Université de Nice-Sophia Antipolis

Institut Universitaire de Technologie

Département Informatique

M315 - Conception et Programmation Orientées Objet Avancées

Carnet de bord

Auteurs :
Jeanne Marcadé
François Montigny

Enseignant :
Mme Mireille Blay-Fornarino

Lundi 9 Janvier 2017

Résumé

Ce document a pour but de présenter le travail réalisé dans ce module, ainsi que d'expliquer à d'autres développeurs l'art de la programmation pragmatique, avec l'utilisation de principes comme GRASP, SOLID ou encore l'utilisation de *design patterns*.

Sommaire

1 Introduction

> *"Any fool can write code that a computer will understand. Good programmers write code that humans can understand."*
>
> — Martin Fowler, 2008.

Développer une application qui fonctionne, c'est bien. Développer une application sûre, robuste, portable, maintenable, etc. c'est mieux. Résoudre le problème posé n'est qu'une partie du travail d'un développeur : il faut aussi s'assurer de produire des codes de qualité si l'on veut pouvoir comprendre, maintenir, mettre à jour et améliorer le logiciel produit sans se casser la tête.

Il y a autant de façons de programmer que de programmeurs. Certaines bonnes, d'autres mauvaises. Alors, voici quelques principes dont la validité est certifiée. Ce ne sont pas des solutions miracle, mais des outils puissants quand on les utilise bien.

Dans ce document, `Java` sera mis à l'honneur, mais les principes exposés sont indépendants du langage utlisé et sont, modulo quelques variations syntaxiques, également appliquables quelque soit le langage lorsque vous programmez en orienté-objet.

2 Les principes G.R.A.S.P.

> *"Object-oriented programming works like human organizations. Each object will communicate with another one by sending messages. So the software objects work by just sending those messages."*
>
> — Ron McFadyen, 2004.

G.R.A.S.P. est l'acronyme de *General Responsability Assignment Software Patterns*. Le mot à retenir ici est le mot *responsabilité*. Lorsque vous développez un logiciel en orienté objet, il faut imaginer que vos objets sont une communauté d'êtres humains qui collaborent pour travailler. Dans une telle communauté, qui fait quoi ? Chacun fait ce dont il est responsable. Et c'est ce principe que nous allons appliquer en programmation. En d'autres termes, chaque objet aura un "contrat", des obligations dont il devra s'acquitter.

Attention, une responsabilité n'est pas une méthode ! La relation est plus fine que cela : ce sont les méthodes qui se chargeront de remplir les responsabilités des objets.

Lorsque l'on fait de la conception dirigée par les responsabilités, il est important de comprendre de quelles responsabilités on parle. En fait, il en existe deux types. Un objet doit **savoir** et **faire**. Lorsqu'on dit qu'un objet doit savoir, cela peut vouloir dire :

— devoir se connaître lui-même (valeurs de ses attributs) ;
— connaître les autres objets auxquels il est rattaché ;
— connaître les objets qu'il peut calculer ou sur lesquels il peut agir.

Lorsqu'on dit qu'un objet doit faire, cela peut-être :

— effectuer une action (faire un calcul, créer un objet...) ;
— déclencher une action sur un autre objet ;
— contrôler et coordonner les actions d'un autre objet ;

Vous l'aurez compris, c'est une approche de la CPOO qui se veut aussi rationnelle que possible. On cherche à réduire au maximum la distance entre ce que l'on modélise (le réel) et ce qu'on utilise pour ce faire (le modèle de conception). Cela donne lieu à plusieurs *patterns*, ou **patrons de conception**, c'est à dire un couple composé d'un problème récurrent et de sa solution. Cette notion de *pattern* reviendra souvent... Mais chaque chose en son temps ! Pour l'heure, voici quelques *patterns* GRASP.

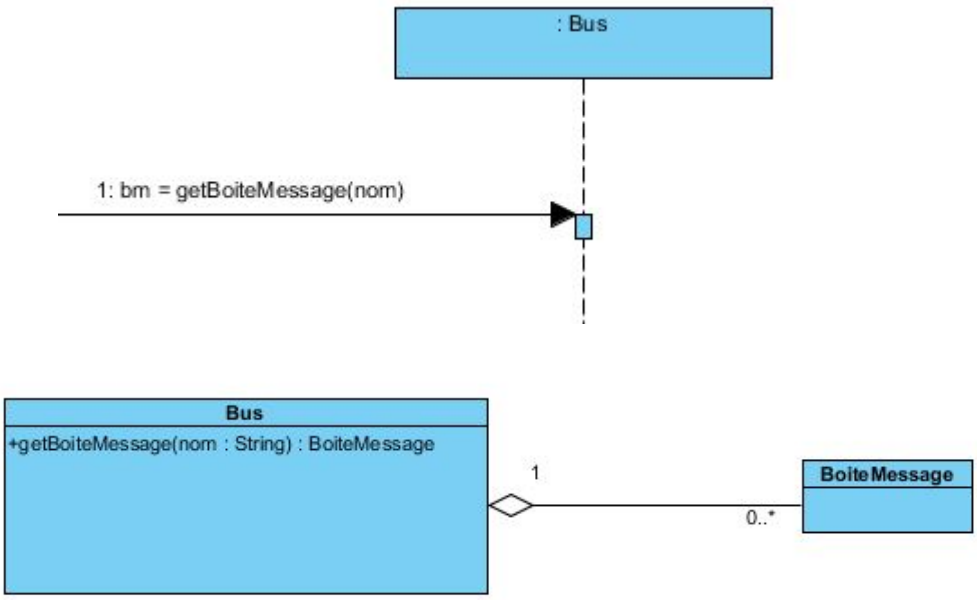
2.1 Information Expert

Problème. Comment décide-t-on les responsabilités des objets ?

Solution. On affecte la responsabilité à l'expert en termes d'informations, c'est à dire à la classe qui possède les informations nécessaires pour assumer la responsabilité. Une fois qu'on a trouvé qui est l'expert en information, on a trouvé le responsable.

Exemple. Prenons un système de bus de messages, qui permet à un "producteur" d'envoyer un "message". On considère qu'un bus peut avoir plusieurs boîtes de messages. Les messages peuvent se trouver dans les boîtes d'un bus de messages ou directement sur le bus lui-même (sans être classés dans une boîte).

Sachant cela, qui est responsable de retrouver une boîte de messages étant donné son nom ?



C'est le bus qui connait les boîtes de messages qui lui sont associées. C'est donc lui qui possède l'information nécessaire pour retrouver une boîte : c'est sa responsabilité.

Conclusion. Ce *pattern* est le plus utilisé de ceux d'attribution de responsabilité. C'est un principe de base en orienté-objet : il encourage à définir des classes plus légères, plus cohésives (les objets utilisent leurs propres informations pour mener à bien leurs tâches), plus faciles à maintenir et comprendre.

2.2 Low coupling

La notion de couplage sert à mesurer à quel point un élément est lié à un autre. Par exemple, A possède un attribut de type B, ou bien une méthode qui référence B... ou encore, B est une sous-classe de A, ou bien une interface qu'A implémente.

Pourquoi veut-on éviter le couplage fort ? Il pose plusieurs problèmes :

— une modification dans une classe force à changer la quasi-totalité, voire la totalité des classes liées ;
— les classes prises à part sont plus difficiles à comprendre ;
— la réutilisation d'une classe devient plus difficile, puisqu'il faut aussi employer toutes les classes liées.

Problème. Comment limiter les dépendances entre les classes, réduire l'impact des modifications et augmenter la réutilisation ?

Solution. Affecter une responsabilité de façon à ce que le couplage reste faible. Utiliser ce principe pour évaluer les solutions possibles.

Exemple. On peut reprendre rapidement l'exemple de la récupération de la boite de messages à partir d'un nom. Si une autre classe que le bus avait eu cette responsabilité (par exemple, la classe Message), on aurait eu trop de dépendance à la classe Message (dont on aurait eu besoin pour réaliser une tâche dont elle n'est pas responsable), et donc une mauvaise conception...

Conclusion. Comme souvent en informatique, il n'y a pas de règle absolue sur ce principe. On n'a pas d'outil de mesure pour dire qu'un couplage est trop fort ou trop faible : il faut regarder cela au cas par cas et être critique. De façon générale, les classes très réutilisables doivent nécessairement avoir un faible couplage. Mais il faut bien noter qu'avoir un fort couplage n'est pas dramatique si on a des éléments très stables, comme c'est le cas pour la bibliothèque `java.util`.

