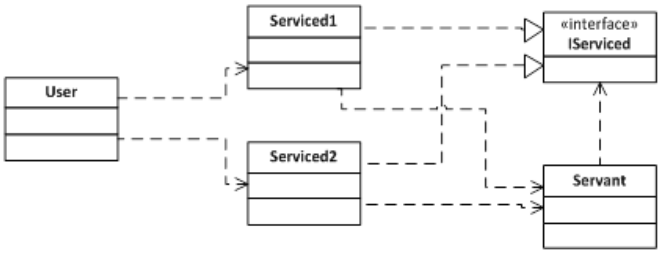
Un livre de Wikilivres.



Patrons de conception

Introduction

Un patron de conception (plus connu sous le terme anglais « *Design pattern* ») est une solution générique permettant de résoudre un problème spécifique.

En général, un patron de conception décrit une structure de classes utilisant des interfaces, et s'applique donc à des développements logiciels utilisant la programmation orientée objet.

Cette notion peut paraître nouvelle, mais il s'agit en fait plutôt d'un nouveau terme pour désigner les algorithmes, et les structures de données permettant de résoudre différents problèmes. Un exemple : une liste chaînée permet d'avoir un groupe d'éléments dont le nombre n'est pas fixe, contrairement aux tableaux.

Cette notion ne s'applique donc pas seulement à la programmation orientée objet. Un autre exemple est l'architecture MVC (Modèle-Vue-Contrôleur) définissant une architecture où les fonctions (ou les classes en POO) ont un rôle bien défini.

Comment lire ce livre ?

Cette page d'introduction présente des notions générales à tous les patrons de conception. Les pages suivantes répertorient les différents patrons de conception existants. Il est donc recommandé de lire cette page, puis de lire/rechercher les pages selon le but recherché par le lecteur :

- Pour apprendre à programmer, il peut être intéressant de voir les solutions proposées par les patrons de conception, en commençant par les plus simples, ou les plus connus.
- Pour le développeur recherchant une solution à un problème, regarder les différents patrons de la catégorie du problème concerné.
- Pour le développeur cherchant à apprendre un nouveau patron de conception, aller directement à la page concernée.

Chaque page est rédigée de façon à pouvoir être lue indépendamment des autres.

Pour plus d'informations, consulter les ouvrages et les sites web cités dans le chapitre Bibliographie et liens.

Pour chercher un patron de conception à partir de son nom en anglais ou en français, utiliser le moteur de recherche suivant :

Pertinence d'utilisation et implémentation

Utiliser des patrons de conception pour le développement de logiciels peut paraître compliqué ou superflu. Dans les applications les plus simples, l'utilisation de patrons de conception peut générer une complexité dans le code source.

Cependant, les patrons de conception sont généralement utiles pour les applications ayant une taille importante et/ou dans les projets où plusieurs applications différentes interagissent entre elles (via un moyen de communication).

Il faut également bien comprendre le rôle d'un patron de conception afin de vérifier qu'il s'applique au cas rencontré, et ne pas ajouter inutilement une complexité si les avantages liés à l'utilisation d'un patron de conception ne sont pas requis.

Apprendre par cœur chaque patron de conception est inutile car ils sont issus de la logique de conception. Une solution bien conçue peut convenir aussi bien qu'un patron de conception, et dans beaucoup de cas elle est mieux adaptée à certaines situations.

Les patrons de conception, lorsqu'ils sont détaillés, ne doivent pas forcément être appliqués à la lettre, car ils apportent des solutions génériques. La solution étant générique, lors de l'implémentation d'un patron de conception dans un langage donné (souvent imposé lors de la reprise d'un projet existant), il sera certainement nécessaire de l'adapter à la situation :

- Si une notion utilisée dans le patron de conception n'existe pas (exemple, la notion de classe en langage C), il faudra trouver une notion proche à adapter (exemple, en C, utiliser une structure) ;
- Le contexte d'utilisation de la solution peut poser un nouveau problème à résoudre dans l'implémentation du patron de conception. Par exemple, utilisé dans un contexte multi-threads, le patron de conception "Singleton" devra être implémenté de façon synchronisée.

Classification

Il y a différentes façons de classer les différents patrons de conception. Ce livre présente les plus connus, classés en fonction de leurs auteurs, puis, si possible, classés par catégorie fonctionnelle.

Ensemble de patrons de conception

Il y a différents ensembles de patrons de conception, créés par différents auteurs.

- Les plus connus sont ceux du « Gang of Four » (ou GoF : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) décrits dans leur livre « Design Patterns -- Elements of Reusable Object-Oriented Software » (voir bibliographie) en 1995. Les patrons de conception tirent leur origine des travaux de l'architecte Christopher Alexander dans les années 70.

- Les patrons GRASP sont des patrons créés par Craig Larman qui décrivent des règles pour affecter les responsabilités aux classes d'un programme orienté objets pendant la conception, en liaison avec la méthode de conception BCE (pour « Boundary Control Entity » - en français MVC « Modèle Vue Contrôleur »).
- Les patrons d'entreprise (*Enterprise Design Pattern*) créés par Martin Fowler, décrivent des solutions à des problèmes courants dans les applications professionnelles. Par exemple, des patrons de couplage entre un modèle objet et une base de donnée relationnelle.
- D'autres patrons créés par divers auteurs existent et décrivent des solutions à des problèmes différents de ceux vus précédemment.

Catégorie fonctionnelle

Les patrons de conception peuvent être classés en fonction du type de problème qu'ils permettent de résoudre. Par exemple, les patrons de conception de création résolvent les problèmes liés à la création d'objets.

Patrons du « Gang of Four »

Les patrons de conception créés par le « Gang of Four » (GoF en abrégé) sont décrits dans leur livre « Design Patterns -- Elements of Reusable Object-Oriented Software ». Les 4 auteurs du livre (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) sont surnommés la bande des quatre (« Gang of Four » en anglais).

Ces patrons de conception sont classés en trois catégories :

- **Les patrons de création** décrivent comment régler les problèmes d'instanciation de classes, c'est à dire de création et de configuration d'objets (objet en unique exemplaire par exemple).
- **Les patrons de structure** décrivent comment structurer les classes afin d'avoir le minimum de dépendance entre l'implémentation et l'utilisation dans différents cas.
- **Les patrons de comportement** décrivent une structure de classes pour le comportement de l'application (répondre à un évènement par exemple).

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.

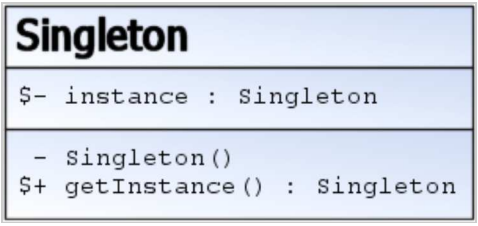


Diagramme de classes UML du patron de conception Singleton

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()
        '
    End Sub

    ' La méthode GetInstance doit être Shared
    Public Shared Function GetInstance() As Singleton

        SyncLock (mylock)
```

```
            ' Si pas d'instance existante on en crée une...
            If instance Is Nothing Then
                instance = New Singleton
            End If

            ' On retourne l'instance de Singleton
            Return instance
        End SyncLock

    End Function

End Class
```

C#

```
public class Singleton
{
    private static Singleton _instance;
    static readonly object instanceLock = new object();

    private Singleton()
    {

    }

    public static Singleton getInstance()
    {
        lock (instanceLock)
        {
        if (_instance == null)
           _instance = new Singleton();


        return _instance;
        }
    }
}
```

ActionScript

```
public class Singleton
{
    private static var _instance:Singleton;

    public static function getInstance():Singleton{
        if (_instance== null) {
            _instance= new Singleton();
        }

        return _instance;
    }

    public function Singleton() {

    }
}
```

Python

Implémentation simple

```
class  Singleton(object):
    _instance = None       # Attribut statique de classe
    def __new__(cls): # __new__ classmethod implicite: la classe 1e paramètre
        "méthode de construction standard en Python"
        if cls._instance is None:
            cls._instance = object.__new__(cls)
        return cls._instance

# Utilisation
mon_singleton1 =  Singleton()
mon_singleton2 =  Singleton()

# mon_singleton1 et mon_singleton2 renvoient à la même instance
assert mon_singleton1 is mon_singleton2
```

Les deux variables référencent ici la même instance. Cette solution ne nécessite pas de méthode spécifique pour accéder à l'instance de classe (comparativement à d'autres langages, où il est d'usage d'implémenter une méthode `getInstance()`, par exemple).

Considérations "avancées"

D'après le développeur Python Alex Martelli, *le patron de conception Singleton a un nom élégant mais trompeur, car il met l'accent sur l'identité plutôt que sur l'état de l'objet.* D'où l'apparition d'un patron alternatif, appelé *Borg* [3], pour lequel toutes les instances partagent le même état. Il est généralement accepté par la communauté de développeurs Python que le partage d'état entre instances est plus élégant que la mise en cache de la création d'instances identiques lors de l'initialisation de la classe.

L'implémentation de cet état partagé est quasiment transparente en Python:

```
class Borg:
   __shared_state = {} # variable de classe contenant l'état à partager
   def __init__(self):
       # copie de l'état lors de l'initialisation d'une nouvelle instance
       self.__dict__ = self.__shared_state
   # suivi de ce qui est nécessaire à la classe, et c'est tout!
```

L'implémentation ci-dessus peut être améliorée en tirant partie du nouveau type de classe de Python [4], ne nécessitant ainsi qu'une seule instance :

```
class Singleton (object):
   instance = None
    def __new__(classe, *args, **kargs):
        if classe.instance is None:
            classe.instance = object.__new__(classe, *args, **kargs)
        return classe.instance

# Utilisation:
monSingleton1 =  Singleton()
monSingleton2 =  Singleton()

# monSingleton1 et  monSingleton2 renvoient à la même instance.
assert monSingleton1 is monSingleton2
```

- La méthode *__init__* est exécutée pour chaque appel à *Singleton()*.
- Afin de permettre à une classe d'hériter d'un *Singleton*, la variable de classe *instance* devrait être un dictionnaire Python appartenant explicitement à la classe *Singleton*, comme illustré ci-dessous:

```
class InheritableSingleton (object):
    # Dictionnaire Python référencant les instances déjà créés
    instances = {}
    def __new__(cls, *args, **kargs):
        if InheritableSingleton.instances.get(cls) is None:
            InheritableSingleton.instances[cls] = object.__new__(cls, *args, **kargs)
        return InheritableSingleton.instances[cls]
```

Ruby

Le patron Singleton existe dans la librairie standard du langage Ruby. C'est un mixin qu'il suffit d'inclure dans la classe qui doit être un singleton.

```
require 'singleton'

class Config
  include Singleton

  def foo
    puts 'foo'
  end
end

config = Config.instance

config.foo
```

Ce script affiche "`foo`".

PHP 5

Voici une solution écrite en PHP :

```php
class Singleton {
    /**
     * Empêche la création externe d'instances.
     */
    private function __construct () {}

    /**
     * Empêche la copie externe de l'instance.
     */
    private function __clone () {}

    /**
     * Renvoi de l'instance et initialisation si nécessaire.
     */
    public static function getInstance() {
        static $instance;

        if ($instance === null) {
            $instance = new self();
        }

        return $instance;
    }

    /**
     * Méthodes dites métier
     */
    public function uneAction () {}
}

// Utilisation
Singleton::getInstance()->uneAction();
```

Avant PHP 5.3 : Difficultés liées à l'héritage d'une classe Singleton

Il faut noter qu'avant PHP 5.3, il n'est pas possible d'hériter une classe de type Singleton. En effet, l'accesseur `self::` se réfère toujours à la classe dans laquelle il est **écrit**. L'exemple ci-dessous illustre ce problème.

```php
class Singleton {
    ... // Voir ci-dessus (remplacer la visibilité private par protected)

    /**
     * Méthodes dites métier
     */
    public function uneAction() {
        echo "Singleton" ;
    }
}

class AnotherSingleton extends Singleton {
    public function uneAction() {
        echo "AnotherSingleton" ;
    }
}

echo AnotherSingleton::getInstance()->uneAction() ; // Affiche "Singleton" au lieu de "AnotherSingleton"
```

Ce problème ne peut être résolu qu'en dupliquant la méthode `getInstance` dans la classe héritée.

```php
class AnotherSingleton extends Singleton {

    public static function getInstance() {
        static $instance;

        if ($instance === null) {
            $instance = new self();
        }

        return $instance;
    }

    ... // Voir ci-dessus
}

echo AnotherSingleton::getInstance()->uneAction() ; // Affiche bien "AnotherSingleton"
```

Depuis PHP 5.3 : le mot-clé `static` et le Late Static Bindings

Avec PHP 5.3, l'apparition du Late Static Bindings (http://fr.php.net/lsb) sous la forme d'un accesseur `static::` similaire à `self::` résout ce problème. En effet, l'accesseur `static::` se réfère à la classe dans laquelle il est **appelé**. Il peut donc être hérité. C'est pourquoi l'on parle de Late Static Bindings (Résolution statique à la volée), par opposition à une résolution statique au moment de la compilation du script.

```
class Singleton {
    ... // Voir ci-dessus (remplacer la visibilité private par protected)

    final public static function getInstance() {
        static $instance;

        if ($instance === null) {
            $instance = new static();
        }

        return $instance;
    }

    ... // Voir ci-dessus
}

echo AnotherSingleton::getInstance()->uneAction() ; // Affiche bien "AnotherSingleton"
```

Perl

Pour les versions de Perl à partir de 5.10, une variable d'état fait l'affaire.

```
package MySingletonClass;
use strict;
use warnings;
use 5.10;

sub new {
    my ($class) = @_;
    state $the_instance;

    if (! defined $the_instance) {
        $the_instance = bless { }, $class;
    }
    return $the_instance;
}
```

Pour les versions plus anciennes, c'est une fermeture (*closure* en anglais) qui fera l'affaire.

```
package MySingletonClass;
use strict;
use warnings;

my $THE_INSTANCE;
sub new {
    my ($class) = @_;

    if (! defined $THE_INSTANCE) {
        $THE_INSTANCE = bless { }, $class;
    }
    return $THE_INSTANCE;
}
```

Si le module Moose est utilisé, il y a l'extension MooseX::Singleton (http://search.cpan.org/perldoc?MooseX::Singleton)

Notes et références

1. anglais Voir l'article « The dreaded double checked locking idiom in Java » (http://www.javacodegeeks.com/2011/03/dreaded-double-checked-locking-idiom-in.html).
2. Voir l'article « The Perfect Singleton », par Marek Piechut sur son blog *Development world stories* (http://jdevel.wordpress.com/2010/10/02/the-perfect-singleton/)
3. anglais Alex Martelli, « Singleton? We don't need no stinkin' singleton: the Borg design pattern (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/66531) », *ASPN Python Cookbook*. Consulté le 2006-09-07
4. anglais New-style classes (http://www.python.org/doc/newstyle/), *Python documentation*

Liens externes

Divers

- français Explication du Singleton et de ses variantes en Java (http://web.archive.org/20071026123713/jvillage.wordpress.com/2007/06/19/design-pattern-de-creation-singleton) par Java Village
- français Explication du Singleton en C++ (http://tfc.duke.free.fr/coding/singleton.html)

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 :

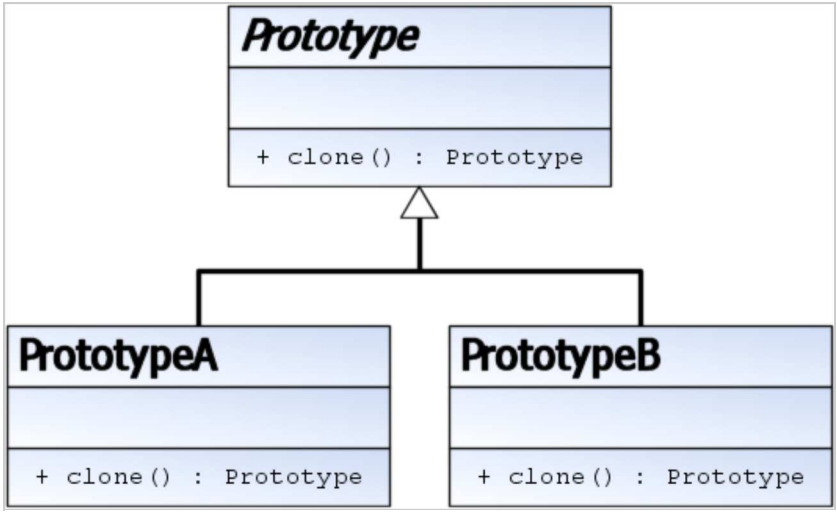


Diagramme UML des classes du patron de conception prototype

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 :

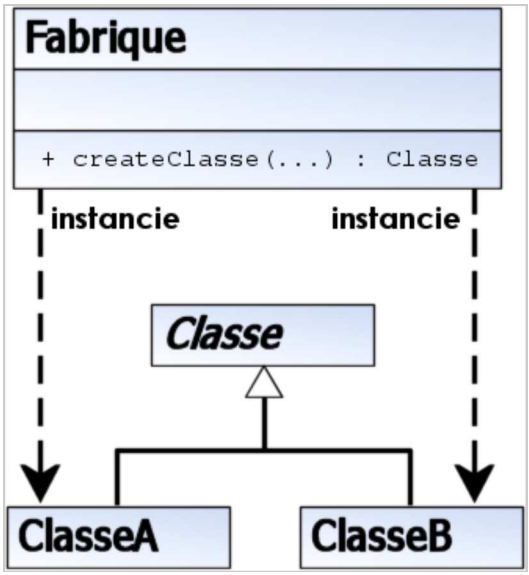


Diagramme de classe pour le patron de conception Fabrique

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 ;
    }

    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{
   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");
   }
}

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

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

Liens externes

- (anglais) A formal definition of Factory method pattern in LePUS 2 (a formal language for defining design patterns) (http://www.eden-study.org/lepus/specifications/gof/factory-method.html)
- (anglais) Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?FactoryMethodPattern)

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 :

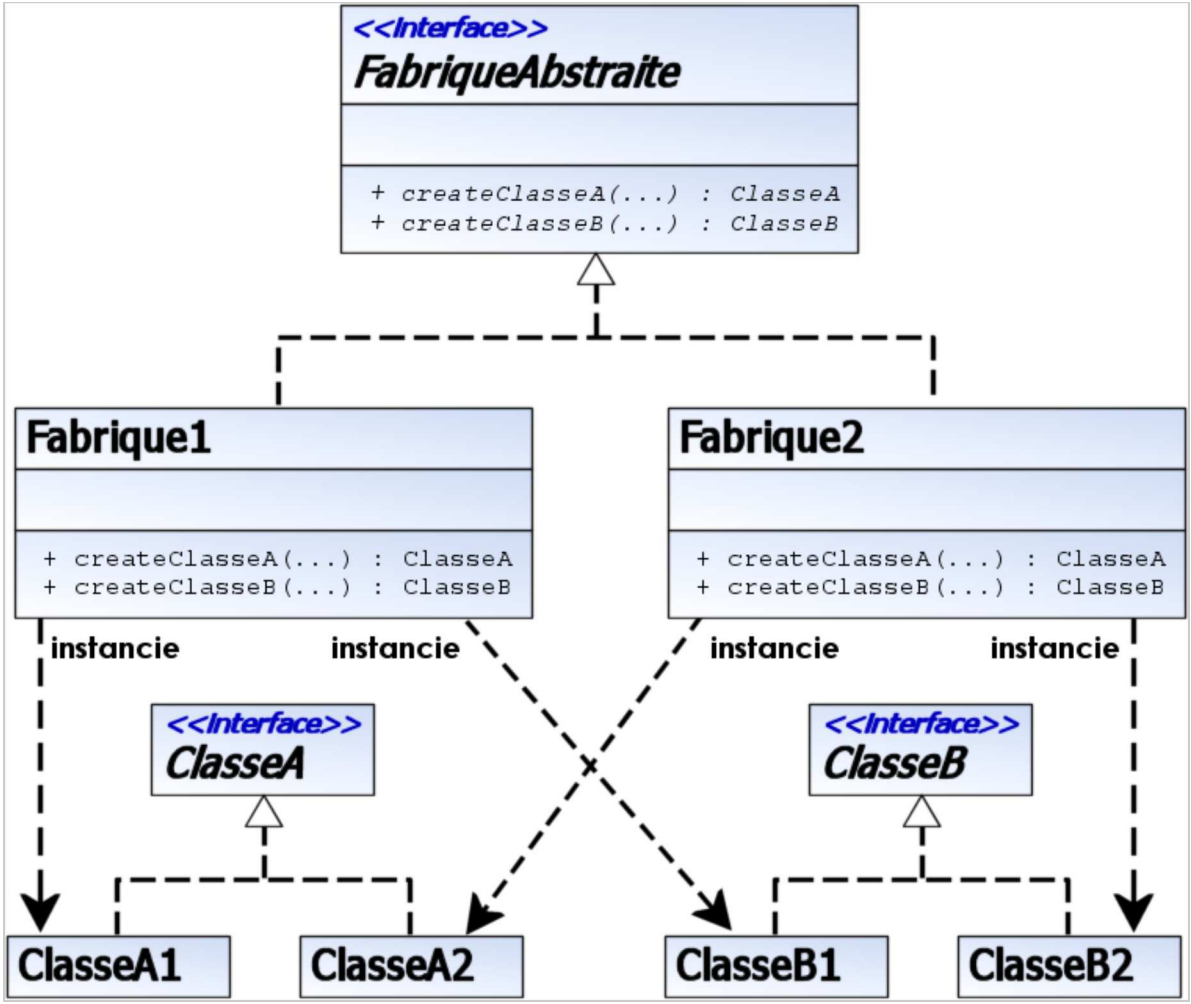


Diagramme UML de classes du patron de conception Fabrique Abstraite

Dans le diagramme ci-dessus, la fabrique abstraite (ou interface) permet de créer des instances de classes dérivées des classes abstraites *ClasseA* et *ClasseB*. Les fabriques concrètes permettent de sélectionner quelles classes concrètes dérivées des classe abstraites *ClasseA* et *ClasseB* sont instanciées.

Exemples

C++

