


Génie Logiciel Avancé

Cours 5 — Conception d'un système à objets

Mo Foughali
foughali@irif.fr

Laboratoire IRIF, Université Paris Cité

2021–2022

URL <https://moodle.u-paris.fr/course/view.php?id=10699>
Copyright © 2022– Mo Foughali
© 2011–2014, 2020–2021 Stefano Zacchiroli
© 2010 Yann Régis-Gianas
License Creative Commons Attribution-ShareAlike 4.0 International License
<https://creativecommons.org/licenses/by-sa/4.0/>


- 1 Patron de conception (design patterns) à objets
 - Création (*Creational Patterns*)
 - Structure (*Structural Patterns*)
 - Comportement (*Behavioral Patterns*)
- 2 Synthèse

1 Patron de conception (design patterns) à objets

- Création (*Creational Patterns*)
- Structure (*Structural Patterns*)
- Comportement (*Behavioral Patterns*)

2 Synthèse

Architecture d'un système à objets

Les **architectures logicielles** donnent des pistes pour organiser un système suivant multiples niveaux d'abstractions, multiples modèles de calcul et multiples composants de façon à :

minimiser les interdépendances et maximiser la cohésion

Les **objets** (et les modules) sont des briques élémentaires permettant de réaliser ces architectures puisque :

- Ils aident à maximiser la cohésion grâce à l'**encapsulation** de leur état.
- Les **classes abstraites** et les **interfaces** permettent de minimiser les interdépendances en contrôlant le niveau de détails des services visibles par les clients d'un objet.

Le sujet de ce cours :

Comment agencer ces briques ?

Deux difficultés

- ① La conception de composants réutilisables est une activité délicate.
- ② La programmation objet a des problèmes intrinsèques.

Les patrons de conception : des « recettes »

Un programmeur ou un concepteur expérimenté **réutilise** des **procédés d'organisation** qui ont fonctionné dans le passé.

- Ces solutions sont indépendantes (en général) du langage objet utilisé.

Exemple

XHTML + CSS : séparer un contenu de la description de sa mise en forme.

- un composant traitant les données (modèle),
- un composant de visualisation (vue).

Idée à retenir :

- la fameuse *separation of concerns* de Dijkstra.
- “un composant bien conçu se focalise sur un unique problème”

~Menu~

Création

Fabrique abstraite (Abstract Factory)

Monteur (Builder)

Fabrique (Factory Method)

Prototype (Prototype)

Singleton (Singleton)

Structure

Adaptateur (Adapter)

Pont (Bridge)

Objet composite (Composite)

Décorateur (Decorator)

Facade (Facade)

Poids-mouche ou poids-plume (Flyweight)

Proxy (Proxy)

Comportement

Chaine de responsabilité (Chain of responsibility)

Commande (Command)

Interpréteur (Interpreter)

Itérateur (Iterator)

Médiateur (Mediator)

Memento (Memento)

Observateur (Observer)

État (State)

Stratégie (Strategy)

Patron de méthode (Template Method)

Visiteur (Visitor)



Gamma, Helm, Johnson, Vlissides
*Design Patterns : Elements of Reusable
Object-Oriented Software.*
Addison-Wesley, 1995.

auteurs AKA "Gang of Four" (GoF)

Classification des patrons de conception

GoF ont explicité trois grandes classes de patrons dans leur livre, chacune spécialisée dans :

création	d'objets	(<i>creational patterns</i>)
structure	des relations entre objets	(<i>structural patterns</i>)
comportement	des objets	(<i>behavioral patterns</i>)

Réutilisation white-box ou black-box ?

- on parle de *white-box reuse* quand on réutilise du code à travers l'héritage
 - ▶ est dangereux, car les sous-classes ont droit à briser l'encapsulation de la classe mère
 - ▶ n'est pas toujours possible (p.ex., code propriétaire dont la documentation interne n'est pas disponible)
- on parle de *black-box reuse* quand on réutilise du code à travers la *composition* d'objets (association/composition/agrégation) et l'implémentation d'interfaces
 - ▶ est dynamique, donc le compilateur et le langage de programmation n'aident pas à imposer des contraintes de bonne forme
 - ★ est plus difficile pour raisonner sur un logiciel
 - ▶ les objets parlent entre eux seulement à travers des interfaces, donc l'encapsulation est sauvegardée

Principe général

Favor object (and interface) composition over class inheritance.

Réutilisation white-box ou black-box ?

Exemple... (vu en amphi)

Dans ce cours, les patterns sont traités de manière générique (diagrammes UML et pseudo-code). Pour quelques patterns, des exemples avec des implémentations en Java sont abordés. La majorité des explications données en amphi ont eu les ressources suivantes comme supports :



James W. Cooper

JavaTM Design Patterns : A Tutorial.