De même, il ne faut pas exagérer dans l'autre sens ! On peut se retrouver, dans des cas extrêmes de faible couplage, avec des objets complexes, incohérents, qui ne communiquent pas entre eux, qui servent uniquement à stocker des données... ce qui va à l'encontre des principes de l'orienté objet.

En bref, il ne faut savoir trouver un juste milieu.

2.3 High Cohesion

La cohésion fonctionnelle est une mesure de l'étroitesse des liens et de la spécialisation des responsabilités d'une classe. En d'autres termes, une classe dont les responsabilités sont étroitement liées et qui n'effectue pas un travail gigantesque (au contraire d'une "classe dieu") est forcément cohésive.

Pourquoi veut-on éviter une faible cohésion ? Les classes à faible cohésion ont plusieurs inconvénients de taille. Elles sont :

— difficiles à comprendre ;
— difficiles à réutiliser ;
— difficiles à maintenir ;
— instables, constamment affectées par le moindre changement.

Problème. Comment parvenir à maintenir la complexité globale ? Comment s'assurer que les objets restent compréhensibles, faciles à gérer, et participent au faible couplage ?

Solution. Affecter une responsabilité de façon à ce que la cohésion reste forte. Utiliser ce principe pour évaluer les solutions possibles.

Exemple. On pourrait trouver un exemple sur n'importe quelle classe. Aussi, voici simplement une illustration avec deux diagrammes de séquence (page suivante).

Conclusion. Il faut garder à l'esprit la même chose que pour le faible couplage : il n'y a pas de règle absolue sur le principe de forte cohésion. Cependant, on peut très vite vérifier la cohésion d'une classe. Au contraire d'une "classe dieu", une classe fortement cohésive a peu de méthodes et ne fait pas trop de travail ; ses fonctionnalités sont très liées entre elles. Pour tester cela, regardez si vous êtes capables de décrire votre classe en **une seule phrase.** Si vous n'y arrivez pas, c'est qu'elle nest pas assez cohésive.

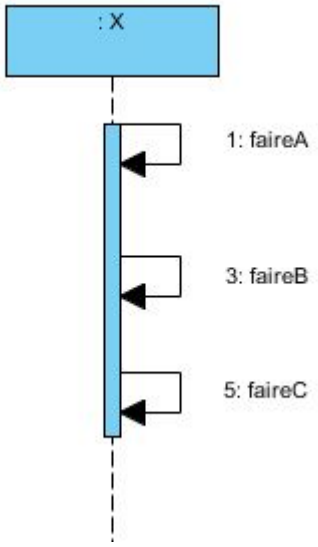


FIGURE 1 – Mauvais design (classe dieu)

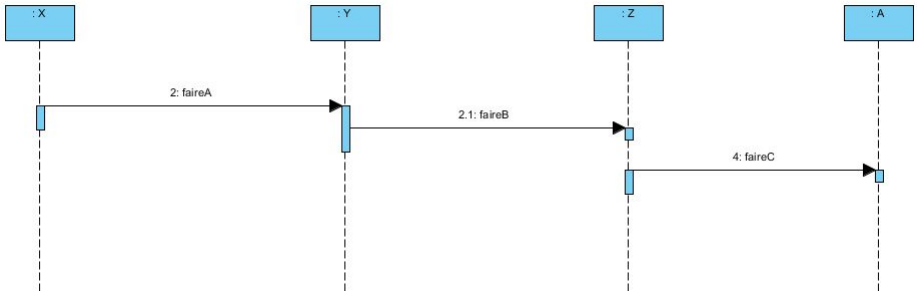


FIGURE 2 – Bon design

2.4 Creator

Problème. Qui doit se charger de créer une nouvelle instance d'une classe ?

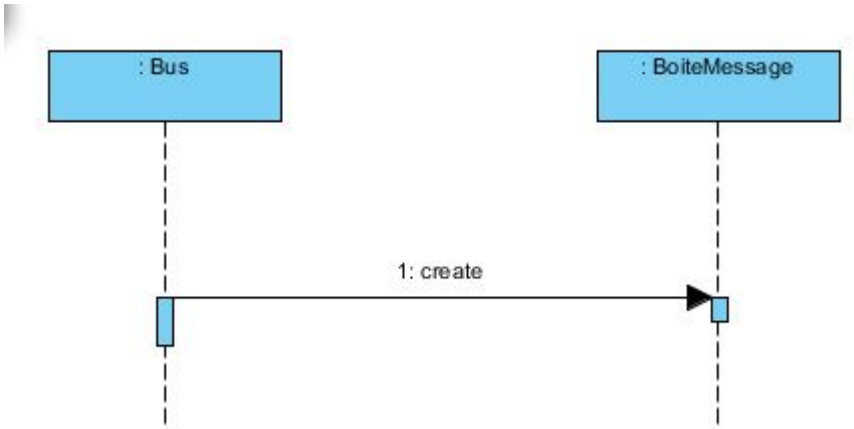
Solution. B est responsable de créer une nouvelle instance de A si l'une des conditions suivantes est vérifiée :

- B contient ou agrège des objets A ;
- B enregistre des objets A ;
- B utilise étroitement des objets A ;

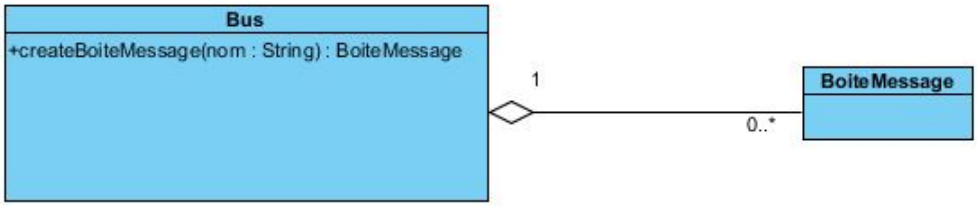
— B possède les données d'initialisation, qui seront transmises aux objets A lors de leur création.

On dit alors que B est un expert en ce qui concerne la création de A.

Exemple. Reprenons l'exemple du système de bus de messages. Qui a la responsabilité, intuitivement, de créer une boîte de messages ? Réponse : le bus, puisqu'il contient des boîtes de messages. Confirmons cette intuition avec un diagramme de séquence :



On remarque que cela nous amène directement au diagramme de classes, niveau conception.



Conclusion. Créer un objet est bien évidemment chose commune en orienté-objet, et ce *pattern* est un guide pour réaliser cette tâche. Dans quel but ? Pour savoir qui doit nécessairement être connecté aux objets créés. Ce *pattern*

est très lié aux notions de faible couplage, ainsi qu'aux *patterns* Composite et Factory. Ce dernier est d'ailleurs un des outils qui se révèlent très puissants lorsque la création d'objets devient complexe...

2.5 Controller

Problème. Quel est le premier objet, au delà de l'IHM, qui reçoit et coordonne une opération système, c'est à dire un évènement majeur entrant dans le système ?

Solution. Choisir ou inventer un objet dans la couche application qui aura ce rôle.

On appelle un tel objet un Contrôleur : il n'appartient pas à l'IHM, mais il a la responsabilité de recevoir ou de coordonner un évènement système. Une telle classe peut représenter le système dans son entièreté, une partie seulement, un équipement sur lequel le logiciel s'exécute, ou encore un scénario de cas d'utilisation dans lequel l'évènement système se produit. Les contrôleurs font le lien entre la couche "Présentation" (l'IHM) et la couche "Application" (ce que le logiciel fait). Vous pouvez voir le contrôleur comme un guide, qui indique aux objets ce qu'ils doivent faire, et quand ils doivent le faire, lors du déroulement d'un cas d'utilisation.

Il existe plusieurs types de contrôleurs :

— le contrôleur façade, qui représente tout le système. Il faut l'utiliser quand il y a peu d'évènements systèmes.

— Le contrôleur cas d'utilisation : un seul contrôleur pour tous les évènements d'un cas d'utilisation, un contrôleur par cas d'utilsation. Il faut l'utiliser quand les autres possibilités amènent à un contrôleur trop chargé (faible cohésion) ou lorsqu'il y a un grand nombre d'évènements système, répartis entre plusieurs processus : cela permet de les gérer plus facilement.

Un contrôleur **ne doit pas être trop chargé**. Si besoin, il peut (et doit) déléguer ses tâches à d'autres objets. Il effectue la majorité des tâches nécessaires pour répondre aux évènements systèmes... pas toutes ! N'oubliez pas le principe de responsabilité. Si on commence à avoir un contrôleur trop chargé en responsabilités ou en information, il faut ajouter des contrôleurs, voire concevoir des contrôleurs dont le seul rôle est de déléguer les responsabilités.

