Programmation Orientée Objet- Design Pattern

Auteur : Rémi Forax, Philippe Finkel

Date	Version	Commentaires
2015-09-27	1	
2015-10-04	1.1	Corrections de quelques typos. Ajout lien JUnit
2015-12-29	1.2	
2016-06-01	1.3	
2016-11-27	1.4	ajout de quelques précisions sur les factories

2017-11-13	1.5	quelques détails
2018-11-10	1.6	notes sur PlantUML, passage à JUnit 5.

Table des matières

1 Introduction

1.1 Objectifs du cours

A l'issue de ce cours:

- vous serez capable de réaliser des tests unitaires JUnit pour toutes les classes que vous écrivez.
- vous serez capable de mettre en œuvre une dizaine de design patterns dans vos projets d'école, en respectant correctement la terminologie.
- vous serez capable d'écrire des codes simples (inférieur à 5 j.h) respectant les principes essentiels de responsabilité unique des classes, de localité.
- vous serez capable de décrire les dépendances d'un code simple et vous serez en mesure de jugez la pertinence ce ces dépendances.
- vous serez capable de concevoir et développer des logiciels relativement complexes (charge inférieure à 30 j.h) en mettant en œuvre les principes S.O.L.I.D. de la programmation orientée objet et les design patterns étudiés.
- vous serez capable, pour de tels logiciels, de modéliser votre conception à l'aide de schémas UML.
- vous serez capable de reprendre un code relativement complexe (charge inférieure à 15 j.h), de le comprendre, de le tester avec des tests JUnit et de le restructurer pour améliorer la distribution des responsabilités entre classes.

1.2 "Poly"

L'objectif du poly est de servir de référence à la de la session. On peut le considérer comme un "poly participatif" : on le complètera au fur et à mesure avec des explications, des schémas et des exemples de code, en fonction de vos besoins, de vos demandes et des discussions que nous aurons.

Pendant les 6 semaines, le poly vous servira surtout de "table des matières", vous aurez à faire un travail personnel de réflexion et de recherche à partir des sujets brièvement traités.

1.3 Corrections par les pairs

Lire et comprendre du code existant fait partie du quotidien des développeurs.

Il faut savoir le comprendre, l'apprécier mais aussi le critiquer et l'améliorer.

Les évaluations qui vous sont demandées doivent vous aider à comprendre les erreurs faites par vos camarades, ou les solutions simples et astucieuses qu'ils ont trouvées.

Les commentaires constructifs que vous leur ferez doivent les aider à comprendre leur erreur, ou même tout simplement pourquoi leur code a été compliqué à lire et à comprendre.

Les modalités d'évaluations par les pairs sont décrites sur la page du cours.

En résumé, les objectifs de la correction par les pairs :

- vous permettre de voir et d'essayer de comprendre les productions (schéma, explications, code) de vos camarades
- vous obliger à "évaluer" ce travail (pas pour une note !)
- vous permettre sur un temps très court de faire ce travail de réflexion
- permettre, par l'usage du forum Moodle, de mutuellement enrichir vos évaluations
  - Attention ! cela implique / oblige à ce que toutes les évaluations soient respectueuses !

1.4 Évaluation

- TP noté de 4 heures
  - le "format" sera le même que pour les TP précédents, qui vous entraîneront.
  - le rendu attendu : UML et design + code
- pénalités sur non rendu ou rendu partiel récurrent, ou travail d'auto-correction manquant de sérieux

2 Contexte

2.1 Projets

Les principes POO et les Design Patterns s'appliquent d'abord aux contextes suivants :

- projet à plusieurs / taille importante (plusieurs milliers de lignes de code)
- dépendances sur librairies qu'on ne contrôle pas
- temps / durée. il n'y a pas que la conception ou la première version qui compte, mais aussi la maintenance

Un projet est *vivant* :

```
- les besoins/exigences changent
- l'équipe de développement change
- les bonnes pratiques de la communauté changent
- le point de vue des développeurs sur le design change
```

Les conséquences sont simples et logiques : - le code doit être lisible !

> on passe plus de temps à lire du code qu'à en écrire !

- la terminologie est littéralement fondamentale. A tous les étages ! packages, classes, méthodes, champs, variables locales, ...
- il est plus facile de réorganiser des "petits" trucs que des gros ! petites méthodes, petites classes, ...
- pas de duplication
- pas de choses compliquées, pas de choses inutilement compliquées
- pas de *piège*. les choses doivent être explicites, logiques et respecter les règles d'usage et les bonnes pratiques
- le moins possible d'effets de bord

> on veut être réactif au changement et pouvoir le faire à moindre coût (temps minimum, pas de régression)

Une fois qu'on se les approprie, les bonnes pratiques de design et de code s'appliquent ensuite même sur des projets plus simples / plus petits. Et cela les rend plus facile, plus rapide, plus agréable.

> Avant d'atteindre ces objectifs, il faut apprendre les bases. C'est ce qu'on va chercher à faire dans ce cours. Explorer les éléments un par un sur des cas simples, tout en gardant à l'esprit que l'objectif sera de pouvoir les appliquer ensemble sur des cas complexes.

2.2 Pourquoi ?

2.2.1 La plupart des projets informatiques ...

Pourquoi cette préoccupation de qualité, d'adaptation aux changements, d'industrialisation, de maintenance "heureuse" ?

Quelques rappels :

> La plupart des projets informatiques se passent mal !

- dépassement important des délais
- dépassement important des coûts
- non conformité (au cahier des charges) :
  - bugs,
  - limitations,
  - "flou" car comportement attendu "discutable" (imprécision, quiproquo, ...)

et :

- difficulté et coût important de de maintenance

2.3 Vie d'un logiciel

- POURQUOI: Besoins / exigences
- QUOI: Cahier des charges fonctionnel
- COMMENT : Architecture (*ce qu'on ne veut pas changer ! les fondations*)
- COMMENT : Conception, Spécifications techniques
  - Dev + tests unitaires, + tests d'intégration
  - Maintenance corrective / évolutive
  - Réutilisation / évolutions majeures / refonte

En tant qu'ingénieur informaticien, vous interviendrez sur le COMMENT.

2.4 Conceptions

Deux approches :

> Conception descendante - top-down design

Le problème global est décomposé en sous-problèmes, eux-même décomposés en opérations plus élémentaires jusqu'à obtenir la granularité souhaitée. Ce qu'on vous a appris depuis des années ! Symbole de l'esprit de synthèse et de l'analyse.

Conception ascendante - bottom-up design

On commence par définir les fonctions les plus élémentaires, pour ensuite les utiliser et définir des fonctions de plus en plus spécifiques et complexes.

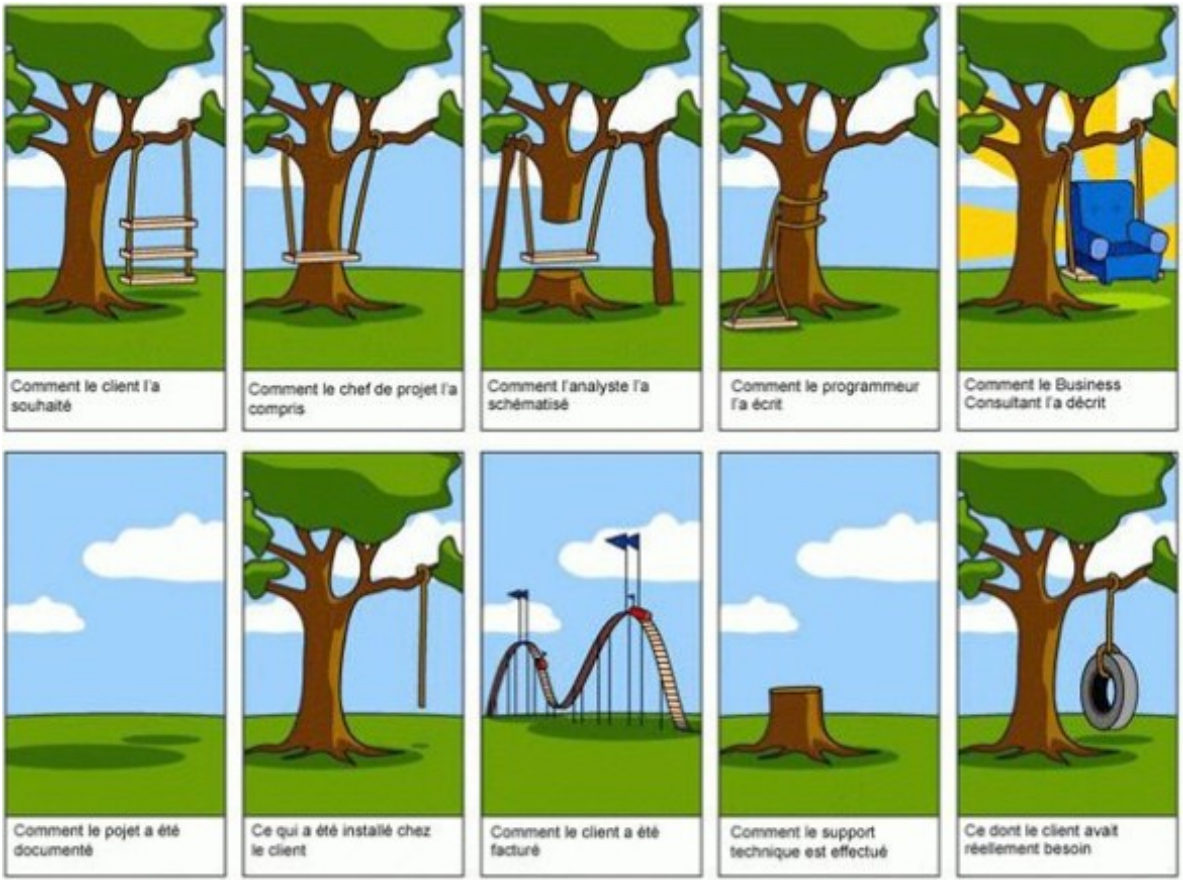
En pratique les deux approchent se combinent et se complètent :

- bottom-up : toutes les études techniques. Il faut connaître et savoir utiliser sa *boîte à outils* avant de construire.
- top-down : vue d'ensemble du système. coordination de haut niveau des sous-ensembles. rendre la gestion de projet faisable.