Addison-Wesley, 2000. (PDF disponible sur Moodle)

<https://refactoring.guru/design-patterns/java>

La dernière ressource est ludique, plus récente mais un peu amateur (comme toujours, le contenu du support du cours et les explications données en amphi restent la référence qui fait foi)

- 1 Patron de conception (design patterns) à objets
 - Création (*Creational Patterns*)
 - Structure (*Structural Patterns*)
 - Comportement (*Behavioral Patterns*)
- 2 Synthèse

Fabrique (Factory)

Principe Créer des objets différents en fonction de certains arguments.
Pourquoi faire ?

Exemple Vous voudriez implémenter un dictionnaire. La structure de données la plus adaptée possible (une table de hachage tant que sa taille n'excède pas quelques Mb, une base de données sinon) doit être choisie. Les prochaines versions du dictionnaire intégreront de nouvelles structures de données.

Problème Comment rendre le choix de la structure de données indépendant de l'acte d'instanciation ?

Fabrique (Factory)

Principe Créer des objets différents en fonction de certains arguments.

Pourquoi faire ?

Exemple Vous voudriez implémenter un dictionnaire. La structure de données la plus adaptée possible (une table de hachage tant que sa taille n'excède pas quelques Mb, une base de données sinon) doit être choisie. Les prochaines versions du dictionnaire intégreront de nouvelles structures de données.

Problème Comment rendre le choix de la structure de données indépendant de l'acte d'instanciation ?

Solution On utilise une fabrique (factory) qui, en fonction de la taille estimée du dictionnaire, choisit la bonne sous-classe (dictionnaire utilisant une table de hachage ou dictionnaire utilisant une base de données) et en retourne une instance.

Fabrique (factory)

On distingue deux types de fabriques :

- La fabrique simple (simple factory)
- La fabrique abstraite (abstract factory)

Recette

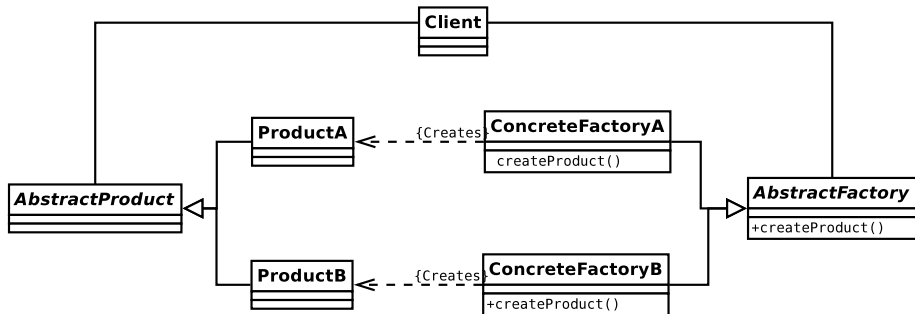
- Soit X la classe dont on veut instancier des sous-classes X_1, X_2, \dots en fonction de certains arguments,
- $XFactory$, la fabrique simple de X , est la classe qui passera les arguments en question à X ,
- X décidera en fonction des arguments passés d'instancier X_1 ou $X_2 \dots$
- Les implémentations des sous-classes de X ne devraient donc pas être connues, on ne divulgue que X et $XFactory$.

Le diagramme de classe de la fabrique simple ainsi que deux exemples avec du code Java (tous vus en amphi) sont disponibles dans le livre de Cooper (pages 25-29)

- Une fabrique abstraite est un pattern à “un niveau d’abstraction plus haut” de la fabrique simple,
- Il s’agit d’appliquer le motif de la fabrique simple à la fabrique simple elle-même,
- Dans un premier temps, et en fonction de certains arguments, une “famille d’objets” est désignée,
- Dans un second temps, et en fonction d’autres arguments, des objets sont créés au sein de la famille désignée précédemment.

Un exemple avec du code Java (vu en amphi) est disponible dans le livre de Cooper (pages 37-41)

Fabrique abstraite — UML



<http://thisblog.runsfreesoftware.com/?q=Abstract+Factory+Design+Pattern+UML+Class+Diagram>

Singleton (*Singleton*)

Principe interdire la création de plusieurs objets d'une même classe.

Pourquoi faire ?

Exemple Un programme s'exécute dans un environnement qui a une existence physique particulière (un système d'exploitation, un numéro de processus, ...). On veut le représenter par un objet qui est une instance d'une classe *Environment*.

Problème Pour une exécution donnée, il ne doit exister qu'une unique instance de la classe *Environment*.

Singleton (*Singleton*)

Principe interdire la création de plusieurs objets d'une même classe.