Exemple. Reprenons notre fameux système de bus de messages, et intéressons nous au cas où l'on veut créer des bus, ce qui est une responsabilité du registre des bus. La figure de la page suivante montre un exemple de contrôleur. On y définit des méthodes correspondant aux actions à effectuer, et dans le corps de celles-ci, on appelle les méthodes des entités responsables de l'exécution de ces actions.

```
1  public class Controleur {
2    RegistreDeBus registre = new RegistreDeBus();
3    UI ui = new UI();
4
5    public void start() {
6      String commande = ui.lireCommande();
7      switch (commande.charAt(0)) {
8        case 'c':
9          creerBus();
10         start();
11         break;
12       case 'b':
13         creerBoite();
14         start();
15         break;
16       /*
17       ...
18       */
19     }
20   }
21
22   private void creerBus() {
23     String nomDuBus = ui.getNomDuBus(registre.getNomsBus());
24     boolean reussi = registre.creerBus(nomDuBus);
25     if (!reussi)
26         ui.afficher("Error: this bus already exists.");
27     else
28         ui.afficher("Bus " + nomDuBus + " created");
29 }
```

FIGURE 3 – Un exemple de contrôleur

Conclusion. Un contrôleur permet de bien séparer les couches Présentation et Application tout en faisant le lien entre elles. Les données et responsabilités sont alors mieux réparties, ce qui aboutit à un meilleur couplage, une

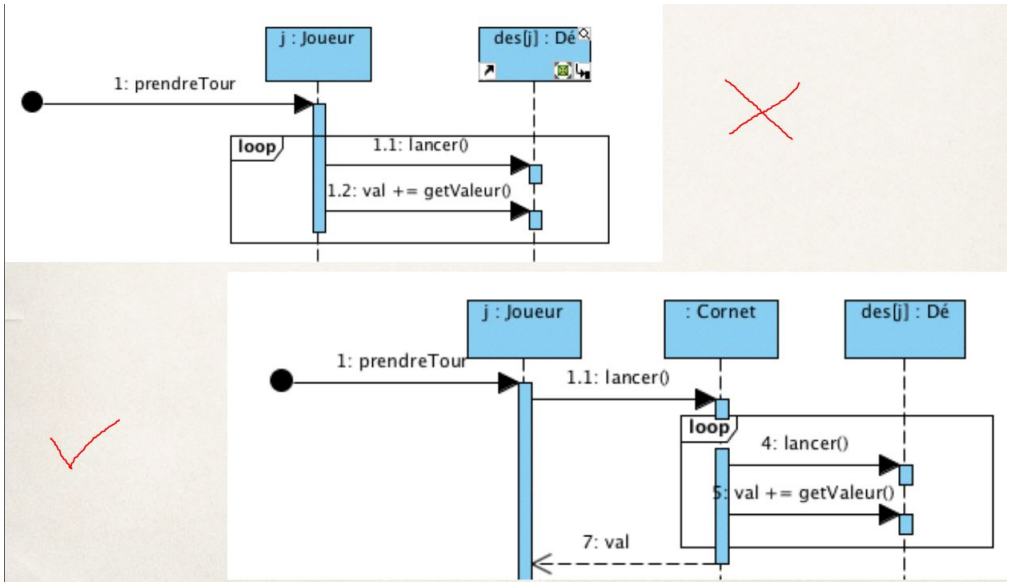
meilleure cohésion, et donc plus de potentiel de réutilisation. C'est également un excellent moyen de vérifier que "tout va bien", c'est à dire que les évènements systèmes se produisent dans le bon ordre lors d'un cas d'utilisation.

2.6 Fabrication pure

Problème. Que faire lorsque les concepts à modéliser ne sont pas utilisables tout en ayant un faible couplage et une forte cohésion ?

Solution. Créer une classe de toutes pièces, qui ne représente pas un concept du monde réel, et lui affecter un ensemble de données-responsabilités fortement cohésif.

Exemple. Comment modélise-t-on un lancer de dé par un joueur au Monopoly ?



Conclusion. Ce *pattern* permet de se tirer des situations délicates où l'on arrive pas à concilier faible couplage et forte cohésion au sein de notre modélisation. Attention toutefois : l'esprit de la CPOO repose sur les objets que l'on modélise, et pas sur les fonctions qu'ils remplissent... donc, il ne faut pas non plus abuser de cet outil.

3 Principes S.O.L.I.D. et programmation pragmatique

> *"Les développeurs avancés voient très vite l'intérêt, les débutants beaucoup moins. Quelques années plus tard, ils comprennent pourquoi c'était important !"*
>
> — Anonyme.

G.R.A.S.P. était une base, mais on peut aller plus loin avec des patrons et des principes de plus haut niveau. On s'intéresse ici surtout à écrire du bon code.

S.O.L.I.D. est un acronyme représentant cinq principes de programmation orientée objet dite pragmatique :

- Single-Responsability Principle : une classe n'a qu'une seule responsabilité.
- Open/Closed Principle : une classe doit être ouverte à l'extension (ex : héritage) mais fermée à la modification (ex : attributs privés).
- Liskov Substitution Principle : les objets d'un programme doivent pouvoir être remplacés par des instances de leurs sous-types sans "casser" le programme.
- Interface Segregation Principle : il vaut mieux avoir plusieurs interfaces spécifiques qu'une unique interface générique.
- Dependency Inversion Principle : il faut dépendre des abstractions, pas des réalisations concrètes.

3.1 Single-Responsability Principle

Principe. Ce principe rejoint ceux de G.R.A.S.P. sans les contredire. Il ne s'agit pas qu'une classe ait une seule responsabilité au sens d'une seule méthode (encore une fois : les méthodes ne sont pas des responsabilités, elles s'en acquittent), mais plutôt qu'une classe doit avoir de la responsabilité sur une, et une seule fonctionnalité du logiciel. Cette responsabilité doit être entièrement encapsulée par la classe, et tous ses services en lien étroit avec elle (forte cohésion).

Exemple. Imaginons une entité, comme une classe Livre, dont nous voudrions trier les objets. Une solution de facilité serait celle-ci :

```
1 public class Book implements Comparable<Book> {
2   /*
3   ...
4   */
5
6   int compareTo(Object o) {...}
7 }
```

Mais un livre ne sait pas comment il doit être trié ! Le client peut vouloir le trier par nom, par auteur, par date... Pire, chaque fois qu'on veut changer la façon de trier, il faut recompiler la classe Book ainsi que son client. Pourquoi ? Parce que deux responsabilités ont été mélangées dans la classe Book : la connaissance des informations relatives à un livre, et le tri de ceux-ci.

La bonne façon de faire est de créer autant de classes que de façons de trier les livres. Ces classes implémenteront l'interface `Comparator<Book>` et redéfinieront la méthode `compare`. Ainsi, on évite les problèmes mentionnés plus haut.

Conclusion. Ce principe renforce ceux vus dans le premier chapitre et nous apprend quelque chose de plus : lorsque l'on attribue les responsabilités, il ne faut pas les mélanger en termes de fonctionnalités.

3.2 Open/Closed Principle

Principe. Le concept, c'est de pouvoir étendre le comportement d'une classe sans la modifier : le code a été écrit et testé, plus besoin d'y toucher.

Exemple. Avec une telle conception, que se passe-t-il si on veut que la voiture ait un autre type de moteur ? Il faut changer la voiture... Ou alors, changer la conception.

En fait, il faut surtout retenir qu'**une classe concrète ne doit pas dépendre d'une autre classe concrète.** Elle peut dépendre d'une classe abstraite, et utiliser le polymorphisme.

On peut imaginer beaucoup d'autres exemples. Par exemple, si on définit des coefficients de modification (par exemple pour modifier des prix) en dur

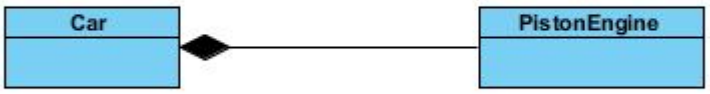


FIGURE 4 – Mauvais design

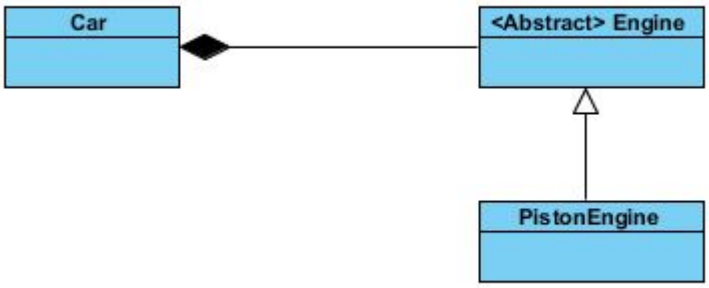


FIGURE 5 – Bon design

dans le code, que se passe-t-il si on veut un jour les changer ? Il faut modifier le code... Ou alors, on peut définir une classe abstraite qui gère la politique des prix.