3 Terminologies

3.1 Se comprendre ?

Vous connaissez le classique ?



compréhention du besoin ?

Il est si célèbre car malheureusement souvent proche de la réalité quotidienne de très nombreux projets informatiques.

3.2 Terminologie de ce poly

La terminologie est essentielle.

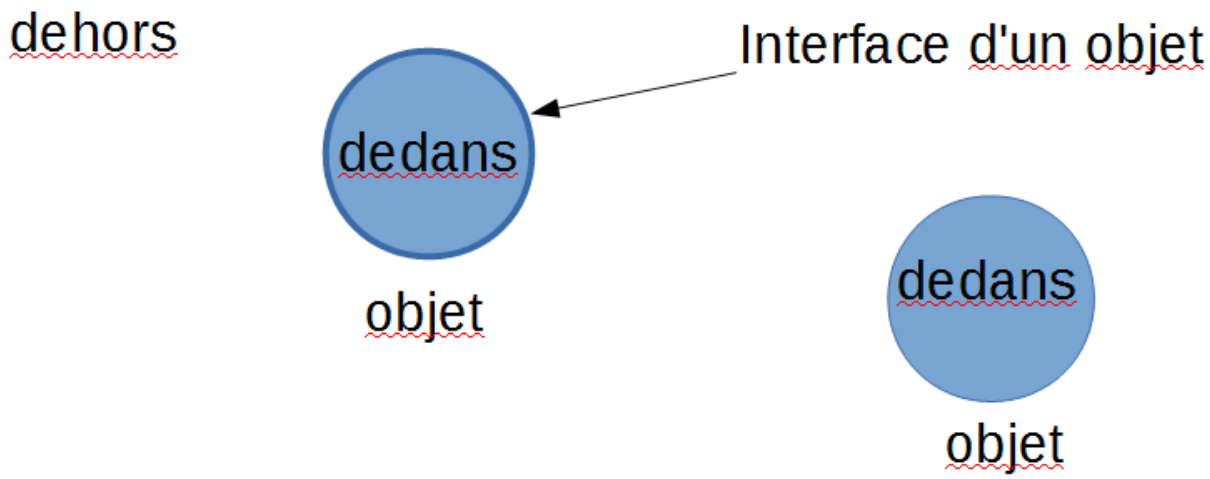
Dans le monde des Design Patterns, il y a quelques variantes.

Nous utiliserons une terminologie proche du GoF [1] mais légèrement différente.

1 Design Patterns, by the Gang Of Four. L'ouvrage commence à dater mais à lire une fois dans sa vie.

4 Qu'est ce qu'un objet ?

- Un objet définit un en-dedans et un en-dehors
- L'idée est que le dehors ne doit pas connaître la façon dont le dedans fonctionne



dedans et dehors

4.1 Responsabilité & intégrité

Une classe doit avoir 1 et 1 seule responsabilité

Une classe est elle seule responsable de l'intégrité de ses données (encapsulation)

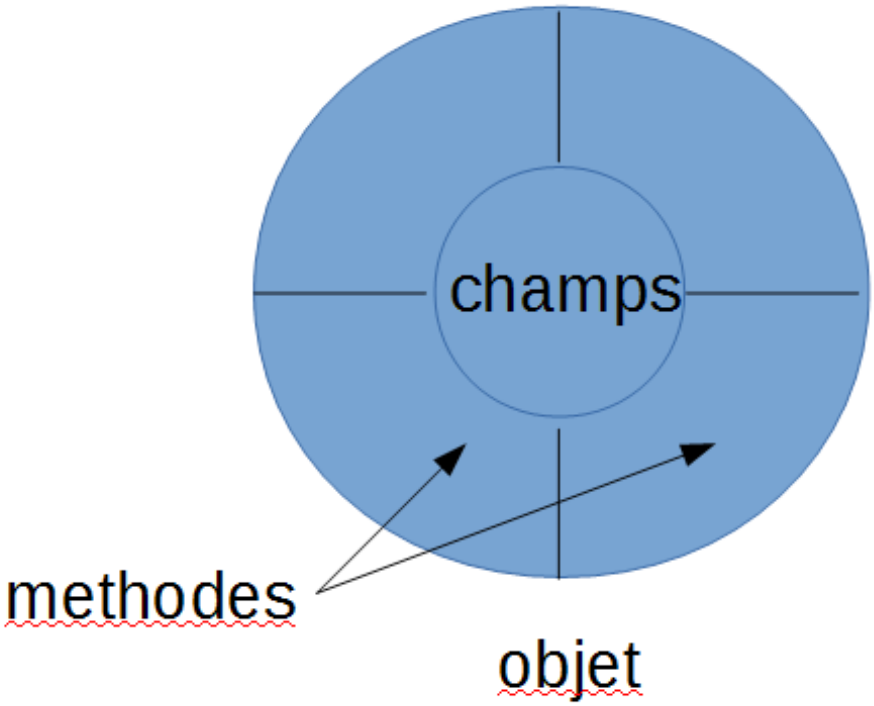
Un système complexe est donc modélisé par des interactions entre des objets simples

4.2 Instance de classe et objet

Une classe définit un moule à partir duquel on fabrique des objets On appelle ces objets des instances de la classe

4.3 Méthodes & POO

Une méthode représente un point d'accès vis à vis de l'extérieur d'un objet Le code d'une méthode a accès à l'intérieur d'un objet



méthodes

4.4 Encapsulation

la seule façon de modifier l'état d'un objet est d'utiliser les méthodes de celui-ci

- Restreint l'accès/modification d'un champ à un nombre fini de code > Les méthodes de la classe
- Permet de contrôler qui fait les effets de bord > Le “mieux” est de ne pas faire d'effet de bord !

4.5 Intégrité & Contrat

La classe doit garantir les invariants

- Exemple d'invariant (une propriété)
  - par ex, le champ x est toujours positif
- Le constructeur sert de point d'entrée
- Pas de setter inutile !

5 UML

5.1 Introduction

UML = Unified Modeling Language

C'est un *langage* graphique uniformisé pour la spécification de modèles objets. Les éléments de conception, voire d'implémentation peuvent être décrits avec des graphiques standardisés.

Il existe dfe nombreux types de diagrammes UML. On peut citer :

- Diagrammes comportementaux
  - Diagramme des cas d'utilisation
  - Diagramme états-transitions
  - Diagramme d'activité
- Diagrammes structurels ou statiques
  - Diagramme de classes
  - Diagramme des paquetages
  - Diagramme d'objets
  - ...
- Diagrammes d'interaction ou dynamiques
  - Diagramme de séquence
  - ...

Cette année, nous nous intéresserons **exclusivement aux ddiagrammes de classes**.

L'année prochaine, vous aborderez les diagrammes de séquence et diagramme de cas d'utilisation.

5.2 A quoi ça sert ?

UML aide lors des phases de brainstorming pour représenter les visions possibles.

UML aide à partager, faire comprendre son design.

UML aide à transmettre.

Et, avant tout, UML doit vous aider à réfléchir !

- si votre schéma est clair dans votre esprit et sur la papier, vous pouvez partir sur le code
- si votre schéma est confus, démarrer à coder va accentuer les problèmes car vous allez vous noyer dans les détails de code

5.3 Diagramme de classe

5.3.1 Classe

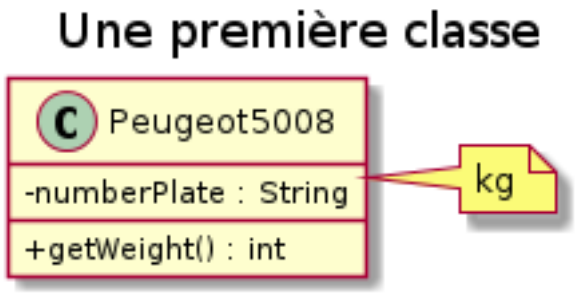
Chaque classe est représentée par un rectangle avec :

- son nom
- ses champs
- ses méthodes

Les types sont optionnels, avec notation à la pascal.

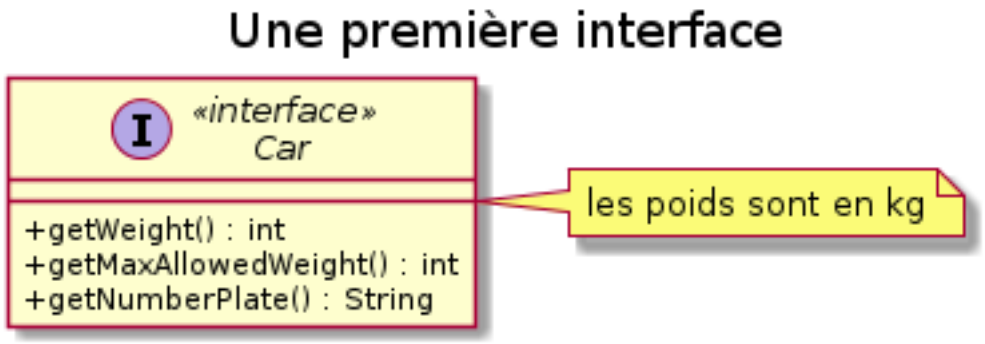
L'indication du niveau de protection se fait avec un cacactère :

```
+ signifie "public"
- signifie "private"
# signifie "protected"
~ signifie "package"
```



5.3.2 Interface

Une interface est un rectangle avec seulement deux parties (pas de champs !) et avec l'indication <>.