```
/* exemple d'une fabrique abstraite d'eléments d'IHM en C++ */

struct Button
{
    virtual void paint() = 0;
};

struct WinButton : public Button
{
    void paint()
    {
        std::cout << " I'm a window button \n";
    }
};

struct OSXButton : public Button
{
    void paint()
    {
        std::cout << " I'm a OSX button \n";
    }
};

struct GUIFactory
{
    virtual Button* createButton() = 0;
};
```

```
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
{
    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());
    }
}

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

Perl

```
# GUIFactory example on Perl

package GUIFactory;

sub getFactory($$) {
    shift; # skip class
    my $toolkit = shift;
    if ($toolkit eq 'GTK') {
        return(GtkFactory->new);
    } else {
        return(TkFactory->new);
    }
}

package GtkFactory;
use base 'GUIFactory';

sub new {
    bless({}, shift);
}

sub createButton {
    return(GtkButton->new);
}

package TkFactory;
use base 'GUIFactory';

sub new {
    bless({}, shift);
}

sub createButton() {
    return(TkButton->new);
}

package Button;

sub new {
    $class = shift;
    my $self = {};
    $self{caption} = '';
    bless($self, $class);
    return $self;
}

package GtkButton;
use base 'Button';

sub paint() {
    print "I'm a GtkButton\n";
}

package TkButton;
use base 'Button';
```

```
sub paint() {
    print "I'm a TkButton\n";
}

package main;

my $aFactory = GUIFactory->getFactory;
my $aButton = $aFactory->createButton;
$aButton->{caption} = "Play";
$aButton->paint();
```

PHP

```
/*
 * Fabrique abstraite
 */
abstract class GUIFactory {
    public static function getFactory() {
        $sys = readFromConfigFile("OS_TYPE");
        if ($sys == 0) {
            return(new WinFactory());
        } else {
            return(new OSXFactory());
        }
    }
    public abstract function createButton();
}

class WinFactory extends GUIFactory {
    public function createButton() {
        return(new WinButton());
    }
}

class OSXFactory extends GUIFactory {
    public function createButton() {
        return(new OSXButton());
    }
}

abstract class Button {
    private $_caption;
    public abstract function render();

    public function getCaption(){
        return $this->_caption;
    }
    public function setCaption($caption){
        $this->_caption = $caption;
    }
}

class WinButton extends Button {
    public function render() {
        return "Je suis un WinButton: ".$this->getCaption();
    }
}

class OSXButton extends Button {
    public function render() {
        return "Je suis un OSXButton: ".$this->getCaption();
    }
}

$aFactory = GUIFactory::getFactory();
$aButton = $aFactory->createButton();
$aButton->setCaption("Démarrage");
echo $aButton->render();

// affiche :
//   Je suis un WinButton: Démarrage
// ou :
//   Je suis un OSXButton: Démarrage
```

Eiffel

```
--
-- GUI Factory example
--
```

```
class GUI_FACTORY_FOR_CONFIG feature
    get_factory: GUI_FACTORY is
        once
            inspect read_from_config_file("OS_TYPE")
            when 0 then
                create {WIN_FACTORY} Result
            else
                create {OSX_FACTORY} Result
            end
        end
    end

deferred class GUI_FACTORY feature
    create_button: BUTTON is deferred end
end

class WIN_FACTORY inherit GUI_FACTORY feature
    create_button: WIN_BUTTON is do create Result end
end

class OSX_FACTORY inherit GUI_FACTORY feature
    create_button: OSX_BUTTON is do create Result end
end

deferred class BUTTON feature
    caption: STRING
    set_caption(value: like caption) is do caption := value end
    paint is deferred end
end

class WIN_BUTTON inherit BUTTON feature
    paint is do print("I'm a WIN_BUTTON: "+caption+"%N") end
end

class OSX_BUTTON inherit BUTTON feature
    paint is do print("I'm a OSX_BUTTON: "+caption+"%N") end
end

class APPLICATION inherit GUI_FACTORY_FOR_CONFIG creation main feature
    main is local button : BUTTON do
        button := get_factory.create_button
        button.set_caption("Play")
        button.paint
    end
end
```

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 crypterait 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 :

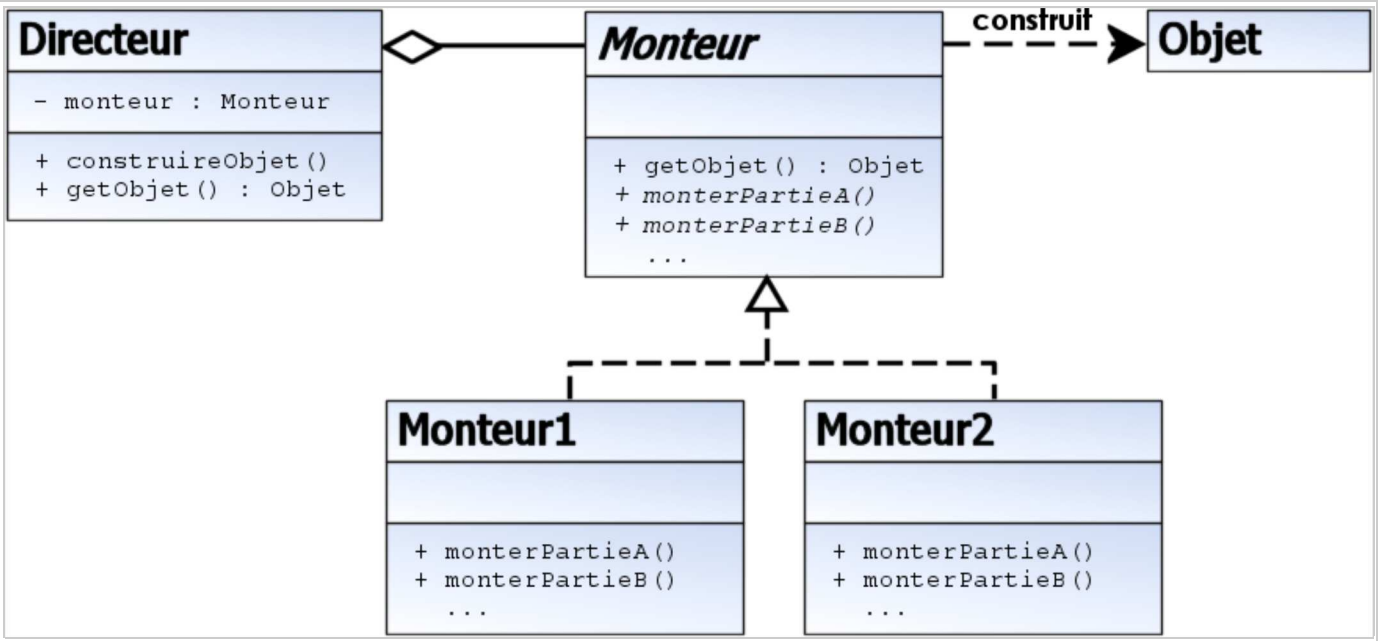


Diagramme UML des classes du patron de conception Monteur

- **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

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

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

```
    public function setPate($pate)           { $this->_pate = $pate; }
    public function setSauce($sauce)         { $this->_sauce = $sauce; }
    public function setGarniture($garniture) { $this->_garniture = $garniture; }
}


/* Monteur */
abstract class MonteurPizza {
    protected $_pizza;

    public function getPizza()           { return $this->_pizza; }
    public function creerNouvellePizza() { $this->_pizza = new Pizza(); }

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

/* MonteurConcret */
class MonteurPizzaHawaii extends MonteurPizza {
    public function monterPate()      { $this->_pizza.setPate("croisée"); }
    public function monterSauce()     { $this->_pizza.setSauce("douce"); }
    public function monterGarniture() { $this->_pizza.setGarniture("jambon+ananas"); }
}

/* MonteurConcret */
class MonteurPizzaPiquante extends MonteurPizza {
    public function monterPate()      { $this->_pizza.setPate("feuilletée"); }
    public function monterSauce()     { $this->_pizza.setSauce("piquante"); }
    public function monterGarniture() { $this->_pizza.setGarniture("pepperoni+salami"); }
}


/* Directeur */
class Serveur {
    private $_monteurPizza;

    public function setMonteurPizza(MonteurPizza $mp) { $this->_monteurPizza = $mp; }
    public function getPizza()                        { return $this->_monteurPizza->getPizza(); }

    public void construirePizza() {
        $this->_monteurPizza->creerNouvellePizza();
        $this->_monteurPizza->monterPate();
        $this->_monteurPizza->monterSauce();
        $this->_monteurPizza->monterGarniture();
    }
}

/* Un client commandant une pizza. */
$serveur = new Serveur();
$monteurPizzaHawaii  = new MonteurPizzaHawaii();
$monteurPizzaPiquante = new MonteurPizzaPiquante();

$serveur->setMonteurPizza($monteurPizzaHawaii);
$serveur->construirePizza();

$pizza = $serveur->getPizza();
```

Voir aussi

Patrons associés

- Fabrique
- Fabrique Abstraite
- Patron de méthode

Patrons de structure

Un patron de structure permet de résoudre les problèmes liés à la structuration des classes et leur interface en particulier.

Les différents patrons de structure sont les suivants :

Pont
: Utilisation d'interface à la place d'implémentation spécifique pour permettre l'indépendance entre l'utilisation et l'implémentation.

Façade
: Ce patron de conception permet de simplifier l'utilisation d'une interface complexe.

Adaptateur
: Ce patron permet d'adapter une interface existante à une autre interface.

Objet composite
: Ce patron permet de manipuler des objets composites à travers la même interface que les éléments dont ils sont constitués.

Proxy
: Ce patron permet de substituer une classe à une autre en utilisant la même interface afin de contrôler l'accès à la classe (contrôle de sécurité ou appel de méthodes à distance).

Poids-mouche
: Ce patron permet de diminuer le nombre de classes créées en regroupant les classes similaires en une seule et en passant les paramètres supplémentaires aux méthodes appelées.

Décorateur
: Ce patron permet d'attacher dynamiquement de nouvelles responsabilités à un objet.

Pont

Le **pont** est un patron de conception qui permet de découpler l'interface d'une classe et son implémentation. Ainsi l'interface et l'implémentation peuvent varier séparément.

Attention, à ne pas confondre ce patron avec l'adaptateur. En effet, l'adaptateur est utilisé pour adapter l'interface d'une classe vers une autre interface (donc pour faire en sorte que l'interface d'une ou plusieurs classes ressemble à l'interface d'une classe en particulier).

Le pont est lui utilisé pour découpler l'interface de l'implémentation. Ainsi, vous pouvez modifier ou changer l'implémentation d'une classe sans devoir modifier le code client (si l'interface ne change pas bien entendu).

Exemple : formes géométriques

Considérons une classe représentant la classe de base de formes géométriques, et ses classes (cercles, rectangles, triangles, ...). Tous les types de formes ont des propriétés communes (une couleur par exemple) et des méthodes abstraites communes (calcul de surface par exemple) implémentées par les classes dérivées (comment calculer la surface d'un cercle, ...).

Toutes les formes peuvent également se dessiner à l'écran. Mais la façon de dessiner dépend de l'environnement graphique et du système d'exploitation. Plutôt que d'ajouter une méthode par environnement possible à chacune des formes, le patron de conception Pont suggère de créer une interface séparée pour les primitives de dessin. Cette interface est utilisée par les différentes formes qui alors ne dépendent pas de l'implémentation.

Diagramme de classes UML

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

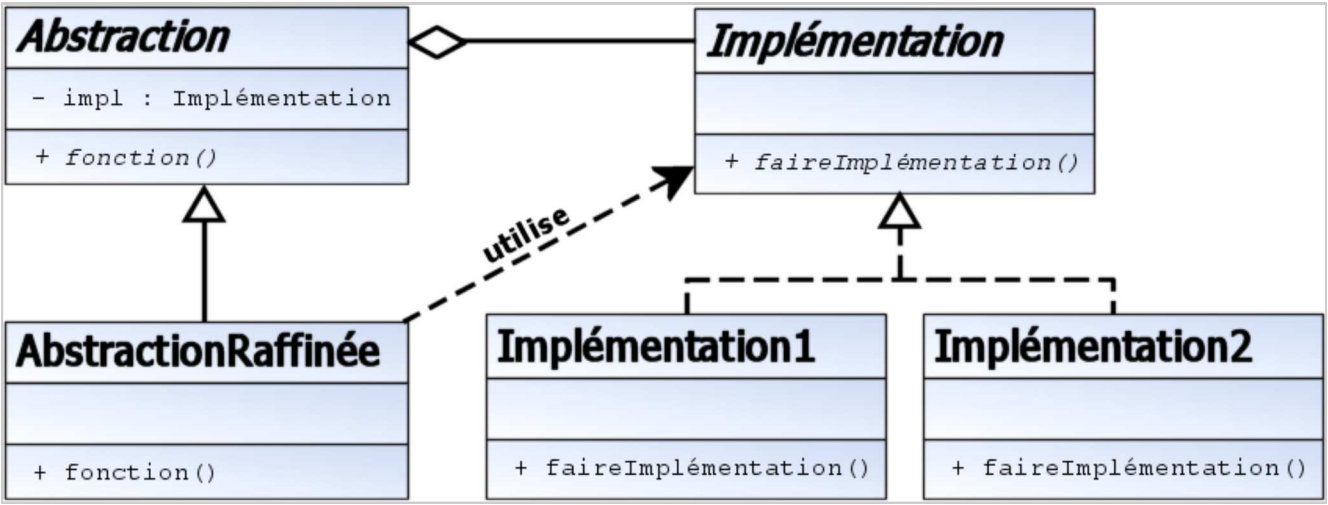


Diagramme UML de classes du patron de conception Pont

Exemples de code

Ruby

```ruby
class Abstraction
  def initialize(implementor)
    @implementor = implementor
  end

  def operation
    raise 'Implementor object does not respond to the operation method' unless @implementor.respond_to?(:operat
    @implementor.operation
  end
end

class RefinedAbstraction < Abstraction
  def operation
    puts 'Starting operation...'
    super
  end
end

class Implementor
```

```
  def operation
    puts 'Doing neccessary stuff'
  end
end

class ConcreteImplementorA < Implementor
  def operation
    super
    puts 'Doing additional stuff'
  end
end

class ConcreteImplementorB < Implementor
  def operation
    super
    puts 'Doing other additional stuff'
  end
end

normal_with_a = Abstraction.new(ConcreteImplementorA.new)
normal_with_a.operation
# Doing neccessary stuff
# Doing additional stuff

normal_with_b = Abstraction.new(ConcreteImplementorB.new)
normal_with_b.operation
# Doing neccessary stuff
# Doing other additional stuff

refined_with_a = RefinedAbstraction.new(ConcreteImplementorA.new)
refined_with_a.operation
# Starting operation...
# Doing neccessary stuff
# Doing additional stuff

refined_with_b = RefinedAbstraction.new(ConcreteImplementorB.new)
refined_with_b.operation
# Starting operation...
# Doing neccessary stuff
# Doing other additional stuff
```

Java

Le programme Java 5 ci-dessous illustre l'exemple des formes géométriques donné précédemment et affiche :

```
API1.cercle position 1.000000:2.000000 rayon 7.500000
API2.cercle position 5.000000:7.000000 rayon 27.500000
```

```
/** "Implémentation" */
interface DrawingAPI
{
    public void drawCircle(double x, double y, double radius);
}

/** "Implémentation1" */
class DrawingAPI1 implements DrawingAPI
{
    public void drawCircle(double x, double y, double radius)
    {
        System.out.printf("API1.cercle position %f:%f rayon %f\n", x, y, radius);
    }
}

/** "Implémentation2" */
class DrawingAPI2 implements DrawingAPI
{
    public void drawCircle(double x, double y, double radius)
    {
         System.out.printf("API2.cercle position %f:%f rayon %f\n", x, y, radius);
    }
}

/** "Abstraction" */
interface Shape
{
   public void draw();                             // bas niveau
   public void resizeByPercentage(double pct);     // haut niveau
}

/** "AbstractionRaffinée" */
class CircleShape implements Shape
```

```
{
    private double x, y, radius;
    private DrawingAPI drawingAPI;

    public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI)
    {
        this.x = x;  this.y = y;  this.radius = radius;
        this.drawingAPI = drawingAPI;
    }

    // bas niveau, càd spécifique à une implémentation
    public void draw()
    {
        drawingAPI.drawCircle(x, y, radius);
    }

    // haut niveau, càd spécifique à l'abstraction
    public void resizeByPercentage(double pct)
    {
        radius *= pct;
    }
}

/** Classe utilisatrice */
class BridgePattern
{
    public static void main(String[] args)
    {
        Shape[] shapes = new Shape[2];
        shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
        shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

        for (Shape shape : shapes)
        {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}
```

Voir aussi

Patrons associés

- Patron de méthode
- Stratégie

Façade

Le patron de conception **façade** a pour but de cacher une conception et une interface ou un ensemble d'interfaces complexes difficiles à comprendre (cette complexité étant apparue "naturellement" avec l'évolution du sous-système en question). La façade permet de simplifier cette complexité en fournissant une interface simple du sous-système. Habituellement, la façade est réalisée en réduisant les fonctionnalités de ce dernier mais en fournissant toutes les fonctions nécessaires à la plupart des utilisateurs.

La façade encapsule la complexité des interactions entre les objets métier participant à un workflow.