Conclusion. Il est impossible que tous les éléments d'un système satisfassent ce principent. L'objectif, c'est d'en avoir le plus possible. Plus on a d'éléments qui satisfont ce principe, plus on peut réutiliser et maintenir facilement le logiciel.

3.3 Liskov Substitution Principle

Principe. Les instances d'une classe doivent être remplaçables par des instances de leurs sous-classes (ou classes filles) sans altérer le comportement du programme. En gros, ce qui est vrai pour une classe A doit rester vrai

pour toute classe héritant de A. Cela implique que le "contrat" défini par la classe doit être respecté par ses sous-classes, et surtout, que l'appelant n'a pas besoin de connaître le type exact de l'objet qu'il utilise : n'importe quelle classe dérivée fera l'affaire.

Cela rejoint un principe de base du polymorphisme : si on substitue une classe par une de ses sous-classes, le comportement aura bien sûr quelques variations, mais il restera identique dans les grandes lignes : il restera conforme.

Exemple. Pour mieux comprendre ce principe, prenons un exemple de ce qu'il ne faut pas faire. On s'intéresse à la modélisation d'oiseaux. Regardons le code suivant :

```
1  public class Bird {
2    public void fly(float distance) {...}
3  }
4
5  public class Eagle extends Bird {
6    public void fly(float distance) {...} //An eagle can fly
7  }
8
9  public class Penguin extends Bird {
10   public void fly(float distance) throws Exception {
11     throw new Exception("Penguins don't fly");
12   }
13 }
```

Le problème de ce code, et plus particulièrement de la classe `Penguin`, c'est qu'on ne modélise pas "Les pingouins ne savent pas voler" mais "Une erreur se produit lorsqu'un pingouin tente de voler".

En fait, pour éviter de violer ce principe, il faut se rappeler qu'**il ne faut pas, dans une sous-classe, redéfinir une méthode de la classe mère avec une méthode qui ne fait rien** (une "NOP method"). Il y a deux façons d'éviter cela :

- inverser l'héritage (on peut s'être trompé dans notre modèle) ;
- ajouter des classes supplémentaires pertinentes dans la hiérarchie. Dans notre exemple, il faudrait une classe `FlyingBird` qui héritera de `Bird` et qui se chargera de définir la méthode `fly()`.

Conclusion. Ce principe n'est pas très difficile à comprendre, et est même assez intuitif : lorsque l'on fait de l'Héritage, il faut que cela soit cohérent. Notons que c'est une invitation à aller plus loin, et à reconsidérer l'utilisation de l'héritage : parfois, il vaut mieux privilégier la composition...

3.4 Interface Segregation Principle

Principe. Plusieurs interfaces spécifiques à différents clients valent mieux qu'une seule interface générale. Dans l'autre sens, un client doit avoir une interface avec uniquement ce dont il a besoin. Et pas plus ! Cela incite à ne pas créer d'interface sans réfléchir, et à faire des interfaces limitées en taille afin d'éviter de se retrouver à implémenter des méthodes dans des classes qui n'en ont pas besoin. Là encore, il faut éviter d'exagérer dans l'autre sens et de faire des interfaces à une méthode : la cohésion ne doit pas être sacrifiée. Il faut donc être pragmatique, utiliser son expérience, et même, beaucoup plus simplement... utiliser le bon sens !

Exemple. On s'intéresse à la modélisation de véhicules de différents types (bateau, voiture, sous-marin...). On considère l'interface suivante :

```
1  /**
2  * Interface VehiculeInterface provides
3  * the specification for a vehicle.
4  */
5
6  public interface VehiculeInterface {
7    public void turnLeft();
8    public void turnRight();
9    public void forward();
10   public void reverse();
11   public void climb();
12   public void dive();
13   public void fly();
14   public void setSpeed(double speed);
15   public double getSpeed();
16 }
```

Voyez-vous venir le problème ?

Un avion n'a aucun besoin d'escalader quoi que ce soit, pas plus qu'un sous-marin... une voiture ne volera jamais bien longtemps, et si un bateau se

```
1 public class Car implements VehiculeInterface {...}
2
3 public class Plane implements VehiculeInterface {...}
4
5 public class Boat implements VehiculeInterface {...}
6
7 public class Submarine implements VehiculeInterface {...}
```

retrouve en dessous du niveau de la mer, c'est soit qu'il est en cale-sèche aux Pays-Bas, soit qu'il y a un problème.

Il faudrait ici avoir autant d'interfaces que de types de véhicules. Par exemple `LandVehicule`, `AerialVehicule`, `NavalVehicule`, etc.

Conclusion. En fait, ce principe peut-être rapproché du principe précédent, et plus précisément de la technique qui consiste à enrichir la hiérarchie (exemple des pingouins). Parfois, on exhibe des classes "intermédiaires" dans la hiérarchie d'un héritage, parfois on exhibe plutôt des interfaces. On se base surtout sur le bon sens pour créer ces interfaces : la séparation des véhicules en différentes catégories est assez évidente. Lorsque ça l'est moins, il suffit d'inspecter les méthodes de l'interface et les objets qui l'implémentent avec un peu d'esprit critique pour savoir si le principe est respecté ou non.

Il est important de retenir que l'on programme vers une interface (un client) et pas une implémentation (pas tous les clients).

3.5 Dependency Inversion Principle

Principe. L'idée est de réduire les dépendances sur les classes concrètes : comme on l'a vu avec le principe Open/Closed, une classe concrète ne doit pas dépendre d'une autre classe concrète, mais d'une abstraction. C'est ce que permet l'inversion de dépendance : si une classe concrète A dépend d'une classe concrète B, on peut faire en sorte, à la place, que A spécifie les détails d'une interface I que B va alors implémenter.

L'objectif est donc de faire en sorte que toute classe concrète ne dépende que d'abstractions, y compris les classes de bas niveau. Ce principe est une technique pour respecter le principe Open/Closed !

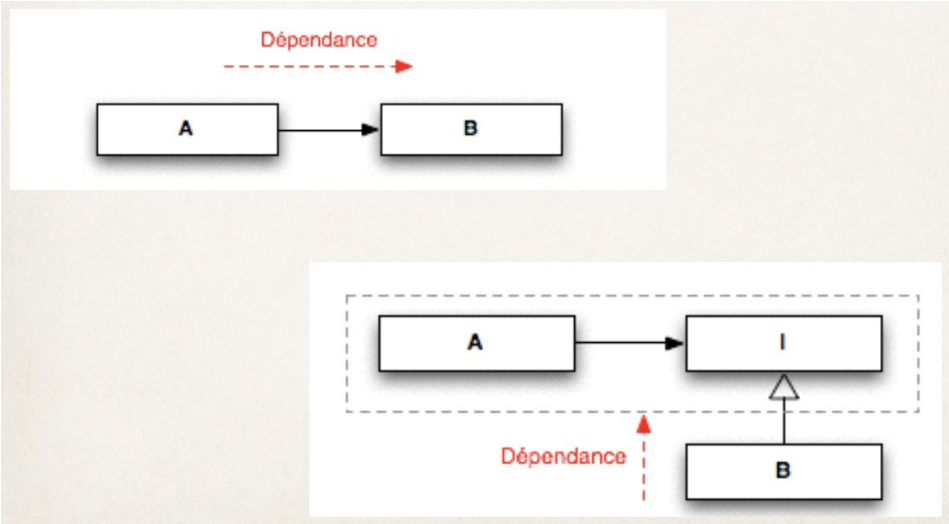
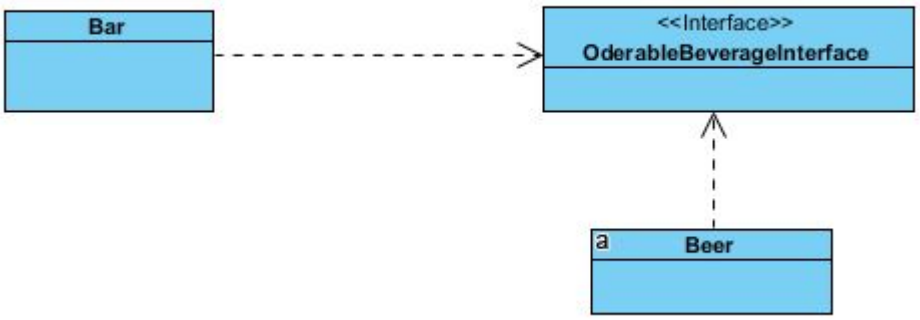


FIGURE 6 – Illustration du principe d'inversion de dépendance

Exemple. Prenons un exemple très simplifié d'un bar. On y boit, entre autres, des bières. Que se passe-t-il si on fait cela ?