5.3.3 Relations

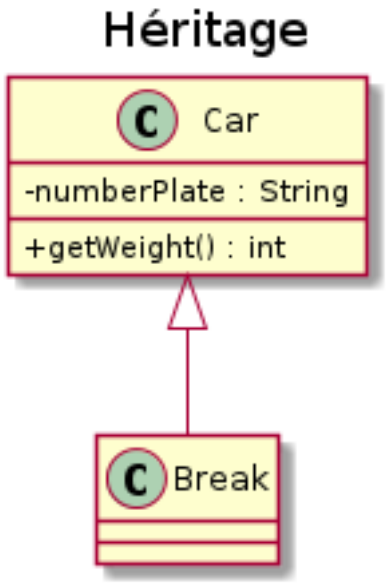
Dans un diagramme de classes, on indique les relations entre classes et interfaces :

- généralisation (héritage)
- réalisation (implémentation)
- Association
  - unidirectionnelle ou bidirectionnelle
  - cas particulier de l'agrégation
  - cas particulier de la composition
- dépendance

Généralisation

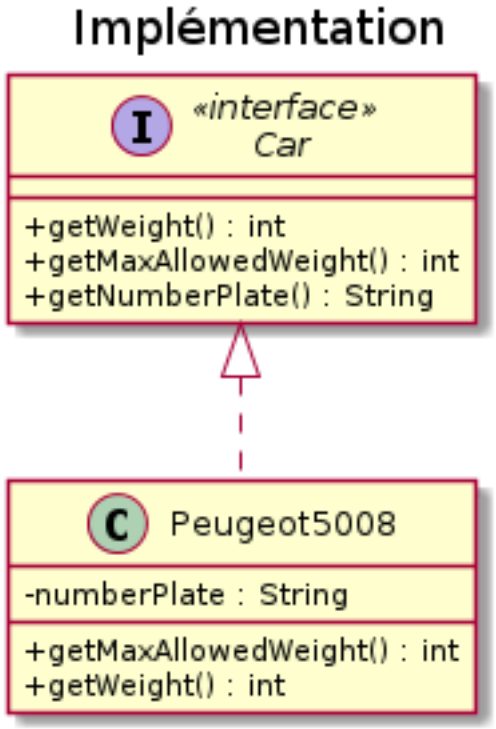
> On ne recopie pas les membres hérités

- on indique les méthodes redéfinies



Réalisation

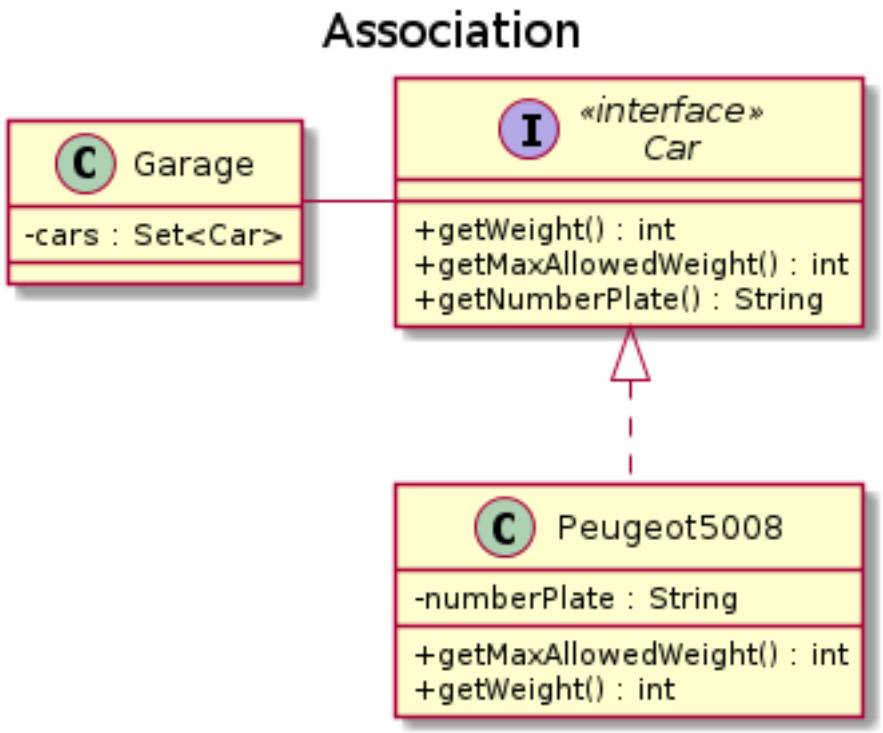
- on indique les méthodes redéfinies



Association

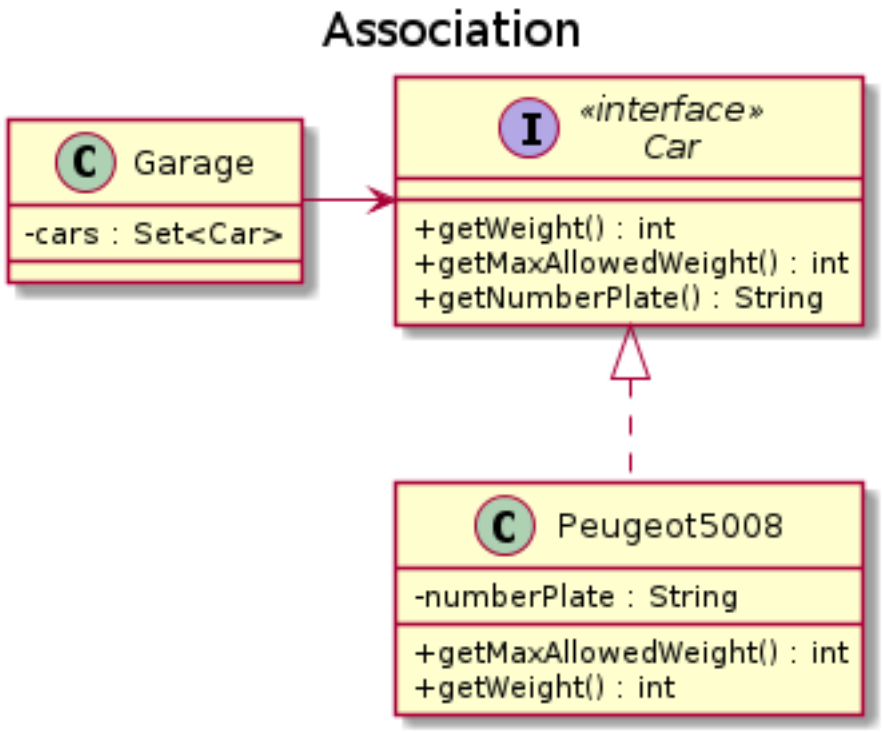
> Typiquement, un lien qui peut s'exprimer "utilise un", "possède un", "connaît un", "fait partie de", ...

Premier exemple **incomplet**



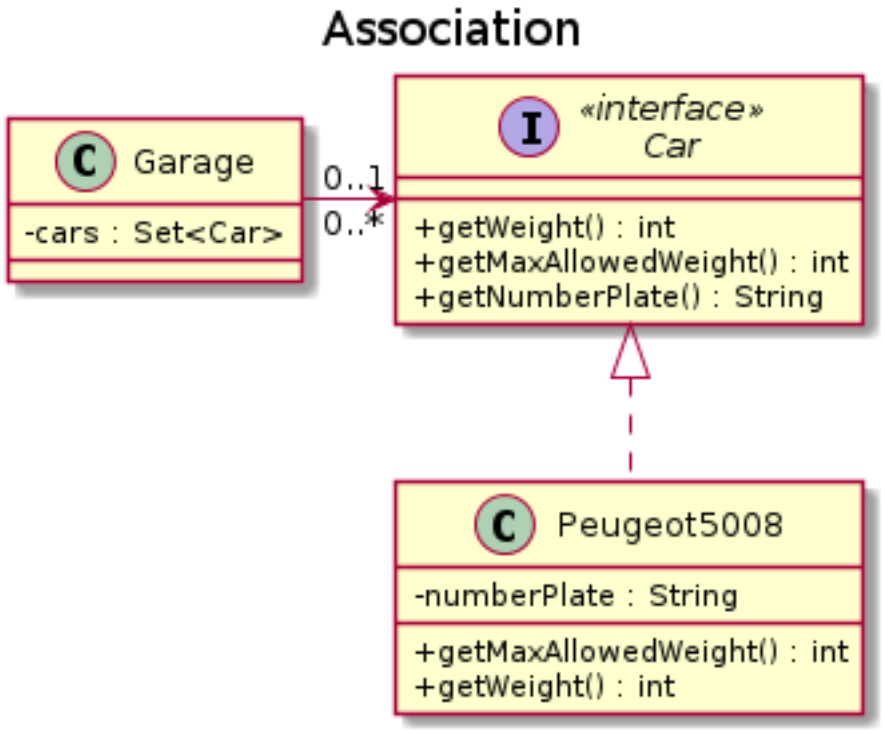
Dans cet exemple, on ne précise pas la *navigabilité* : le lien entre Garage et Car ne montre pas si un garage "connaît" les voitures qu'il contient, ni si une voiture "connaît" le garage où elle est en réparation.

On corrige ce problème en indiquant le sens des liens, c'est ce qu'on appelle la navigabilité.



