Chapitre 04

Les Design Patterns (ou Patron de Conception)

L'objectif de ce chapitre est d'introduire le concept des design patterns en parcourant les principaux.

1. Introduction

Les Designs Patterns sont à la Programmation Objet ce que sont les algorithmes à la Programmation Structurée.

Dès que l'on fait une conception UML d'un programme informatique en utilisant les Diagrammes de Classes, et que l'on reproduit ces diagrammes plusieurs fois, on crée, sans le savoir, des Designs Patterns (ou DP).

L'objectif ici est de découvrir ce que sont ces DP et de les écrire dans nos démarches de conception, dans un acte volontaire et souvent générique.

Ecrire volontairement ces DP en amont de notre démarche de conception est une garantie d'une meilleure conception car les DP représentent la description de nombreuses situations devenues maintenant standards dans la réalisation de nombreux Systèmes d'Information.

Par définition, un DP est :

- réutilisable (au moins) et générique (si possible)

Par définition, l'utilisation des DP permettent une conception :

- plus robuste aux évolutions en cours de réflexion ou à venir
- plus compréhensible car partagée par tous
- **plus déployables car les interfaces permettent la séparation des composants sur le réseau**

~~*Faire un schéma d'explication*~~

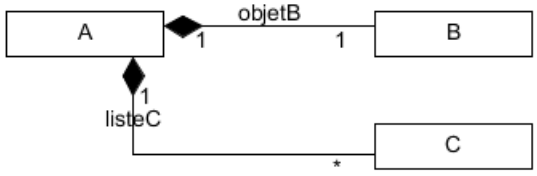
2. Rappels des concepts de la POO

Il est indispensable de connaître parfaitement les concepts de base de la Programmation Orientée Objet :

- Les liens d'associations : Composition et Agrégation
- L'héritage des classes
- La classe abstraite et l'Interface

2.1. Associations

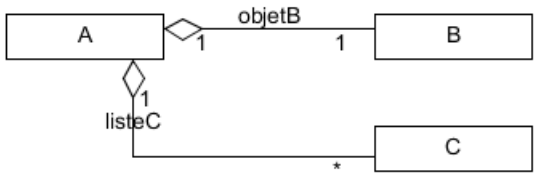
Le lien de Composition est un lien très fort (le plus fort de tous). Il lie deux instances d'objet entre eux. On dit que A est composé de B et C (plusieurs instances de C) :



B est C sont des sous-objets de A. Ils font parti intégrante de sa définition.
Quand A est créé, B et C sont créés.
Quand A est détruit, B et C sont détruits.

Exemple : A = Camion B = Moteur C = Roue

Le lien d'Agrégation est aussi un lien de composition mais moins fort que le précédent car la destruction de A n'entraine pas la destruction de B et C.



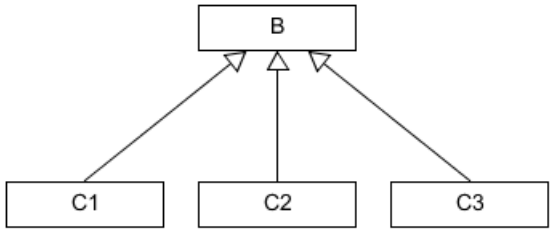
A est bien composé de B et C (plusieurs instances de C).

On ne sait pas dans cette description comment B et C sont créés. B est par exemple passé en paramètre du constructeur de A ou en utilisant le setteur de B. Une instance de C est passé en paramètre d'une méthode de A qui permet d'ajouter une instance de C à A.

Exemple : A = Entreprise B = Patron C = Camion

2.2. Héritage

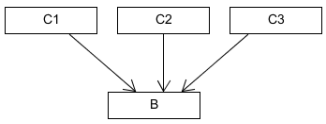
L'héritage de classe est le lien qui justifie toute l'approche des langages à objet.
On dit que les classes C1, C2 et C3 héritent de B :



C1 "est un" B.

On utilise l'héritage de classe simple (à opposer à la notion de classe abstraite que nous verrons plus loin) pour 3 raisons :

- **Factoriser** du code. B contient les méthodes communes à C1, C2 et C3. Cela correspond à la notion de couches de librairies dans la programmation classique (non objet) :

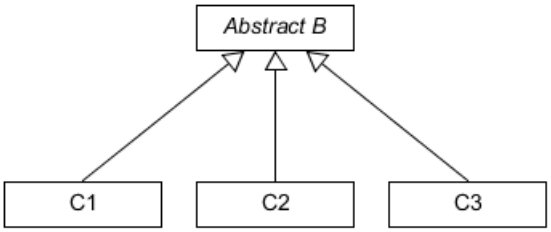


- **Etendre** (extends) les propriétés de la classe B. Cela correspond à la notion d'héritage la plus courante. On ajoute dans les classes C1, C2 et C3 de nouvelles méthodes et de nouveaux attributs qui s'ajoutent à la classe B sans impacter cette dernière.
- **Spécialiser** la classe B en redéfinissant une partie de ses méthodes dans les classes C1, C2 et C3.

Il est à noter que ces 3 raisons ne sont pas exclusives. Elles peuvent se cumuler.

2.3. Classe abstraite et interface

La classe Abstraite est une classe comme une autre (tout ce qui a été dit précédemment sur l'héritage s'applique ici) mais il n'est pas possible de créer une instance d'une classe abstraite (new).



Il faut créer des instances de C1, C2 et C3 qui héritent de B.

Toutes les méthodes abstraites (sans code) de B doivent impérativement être définies dans les classes C1, C2 et C3 (sinon erreur de compilation).

Ainsi, une méthode dans le programme qui utilise en paramètre une classe B utilisera les méthodes abstraites sans savoir à priori quelles seront les méthodes réelles de C1, C2 ou C3 qui seront utilisées dans l'algorithme de cette méthode.

On parle de traitement générique.

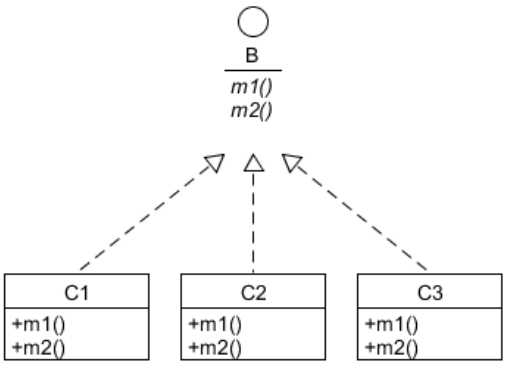
Cette notion est primordiale dans la conception d'un programme objet. Elle permet d'abstraire sa conception et donc d'être générique dans sa conception.

Par définition, un DP est un concept générique.

Une classe abstraite peut contenir à la fois des méthodes abstraites et des méthodes réelles étant donné qu'elle contient ses propres attributs et donc ses propres traitements.

Dans le cas où la classe abstraite ne contient aucun attribut et ne contient que des méthodes abstraites, on parle d' **Interface**.

La notion d'interface est symbolisée en UML ainsi :



On dit que les classes C1, C2 et C3 **implémentent** l'interface B.

Par définition, les classes C1, C2 et C3 doivent obligatoirement implémenter toutes les méthodes de l'interface.

Une classe peut implémenter plusieurs interfaces.

2.4. Association vs héritage

Souvent, il n'est pas facile de faire son choix entre un lien d'association et un lien d'héritage.

Il faut pouvoir répondre aux questions :

- "X est composé de Y" ➔ Composition
- "X est un Y" ➔ Héritage
- "X s'appuie sur Y" ou "X utilise Y" ➔ Agrégation

Nous verrons comment on peut implémenter un lien d'héritage par un lien de composition.

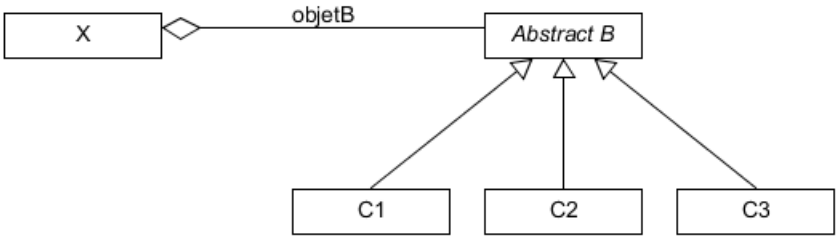
Beaucoup de langages ne font pas d'héritage multiple. Il est donc nécessaire de remplacer l'héritage par des liens de compositions ou par des interfaces car une classe peut implémenter plusieurs interfaces en même temps.

2.5. Les techniques d'association

Il existe de nombreuses façons de réaliser une association basée sur une classe abstraite ou une interface (dans les exemples qui suivent on utilise une classe abstraite mais cela est tout à fait valable sur une Interface).

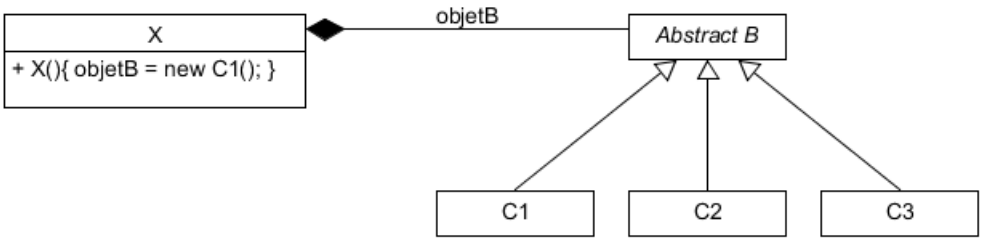
Une association explicite dans le programme :

```
C1 c1 = new C1() ;
[…]
X x = new X(c1) ;
```



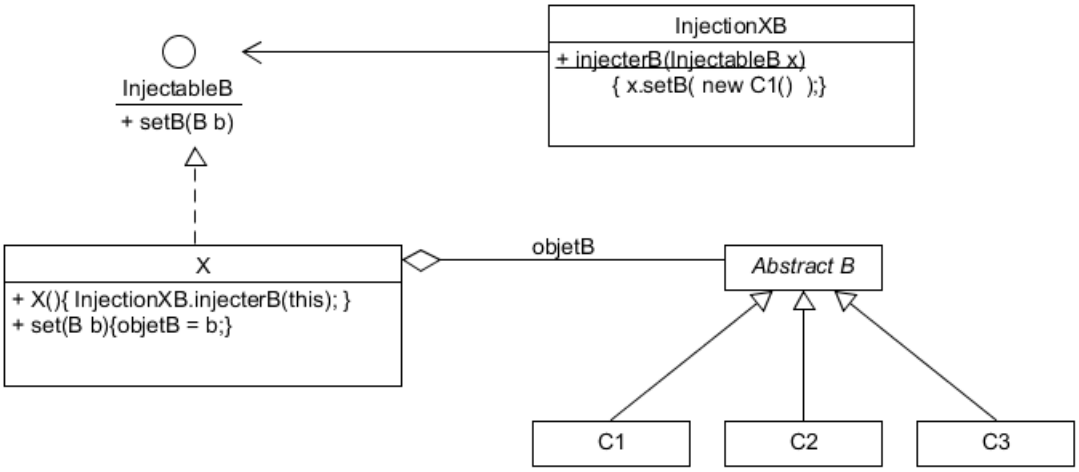
Une association explicite dans le constructeur :

```
X x = new X()
```



Une association implicite dans le constructeur. On parle d'injection de dépendance (nous verrons cela plus loin en détail) :

```
X x = new X()
```



Ici, la classe X délègue à une autre classe (l'injecteur de dépendance) de faire l'association entre lui-même (this) et une classe qui hérite de B (ici C1) à travers une méthode statique afin d'accentuer l'indépendance de X par rapport aux autres classes. De plus, grâce à l'utilisation de l'interface InjectableB, il n'existe pas de dépendance entre InjectionXB et X.

2.6. Les principes d'instanciation

Dans un programme informatique (qu'il soit réalisé avec un LOO ou non), il existe deux raisons (toujours revenir au besoin 😊) pour créer un objet.

Soit on crée un objet afin de l'utiliser (lui physiquement et pas un autre) dans tout ou une "partie" du programme. Soit on crée un objet parce qu'il est naturellement la décomposition d'un autre objet.

La deuxième raison ne pose pas de difficulté de compréhension et est directement la suite naturelle de la programmation structurée classique issue de la réflexion et du besoin de composer ses données.

La première raison, par contre, nécessite que l'on s'y arrête un moment.

A quel moment doit-on créer cet objet ? Bien avant son utilisation ou au moment de son utilisation ?

Comment accède-t-on à cet objet (comment on obtient sa référence) ? Par passage en paramètre dans tous les constructeurs et/ou méthodes traversés ? Globalement ?

A quel moment doit-on le détruire ? et le recréer ?

Comment accéder à cet objet depuis l'extérieur du programme (réseau) ?

S'il existe plusieurs occurrences de cet objet, comment peu-on gérer l'ensemble de ces occurrences ?

De plus, comparons deux techniques permettant de créer l'instance d'une classe :

```
TypeObjet obj = new TypeObjet(paramètres)
```

```
TypeObjet objet = TypeObjet.getInstance(paramètres);
// TypeObjet.new(paramètres)
```

avec

```
static public TypeObjet getInstance(paramètres) {
…….
   return new TypeObjet(paramètres); }
```

Les deux techniques sont strictement équivalentes.

```
OU

TypeObjet objet = f.getInstance(paramètres);
// TypeObjet.new(paramètres)
```

La première technique ne répond pas aux questions posées alors que la deuxième technique permet de répondre à toutes ces questions puisque l'on crée un point d'entrée commun d'instanciation sur laquelle on a la main.

Il faudrait donc toujours créer des objets de référence avec la deuxième technique.
Cela signifie que conceptuellement il faut toujours avoir en tête cette deuxième façon de faire.
Cette approche se traduit par la mise en œuvre de différents Designs Patterns et composition de Designs Patterns permettant sa création et son utilisation (Singleton, Factory, Builder, Injection, Stratégie, Proxy, Interface, ….)

Cela démontre que les DP ne sont pas avant tout une démarche d'architecture, de réutilisabilité, ou de factorisation mais aussi une démarche de développement au plus bas niveau.

3. Définitions

3.1. Le « Patron »

Les "patrons de conception" ou "Design Patterns" sont des modèles standard et réutilisables de conception de la solution à la problématique de la réalisation d'une partie d'un logiciel informatique.

"Ensemble de règle (définition d'éléments, principes de composition, règles d'usage) permettant de répondre à une classe de besoins spécifiques dans un environnement donné."

Dans une démarche de conception, quand on crée des programmes informatiques, on retrouve souvent des démarches identiques (même spécifiques à votre SI) qui se traduisent par des assemblages de composants informatiques semblables.
Certains de ces assemblages ou architecture semblables ont été modélisés et forment un ensemble de modèles qu'il faut apprendre à connaître car pourquoi réinventer la roue à chaque fois.

De la même manière que nous avons des algorithmes types (modèle de code ou méthode générique) qui reviennent souvent, pour modéliser les traitements informatiques, nous avons les design patterns pour les assemblages des classes d'objet.

Ainsi, là où un algorithme s'attache à décrire d'une manière précise comment résoudre un problème particulier, les patrons de conception décrivent des procédés de conception généraux et permettent en conséquence de mieux **capitaliser** l'expérience appliquée à la conception logicielle. Il a donc également une grande influence sur l'architecture logicielle d'un système.

L'objectif visé est la **réutilisation** des moyens de conception logicielle afin de réduire les coûts d'ingénierie et de garantir un bon niveau de qualité.

Propriétés d'un PATRON :

- Un patron est élaboré à partir de l'expérience acquise au cours de la résolution d'une classe de problèmes apparentés. Il capture des éléments de solution communs.
- Un patron définit des principes de conception, non des implémentations spécifiques de ces principes
- Un patron fournit une aide à la documentation, par ex. en définissant une terminologie, voire une description formelle (‘langage de patrons »)

Un design pattern est défini par :
- un nom
- une description du problème à résoudre
- une description de la solution : le patron de conception (schémas UML par ex) + texte explicatif.

Certains patrons sont un assemblage d'autres patrons de plus bas niveau. Cela signifie bien que ces patrons constituent bien une boite à outils de conception.
Par exemple : le design pattern MVC (Model Vue Controller) utilise les design patterns Observateur, Stratégie et Composite.

Les patrons de conception les plus connus sont plus d'une vingtaine.

On distingue 3 familles de patrons de conception selon leur utilisation :
- de **construction** (ou **création**) : ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- **structuraux** : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
- **comportementaux** : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

(En jaune les DP qui sont développés dans ce cours)

Création :

- Moniteur	Builder
- Fabrique	Factory
- Prototype	Prototype
- Singleton	Singleton

Structure :

- Interface (ou contrat)	Interface
- Délégateur	Delegate
- Inversion de contrôle	IoC
- Injection de dépendance	Injection
- Adaptateur	Adapter
- Pont	Bridge
- Objet composite	Composite
- Décorateur	Decorator
- Façade	Facade
- Poids-plume	Flyweight
- Proxy	Proxy

Comportement :

- Chaîne de responsabilité	Chain of responsibility
- Commande	Command
- Interpréteur	Interpreteur
- Itérateur	Iterator

- Médiateur	Mediator
- Mémento	Memento
- Observateur	Observer
- Etat	State
- Stratégie	Strategy
- Patron de méthode	Template Method
- Visiteur	Visitor
- Fonction de rappel	Callback == Observer == Listener
- MVC	MVC

3.2. Le « Canevas » (ou "Templates")

Un canevas est un squelette de code représentant l'implémentation de un ou plusieurs patrons.

Dans les langages objets :

- Un canevas est un « squelette » de plusieurs classes qui peuvent être réalisées et adaptées pour une famille d'applications données
- Il est un ensemble de **classes**, souvent abstraites, devant être adaptées par exemple par surcharge.

Généralement ; on accompagne les canevas d'un ensemble de **règles d'usage** décrites dans une documentation.

3.3. En conclusion

- les patrons et canevas sont deux techniques de **réutilisation**
- les patrons réutilisent un schéma de **conception** ; les canevas réutilisent du **code**.

Notre objectif n'est pas de décrire exhaustivement tous ces patrons mais d'explorer les plus courants. Et à travers ces explorations d'apprendre à lire et manipuler les design patterns mais aussi à assimiler les concepts d'architecture des langages objets.

Il est entendu qu'un minimum de connaissance en UML est nécessaire pour lire les diagrammes de description des design patterns.

Pour quelqu'un qui découvre la programmation objet, il pourrait paraître inutile d'utiliser une telle démarche dans la réalisation de ces premiers programmes.

C'est vrai que dans un premier temps, on préfère se consacrer aux bases de la programmation objet avant de parler réutilisation, conception, architecture ou performance.

Très vite, le programmeur est un concepteur et donc très vite, il est confronté à faire des choix de conception pour lesquels les design patterns sont des éléments de sa conception.

Il ne faut donc pas négliger la connaissance de ces design patterns.

C'est pourquoi, dans le cadre de la présente formation, nous allons aborder certains design patterns parmi les plus connus.

4. Les modèles de création

Ce qui guide ce type de modèle est de pouvoir faire la création d'objet (instanciation) sans être nécessairement impacté par l'évolution de la classe et de ses constructeurs.

4.1. Le singleton

4.1.1. Nom et rôle

Singleton

Le rôle de ce design pattern est la création d'une instance d'objet unique dans l'Application (la JVM).

L'objectif est de limiter le nombre d'instance d'une classe qui dans le cas d'un singleton est toujours égal à 1.