On a une dépendance entre deux classes concrètes... Pire encore, soit ce bar ne propose que des bières, soit le diagramme UML complet d'un tel modèle ne doit pas être beau à voir (une association avec Bar par type de produit...). En appliquant l'inversion de dépendance, on aboutit à un modèle bien plus cohérent (voir page suivante).

Conclusion. En fait, ce principe est un genre de "couteau suisse" pour se tirer de mauvais pas : si on a fait une erreur de conception et qu'on a violé le principe Open/Closed, on peut retomber très élégamment sur nos pattes avec une inversion de dépendance. C'est d'une efficacité redoutable !



4 Polymorphisme et composition

"Le polymorphisme, ce n'est pas une maladie contagieuse."

— Mireille Blay-Fornarino, 2016.

Ce chapitre a deux objectifs : d'une part, revenir sur un principe fondamental de l'orienté objet, à savoir le polymorphisme, afin de mieux comprendre sa portée et ses différentes utilisations possibles ; d'autre part, étudier l'une de ces applications en détail et s'intéresser ce faisant à une autre idée : l'art de bien programmer.

4.1 Le polymorphisme

Principe. De *poly* pour plusieurs et *morphe* pour la forme, le polymorphisme désigne la possibilité d'avoir plusieurs formes. Mais encore ?

En fait, le polymorphisme, c'est la possibilité de donner le même nom et la même signature (modulo le type de retour, sans influence) à des méthodes dans différents objets... même si ces derniers sont associés d'une quelconque façon. Ce principe de polymorphisme, que nous avons déjà évoqué plusieurs fois, permet de se placer dans un cercle vertueux basé sur l'ignorance, ou, plus précisément, sur le désintérêt. Peu importe le type d'un objet, si on sait qu'il possèdera forcément tel service !

L'idée de base du polymorphisme, c'est de permettre de gérer des variations de différents comportements proches (on retrouve le principe de substitution de Liskov et notre exemple des pingouins) sans devoir tester le type :

il suffit juste de redéfinir une méthode en prenant en compte les variations souhaitées, et le tour est joué.

Applications. Plutôt que de faire une longue liste, soyons simples : sans le polymorphisme, pas grand chose ne fonctionnerait en orienté-objet. Et pour cause : le polymorphisme en est un des principes fondamentaux. Notamment, il devient indispensable lorsque l'on veut faire de l'héritage, appliquer le principe de substitution de Liskov, etc.

Exemple. On cherche à simuler le comportement de divers véhicules, certains pouvant porter une charge, d'autres non. En particulier, on s'intéresse au calcul de leur vitesse maximale, chaque véhicule ayant différentes contraintes en fonction de la charge transportée.

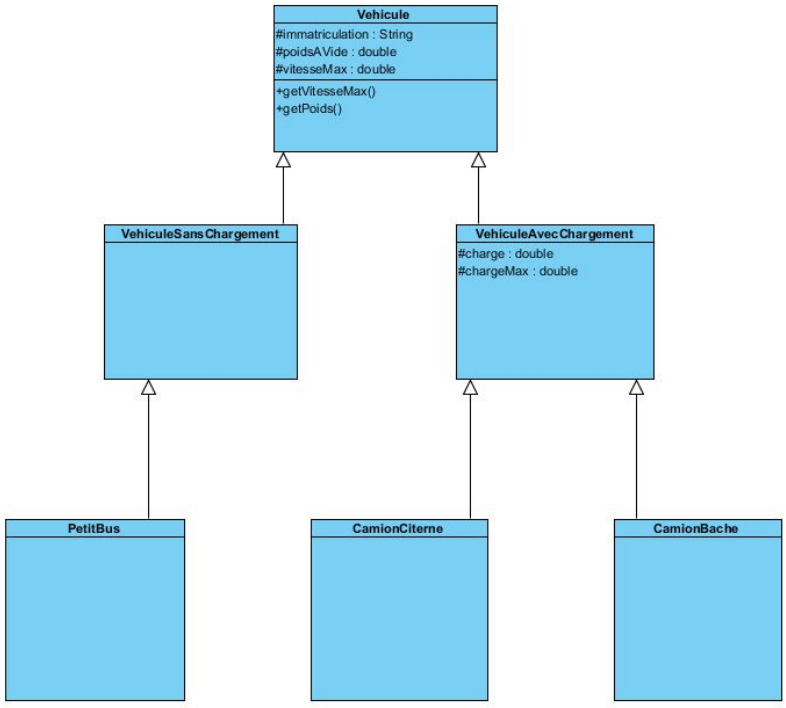


FIGURE 7 – Un premier modèle

Plutôt que de faire une seule méthode `getVitesseMax()` dont le corps sera une abomination à base d'`instanceof`, utilisons le polymorphisme ! On peut définir autant de méthodes que nécessaire :

```
1  public class PetitBus extends VehiculeSansChargement {
2    public double getVitesseMax() {
3      return vitesseMax;
4    }
5  }
```

FIGURE 8 – Méthode `getVitesseMax()`, très simple, de la classe `PetitBus`

```
1  public class CamionCiterne extends VehiculeAvecChargement {
2    public double getVitesseMax() {
3      double res = 0;
4
5      if (charge == 0)
6        res = 130;
7      else if (charge <= 1)
8        res = 110;
9      else if (charge > 1 && charge <= 4)
10       res = 90;
11     else
12       res = 80;
13
14     return res;
15   }
16 }
```

FIGURE 9 – Méthode `getVitesseMax()`, un peu plus complexe, de la classe `CamionCiterne`

Et de même (modulo les conditions) pour la classe `CamionBache`. En bref, `v.getVitesseMax()` fonctionnera toujours comme il faut, sans test supplémentaire, sans cast, et ce, que `v` soit un petit bus, un camion bâché ou autre !

Conclusion. Faire de l'orienté-objet sans utiliser le polymorphisme, c'est aller dans le mur. D'une part, c'est extrêmement simple et ça évite beaucoup de soucis d'écriture (amalgame d'`instanceof`), et d'autre part, une grande partie des principes énoncés dans ce document repose sur le polymorphisme...

4.2 Composition versus héritage

Principe. L'héritage est très souvent mis en avant pour la réusilisation. En fait, il est même trop mis en avant. Dans bon nombre de cas, la composition est une meilleure alternative, car plus souple. On retiendra qu'il n'y a pas de risque à utiliser l'héritage dans un même package, lorsque les implémentations des sous-classes et des superclasses sont contrôlées par les mêmes programmeurs, ou lorsque l'on étend des classes conçues et documentées pour être étendues.

Du côté de la composition, comment ça marche ? C'est un autre principe de réutilisation qui consiste à créer un objet composé d'autres objets, ce qui permet d'obtenir une nouvelle fonctionnalité en déléguant la responsabilité à cet objet composé.

Cette notion peut même aller plus loin (aggrégation), mais c'est inutilement compliqué pour ce que nous voulons exposer ici.

Exemple. Reprenons notre simulation de véhicules. On s'intéresse désormais à la notion de convois, c'est à dire un ensemble composé de véhicules de différents types. On souhaite pouvoir calculer la vitesse maximale d'un convoi, qui est la plus petite des vitesses maximales des véhicules du convoi.

0n pourrait considérer qu'un véhicule est un convoi de taille 1, et faire en sorte que `Vehicule` hérite de `Convoi`... Mais que se passe-t-il si on fait cela ? On aura un modèle peu cohérent, et un logiciel très fragile : à la moindre modification de `Convoi`, on risque de casser les sous-classes, qui s'élèvent déjà au nombre de 6 (et c'est sans compter les véhicules que l'on pourrait rajouter plus tard). En plus de cela, on mélange de façon hasardeuse deux responsabilités différentes (trouver une vitesse max en fonction d'une charge et trouver une vitesse max en fonction des véhicules d'un convoi)... Bref, il faut ici préférer la composition, bien plus logique et bien plus adaptée d'un point de vue qualité.