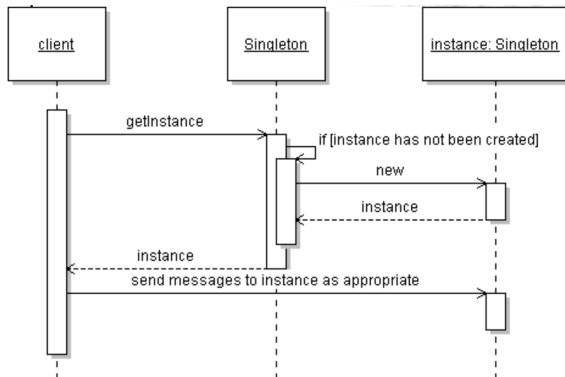
Pourquoi faire ?

Exemple Un programme s'exécute dans un environnement qui a une existence physique particulière (un système d'exploitation, un numéro de processus, ...). On veut le représenter par un objet qui est une instance d'une classe *Environment*.

Problème Pour une exécution donnée, il ne doit exister qu'une unique instance de la classe *Environment*.

Solution On empêche l'instanciation explicite de la classe *Environment* en en construisant une seule instance. Cette même instance est retournée à travers l'invocation d'une méthode publique.

Singleton (*Singleton*) — UML (diagramme de séquence)



Singleton (*Singleton*)

Recette pour une classe X :

- Rendre le constructeur de X privé (private) pour empêcher son invocation de l'extérieur,
- Déclarer une seule instance privé et statique (static private) de X ,
- Définir une méthode publique (p.ex. getInstance()) qui permet d'accéder à la seule instance de X .

Singleton (*Singleton*) — exemple en Java

```
// random number generator
public class RandomGenerator {
    private static RandomGenerator gen = new
        RandomGenerator();

    public static RandomGenerator getInstance() {
        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

Problème : une exécution de ce code ne collerait pas avec le diagramme de séquence vu précédemment. Pourquoi ? Solution ?
(Réponses vues en amphi).

1 Patron de conception (design patterns) à objets

- Création (*Creational Patterns*)
- Structure (*Structural Patterns*)
- Comportement (*Behavioral Patterns*)

2 Synthèse

Adaptateur (*Adapter*)

Principe Permettre la collaboration d'objets implémentant des interfaces "incompatibles".

Pourquoi faire ?

Exemple Vous avez développé une bibliothèque d'entrées/sorties fournissant une abstraction sur des tableaux unidimensionnels stockés dans p.ex. une base de données. Vous voudriez l'interfacer avec une bibliothèque de traitement d'images (dont le code source est inaccessible) qui attend un objet fournissant une interface d'accès en lecture et en écriture à un tableau en deux dimensions contenant des triplets d'octets.

Problème Comment concilier les services proposés par la bibliothèque d'entrées/sorties et l'interface attendue par la bibliothèque de traitement d'images ?

Adaptateur (*Adapter*)

Principe Permettre la collaboration d'objets implémentant des interfaces "incompatibles".

Pourquoi faire ?

Exemple Vous avez développé une bibliothèque d'entrées/sorties fournissant une abstraction sur des tableaux unidimensionnels stockés dans p.ex. une base de données. Vous voudriez l'interfacer avec une bibliothèque de traitement d'images (dont le code source est inaccessible) qui attend un objet fournissant une interface d'accès en lecture et en écriture à un tableau en deux dimensions contenant des triplets d'octets.

Problème Comment concilier les services proposés par la bibliothèque d'entrées/sorties et l'interface attendue par la bibliothèque de traitement d'images ?

Solution Utiliser un objet qui implémente l'interface attendue en faisant appel aux services proposés par une instance de la bibliothèque d'entrées/sorties.

Adaptateur (*Adapter*) — en Java

```
// image manipulation library
public interface ColorImage2D {
    public int    get_width    ();
    public int    get_height  ();
    public short  get_red      (int x, int y);
    public short  get_green    (int x, int y);
    public short  get_blue     (int x, int y);
}

public static class ImageProcessing {
    public static void blur (ColorImage2D image) {}
}

// input/output library
public class IOArray {
    public short  get (int x) { return 0; }
    public int    size ()    { return 0; }
}
```

Adaptateur (*Adapter*) — en Java

```
public class IOArrayToColorImage2DAdapter implements ColorImage2D {
    private int    width;
    private int    height;
    private IOArray array;