L'usage d'un singleton est principalement de pouvoir accéder à un objet n'importe où dans le code (sans avoir besoin de le passer en paramètre ou en attribut d'un objet).

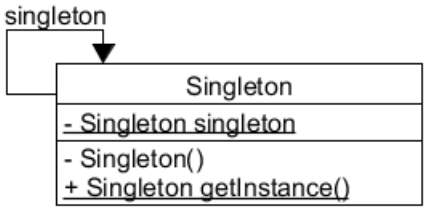
Un singleton est l'équivalent strict de ce que l'on appelle dans d'autre langage : une variable globale.

Dans la programmation classique, on connaît que trop l'utilisation nocive des variables globales qui entraîne des effets de bord indésirables et souvent désastreux.
En programmation objet, la situation est différente car la variable globale est avant tout un objet. Et cela change tout, car si on ne définit aucun setteur, l'objet ne sera jamais modifié et donc plus d'effet de bord.
Si on a, quand même, besoin de changer certains attributs de l'objet global (assez rare), on peut le limiter à des attributs bien précis qui ne pourront pas occasionner des effets de bord désastreux et contrôler leurs changements via les setteurs.

Il nous reste à déterminer comment on accède à un objet globale en PO : l'attribut statique d'une classe.

L'accès à un tel objet se fait donc par l'appel d'une méthode statique.

4.1.2. Description de la solution



Il existe deux implémentations possibles :
- soit l'instance unique est créée lors de la définition de la classe (CAS 1)
- soit l'instance unique est créée lors de la 1ère utilisation du singleton (CAS 2)

Dans le CAS 1, la méthode **getInstance** retourne l'instance.
Dans le CAS 2, la méthode **getInstance** retourne la création du singelton la 1ère fois mais on fera le nécessaire pour que si on appelle une seconde fois la méthode *getInstance,* au lieu de créer un nouvel objet, la méthode retournera l'objet précédemment créé.

4.1.3. Exemple de Singleton

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **ExempleCh04_02_SingletonCAS1**

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **ExempleCh04_02_SingletonCAS2**

4.1.4. Utilisation du DP

On rencontre ce DP très souvent, mais hélas trop souvent !

Sans l'utilisation du singleton, dans la programmation objet, pour utiliser la référence d'un objet dans un traitement (méthode), il faut que cette référence ait été créée par la classe (association de composition) ou passée en paramètre du constructeur (association d'agrégation) ou passé en paramètre de la méthode concerné, et donc passé en paramètre du constructeur ou créée par la classe appelante de la méthode, ce qui revient au même.

Cela pouvant être trop contraignant, vu la profondeur éventuelle d'appel, on utilise un singleton. Il ne faut pas tomber dans cette facilité.

Seul le caractère unique de l'objet doit être l'élément déclencheur.

Le principe du singleton est que, en n'importe quel endroit du programme, on peut faire appel au singleton. Ainsi, si on veut revenir en arrière, il faudra modifier partout où il est utilisé.

Quand on veut créer un service au sein de son programme, et que ce service est par définition unique alors on crée un singleton :

- Le Contrôleur d'un modèle MVC
- Le proxy de communication à un serveur de traitement externe
- Un gestionnaire de données métiers
- Le traitement de persistance en base de ses données
- Un repository d'évènement pour des traitements asynchrone
- Le service de temps (ntp)
- La création des sémaphores de synchronisation
- Le système de log de trace et d'erreur
- Un annuaire de communication
- … etc …

4.2. Factory ou Fabrique de création

4.2.1. Nom et rôle

Factory ou Usine de fabrication

Une telle usine fabrique des objets, à la demande, au sens des LOO.

L'objectif : abstraire la façon de créer (en mémoire) les objets

Dans un système d'information (SI), il existe des objets particuliers qui sont des données récurrentes devant être créées de nombreuses fois durant le fonctionnement du programme, et ceci à de multiples endroits du programme.

Ces données sont des objets métiers (dans une architecture Client-Serveur, on parle de Business Object). Ces objets sont souvent stockés ou persistants dans une base de données). Le factory assure alors les propriétés liées à la base de données.

Il y a donc un besoin de centraliser cette création d'objet mais aussi d'être le moins impacté possible si la classe de ces objets évolue.

4.2.2. Description de la solution

La solution est de créer une classe qui va servir de point central de création de l'objet.

a. Variante de base

Une usine (ou factory) est une classe qui contient des méthodes de création d'objet qui reposent sur les principes suivants :

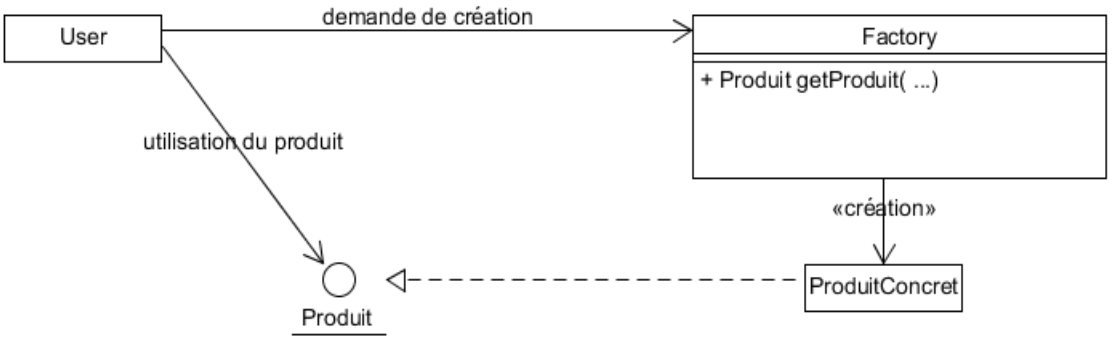
1/ Chaque méthode de création retourne une interface donnée (vision abstraite)

2/ Chaque méthode de création exploite un constructeur d'une classe (vision concrète)

3/ Cette classe concrète implémente les méthodes de l'interface

L'adhérence de l'évolution d'un factory est donc restreint :

- aux paramètres de la méthode de création (nous verrons plus loin que le DP de Décorateur permet de limiter cette adhérence)
- à l'interface du Produit.



Le factory retourne l'objet qui est vu par le User sous l'aspect d'une interface **Produit.**

```
Factory     f = new Factory() ;

Produit p = f.getProduit(…);
```

Cet objet (p) est une instance de la classe **ProduitConcret** qui implémente l'interface **Produit**.

Dans cette variante de Factory, les produits ne sont pas stockés en mémoire fonctionnel.

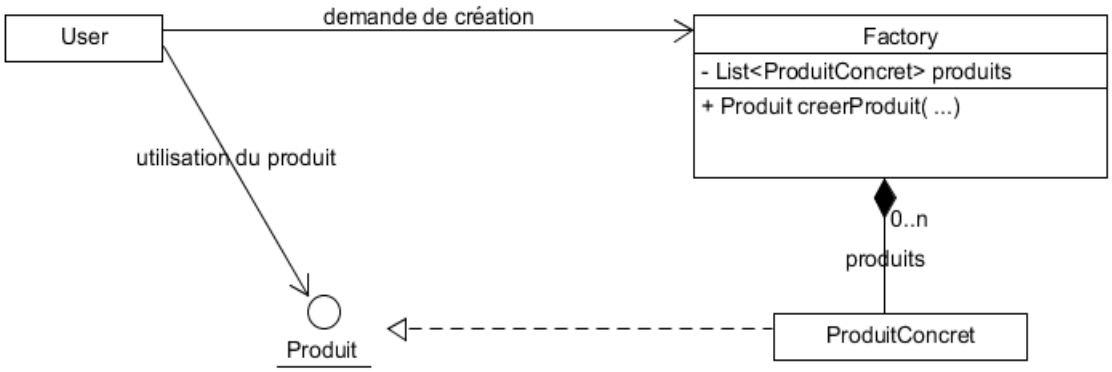
On parle ici de « mémoire fonctionnel » à opposer à la mémoire du langage de programmation.
En Java, le produit retourné est un pointeur mémoire. Le produit n'est pas stocké dans une collection mais dans le « collector ».
Dans d'autres langages, comme le C++, le produit retourné n'est pas un pointeur mémoire mais l'objet complet (stocké en mémoire). La destruction de l'objet est à la charge de l'utilisateur du Factory.

Exemples :

- un Factory de numéro de carte de d'immatriculation
- Un Factory de date calendaire
- Un Factory de connexions à des serveurs répartis et/ou de répartition de charge
- …

b. Variante avec Collection

Une usine (ou factory) est une classe qui contient des méthodes de création d'objet qui reposent sur les principes suivants :
1/ Chaque méthode de création retourne une interface donnée (vision abstraite)
2/ Chaque méthode de création exploite un constructeur d'une classe (vision concrète)
3/ Cette classe concrète implémente les méthodes de l'interface
4/ **L'interface gère l'objet situé dans le factory**. Comme nous le verrons plus loin, si le factory est distant, l'interface du Produit sera une interface distante (Proxy de communication).
5/ Les produits sont stockés dans le Factory.



Une classe utilisatrice **User** fait une demande de création d'un objet à un **Factory**.
Le factory retourne l'objet qui est vu par le User sous l'aspect d'une interface **Produit.**

```
Factory     f = new Factory() ;
```

```
Produit p = f.creerProduit(…);
```

Cet objet (p) est une instance de la classe **ProduitConcret** qui implémente l'interface **Produit**.

Deux sous- variantes :

- si le produit existe déjà dans le Factory, le produit n'est pas recréer, et l'interface retournée (pointeur) gère le même produit
- sinon un produit est créé à chaque demande de création.

Dans le premier cas, la méthode est toujours « getProduit », dans le deuxième cas, la méthode est « creerProduit ». (Affaire de sémantique).
Dans le premier cas, les paramètres de création sont des clefs fonctionnelles.

Dans un tel Factory, les produits étants stockés, il est possible de définir des méthodes **d'accès** et de **recherche** des produits dans la collection :

- Rechercher un objet sur des **critères**.
- Accéder à un objet à travers un **index** ou des **clefs fonctionnels.**

Dans tous les cas, les produits retournés sont des interfaces de Produit.

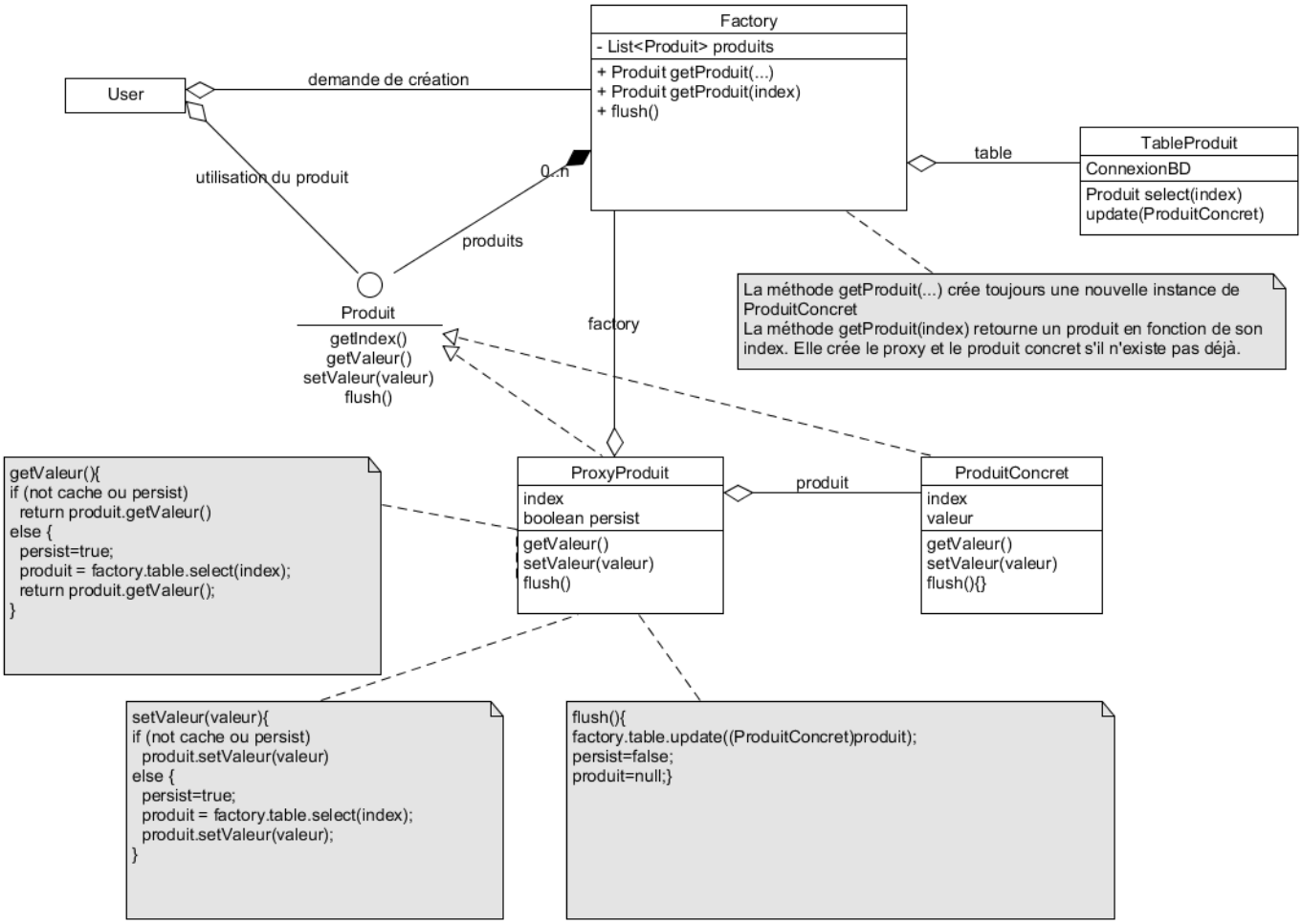
Au-delà de la création, le factory définit des méthodes de gestion des produits : recherche, accès, indexation, comparaison, tri, modifications, …

On peut aussi avoir des méthodes du factory comme :
- Produit search(….)
- void removeAll()
- void Produit getProduitClone(…) // si le produit n'a pas d'adhérence dans le factory
- void **ProduitWrite** getProduitClone(…) // si le produit n'a pas d'adhérence dans le factory
- void add(ProduitWrite p)
- … etc..

c. Un Factory de données stockées en base de données

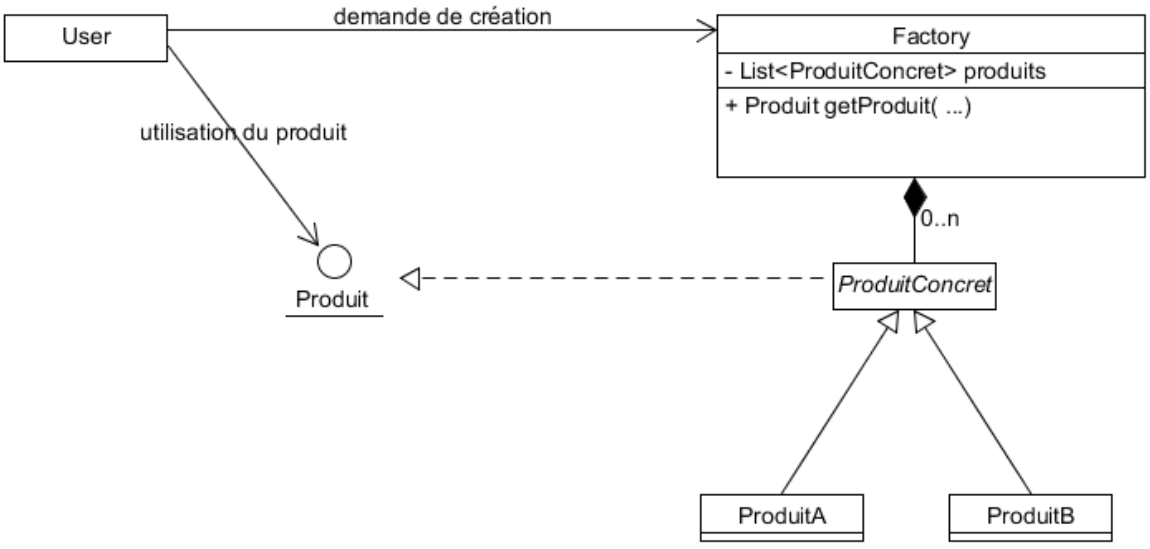
Si les données d'un Factory doivent être stockés en base de données, le factory va servir de **cache** pour optimiser l'accès aux données.

Ci-après une proposition parmi tant d'autres répondant à ce nouveau besoin.
Il existe de nombreux DP réalisés dans le cadre des DAO (Data Acces Object).



COMMENTER EN COURS

d. Une classe abstraite du Produit : plusieurs types de produits



Il existe des Factory qui créent des objets concrets de différentes natures en fonction des paramètres de la méthode getProduit différents du user.

Dans ce cas le factory crée des objets concrets de différentes natures dont les classes d'appartenance héritent d'une classe abstraite ProduitConcret qui implémente l'interface des produits Produit. Ceci pour que le User perçoit chacun des produits d'une manière unique.

Le factory abstrait ainsi les choix de conception et d'implémentation des objets créés.

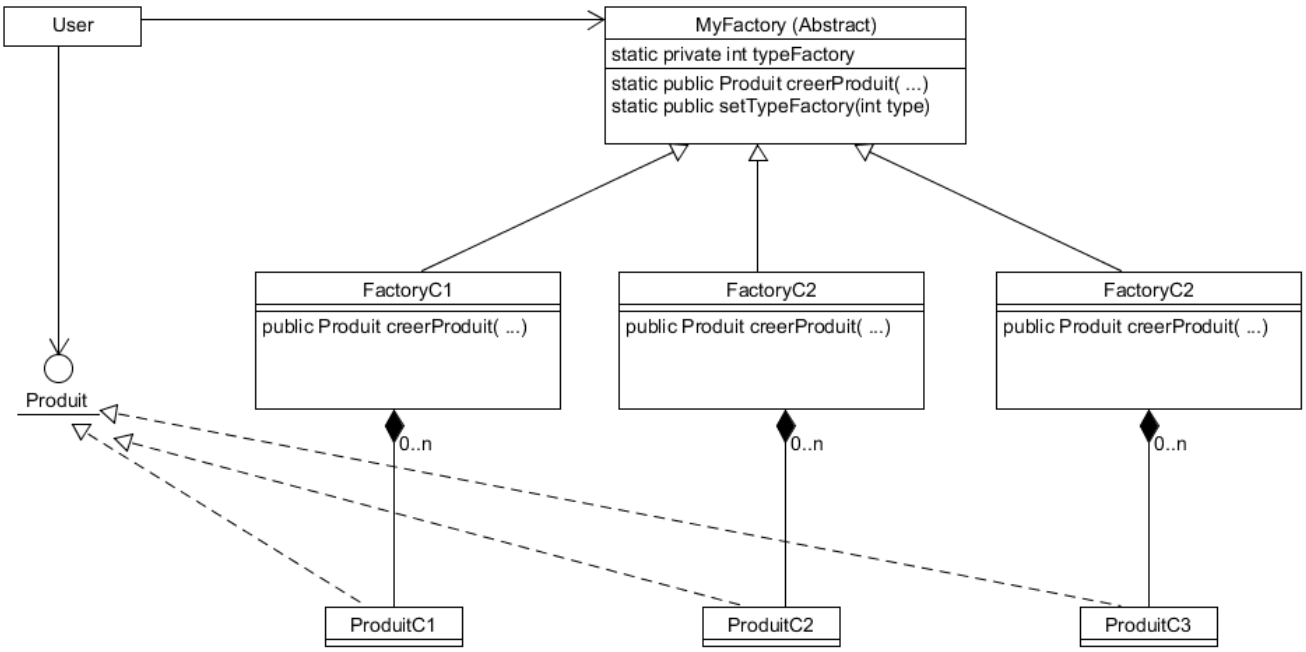
La classe abstraite est utilisée pour créer une **collection polymorphe** dans le factory.