Conclusion. Si l'héritage permet d'écrire des classes très facilement, c'est à double tranchant : on est obligé de suivre le contrat imposé par la classe-mère, et si on change cette dernière, il est probable que l'on soit obligé de changer les classes-filles aussi... Accessoirement, les données sont moins bien encapsulées, car connues par d'autres classes (les classes-filles).

De l'autre côté, la composition permet une bonne encapsulation, réduit les dépendances, recentre les responsabilités... Beaucoup de bons côtés, mais

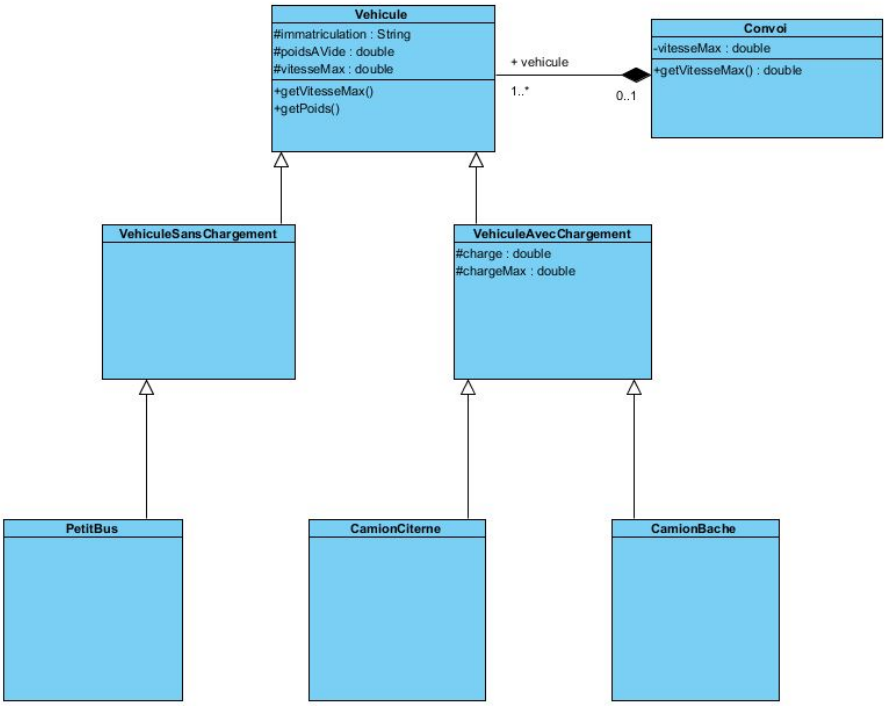


FIGURE 10 – Modèle avec convoi

un petit bémol néanmoins : les systèmes résultants ont tendance à avoir plus d'objets.

Savoir choisir avec discernement entre composition et héritage est un des principes importants des différentes facettes de l'art de bien programmer, avec d'autres (éviter les duplications de code, optimisation, estimation des algorithmes, qualité des commentaires...). Nous ne les développerons pas plus dans ce document, mais nous vous invitons à faire des recherches sur le sujet si vous voulez en savoir plus.

5 Les design patterns

> *"A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*
>
> — Christopher Alexander, professor of architecture

Revenons sur la notion de *patterns* (couple problème-solution). On peut aller plus loin que les *patterns* G.R.A.S.P. avec des outils de plus haut niveau. Certains d'entre eux découlent d'ailleurs directement des principes G.R.A.S.P. En fait, cela va plus loin qu'un simple couple problème-solution : les *design patterns* précisent dans quelles situations suivre les principes de Gamma (notamment celui de la ségrégation d'interfaces et l'utilisation de la composition plutôt que l'héritage) et comment les appliquer. Attention toutefois, car les directives que nous allons présenter sont propres à `Java` et, pour le coup, les *patterns* peuvent être utilisés différemment selon le langage de programmation utilisé.

Voici donc quelques uns des *patterns* définis par le "gang des quatre" et répartis en trois catégories (Création, Structuration et Comportement). Nous ne les présenterons pas tous, mais nous espérons que ces trois-là vous donneront envie d'en savoir plus.

5.1 Factory et Singleton

Commençons avec deux *patterns* plutôt simples que nous avons "teasé" dans la section dédiée au *pattern* Creator.

Problème. On veut pouvoir construire des objets sans dépendre de la classe définissant ces objets. Comment faire (si l'on veut ensuite pouvoir changer cette classe) ?

Solution. Une solution est de déléguer cette responsabilité à un autre objet, que l'on appelle une Factory. On utilisera cette classe plutôt que de faire `new` : chaque fois que l'on voudra créer un objet, on appellera une méthode

`static` de la Factory, qui se chargera de faire `new`. C'est également à elle de stocker les objets créés (si cela est nécessaire).

Exemple. On considère un jeu où l'on peut manipuler des personnages qui possèdent des armes. Une fois le jeu officiellement sorti, pour continuer de générer des revenus, on veut pouvoir ajouter de nouvelles armes sans modifier l'ensemble du jeu, ni les classes d'armes déjà définies. On veut aussi ajouter la possibilité de nommer les armes.

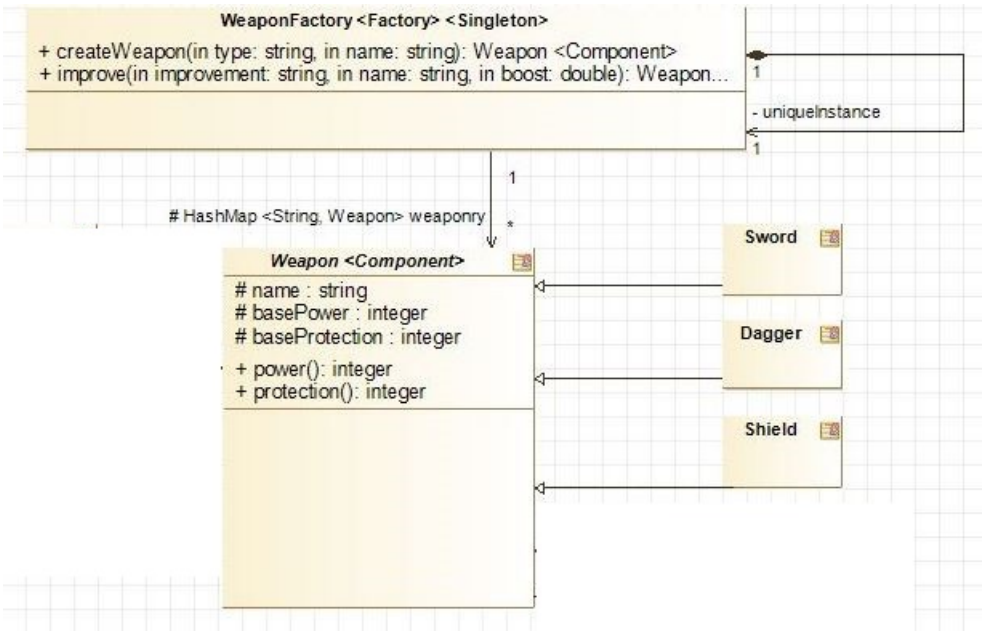


FIGURE 11 – Présentation des *patterns* Factory et Singleton

Ainsi, on ne dépend plus de `Weapon` lorsque l'on veut créer une arme : on évite ainsi les dépendances entre classes concrètes, ce qui est préférable (rappelez vous d'Open/Closed !). On n'a pas besoin de connaitre le type exact de ce que l'on fabrique.

Autre problème. On veut qu'il ne soit possible de construire qu'une seule instance d'un objet. Comment faire ? En fait, ce problème est étroitement lié à la notion de Factory. En effet, il faudra bien créer la Factory à un moment ou à un autre. Que se passe-t-il si quelqu'un en a déjà créé une ? On a créé

deux objets là où un seul suffit. Pire encore, si la Factory a la responsabilité de stocker ses créations, les objets créés seront éparpillés entre les différentes Factories ! D'où la nécessité d'une unique instance...

Solution. On utilise un autre *pattern*, celui du Singleton. On utilise une méthode *static* (car c'est la classe, et non les objets, qui est responsable de gérer son nombre d'instances). Cette méthode retournera l'unique instance de la Factory ; si nécessaire, elle la créera ! Mais si elle existe déjà, pas de création supplémentaire. On a donc bien une unique instance.