    IOArrayToColorImage2DAdapter (IOArray a, int w, int h) {
        assert (array.size () == w * h);
        this.width  = w;
        this.height = h;
        this.array  = a;
    }

    public int    get_width  () { return width; }
    public int    get_height () { return height; }
    public int    get_point  (int x, int y) {
        return array.get (y * width + x); }
    public short  get_red    (int x, int y) {
        return (short) (this.get_point (x, y) & 0xff); }
    public short  get_green  (int x, int y) {
        return (short) (this.get_point (x, y) >> 8 & 0xff); }
    public short  get_blue   (int x, int y) {
        return (short) (this.get_point (x, y) >> 16 & 0xff); }
}
```

Adaptateur (*Adapter*)

Recette

- Soit I l'interface de l'application que vous avez développée (interface client) et S le service tiers que vous voulez utiliser (p.ex. bibliothèque externe),
- *Adapter* est la classe que vous créerez pour “adapter” S (appelé “Adaptee”) à I ,
- *Adapter* implémente donc I et emballe (wrap) S pour qu'il puisse être utilisé par le client (sans briser l'abstraction de S , cf. black-box reuse).

Le diagramme de classe de l'adapter ainsi qu'un exemple en pseudo-code (vus en amphi) sont disponibles ici

<https://refactoring.guru/design-patterns/adapter>

Principe Séparer les classes en deux hiérarchies indépendantes (on parle d'une hiérarchie "Abstraction" et d'une hiérarchie "Implémentation").

Pourquoi faire ? (et d'abord, de quoi parle-t-on ?)

Exemple En amphi nous avons vu rapidement un exemple basé sur des politiques d'ordonnancement et leurs implémentations dans des OS différents. Une version "simpliste" de cet exemple est disponible ici

https://sourcemaking.com/design_patterns/bridge

Principe Séparer les classes en deux hiérarchies indépendantes (on parle d'une hiérarchie "Abstraction" et d'une hiérarchie "Implémentation").

Pourquoi faire ? (et d'abord, de quoi parle-t-on ?)

Exemple 2 Un logiciel de dessin propose une hiérarchie de formes (carré, cercle, losange, ...). Ces formes sont implémentées comme des ensembles de points qui peuvent être décrits de multiples façons (équation vectorielle, image *bitmap*, ...).

Problème Pour éviter la multiplication des classes par héritage, il est nécessaire de décorréler la hiérarchie des abstractions (les formes) de leurs implémentations (les ensembles de points) car elles peuvent varier de manière indépendante.

Pont (*Bridge*)

Principe Séparer les classes en deux hiérarchies indépendantes (on parle d'une hiérarchie "Abstraction" et d'une hiérarchie "Implémentation").

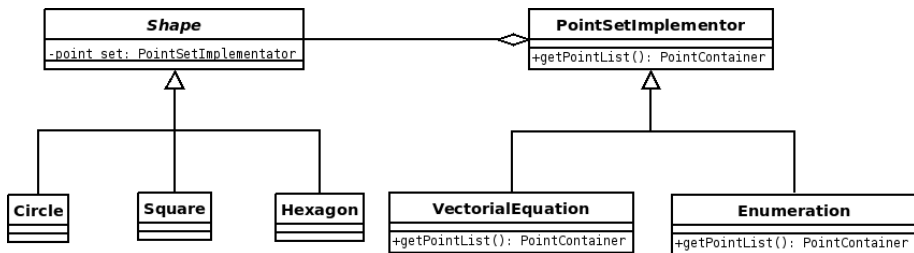
Pourquoi faire ? (et d'abord, de quoi parle-t-on ?)

Exemple 2 Un logiciel de dessin propose une hiérarchie de formes (carré, cercle, losange, ...). Ces formes sont implémentées comme des ensembles de points qui peuvent être décrits de multiples façons (équation vectorielle, image *bitmap*, ...).

Problème Pour éviter la multiplication des classes par héritage, il est nécessaire de décorréler la hiérarchie des abstractions (les formes) de leurs implémentations (les ensembles de points) car elles peuvent varier de manière indépendante.

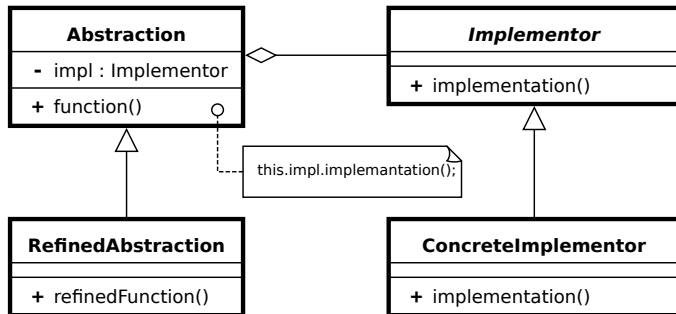
Solution Les implémentations sont organisées dans une hiérarchie indépendante. Une instance de la classe abstraite du sommet de cette hiérarchie (une classe nommée *Implementor*) est agrégée à toute forme.