Cette variante s'adapte aussi aux variantes précédentes.

e. Une classe abstraite de Factory

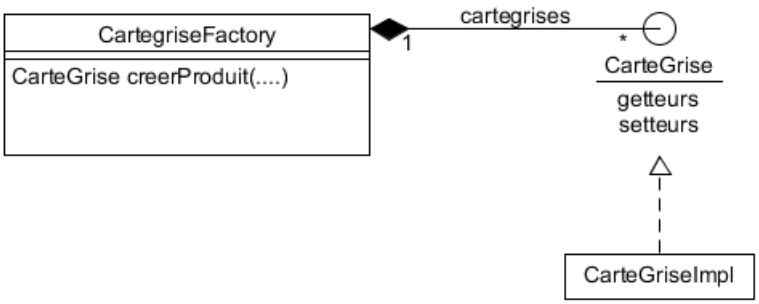
La problématique : quand il existe des algorithmes indépendants pour créer les produits. Par exemple créer des objets de DAO (Data Acces Object) en fonction de différents types de bases de données.

Le principe : le factory est une classe abstraite qui implémente une interface contenant la méthode de création des objets. Des classes de factory concrètes héritent de la classe abstraite. Le choix de l'algorithme se fait par l'utilisation d'une méthode statique de la classe abstraite.

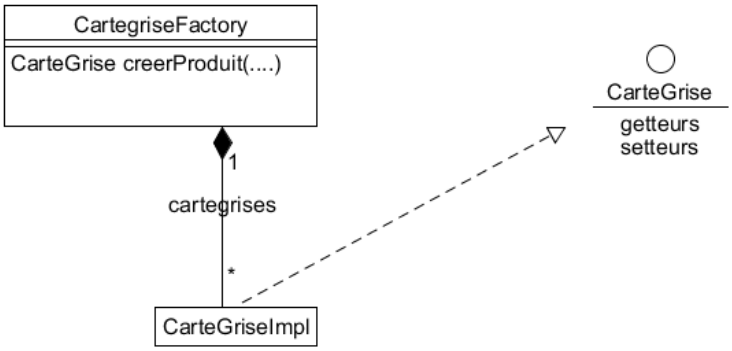


4.2.3. Exemple de Factory

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **ExempleCh04_01_Factory**



ou



Commentaires en cours

4.2.4. Utilisation du DP

Le DP du Factory est incontournable dans la conception d'une application informatique car dès que l'on doit gérer une "collection" d'informations que l'on veut gérer en mémoire de l'application, on peut créer un Factory.

Le Factory offre une sécurité des données, il gère le cycle de vie de ces données.

Il est souvent utilisé en interface d'une base de données pour laquelle on veut optimiser l'accès à certaines données. Il sert alors de système de cache.

Il permet un accès direct à certaines données à travers un réseau informatique en utilisant le DP de Proxy de communication.

Il permet la centralisation unique des données et est donc souvent associé au DP Singleton. Il facilite les moyens de recherche, d'interrogation d'information.

Par exemple, il gère les EJB Session dans les conteneurs de l'architecture JEE, les files de messages dans les architectures MOM, les canaux de communication, les annuaires, … etc …

Il sert de structure d'accueil à l'implémentation du système de mapping entre des données et la base de données (ex hibernate).

La liste serait longue d'énumérer tous les exemples d'utilisation d'un factory.

En résumé : Gestion de données => factory
et comme les données sont incontournables dans un SI ….

4.3. Le builder

4.3.1. Nom et rôle

Builder (ou Monteur).

Le rôle de ce design pattern est la construction d'un objet par assemblage.

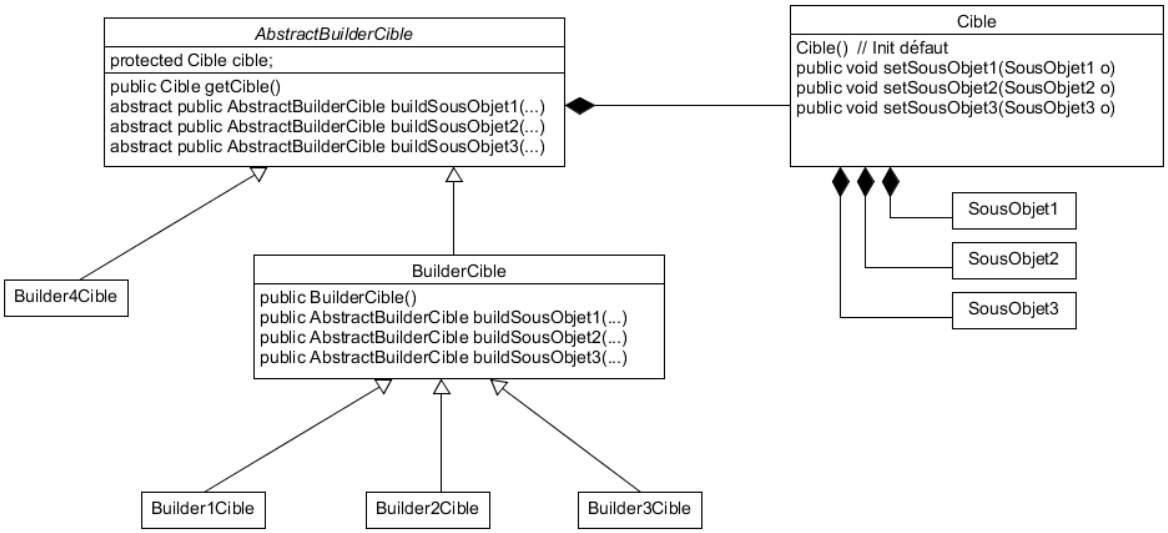
L'objectif est d'éviter la définition de nombreux constructeurs qui ont de plus en plus de paramètre et/ou d'éviter de passer en paramètre d'un long constructeur des valeurs "vides".

Essentiellement, le builder est utilisé quand l'objet à créer est composé de sous-objets.

La problématique est de pouvoir renseigner l'objet avec une partie de ces sous-objets et que la partie complémentaire de ces sous-objets soit initialisés avec des valeurs par défaut.

Remarque : le DP builder est souvent utilisé dans les factory puisque le rôle d'un factory est de créer un produit souvent complexe.

4.3.2. Description de la solution



Une classe abstraite crée l'objet cible "vide" en utilisant un constructeur par défaut.

Cette classe abstraite définit autant de méthodes abstraites (méthodes d'assemblage) qu'il existe de sous-objet qui prend en entrée les attributs de chacun des sous-objet et qui initialise l'objet cible.

On crée une classe builder concrète qui implémente les méthodes d'assemblage.
Cela permet de créer des variantes de l'assemblage en créant d'autres classes concrètes.

Chacune des méthodes d'assemblage retourne le builder lui-même, cela permet d'enchainer l'appel des méthodes d'assemblage dans une même instruction.

```
Builer3Cible b1 = new Builder3Cible() ;
```

```
b1.b……
```

4.3.3. Exemple de builder

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **ExempleCh04_03_DPBuilder**

```
// Exemple du DP Builder
//
import java.util.*;

public class Exemple20
{
    public static void main(String[] args)
    {
        RequeteSQL req1 = new BuilderRequeteSQL()
            .buildTypeSQL("SELECT")
            .addSelectSQL("nom")
            .addSelectSQL("prenom")
            .buildFromSQL("PERSONNE")
            .getRequeteSQL();
        System.out.println(req1);
        System.out.println("-------------------------");

        RequeteSQL req2 = new BuilderRequeteSQL()
            .buildTypeSQL("SELECT")
            .addSelectSQL("*")
            .buildFromSQL("PERSONNE")
            .buildWhereSQL("nom='LAFONT'")
            .getRequeteSQL();
        System.out.println(req2);
        System.out.println("-------------------------");

        RequeteSQL req3 = new BuilderRequeteSQL_SELECT_PERSONNE()
            .buildWhereSQL("age > 20")
            .getRequeteSQL();
        System.out.println(req3);
        System.out.println("-------------------------");

    }
}
//---------------------------------------
class RequeteSQL {
    private TypeSQL typeSQL;
    private SelectSQL selectSQL;
    private UpdateSQL updateSQL;
    private DeleteSQL deleteSQL;
    private FromSQL fromSQL;
    private WhereSQL whereSQL;
    private OrderSQL orderSQL;


    public RequeteSQL()
```

```
      {
          typeSQL = new TypeSQL();
          selectSQL = new SelectSQL();
          updateSQL  = new UpdateSQL();
          deleteSQL = new DeleteSQL();
          fromSQL = new FromSQL();
          whereSQL = new WhereSQL();
          orderSQL = new OrderSQL();
      }

      public String toString()
      {
          if (typeSQL.getType().equals("SELECT"))
              return
                  selectSQL+" "+
                  fromSQL+" "+
                  whereSQL+" "+
                  orderSQL+" ";
          // ...
          return "";
      }
      public void setTypeSQL(TypeSQL w){typeSQL=w;}
      public void setSelectSQL(SelectSQL w){selectSQL=w;}
      public void setUpdateSQL(UpdateSQL w){updateSQL=w;}
      public void setDeleteSQL(DeleteSQL w){deleteSQL=w;}
      public void setFromSQL(FromSQL w){fromSQL=w;}
      public void setWhereSQL(WhereSQL w){whereSQL=w;}
      public void setOrderSQL(OrderSQL w){orderSQL=w;}

      public SelectSQL getSelectSQL(){return selectSQL;}
      public FromSQL getFromSQL(){return fromSQL;}

  }

  //----------------------------------------
  abstract class AbstractBuilderRequeteSQL {
      protected RequeteSQL requeteSQL;

      public AbstractBuilderRequeteSQL() {
          requeteSQL = new RequeteSQL();  // Par defaut
      }

      public RequeteSQL getRequeteSQL(){return requeteSQL;}

      abstract public AbstractBuilderRequeteSQL buildTypeSQL(String type);
           abstract  public  AbstractBuilderRequeteSQL  addSelectSQL(String
colonne);
         abstract  public  AbstractBuilderRequeteSQL  buildFromSQL(String...
tableau);
          abstract  public  AbstractBuilderRequeteSQL  buildWhereSQL(String
where);
      abstract public AbstractBuilderRequeteSQL buildOrderSQL(String asc);
  }

  //----------------------------------------
  class BuilderRequeteSQL extends AbstractBuilderRequeteSQL {

      public BuilderRequeteSQL() {
          super();
          // Par defaut par le builder
```

```
            requeteSQL.setWhereSQL(new WhereSQL("1 = 1"));
        }

        public AbstractBuilderRequeteSQL buildTypeSQL(String type){
            requeteSQL.setTypeSQL(new TypeSQL(type));
            return this;
        }

        public AbstractBuilderRequeteSQL addSelectSQL(String colonne){
            requeteSQL.getSelectSQL().addColonne(colonne);
            return this;
        }
        public AbstractBuilderRequeteSQL buildFromSQL(String... tables){
            for(String s:tables)
                requeteSQL.getFromSQL().addTable(s);
            return this;
        }

        public AbstractBuilderRequeteSQL buildWhereSQL(String where){
            requeteSQL.setWhereSQL(new WhereSQL(where));
            return this;
        }

        public AbstractBuilderRequeteSQL buildOrderSQL(String order){
            requeteSQL.setOrderSQL(new OrderSQL(order));
            return this;
        }

    }

    //---------------------------------------
    class BuilderRequeteSQL_SELECT_PERSONNE extends BuilderRequeteSQL
    {
        public BuilderRequeteSQL_SELECT_PERSONNE() {
            super();
            buildFromSQL("PERSONNE");
            addSelectSQL("nom");
            addSelectSQL("prenom");
            addSelectSQL("age");
            buildTypeSQL("SELECT");
        }
    }

    //---------------------------------------
    class TypeSQL
    {
        private String type;

        public TypeSQL(){type="";}
        public TypeSQL(String type){this.type=type;}

        public String toString(){return type;}
        public String getType(){return type;}
    }

    //---------------------------------------
    class SelectSQL
    {
        private ArrayList<String> colonnes;
```

```
    public SelectSQL(){
        colonnes = new  ArrayList<String> ();
    }

    public void addColonne(String colonne)
    {
        colonnes.add(colonne);
    }

    public String toString()
    {
        String res="SELECT ";
        for(String s:colonnes)res=res+s+" ";
        return res;
    }
}

//---------------------------------------
class FromSQL
{
    private ArrayList<String> tables;

    public FromSQL(){
        tables = new  ArrayList<String> ();
    }

    public void addTable(String table)
    {
        tables.add(table);
    }

    public String toString()
    {
        String res="FROM ";
        for(String s:tables)res=res+s+" ";
        return res;
    }
}

//---------------------------------------
class WhereSQL
{
    private String where;

    public WhereSQL(){where="";}
    public WhereSQL(String where){this.where=where;}

    public String toString(){return "WHERE " + where;}

}

//---------------------------------------
class OrderSQL
{
    private String order;

    public OrderSQL(){order="";}
    public OrderSQL(String order){this.order=order;}

    public String toString(){return order;}
```

```
}

//---------------------------------------
class DeleteSQL{}  // A completer

//---------------------------------------
class UpdateSQL{}  // A completer
```

Résultat de l'exécution :

```
SELECT nom prenom  FROM PERSONNE  WHERE 1 = 1
-------------------------
SELECT *  FROM PERSONNE  WHERE nom='LAFONT'
-------------------------
SELECT nom prenom age  FROM PERSONNE  WHERE age > 20
-------------------------
```

4.3.4. Utilisation du DP

On veut éviter de passer par la création de nombreux constructeurs différents pour l'instanciation d'un objet complexe et éviter que l'on doive faire des *new* répartis dans tous le code, qu'il faut ensuite maintenir en cas d'évolution.

Dès que l'objet est :

- Complexe,
- Se compose de plusieurs objets,
- **Il existe différentes façons de créer l'objet**
- les valeurs par défaut des attributs sont variables

Alors, il faut utiliser le Builder.

La technique de faire un new sans paramètres d'un objet, puis de l'initialiser à coup de setteurs est une pratique ancienne, facile pas très "objet", avec des effets de bord potentiels.

Le Builder apporte un cadre de travail en encapsulant l'objet dans un "container" qui se charge d'exposer des services de construction de l'objet.

On rencontre le builder dans le singleton qui par définition n'a pas de paramètres d'instanciation. Pour un singleton complexe, le builder utilise des fichiers de configuration contenant ces paramètres, fait appel à d'autres singletons.

On le rencontre aussi dans le Factory dont le rôle est de créer des instances.

Ce DP est très utilisé pour rendre compatible des données en versions différentes d'un même objet métier qui évolue dans le programme.

5. Les modèles de structure

L'usage commun des design patterns de structure est d'utiliser les propriétés des langages objets (héritage, interface, classe abstraite, …) pour regrouper les objets et leurs appliquer ou définir des traitements sans avoir besoin de le faire explicitement sur chacun d'eux.

Par exemple, l'interface elle-même est un design patttern qui permet, de par son implémentation dans différentes classes, de créer des collections polymorphes et des traitements génériques.

Ces modèles de structures permettent aussi de concevoir des interfaces complexes d'échange et de communication entre différents composants informatiques d'un SI.

Par exemple, la séparation entre l'applicatif et son ihm via l'usage de l'interface et de proxy pour sa séparation dans une architecture client-serveur.

5.1. La classe abstraite et/ou l'interface

5.1.1. Nom et rôle

Le rôle de la classe abstraite et/ou de l'interface est de définir un contrat de comportement entre une classe utilisatrice d'un certain nombre de méthodes et une autre classe qui implémente ces méthodes.

Le problème à résoudre est de faire en sorte que l'utilisateur ne voit qu'une définition abstraite des méthodes.

L'objectif est ainsi de pouvoir :

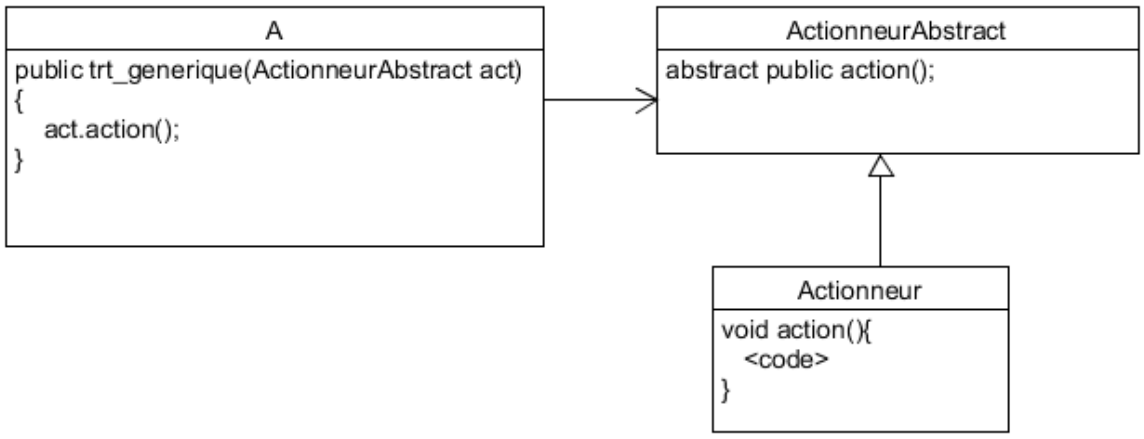
- définir des traitements génériques
- définir des données polymorphes

5.1.2. Description de la solution pour un traitement générique

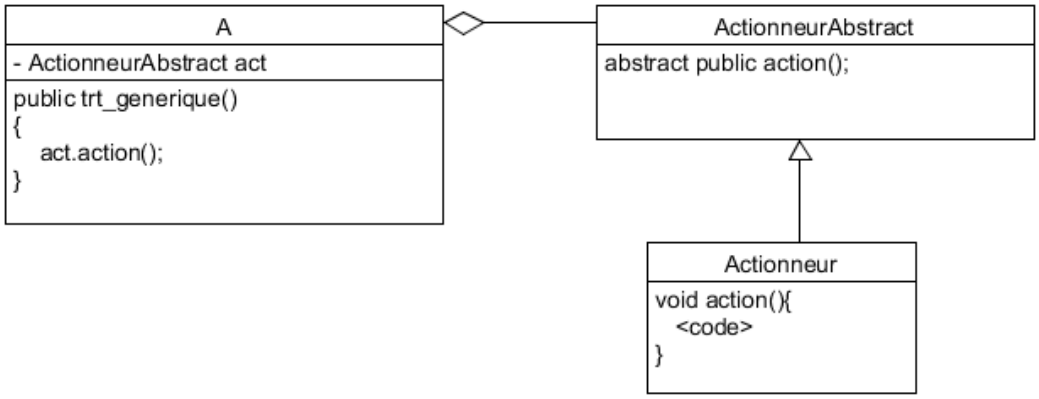
La solution pour définir un traitement générique se fait en appliquant les principes fondamentaux de la programmation objet suivants :

- un traitement est une méthode public d'un objet qui appartient à une classe
- le cast (ou conversion de type) d'un objet se fait par déférencement entre la définition réelle de l'objet et sa représentation abstraite
- l'abstraction d'un objet se fait en utilisant une classe abstraite ou en utilisant une interface (en Java).