L'utilisation d'une façade a les avantages suivants :

- simplifier l'utilisation et la compréhension d'une bibliothèque logicielle car la façade possède des méthodes pratiques pour les tâches courantes,
- rendre le code source de la bibliothèque plus lisible pour la même raison,
- réduire les dépendances entre les classes utilisatrices et les classes internes à la bibliothèque puisque la plupart des classes utilisatrices utilisent la façade, ce qui autorise plus de flexibilité pour le développement du système,
- rassembler une collection d'API complexes en une unique et meilleure API (orientée tâches utilisateurs).

Un adaptateur est utilisé quand la façade doit respecter une interface particulière et doit supporter un comportement polymorphique.

Diagramme des classes UML

Le patron de conception Façade peut être représenté par le diagramme de classe UML suivant :

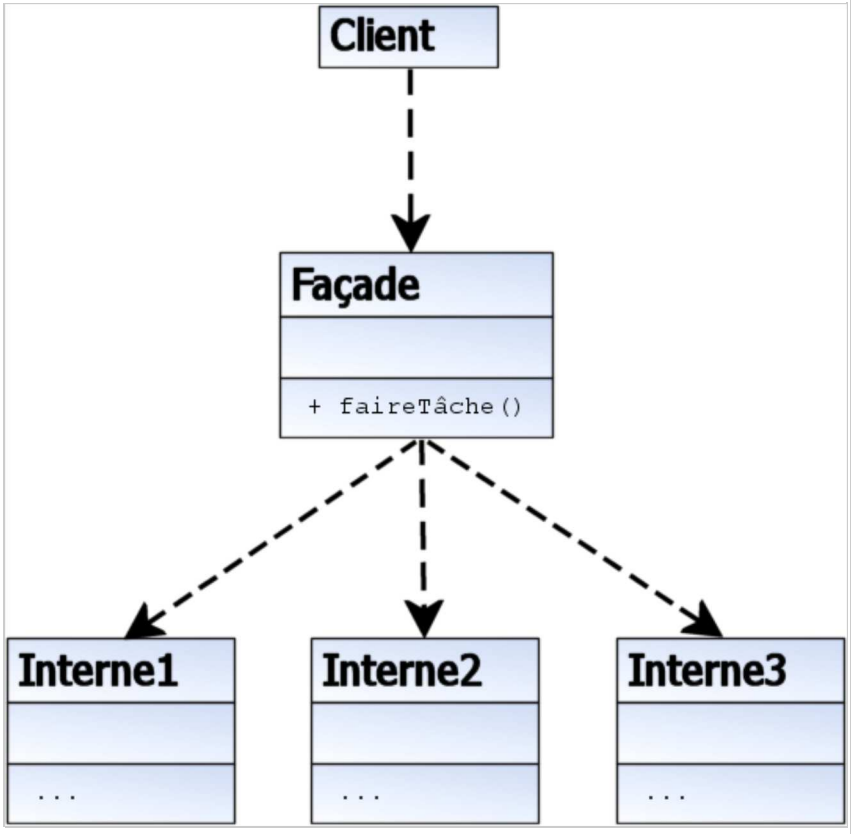


Diagramme de classes UML du patron de conception Façade

Exemple

Java

L'exemple suivant cache une API de gestion de calendrier compliquée, derrière une façade plus simple. Il affiche :

```
Date : 1980-08-20
20 jours après : 1980-09-09
```

```
import java.util.*;
```

```java
/* Façade */
class UserfriendlyDate
{
    GregorianCalendar gcal;

    public UserfriendlyDate(String isodate_ymd)
    {
        String[] a = isodate_ymd.split("-");
        gcal = new GregorianCalendar(
            Integer.parseInt(a[0]),    // année
            Integer.parseInt(a[1])-1,  // mois (0 = Janvier)
            Integer.parseInt(a[2]) );  // jour
    }

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

    public String toString()
    {
        return String.format("%1$tY-%1$tm-%1$td", gcal);
    }
}

/* Classe utilisatrice */
class FacadePattern
{
    public static void main(String[] args)
    {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println("Date : "+d);
        d.addDays(20);
        System.out.println("20 jours après : "+d);
    }
}
```

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 des 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 :

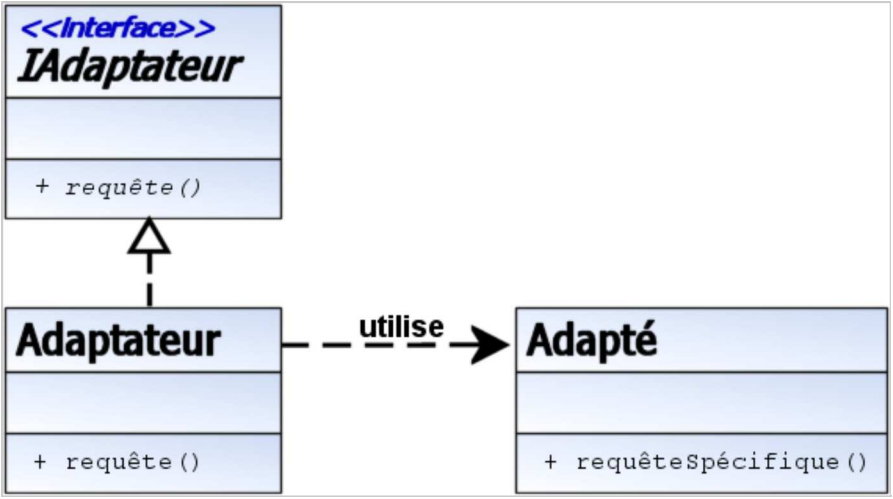


Diagramme de classes UML du patron de conception Adaptateur

- **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 :

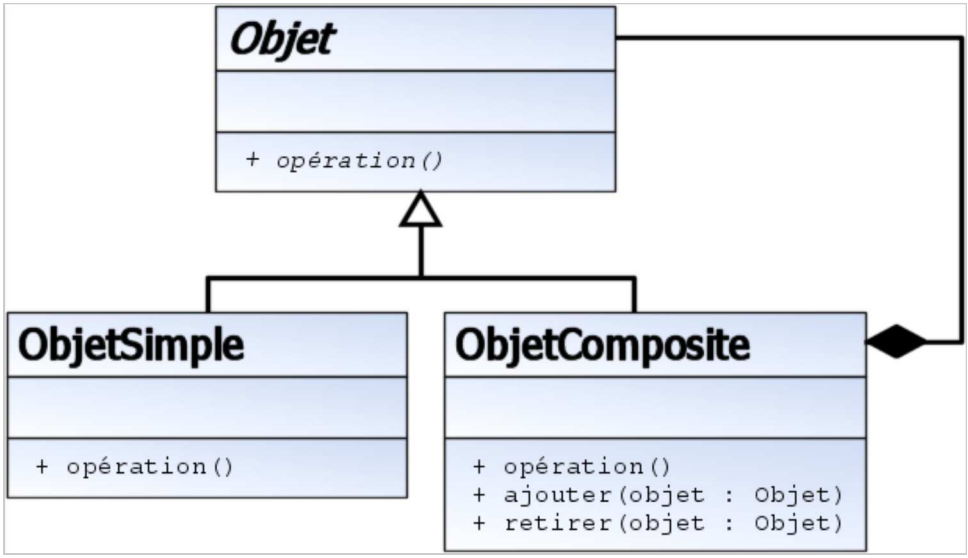


Diagramme des classes UML du patron de conception Objet composite

- **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

La classe utilisatrice manipule les objets de la composition à travers l'interface `Objet`.

Implémentations

C++

```
// Cet exemple représente la hiérarchie d'un système de fichiers : fichiers/répertoires

class Composant
{
// L'objet abstrait qui sera 'composé' dans le composite
// Dans notre exemple, il s'agira d'un fichier ou d'un répertoire

    public:
        // Parcours récursif de l'arbre
        // (Utilisez plutôt le Design Pattern "Visiteur" à la place de cette fonction)
        virtual void listerNoms() = 0 ;

    protected:
        std::string name;
};

class File : public Composant
{
    public:
        void listerNoms()
        {
            std::cout << name << std::endl;
        }
};

class Repertoire : public Composant
{
//Le répertoire est aussi un 'composant' car il peut être sous-répertoire
    protected:
        std::list<Composant*> files; // Liste des composants enfants

    public:
        void listerNoms()
        {
            std::cout << "[" << name << "]" << std::endl;
            for( std::list<Composant*>::iterator it = files.begin();
                it != files.end(); it ++ )
            {
                it->listerNoms();
            }
        }
};
```

Java

L'exemple qui suit, écrit en Java, met en œuvre une classe graphique qui peut être ou bien une ellipse ou une composition de différents graphiques. Chaque graphique peut être imprimé.

Il pourrait être étendu en y ajoutant d'autres formes (rectangle etc.) et méthodes (translation etc.).

```
import java.util.ArrayList;

interface Graphic
{
    //Imprime le graphique.
    public void print();
}

class CompositeGraphic implements Graphic
{
    //Collection de graphiques enfants.
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    //Imprime le graphique.
    public void print()
    {
        for (Graphic graphic : mChildGraphics)
        {
            graphic.print();
        }
    }

    //Ajoute le graphique à la composition 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 = '';
        }
    }

    // Getters
    public function getBasePath() { return $this->basePath; }
    public function getName() { return $this->name; }
    public function getParent() { return $this->parent; }

    // Setters
```

```
    public function setBasePath($basePath) { $this->basePath = $basePath; }
    public function setName($name) { $this->name = $name; }
    public function setParent(CDirectory $parent) { $this->parent = $parent; }

    // Méthode
    public function getPath() { return $this->getBasePath().'/'.$this->getName(); }
}

class CFile extends Component
{
    // Attributs
    private $type;

    public function __construct($name, $type, CDirectory $parent = null)
    {
        // Debug : echo "constructor CFile";
        $this->type = $type;

        // Retrieve constructor of Component
        parent::__construct($name, $parent);
    }

    // Getters
    public function getType() { return $this->type; }

    // Setters
    public function setType($type) { $this->type = $type; }

    // Méthodes de la classe Component
    public function getName() { return parent::getName().'.'.$this->getType(); }
    public function getPath() { return parent::getPath().'.'.$this->getType(); }
}

class CDirectory extends Component
{

    // Attributs
    private $childs;

    public function __construct($name, CDirectory $parent = null)
    {
        // Debug : echo "constructor CDirectory";
        $this->childs = array();

        // Retrieve constructor of Component
        parent::__construct($name, $parent);
    }

    // Getters
    public function getChilds() { return $this->childs; }

    // Setters
    public function setChilds($childs) { $this->childs = $childs; }

    // Méthodes
    public function addChild(Component $child)
    {
        $child->setParent($this);
        $this->childs[] = $child;
    }

    public function showChildsTree($level = 0)
    {
        echo "<br/>".str_repeat(' ', $level).$this->getName();
        foreach($this->getChilds() as $child)
        {
            if($child instanceof self)
            {
                $child->showChildsTree($level+1);
            }
            else
            {
                echo "<br/>".str_repeat(' ', $level+1).$child->getName();
            }
        }
    }
}
?>
```

Exemple d'utilisation :

```
<?php
$root = new CDirectory('root');
$dir1 = new CDirectory('dir1', $root);
```

```
$dir2 = new CDirectory('dir2', $root);
$dir3 = new CDirectory('dir3', $root);
$dir4 = new CDirectory('dir4', $dir2);
$file1 = new CFile('file1','txt', $dir1);
$file2 = new CFile('doc', 'pdf', $dir4);

$root->showChildsTree();
?>
```

Résultat à l'écran :

```
root
 dir1
  file1.txt
 dir2
  dir4
   doc.pdf
 dir3
```

C#

Produisant un résultat similaire à l'exemple en PHP, une variante en C#.

Dans ce code la méthode Draw() correspond à la méthode opération() du diagramme de classes.

```
abstract class Composant
{
    public int Level = 0;
    public string Nom;
    public virtual void Draw()
    {
        for (var i = 0; i < Level; i++)
            Console.Write("    ");
    }
}

class Feuille : Composant
{
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Feuille : {0}", Nom);
    }
}

// Par simplicité, ni méthode Add ni Remove ni GetChild.
class Composite : Composant
{
    public Composant[] Composants; // serait private s'il y avait une méthode Add.
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine("Composite : {0}", Nom);
        foreach (Composant composant in Composants)
        {
            composant.Level = this.Level + 1;
            composant.Draw();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        //_____________________________________________________________________
        // On crée en général la structure par de multiples appels à la méthode Add.

        var cadre= new Composite() {
            Nom = "fond d'écran",
            Composants = new Composant[] {
                new Composite() {
                    Nom = "ciel",
                    Composants = new Composant[] {
                        new Feuille() { Nom="soleil" }}},
                new Composite() {
                    Nom = "mer",
                    Composants = new Composant[] {
                        new Composite() {
                            Nom="bateau",
```

```
                            Composants = new Composant[] {
                                new Feuille() { Nom="kevin" },
                                new Feuille() { Nom="katsumi" }}}}}}
        };

        //_____________________________________________________________________
        // Et voilà le pourquoi de l'utilisation du pattern:
        //     un seul appel à Draw dessine tout l'écran.

        cadre.Draw();
    }
}
```

Et le résultat :

```
Composite : fond d'écran
    Composite : ciel
        Feuille : soleil
    Composite : mer
        Composite : bateau
            Feuille : kevin
            Feuille : katsumi
```

Proxy

Un proxy est une classe se substituant à une autre classe. Par convention et simplicité, le proxy implémente la même interface que la classe à laquelle il se substitue. L'utilisation de ce proxy ajoute une indirection à l'utilisation de la classe à substituer. Le proxy sert à gérer l'accès à un objet, il agit comme un intermédiaire entre la classe utilisatrice et l'objet.

Un proxy est un cas particulier du patron de comportement État. Un proxy implémente une et une seule interface, donc se substitue à une seule classe. Un état peut implémenter un nombre quelconque d'interfaces.

Un proxy est utilisé principalement pour contrôler l'accès aux méthodes de la classe substituée. Un état est utilisé pour changer dynamiquement d'interface.

Outre l'utilisation principale du proxy (contrôle des accès), ce dernier est également utilisé pour simplifier l'utilisation d'un objet « complexe » à la base. Par exemple, si l'objet doit être manipulé à distance (via un réseau) ou si l'objet est consommateur de temps.

Il existe différents types de Proxy ayant un comportement ou un rôle différent :

- Remote proxy : fournir une référence sur un objet situé sur un espace d'adressage différent, sur la même machine ou sur une autre,
- Virtual proxy : retarder l'allocation mémoire des ressources de l'objet jusqu'à son utilisation réelle,
- Copy-on-write proxy : une forme de proxy virtuel pour retarder la copie de l'objet jusqu'à demande par la classe utilisatrice, utilisé notamment pour la modification concurrente par différents threads,
- Protection (access) proxy : fournir à chaque classe cliente un accès à l'objet avec des niveaux de protection différents,
- Firewall proxy : protéger l'accès à l'objet par des classes « malveillantes » ou vice-versa,
- Synchronization proxy : fournir plusieurs accès à l'objet synchronisé entre différentes classes utilisatrices (cas de threads multiples),
- Smart reference proxy : fournir des actions supplémentaires à chaque accès à l'objet (compteur de références, ...),
- Cache proxy : stocker le résultat d'opérations coûteuse en temps, afin de pouvoir les partager avec les différentes classes utilisatrices.

Diagramme de classes

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

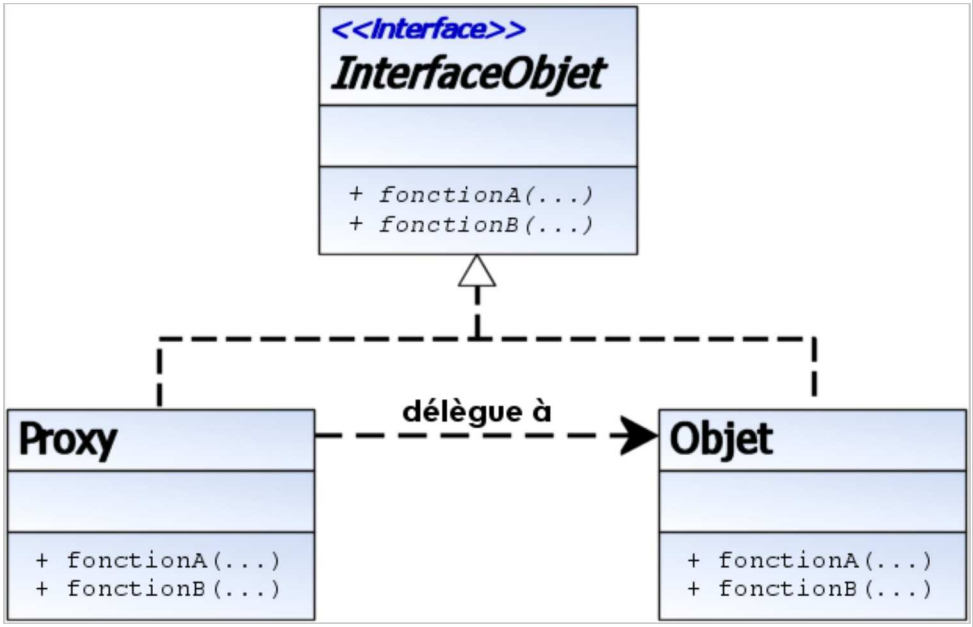


Diagramme de classes UML du patron de conception Proxy

- La classe cliente utilise l'interface InterfaceObjet pour accéder à l'objet via le Proxy,
- InterfaceObjet : interface partagée par le Proxy et l'objet accédé,
- Proxy : contrôle l'accès à l'objet. Chaque méthode délègue sa tâche à l'objet accédé, en contrôlant l'accès (vérifications, requête par réseau, ...),
- Objet : objet accédé indirectement.

Exemples

L'exemple Java suivant implémente un Proxy virtuel. La classe `ProxyImage` est utilisée pour retarder le long chargement d'un fichier jusqu'à ce que le chargement soit réellement nécessaire. Si le fichier n'est pas nécessaire, le chargement coûteux en temps n'a pas du tout lieu.

```
import java.util.*;

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("Chargement de "+filename);
        // Opération potentiellement coûteuse en temps
    }

    public void displayImage()
    {
        System.out.println("Affichage de  "+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); // Chargement sur demande seulement
        }
        image.displayImage();
    }
}

class ProxyExample
{
    public static void main(String[] args)
    {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");
        Image image3 = new ProxyImage("HiRes_10MB_Photo3");

        image1.displayImage(); // chargement nécessaire
        image2.displayImage(); // chargement nécessaire
        image1.displayImage(); // pas de chargement nécessaire, déjà fait
        // la troisième image ne sera jamais chargée : pas de temps gaspillé
    }
}
```

Le programme affiche :

```
Chargement de HiRes_10MB_Photo1
Affichage de  HiRes_10MB_Photo1
Chargement de HiRes_10MB_Photo2
Affichage de  HiRes_10MB_Photo2
Affichage de  HiRes_10MB_Photo1
```

Poids-mouche

Le **poids-mouche** est un patron de conception structurel.

Lorsque de nombreux (petits) objets doivent être manipulés, mais qu'il serait trop coûteux en mémoire s'il fallait instancier tous ces objets, il est judicieux d'implémenter le poids-mouche.

Dans le cas d'une classe représentant des données, il est parfois possible de réduire le nombre d'objets à instancier si tous ces objets sont semblables et se différencient sur quelques paramètres. Si ces quelques paramètres peuvent être extraits de la classe et les passer ensuite via des paramètres des méthodes, on peut réduire grandement le nombre d'objets à instancier.

Le patron poids-mouche est l'approche pour utiliser de telles classes. D'une part la classe avec ses données internes qui la rendent unique, et d'autre part les données externes passées à la classe en tant qu'arguments. Ce modèle est très pratique pour des petites classes très simples. Par exemple pour représenter des caractères ou des icônes à l'écran, ce type de patron de conception est apprécié. Ainsi, chaque caractère peut être représenté par une instance d'une classe contenant sa police, sa taille, etc. La position des caractères à afficher est stockée en dehors de cette classe. Ainsi, on a une et une seule instance de la classe par caractère et non une instance par caractère affiché à l'écran.