Pont (*Bridge*) — UML (de l'exemple 2)



Nous retrouvons encore une fois le principe de black-box reuse (favor composition over inheritance). Ce pattern résout une problématique classique liée à l'utilisation systématique de l'héritage (explosion de nombre de classes).

Pont (*Bridge*) — UML (cas général)



http://en.wikipedia.org/wiki/File:Bridge_UML_class_diagram.svg

Décorateur (*Decorator*)

Principe Rajouter de nouveaux comportements à un objet existant sans modifier ou casser l'encapsulation de ce dernier.

Pourquoi faire ?

Exemple Dans un traitement de texte, un document est formé de différents composants (texte, image, ...). L'utilisateur doit pouvoir associer une URL à chacun de ces composants. Une fois associé, on peut cliquer sur le composant pour suivre le lien.

Problème Comment rajouter une fonctionnalité **dynamiquement** à un objet ?

Décorateur (*Decorator*)

Principe Rajouter de nouveaux comportements à un objet existant sans modifier ou casser l'encapsulation de ce dernier.

Pourquoi faire ?

Exemple Dans un traitement de texte, un document est formé de différents composants (texte, image, ...). L'utilisateur doit pouvoir associer une URL à chacun de ces composants. Une fois associé, on peut cliquer sur le composant pour suivre le lien.

Problème Comment rajouter une fonctionnalité **dynamiquement** à un objet ?

Solution Une classe *Decorator* agrège un composant particulier et implémente l'interface d'un composant en *délégant tous les messages correspondants à son instance agrégée*. En héritant de *Decorator*, on peut rajouter un comportement particulier p.ex. la gestion des hyperliens.

Décorateur (*Decorator*)

Principe Rajouter de nouveaux comportements à un objet existant sans modifier ou casser l'encapsulation de ce dernier

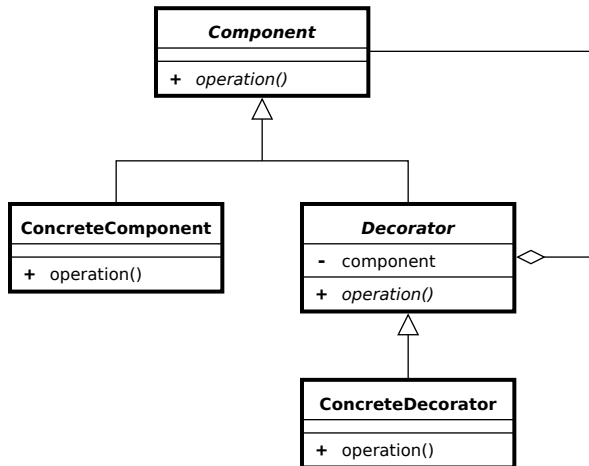
Pourquoi faire ?

Exemple 2 En amphi nous avons vu l'exemple de décorateurs permettant de rajouter des fonctionnalités telles que la compression/décompression et le chiffage/déchiffage de fichiers aux objets permettant leur écriture/lecture.

Cet exemple est disponible ici

<https://refactoring.guru/design-patterns/decorator>

Décorateur (*Decorator*) — UML



http://en.wikipedia.org/wiki/File:Decorator_UML_class_diagram.svg

Façade (*Facade*)

Principe Fournir une interface simplifiée pour accéder plus facilement à un ensemble de classes complexe en fonction des besoins du client.

Exemple Pour traiter les données d'un programme de gestion de bibliothèque multimédia, on désire utiliser l'API de Postgresql, un gestionnaire de base de données très élaboré. L'utilisation de la base de données est essentiellement un dictionnaire associant des noms de fichiers et des descriptions à des noms symboliques.

Problème Comment fournir une interface simple en utilisant un système complexe ?

Façade (*Facade*)

Principe Fournir une interface simplifiée pour accéder plus facilement à un ensemble de classes complexe en fonction des besoins du client.

Exemple Pour traiter les données d'un programme de gestion de bibliothèque multimédia, on désire utiliser l'API de Postgresql, un gestionnaire de base de données très élaboré. L'utilisation de la base de données est essentiellement un dictionnaire associant des noms de fichiers et des descriptions à des noms symboliques.

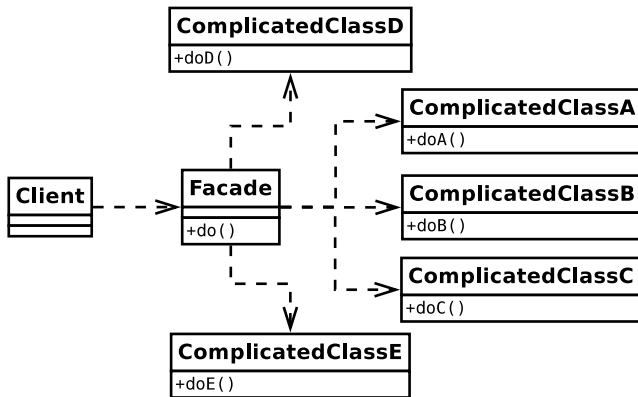
Problème Comment fournir une interface simple en utilisant un système complexe ?