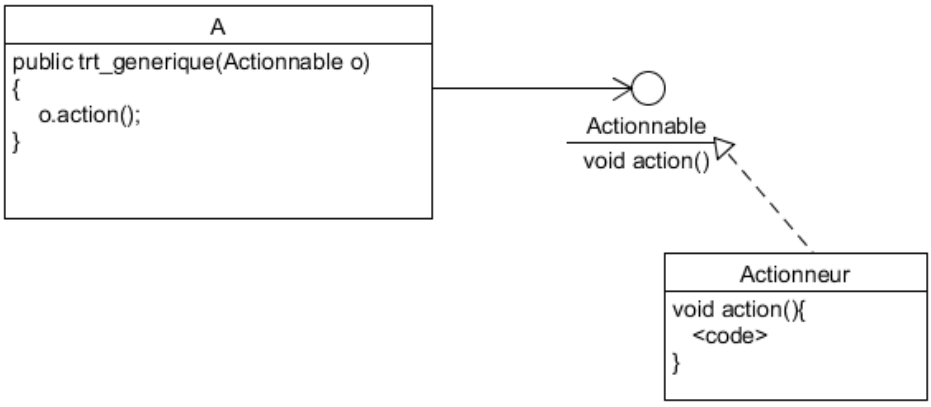
En utilisant une classe abstraite, on obtient le schéma UML de cette description suivante :



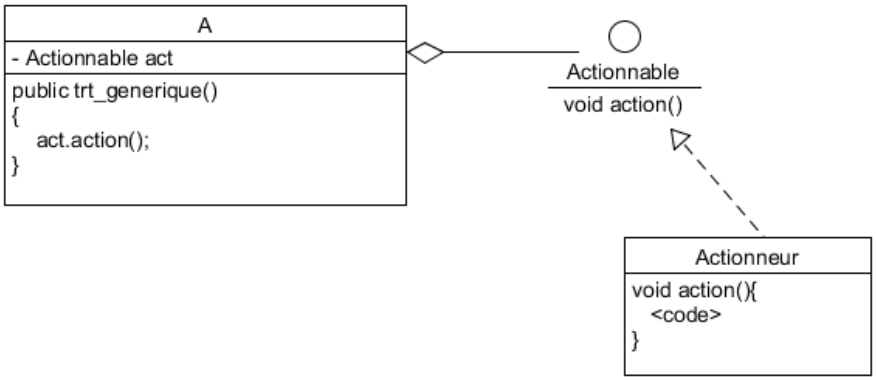
ou



En utilisant une interface, on obtient le schéma UML de cette description suivante :



ou



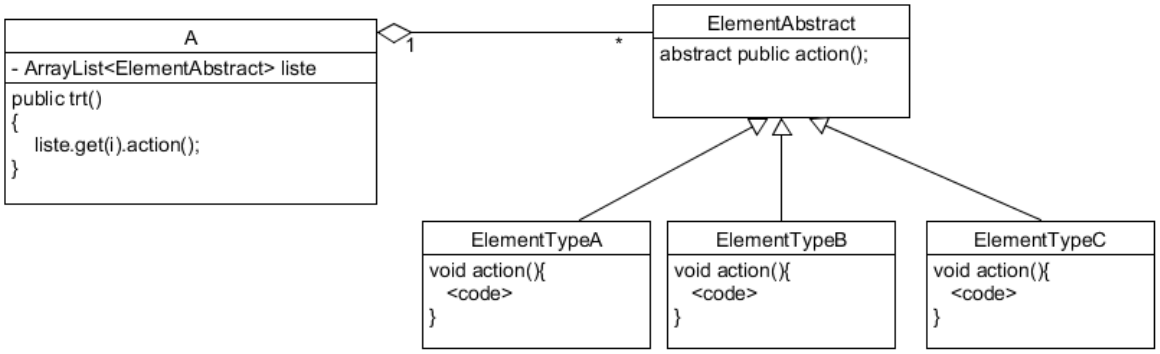
5.1.1. Description de la solution pour une donnée polymorphe

La solution pour définir une donnée polymorphe se fait aussi en appliquant les principes fondamentaux de la programmation objet :

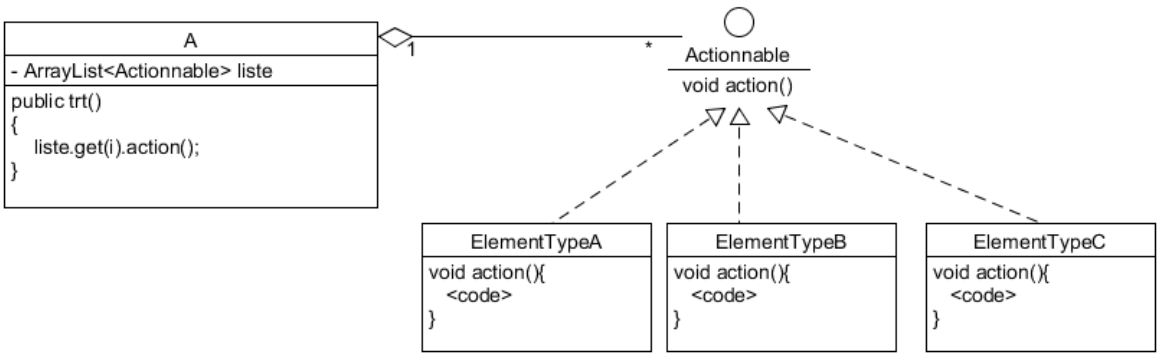
- les classes qui héritent toutes d'une certaine classe sont du type de cette classe
- la classe abstraite définit des méthodes abstraites implémentées par les classes qui en héritent
- les traitements de la donnée polymorphe utilisent les méthodes abstraites.

On obtient le schéma UML de cette description suivante :

En utilisant une classe abstraite :



En utilisant une interface :



5.2. La délégation

5.2.1. Nom et rôle

Delegate ou Délégation

Le rôle de ce DP est de déléguer à une autre classe de réaliser des traitements que la classe aurait dû implémenter.

Ce DP est :

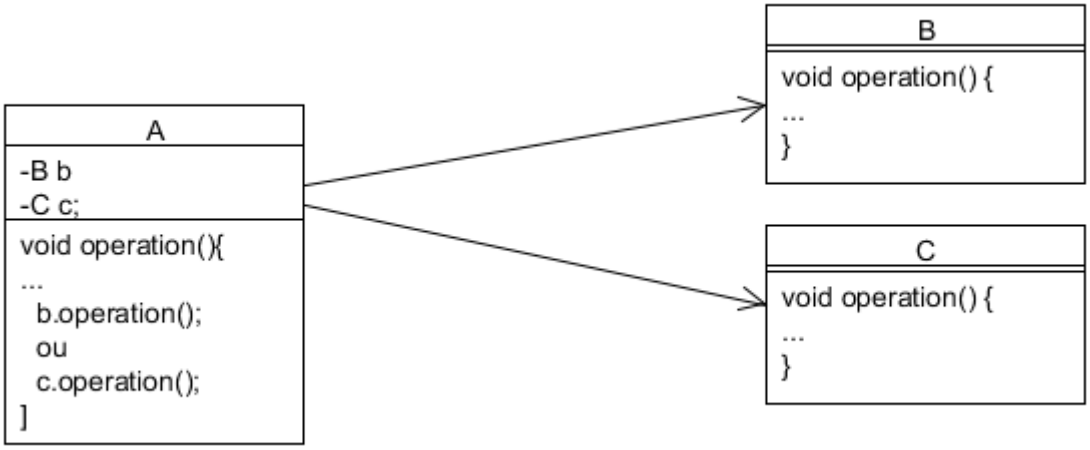
- un moyen de faire de la composition un outil de réutilisation aussi puissant que l'héritage (permet de faire de l' "héritage multiple" dans un langage qui ne le possède pas)
- un moyen qu'un objet puisse avoir différentes sous-classes à différents instants
- utilisé dans le DP d'injection de dépendance.

5.2.2. Description de la solution

a. Principe de base

Au lieu que A réalise le code de **operation()**, il délègue à un autre objet **B** de réaliser le traitement.

On a plus de puissance conceptuelle si on a plusieurs sous-classes pouvant sous-traiter la demande de traitement :

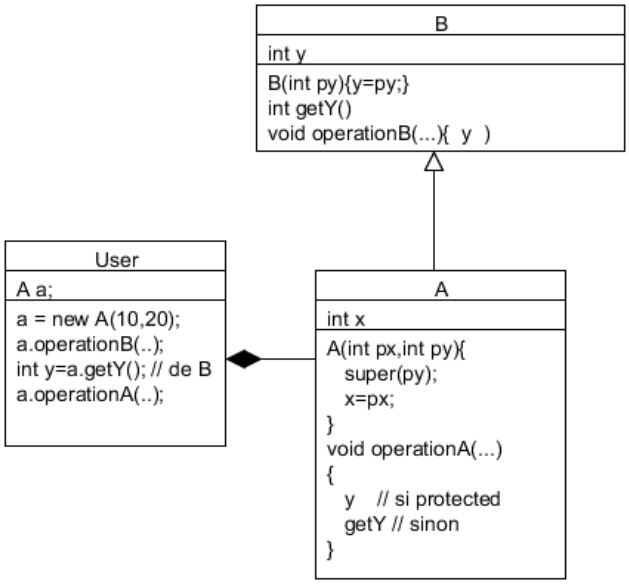


Ici, on ne précise pas comment les instances des sous-classes B et C sont créées, soit par composition, soit par agrégation.

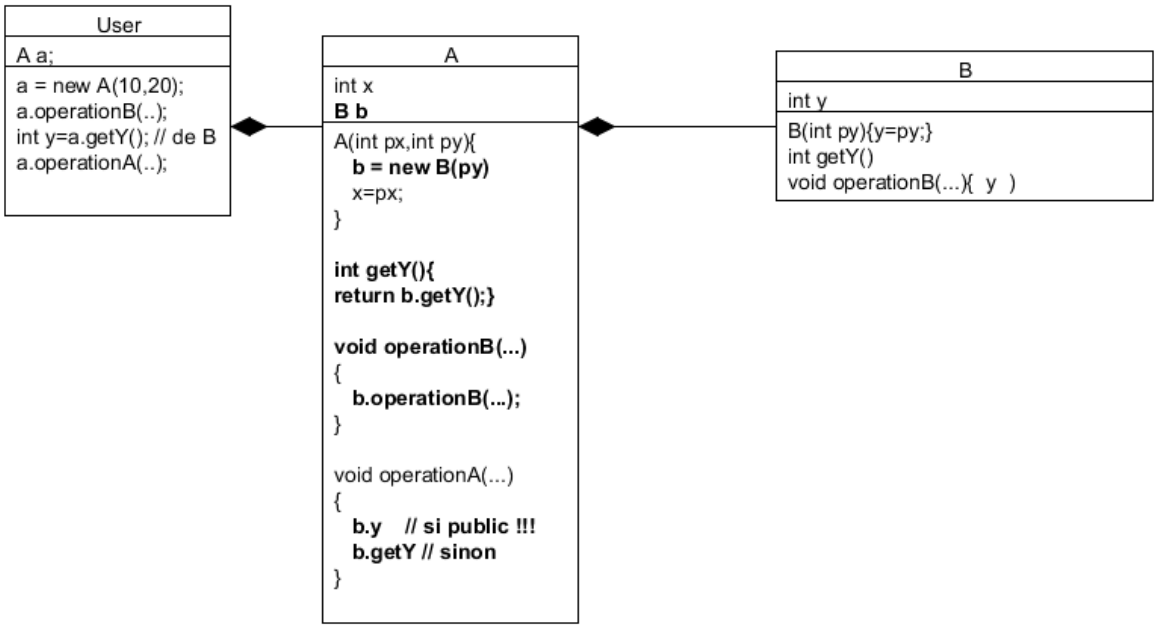
A est le délégateur et **B et C** sont les délégués.

b. Un exemple : l'héritage par délégation

Voici le DP de l'héritage :



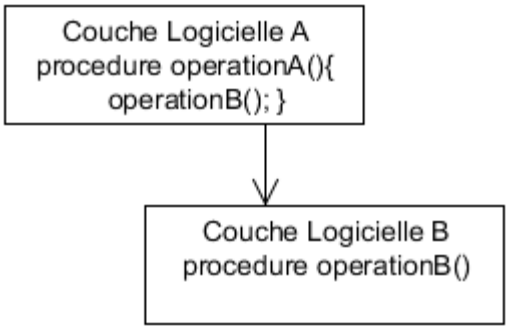
Et voici le même besoin en utilisant le DP de délégation :



5.3. L'Inversion de Contrôle

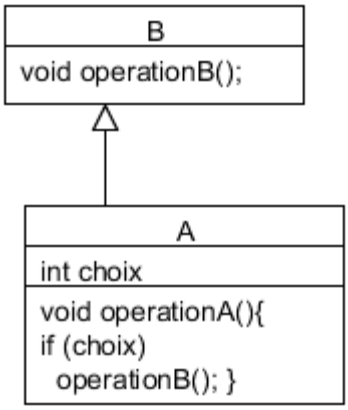
Avant toute chose le « contrôle » peut se résumer ainsi :

La couche logicielle A contrôle l'exécution de la couche B :
C'est A qui décide quand appelé B.

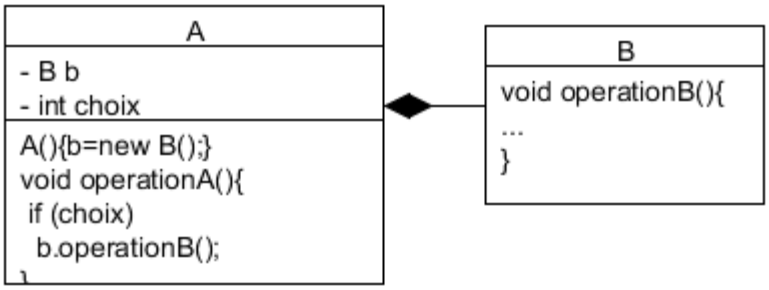


La même chose mais en programmation objet :

- Par héritage :



- Par composition :



5.3.1. Nom et rôle

L' « Inversion de Contrôle »

L'inversion de contrôle (Inversion of Control, IoC) est un patron d'architecture commun à tous les frameworks (ou cadre de développement et d'exécution).

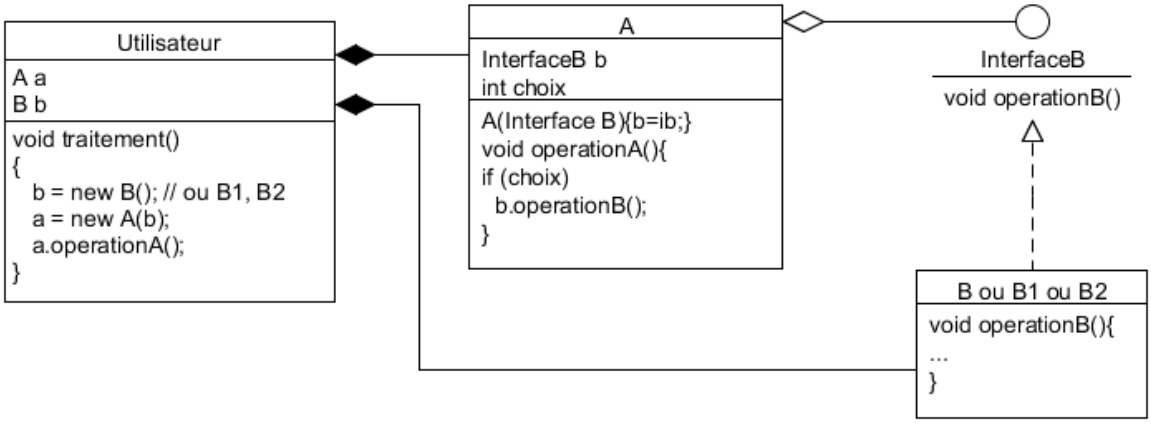
Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.

C'est B qui décide à partir de quel moment il doit être appelé par A. (Listener, Callback)
ou
C'est le Framework qui décide à partir de quel moment A appellera B (Injection de dépendance)

L'inversion de contrôle est un terme générique.
Selon le problème, il existe différentes formes, ou représentation d'IoC, le plus connu étant l'injection de dépendances qui est un patron de conception permettant, en programmation orientée objet, de découpler les dépendances entre objets.

5.3.2. Description de la solution

Une première forme d'IoC est que, au lieu que A appelle B, un « appelant » dit à A ce qu'il doit appeler. Cela passe par l'utilisation du DP Delagate et Interface.



5.4. *L'Injection de dépendance (entre 2 classes (instances))*

5.4.1. Nom et rôle

Injection de dépendance.

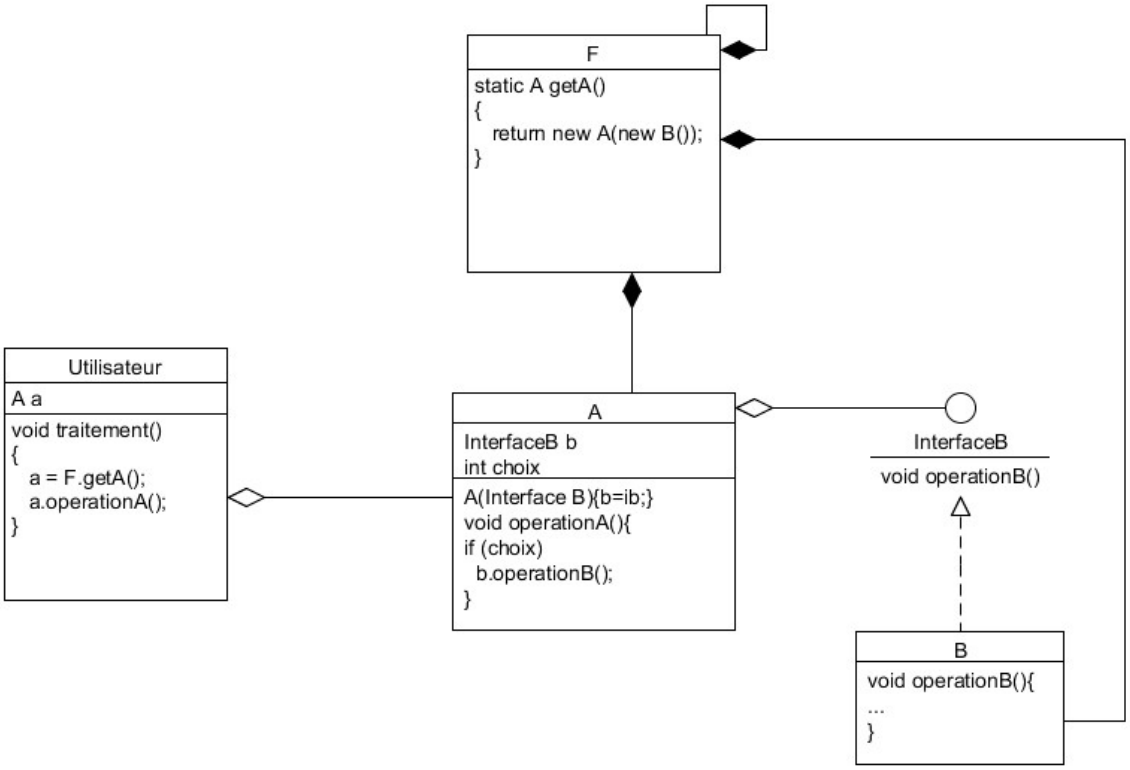
L'injection de dépendance utilise le principe de l'inversion de contrôle (IoC) appliqué au contrôle de la dépendance entre deux classes.