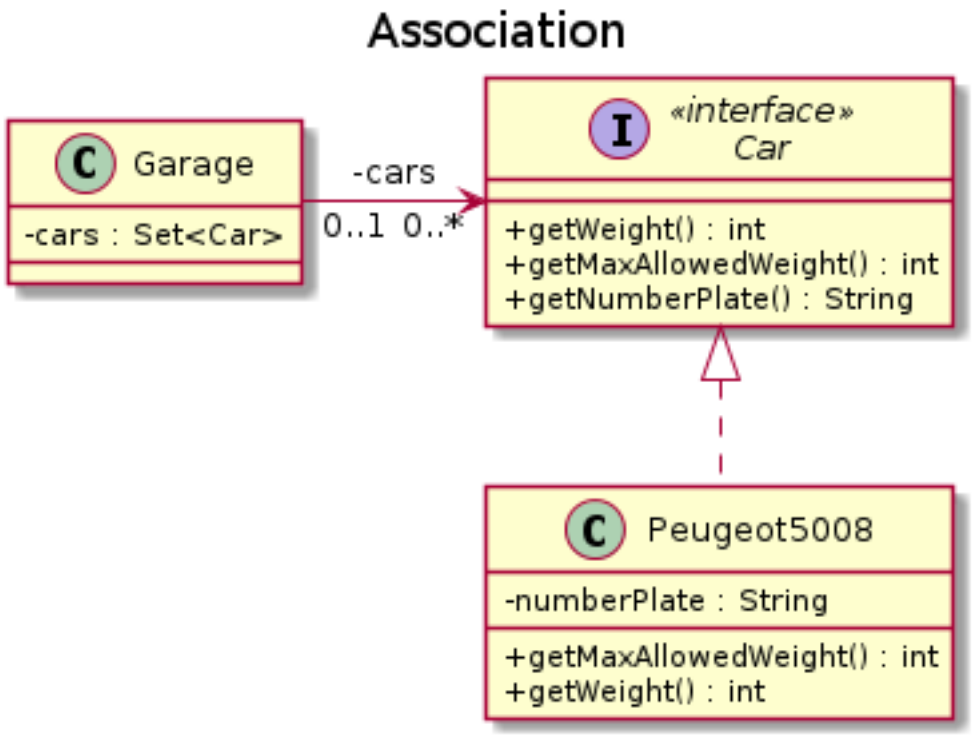
Ce deuxième exemple est un peu mieux mais on ne connaît pas la cardinalité : combien de voitures un garage peut contenir ? au minimum ? au maximum ? Une voiture peut-elle être liée à plusieurs garages ?

On corrige ce problème en indiquant la cardinalité des liens, c'est à dire le nombre minimum et maximum d'objets auxquels on est lié.



Ce troisième exemple est un peu mieux mais il ne donne pas d'information sur la nature du lien entre Garage et Car. En ajoutant une information sur le lien, on peut soit préciser de manière fonctionnelle comment ces objets sont liés, soit préciser par quel champ ils sont liés.

Ici, on précise qu'ils sont liés par le champ *cars*, qui est privé.



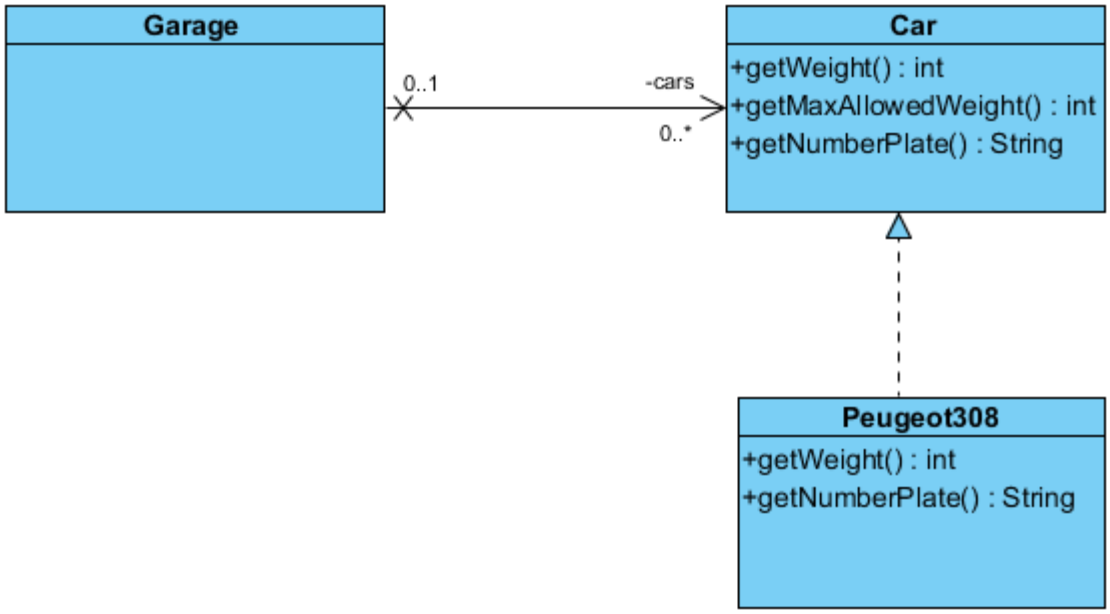
Remarques :

- l'indication que *cars* porte le lien n'est pas à l'extrémité à cause de limitation de l'outil utilisé
- En général pas de membres privés mais jamais de dogmes. Ici on a eu envie de montrer clairement que le Garage a un ensemble de voitures.

En résumé :

- la navigabilité : la flèche de Garage vers Car montre qu'un garage "connaît" les voitures qu'il contient mais l'inverse est faux
- la cardinalité : un garage peut être vide ou contenir un nombre quelconque de véhicules. Un véhicule ne peut être que dans un garage à la fois !
- le libellé à l'extrémité du lien peut indiquer le nom du champ privé *cars* qui porte ce lien

Un autre schéma représentant les mêmes classes/interfaces :



assoc1

> suivant le logiciel utilisé, le style change ... mais pas la signification des éléments du langage graphique !

Agrégation et Composition

Ce sont des cas particuliers d'associations.

L'agrégation est un cas particulier d'association :

- Relation non symétrique, lien de subordination
- Couplage fort
- Relation « non vitale » les cycles de vie peuvent être indépendant

La composition est un cas particulier d'agrégation :

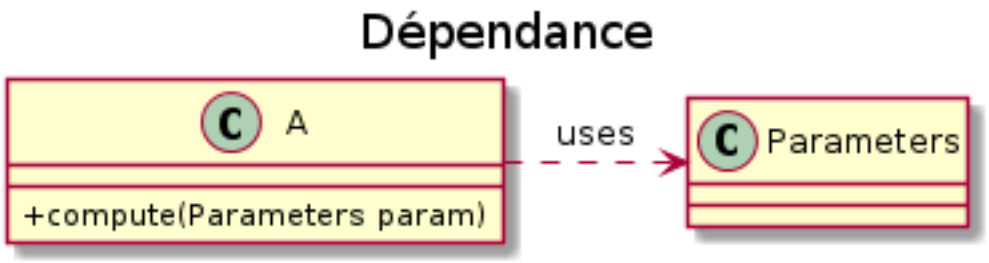
- Agrégation forte
- Cycles de vie liés
- À un instant t, une instance de composant ne peut être liée qu'à un seul composé.

Dépendance

Signifie un « dépendance » de n'importe quel ordre. Par exemple :

- une méthode de la classe A crée un objet de type B
- une méthode de la classe A prend en argument un objet de type B

On peut indiquer la nature de la dépendance par un libellé sur le lien.



5.4 Comment faire ?

5.4.1 Règle n°1

Pas de détail inutile !

- pas de champ private
- en général pas de champ du tout
- pas de méthodes privées, très peu de protected

5.4.2 Règle n°2

Ne pas oublier ce qui compte !

- les liens entre classes. héritage, implémentation, dépendance, association
- la navigabilité des liens
- la cardinalité des liens

5.4.3 Règle n°3

Propre mais pas trop !

- Il faut que vos schémas soient lisibles, compréhensibles
- Pour ce cours, nous vous demandons d'apprendre et d'utiliser PlantUML
- n'hésitez pas à commencer à réfléchir sur la feuille blanche ou au tableau.

5.4.4 Les schémas de ce document

Dans ce poly, on a utilisé :

- PlantUML
- Visual Paradigm
- le crayon

5.5 Le minimum nécessaire / exigé

> avoir TOUJOURS du papier et un crayon !

Les éléments indispensables :

- respect de la syntaxe UML : il y a 4 sortes de relations, il faut les dessiner correctement

- navigabilité : se poser systématiquement la question : A connaît-il B ? et écrire la réponse avec le sens des flèches
- cardinalité : se poser systématiquement la question. Exemple d'impact, une cardinalité 0..1 ou 1 va fortement influer sur le contrat d'une classe. si un A est toujours lié à UN B, cela va déjà se traduire dans le constructeur.

5.6 UML et lambdas

Avoir une lambda ou une classe nommée n'a aucune influence sur un schéma UML ... mais l'utilisation des lambdas incite souvent à plus de composition entre classes, ce qui modifie le schéma UML.