Solution Le patron façade implémente seulement les méthodes utiles au dictionnaire en utilisant l'API de Postgresql. Il s'agit donc d'implémenter une couche d'abstraction.

Principe Fournir une interface simplifiée pour accéder plus facilement à un ensemble de classes complexe en fonction des besoins du client.

Dans le cas général, façade fournit une abstraction unique (simplifiée) sur plusieurs sous-systèmes, sans empêcher de les utiliser directement (*make simple tasks simple, complex tasks possible*).

Façade (Facade) — UML



<http://thisblog.runsfreesoftware.com/?q=Facade+Design+Pattern+UML+Class+Diagram>

Poids-mouche (*Flyweight*) — description

Principe Factoriser les attributs ayant la même valeur pour un groupe d'objets dans une classe à part.

Pourquoi faire ?

Exemple Vous avez développé une application qui permet de planter et afficher des arbres dans une forêt. Chaque arbre (classe *Tree*) a des attributs stockant ses coordonnées, mais aussi des attributs pour sa couleur, son nom et sa texture.

Problème Avec un nombre conséquent d'arbres plantés, la mémoire vive risque de vite saturer. Ceci est dû à la création de plusieurs objets de la classe *Tree* avec plusieurs attributs chacun.

Poids-mouche (*Flyweight*) — description

Principe Factoriser les attributs ayant la même valeur pour un groupe d'objets dans une classe à part.

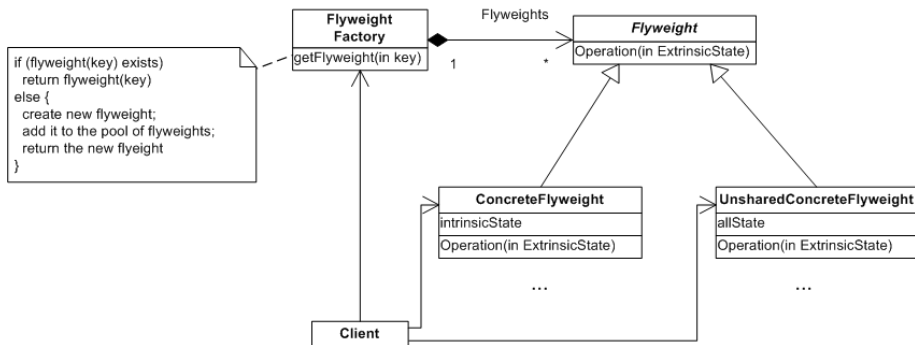
Pourquoi faire ?

Exemple Vous avez développé une application qui permet de planter et afficher des arbres dans une forêt. Chaque arbre (classe *Tree*) a des attributs stockant ses coordonnées, mais aussi des attributs pour sa couleur, son nom et sa texture.

Problème Avec un nombre conséquent d'arbres plantés, la mémoire vive risque de vite saturer. Ceci est dû à la création de plusieurs objets de la classe *Tree* avec plusieurs attributs chacun.

Solution Vue en amphi avec un diagramme UML et du pseudo-code disponibles ici <https://refactoring.guru/design-patterns/flyweight>

Poids-mouche (*Flyweight*) — UML



<http://www.lepus.org.uk/ref/companion/Flyweight.xml>

1 Patron de conception (design patterns) à objets

- Création (*Creational Patterns*)
- Structure (*Structural Patterns*)
- Comportement (*Behavioral Patterns*)

2 Synthèse

Chaîne de responsabilité (*Chain of responsibility*)

Principe Des objets forment une chaîne : un objet répond à un message s'il peut sinon il le transfère à son voisin.

Pourquoi faire ?

Exemple On veut intégrer un mécanisme de greffons (*plugins*) a un traitement de texte pour pouvoir rajouter dynamiquement la gestion de certains types de composants.

Problème Comment écrire une fois pour toute la fonction qui choisit le bon greffon à utiliser par observation du type du composant à traiter ?

Chaîne de responsabilité (*Chain of responsibility*)

Principe Des objets forment une chaîne : un objet répond à un message s'il peut sinon il le transfère à son voisin.

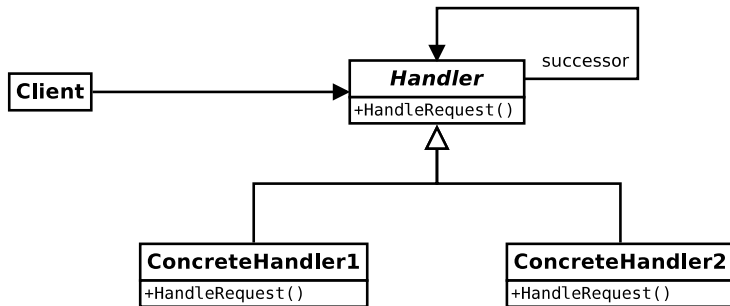
Pourquoi faire ?

Exemple On veut intégrer un mécanisme de greffons (*plugins*) a un traitement de texte pour pouvoir rajouter dynamiquement la gestion de certains types de composants.

Problème Comment écrire une fois pour toute la fonction qui choisit le bon greffon à utiliser par observation du type du composant à traiter ?