Exemple. Dans notre exemple de la Factory d'armes, voici ce que ça donne :

```
1  public class WeaponFactory {
2    public HashMap<String, Weapon> weaponry = new HashMap<~>();
3    private static WeaponFactory uniqueInstance;
4
5    public static WeaponFactory getInstance() {
6      if (uniqueInstance == null)
7        uniqueInstance = new WeaponFactory();
8      return uniqueInstance;
9    }
10 }
```

Ainsi, on centralise parfaitement à la fois la création et l'enregistrement des armes. Cela permet :

- de réduire les dépendances entre classes concrètes ;
- de réduire les changements à apporter si on veut modifier la création de `Weapon` ;
- de retrouver facilement une arme.

Notez, enfin, que la classe `WeaponFactory` porte les mentions `<Factory>` et `<Singleton>`. À chaque fois que l'on utilise un *design pattern*, notre modèle UML doit le refléter et c'est comme cela qu'on le représente.

Conclusion. Le responsable de la création d'un objet n'est pas forcément celui qui possède les informations nécessaires à cela : on peut aussi exhiber un objet à qui on va directement donner cette responsabilité. Cela ne vous rappelle rien ? Si... La fabrication pure !

5.2 Observer

Problème. Comment supporter une relation "un vers plusieurs" de façon à ce que plusieurs objets puissent être notifiés du changement d'état d'un objet, et puissent réagir en conséquence ? C'est un problème très courant en IHM.

Solution. Dans un premier temps, il faut identifier les rôles : il y a **des** observateurs, qui observent **un** sujet. Le sujet peut changer d'état et prévenir qu'il l'a fait, en donnant son nouvel état. Les observateurs, eux, peuvent "s'abonner" à un sujet (ou s'en désabonner), être notifiés d'un changement et obtenir le nouvel état du sujet. Cela donne non pas deux, mais quatre classes :

- le sujet abstrait. Il gère les observateurs (méthode `addObserver()`) et prévient (méthode `notify()`) de son changement d'état.
- Le sujet concret. Il peut changer d'état (en appelant `notify()`) et donner son nouvel état (méthode `getEtat()`).
- Les observateurs abstraits. Ils sont notifiés (méthode `update()`).
- Les observateurs concrets. Ils peuvent obtenir le nouvel état du sujet concret.

Le sujet (respectivement observateur) abstrait est responsable de gérer tout ce qu'on sait faire quel que soit le sujet (respectivement observateur). Le sujet (respectivement observateur) concret gère tout ce qui est particulier à chaque sujet (respectivement observateur).

Exemple. Prenons l'exemple de deux réseaux sociaux. Mon réseau social, IUTGo, possède certains membres du réseau social FacebookGhost. Chaque fois qu'une nouvelle relation est ajoutée dans FacebookGhost, on veut en être informé dans IUTGo, et vérifier si on ne peut pas reproduire cette relation (si IUTGo possède les deux membres).

FacebookGhost est notre sujet, et IUTGo sera un observateur. Voyons ce que cela donne, avec une petite partie du diagramme modélisant le système complet (en page suivante).

Et un exemple pour la méthode `update()` (plus loin).

Conclusion. C'est un *pattern* un peu plus compliqué à comprendre et à mettre en place, mais le jeu en vaut la chandelle. Ce problème d'observation

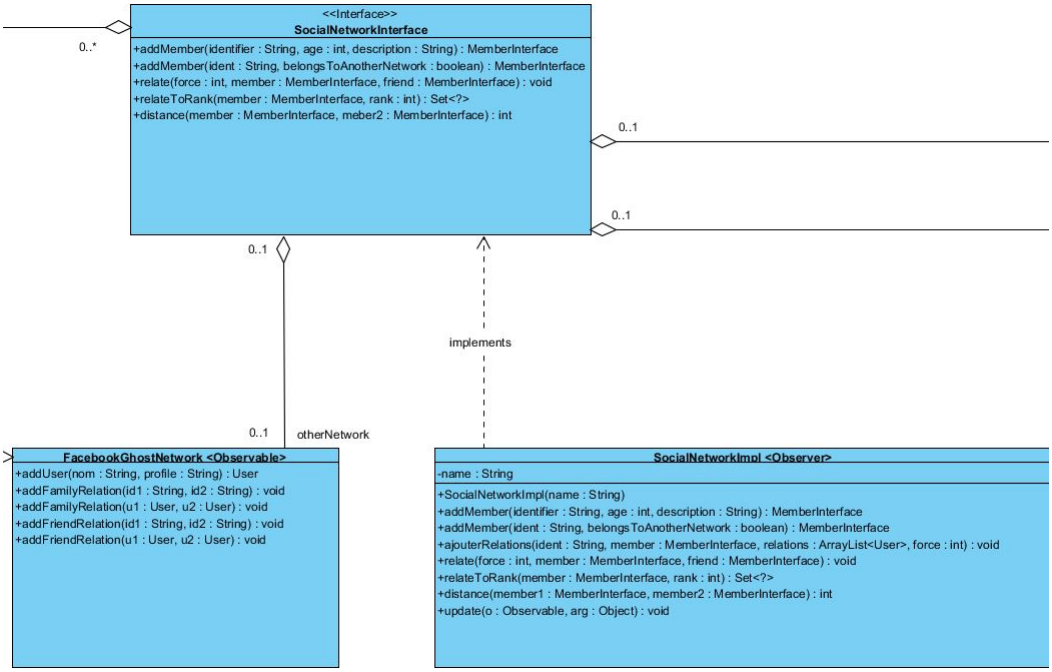


FIGURE 12 – Modélisation du *pattern* Observer sur un exemple simple

est monnaie courante en IHM : le *pattern* Observer devient alors un véritable kit de survie ! Et il peut aussi servir ailleurs, comme le montre l'exemple.

5.3 Composite

Problème. Comment manipuler des collections d'objets dont certains sont "composites" et d'autres "primitifs" ? Ces objets doivent tous répondre à une méthode, dont le comportement sera différent selon que l'objet soit composite ou non...

Solution. Organiser les objets dans un arbre, qui capture la hiérarchie "Composant-Composé"). On peut alors s'adresser à tous les objets de façon uniforme. Là encore, on définit quatre rôles :

— Component : déclare l'interface des objets pris en compte dans la composition, et, autant que possible, implémente le comportement par défaut des composants.

```
1  public void update(Observable o, Object arg) {
2    if (arg instanceof RelationEvent) { //RelationEvent est l'ajout
3                                        //d'une relation dans FG
4      User u1 = ((RelationEvent) arg).getU1();
5      User u2 = ((RelationEvent) arg).getU2(); //Membres de FG
6      if (((this.getMember(u1.getId())) != null)
7          && (this.getMember(u2.getId()) != null)) {
8        if (((RelationEvent) arg).getNature().equals("family")) {
9          relate(2, this.getMember(u1.getId()),
10                   this.getMember(u2.getId()));
11         relate(2, this.getMember(u2.getId()),
12                   this.getMember(u1.getId()));
13       }
14       else if (((RelationEvent) arg).getNature().equals("Friend"))
15       {
16         relate(3, this.getMember(u1.getId()),
17                   this.getMember(u2.getId()));
18         relate(3, this.getMember(u2.getId()),
19                   this.getMember(u1.getId()));
20       }
21     }
22   }
23 }
```

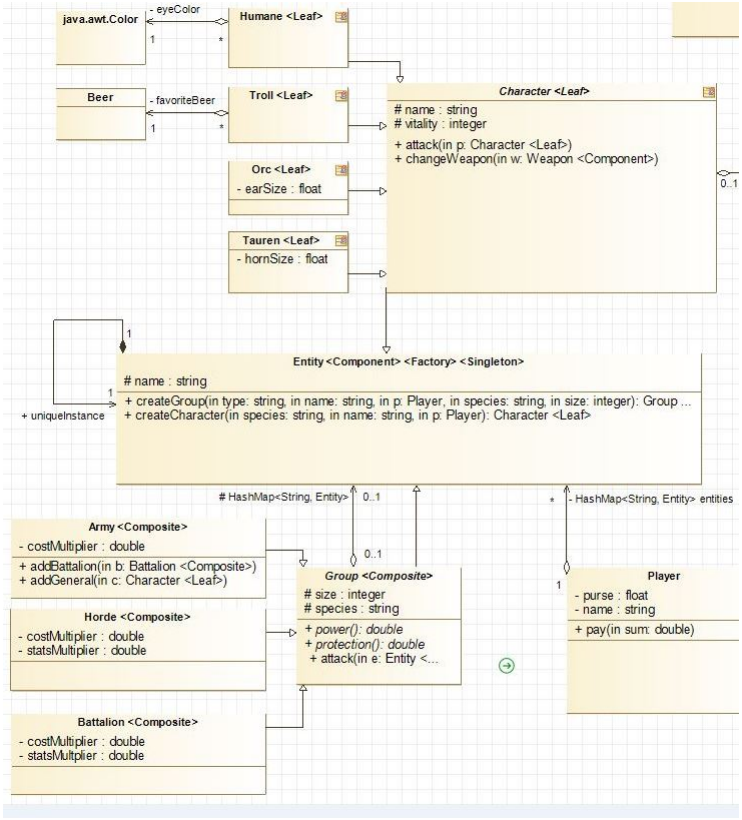
FIGURE 13 – Un exemple de méthode `update()`

— Composite : définit le comportement des composants ayant des "enfants" dans la hiérarchie de la composition, et référence les composants fils (qui peuvent, à leur tour, être composites !)
— Leaf : définit le comportement des objets primitifs dans la composition.
— Client : manipule les objets de la composition au travers de l'interface Component.