Dans le patron poids-mouche, les données n'ont pas de pointeurs vers les méthodes du type de données, parce que cela consommerait trop d'espace mémoire. À la place, les routines sont appelées directement.

Un exemple classique du patron poids-mouche : les caractères manipulés dans un traitement de texte. Chaque caractère correspond à un objet ayant une police de caractères, une taille de caractères, et d'autres données de formatage. Un long document contient beaucoup de caractères ainsi implémentés…

Exemple

Le programme Java suivant illustre l'exemple du document de traitement de texte donné ci-dessus : les « mouches » sont appelées FontData dans le programme.

Cet exemple illustre l'utilisation du poids-mouche pour réduire la mémoire allouée en chargeant seulement les données nécessaires pour faire quelques tâches immédiates à partir d'un grand objet Font en utilisant de plus petits objets FontData.

```
public enum FontEffect
{
    BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}

public final class FontData
{
    /**
     * Une table de hachage faible (weak) supprime les FontData non utilisés.
     * Les valeurs doivent être encapsulées dans des WeakReferences car dans
     * la table de hachage,les valeurs sont stockées avec des références fortes.
     */
    private static final WeakHashMap<FontData, WeakReference<FontData>>
        FLY_WEIGHT_DATA = new WeakHashMap<FontData, WeakReference<FontData>>();

    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

    private FontData(int pointSize, String fontFace, Color color, EnumSet<FontEffect> effects)
    {
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }

    public static FontData create(int pointSize, String fontFace, Color color,
        FontEffect... effects)
    {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        for (FontEffect fontEffect : effects)
        {
            effectsSet.add(fontEffect);
        }
        // seule la réduction de la quantité de mémoire occupée nous préoccupe,
        // pas le coût de création de l'objet
        FontData data = new FontData(pointSize, fontFace, color, effectsSet);
        if (!FLY_WEIGHT_DATA.containsKey(data))
        {
            FLY_WEIGHT_DATA.put(data, new WeakReference<FontData> (data));
```

```
        }
        // retourner l'unique copie non modifiable avec les données spécifiées
        return FLY_WEIGHT_DATA.get(data).get();
    }

    @Override
    public boolean equals(Object obj)
    {
        if (obj instanceof FontData)
        {
            if (obj == this)
            {
                return true;
            }
            FontData other = (FontData) obj;
            return other.pointSize == pointSize && other.fontFace.equals(fontFace)
                && other.color.equals(color) && other.effects.equals(effects);
        }
        return false;
    }

    @Override
    public int hashCode()
    {
        return (pointSize * 37 + effects.hashCode() * 13) * fontFace.hashCode();
    }

    // Accesseurs de lecture, mais pas en écriture (objet non modifiable)
}
```

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
{
    public abstract double Prix { get; }
}
```

```
class  AstonMartin : Voiture
{
    public override double Prix { get { return 999.99; } }
}

//_____________________________________________________________________
// Décorateurs

class Option : Voiture
{
    protected Voiture _originale;
    protected double _tarifOption;
    public Option(Voiture originale, double tarif)
    {
        _originale = originale;
        _tarifOption = tarif;
    }
    public override double Prix
    {
        get { return _originale.Prix + _tarifOption; }
    }
}

class Climatisation : Option
{
    public Climatisation (Voiture originale) : base(originale, 1.0) { }
}
class Parachute : Option
{
    public Parachute (Voiture originale) : base(originale, 10.0) { }
}
class Amphibie : Option
{
    public Amphibie (Voiture originale) : base(originale, 100.0) { }
}

//_____________________________________________________________________
// Implémentation

class Program
{
    static void Main()
    {
        Voiture astonMartin= new AstonMartin();
        astonMartin = new Climatisation(astonMartin);
        astonMartin = new Parachute(astonMartin);
        astonMartin = new Amphibie(astonMartin);

        Console.WriteLine(astonMartin.Prix); // affiche 1110.99
    }
}
```

Patrons connexes

- Adaptateur

Références externes

- Le patron "décorateur" (http://smeric.developpez.com/java/uml/decorateur/) sur le site *developpez.com*
- (anglais) Description (http://web.archive.org/20030109113839/home.earthlink.net/~huston2/dp/decorator.html) par Vince Huston
- (anglais) Decorator Pattern (http://www.oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf)
- (anglais) C# Conception (http://awprofessional.com/articles/article.asp?p=31350&rl=1) par James W. Cooper
- (anglais) Approche PHP (http://phppatterns.com/index.php/article/articleview/30/1/1) et redécorée (http://phppatterns.com/index.php/article/articleview/92/1/1)
- (anglais) Approche Delphi (http://www.castle-cadenza.demon.co.uk/decorate.htm)
- (anglais) Article Trois approches java pour "*décorer*" votre code (http://www.javaworld.net/javaworld/jw-04-2004/jw-0412-decorator.html)" par Michael Feldman
- (anglais) Article "Utiliser le patron "*décorateur*" (http://www.onjava.com/pub/a/onjava/2003/02/05/decorator.html)" par Budi Kurniawan

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.

Chaîne de responsabilité

Le patron de conception **Chaîne de responsabilité** permet à un nombre quelconque de classes d'essayer de répondre à une requête sans connaître les possibilités des autres classes sur cette requête. Cela permet de diminuer le couplage entre objets. Le seul lien commun entre ces objets étant cette requête qui passe d'un objet à l'autre jusqu'à ce que l'un des objets puisse répondre.

Utilisation

Dès lors qu'une information doit recevoir plusieurs traitements, ou juste être transmise entre différents objets.

Variante

Une variante de ce patron de conception est un arbre de responsabilité, où chaque nœud de traitement transmet l'objet non plus à un seul autre nœud mais à plusieurs nœuds (exemple : un interpréteur de document XML).

Exemples

C++

L'exemple ci-dessous présente un système de journalisation (*log* en anglais). Lors de la réception d'un message, ce dernier va passer d'un Logger à l'autre, déclenchant ou non le traitement associé.

```
#include <iostream>
#include <string>
using namespace std;

class Logger
{
protected:
    int level;
    Logger* next;

public:
    enum
    {
        ERR,
        NOTICE,
        DEBUG
    };

    Logger* setNext(Logger* next)
    {
        this->next = next;
        return (this->next);
    }

    void message(string msg, int priority)
    {
        if (priority <= this->level)
            this->writeMessage(msg);
        if (this->next != NULL)
            this->next->message(msg, priority);
    }

    virtual void writeMessage(string msg) = 0;
};

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

    void writeMessage(string msg)
    {
        cout << "Message de debug : " << msg << endl;
    }
};

class EmailLogger : public Logger
{
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.
```

Commande

Commande est un patron de conception de type comportemental qui encapsule la notion d'*invocation*. Il permet de séparer complètement le code initiateur de l'action, du code de l'action elle-même. Ce patron de conception est souvent utilisé dans les interfaces graphiques où, par exemple, un *item* de menu peut être connecté à différentes Commandes de façons à ce que l'objet d'item de menu n'ait pas besoin de connaître les détails de l'action effectuée par la Commande.

À utiliser lorsque : il y a prolifération de méthodes similaires, et que le code de l'interface devient difficile à maintenir.

Symptômes:

- Les objets possèdent trop de méthodes publiques à l'usage d'autres objets.
- L'interface est inexploitable et on la modifie tout le temps.
- Les noms des méthodes deviennent de longues périphrases.

Un **objet Commande** sert à communiquer une action à effectuer, ainsi que les arguments requis. L'objet est envoyé à une seule méthode dans une classe, qui traite les Commandes du type requis. L'objet est libre d'implémenter le traitement de la Commande par un *switch*, ou un appel à d'autres méthodes (notamment des méthodes surchargées dans les sous-classes). Cela permet d'apporter des modifications aux Commandes définies simplement dans la définition de la Commande, et non dans chaque classe qui utilise la Commande.

Diagramme de classes

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

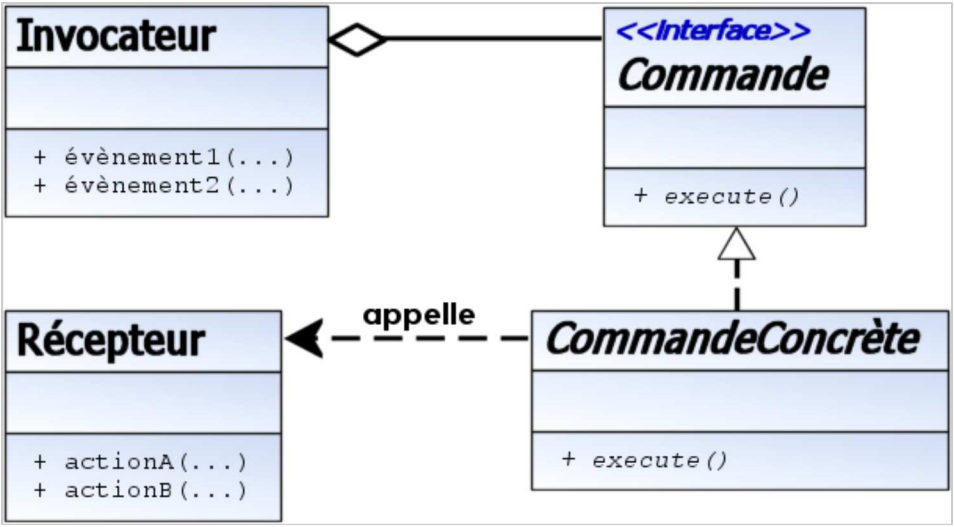


Diagramme de classes UML du patron de conception Commande

Utilisations

Ce patron de conception peut être utilisé pour implémenter divers comportements :

Défaire sur plusieurs niveaux
: Les actions de l'utilisateur sont enregistrées par empilement de commandes. Pour les défaire, il suffit de dépiler les dernières commandes et d'appeler leur méthode `undo()` pour annuler chaque commande.

Comportement transactionnel
: La méthode d'annulation est appelée `rollback()` et permet de revenir en arrière si quelque chose se passe mal au cours d'une transaction (un ensemble de commandes). Exemples : installateurs de programmes, modification de base de données.

Barre de progression
: Si chaque Commande possède une méthode d'estimation de durée, il est possible de représenter la progression de l'exécution d'un ensemble de tâches (Commandes).

Menu et boutons (interface graphique)
: En Swing et Delphi, un objet Action est une Commande à laquelle on peut associer un raccourci clavier, une icône, un texte d'info-bulle ...

Wizards
: Pour implémenter les boîtes de dialogue de type Wizard, une instance de Commande est créée. Chaque fois que l'utilisateur passe à la page suivante avec le bouton "Suivant" ("*Next*" en anglais), les valeurs entrées sont enregistrées dans la Commande. Le bouton "Terminer" ("*Finish*" en anglais) provoque l'exécution de la Commande.

Ensemble de threads (*ThreadPool* en anglais)
: Un ensemble de threads exécute des tâches (Commandes) stockées dans une file.

Enregistrement de macros

Chaque action de l'utilisateur peut être enregistrée sous la forme d'une séquence de Commande qui peut être rejouée par la suite. Pour enregistrer les macros sous la forme de scripts, chaque commande possède une méthode `toScript()` pour générer le script correspondant.

Exemple

Java

Considérons un interrupteur simple (*switch* en anglais). Dans cet exemple, on configure le switch avec deux commandes : une pour allumer la lumière, et une pour l'éteindre.

L'avantage de cette implémentation particulière du patron Commande est que l'interrupteur peut être utilisé avec n'importe quel périphérique, pas seulement une lampe. Dans l'exemple suivant l'interrupteur allume et éteint une lampe, mais le constructeur accepte toute classe dérivée de Command comme double paramètre. On peut, par exemple, configurer le switch pour démarrer et arrêter un moteur.

```
/* Invocateur */
public class Switch
{
    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd,Command flipDownCmd)
    {
        this.flipUpCommand=flipUpCmd;
        this.flipDownCommand=flipDownCmd;
    }

    public void flipUp()
    {
        flipUpCommand.execute();
    }

    public void flipDown()
    {
        flipDownCommand.execute();
    }
}

/* Récepteur */
public class Light
{
    public Light() {  }

    public void turnOn()
    {
        System.out.println("The light is on");
    }

    public void turnOff()
    {
        System.out.println("The light is off");
    }
}

/* Commande */
public interface Command
{
    void execute();
}


/* Commande concrète pour allumer la lumière */
public class TurnOnCommand implements Command
{
    private Light theLight;

    public TurnOnCommand(Light light)
    {
        this.theLight=light;
    }

    public void execute()
    {
        theLight.turnOn();
    }
}

/* Commande concrète pour éteindre la lumière */
public class TurnOffCommand implements Command
```

```
{
    private Light theLight;

    public TurnOffCommand(Light light)
    {
        this.theLight=light;
    }

    public void execute()
    {
        theLight.turnOff();
    }
}

/* Classe de test */
public class TestCommand
{
    public static void main(String[] args)
    {
        Light lamp = new Light();
        Command switchUp=new TurnOnCommand(lamp );
        Command switchDown=new TurnOffCommand(lamp );

        Switch s=new Switch(switchUp,switchDown);

        s.flipUp();
        s.flipDown();
    }
}
```

Perl

```
# exemple de style "switch":

sub doCommand {
  my $me = shift;
  my $cmd = shift; $cmd->isa('BleahCommand') or die;
  my $instr = $cmd->getInstructionCode();
  if($instr eq 'PUT') {
    # PUT logic here
  } elsif($instr eq 'GET') {
    # GET logic here
  }
  # etc
}

# exemple de style "appel de méthode":

sub doCommand {
  my $me = shift;
  my $cmd = shift; $cmd->isa('BleahCommand') or die;
  my $instr = $cmd->getInstructionCode();
  my $func = "process_" . $instr;
  return undef unless defined &$func;
  return $func->($cmd, @_);
}

# exemple de style "sous-classe".
# on suppose que %commandHandlers contient une liste de pointeurs d'objets.

sub doCommand {
  my $me = shift;
  my $cmd = shift; $cmd->isa('BleahCommand') or die;
  my $insr = $cmd->getInstructionCode();
  my $objectRef = $commandHandlers{$instr};
  return $objectRef ? $objectRef->handleCommand($cmd, @_) : undef;
}
```

Comme Perl dispose d'un *AUTOLOAD*, le principe pourrait être émulé. Si un *package* voulait effectuer un ensemble de commandes arbitrairement grand, il pourrait recenser toutes les méthodes *undefined* grâce à *AUTOLOAD*, puis tenter de les répartir (ce qui suppose que *%commandHandlers* contient une table de pointeurs, dont les clés sont les noms des méthodes):

```
sub AUTOLOAD {
  my $me = shift;
  (my $methodName) = $AUTOLOAD m/.*::(\w+)$/;
  return if $methodName eq 'DESTROY';
  my $objectRef = $commandHandlers{$methodName};
  return $objectRef ? $objectRef->handleCommand($methodName, @_) : undef;
}
```

Cela convertit les appels aux différentes méthodes dans l'objet courant, en appels à une méthode *handleCommand* dans différents objets.
Cet exemple utilise Perl pour adapter un patron de conception à base d'objets Commandes, dans une interface qui en est dépourvue.

Interpréteur

Le patron de conception **Interpréteur** est utilisé pour des logiciels ayant besoin d'un langage afin de décrire les opérations qu'ils peuvent réaliser (exemple : SQL pour interroger une base de données).

Le modèle de conception **Interpréteur** définit la grammaire de ce langage et utilise celle-ci pour interpréter des états dans ce langage.

Ce patron de conception est très utile dans deux cas:

1. lorsque le logiciel doit analyser/interpréter une chaîne algébrique. C'est un cas assez évident où le logiciel doit exécuter des opérations en fonction d'une équation (dessiner la courbe d'une fonction par exemple),
2. lorsque le logiciel doit produire différents types de données comme résultat. Ce cas est moins évident, mais l'interpréteur y est très utile. Prenez l'exemple d'un logiciel capable d'afficher des données dans n'importe quel ordre, en les triant ou pas, etc.

Ce patron définit comment interpréter les éléments du langage. Dans ce patron de conception, il y a une classe par symbole terminal et non-terminal du langage à interpréter. L'arbre de syntaxe du langage est représenté par une instance du patron de conception Objet composite.

Diagramme de classes

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

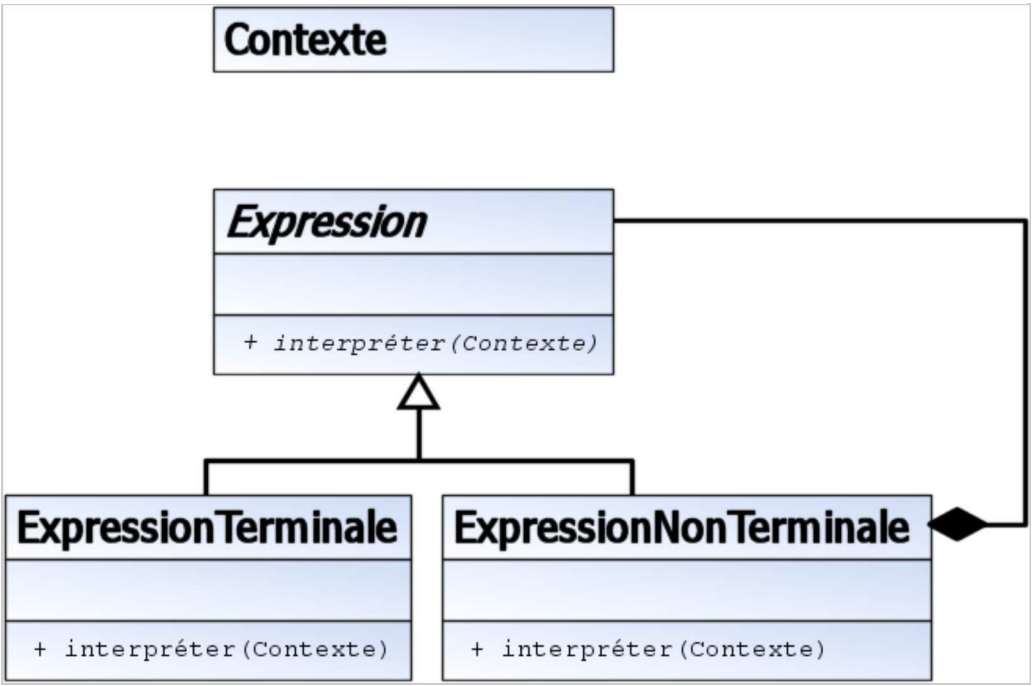


Diagramme de classes UML du patron de conception Interpréteur

Exemples

Java

L'exemple Java suivant montre comment interpréter un langage spécialisé, tel que les expressions en notation polonaise inversée. Dans ce langage, on donne les opérandes avant l'opérateur.