Le rôle de "IoC de dépendance" (= "Injection de dépendance") est de déléguer à une autre classe de réaliser l'inversion de controleur.
Cette classe utilise souvent un fichier de configuration pour déterminer quelle injection doit être réalisée.

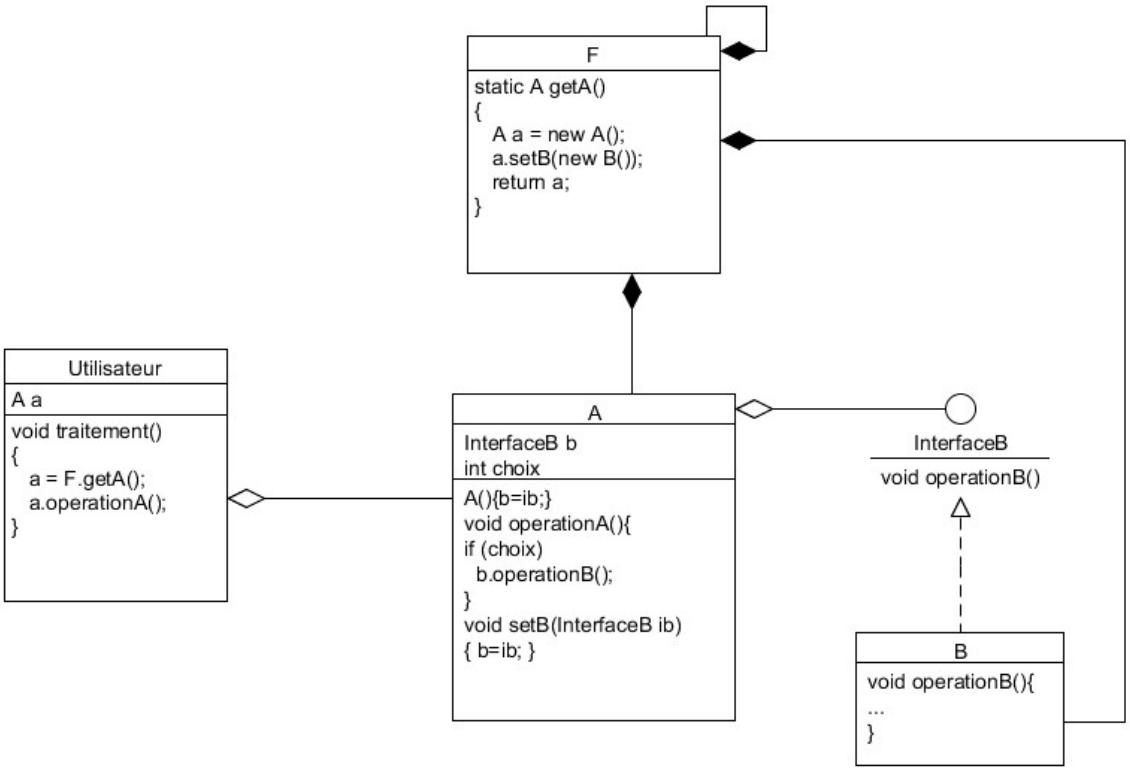
Si en plus cette classe permet de réaliser cette injection sans recompilation, on est dans le cas des framework, comme JEE ou Spring,

5.4.2. Description de la solution

L'injection de dépendance **par le constructeur** :



L'injection de dépendance **par setteur** :



La méthode d'injection est générique pour tout composant B qui implémente l'interface InterfaceB.

5.4.3. Exemple de code

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **ExempleCh04_04_InjectionDependance**

Voir dans cet exemple le fichier pdf joint.

5.5. L'adaptateur

5.5.1. Nom et rôle

Adaptateur ou Adapter.

Le rôle est d'adapter (transformer) un objet afin qu'il soit utilisé dans une classe alors qu'il ne peut pas l'être directement. ("rendre compatible un objet).

Le DP Adaptateur utilise le DP Délégation pour fournir son service

Il existe 2 sortes d'adaptateur dont le rôle est :

- rendre compatible un objet dans un contexte donné
- convertir un objet d'un type en un autre type

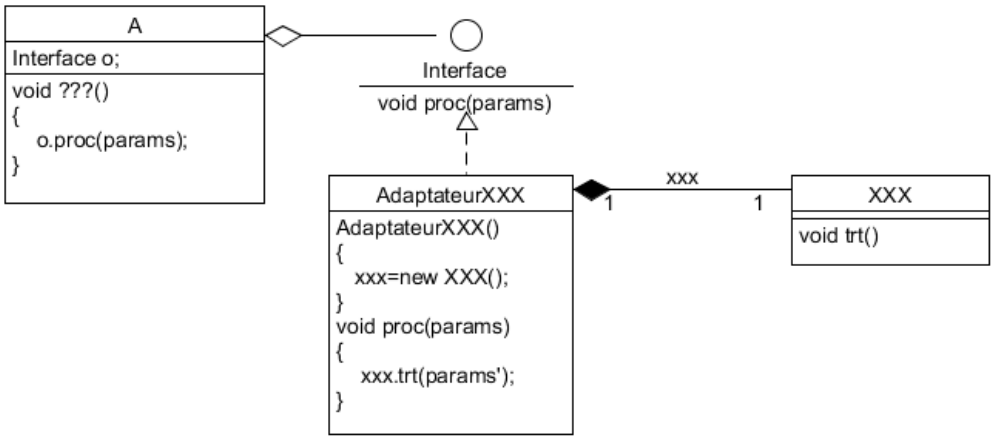
5.5.2. Compatible

a. Description du problème à résoudre

On a des classes qui ne peuvent pas implémenter une interface utilisée dans un traitement générique alors que l'on veut que cette classe soit utilisée dans le traitement générique.

b. Description de la solution

On crée une classe intermédiaire, l'adaptateur, qui agrège l'objet incompatible et qui implémente l'interface dont le code exploite les méthodes de la classe incompatible.



c. Exemple de Adapter

```
public class Collaboration
{
    public static void main(String[] args)
    {
        //on crée le chargeur
        Chargeur chargeur = new Chargeur();

        /******************* Portable Compatible ******************/
        PortableCompatible portableCompatible = new PortableCompatible();
        chargeur.brancherPortable(portableCompatible);
```

```
          System.out.println ("");

          /******************** Portable SonneEricSonne ***************/

          //on crée le portable et son adaptateur
          PortableSonneEricSonne portableSonne = new
PortableSonneEricSonne(20);
          AdaptateurSonneEricSonne adaptateurSonne = new
              AdaptateurSonneEricSonne(portableSonne);

          //on donne le portable à charger mais en utilisant son adaptateur
          chargeur.brancherPortable(adaptateurSonne);

          System.out.println ("");

          /******************** Portable SamSaoule ******************/

          //on crée le portable et son adaptateur
          PortableSamSaoule portableSam = new PortableSamSaoule();
          AdaptateurSamSaoule adaptateurSam = new
              AdaptateurSamSaoule(portableSam);

          //on donne le portable à charger mais en utilisant son adaptateur
          chargeur.brancherPortable(adaptateurSam);
      }
  }

  interface IChargeable
  {
      public void recharger(int voltage);
  }

  class Chargeur
  {
      // le voltage en sortie du chargeur
      private int voltage = 10;

      // branchement d'un portable pour le charger
      public void brancherPortable(IChargeable portable)
      {
          System.out.println("branchement d'un portable");
          portable.recharger(voltage);
      }
  }

  //================================================================

  // Un portable compatible avec l'interface IChargeable
  //
  class PortableCompatible implements IChargeable
  {
      public void recharger(int volts)
      {
          System.out.println ("Portable Compatible en charge");
          System.out.println ("voltage: " + volts);
      }
  }

  // Un portable non compatible avec l'interface et
```

```
//   dont la charge n'est pas la meme pour chaque instance
//
class PortableSonneEricSonne
{
    // ne se recharge
    int volts;

    public PortableSonneEricSonne(int volts)
    {
        this.volts=volts;
    }

    public void chargerBatteries()
    {
        System.out.println ("Portable SonneEricSonne en charge");
        System.out.println ("voltage: " + volts);
    }
}

// Portable non compatible avec l'interface et
//   dont la charge est 5 volts
//
class PortableSamSaoule
{
    // ne se recharge qu'avec du 5 volts
    public final int volts = 5;

    public void chargerPortable()
    {
        System.out.println ("Portable SamSaoule en charge");
        System.out.println ("voltage: " + volts);
    }
}

// ============================================================
//           Les adaptateurs

class AdaptateurSonneEricSonne implements IChargeable
{
    // référence sur le portable adapté
    private PortableSonneEricSonne telephone;

    public AdaptateurSonneEricSonne(PortableSonneEricSonne portable)
    {
        this.telephone = portable;
    }

    public void recharger(int volts)
    {
        this.telephone.chargerBatteries();
    }
}

class AdaptateurSamSaoule implements IChargeable
{
    // référence sur le portable adapté
    private PortableSamSaoule telephone;

    public AdaptateurSamSaoule(PortableSamSaoule portable)
    {
```

```
        this.telephone = portable;
    }

    //le portable SamSaoule n'a besoin que de 5 volts
    public void recharger(int volts)
    {
        this.telephone.ChargerPortable();
    }
}

Exécution :
branchement d'un portable
Portable Compatible en charge
voltage: 10

branchement d'un portable
Portable SonneEricSonne en charge
voltage: 20

branchement d'un portable
Portable SamSaoule en charge
voltage: 5
```

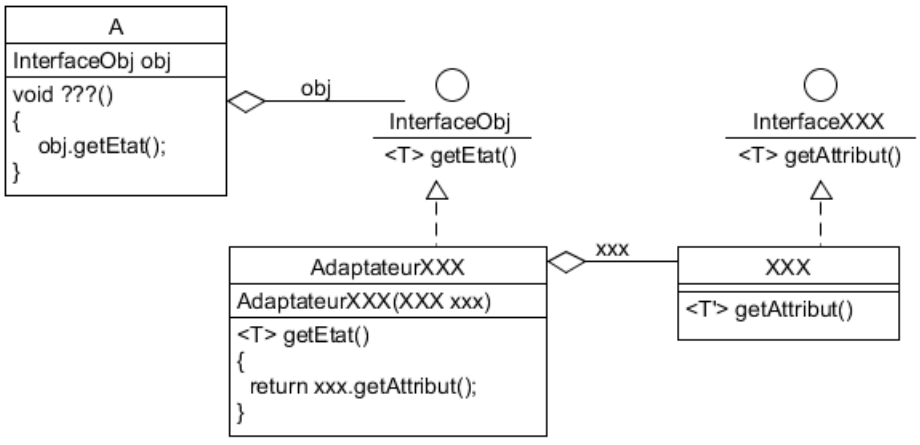
5.5.3. Conversion

a. Description du problème à résoudre

On a besoin de manipuler des objets décrit par une interface sous la forme d'une autre interface.

b. Description de la solution

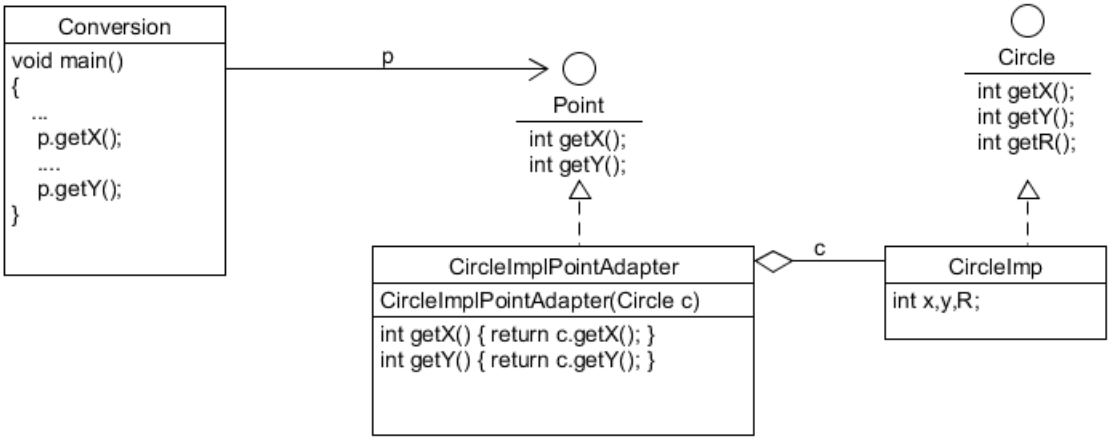
On crée un objet intermédiaire, appelé un adaptateur, qui va servir de "pont" entre la nouvelle interface et l'ancienne. Le pont implémente les méthodes de la nouvelle interface dont le code appelle les méthodes de la classe qui implémente l'ancienne interface.



c. Exemple de Conversion

```
public class Conversion
{
    public static void main(String[] args)
```

```
    {
        Circle c = new CircleImpl(10,20,100);

       System.out.println(c.getX()+" "+c.getY());

        Point p = new CircleImplPointAdapter(c);

        System.out.println(p.getX()+" "+p.getY());
    }
}

/** Interface de représentation d'un cercle */
interface Circle
{
    /** Retourne l'abscisse du centre du cercle */
    public int getX();
    /** Retourne l'ordonnée du centre du cercle */
    public int getY();
    /** Retourne le rayon du cercle */
    public int getR();
}

/** Classe implémentant l'interface Circle */
class CircleImpl implements Circle
{
    int x,y;
    int r;

    public CircleImpl(int x,int y,int r)
    {
        this.x=x;
        this.y=y;
        this.r=r;
    }
    public int getX(){return x;}
    public int getY(){return y;}
    public int getR(){return r;}
}

/** Interface de représentation d'un point */
interface Point
{
    /** Retourne l'abscisse du point */
    public int getX();
    /** Retourne l'ordonnée du point */
    public int getY();
}

/** Adapteur pour transformer le circle en un point */
class CircleImplPointAdapter implements Point
{
    private Circle c;
    public CircleImplPointAdapter( Circle c )
    {
        this.c = c;
    }
                public int getX() { return c.getX(); }
                public int getY() { return c.getY(); }
}
```


Conversion
void main()
{
...
p.getX();
....
p.getY();
}
p
Point
int getX();
int getY();
Circle
int getX();
int getY();
int getR();
CircleImplPointAdapter
CircleImplPointAdapter(Circle c)
int getX() { return c.getX(); }
int getY() { return c.getY(); }
c
CircleImp
int x,y,R;

5.6. Le Proxy

5.6.1. Nom et rôle

Proxy

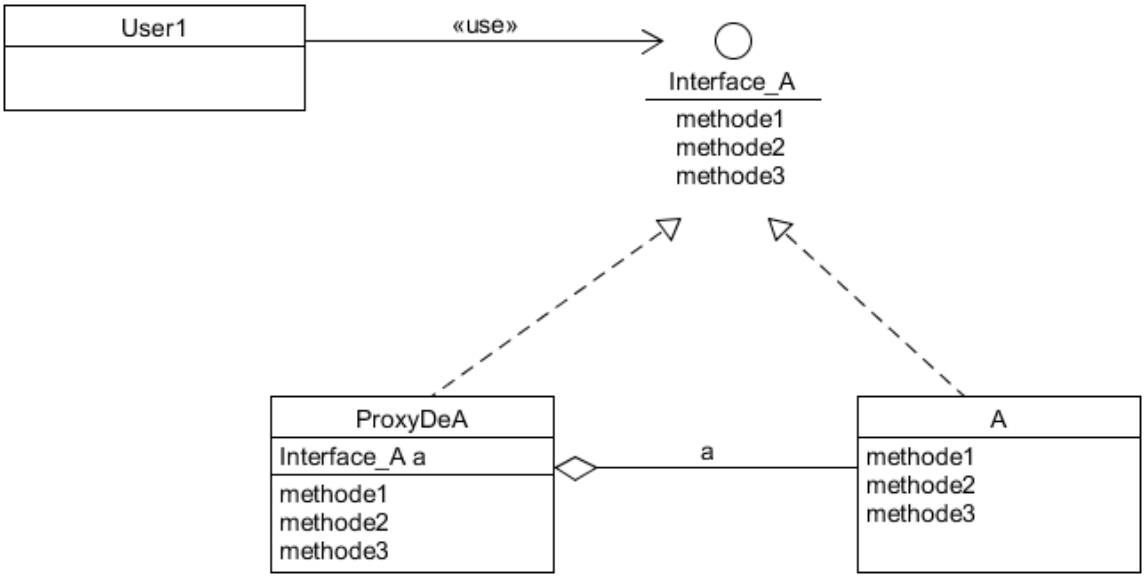
Soit un utilisateur d'un objet cible donné, le rôle d'un Proxy est de servir d'intermédiaire entre l'objet cible et son utilisateur tout en imitant parfaitement tout le comportement de l'objet cible.
L'utilisateur ne voit pas de différence entre l'utilisation directement de l'objet cible ou du Proxy.

De plus, il est ainsi possible de changer l'objet cible sans impact sur l'objet manipulé par l'utilisateur.

5.6.2. Description de la solution

A cet effet, on construit une nouvelle classe implémentant l'interface de l'objet à manipuler et déportant toutes les actions sur un autre objet implémentant la même interface.

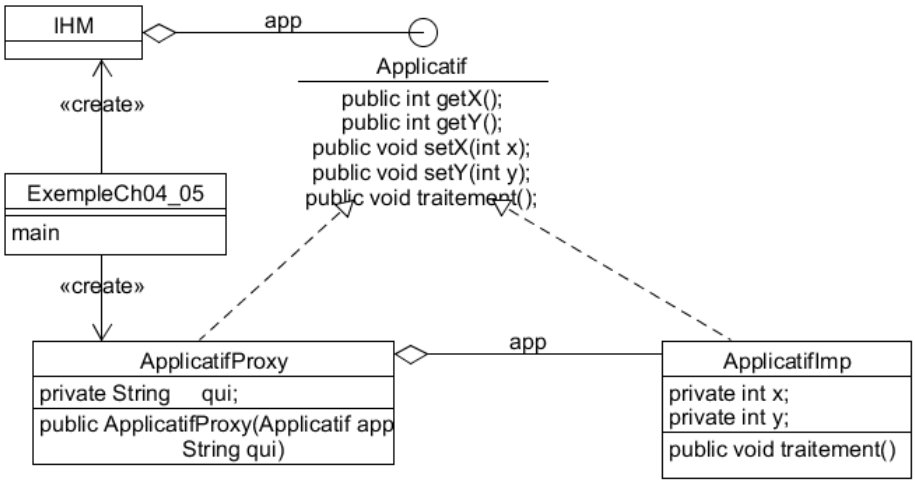
Nous verrons plus loin que les Proxy sont très utilisés pour la gestion d'objets distribués (protocole RMI en Java par exemple). L'idée étant de construire des Proxy capable de communiquer avec des objets distants (usage de la sérialisation en Java) sans que l'exploitant fasse de différences entre un accès local ou un accès distant.



5.6.3. Exemple de Proxy

Séparation de l'applicatif et de l'IHM

Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_05_DPProxy



Pour mieux illustrer, l'usage de ce design pattern, on peut aller voir l'exemple ExempleCh05_03 qui est vu en détail dans le chapitre NSY102-Chapitre-05_ServiceEtInterface.

Cet exemple montre la séparation de l'applicatif et de son Ihm par une interface qui permet de réaliser une architecture client/serveur basé sur le principe du Proxy avec deux solutions d'implémentation du proxy : une en RMI et une autre en http.