Solution On donne un identifiant unique pour les types de composants à traiter. Cet identifiant va servir de message de première classe. On définit une interface pour les greffons contenant une méthode d'interprétation de ces messages. La liste des greffons est stockée par l'application. Lorsqu'un greffon ne sait pas traiter un message, il le transfère au suivant.

Chaîne de responsabilité (*Chain of responsibility*) — UML



Chaîne de responsabilité (*Chain of responsibility*)

Un second exemple vu en amphi avec du pseudo-code est disponible à <https://refactoring.guru/design-patterns/chain-of-responsibility>

Memento (*Memento*)

Principe Sauvegarder et restorer l'état d'un objet sans le divulguer.

Pourquoi faire ?

Exemple Dans un contexte d'accès concurrent à une ressource, il est parfois utile d'être optimiste : on lance la requête d'accès de façon asynchrone, on continue la partie du calcul qui ne dépend pas de l'accès et on termine le calcul une fois que le résultat de la requête est arrivé. Mais que faire lorsque la requête a échoué ? On doit se replacer dans l'état précédant la continuation des calculs. . .

Problème Comment restaurer l'état d'un objet ?

Memento (*Memento*)

Principe Sauvegarder et restaurer l'état d'un objet sans le divulguer.

Pourquoi faire ?

Exemple Dans un contexte d'accès concurrent à une ressource, il est parfois utile d'être optimiste : on lance la requête d'accès de façon asynchrone, on continue la partie du calcul qui ne dépend pas de l'accès et on termine le calcul une fois que le résultat de la requête est arrivé. Mais que faire lorsque la requête a échoué ? On doit se replacer dans l'état précédant la continuation des calculs. . .

Problème Comment restaurer l'état d'un objet ?

Solution Une classe *Memento* sert à représenter l'état d'un objet de type *Originator*. Une classe *CareTaker* sert à stocker ses états, donc des objets de type *Memento*, sans briser leur encapsulation. Si la requête échoue, l'état que l'on veut restaurer est requis par *Originator* et fourni en retour de message par *CareTaker*.

Memento (*Memento*)

Recette

- La classe *Memento* est typiquement une classe interne à *Originator*. Ceci permet à la seconde d'accéder aux attributs et méthodes privés de la première.
- Tout se passe entre *Originator* et *CareTaker* par envoi de messages. En aucun cas l'objet de type *CareTaker* ne modifie directement l'état de l'objet de type *Originator*.
- On retrouve encore une fois le principe de black-box reuse : la classe *CareTaker* agrège la classe *Memento* ce qui permet de maintenir la cohésion de cette dernière.

Le diagramme UML général de ce pattern ainsi qu'un exemple avec du pseudo-code, vus en amphi, sont disponibles ici

<https://refactoring.guru/design-patterns/memento>

Observateur (*Observer*)

Principe Permettre à des objets de suivre et arrêter de suivre les changements d'état d'un autre objet.

Pourquoi faire ?

Exemple Imaginons l'application d'une séquence de filtres dans un logiciel de retouche d'images. Pour tenir l'utilisateur au courant de la progression du calcul, plusieurs objets s'affichent dans son interface : une barre de progression, un pourcentage dans la barre de titre, une icône sur son bureau. . . . Comment s'assurer qu'ils seront tous mis au courant de l'avancement du processus ?

Problème Comment modéliser l'observation de l'état d'un objet par un ensemble dynamiquement défini d'autres objets ?

Observateur (*Observer*)