```
import java.util.*;

interface Expression
{
    public void interpret(Stack<Integer> s);
}

class TerminalExpression_Number implements Expression
{
    private int number;

    public TerminalExpression_Number(int number)
    { this.number = number; }
```

```
    public void interpret(Stack<Integer> s)
    { s.push(number); }
}

class TerminalExpression_Plus implements Expression
{
    public void interpret(Stack<Integer> s)
    { s.push( s.pop() + s.pop() ); }
}

class TerminalExpression_Minus implements Expression
{
    public void interpret(Stack<Integer> s)
    { s.push( - s.pop() + s.pop() ); }
}

class Parser
{
    private ArrayList<Expression> parseTree =
        new ArrayList<Expression>(); // only one NonTerminal Expression here

    public Parser(String s)
    {
        for (String token : s.split(" "))
        {
                if (token.equals("+"))
                    parseTree.add( new TerminalExpression_Plus() );
            else if (token.equals("-"))
                    parseTree.add( new TerminalExpression_Minus() );
            // ...
            else
                    parseTree.add( new TerminalExpression_Number(Integer.parseInt(token)) );
        }
    }

    public int evaluate()
    {
        Stack<Integer> context = new Stack<Integer>();
        for (Expression e : parseTree) e.interpret(context);
        return context.pop();
    }
}

class InterpreterExample
{
    public static void main(String[] args)
    {
        String expression = "42 4 2 - +";
        Parser p = new Parser(expression);
        System.out.println("'" + expression +"' equals " + p.evaluate());
    }
}
```

Ce programme affiche :

```
'42 4 2 - +' equals 44
```

Itérateur

L'**itérateur** est un patron de conception comportemental.

Un itérateur est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc.). Un synonyme d'itérateur est curseur, notamment dans le contexte des bases de données.

Description

Un itérateur ressemble à un pointeur disposant essentiellement de deux primitives : *accéder* à l'élément pointé en cours (dans le conteneur), et *se déplacer* pour pointer vers l'élément suivant. En sus, il faut pouvoir créer un itérateur pointant sur le premier élément ; ainsi que déterminer à tout moment si l'itérateur a épuisé la totalité des éléments du conteneur. Diverses implémentations peuvent également offrir des comportements supplémentaires.

Le but d'un itérateur est de permettre à son utilisateur de *parcourir* le conteneur, c'est-à-dire d'accéder séquentiellement à tous ses éléments pour leur appliquer un traitement, tout en isolant l'utilisateur de la structure interne du conteneur, potentiellement complexe. Ainsi, le conteneur peut stocker les éléments de la façon qu'il veut, tout en permettant à l'utilisateur de le traiter comme une simple liste. Le plus souvent l'itérateur est conçu en même temps que la classe-conteneur qu'il devra parcourir, et ce sera le conteneur lui-même qui créera et distribuera les itérateurs pour accéder à ses éléments.

Différences avec l'indexation

Dans les langages procéduraux on utilise souvent un index dans une simple boucle, pour accéder séquentiellement à tous les éléments, notamment d'un tableau. Quoique cette approche reste possible en programmation objet pour certains conteneurs, l'utilisation des itérateurs a certains avantages :

- Un simple compteur dans une boucle n'est pas adapté à toutes les structures de données, en particulier :
  - celles qui n'ont pas de méthode d'accès à un élément quelconque,
  - celles dont l'accès à un élément quelconque est très lent (c'est le cas des listes chaînées et des arbres).
- Les itérateurs fournissent un moyen cohérent d'*itérer* sur toutes sortes de structures de données, rendant ainsi le code client plus lisible, réutilisable, et robuste même en cas de changement dans l'organisation de la structure de données.
- Une structure arborescente peut fournir différents types d'itérateurs retournant les nœuds de l'arbre dans un ordre différent : parcours récursif, parcours par niveau, ...
- Un itérateur peut implanter des restrictions additionnelles sur l'accès aux éléments, par exemple pour empêcher qu'un élément soit « sauté », ou qu'un même élément soit visité deux fois.
- Un itérateur peut *dans certains cas* permettre que le conteneur soit modifié, sans être invalidé pour autant. Par exemple, après qu'un itérateur s'est positionné derrière le premier élément, il est possible d'insérer d'autres éléments au début du conteneur avec des résultats prévisibles. Avec un index on aurait plus de problèmes, parce que la valeur de l'index devrait elle aussi être modifiée en conséquence.

Important : il est indispensable de bien consulter la documentation d'un itérateur pour savoir dans quels cas il est invalidé ou non.

La possibilité pour un conteneur de se voir modifié pendant une itération s'est imposée comme nécessaire dans la programmation objet moderne, où les relations entre objets et l'effet de certaines opérations peut devenir un casse-tête. En utilisant un tel itérateur « robuste », ces désagréments nous sont épargnés.

Utilisation d'un itérateur explicite

Dans un langage à objets comme le C#, un itérateur est un objet qui implémente l'interface IEnumerator.

```
interface IEnumerator
{
    void Reset();
    bool MoveNext();
    object Current { get; }
}
```

On utilise l'itérateur pour accéder aux valeurs disponibles.

```
IterateurTypique iterateur = new IterateurTypique ();

iterateur.Reset(); // optionnel : cet appel peut ne pas être effectué.
while(iterateur.MoveNext())
{
    Console.WriteLine(iterateur.Current);
}
```

Une des nombreuses implémentations de l'objet possibles peut ressembler à celle-ci.

```
class IterateurTypique : IEnumerator
{
    private string[] _chainesAParcourir = new string[] { "TF1", "France2", "FR3", "Canal+" };
    private int _positionCourante = -1;

    public void Reset()
    { _positionCourante = -1; }

    public bool MoveNext()
    {
        if( _positionCourante + 1 >= _chainesAParcourir.Length ) return false;

        _positionCourante +=1;
        return true;
    }

    public object Current
    {
        get { return _chainesAParcourir[_positionCourante]; }
    }
}
```

L'interface IEnumerable de C# permet le passage à un itérateur implicite.

```
interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

les tableaux, listes ou dictionnaires de C# sont des types dérivés de IEnumerable et possèdent une méthode GetEnumerator() qui appelle l'itérateur explicite.

l'instruction foreach du C# appelle cette méthode GetEnumerator() et procède à une itération explicite tout en cachant les détails de l'implémentation.

```
if(Television is IEnumerable)
{
    foreach(object chaine in Television)
    {
        Console.WriteLine(chaine);
    }
}
```

Itérateurs implicites

Des langages à objets comme Perl et Python fournissent un moyen « interne » d'itérer sur les éléments d'un conteneur sans introduire explicitement un itérateur. Cela est souvent implémenté par une structure de contrôle *for-each*, comme dans les exemples suivants :

```
# Tcl: itérateur implicite
foreach val $list {
    puts stdout $val
}
```

```
# Perl: itérateur implicite
foreach $val (@list) {
    print "$val\n";
}
```

```
# Python, itérateur implicite
for Value in List:
    print Value
```

```
// PHP, itérateur implicite
foreach ($list as $value)
    print $value;
```

```
// Java, J2SE 5.0, itérateur implicite
for (Value v : list)
```

```
    System.out.print(v);
```

```
// C#, itérateur implicite
foreach (object obj in list)
    Console.WriteLine(obj );

// C#, itérateur explicite avec un yield
foreach (object obj in IndicesPairs() )
    Console.WriteLine(obj);

// ou IndicesPairs() est une méthode
IEnumerable IndicesPairs()
{
    for(int i=0; i<tableau.Length; i++)
        if(i%2==0)
            yield return tableau[i];
}
```

```
# Ruby, itérateur de bloc, (yield)
list.each do |value|
    puts value
end

# ou each est une méthode de Array tel que :
def each
    for i in 0...size
        yield(self[i])
    end
end
```

Attention, en Javascript, on itère pas directement sur les objets mais sur leur nom

```
// Javascript, itérateur implicite
for(nom in Object)
{
    var valeur =  Object[nom];
    alert(Value+" = "+valeur ]);
}
```

Le langage C++ dispose également de la fonction template `std::for_each()` qui permet des itérations implicites similaires, mais requiert toujours de fournir des objets itérateurs en paramètres d'entrée.

***Remarque** : le Ruby (ainsi que le C# à partir de sa version 2.0), offre via yield un outil spécifique pour construire des itérateurs.*

Médiateur

Le patron de conception **Médiateur** fournit une interface unifiée pour un ensemble d'interfaces d'un sous-système. Il est utilisé pour réduire les dépendances entre plusieurs classes.

Lorsqu'un logiciel est composé de plusieurs classes, les traitements et les données sont répartis entre toutes ces classes. Plus il y a de classes, plus le problème de communication entre celles-ci peut devenir complexe. En effet, plus les classes dépendent des méthodes des autres classes plus l'architecture devient complexe. Cela ayant des impacts sur la lisibilité du code et sa maintenabilité dans le temps.

Le modèle de conception **Médiateur** résout ce problème. Pour ce faire, le Médiateur est la seule classe ayant connaissance des interfaces des autres classes. Lorsqu'une classe désire interagir avec une autre, elle doit passer par le médiateur qui se chargera de transmettre l'information à la ou les classes concernées.

Exemples

Java

```
// Interface Collègue
interface Command
{
    void execute();
}

// Médiateur concret
class Mediator
{
    BtnView btnView;
    BtnSearch btnSearch;
    BtnBook btnBook;
    LblDisplay show;

    void registerView(BtnView v)
    {
        btnView = v;
    }

    void registerSearch(BtnSearch s)
    {
        btnSearch = s;
    }

    void registerBook(BtnBook b)
    {
        btnBook = b;
    }

    void registerDisplay(LblDisplay d)
    {
        show = d;
    }

    void book()
    {
        btnBook.setEnabled(false);
        btnView.setEnabled(true);
        btnSearch.setEnabled(true);
        show.setText("booking...");
    }

    void view()
    {
        btnView.setEnabled(false);
        btnSearch.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("viewing...");
    }

    void search()
    {
        btnSearch.setEnabled(false);
        btnView.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("searching...");
    }
}

// Un collègue concret
class BtnView extends JButton implements Command
{
```

```
    Mediator med;

    BtnView(ActionListener al, Mediator m)
    {
        super("View");
        addActionListener(al);
        med = m;
        med.registerView(this);
    }

    public void execute()
    {
        med.view();
    }
}

// Un collègue concret
class BtnSearch extends JButton implements Command
{
    Mediator med;

    BtnSearch(ActionListener al, Mediator m)
    {
        super("Search");
        addActionListener(al);
        med = m;
        med.registerSearch(this);
    }

    public void execute()
    {
        med.search();
    }
}

// Un collègue concret
class BtnBook extends JButton implements Command
{
    Mediator med;

    BtnBook(ActionListener al, Mediator m)
    {
        super("Book");
        addActionListener(al);
        med = m;
        med.registerBook(this);
    }

    public void execute()
    {
        med.book();
    }
}

class LblDisplay extends JLabel
{
    Mediator med;

    LblDisplay(Mediator m)
    {
        super("Just start...");
        med = m;
        med.registerDisplay(this);
        setFont(new Font("Arial", Font.BOLD, 24));
    }
}

class MediatorDemo extends JFrame implements ActionListener
{
    Mediator med = new Mediator();

    MediatorDemo()
    {
        JPanel p = new JPanel();
        p.add(new BtnView(this, med));
        p.add(new BtnBook(this, med));
        p.add(new BtnSearch(this, med));
        getContentPane().add(new LblDisplay(med), "North");
        getContentPane().add(p, "South");
        setSize(400, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent ae)
    {
        Command comd = (Command) ae.getSource();
        comd.execute();
```

```
    }

    public static void main(String[] args)
    {
        new MediatorDemo();
    }
}
```

Collègue
Définit l'interface de communication entre objets Collègues.

MédiateurConcret
Implémente l'interface Médiateur et coordonne la communication entre les objets Collègues. Il connaît tous les objets Collègues et comment ils communiquent.

CollègueConcret
Communique avec les autres Collègues à travers son Médiateur.

Mémento

Le **patron mémento** est un patron de conception qui fournit la manière de renvoyer un objet à un état précédent (retour arrière) sans violer le principe d'encapsulation.

Le mémento est utilisé par deux objets : le créateur et le gardien. Le créateur est un objet ayant un état interne (état à sauvegarder). Le gardien agira sur le créateur, tout en conservant la possibilité de revenir en arrière. Le gardien demande alors au créateur l'objet mémento pour enregistrer son état actuel. Il effectue l'opération (ou séquence d'opérations) souhaitée. Afin de permettre le retour arrière dans l'état d'avant les opérations, le mémento est retourné au créateur. L'objet mémento même est opaque (le gardien ne peut, ou ne devrait pas, le modifier). Lors de l'utilisation de ce patron, une attention toute particulière doit être prise si le créateur modifie d'autres objets ou ressources : Le patron mémento n'opère que sur un seul objet.

Il faut souligner que le fait de sauvegarder l'état interne de l'objet créateur doit s'effectuer sans casser le principe d'encapsulation. Cela n'est pas toujours possible (exemple : SmallTalk ne le permet pas de façon directe).

Des exemples classiques du patron mémento incluent le générateur de nombres pseudo-aléatoires, la machine à états finis, la fonction "Annulation" / "Undo".

Diagramme de classes

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

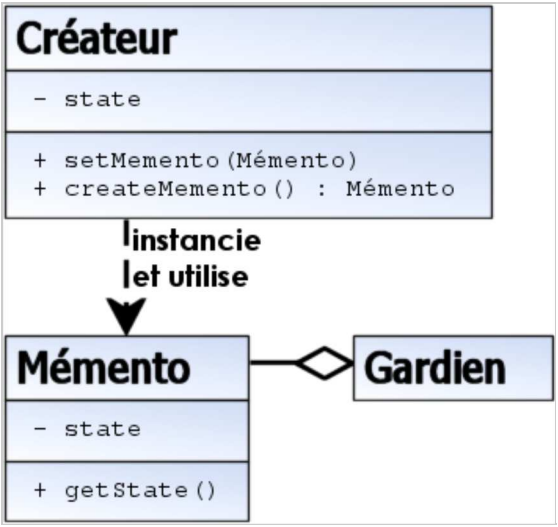


Diagramme de classes UML du patron de conception Mémento

Exemples

Java

Cet exemple illustre l'usage du pattern Mémento pour réaliser une commande de type annuler.

```
import java.util.*;

class Originator
{
    private String state;

    public void set(String state)
    {
        System.out.println("Originator : état affecté à : "+state);
        this.state = state;
    }

    public Object saveToMemento()
    {
        System.out.println("Originator : sauvegardé dans le mémento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Object m)
    {
        if (m instanceof Memento)
```

```
        {
            Memento memento = (Memento)m;
            state = memento.getSavedState();
            System.out.println("Originator : État après restauration : "+state);
        }
    }

    private static class Memento
    {
        private String state;

        public Memento(String stateToSave) { state = stateToSave; }
        public String getSavedState() { return state; }
    }
}

class Caretaker
{
    private ArrayList savedStates = new ArrayList();

    public void addMemento(Object m) { savedStates.add(m); }
    public Object getMemento(int index) { return savedStates.get(index); }
}

class MementoExample
{
    public static void main(String[] args)
    {
        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State3");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State4");

        originator.restoreFromMemento( caretaker.getMemento(1) );
    }
}
```

Ce programme affiche :

```
Originator : état affecté à : State1
Originator : état affecté à : State2
Originator : sauvegarde dans le memento.
Originator : état affecté à : State3
Originator : sauvegarde dans le memento.
Originator : état affecté à : State4
Originator : État après restauration : State3
```

Cet exemple utilise la classe String pour l'état (state dans l'exemple), or cette classe est un type immuable en Java. Lorsque l'état n'est pas un objet immuable, il devra être cloné avant d'être ajouté dans le memento.

```
private Memento(State state)
{
    // state doit être cloné avant d'être ajouté au memento,
    // sinon plusieurs memento retournerons tous une référence sur le même objet
    this.mementoState = state.clone();
}
```

Observateur

Le patron de conception **observateur**/**observable** est utilisé en programmation pour envoyer un signal à des modules qui jouent le rôle d'**observateur**. En cas de notification, les **observateurs** effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les "observables").

Utilité

La notion d'**observateur/observable** permet de découpler des modules de façon à réduire les dépendances aux seuls phénomènes observés.

Utilisation

Dès que l'on a besoin de gérer des événements, quand une classe déclenche l'exécution d'une ou plusieurs autres.

Dans une interface graphique utilisant MVC (Modèle-Vue-Contrôleur), le patron Observateur est utilisé pour associer Modèle et Vue.

Par exemple, en Java Swing, le modèle est censé notifier la vue de toute modification en utilisant PropertyChangeNotification. Les Java beans sont les observés, les éléments de la vue sont les observateurs. Tout changement dans le modèle est alors visible sur l'interface graphique.

Illustration

Prenons comme exemple une classe qui produit des signaux (données observables), visualisés à travers des panneaux (**observateurs**) d'une interface graphique. On souhaite que la mise à jour d'un signal modifie le panneau qui l'affiche. Afin d'éviter l'utilisation de threads ou encore d'inclure la notion de panneau dans les signaux il suffit d'utiliser le patron de conception **observateur/observable**.

Le principe est que chaque classe observable contienne une liste d'**observateurs**, ainsi à l'aide d'une méthode de notification l'ensemble des **observateurs** sont prévenus. La classe observée hérite de "Observable" qui gère la liste des **observateurs**. La classe **Observateur** est quant à elle purement abstraite, la fonction de mise à jour ne pouvant être définie que par une classe spécialisée.

Exemples

Java

L'exemple ci-dessous montre comment utiliser l'API du langage Java qui propose des interfaces et des objets abstraits liées à ce patron de conception.

- On crée une classe qui étend *java.util.Observable* et dont la méthode de mise à jour des données *setData* lance une notification des observateurs (1) :

```
class Signal extends Observable
{
    void setData(byte[] lbData)
    {
        setChanged(); // Positionne son indicateur de changement
        notifyObservers(); // (1) notification
    }
}
```

- On crée le panneau d'affichage qui implémente l'interface *java.util.Observer*. Avec une méthode d'initialisation (2), on lui transmet le signal à observer (2). Lorsque le signal notifie une mise à jour, le panneau est redessiné (3).

```
class JPanelSignal extends JPanel implements Observer
{
    void init(Signal lSigAObserver)
    {
        lSigAObserver.addObserver(this); // (2) ajout d'observeur
    }

    void update(Observable observable, Object objectConcerne)
    {
        repaint();  // (3) traitement de l'observation
    }
}
```

C++

Dans cet exemple en C++, on veut afficher les événements qui se produisent dans une classe Exemple.

```
#include <string>
#include <list>
#include <iostream>
using namespace std;

class IObserver
{
public:
    virtual void update(string data) = 0;
};

class Observable
{
private:
    list<IObserver*> list_observers;

public:
    void notify(string data)
    {
        // Notifier tout les observers
        for ( list<IObserver*>::iterator it = this->list_observers.begin() ;
              it != this->list_observers.end() ;
              ++it )
            (*it)->update(data);
    }

    void addObserver(IObserver* observer)
    {
        // Ajouter un observer a la liste
        this->list_observers.push_back(observer);
    }
};

class Display : public IObserver
{
    void update(string data)
    {
        cout << "Événement : " << data << endl;
    }
};

class Exemple : public Observable
{
public:
    void message(string message)
    {
        // Lancer un événement lors de la réception d'un message
        this->notify(message);
    }
};

int main()
{
    Display display;
    Exemple exemple;

    // On veut que "Display" soit prévenu à chaque réception d'un message dans "Exemple"
    exemple.addObserver(&display);

    // On envoie un message a Exemple
    exemple.message("réception d'un message"); // Sera affiché par Display

    return (0);
}
```

C#

Tout comme Itérateur, Observateur est implémenté en C# par l'intermédiaire d'un mot clé : `event`. La syntaxe a été simplifiée pour l'abonnement ou appel d'une méthode sur levée d'un événement. Un événement possède une signature : le type de la méthode que doit lever l'évènement. Dans cet exemple c'est EventHandler.

```
event EventHandler observable;
```

La signature du type délégué EventHandler est :

```
void (object emetteur, EventArgs argument)
```

```
using System;

///<summary> un observateur </summary>
class Kevin
{
    public void Reception(object sender, EventArgs e)
    {
        Console.WriteLine("Kevin a reçu : {1} de : {0}", sender.ToString(), e.ToString());
    }
}

class Program
{
    ///<summary> la liste d'abonnés </summary>
    static event EventHandler observable;

    static void Main()
    {
        var kevin = new Kevin();

        // enregistrement de Kevin dans la liste d'abonnés
        observable += new EventHandler(kevin.Reception);

        // si la liste n'est pas vide, prévenir les abonnés
        if(observable!=null)
            observable(AppDomain.CurrentDomain, new BeerEventArgs() { Bottles = 0 });
    }
}

/// <summary> que du fonctionnel </summary>
class BeerEventArgs : EventArgs
{
    public uint Bottles;
    public override string ToString()
    {
        return string.Format("{0} bottle{1}",
            (Bottles > 0) ? Bottles.ToString() : "no more",
            (Bottles > 1) ? "s" : string.Empty);
    }
}
```

État

La technique de l'**État** est un patron de conception comportemental utilisé en génie logiciel. Ce patron de conception est utilisé entre autres lorsqu'il est souhaité pouvoir changer le comportement de l'**État** d'un objet sans pour autant en changer l'instance.