6 Bonnes pratiques d'implémentation

6.1 Interfaces

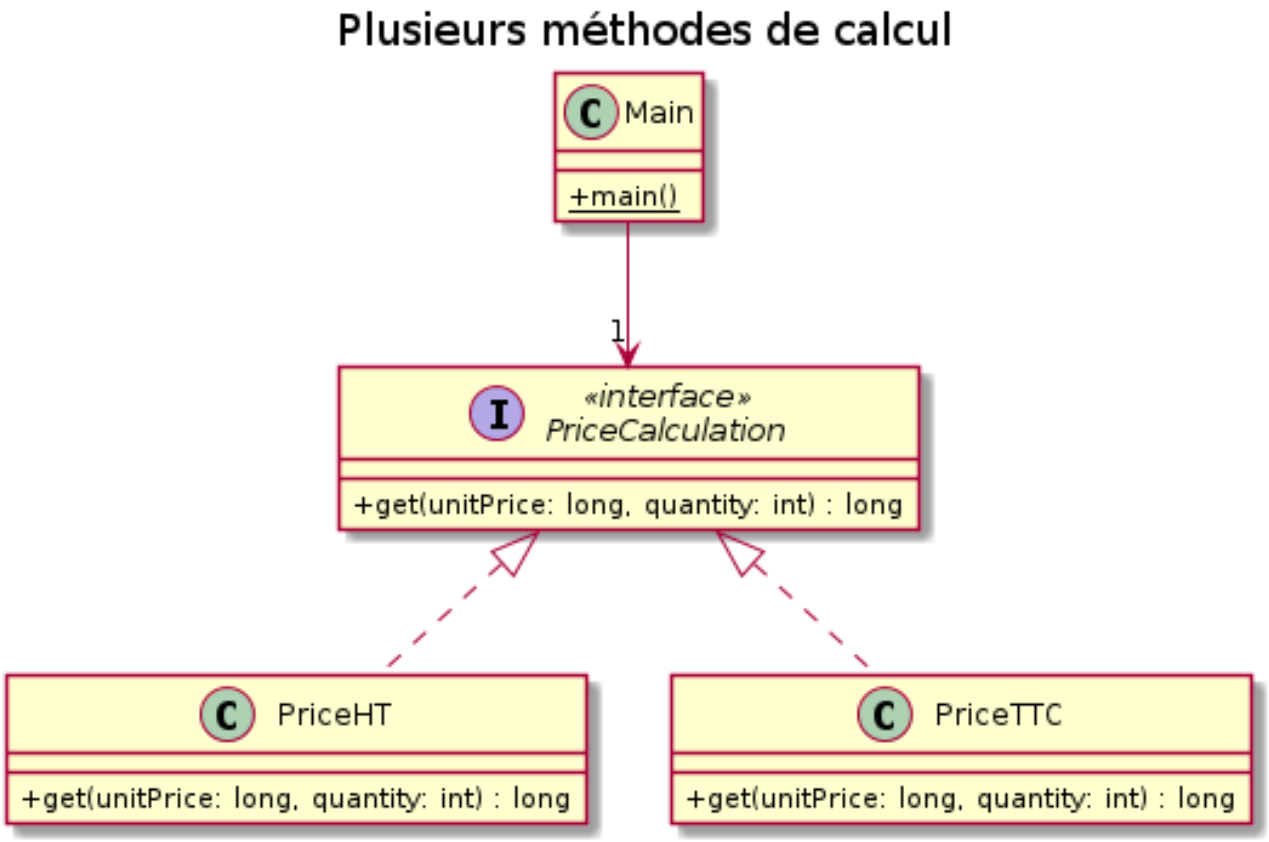
Il existe 3 manières principales d'implémenter une interface :

- classe concrète
- classe interne / anonyme
- lambda

> Ces différentes implémentations correspondent **au même diagramme UML**.

Les sections suivantes illustrent ces possibilités sur un exemple très simple : un programme qui prend trois arguments, un nombre de pommes et un prix HT par pomme, et le type de prix total souhaité (ht ou ttc).

Pour effectuer les calculs, nous avons conçu le diagramme de classe UML suivant :




```
public interface PriceCalculation {
    public long get(long unitPrice, int quantity);
}
```

6.1.1 Classe concrète

Chacune dans son .java :

```
public class PriceHT implements PriceCalculation {

    @Override
    public long get(long unitPrice, int quantity) {
        return unitPrice * quantity;
    }

}

public class PriceTTC implements PriceCalculation {

    @Override
    public long get(long unitPrice, int quantity) {
        return (long)(unitPrice * quantity * 100 * 1.2);
    }

}
```

6.1.2 Classe interne / anonyme

Soit une classe interne nommée :

```
public static class PriceHT implements PriceCalculation {

    @Override
    public long get(long unitPrice, int quantity) {
        return unitPrice * quantity;
    }

}

...

PriceCalculation priceCalculation = new PriceHT();
```

Soit une classe anonyme :

```
PriceCalculation priceCalculation = new PriceCalculation() {

    @Override
    public long get(long unitPrice, int quantity) {
        return unitPrice * quantity;
    }

}
```

6.1.3 Lambda

```
    PriceCalculation priceCalculation = (unitPrice, quantity) -> unitPrice *
quantity * 100;
```

6.2 Partage / factorisation de code

La duplication de code engendre la duplication de bugs !

Attention au nommage :

- pour pouvoir partager, vous devez **être capable de nommer** ce qui est partagé
- et le nommer **bien**
- une difficulté de nommage indique la plupart du temps un partage inutile ou mal conçu

Attention aux excès :

- essayer de partager du code très simple/court peut rajouter de la complexité.

6.3 Méthode statique

> Tip eclipse : *Refactor/Extract Method*

6.4 Délégation à un objet

> Tip eclipse : *Source/Generate delegate methods*

6.5 Implémentation par défaut dans les interfaces

6.6 Classe abstraite

> Important : Une classe abstraite ne doit pas être publique sinon on expose des détails d'implémentation.
>
> Important : on ne fait pas une classe absraite pour partager *juste* des champs ! on partage avant tout des comportements (du code et les champs associés).
>
> Rappel : la classe abstraite est l'exception, pas la règle !
>
> Tip eclipse : *Refactor/Extract superclass*

6.7 Héritage

> C'est la technique du XX ème siècle, plus la votre !

> Le seul cas où vous en faites : quand du code existant vous oblige à le faire. Par exemple un code de librairie externe non contrôlé.

6.8 Du bon usage des String

> Les chaînes de caractères sont utilisées en début et fin de programme, pour discuter avec l'utilisateur (ou système externe). Au milieu du programme, on manipule des objets typés et avec une sémantique forte.

6.9 Du bon usage des identifiants

6.10 Du bon usage des *print*

> Rappel : pour réutiliser du code, il faut séparer le traitement sur les données et l'affichage.

6.11 Utiliser le polymorphisme pour remplacer les switchs/if en cascade

> A noter que cette règle vous aide aussi de respecter la précédente !

7 SOLID

7.1 Introduction

SOLID est un acronyme représentant 5 principes de bases pour la programmation orientée objet.

Notre cible pour ce cours est que vous les compreniez, que vous vous les appropriez, que vous sachiez les mettre en oeuvre ou les repérer.

	Définition	Remarques
S	Single responsibility principle	Une classe = une et une seule responsabilité
O	Open/closed principle	Les modules livrés restent fonctionnellement extensibles
L	Liskov Substitution Principle	Travailler avec des interfaces
I	Interface segregation principle	Petites interfaces, adaptées au besoin
D	Dependency Inversion Principle	*Le commandant n'a pas besoin de tout savoir faire*

7.2 Rappels

7.2.1 Interfaces

Interface (compilation)

- Pour éviter la duplication de code, si un code manipule indifféremment des objets de plusieurs classes différentes alors les objets sont manipulés à travers une interface Cf principe de sous-typage de Liskov

- L'interface contient l'intersection des méthodes communes utilisées par ce code

Polymorphisme (exécution)

- Un appel d'une méthode à travers une interface exécute la méthode de la classe (dynamique) de l'objet

- Il y a redéfinition à la compilation (override) lorsque la substitution par polymorphisme à l'exécution est possible

Cercle vertueux de l'ignorance

1. On manipule les objets par des méthodes, on ignore donc l'implantation de celles-ci
2. On manipule les objets à travers leurs interfaces, on ignore donc la classe de l'objets
3. Cela permet de limiter l'écriture de code spaghetti et renforcer la localité

7.2.2 Modules

Définition

- Unité de code à forte cohésion
  - En Java, c'est souvent un package, packagé sous forme d'un jar
- Définit aussi un intérieur et extérieur
  - Visibilité de package (au lieu de privée)
  - Étend les concepts de responsabilité et de contrat
  - Étage au dessus de la classe qui permet de concevoir des systèmes complexes

Programme vs librairie

Lorsque l'on écrit un programme, on utilise des libraries de façon externe

Lorsque l'on écrit une librairie, on est en interne On utilise les tests de cas d'utilisation pour vérifier la partie externe

7.3 Single responsability principle

Une classe = une et une seule responsabilité

Mais aussi : une méthode = une seule fonctionnalité

Et : un package = un seul ensemble cohérent de fonctionnalité

> LE principe numéro UN !

Bon nombre de design patterns servent à respecter ce principe.

Objectifs :

- Limiter rigidité et fragilité
- Aider à la localité

Une des principales difficultés du développement d'un logiciel complexe est la gestion et le contrôle des dépendances. Le respect du *Single responsability principle* est essentiel pour concevoir des composants/packages/classes sans dépendances excessives et sans dépendances cachées.

7.4 Open / Close principle

Une classe ou un module est soit

- Open
  - En cours développement, où tout est possible
- Close
  - Débuggé, Testé, Versionné

En théorie, on peut passer un module que de open vers close mais pas vice-versa

> Attention à ce O : extension fonctionnelle / réutilisation ne veut pas forcément dire héritage !

Au quotidien, la plupart des modules sont fermés :

- parce qu'on ne PEUT pas les modifier (au hasard, les classes du JDK; mais aussi le travail de l'équipe d'en face)
- parce qu'on ne VEUT pas les modifier : pour ne pas les complexifier inutilement, ne pas leur donner une nouvelle responsabilité, ...

