

    protected Logger(int level)
    {
        this.level = level;
        this.next = null;
    }

    public Logger setNext( Logger l)
    {
        next = l;
        return l;
    }

    public void message( String msg, int priority )
    {
        if ( priority <= level )
            writeMessage( msg );
```

```
        if ( next != null )
            next.message( msg, priority );
    }

    abstract protected void writeMessage( String msg );
}

/**
    Journalisation sur la sortie standard.
*/
class StdoutLogger extends Logger
{
    public StdoutLogger( int level ) { super(level); }

    protected void writeMessage( String msg )
    {
        System.out.println( "Writing to stdout: " + msg );
    }
}

/**
    Journalisation par courriel.
*/
class EmailLogger extends Logger
{
    public EmailLogger( int level ) { super(level); }

    protected void writeMessage( String msg )
    {
        System.out.println( "Sending via email: " + msg );
    }
}

/**
    Journalisation sur l'erreur standard.
*/
class StderrLogger extends Logger
{
    public StderrLogger( int level ) { super(level); }

    protected void writeMessage( String msg )
    {
        System.err.println( "Sending to stderr: " + msg );
    }
}

/**
    Classe principale de l'application.
*/
public class ChainOfResponsibilityExample
{
    public static void main( String[] args )
    {
        // Construire la chaîne de responsabilité
        //   StdoutLogger -> EmailLogger -> StderrLogger
        Logger l,l1;
        l1 = l = new StdoutLogger( Logger.DEBUG );
        l1 = l1.setNext(new EmailLogger( Logger.NOTICE ));
        l1 = l1.setNext(new StderrLogger( Logger.ERR ));

        // Traité par StdoutLogger
        l.message( "Entering function y.", Logger.DEBUG );

        // Traité par StdoutLogger et EmailLogger
        l.message( "Step1 completed.", Logger.NOTICE );

        // Traité par les trois loggers
        l.message( "An error has occurred.", Logger.ERR );
    }
}
```

Ce programme affiche :

```
Writing to stdout:   Entering function y.
Writing to stdout:   Step1 completed.
Sending via e-mail:  Step1 completed.
Writing to stdout:   An error has occurred.
Sending via e-mail:  An error has occurred.
Writing to stderr:   An error has occurred.
```