Principe Général

La classe censée changer d'état a un lien vers une classe abstraite "État". Cette classe abstraite "État" définit les différentes méthodes qui seront à redéfinir dans les implémentations. Dans chaque classe dérivée d'État, l'appel à la méthode X pourra avoir un comportement différent.

La classe pouvant changer d'état appellera les services de sa classe d'état dont l'instance change quand le comportement de notre classe change. De plus l'instance de la classe pouvant changer d'état peut être passée en paramètre à la méthode X de sa classe d'état. Ceci permet de changer l'état de la classe pendant l'exécution de la méthode X en instanciant un nouvel état.

Ce patron permet donc à la classe de passer d'un état à l'autre de telle façon que cette dernière apparaît changer de type dynamiquement (sans changer d'instance).

Exemple : Un programme de dessin utilise une interface abstraite pour représenter un outil. Chaque instance de ses classes dérivées concrètes représente un type d'outil différent. Quand l'utilisateur change d'outil, une nouvelle instance de la classe dérivée associée est créée.

Diagramme de classes

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

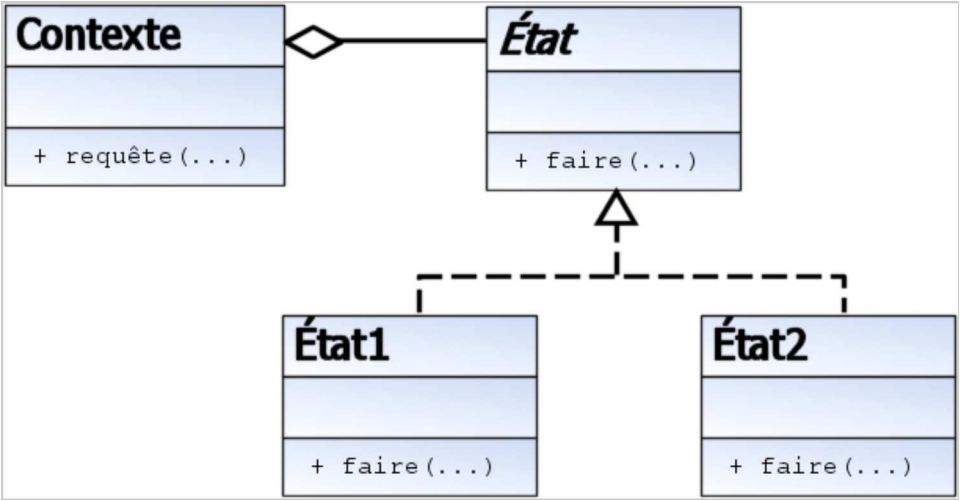


Diagramme de classes UML du patron de conception État

La classe Contexte utilise différentes instances des classes concrètes dérivées de la classe abstraite État afin de représenter ses différents états. Chaque état concret traite la requête de manière différente.

Exemples

C++

En reprenant l'exemple du programme de dessin :

```
class AbstractTool
{
public:
    virtual void MoveTo(const Point& inP) = 0;
    virtual void MouseDown(const Point& inP) = 0;
    virtual void MouseUp(const Point& inP) = 0;
};
```

Chaque outil a besoin de gérer les évènement de souris : déplacement, bouton enfoncé, bouton relâché. L'outil plume est alors le suivant :

```
class PenTool : public AbstractTool
```

```
{
public:
    PenTool() : mMouseIsDown(false) {}

    virtual void MoveTo(const Point& inP)
    {
        if(mMouseIsDown)
        {
            DrawLine(mLastP, inP);
        }
        mLastP = inP;
    }

    virtual void MouseDown(const Point& inP)
    {
        mMouseIsDown = true;
        mLastP = inP;
    }

    virtual void MouseUp(const Point& inP)
    {
        mMouseIsDown = false;
    }

private:
    bool mMouseIsDown;
    Point mLastP;
};

class SelectionTool : public AbstractTool
{
public:
    SelectionTool() : mMouseIsDown(false) {}

    virtual void MoveTo(const Point& inP)
    {
        if(mMouseIsDown)
        {
            mSelection.Set(mLastP, inP);
        }
    }

    virtual void MouseDown(const Point& inP)
    {
        mMouseIsDown = true;
        mLastP = inP;
        mSelection.Set(mLastP, inP);
    }

    virtual void MouseUp(const Point& inP)
    {
        mMouseIsDown = false;
    }

private:
    bool mMouseIsDown;
    Point mLastP;
    Rectangle mSelection;
};
```

Une classe utilisant le patron d'état ci-dessus :

```
class DrawingController
{
public:
    DrawingController() { selectPenTool(); } // Démarrer avec un outil.

    void MoveTo(const Point& inP)    { currentTool->MoveTo(inP);    }
    void MouseDown(const Point& inP) { currentTool->MouseDown(inP); }
    void MouseUp(const Point& inP)   { currentTool->MouseUp(inP);   }

    void selectPenTool()
    {
        currentTool.reset(new PenTool);
    }

    void selectSelectionTool()
    {
        currentTool.reset(new SelectionTool);
    }

private:
    std::auto_ptr<AbstractTool> currentTool;
};
```

L'état de l'outil de dessin est complètement représenté par une instance de la classe AbstractTool. Cela facilite l'ajout de nouveaux outils et conserve leur comportement dans les sous-classes de la classe AbstractTool.

Contre-exemple utilisant switch

Le patron de conception peut être utilisé pour remplacer des instructions `switch` et `if`, qui peuvent être difficile à maintenir et moins typées.

L'exemple suivant est similaire au précédent, mais ajouter un nouvel outil dans cette version est beaucoup plus difficile.

```
class Tool
{
public:
    Tool() : mMouseIsDown(false) {}

    virtual void MoveTo(const Point& inP);
    virtual void MouseDown(const Point& inP);
    virtual void MouseUp(const Point& inP);

private:
    enum Mode { Pen, Selection };
    Mode mMode;
    Point mLastP;
    bool mMouseIsDown;
    Rectangle mSelection;
};

void Tool::MoveTo(const Point& inP)
{
    switch(mMode)
    {
    case Pen:
        if (mMouseIsDown)
        {
            DrawLine(mLastP, inP);
        }
        mLastP = inP;
        break;

    case Selection:
        if (mMouseIsDown)
        {
            mSelection.Set(mLastP, inP);
        }
        break;

    default:
        throw std::exception();
    }
}

void Tool::MouseDown(const Point& inP)
{
    switch(mMode)
    {
    case Pen:
        mMouseIsDown = true;
        mLastP = inP;
        break;

    case Selection:
        mMouseIsDown = true;
        mLastP = inP;
        mSelection.Set(mLastP, inP);
        break;

    default:
        throw std::exception();
    }
}

void Tool::MouseUp(const Point& inP)
{
    mMouseIsDown = false;
}
```

Stratégie

Le **patron *stratégie*** est un patron de conception de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours de l'exécution selon certaines conditions, comme les stratégies utilisées en temps de guerre.

Le patron de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Le patron stratégie est prévu pour fournir des moyens de définir une famille d'algorithmes, encapsuler chacun comme objet, et les rendre interchangeables. Le patron stratégie laisse les algorithmes changer indépendamment des clients qui les emploient.

Utilisation

Dès lors qu'un objet peut effectuer plusieurs traitements différents, dépendant d'une variable ou d'un état.

Exemples

C++

```
#include <iostream>
using namespace std;

class IStrategie
{
public:
    virtual void execute() = 0;
};

class AlgorithmeA: public IStrategie
{
public:
    void execute()
    {
        cout << "Traitement A" << endl;
    }
};

class AlgorithmeB: public IStrategie
{
public:
    void execute()
    {
        cout << "Traitement B" << endl;
    }
};

class AlgorithmeC: public IStrategie
{
public:
    void execute()
    {
        cout << "Traitement C" << endl;
    }
};

class Element
{
private:
    IStrategie* strategie;

public:
    Element(IStrategie* strategie) : strategie(strategie)
    {
    }

    void execute()
    {
        this->strategie->execute();
    }
};

int main(int argc, char *argv[])
{
    AlgorithmeA algoA;
    AlgorithmeB algoB;
    AlgorithmeC algoC;

    Element elementA(&algoA);
    Element elementB(&algoB);
    Element elementC(&algoC);
```

```
    elementA.execute(); // L'élément A va effectuer le traitement A
    elementB.execute(); // L'élément B va effectuer le traitement B
    elementC.execute(); // L'élément C va effectuer le traitement C

    return (0);
}
```

C#

Des idées semblables amènent à une réalisation à l'aide d'interface.

L'objet qui doit avoir une stratégie adaptable à l'exécution implémente IStrategie : la même interface que d'autres objets. L'objet principal délègue l'exécution de la tâche à un autre objet membre qui implémente IStrategie.

L'objet membre étant déclaré dans la classe comme une interface, son implémentation importe peu, on peut donc changer de stratégie à l'exécution. Cette manière de faire se rapproche du Principe de l'injection de dépendance (inversion de contrôle).

```
using System;

/// <summary> La manière dont le grand général guidera ses troupes </summary>
interface IStrategie
{
    void MettreEnOeuvre();
}

/// <summary> Ce grand homme qui fera bientôt des choix décisifs </summary>
class SeigneurDeLaGuerre
{
    /// <summary> une stratégie générique </summary>
    IStrategie _strategie;

    /// <summary> comment changer de stratégie </summary>
    public IStrategie Strategie
    {
        set { _strategie = value; }
    }

    /// <summary> délégation de la tâche </summary>
    public void PrendreLaVille()
    {
        _strategie.MettreEnOeuvre();
    }
}

class DéfoncerLePontLevisDeFace : IStrategie
{
    public void MettreEnOeuvre()
    {
        Console.WriteLine("Prendre la ville de face en défonçant le pont levis.");
    }
}

class PasserParLaFaceNord : IStrategie
{
    public void MettreEnOeuvre()
    {
        Console.WriteLine("Prendre la ville en escaladant la muraille nord.");
    }
}

class AttendreQueLaVilleSeRende : IStrategie
{
    public void MettreEnOeuvre()
    {
        Console.WriteLine("Attendre qu'il n'y ait plus rien à manger en ville "
            + "et que tout le monde meure de faim.");
    }
}

class SeMarierAvecLaCousineDuDuc : IStrategie
{
    public void MettreEnOeuvre()
    {
        Console.WriteLine("Organiser un mariage avec la cousine du Duc "
            + "alors qu'elle rejoint la ville de retour des baléares "
            + "et inviter toute la ville à une grande fête.");
    }
}

/// <summary> Différentes situations </summary>
enum Météo
```

```
{
    IlFaitBeau,
    IlYADuBrouillard,
    IlFaitTropChaudPourTravailler,
    IlPleut
}

class Program
{
    static void Main()
    {
        // notre acteur
        var kevin = new SeigneurDeLaGuerre();

        // les aléas du système
        var météo = (Météo)(new Random().Next(0, 3));

        // une liaison tardive
        switch (météo)
        {
            case Météo.IlFaitBeau:
                kevin.Strategie = new DéfoncerLePontLevisDeFace(); break;

            case Météo.IlYADuBrouillard:
                kevin.Strategie = new PasserParLaFaceNord(); break;

            case Météo.IlFaitTropChaudPourTravailler:
                kevin.Strategie = new AttendreQueLaVilleSeRende(); break;

            case Météo.IlPleut:
                kevin.Strategie = new SeMarierAvecLaCousineDuDuc(); break;

            default:
                throw new Exception("Nan finalement seigneur de la guerre c'est "
                    + "pas cool comme job : vous décidez d'aller élever "
                    + "des chèvres dans le Larzac.");
        }

        // une exécution aux petits oignons
        kevin.PrendreLaVille();
    }
}
```

Patron de méthode

La technique du **patron de méthode** est un patron de conception comportemental utilisé en génie logiciel.

Un patron de méthode définit le squelette d'un algorithme à l'aide d'opérations *abstraites* dont le comportement concret se trouvera dans les sous-classes, qui implémenteront ces opérations.

Cette technique, très répandue dans les classes abstraites, permet de:

- Fixer clairement des comportements standards qui devraient être partagés par toutes les sous-classes, même lorsque le détail des sous-opérations diffère.
- Factoriser du code qui serait redondant s'il se trouvait répété dans chaque sous-classe.

La technique du patron de méthode a ceci de particulier que c'est la méthode de la classe parent qui appelle des opérations n'existant que dans les sous-classes. C'est une pratique courante dans les classes abstraites, alors que d'habitude dans une hiérarchie de classes concrètes c'est le contraire : ce sont plutôt les méthodes des sous-classes qui appellent les méthodes de la super-classe comme morceau de leur propre comportement.

L'implémentation d'un patron de méthode est parfois appelée **méthode socle** parce qu'elle ancre solidement un comportement qui s'applique alors à toute la hiérarchie de classes par héritage. Pour s'assurer que ce comportement ne sera pas redéfini arbitrairement dans les sous-classes, on déclare la méthode socle **final** en Java, ou bien **non virtuelle** en C++.

Les méthodes servant de "briques de comportement" à la méthode *socle* devraient être déclarées **abstract** en Java, ou bien **virtuelles pures** en C++.

Diagramme UML

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

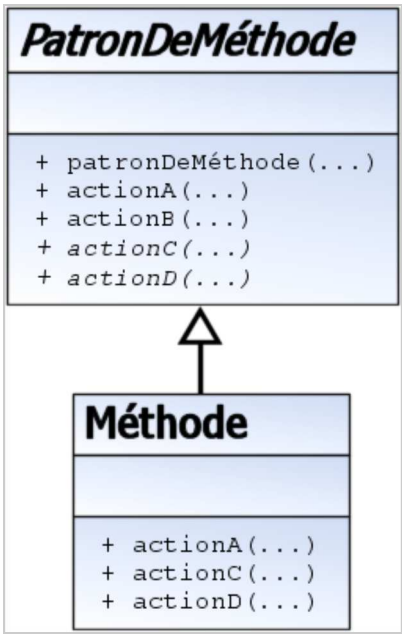


Diagramme de classes UML du patron de conception Patron de méthode

La méthode `patronDeMéthode()` appelle les méthodes abtraites ou concrètes. Les méthodes abstraites sont implémentées par les classes dérivées. Les méthodes concrètes peuvent être redéfinies par les classes dérivées (comportement différent de celui par défaut).

Exemples

Java

```
/**
 * Classe abstraite servant de base commune à divers
 * jeux de société où les joueurs jouent chacun leur tour.
 */
abstract class JeuDeSociété
{
```

```
    protected int nombreDeJoueurs;

    abstract void initialiserLeJeu();

    abstract void faireJouer(int joueur);

    abstract boolean partieTerminée();

    abstract void proclamerLeVainqueur();

    /* Une méthode socle : */
    final void jouerUnePartie(int nombreDeJoueurs)
    {
        this.nombreDeJoueurs = nombreDeJoueurs;
        initialiserLeJeu();

        // Premier joueur :
        int j = 0;
        while( ! partieTerminée() )
        {
            faireJouer( j );
            // Joueur suivant :
            j = (j + 1) % nombreDeJoueurs;
        }

        proclamerLeVainqueur();
    }
}
```

On peut maintenant dériver cette classe pour implanter divers jeux :

```
class Monopoly extends JeuDeSociété
{
    /* Implémentation concrète des méthodes nécessaires */

    void initialiserLeJeu()
    {
        // distribuer les billets, placer les pions, ...
    }

    void faireJouer(int joueur)
    {
        // lancer le dé, avancer, action selon la case d'arrivée...
    }

    boolean partieTerminée()
    {
        // il y a un joueur ruiné ou nombre de tours prédéfini écoulé, ...
    }

    void proclamerLeVainqueur()
    {
        // ...
    }

    /* Déclaration des composants spécifiques au jeu du Monopoly */

    // Pions, cases, cartes, billets, ...
}
```

```
class Echecs extends JeuDeSociété
{
    /* Implémentation concrète des méthodes nécessaires */

    void initialiserLeJeu()
    {
        // Placer les pions sur l'échiquier, ...
    }

    void faireJouer(int joueur)
    {
        // Choisir une pièce, l'avancer, prise ou promotion, ...
    }

    boolean partieTerminée()
    {
        // Échec et mat ou abandon, ...
    }

    void proclamerLeVainqueur()
    {
        // ...
```

```
    }
    /* Déclaration des composants spécifiques au jeu d'échecs */
    // Pions et échiquier, ...
}
```

La technique du patron de méthode fixe un cadre pour toutes les sous-classes. Cela implique certaines restrictions : dans l'exemple ci-dessus, on ne peut pas faire hériter une classe `JeuDuTarot` de la classe abstraite `JeuDeSociété`, parce que dans une partie de Tarot, l'ordre des joueurs n'est pas linéaire : il dépend du joueur qui vient de ramasser le pli.

On peut décider de ne **pas** déclarer la méthode socle comme *final* en Java (ou bien décider de la déclarer *virtual* en C++), afin de la rendre plus souple. Ainsi la classe `JeuDuTarot` pourrait parfaitement hériter de la classe `JeuDeSociété`, à condition de redéfinir la méthode `jouerUnePartie()` pour tenir compte des règles du Tarot. Mais cette pratique est critiquable.

Il est important de se poser la question dès l'écriture de la super-classe : *Les sous-classes auront-elles le droit de redéfinir les comportements fondamentaux codés dans la super-classe ?*. L'avantage est bien sûr une souplesse accrue. L'inconvénient peut être la perte de la cohérence interne de l'objet, si la surcharge des méthodes socles est mal conçue.

Pour reprendre l'exemple précédent, on pourrait mettre en place une méthode qui retourne le prochain joueur, qui serait implémentée différemment dans la classe `JeuDuTarot` et dans une classe d'un jeu où chaque joueur joue successivement.

Voir aussi

Patrons associés

- Fabrique
- Fabrique Abstraite
- Monteur

Visiteur

Un **visiteur** est le nom d'une des structures de patron de conception comportemental.

Le *visiteur* est une manière de séparer un algorithme d'une structure de données. Un visiteur possède une méthode par type d'objet traité. Pour ajouter un nouveau traitement, il suffit de créer une nouvelle classe dérivée de la classe Visiteur. On n'a donc pas besoin de modifier la structure des objets traités, contrairement à ce qu'il aurait été obligatoire de faire si on avait implémenté les traitements comme des méthodes de ces objets.

L'avantage du patron visiteur est qu'un visiteur peut avoir un état. Ce qui signifie que le traitement d'un type d'objet peut différer en fonction de traitements précédents. Par exemple, un visiteur affichant une structure arborescente peut présenter les nœuds de l'arbre de manière lisible en utilisant une indentation dont le niveau est stocké comme valeur d'état du visiteur.