5.7. Le Décorateur

5.7.1. Nom et rôle

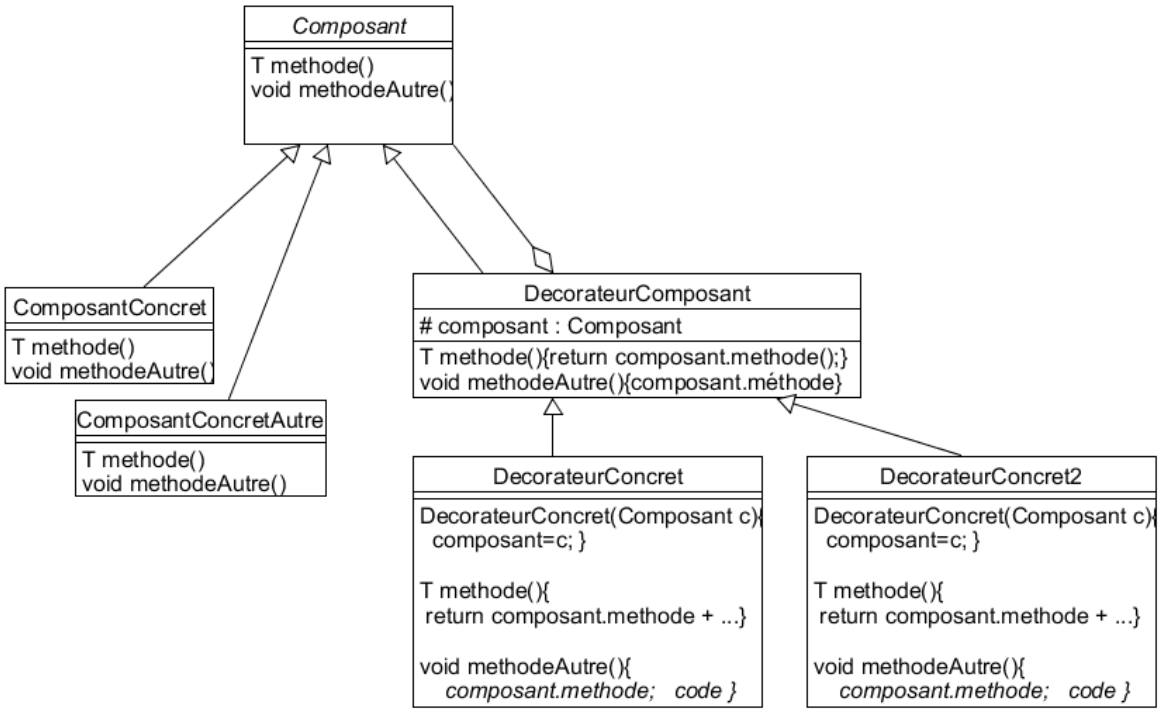
Decorator (Décorateur)

Le rôle de ce DP est de surcharger (étendre ou remplacer) les méthodes d'une classe sans utiliser le mécanisme de surcharge (ou redéfinition) de l'héritage qui a ses limites.

D'une manière générale on constate que l'ajout de fonctionnalités (à travers des méthodes connues) dans un programme s'avère parfois délicat et complexe.
Ce problème peut être résolu si le développeur a identifié, dès la conception, qu'une partie de l'application serait sujette à de fortes évolutions. Il peut alors faciliter ces modifications en utilisant le pattern Décorateur.

5.7.2. Description de la solution

La solution est de créer une sorte de « proxy » de la classe que l'on veut « décorer », qui va permettre de surcharger les méthodes que l'on veut étendre.




```
  Composant   c   =   new   DecorateurConcret2   (new   DecorateurConcret(new
ComposantConcret(…), …),…) ;
```

Le constructeur du décorateur concret prend en paramètre le composant concret, et surcharge les méthodes du composant concret.

Étant donné que le décorateur concret hérite de la classe abstraite Composant, il est vu comme un composant (principe du proxy).

Si le décorateur concret ne surcharge pas toutes les méthodes, la classe DecorateurComposant (DECORATEUR NEUTRE) définit par défaut l'appel aux méthodes du composant concret (pas de changement).

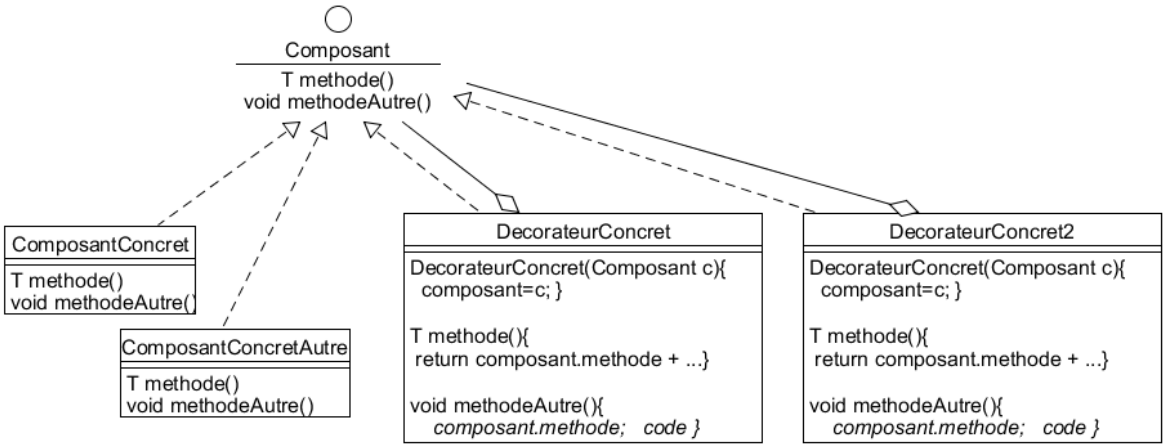
De plus, cette classe peut contenir des méthodes utilisées dans tous les décorateurs qui en héritent.

On peut donc obtenir une hiérarchie complexe des décorateurs.

La classe abstraite Composant peut contenir des attributs utilisés pour stocker une information qui est mise à jour sur les appels successifs des décorateurs.

A l'image du Builder qui réalise la création d'objets par assemblage, ce DP est utilisé dans les Factory pour faire la création d'objets par extension.

En utilisant, l'interface, la forme la plus simple d'un Décorateur est :



5.7.3. Exemple

Description du problème

Afin de mettre en pratique le pattern Décorateur, nous allons concevoir une application qui permet de gérer la vente de desserts. Celle-ci doit permettre d'afficher dans la console le nom complet du dessert choisi et son prix. Les clients ont le choix entre deux desserts : crêpe ou gaufre. Sur chaque dessert ils peuvent ajouter un nombre quelconque d'ingrédients afin de faire leurs propres assortiments. Pour simplifier notre exemple, nous choisirons uniquement deux ingrédients (le chocolat et la chantilly) mais il faut garder à l'esprit que l'ajout de nouveaux ingrédients doit être simplifié. Le système de tarification est simple. Une crêpe (nature) coûte 1.50€ et une gaufre 1.80€. L'ajout de chocolat est facturé 0.20€ et de chantilly 0.50€.

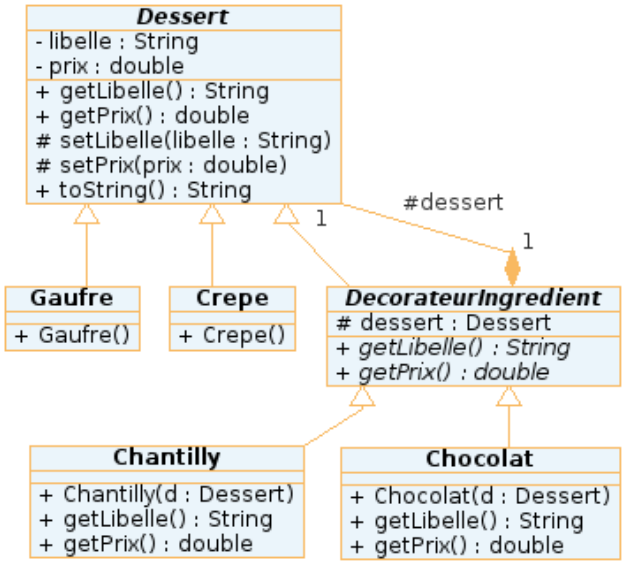
Voyons comment concevoir cette application.

Une première solution (pas excellente) est de mettre en place une classe abstraite Dessert ayant deux attributs (libelle et prix) et les accesseurs en lecture/écriture correspondants. Puis, créer pour chaque combinaison de desserts et d'ingrédients une classe (CrepeChocolat, CrepeChantilly, GaufreChocolat, GaufreChantilly). Bien sûr cette solution n'est pas évolutive. Si l'on souhaite modifier le prix de l'ingrédient Chocolat on doit le modifier dans deux classes. De plus, si on ajoute une dizaine d'ingrédients nous allons obtenir une centaine de classes.

Une deuxième solution (pas excellente non plus) consiste à garder la classe Dessert en la modifiant légèrement. On peut rajouter un booléen pour savoir si l'ingrédient chocolat est ajouté à ce dessert de même pour Chantilly. Puis, on crée des classes Crepe et Gaufre qui héritent de Dessert. Le calcul du prix des ingrédients est effectué dans la classe Dessert auquel on rajoute le prix spécifique du dessert suivant le type de l'objet (Gaufre ou Crepe). Cette solution semble plus satisfaisante mais pose toujours certains problèmes. Si l'on souhaite rajouter un ingrédient, il faut ajouter un attribut dans la classe Dessert et modifier la méthode qui calcule le prix afin de le prendre en considération. De plus, toutes les classes héritant de Dessert posséderont ces attributs qui n'auront pas toujours de sens. Si on crée une classe SaladeDeFruit héritant de Dessert on aura un attribut (hérité de la classe mère) nommé chocolat (bizarre pour une salade de fruit).

Une troisième solution : celle du décorateur.

Voyons ce qu'apporte le pattern décorateur à notre problématique :



La solution consiste à utiliser à la fois l'héritage et la composition. Une classe abstraite Dessert regroupe les attributs et les méthodes communes. Puis, des desserts concrets tel que Gaufre et Crepe héritent de cette classe. Dans le constructeur de ces classes on met à jour les attributs définis dans Dessert à l'aide des accesseurs. Afin de gérer les ingrédients, il faut une classe abstraite nommée DecorateurIngredient. Celle-ci possède un Dessert en attribut et oblige la redéfinition de deux méthodes getLibelle() et getPrix(). Chaque ingrédient (Chantilly, Chocolat...) doit hériter de la classe DecorateurIngredient. Le constructeur de ces classes permet d'initialiser l'attribut dessert présent dans la classe mère. De plus, la redéfinition des méthodes getLibelle() et getPrix() va permettre d'ajouter des fonctionnalités. Pour comprendre comment cela fonctionne voyons le code Java correspondant au diagramme UML.

```
// Creation et affichage d'une gaufre au chocolat.
    //
    System.out.println(new Chocolat(new Gaufre()));

    // Creation et affichage d'une crepe au chocolat et chantilly.
    //
    System.out.println(new Chantilly(new Chocolat(new Crepe())));
  }
```

Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_06_DPDecorateur

L'inconvénient de ce pattern est la multiplicité des objets qu'il faut créer pour l'utiliser. On peut le remarquer avec le nombre de « new » présent dans la classe Main. Pour résoudre ce problème on pourra combiner Décorateur avec un autre pattern tel que Fabrique ou Monteur.

5.8. Le Composite

5.8.1. Nom et rôle

Composite

Le rôle de ce DP est de décrire la conception d'objet qui sont composés d'objets similaires.

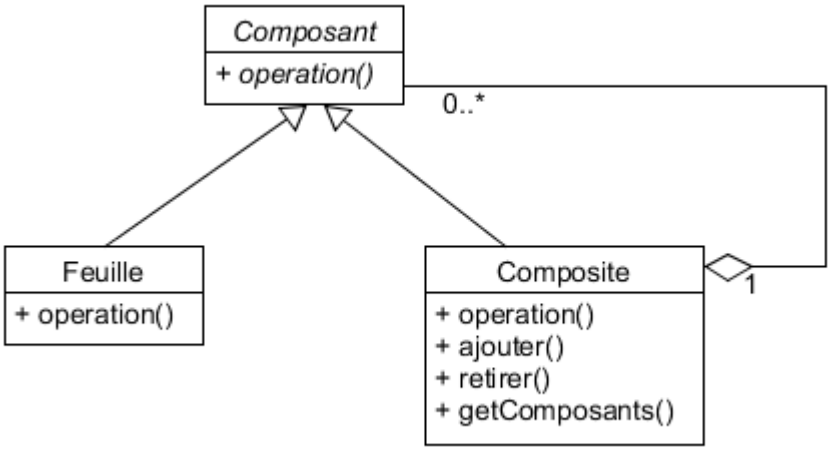
L'objectif est de manipuler tout un groupe d'objet comme s'il s'agissait d'un seul objet.

Ainsi l'appel d'un traitement à un objet peut se propager à tous les objets du groupe.

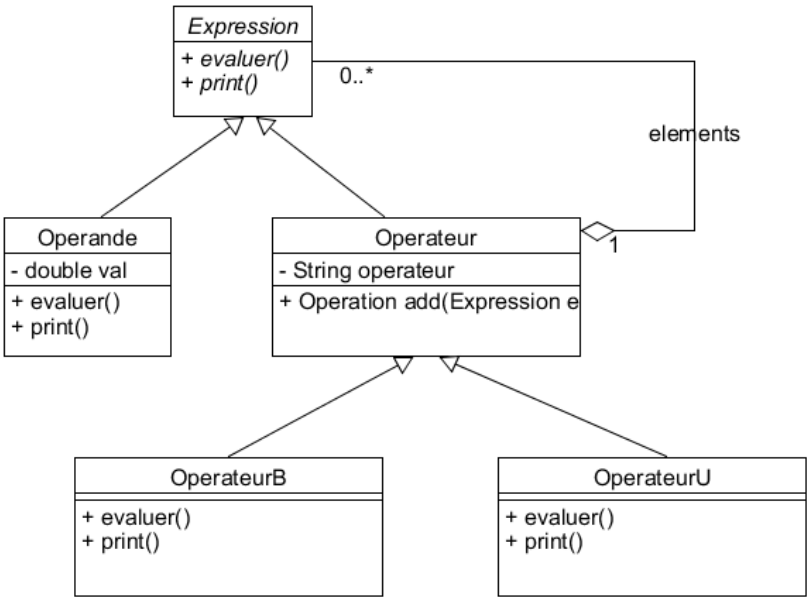
Il est à noter que ce type de classe est à rapprocher des classes récursives. Une classe récursive est une classe qui a un lien de dépendance avec la même classe. (Exemple ArrayList, Arbre, …).

5.8.2. Description de la solution

La solution est de rendre concret la différence entre l'objet composite et l'objet non composite (ou "feuille") par la conception de de deux classes concrètes, et de rendre abstrait l'appartenance de tous les objets (composites ou non) par une classe abstraite ou une interface.



Exemple : Une expression arithmétique.



Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_07_DPComposite

Dans cet exemple, on peut voir que, à la différence d'une classe récursive, le traitement récursif de l'évaluation d'une expression arithmétique n'est pas explicitement décrite dans un algorithme mais dilué dans les différentes classes.

6. Les modèles de comportement

Les modèles de comportement sont :

- des modèles d'interfaces dont les méthodes virtuelles sous-entendent un comportement global attendu par l'utilisateur et celui qui doit implémenter les méthodes. Ceci permet de mettre en commun des descriptions de comportements implémentés différemment dans différentes classes mais vu pour l'utilisateur de la même façon (exemple les interfaces des collections List, Iterator, …

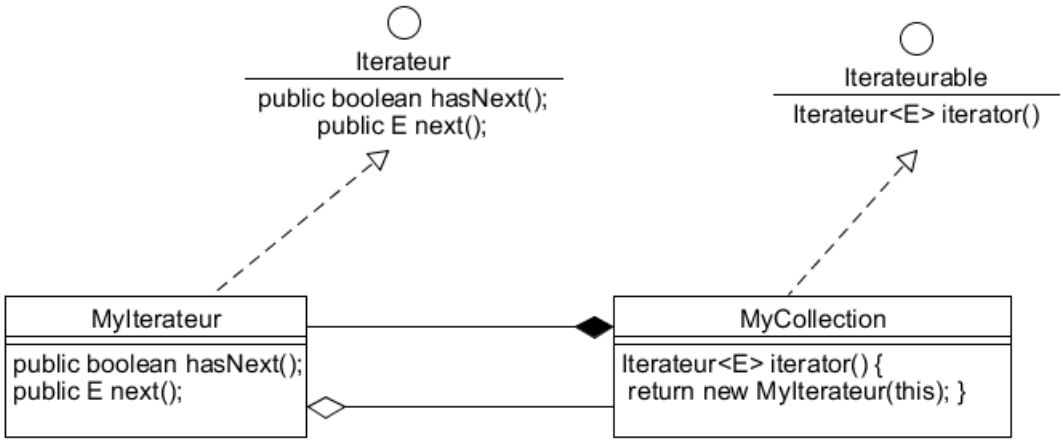
- des architectures de classes abstraites dont l'architecture peut être modélisée et donc réutilisable dans des besoins et conceptions similaire (très proche de Best Pratices). Exemple : gestion d'un moteur de production et de consommation d'évènement en "Push" et "Pull"

Autre exemple : le modèle MVC

6.1. Iterator

Le DP itérateur est utilisé sur une classe quand celle-ci contient des éléments pour lesquels il est intéressant de les parcourir ou quand elle peut être vue sous une forme récurrente.

C'est le cas de :
- toutes les collections d'éléments
- des fichiers séquentiels
- des feuilles d'une arborescence
- des données récurrentes comme des compteurs sophistiqués (nombre aléatoire)
- ...



Commentaires :
- la collection MyCollection crée une instance de MyIterateur dont le rôle est de parcourir les éléments de la collection.
- l'interface Iterateurable décrit la méthode qui permet d'obtenir un itérateur
- l'interface Iterateur décrit les méthodes permettant de réaliser une itération.

Ce DP existe dans l'API JAVA (Iterator, Iterable) et est utilisé par les collections (Collections).

Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_08_DPIterateur

Une meilleure implémentation permet de ne pas rendre public les méthodes de la collection qui permettent de réaliser l'itération (méthode getNb et getElement) en créant l'itérateur dans une inner-classe.

Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_09_DPIterateur2

6.2. Visitor

6.2.1. Nom et rôle

Visiteur.

Visiteur est un DP qui permet de séparer les algorithmes et les objets sur lesquels ils opèrent.
Visiteur est un DP qui permet d'ajouter de nouveaux comportements à une hiérarchie de classes sans modifier l'existant.

La problématique est la suivante.
Soit toute une famille de classe, appartenant à une arborescence complexe d'héritage. On veut ajouter de nouveaux traitements à ces classes qui correspondent à des algorithmes réalisés sur les instances de ces classes.

Pour chacun des algorithmes, la solution classique est d'implémenter ces traitements sous forme de méthodes dans chacune des classes. Et rendre ces méthodes génériques. Cela entraine un impact sur toutes les classes. Si ces traitements algorithmiques ont besoin de nouvelles données communes ou des sous-traitements communs, il faut mettre ces données et ces sous-traitements dans la classe abstraite dont hérite toutes les classes. Cela est peut devenir un impact fort.