Et pourtant, on veut pouvoir les étendre. ce sera possible s'ils sont ouverts aux extensions.

- les principes de base pour arriver à faire des modules Open-Closed :
  - "program to an interface" ou "depend on abstractions, not concrete classes",
  - "Single responsability principle"
- ensuite, les Design Patterns donneront des idées/pistes de solutions en fonction du problème et du contexte

7.5 Liskov Substitution Principle

> « Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »

Implications :

- **Le contrat** défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classe dérivées
- L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être substituée à la classe qu'il utilise
  - C'est le principe de base du polymorphisme : si on substitue une classe par une autre dérivée de la même hiérarchie, le comportement est (bien sûr) différent mais **conforme au contrat**.

7.6 Interface segregation principle

> Un client doit avoir des interfaces avec uniquement ce dont il a besoin

- Incite à ne pas faire "extract interface" sans réfléchir
- Incite à avoir des interfaces petites

Attention : cela pourrait amener à l'extrème à une multiplication excessive du nombre d'interfaces. Expérience, pragmatisme et bon sens doivent vous aider.

7.7 Dependency Inversion Principle

Le principe de base qui sous-tend l'inversion de dépendance : "Depend upon Abstractions. Do not depend upon concretions."

Sur la page wikipedia du principe d'inversion de dépendance, deux aspects à bien comprendre :

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Exemple : eclipse ne dépend pas des différents plugins, eclipse dépend de l'abstraction de plugin qui a été définie par eclipse
- Abstractions should not depend on details. Details should depend on abstractions.
  - Exemple : l'abstraction de plugin est évidemment indépendante des différents plugins qui peuvent exister. L'implémentation des différents plugins ne dépend pas non plus des détails d'eclipse mais de l'abstraction de container que représente eclipse.

8 Design Patterns

8.1 Introduction

La conception orienté objet est une activité difficile il faut trouver les bons objets les factoriser en classes de la bonne granularité, définir des interfaces et des hiérarchies de classes et surtout établir des relations entre tout cela.

Écrire des éléments modifiables est difficile, des éléments réutilisables est encore plus difficile.

Une conception flexible et réutilisable est rarement obtenue du premier jet !

Avec l'expérience on tend a produire de la bonne conception, comment capitaliser cette expérience, c'est l'objectif des Design Patterns.

> Important : nous n'en étudierons que quelques uns ! A vous de découvrir les autres par vos lectures ou en les expérimentant.

8.2 Définition

- Patrons de conception communément utilisés pour résoudre des problèmes classiques de conception

- Solution *éprouvée* par la pratique à un *problème* dans un *contexte* donné

8.3 Quelques règles

- Un DP Répond à un problème
  - Pas de problème, pas de design pattern
- Possède un nom utilisé par tous
- La solution est décrite par un schéma UML des classes impliquées ainsi que les responsabilités de chacune.
  - mais attention : le schéma n'explique pas tout
- Les conséquences : un design pattern a des impacts qui peuvent impliquer des compromis

8.4 Quand les utiliser ?

Les design patterns aident à la conception

- Au début, faites de l'over-design !
- Après, utilisez votre esprit critique !

Un design pattern est souvent adapté/tordu pour répondre à votre problème/contexte

Les design patterns aident à la formalisation

```
- Définit un vocabulaire pour les échanges avec les membres de l'équipe
```

A quelle étape du projet :

- Phase de conception
- Phase de refactoring
- Transmission

8.4.1 Bonnes pratiques

- Ne pas réinventer la roue
- Terminologie commune
- À votre service, adaptables
- Souvent combinés
- faire l'UML systématiquement

8.5 Design Patterns “creational”

8.5.1 Introduction

Les catégories de Design Pattern ne sont pas rigides. Elles donnent un axe d'approche d'un Design Pattern.

Les Design Pattern classés dans cette catégorie :

- Singleton
- Factory
- Static factory
- Method factory
- Abstract factory - Kit
- Builder
- (Prototype)

8.5.2 Singleton

- Une seule instance
- Accès global
- … et POO !

> Souvent, une mauvaise solution pour cacher une variable globale (c'est mal)
>
> Attention, l'implémentation n'est pas aussi triviale qu'il n'y paraît :

- Lazy ou pas ?
- Thread-safe ?

8.5.3 Factory & Co

Factory = usine.

Quand on utilise une factory pour un type d'objet ou une hiérarchie, les *new XX* ne se trouvent QUE dans la factory.

Terminologie

> Il existe plusieurs sortes de Factory et pas de consensus sur le nommage des différentes sortes !

Plusieurs sources

- GOF
- Java
- Votre entreprise ?

Les apports des factories

Le point commun à toutes les factories :

- S de Solid
- Délégation de la création

Cordon sanitaire autour des new :

- Choix des classes
- Calculs éventuels sur les arguments à donner aux constructeurs
- Les clients ne voient que les interfaces

Method Factory

- Design Pattern du GoF [2].
- Déléguer le choix du type d'objet à créer à ses classes dérivées
- Peu utilisée ! (Rappel : *Préférer la délégation à l'héritage*)

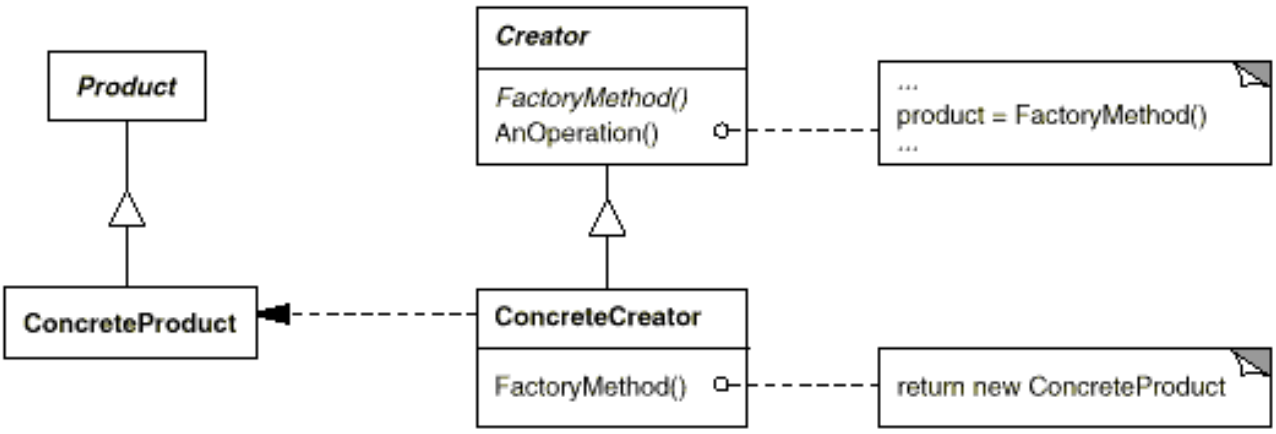


Schéma Method Factory

2 Design Patterns, by the Gang Of Four. L'ouvrage commence à dater mais à lire une fois dans sa vie.

Static Factory Method

Une méthode dont la seule responsabilité est la création d'objets.

- Éviter la duplication du code de création
  - Centralise le code de création
  - Cordon sanitaire autour des “new”
  - 1 méthode 1 responsabilité

> Codée en 2 minutes

Ce n'est pas un Design Pattern du GoF [3] mais il est très utilisé.

Cf "Effective Java" [^2] pour une discussion sur ce design pattern.

> Attention de ne pas le confondre avec "Method Factory" !

Static Factory

Une classe, avec uniquement des méthodes static, dont la seule responsabilité est la création d'objets.

- Éviter la duplication du code de création
  - Centralise le code de création
  - Cordon sanitaire autour des “new”
  - 1 classe 1 responsabilité

> Codée en 2 minutes

Pas un Design Pattern du GoF [4] mais très utilisée.

“Factory non static”

- Comme la static factory mais non static !
- Permet d'avoir une hiérarchie de factory
- Simple évolution de la static factory pour avoir plus de flexibilité

Pas un Design Pattern du GoF [5] mais très utilisée.

Abstract factory - Kit

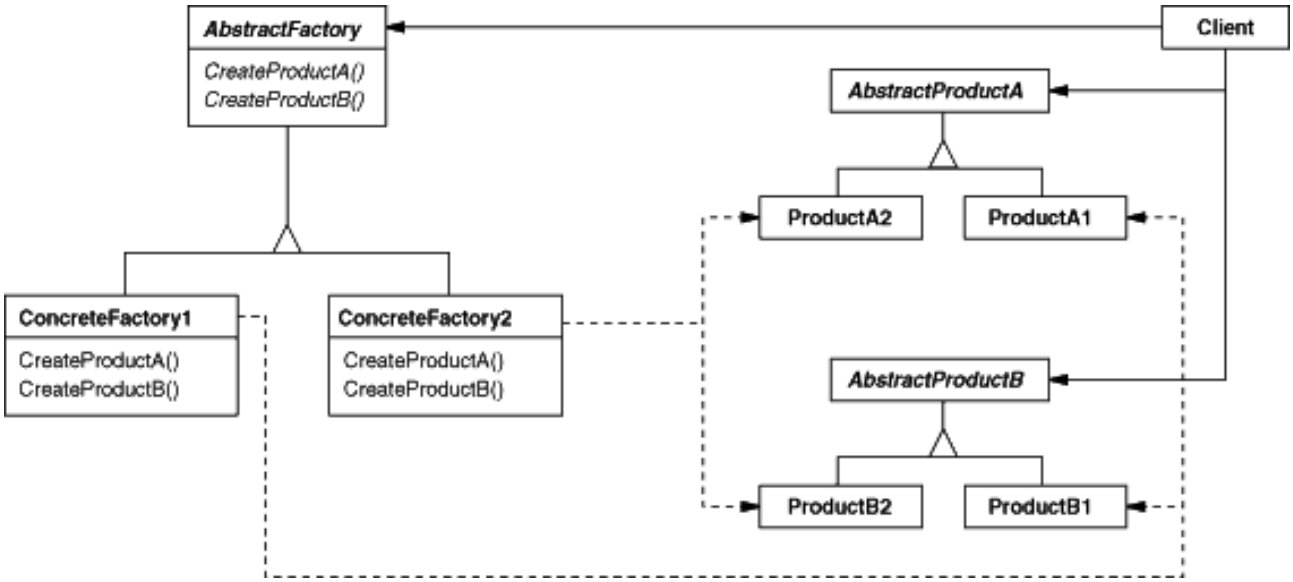
- On a plusieurs familles d'objets
- Avec des règles de compatibilité particulières