Exemples

C++

Prenons une classe `ObjetPere`, de laquelle hériteront `Objet1, Objet2 et Objet3`, elles posséderont la méthode `accept(Visitor v)`.

```
void ObjetDeType1::accept( Visitor * v )
{
    v->visitObjetDeType1( this );
}
```

Créons la classe `Visitor`, dont hériteront `Visiteur1` et `Visiteur2`. Dans chacun de ces objets, on retrouvera une méthode `visiterObjet1(Objet1 a)`, `visiterObjet2(Objet2 b)` et `visiterObjet3(Objet3 c)`.

```
void MonVisiteur::visitObjetDeType1( ObjetDeType1 * objet )
{
    // Traitement d'un objet de type 1
}

void MonVisiteur::visitObjetDeType2( ObjetDeType2 * objet )
{
    // Traitement d'un objet de type 2
}

void MonVisiteur::visitObjetDeType3( ObjetDeType3 * objet )
{
    // Traitement d'un objet de type 3
}
```

Java

L'exemple suivant montre comment afficher un arbre de nœuds (les composants d'une voiture). Au lieu de créer des méthodes d'affichage pour chaque sous-classe (Wheel, Engine, Body, et Car), une seule classe est créée (CarElementPrintVisitor) pour afficher les éléments. Parce que les différentes sous-classes requièrent différentes actions pour s'afficher proprement, la classe CarElementPrintVisitor répartit l'action en fonction de la classe de l'argument qu'on lui passe.

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

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);
    }
}

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()
            };
    }
}

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");
    }
}

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");
    }
}

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

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

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

Lien externe

- français Article sur Développez.com (http://pcaboche.developpez.com/article/design-patterns/programmation-modulaire/?page=page_5)

Patrons GRASP

GRASP signifie ***G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns/**P**rinciples.*

Ces patrons de conception donnent des conseils généraux sur l'assignation de responsabilité aux classes et objets dans une application. Ils sont issus du bon sens de conception, intuitifs et s'appliquent de manière plus générale.

Une responsabilité est vue au sens conception (exemples : création, détention de l'information, ...) :

- elle est relative aux méthodes et données des classes,
- elle est assurée à l'aide d'une ou plusieurs méthodes,
- elle peut s'étendre sur plusieurs classes.

En UML, l'assignation de responsabilité peut être appliquée à la conception des diagrammes de collaboration.

Les patrons de conception GRASP sont les suivants :

Expert en information
: Affecter les responsabilités aux classes détenant les informations nécessaires.

Créateur
: Déterminer quelle classe a la responsabilité de créer des instances d'une autre classe.

Faible couplage
: Diminuer le couplage des classes afin de réduire leur inter-dépendances dans le but de faciliter la maintenance du code.

Forte cohésion
: Avoir des sous-classes terminales très spécialisées.

Contrôleur
: Affecter la responsabilité de réception et traitement de messages systèmes.

Polymorphisme
: Affecter un nouveau comportement à l'endroit de la hiérarchie de classes où il change.

Fabrication pure
: Créer des classes séparées pour des fonctionnalités génériques qui n'ont aucun rapport avec les classes du domaine applicatif.

Indirection
: Découpler des classes en utilisant une classe intermédiaire.

Protection
: Ne pas communiquer avec des classes inconnues.

Expert en information

Le principe de l'expert en information est d'assigner la responsabilité d'une requête à la classe qui détient les informations nécessaires. Il s'agit de créer la méthode là où les données se trouvent.

Il s'agit d'appliquer le principe « Celui qui sait le fait » ou de mettre les services avec les attributs.

Exemples

Bibliothèque

Dans un logiciel de gestion de bibliothèque, nous avons créé les classes `Bibliothèque` et `Catalogue`. Nous voulons ajouter une méthode retournant le livre en fonction de son numéro. À quelle classe assigner cette responsabilité ?

La classe susceptible de détenir les informations nécessaires est la classe `Catalogue`. On lui ajoute donc cette méthode de recherche de livre.

Facturation

Dans un logiciel de gestion de vente, nous avons les classes et attributs suivants :

`Facture`
: Contient un ensemble de produit facturé. Attribut : `client`,

`ProduitFacturé`
: Contient un lien vers la description d'un article. Attribut : `quantité`,

`Article`
: Description d'un article. Attributs : `nom`, `prix_unitaire` (TTC), `identifiant`.

À quelle(s) classe(s) assigner la responsabilité de calculer le total de la facture ?

La classe `ProduitFacturé` connaît l'article utilisé et la quantité commandée. On lui ajoute une méthode `sousTotal()` qui demandera le prix unitaire à l'`Article` et le multipliera par la quantité.

La classe `Facture` connaît la liste de tous les produits facturés. On lui ajoute une méthode `total()` qui effectue la somme des sous-totaux demandés aux différentes instances de la classe `ProduitFacturé` en appelant la méthode `sousTotal()`.

Cet exemple illustre comment une responsabilité peut être répartie sur plusieurs classes. Chaque classe est responsable de la partie qui la concerne.

Cet exemple démontre également que ce patron de conception favorise l'encapsulation : Une classe n'a pas besoin de connaître la structure interne de la classe `Facture` pour calculer le montant total dû.

Créateur

Le principe du créateur est d'assigner la responsabilité de la création des instances d'une classe A. Cette responsabilité est assignée à la classe B si au moins l'une des conditions suivantes est vraie :

- B contient des instances de A,
- B est une agrégation d'instances de A,
- B utilise des instances de A de manière détaillée,
- B possède des informations pour créer des instances de A.

Exemples

Bibliothèque

Dans un logiciel de gestion de bibliothèque, nous avons créé les classes Bibliothèque, Catalogue et Livre. Nous voulons ajouter une méthode de création de livres. À quelle classe assigner cette responsabilité ?

La classe dont l'utilisation est très liée à celle de Livre est la classe Catalogue. On lui ajoute donc cette méthode de création de livre.

Facturation

Dans un logiciel de gestion de vente, nous avons les classes et attributs suivants :

`Facture`
: Contient un ensemble de produit facturé. Attribut : `client`,

`ProduitFacturé`
: Contient un lien vers la description d'un article. Attribut : `quantité`,

`Article`
: Description d'un article. Attributs : `nom`, `prix_unitaire` (TTC), `identifiant`.

À quelle(s) classe(s) assigner la responsabilité d'ajouter un nouveau produit facturé (une nouvelle instance de la classe `ProduitFacturé`) ?

La classe `Facture` contient des instances de `ProduitFacturé`. Elle est donc responsable de leur création.

Faible couplage

Le couplage mesure la dépendance entre des classes ou des modules.

Le faible couplage favorise :

- la faible dépendance entre les classes,
- la réduction de l'impact des changements dans une classe,
- la réutilisation des classes ou modules.

Pour affaiblir le couplage, il faut :

- diminuer la quantité de paramètres passés entre les modules,
- éviter d'utiliser des variables globales (par exemple, si une mauvaise valeur est assignée, détecter la fonction/classe incorrecte est plus difficile), il vaut mieux passer les valeurs en paramètres.

Exemples

Facturation

Dans un logiciel de gestion de vente, nous avons les classes suivantes :

`Facture`
: Contient un ensemble de produit facturé et est associée à un mode de paiement,

`Paiement`
: Décrit un mode de paiement (espèces, chèque, carte bancaire, à crédit, ...),

`Client`
: Effectue les commandes.

On ajoute une méthode `payer()` à la classe `Client`. On étudie le couplage dans les deux cas suivants :

1. La méthode `payer()` crée une instance de `Paiement` et l'assigne à l'objet `Facture`.
2. La méthode `payer()` délègue l'action à la classe `Facture` qui crée une instance de `Paiement` et l'assigne.

Le couplage est plus faible dans le deuxième car la méthode `payer()` de la classe `Client` n'a pas besoin de savoir qu'il existe une classe `Paiement`, c'est à dire qu'elle **ne dépend pas** de l'existence ou non de cette classe.

Forte cohésion

La cohésion mesure la compréhensibilité des classes. Une classe doit avoir des responsabilités cohérentes, et ne doit pas avoir des responsabilités trop variées. Une classe ayant des responsabilités non cohérentes est difficile à comprendre et à maintenir.

La forte cohésion favorise :

- la compréhension de la classe,
- la maintenance de la classe,
- la réutilisation des classes ou modules.

Ce patron de conception peut être utilisé pour le faible couplage.

Contrôleur

Un contrôleur est une classe qui traite les messages systèmes. Certains messages systèmes sont générés par des sources externes à l'application en cours de conception.

Il est conseillé d'assigner cette responsabilité :

- soit à une classe représentant l'organisation globale ou le système global (contrôleur de façade),
- soit à une classe représentant un acteur du domaine (contrôleur de rôle),
- soit à une classe artificielle représentant un cas d'utilisation (contrôleur de cas d'utilisation).

Dans la cas des interfaces graphiques, en suggérant la création d'une classe contrôleur séparée, ce patron de conception encourage le faible couplage en réduisant les dépendances entre l'interface graphique et le modèle.

Exemples

Bibliothèque

Dans un logiciel de gestion de Bibliothèque, la responsabilité de traiter le message "Supprimer tel livre" peut être assigné :

- soit à la classe Bibliothèque (contrôleur de façade),
- soit à la classe Libraire (contrôleur de rôle),
- soit à la classe Suppression (contrôleur de cas d'utilisation).

Polymorphisme

Le polymorphisme est une propriété de la programmation objet où une classe peut définir un comportement par défaut (une méthode de classe), voire ne définir que la signature de la méthode (classe abstraite). Ce comportement pouvant être redéfini par des sous-classes afin de le modifier pour certains types d'objets.

Sans le polymorphisme, il serait nécessaire d'utiliser plusieurs tests (instructions *if* ou *switch* en général) dans la méthode afin de définir différents comportement en fonction du type d'objet. L'ajout d'un nouveau type oblige la modification de la méthode pour ajouter un test et un nouveau comportement.

Dans le cas du polymorphisme, l'ajout d'un nouveau type d'objet implique uniquement la création d'une nouvelle classe étendant la classe de base par un nouveau comportement. Le polymorphisme permet donc une meilleure extensibilité de l'application.

Le but du patron polymorphisme est d'assigner la responsabilité du changement de comportement au type (à la classe) concerné.

Fabrication pure

Certaines opérations complexes requièrent un ensemble d'objets génériques qui n'ont aucun rapport avec le domaine et dont la responsabilité ne peut être assignée à des classes du domaine, afin d'éviter le fort couplage.

La responsabilité doit être assignée à de nouvelles classes, **fabriquées** expressément pour effectuer l'opération.

Exemples

Bibliothèque

Dans un logiciel de gestion de Bibliothèque, la responsabilité de sauvegarder les informations à propos d'un livre dans une base de données ne doit pas être assignée à la classe Livre elle-même, mais à des classes séparées, indépendantes (donc plus génériques et réutilisables). Ces classes sont alors utilisées par la classe Livre pour effectuer la sauvegarde.

Indirection

Afin d'éviter le fort couplage entre deux classes, ou une dépendant à une interface spécifique, il est recommandé de créer une classe intermédiaire.

Il faut cependant ne pas créer trop de classes intermédiaires afin d'éviter de mauvaises performances (temps/mémoire).

Exemples

Paiement en ligne

La communication avec un serveur bancaire peut se faire grâce à un modem en utilisant des fonctions de l'API du système d'exploitation.
La création d'une classe Modem pour appeler l'API permet d'éviter l'appel à celle-ci dans les classes communiquant par modem.
L'application sera plus facilement portée pour un autre système en ne changeant que la classe Modem.

Protection

Le but du patron de conception Protection est d'assigner les responsabilités afin que des variations dans des classes (instabilité) ne produisent aucun effet indésirable sur le reste du système. Ce patron de conception est utilisé en prévision d'évolutions de l'application logicielle, afin d'éviter que de nouvelles fonctionnalités ne viennent perturber l'existant.

Pour mettre en œuvre la protection contre les variations, on peut :

- créer des méthodes pour accéder aux attributs d'une classe,
- créer une interface pour gérer de nouvelles classes en définissant les méthodes qui pourront être appelées,
- créer des classes intermédiaires (voir le patron de conception indirection).

Il faut veiller à ne pas trop anticiper les changements possibles afin d'éviter de complexifier la structure de l'application en créant trop de classes intermédiaires ou d'interfaces.

Ce patron est aussi utilisé en prévision d'évolution de classes externes utilisées dans l'application. Il faut éviter d'utiliser directement ces classes externes afin de ne pas avoir une forte dépendance. Si une classe externe évolue (changement de nom de méthode, ou du type d'objet retourné), l'application n'aura pas besoin d'être modifiée en profondeur.

Exemples

Bibliothèque

Dans le pseudo-code suivant, l'obtention d'un livre particulier passe par l'utilisation de la classe Collection (classe que l'on ne connait pas, qui pourrait évoluer, ou ne plus faire partie de l'application) :

```
Collection c = bibliothèque.getCollection();
Livre l = c.getLivre("Patrons de conception");
```

En appliquant le principe du patron de conception protection, la bibliothèque est responsable de l'utilisation de la classe inconnue pour obtenir un livre particulier :

```
Livre l = bibliothèque.getLivre("Patrons de conception");
```

Patrons d'entreprise

Les patrons de conception d'entreprise répondent aux problèmes d'architecture des applications d'entreprise (base de données, service web, ...) et sont décrits dans le livre « Patterns of Enterprise Application Architecture » écrit par Martin Fowler.

Les patrons d'entreprises sont nombreux et sont catégorisés de la façon suivante :

Logique du domaine (*Domain Logic Patterns*)

.

Architecture de source de données (*Data Source Architectural Patterns*)

.

Comportement objet-relationnel (*Object-Relational Behavioral Patterns*)

.

Structure objet-relationnel (*Object-Relational Structural Patterns*)

.

Association méta-données objet-relationnel (*Object-Relational Metadata Mapping Patterns*)

.

Présentation web (*Web Presentation Patterns*)

.

Distribution (*Distribution Patterns*)

.

Concurrence locale (hors-ligne) (*Offline Concurrency Patterns*)

.

État de session (*Session State Patterns*)

.

Patrons de base (*Base Patterns*)

Les patrons de base décrivent des structures basiques.
Passerelle (Gateway), Mapper, Type de base pour la couche, Interface séparée, Registre, Plugin.


Cette section est vide, pas assez détaillée ou incomplète.

Patrons de base

Les patrons de base décrivent des structures basiques.

Les différents patrons de base sont les suivants :

Passerelle
: Encapsuler une API non objet d'accès à une ressource externe dans une classe afin d'avoir un objet et de pouvoir changer de type de ressource plus facilement.

Mapper
: Faire communiquer différents objets sans créer une dépendance entre eux.

Type de base pour la couche
: Créer un type de base pour les comportements communs.

Interface séparée
: Définir l'interface dans un paquetage différent de l'implémentation afin de réduire le couplage entre deux parties d'un système.

Registre
: Créer un objet global permettant d'obtenir des objets et services communs de l'application.

Plugin
: Utiliser des services sans dépendre de l'implémentation.

Passerelle

Les logiciels les plus intéressants fonctionnent rarement seuls. Même le système orienté objet le plus pur a souvent affaire à des choses non objets comme une base de données relationnelle, des transactions, et des structures de données XML.

On utilise généralement une API spécialisée pour accéder à de telles ressources externes. Ces API sont naturellement compliquées du fait qu'elles prennent en compte la nature de la ressource. Pour utiliser une ressource (telle une base de données SQL, ou des données XML), il est nécessaire de comprendre l'API (Par exemple, JDBC pour la base de données, ou W3C ou JDOM pour le XML). Cela ne fait pas que compliquer la compréhension de l'appplication, mais cela complique également le changement : passer d'une base de données SQL à une structure XML.

La solution est d'encapsuler le code spécial API dans une classe dont l'interface ressemble à un objet normal. Les autres objets accèdent à la ressource à travers cette passerelle qui transforme les appels de méthodes en appels appropriés à l'API spécialisée.

Mapper

Dans une application, il peut être utile que certains objets communiquent entre eux. Mais la création d'une dépendance directe n'est parfois pas possible (modules externes non modifiables), ou pas souhaitable (garder une généricité).

La solution est de créer une nouvelle classe (ou un ensemble de classes) qui va gérer la communications entre ces objets.

Type de base pour la couche

Les classes d'une même couche logicielle ont souvent un comportement similaire ou besoin des mêmes méthodes.

Afin d'éviter que ces méthodes soient définies en plusieurs endroits, il est préférable de créer une classe de base pour toutes ces classes afin de rassembler les méthodes communes dans une seule classe.

Interface séparée

La qualité de la structure d'un système peut être améliorer en réduisant le couplage entre les parties du système. Une solution est de regrouper les classes en paquetages et de contrôler les dépendances entres eux. Des règles de dépendances peuvent alors être établies et respectées, comme par exemple éviter qu'une classe du domaine ne dépende d'une classe de la partie présentation.

Cependant, il est parfois nécessaire d'appeler des méthodes en contredisant ces règles de structure. Pour cela, il suffit de créer une interface dans le même paquetage, et de l'implémenter dans un autre. De cette façon, les classes utilisant l'interface ne dépendront pas de l'implémentation dont elles ne savent rien.

L'interface séparée fournie un point d'entrée pour une passerelle.

Registre

Pour trouver un objet, on commence à partir d'un autre objet en suivant les associations et en navigant dans la structure de l'application. Par exemple, pour trouver toutes les commandes d'un client, on démarre avec une référence à l'objet client et on utilise une méthode pour obtenir les commandes passées par celui-ci.

Cependant, il n'y a pas toujours une référence directe à l'objet de départ. Par exemple, on part du numéro du client sans avoir directement l'objet représentant le client. L'application peut prévoir une classe fournissant le service de trouver l'objet client à partir d'un numéro, mais il reste à savoir comment obtenir le service lui-même.

La solution est de créer un objet global permettant de trouver les différents services communs de l'application. Par exemple, avoir un registre des services, dont celui de trouver un client à partir de son numéro.

Plugin

Une classe peut utiliser les services d'une autre classe sans avoir à dépendre d'une implémentation concrète des services. Il peux exister plusieurs implémentation de ces services, pour différents contextes d'utilisation (ex: communication locale, communication distante, ...). Toutes les classes implémentant ces services doivent implémenter une interface commune qui pourra être utilisée par l'application. Ainsi l'application peut utiliser n'importe quelle implémentation des services sans avoir à en connaître les détails ni dépendre d'une implémentation particulière.

L'instance de la classe implémentant l'interface des services est passée à la classe utilisatrice, soit au constructeur pour maintenir le lien durant la vie de l'objet, soit à la méthode appelée pour une utilisation ponctuelle des services.

Autres patrons

D'autres patrons de conception que ceux vus précédemment existent. En voici quelques uns :

Type fantôme
: Utiliser un type pour ajouter une contrainte à la compilation.

Double-dispatch
: Permettre l'appel à une méthode surchargée en recourant au type dynamique d'un argument.

Post-Redirect-Get
: Éviter la soumission multiple d'un formulaire web lors d'un rafraichissement.

Map-Reduce
: Parallélisation d'un traitement sur des données volumineuses.

Évaluation retardée
: Retarder l'évaluation d'une fonction ou expression jusqu'à utilisation concrète du résultat.

Copie sur modification
: Retarder la création d'une copie privée d'une structure tant qu'elle n'est pas modifiée.

Injection de dépendance
: Ce patron de conception est utilisé pour le couplage dynamique.

Inversion de contrôle
: Ce patron de conception est utilisé pour réduire la dépendance à une séquence d'exécution particulière.

Modèle-Vue-Présentateur
: Ce patron de conception est dérivé du patron Modèle-Vue-Contrôleur.

Écart de génération
: Ce patron de conception est utilisé pour séparer une classe générée automatiquement et la partie personnalisation du code.

Objet nul
: Utiliser un objet nul dont les méthodes ne font rien au lieu d'utiliser une référence nulle.

Type fantôme

En général, un type de données est créé afin de pouvoir utiliser ses valeurs possibles.

Un type fantôme n'est utilisé que pour ajouter une contrainte qui puisse être vérifiée à la compilation. Cette fonctionnalité est utilisable avec la programmation fonctionnelle et également la programmation orientée objet en utilisant les templates ou types génériques.