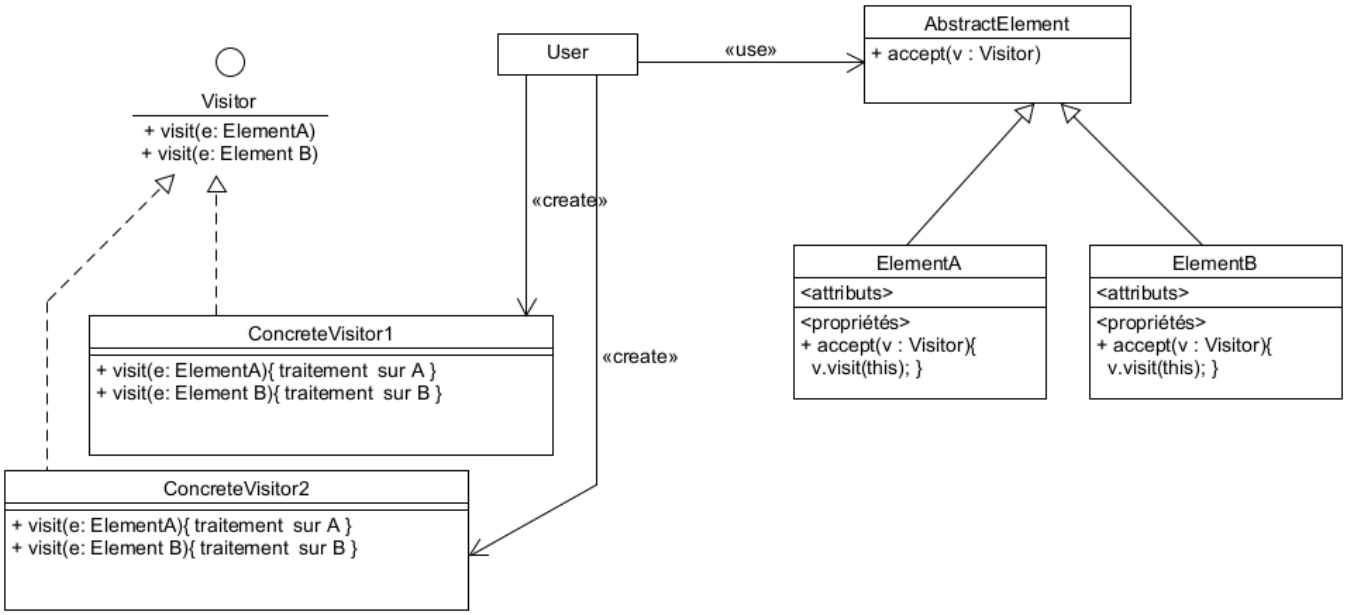
Et si en plus ces nouveaux traitements que l'on veut réaliser sont des traitements temporaires (besoin de test, contrainte d'intégration continu, phase de mise au point, …).

Alors le DP Visiteur est une solution.

Le désavantage est que l'on perd l'aspect redéfinition de ces traitements dans le graphe d'héritage.

Ces nouveaux traitements ne sont pas des traitements qui s'exécutent à travers l'objet (propriété de l'objet) mais s'exécutent sur l'objet.

6.2.2. Description de la solution



Les classes concrète des éléments implémente la méthode **accept** qui accepte en paramètre une instance d'un visiteur.

L'interface **Visitor** liste toutes les classes qui acceptent un visiteur.

Une classe **ConcreteVisitor** définit tous les traitements qui doivent être réalisés pour chacune des classes des éléments.

L'utilisateur crée le visiteur et appel les traitements pour chacune des classes en utilisant la méthode **accept**.

User → accept →visit →traitement sur A

L'objectif est de retrouver une forme simple d'appel générique sur tous les objets appartenant à la même famille de classe :

for( ) o.accept(visteur) ;

Si le visiteur ne s'applique pas sur toutes les classes des éléments concrets, il faut que la méthode **accept** de **AbstratcElement** ne soit pas abstraite, et peut contenir du code vide.

Dans ce cas, si on veut créer des groupes de classe d'élément différents il faut définir des interfaces de visiteur différents. Ce qui entraine de définir autant de méthode **accept** que de types de visiteur différents.

La signature des méthodes de visite est libre. On donne souvent des noms de méthodes différentes (visitElementA, visitElementB, …)

Cela est obligatoire si les paramètres de ces méthodes pourraient être identiques.

Les méthodes de visite peuvent accepter des paramètres variés dont notamment passés en paramètre des valeurs d'attribut privée de l'élément, ou une valeur de l'exécution d'une méthode privée de l'élément. Cela n'est pas « protocolaire » mais le besoin peut exister.

6.2.3. Exemple de visiteur

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **ExempleCh04_12_Visitor**

6.3. Strategy

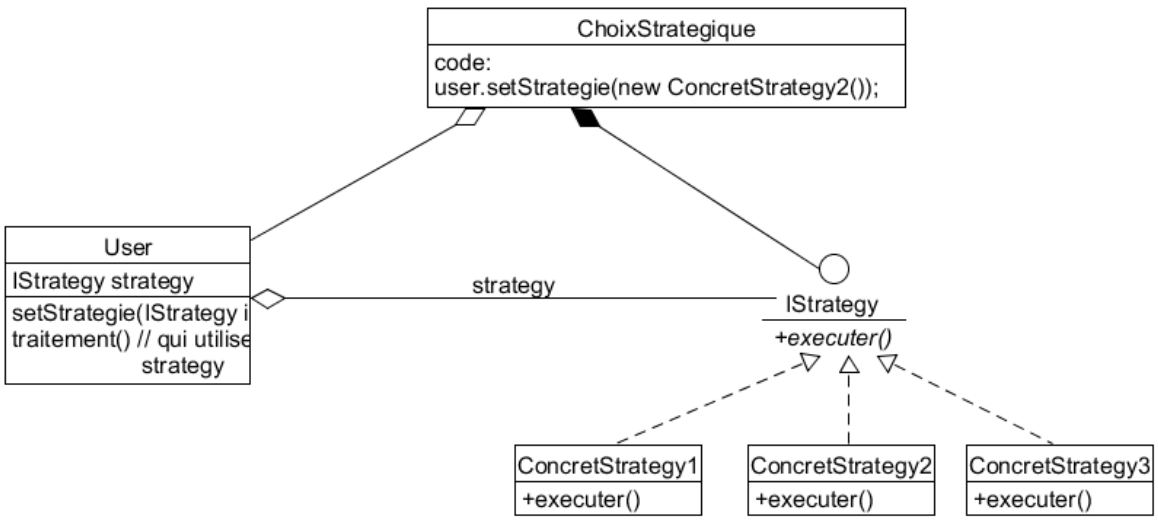
Le patron de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application.

Le patron stratégie est prévu pour fournir le moyen de définir une famille d'algorithmes, **encapsuler chacun d'eux en tant qu'objet**, et les rendre interchangeables.

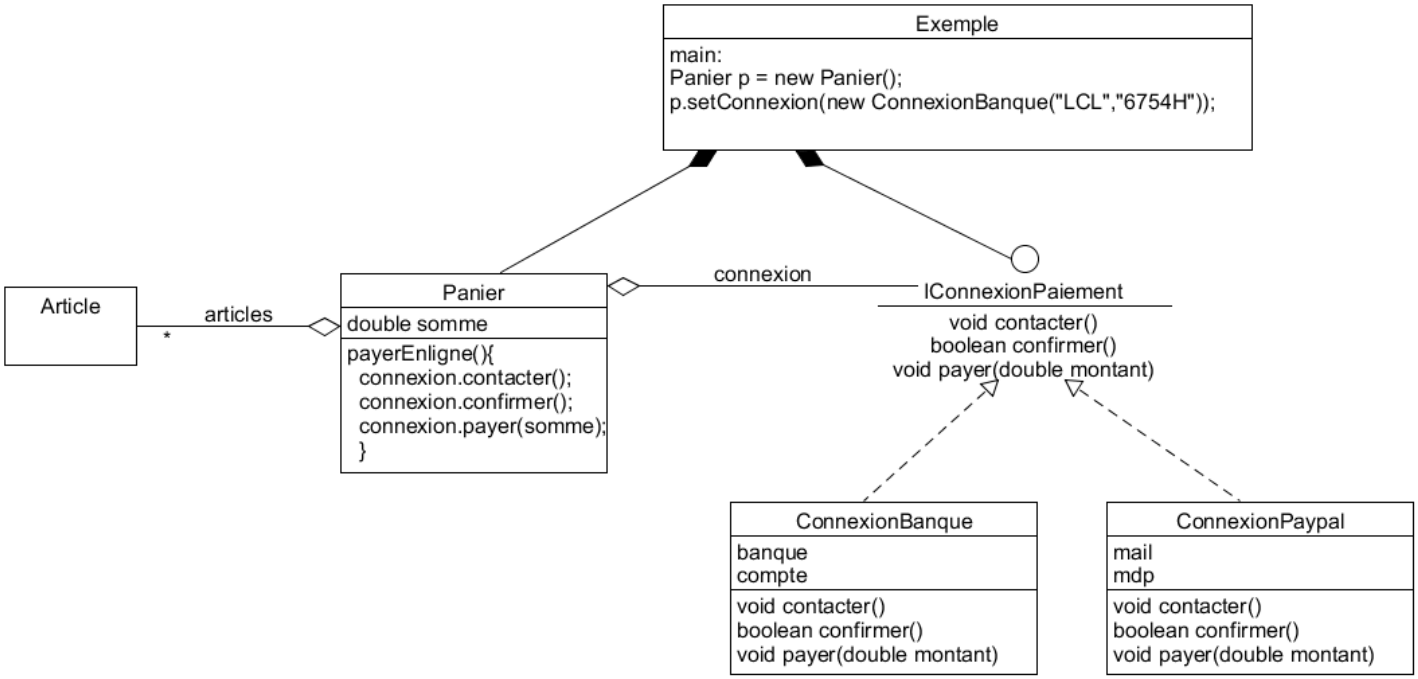
Ce patron laisse les algorithmes changer **indépendamment des clients** qui les emploient.

~~Ce DP est voisin de l'injection de dépendance et utilise le DP **Interface** dans le cas d'un traitement générique.~~

On "injecte" le traitement concret dans l'utilisateur soit par le constructeur, soit par un setteur.



Exemple :



Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh4_09_DPStrategie

6.4. L'observateur

6.4.1. Nom et rôle

Observer ou observateur = listener

Le rôle de ce DP est de prévenir des utilisateurs du « changement d'état » d'un objet cible, sans que les utilisateurs soient directement liés à l'objet cible, et sans que le nombre des utilisateurs soit figé dans le temps.
On parle de « couplage faible ou lâche ».

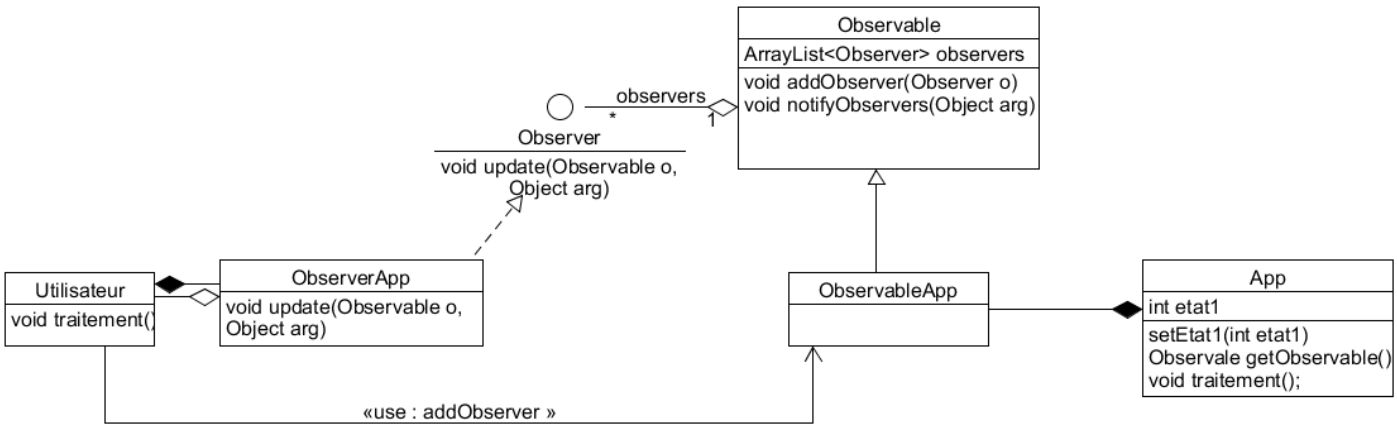
Il existe différentes formes d'observateur en fonction de l'impact sur l'objet cible et/ou du mode synchrone ou asynchrone de l'observation :

- synchrone minimal (classique)
- synchrone sur changement d'état
- synchrone proxy
- asynchrone push
- asynchrone ~~pull~~ push-and-push

6.4.2. Synchrone minimal

Ce DP sert de base à tous les DP Observateur.
Les principes sont les suivants :

- Chaque utilisateur s'abonne à l'objet cible (à travers un "Observable")
- Chaque utilisateur crée un "Observer" qui est vu par l'Observable sous la forme d'une Interface
- L'Observable notifie chaque Observer du changement de son état
- Il y a au moins autant d'Observer qu'il existe d'utilisateur
- Il y a au moins autant d'Observable qu'il existe d'objet cible



Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_10a_DPObserverSimple

La notification aux observateurs est synchrone.
Cela implique que, tant que les observateurs n'ont pas finis leurs méthodes de mise à jour (méthode update) alors l'objet cible est en attente et les autres observers aussi.

6.4.3. Synchrone sur changement d'état

Pour économiser les notifications, on sépare les notions de changement d'état et de notification :

- chaque changement d'état mais à jour un booléen précisant qu'il y a eu au moins 1 changement d'état
- un changement d'état particulier (il peut ne pas être le seul) teste s'il y a eu au moins un changement d'état. Et si c'est le cas alors réalise la notification aux observateurs

Voir sur le site http://jacques.laforgue.free.fr l'exemple
Exemple Ch04_10b_DPObserverEtat

En JAVA, comme l'exemple le démontre, la classe Observable permet cette gestion de changement d'état :

- la méthode **setChanged()** permet de préciser qu'il y a eu au moins 1 changement
- la méthode **notifyObservers()** notifie les observateurs que s'il y a eu au moins un changement.

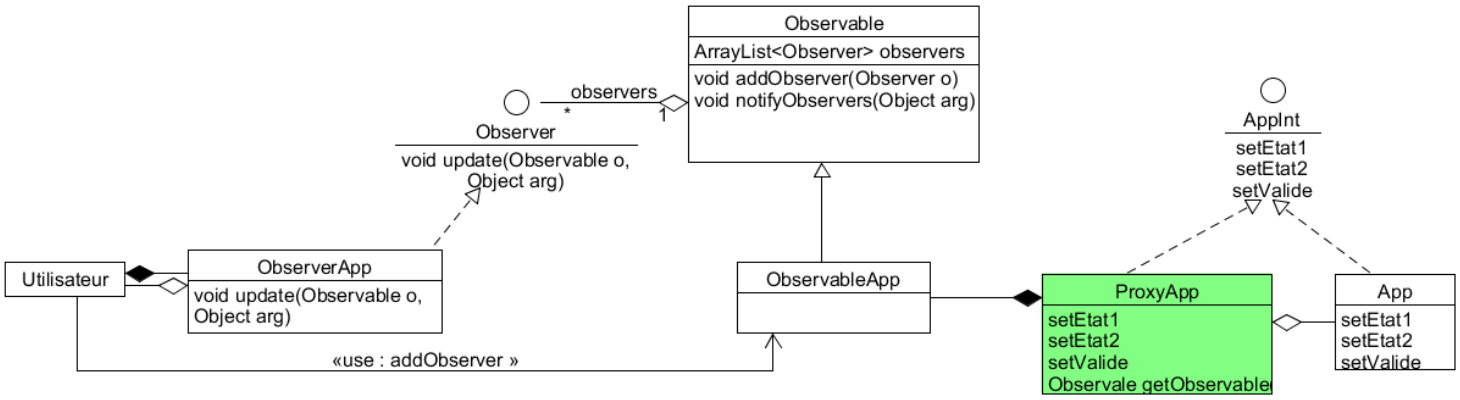
6.4.4. Synchrone proxy

Pour limiter l'impact sur l'objet cible et pour formaliser rigoureusement les changements d'états, on crée un Proxy qui sert d'intermédiaire avec l'objet cible.

Nous pouvons noter que :

- les états de l'objet cible sont tous vus comme des setteurs d'attribut
- tous les changements d'état nécessitant une notification sont formalisés dans une interface
- le proxy assure la mécanique Observer/Observable. Ainsi le codage de l'objet cible devient indépendant de cette mécanique. A condition que les setteurs utilisés dans les traitements soient ceux du Proxy

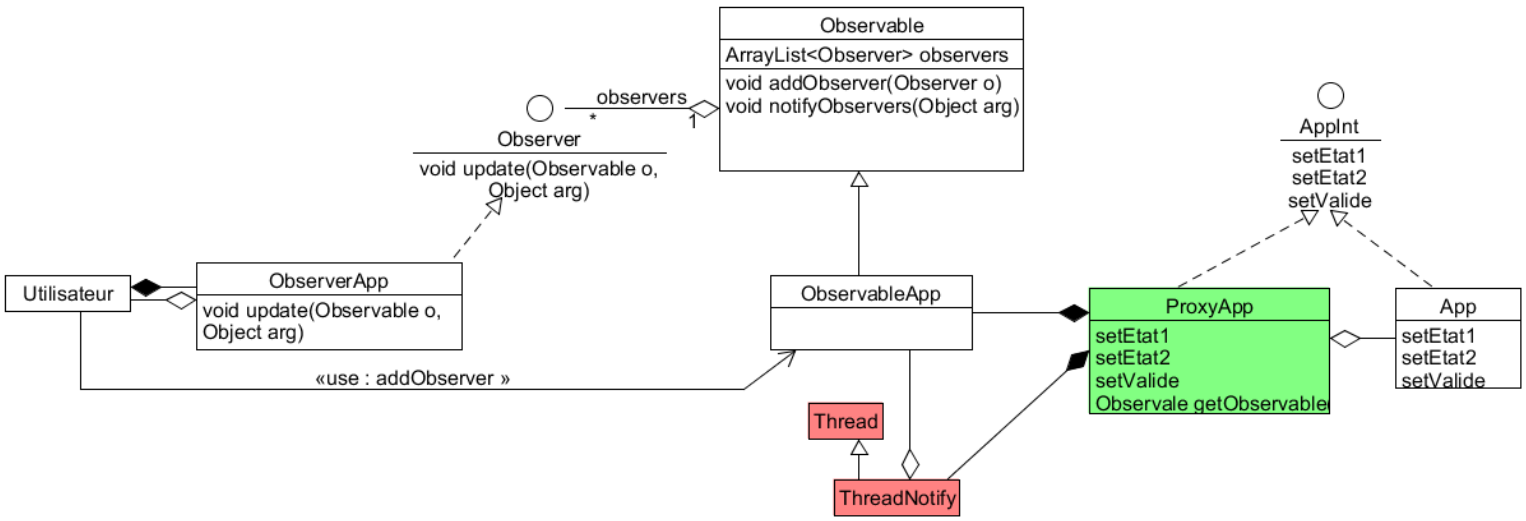
Voir sur le site http://jacques.laforgue.free.fr l'exemple
Exemple Ch04_10c_DPObserverProxy



6.4.5. Asynchrone push

Afin que l'applicatif ne soit pas en attente du traitement de notification par l'Observable, l'Obervable réalise l'ensemble des notifications à travers un Thread.