3 Design Patterns, by the Gang Of Four. L'ouvrage commence à dater mais à lire une fois dans sa vie.

4 Design Patterns, by the Gang Of Four. L'ouvrage commence à dater mais à lire une fois dans sa vie.

5 Design Patterns, by the Gang Of Four. L'ouvrage commence à dater mais à lire une fois dans sa vie.

- Le Kit garantit de créer des objets compatibles



abstract factory

Exemple : SQL JDBC

Une abstract factory typique, très légèrement adaptée par rapport au schéma de principe.

```
String url = "jdbc:mysql://localhost/test";
String user = "root";
String passwd = "";
Connection conn = null;
try {
    conn = DriverManager.getConnection(url, user, passwd);
} catch (SQLException e) {
    System.err.println(e.getMessage());
}
String query = "SELECT ID, post_content FROM wp_posts";
try {
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(query);
    while (rs.next()) {
        String id = rs.getString("ID");
        String content = rs.getString("post_content");
        System.out.println(id + " -->  " + content);
    }
} catch (SQLException ex) {
    System.err.println(ex.getMessage());
}
```

- Statement est un com.mysql.jdbc.Statement
- ResultSet est un com.mysql.jdbc.ResultSetImpl

> Le DP AbstractFactory est un peu caché : on ne s'adresse pas qu'un Driver pour créer les objets, cela fonctionne en cascade. Mais ça ne change rien au principe : Driver & co forment bien une AbstractFactory !

- Au chargement des jar, le répertoire META-INF/services permet l'enregistrement auto de services.
- Java.sql.Driver contenait le full name de la classe Driver : com.mysql.jdbc.Driver
- Java a fait un new de cette classe et l'a enregistré dans DriverManager (supposons tout en static)
- Sur la demande de connection, c'est seulement le Driver mysql qui a accepté de fabriquer une Connection avec cette URL. Une com.mysql.jdbc.ConnectionImpl a été renvoyée

Partage d'objets ?

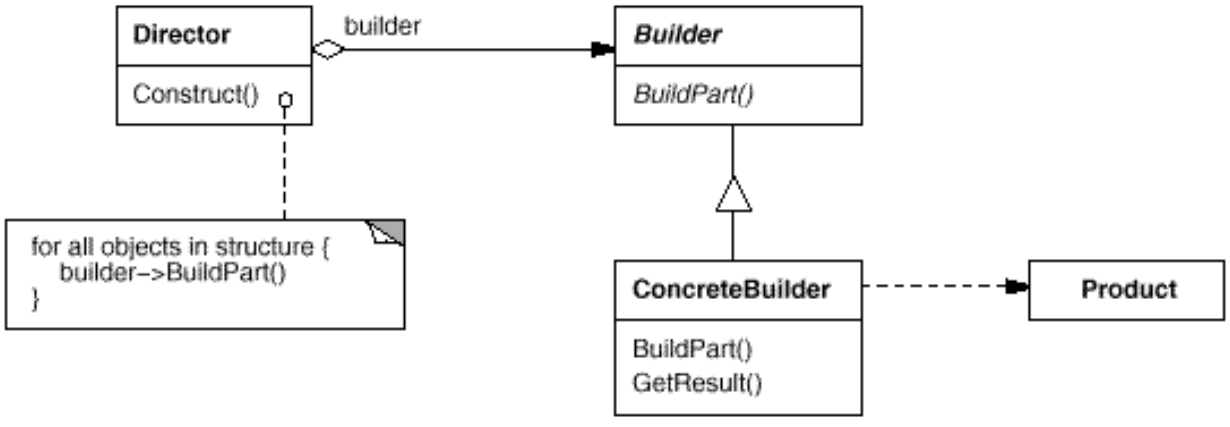
Même si ce n'est pas le but premier, une factory peut gérer le partage des objets.

Dans l'esprit, c'est ce que faisait String s = “hello” ou Integer i = 4; En particulier pour des objets non mutables !

Exemple : - Quand les objets ne sont pas mutables - et que le nombre d'objets différents est petit, - une factory peut gérer un *pool* d'objets partagés - Cette possibilité est un élément important du Design Pattern *Flyweight*.

8.5.4 Builder

- Quand la construction devient trop compliquée
- Évite un grand nombre de constructeurs
- “Construction” par étapes
  - L'objet n'est créé qu'à la dernière étape et renvoyé au client



builder

Exemple :

- StringBuilder

8.6 Design Patterns structurels

Design Patterns liés au problème d'organisation des objets dans un logiciel

- Composition des classes et des objets
- Permet de concevoir des structures plus larges, et extensibles. Un peu comme des préfabriqués avec des interfaces normalisées
- Renforce aussi la localité (donc le S de SOLID !).
- Ex sécurisation et proxy

Quelques exemples :

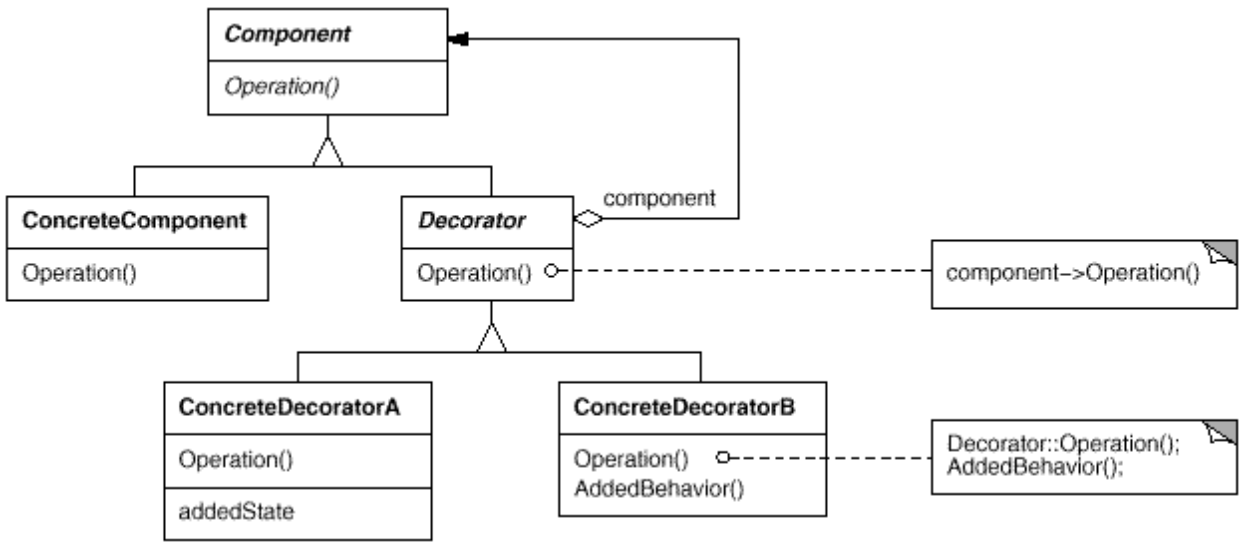
- Decorator interne et externe : ajouter une caractéristique à un objet
- Proxy : l'objet est loin, le calcul est lourd, l'accès est réservé
- Composite : traiter de manière identique un groupe et un élément
- Flyweight : (vous l'apprendrez par vos lectures)

8.6.1 Decorator 's

- Étendre les fonctionnalités d'une classe sans la modifier,
  - Sans la compliquer
  - Sans lui ajouter une deuxième responsabilité !!
- Etendre avec souplesse → pas toutes les mêmes instances de la même manière
- Rester transparent pour le client !!
- Peut être récursif
  - Donc aide à garder des décorateurs avec notre 'S' !

Decorator externe

C'est le Decorator décrit dans le GoF [6].



decorator

6 Design Patterns, by the Gang Of Four. L'ouvrage commence à dater mais à lire une fois dans sa vie.

Decorator interne

- Quand la décoration peut être “prévue”
- On délègue la fonctionnalité
  - En ne connaissant que l'interface
  - Exemple: bordure
- Différences avec decorator externe
  - Pour garder les this
  - Pour garder le type des objets

Decorator interne vs externe

> Attention aux question de terminologie

```
- Decorator GoF = Decorator externe
- Decorator interne = même intention mais implémentation par simple strategy
```

> Un schéma UML ne suffit pas ! il faut aussi connaître / décrire l'intention.