Exemple en Java

Cet exemple utilise une classe dont les instances peuvent être en lecture seule ou écriture seule.

```
public class PName<V>
{
    // Types fantômes utilisés pour marquer les méthodes
    // dont les appels seront vérifiés à la compilation.
    public static interface ReadOnly { }
    public static interface WriteOnly { }
    public static interface ReadWrite extends ReadOnly, WriteOnly { }
    // Des interfaces sont utilisées au lieu de classes afin que
    // ReadWrite hérite des deux types précédents.

    private String name;

    private PName(String s) { this.name = s; }

    public static PName<ReadWrite> pname(String s)       { return new PName<ReadWrite>(s); }
    public static PName<ReadOnly>  readOnly(String s)  { return new PName<ReadOnly>(s); }
    public static PName<WriteOnly> readWrite(String s) { return new PName<WriteOnly>(s); }

    // Opérations de conversion
    @SuppressWarnings("unchecked")
    public static PName<ReadOnly> readable(PName<? extends ReadOnly> pname) { return (PName<ReadOnly>) pname; }
    @SuppressWarnings("unchecked")
    public static PName<WriteOnly> writeable(PName<? extends WriteOnly> pname) { return (PName<WriteOnly>) pnam

    // Opérations de lecture et d'écriture
    public static int get(PName<? extends ReadOnly> pname) { return pname.name; }
    public static void set(PName<? extends WriteOnly> pname, String v) { pname.name = v; }

    public static void main(String[] args) // Test
    {
        // Nom en lecture et écriture
        PName<ReadWrite> rw = pname("Test");
        System.out.println(get(rw));       // Lecture OK
        set(rw,"Exemple");                 // Écriture OK
        System.out.println(get(rw));       // Lecture OK

        // Version en lecture seule
        PName<ReadOnly> ro = readable(rw);
        System.out.println(get(ro));       // Lecture OK
        //set(ro,"Autre");                 // Écriture --> erreur à la compilation

        // Version en écriture seule
        PName<WriteOnly> wo = writeable(rw);
        set(wo,"Modification");            // Écriture OK
        //get(wo);                         // Lecture --> erreur à la compilation
    }
}
```

Les types `ReadOnly`, `WriteOnly` et `ReadWrite` sont des types fantômes (aucune instance existante) utilisés uniquement pour marquer le type `PName` et que le compilateur puisse effectuer les vérifications.

Double-dispatch

Ce patron de conception est lié à la notion de surcharge de méthode, et au polymorphisme. Il utilise le patron de conception Visiteur.

Introduction : notion de dispatch

Une classe peut définir une méthode qui peut être redéfinie dans une sous-classe (en C++ il s'agit de méthodes virtuelles, en Java toutes les méthodes sont virtuelles). La méthode réellement appelée est celle correspondant au type dynamique de l'objet (résolution à l'exécution, via une table de méthodes (vtable) par exemple), même si une référence à la classe de base est utilisée.

Exemple en Java :

```
// Une classe Object comportant un attribut : le nom de l'objet
class Object
{
    protected String name;
    public Object(String name)
    { this.name = name; }
    public String toString()
    { return "a "+name; }
}

// Une classe SimpleObject précisant que l'objet est simple
class SimpleObject extends Object
{
    public SimpleObject(String name)
    { super(name); }
    public String toString()
    { return "a simple "+name; }
}

// Le test :
Object book = new SimpleObject("book");
// référence de type Object mais objet de type SimpleObject
System.out.println(book.toString()); // SimpleObject.toString
```

Affiche :

```
a simple book
```

La plupart des langages de programmation orientés objet (C++, Java) se limite à cela concernant la résolution dynamique de types (donc à l'exécution plutôt qu'à la compilation). Ils ne permettent pas, par exemple, d'utiliser le type dynamique pour les arguments et déterminer quelle méthode surchargée appeler. Ces langages implémentent donc le **simple-dispatch** (un seul type résolu dynamiquement, celui de l'objet de l'appel à la méthode).

D'autres langages de programmation orientés objet permettent la résolution de type dynamique des arguments et la méthode surchargée appelée dépend du type réel et non du type de référence passé en argument. Cette fonctionnalité est appelée **multiple-dispatch**.

Exemple en Java où il n'existe que le simple-dispatch :

```
// NB: Les classes Object et SimpleObject de l'exemple précédent sont également
//     utilisées ici. Voir l'exemple précédent pour leur code source.

// Une classe Display pour afficher Object et SimpleObject
class Display
{
    public void display(Object o1, Object o2)
    { System.out.println("A display for object ("+o1+") and object ("+o2+")"); }
    public void display(Object o1, SimpleObject o2)
    { System.out.println("A display for object ("+o1+") and simple object ("+o2+")"); }
    public void display(SimpleObject o1, Object o2)
    { System.out.println("A display for simple object ("+o1+") and object ("+o2+")"); }
    public void display(SimpleObject o1, SimpleObject o2)
    { System.out.println("A display for simple object ("+o1+") and simple object ("+o2+")"); }
}

// Une classe SimpleDisplay pour afficher simplement Object et SimpleObject
class SimpleDisplay extends Display
{
    public void display(Object o1, Object o2)
    { System.out.println("A simple display for object ("+o1+") and object ("+o2+")"); }
    public void display(Object o1, SimpleObject o2)
    { System.out.println("A simple display for object ("+o1+") and simple object ("+o2+")"); }
```

```
    public void display(SimpleObject o1, Object o2)
    { System.out.println("A simple display for simple object ("+o1+") and object ("+o2+")"); }
    public void display(SimpleObject o1, SimpleObject o2)
    { System.out.println("A simple display for simple object ("+o1+") and simple object ("+o2+")"); }
}

// Le test :
Object book = new SimpleObject("book");
Object card = new SimpleObject("card");
Display disp = new SimpleDisplay();
disp.display(book, card); // -> SimpleDisplay.display(Object, Object)
```

Affiche :

```
A simple display for object (a simple book) and object (a simple card)
```

L'exemple précédent démontre que le langage Java ne permet pas nativement le **multiple-dispatch**. Dans le cas contraire, la méthode `SimpleDisplay.display(SimpleObject, SimpleObject)` aurait été appelée et l'affichage aurait été le suivant :

```
A simple display for simple object (a simple book) and simple object (a simple card)
```

Double-dispatch

Le **double-dispatch** est un patron de conception utilisant le patron Visiteur pour implémenter partiellement le multiple-dispatch, étendant le **simple-dispatch** en utilisant le type dynamique de l'un des arguments.

Le principe est d'ajouter une méthode dans les classes des arguments, appelée grâce au polymorphisme (le **simple-dispatch**). Cette méthode appelant la méthode originale, avec le pointeur this en argument dont le type réel est connu à la compilation.

Exemple en Java :

```
// Une classe Object comportant un attribut : le nom de l'objet
class Object
{
    protected String name;
    public Object(String name)
    { this.name = name; }
    public String toString()
    { return "a "+name; }
// méthode ajoutée pour implémentation du double-dispatch:
    public void display(Display d)
    { d.display(this); } // this de type Object
}

// Une classe SimpleObject précisant que l'objet est simple
class SimpleObject extends Object
{
    public SimpleObject(String name)
    { super(name); }
    public String toString()
    { return "a simple "+name; }
// méthode ajoutée pour implémentation du double-dispatch:
    public void display(Display d)
    { d.display(this); } // this de type SimpleObject
}

// Une classe Display pour afficher Object et SimpleObject
class Display
{
    public void display(Object o1)
    { System.out.println("A display for object ("+o1+")"); }
    public void display(SimpleObject o1)
    { System.out.println("A display for simple object ("+o1+")"); }
}

// Une classe SimpleDisplay pour afficher simplement Object et SimpleObject
class SimpleDisplay extends Display
{
    public void display(Object o1)
    { System.out.println("A simple display for object ("+o1+")"); }
    public void display(SimpleObject o1)
    { System.out.println("A simple display for simple object ("+o1+")"); }
}

// Le test :
```

```
Object book = new SimpleObject("book");
Display disp = new SimpleDisplay();
book.display(disp); // SimpleObject.display(Display) -> SimpleDisplay.display(SimpleObject)
```

Affiche :

```
A simple display for simple object (a simple book)
```

Voir aussi

- Ce patron de conception est une forme spécifique du patron Visiteur.
- (anglais) Double dispatch

Post-Redirect-Get

Le patron de conception Post-Redirect-Get est une solution spécifique au protocole HTTP pour le développement de serveur web.

Quand un utilisateur soumet un formulaire web (exemples : une commande, une souscription, une nouvelle version de page wiki) dont l'envoi est de type POST, le navigateur l'envoie avec la méthode POST, et reçoit une confirmation du serveur après traitement de la requête et du formulaire.

Le problème

Sans utilisation du patron de conception, les échanges sont les suivants (seule la première ligne est montrée) :

- *L'utilisateur envoi le formulaire (bouton Submit, OK, ...)*
- Client -> Serveur : envoi du formulaire

```
POST ...
```

- *Le serveur traite le formulaire*
- Serveur -> Client : envoi de la page de confirmation

```
HTTP/1.0 200 OK
```

- *Le client affiche la page qu'il vient de recevoir, et l'utilisateur décide de rafraichir la page.*
- *Donc le navigateur répète la même requête POST.*
- Client -> Serveur : ré-envoi du formulaire (donc nouveau traitement par le serveur)

```
POST ...
```

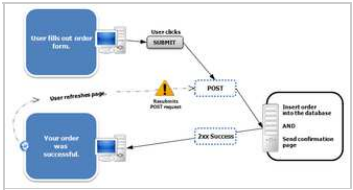


Diagramme d'un problème de multiple soumission.

La solution

Ce problème est résolu coté serveur en répondant par une redirection :

- *L'utilisateur envoi le formulaire (bouton Submit, OK, ...)*
- Client -> Serveur : envoi du formulaire

```
POST ...
```

- *Le serveur traite le formulaire*
- Serveur -> Client : redirection vers la page de confirmation

```
HTTP/1.0 303 See Other
```

- Client -> Serveur : demande avec la nouvelle URL de la page de confirmation en utilisant la méthode GET

```
GET ...
```

- Serveur -> Client : envoi de la page de confirmation

```
HTTP/1.0 200 OK
```

- *Le client affiche la page qu'il vient de recevoir, et l'utilisateur décide de rafraichir la page*
- *Donc le navigateur répète la même requête GET.*
- Client -> Serveur : rafraichir sans ré-envoi du formulaire

```
GET ...
```

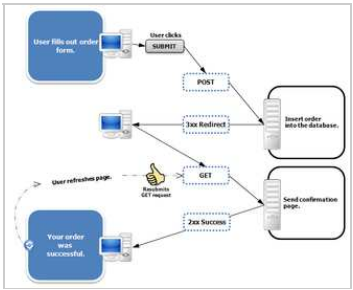


Diagramme de la résolution du problème avec le pattern PRG.

Le client reçoit une redirection au lieu de la page directement en réponse de l'envoi du formulaire. Le protocole HTTP spécifie que la redirection de code 303 doit être faite avec la méthode GET, ce que fait le client aussitôt pour recevoir la page de confirmation. Si l'utilisateur rafraîchit la page, le formulaire n'est plus renvoyé car la redirection a forcé le changement de méthode à GET.

Le nom du patron de conception donne donc un résumé des premiers échanges effectués (requête POST, réponse REDIRECT, requête GET).

Voir aussi

Post-Redirect-Get

Map-Reduce

Map-Reduce est le nom d'un framework de Google permettant de paralléliser un traitement sur des données volumineuses sur plusieurs machines d'un réseau (des nœuds). L'idée de base vient des fonctions Map et Reduce de la programmation fonctionnelle.

Le traitement de déroule en deux étapes principales :

Map
: Le problème est subdivisé par le nœud principal, en sous-problèmes qui sont soumis aux différents nœuds qui peuvent à leur tour le subdiviser davantage. Par exemple, pour rechercher une personne dans une base de données distribuée d'annuaire mondial un nœud pourrait recevoir la tâche de rechercher cette personne dans la base des personnes vivant en Espagne.

Reduce
: Les nœuds remontent le résultat du traitement aux nœuds les ayant sollicités. Ces nœuds parents construisent un résultat et le remonte à leur tour, et ainsi de suite jusqu'au nœud principal. Pour l'exemple précédent, le nœud peut retourner le numéro de téléphone, le pays de la personne trouvée...

Évaluation retardée

Avantage : Gagner du temps d'exécution

L'évaluation retardée consiste à ne pas exécuter du code avant que les résultats de ce code ne soient réellement nécessaires. Cela permet de gagner du temps en évitant de calculer un résultat qui pourrait ne pas être utilisé.

Il peut s'appliquer au calcul des termes d'une série infinie. Seuls les termes utilisés sont calculés. Par opposition, l'évaluation immédiate nécessiterait le calcul de tous les termes qui ne se terminerait donc pas.

Le patron de conception Copie sur modification est une forme d'évaluation retardée.

Copie sur modification

Avantage : Gagner de l'espace mémoire

Quand plusieurs processus (légers ou non), ou classes peuvent demander des ressources impossibles à distinguer initialement, il suffit de leur donner un pointeur commun en mémoire sur la ressource, au lieu de leur créer une copie privée. Au moment ou un processus ou une classe modifie la ressource, une copie privée est créée.

Le patron de conception copie sur modification est une forme d'évaluation retardée.

Injection de dépendance

L'injection de dépendance consiste à éviter une dépendance directe entre deux classes, et 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 ;
- Déclarer la classe B comme implémentation de cette interface I ;
- 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.

Inversion de contrôle

Le contrôle est ici un terme utilisé pour désigner l'ordre d'exécution des instructions d'une fonction ou méthode.

L'inversion de contrôle permet de réduire la dépendance d'une classe à un algorithme particulier ou une configuration particulière quand les méthodes de cette classe ou d'une autre sont utilisées pour effectuer une fonction complexe. Ce patron de conception est utilisé par certains frameworks. Plutôt que de définir un ordre fixe d'appel aux instructions dans une méthode, l'application de l'inversion de contrôle permet de faire appeler les divers traitements quand cela est nécessaire, par une autre classe (une classe du framework) qui implémente un algorithme particulier ou une configuration particulière.

Exemple

Une application d'évaluation calcule le coût d'une facture en fonction des données fournies par l'utilisateur (prix unitaire, description, quantité) pour chaque article acheté, en utilisant un framework d'interface utilisateur.

Une application de type console poserait une série de questions à l'utilisateur dans un ordre particulier, et le prix est donné à la fin en fonction des données entrées. La séquence est codée dans l'application qui **contrôle** donc la séquence d'exécution.

Une application de type graphique définit les différents contrôles de son interface graphique en ayant recours au framework et le prix s'affiche une fois que l'utilisateur clique le bouton "Évaluer". Coté application, le framework fait appel aux fonctions de l'application lors de certains évènements (clic du bouton "Ajouter un article", clic du bouton "Évaluer"). Le contrôle donc est inversé : c'est le framework qui **contrôle** l'exécution de l'application, et non plus l'application qui contrôle le framework.

L'injection de dépendance est un cas particulier d'inversion de contrôle concernant la dépendance d'une classe à une autre.

Modèle-Vue-Présentateur

Ce patron de conception est dérivé du patron Modèle-Vue-Contrôleur (MVC) (https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur). Il définit trois types de rôles :

Modèle
: Les classes représentant les données manipulées à travers l'interface utilisateur.

Vue
: Les classes présentant une vue des données à l'utilisateur.

Présentateur
: Partie communicant avec les deux autres pour traduire et transmettre les commandes de l'utilisateur envoyée de la vue vers le modèle et pour formater et afficher les données du modèle dans la vue.

Le principe est de découpler la vue et le modèle, en utilisant le présentateur comme intermédiaire.

Utiliser ce patron permet d'avoir plusieurs vues d'un même modèle (exemple: une table de données sous la forme d'un tableau modifiable et sous la forme d'un graphique). Et une même vue peut présenter les données de plusieurs modèles (vue combinée ou synthèse).

Plusieurs vues d'un même modèle peuvent être présentées simultanément. Quand l'utilisateur interagit avec une vue pour effectuer une modification, la vue transmet la requête au présentateur. Celui-ci la transmet au modèle (généralement avec transformation des paramètres, ou en appelant plus d'une méthode du modèle). Puis le présentateur notifie toutes les vues afin de les mettre à jour pour prendre en compte la modification effectuée.

Écart de génération

Ce patron de conception est utilisé quand une classe est générée automatiquement par un outil, pour créer une interface graphique par exemple. Tout ne peut être automatiquement généré, il est forcément nécessaire de spécifier les détails du comportement de la classe, impossible à générer automatiquement à partir du modèle graphique de l'outil.

Quand le modèle abstrait à partir duquel est généré la classe est modifié, l'outil peut supprimer le code qui a été modifié quand il régénère la classe. Certains outils marquent avec des commentaires les endroits du code où le développeur peut modifier le code ; l'inconvénient étant que le compilateur n'interdit pas la modification en dehors des marques. D'autres outils comparent le code originalement généré au code personnalisé actuel pour ré-appliquer les changements au nouveau code généré ; mais en général, cela ne fonctionne que si les modifications sont peu nombreuses.

La solution proposée par ce patron de conception est de créer une classe dérivée de celle générée automatiquement et d'effectuer toutes les personnalisations dans cette sous-classe. Seule la sous-classe dépend de la classe générée. Toutes les autres classes doivent utiliser cette sous-classe au lieu de la classe générée.

Ce patron s'applique dans les conditions citées précédemment et il faut également que le code régénéré conserve les attributs et méthodes précédents utilisés par la sous-classe.

Objet nul

Quand une fonction ou une méthode retourne une référence à un objet, celle-ci peut être nulle (aucun objet concret référencé). il faut alors tester la référence retournée afin de ne pas provoquer une erreur dans le programme.

Au lieu de retourner une référence nulle, il est possible de créer une sous-classe ou une classe d'implémentation d'interface spécifique dont les méthodes ne font rien. La fonction ou méthode retourne alors une instance de cette classe.

Les appels de méthodes avec un objet nul sont alors possibles, ce qui n'est pas le cas avec une référence nulle.

Comme l'état interne de cet objet ne change pas (les méthodes ne faisant rien), il est possible d'avoir une instance unique d'un tel objet (voir singleton).

Bibliographie et liens

Livres connexes

Quelques livres disponibles pour l'application des patrons de conception et compléter ses compétences :

- Programmation UML
- Introduction au test logiciel

Autres projets Wikimedia

- (français) Patron de conception sur wikipédia

Autres sites

- (anglais) http://www.vincehuston.org/dp/ Huston Design Pattern - Présente un tableau périodique pour classer les patrons de conception semblable à celui du tableau périodique des éléments.
- (anglais) http://martinfowler.com/articles/enterprisePatterns.html Enterprise Patterns - Site de Martin Fowler sur les patrons d'entreprise.

Articles

- L'article « Design Patterns in the JDK » (http://www.javacodegeeks.com/2011/03/design-patterns-in-jdk.html) énumère les utilisations des designs patterns dans l'implémentation de l'API Java.

Livres

- français *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* (traduction : *Jean-Marie Lasvergères*) - **Design Patterns** : *Catalogue de modèles de conceptions réutilisables* - Éditions Vuibert - 1999 - *490 pages* - ISBN 2711786447
- français *Craig Larman* - **UML 2 et les Design Patterns** *(3ème édition)* - ISBN 2744070904
- français *Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates* - **Design patterns** : *Tête la première (1ère édition)* - 2005 - ISBN 2841773507

- anglais *Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel* - **A Pattern Language : Towns, Buildings, Construction** - 1977 - ISBN 0195019199
- anglais *Martin Fowler* - **Patterns of Enterprise Application Architecture** *(6ème édition)* - Éditions Addison-Wesley - 2004 - ISBN 0321127420
- anglais *James O. Coplien, Douglas C. Schmidt* - **Pattern Languages of Program Design** - 1995 - ISBN 0201607344
- anglais *Craig Larman* - **Applying UML and Patterns** : *An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.)* - Éditions Prentice Hall PTR - 2005 - ISBN 0-13-148906-2