Principe Permettre à des objets de suivre et arrêter de suivre les changements d'état d'un autre objet.

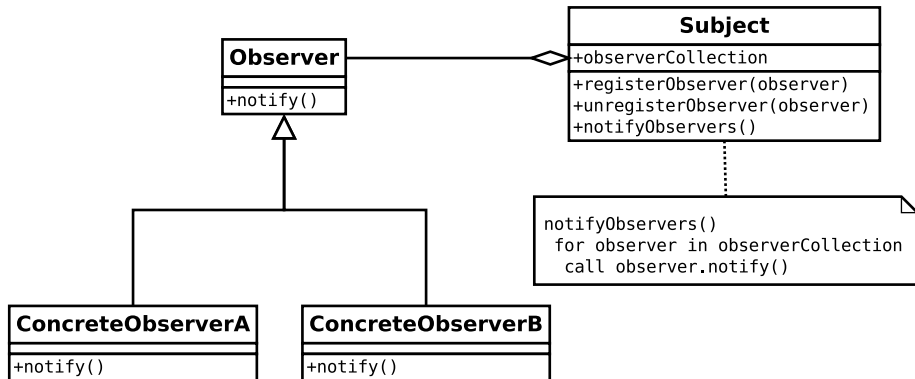
Pourquoi faire ?

Exemple Imaginons l'application d'une séquence de filtres dans un logiciel de retouche d'images. Pour tenir l'utilisateur au courant de la progression du calcul, plusieurs objets s'affichent dans son interface : une barre de progression, un pourcentage dans la barre de titre, une icône sur son bureau. . . . Comment s'assurer qu'ils seront tous mis au courant de l'avancement du processus ?

Problème Comment modéliser l'observation de l'état d'un objet par un ensemble dynamiquement défini d'autres objets ?

Solution L'objet observé offre une méthode *registerObserver* permettant à un observateur de s'enregistrer pour être informé. Pour cela, celui-ci doit proposer une méthode *notifyObservers*. A chaque fois qu'il est mis à jour, l'objet observé itère sur l'ensemble de ses observateurs et les notifie de sa modification.

Observateur (*Observer*) — UML



<http://en.wikipedia.org/wiki/File:Observer.svg>

Observateur (*Observer*)

- Souvent, les observateurs sont appelés “Subscribers” et l'objet observé “Publisher”
- On parle alors du protocole publisher-subscriber
- Un exemple du pattern Observer (avec la terminologie publisher-subscriber) avec du pseudo-code, vu en amphi, est disponible ici [https ://refactoring.guru/design-patterns/observer](https://refactoring.guru/design-patterns/observer)

Exemple Une intelligence artificielle pour un jeu d'échec s'appuie sur une heuristique, c'est-à-dire une évaluation approximative de la valeur d'une situation pour le joueur. Il existe de nombreuses heuristiques possibles. Est-ce à dire qu'il faille créer une sous-classe par heuristique ?

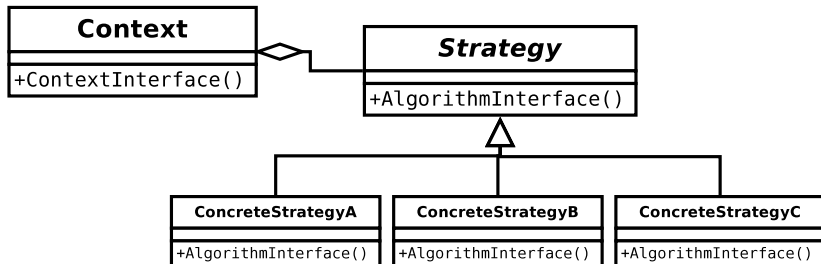
Problème Comment paramétrer une classe par un comportement ?

Exemple Une intelligence artificielle pour un jeu d'échec s'appuie sur une heuristique, c'est-à-dire une évaluation approximative de la valeur d'une situation pour le joueur. Il existe de nombreuses heuristiques possibles. Est-ce à dire qu'il faille créer une sous-classe par heuristique ?

Problème Comment paramétrer une classe par un comportement ?

Solution Un comportement est implémentée par une classe. Le comportement est donné en argument au constructeur (ou aux autres méthodes quand nécessaire) de la classe à paramétrer. Toutes utilisations de l'heuristique passe à travers la classe comportement.

Stratégie (*Strategy*) — UML



<http://thisblog.runsfreesoftware.com/?q=node/64>

- 1 Patron de conception (design patterns) à objets
 - Création (*Creational Patterns*)
 - Structure (*Structural Patterns*)
 - Comportement (*Behavioral Patterns*)
- 2 Synthèse

Pour conclure

- Les patrons de conception permettent de réutiliser une solution qui a déjà fait ses preuves face à des problématiques de design récurrentes.

L'utilisation des patrons de conception est répandue, avec des patterns beaucoup plus populaires que d'autres (le pattern Singleton, par exemple, est très utilisé).

Quelques exemples d'utilisation dans des logiciels connus :

- Les éditeurs de texte utilisent le pattern Memento (pour implémenter le retour en arrière “undo”),
- Java utilise la pattern Decorator pour rajouter des fonctionnalités aux méthodes I/O de base,
- Le pattern Observer est au coeur de ROS, le framework le plus populaire aujourd'hui en robotique,
- etc.