Exemple. Reprenons notre jeu. On souhaite que les personnages puissent désormais se rassembler en groupes. Différents types de groupes sont possibles (armée, bataillon, horde) avec différentes caractéristiques, et les personnages peuvent être de différentes races (humain, troll, orque...).

— Le Component est l'`Entity`, une superclasse regroupant et créant les `Group` ainsi que les `Character`.
— Le Composite est ici la notion de `Group`. Les trois sous-types de `Group`

sont donc également composites.
— Les Feuilles, ou Leaf, sont les objets "primitifs", à savoir `Character` et ses quatre sous-types.
— Le Client est ici la classe `Player`, qui demande à créer des personnages ou des groupes.



Conclusion. Ce pattern est très puissant lorsque l'on veut représenter des hiérarchies d'individus, ou lorsque l'on ne veut pas que le client ait à se préoccuper de la différence entre un objet composite et un objet primitif, pour pouvoir les traiter de façon uniforme.

6 Conseils pour s'améliorer

> *"No matter how good you think you are, there's always somebody quicker, faster, and a helluva lot smarter than you just around the corner.'*
>
> — David Anderson, Mass Effect.

Voici quelques astuces à retenir pour progresser, ainsi que les erreurs à ne pas commettre, afin d'avoir de bonnes bases !

6.1 Astuces et conseils

- Faites preuve de rigueur. N'hésitez pas à passer le temps qu'il faut sur un problème ! On ne peut être rapide que lorsqu'on maîtrise... ça ne sert à rien de se précipiter en pensant qu'il faut absolument faire vite.
- Un design pattern, c'est un outil, pas une solution miracle. Il faut l'utiliser en réponse à un problème précis, et pas dans n'importe quel situation.
- Les tests unitaires, c'est bien pour valider des choses, mais pas forcément pour débugger. N'hésitez pas à utiliser une fonction main dans vos classes, afin d'avoir un bac à sable à votre disposition lorsque vous cherchez à résoudre des bugs. Une fois ceux-ci dépistés et résolus, vous n'aurez qu'à enlever le main.
- Soyez curieux ! Documentez-vous ! On apprend mieux lorsque l'on s'intéresse un minimum aux choses.

6.2 Erreurs à ne pas commettre

- Ne pas oublier les "légendes" (Composite, Factory...) sur les diagrammes UML, SURTOUT quand on utilise des *design patterns*.
- Ne pas oublier de mettre à jour un diagramme UML lors d'une modification importante. Il faut que le modèle soit cohérent par rapport au code...
- Eviter les classe Dieu.
- Eviter les contrôleurs trop chargés (encore un genre de classe Dieu).
- Ne faites pas d'`instanceof` là où vous pouvez faire du polymorphisme !

— Ne soyez pas obsédés par l'idée d'appliquer un principe en particulier : soyez pragmatiques, et sachez décider avec discernement de quelle technique utiliser.

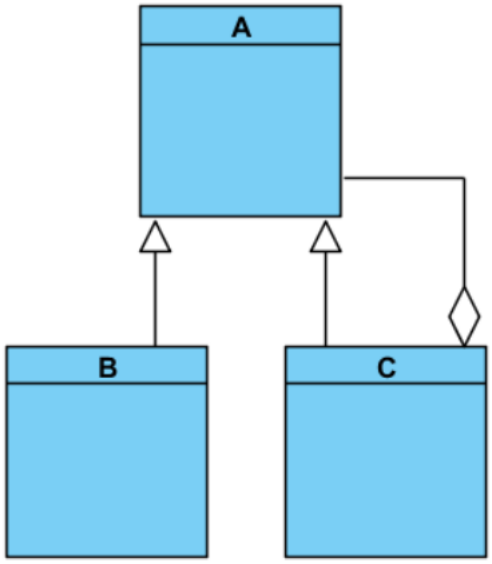
7 Un QCM pour s'auto-évaluer

Si vous voulez vérifier si vous avez bien compris les principes expliqués dans ce document, nous vous proposons un QCM pour vous auto-évaluer.

7.1 Questions

1. Quelle notion retrouve-t-on le plus dans les principes G.R.A.S.P. ?
   ☐ Réalisation
   ☐ Spécification
   ☐ Créativité
   ☐ Responsabilité
   ☐ Adaptation
2. Lesquels de ces *design patterns* font partie des principes G.R.A.S.P. ?
   ☐ Open/Closed Principle
   ☐ Decorator
   ☐ Creator
   ☐ Single-Responsability Principle
   ☐ Low Coupling
   ☐ Information Expert
3. Que doit-on faire pour respecter le principe de substitution de Liskov (LSP) ?
   ☐ On doit pouvoir remplacer des instances d'une classe par des instances de ses classes dérivées, sans casser le programme.
   ☐ Aucune nouvelle exception ne doit être générée par les méthodes des classes filles.
   ☐ Utiliser `instanceof` en `Java` pour gérer tous les types.
   ☐ Redéfinir les arguments des méthodes dans les classes dérivées.
   ☐ Nommer les méthodes et attributs en russe.

4. Le *pattern* Factory permet de :
   ☐ Modéliser une usine.
   ☐ Limiter la dépendance entre les classes.
   ☐ S'affranchir du type d'objet à créer.
   ☐ Créer des objets plus vite.
5. Selon le langage de programmation, l'utilisation de tel ou tel patron de conception peut changer.
   ☐ Vrai.
   ☐ Faux.
6. Qu'est-ce que le polymorphisme ?
   ☐ Une maladie contagieuse.
   ☐ L'autre nom pour l'utilisation d'`instanceof`.
   ☐ La possibilité de donner le même nom et la même signature (sauf valeur de retour) à une méthode dans différents objets.
   ☐ Un outil qui permet d'appliquer certains `patterns`.
7. Qu'est-ce qu'un paradigme en programmation ?
   ☐ La capacité de prendre en compte les besoins initiaux du client et ceux liés aux évolutions.
   ☐ La façon de penser et de structurer un programme.
   ☐ La possibilité de résister aux erreurs de l'utilisateur.
   ☐ Le fait d'être soucieux de la réussite du développement par la pratique.
8. Quel *pattern* reconnait-on ici ? Retrouvez le nom donné à chacune des classes.
   ☐ Observer.
   ☐ Factory.
   ☐ Composite.
   ☐ Singleton.



7.2 Correction

1. Responsabilité (d).
2. Creator, Low Coupling et Information Expert (c, e et f).
3. On doit pouvoir remplacer des instances d'une classe par des instances de ses classes dérivées, sans casser le programme (a).
4. Limiter la dépendance entre les classes et s'affranchir du type d'objet à créer (b et c).
5. Vrai (a).
6. La possibilité de donner le même nom et la même signature (sauf valeur de retour) à une méthode dans différents objets, et un outil qui permet d'appliquer certains `patterns` (c et d).
7. La façon de penser de structurer un programme (b).
8. Composite (c). A = Component, B = Leaf, C = Composite.

8 Le mot de la fin

Nous avons expliqué beaucoup de principes permettant d'aboutir à une bonne conception orientée objet, à du code et des logiciels de qualité... Mais

il en reste beaucoup d'autres !

Si vous ne deviez retenir qu'une seule chose de ce carnet de bord, ce serait celle-ci : il y a beaucoup de principes en orienté-objet. On ne peut pas toujours tous les concilier, on ne peut pas toujours tous les appliquer. Donc, il ne faut pas être dogmatique... mais il faut aussi savoir réfléchir, choisir la meilleure solution, et surtout, il ne faut pas programmer par coïncidence ! Soyons capables, nous, développeurs, d'argumenter et expliquer nos solutions !