Voir sur le site http://jacques.laforgue.free.fr l'exemple
Exemple Ch04_10d_DPObserverAsynchronePush



6.4.6. Asynchrone push-and-push

Dans ce cas, on crée un "intermédiaire" entre l'observer et l'observable précédent ;
Cet intermédiaire est constitué de :
- un observer qui s'abonne à l'observable de l'applicatif
- un observable qui notifie à l'observer de l'utilisateur

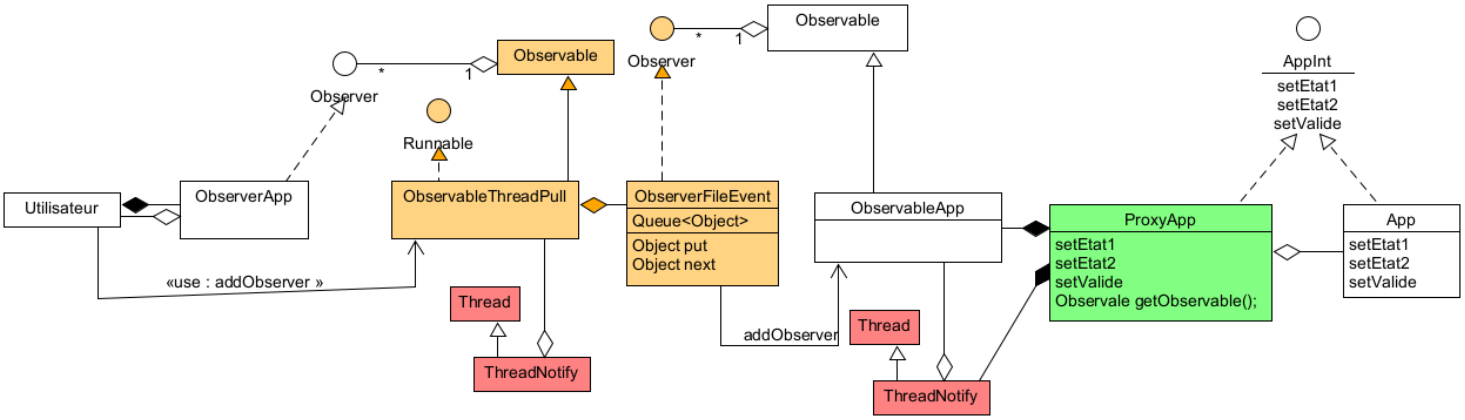
Il existe deux variantes (un paramètre de l'Observer) :

- les évènements ne se cumulent pas
- les évènements se cumulent

Pour ne pas impacter l'observer existant de l'utilisateur et l'observable de l'applicatif existant, on crée un intermédiaire constitué d'un Observer et Observable :

- ce nouvel Observer stocke dans une file les notifications de l'observable de l'applicatif
- ce nouveau Observable est un thread qui interroge cycliquement la file et notifie l'observer de l'utilisateur. ~~Ce thread peut être remplacé par une méthode qui tire l'évènement et appelé par l'utilisateur.~~

Voir sur le site http://jacques.laforgue.free.fr l'exemple
~~**Exemple Ch04_10e_DPObserverAsynchronePull**~~
Exemple Ch04_10e_DPObserverAsynchronePushAndPush



6.5. MVC

6.5.1. Nom et rôle

Modèle-Vue-Contrôleur

Le design pattern Modèle-Vue-Contrôleur (MVC) est un pattern architectural qui sépare les données (le modèle), l'utilisateur du modèle (la vue) et la logique de contrôle du modèle (le contrôleur).

6.5.2. Description du problème à résoudre

Le problème est d'imaginer une architecture logicielle dans la réalisation d'application client/serveur (ex: Application Internet) qui assure une bonne maintenabilité de ses composants.

Cela n'est pas nouveau, il faut séparer les systèmes d'information en au moins 2 couches:

- la partie applicative qui réalise les traitements "métier" et gère les données
- la partie IHM qui réalise l'interface avec les utilisateurs

Ce qui est nouveau est l'apparition d'un nouveau composant : le contrôleur.

Ce composant est issu de la réflexion de la conception des applications internet dont le besoin est de contrôler :

- la logique d'enchaînement des vues (pages)
- le contrôle d'accès aux services (traitements)
- les appels aux services applicatifs
- …

Ce modèle de conception impose donc une séparation en 3 couches :

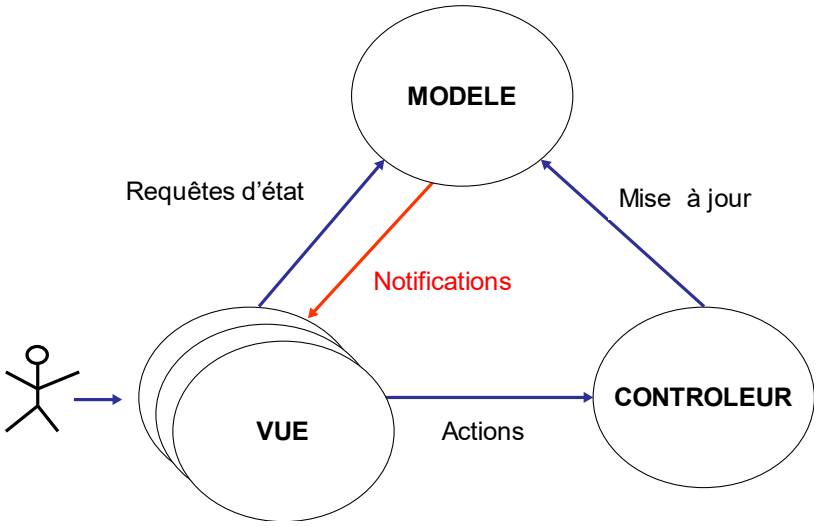
Le modèle : Il représente les données de l'application. Il définit aussi l'interaction avec la base de données et le traitement de ces données.

La vue : Elle représente l'interface utilisateur, ce avec quoi il interagit. Elle n'effectue aucun traitement (métier), elle se contente simplement d'afficher les données que lui fournit le modèle. Il peut tout à fait y avoir plusieurs vues qui présentent les données d'un même modèle.

Le contrôleur : Il gère l'interface entre le modèle et la vue. Il va interpréter la requête pour mettre à jour le modèle. Il effectue la synchronisation entre le modèle et les vues.

La synchronisation entre la vue et le modèle se fait avec le pattern *Observateur* (Voir l'exemple Ch04_11 plus bas) Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour.

Voici un schéma des interactions entre les différentes couches :



Les **Actions** sont contrôlées, filtrées, analysées par le CONTROLEUR. Les traitements du CONTROLEUR sont plus ou moins sophistiqués.

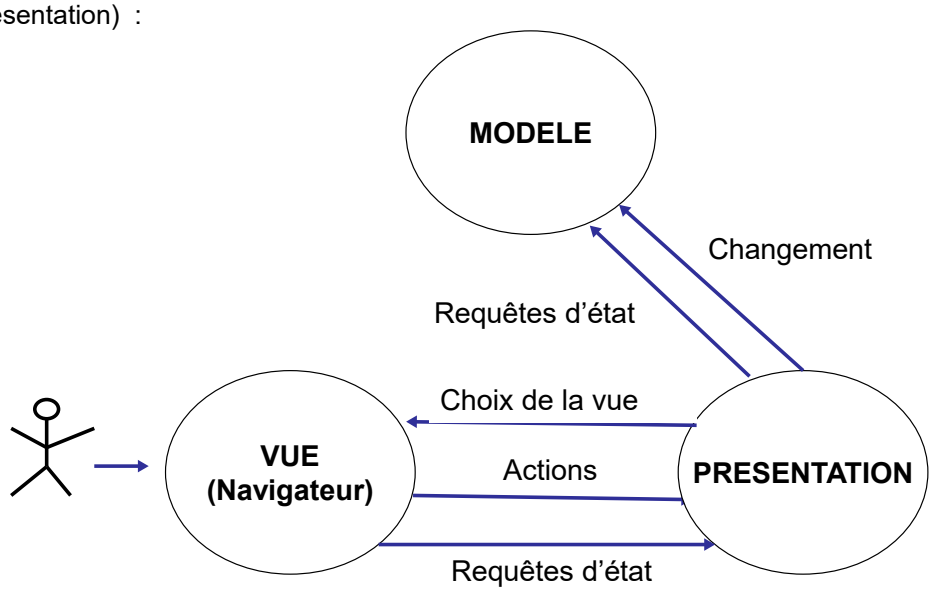
Ces traitements réalisent des **Mises à jour** du modèle. Ainsi, le MODELE réalise des traitements métiers consistant à mettre à jour l'état de ces données. Ces traitements métiers sont souvent très sophistiqués dont la mise à jour de la base de données.

Le MODELE prévient alors TOUTES les VUES en leurs faisant des **Notifications**. Ce qui permet à la VUE de se rafraichir.

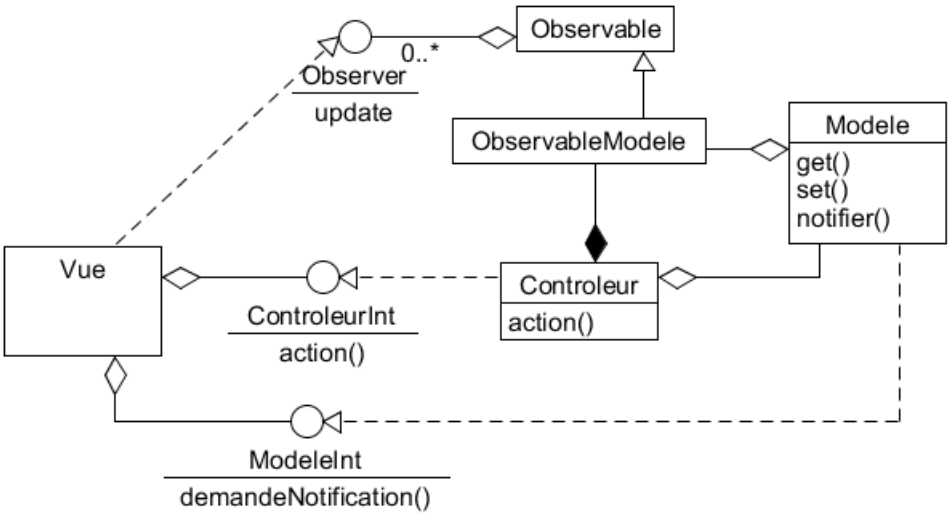
La VUE peut directement demander au MODELE les états de ses données (phase d'initialisation, requêtes cycliques de rafraichissement, …) à travers des **Requêtes d'état**.

L'utilisation de ces requêtes d'état rend la VUE dépendante du MODELE et peut nuire à la vision d'un couplage lâche entre la VUE et la couche métier de l'application.

Une variante du modèle MVC adapté au monde de l'internet le modèle MVP (P pour Présentation) :



6.5.3. Description de la solution (MVC)



6.5.4. Exemple de MVC : l'exemple Ch04_11 avec Observer

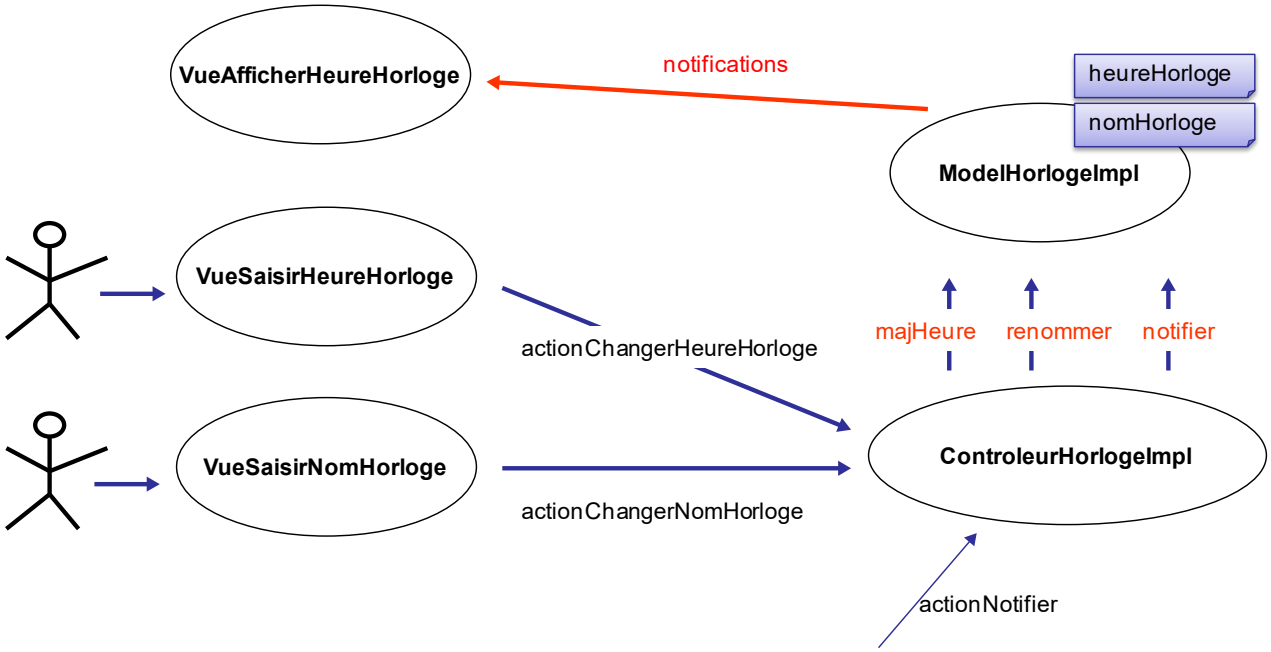
Dans cet exemple on se propose de créer un modèle qui gère un temps d'horloge basé sur l'horloge interne de l'ordinateur.

Les 3 vues ci-dessous sont créées. Ensuite un programme principal utilise ces vues pour afficher l'heure de l'horloge, mettre à jour l'heure de l'horloge et changer le nom de l'horloge.

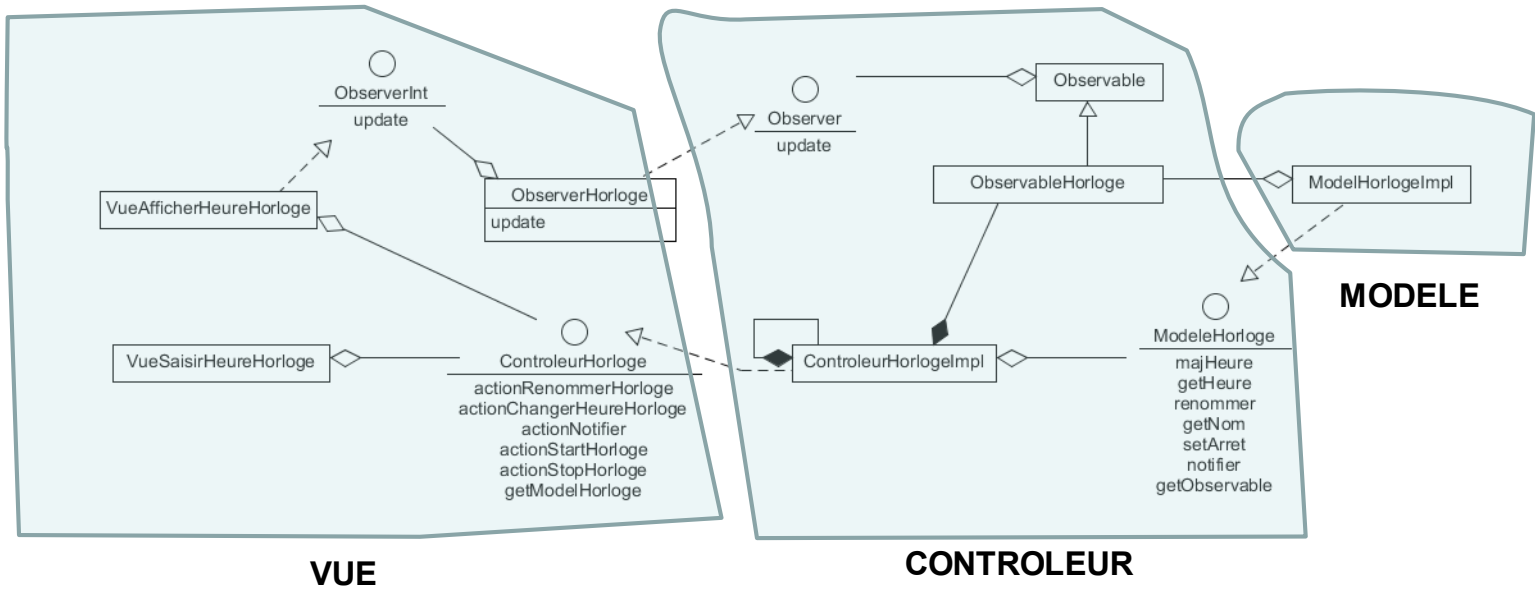
La vue d'affichage de l'heure de l'horloge est prévenue des modifications du modèle (heure et nom) par notification.

Les vues de saisi de l'heure et le nom de l'horloge passe par le contrôleur pour réaliser les actions.

Aucune des vues n'ont de lien avec le modèle.



On obtient le schéma UML suivant :



Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh04_11_ModeleMVC

7. Conclusion

Nous n'avons pas abordé tous les design patterns que nous avons lister en début de cette présentation car l'objectif n'était pas d'être exhaustif sur le sujet (on ne peut pas

l'être) mais par la présentation de certains de montrer ce qu'est vraiment un design patterns et avec des exemples de décrire le codage de certains de ces design patterns.

Nous avons vu que le design pattern n'est pas un élément du langage informatique comme une classe ou une instruction mais plutôt est une bonne pratique de conception et de codage.

L'objectif global de ces designs patterns est de rendre réutilisable des principes et des concepts en utilisant au mieux les propriétés des langages objets.

Pas de panique !! Si vous ne vous sentez pas capable de programmer en Java en utilisant les design patterns cela n'est pas un handicap. Ce qui est important est de savoir qu'ils existent et vous les verrez à force de pratiquer.

8. Annexes

8.1. Le Singleton dans un contexte multi-threadé

Attention, comme implémenté ici, deux threads qui demanderaient en même temps la création du singleton, pourraient obtenir chacun une instance différente du Singleton. Ce que l'on veut éviter.

Il faut donc **synchroniser la méthode getInstance** de création d'instance. Cela fonctionne très bien mais attention aux performances. Cela peut être pénalisant pour l'ensemble de l'application si cette méthode est souvent sollicitée.

Pour pallier ce problème de performance, on peut utiliser la technique du "Holder".

Cette technique consiste à utiliser une classe interne (le holder) :

```
public class Singleton
{
    /** Constructeur privé */
    private Singleton()
    {}

    /** Holder */
    private static class SingletonHolder
    {
        /** Instance unique non préinitialisée */
        private final static Singleton instance = new Singleton();
    }

    /** Point d'accès pour l'instance unique du singleton */
    public static Singleton getInstance()
    {
        return SingletonHolder.instance;
    }
}
```

Cette technique joue sur le fait que la classe interne ne sera chargée en mémoire que lorsque l'on y fera référence pour la première fois, c'est-à-dire lors du premier appel de "getInstance()" sur la classe Singleton. Lors de son chargement, le Holder initialisera ses champs statiques et créera donc l'instance unique du Singleton.

Cerise sur le gâteau, elle fonctionne correctement en environnement multithreadé et ne nécessite aucune synchronisation explicite