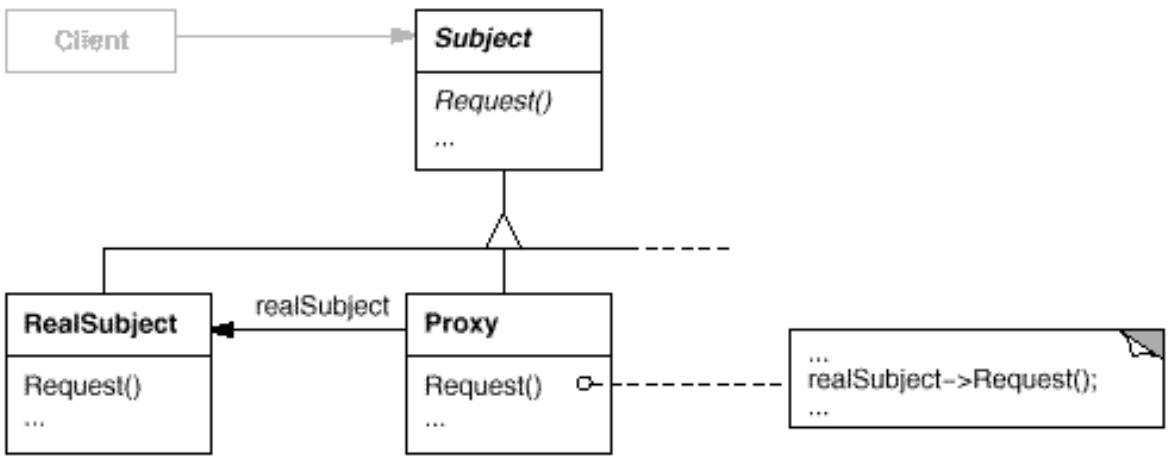
8.6.2 Proxy

- Un intermédiaire
- Pour ne pas compliquer un objet
- Rester tranparent pour le client !!

Cas d'utilisation :

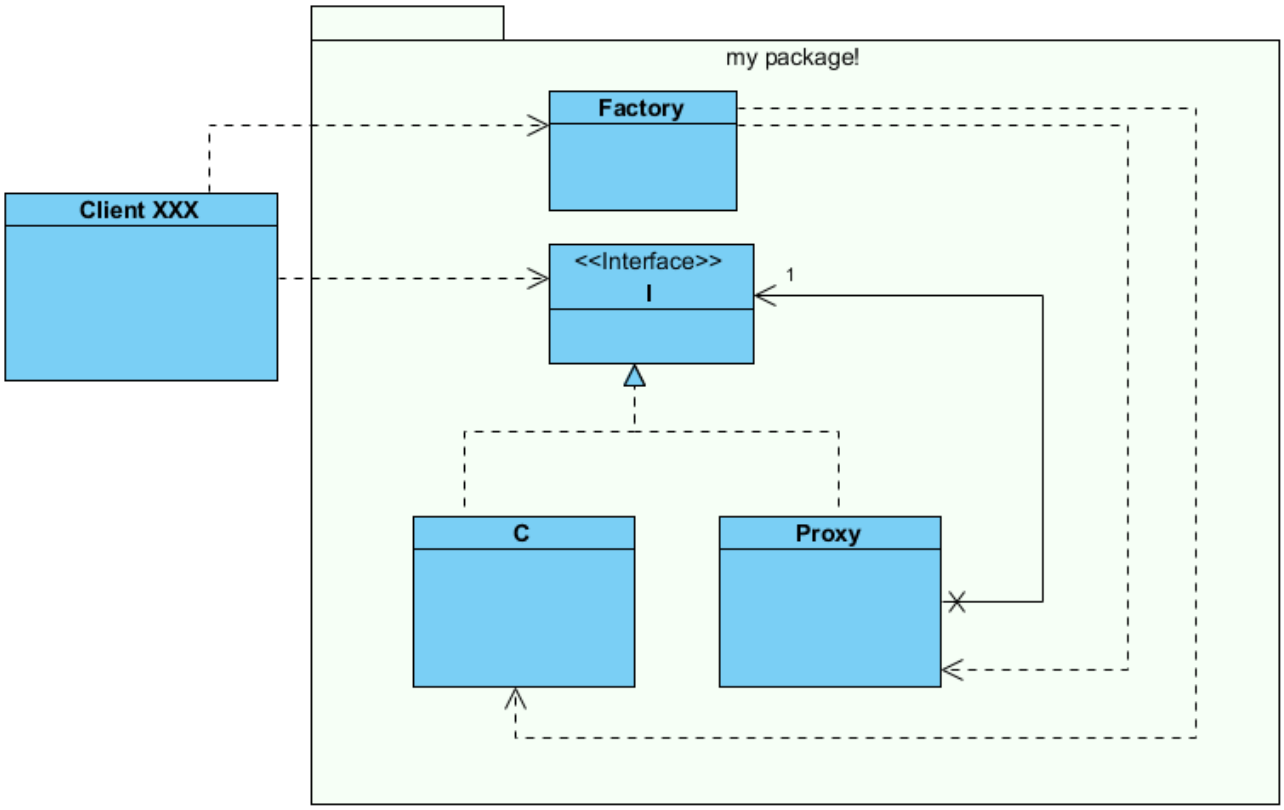
- Vérifications autorisations,
- gestion cache,
- Remote,
- ...

Schéma GoF



proxy-gof

Schéma plus réaliste

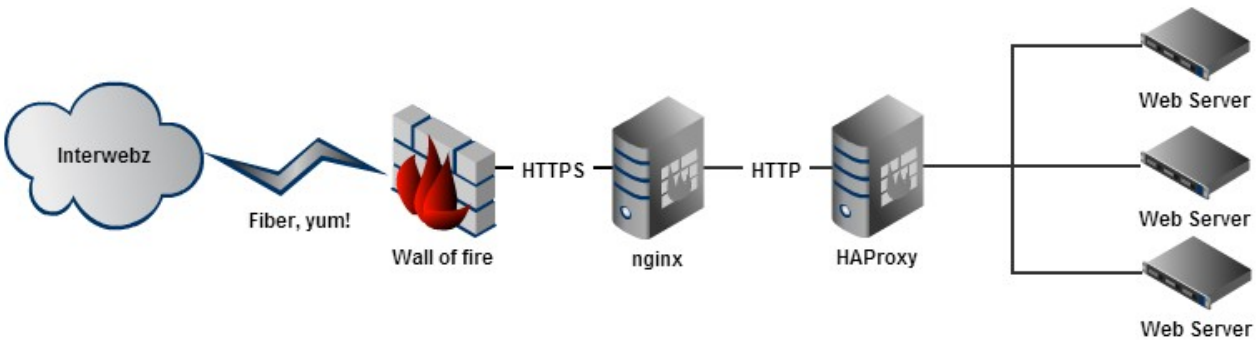


proxy-real

Exemple : contexte serveur http

Vous avez codé un serveur http

- Il faut ajouter https
- Un load balancer
- Une détection d'intrusion



proxy-server-http

8.6.3 Proxy vs Decorator externe

- 2 patterns du GoF
- Quasi même schéma UML
- Intention différente ?
  - Proxy : intermédiaire technique. En général transparent, caché par une factory
  - Decorator externe : apport fonctionnel. Souvent créé explicitement

> Frontière floue donc confusion régulière !

8.6.4 Composite

On veut pouvoir traiter de manière indifférencier un ensemble et un élément

- Exemple : un dessin qui contient des formes géométriques que l'on peut déplacer
- Exemples : les composants graphiques du JDK
  - Les menus auxquels on peut ajouter des entrées ou des sous-menus
  - ...

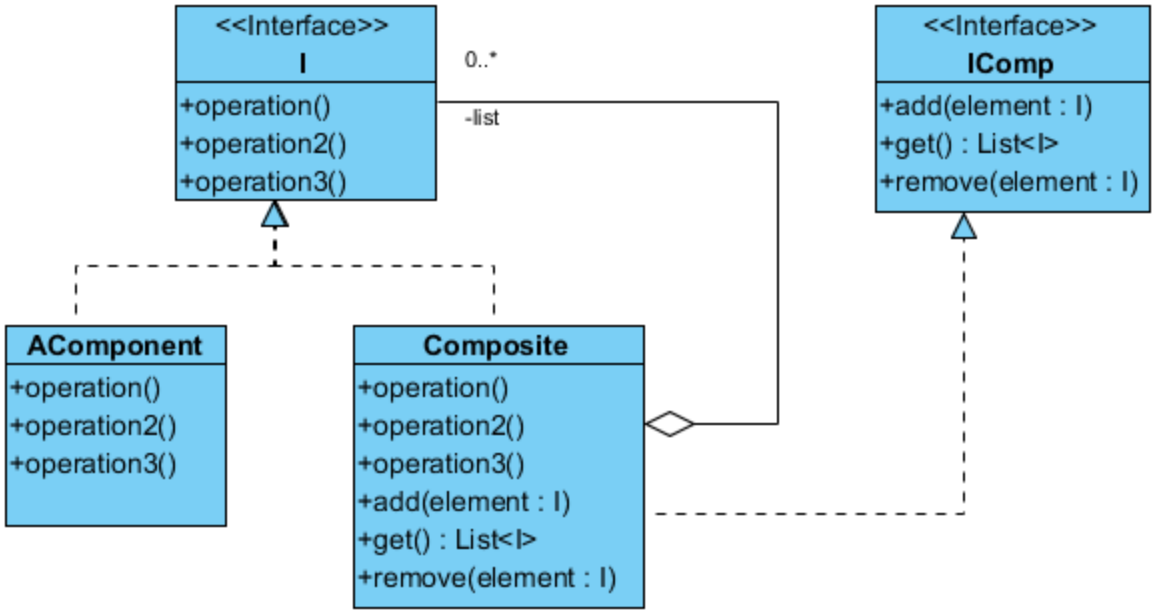
Composite

Un composant et un composite (le contenant) doivent avoir la même interface.

L'interface commune est vraiment la partie commune !

Principalement deux possibilités :

- les composites implémentent une interface spécialisées, comme dans le schéma ci-dessous



composite-safe

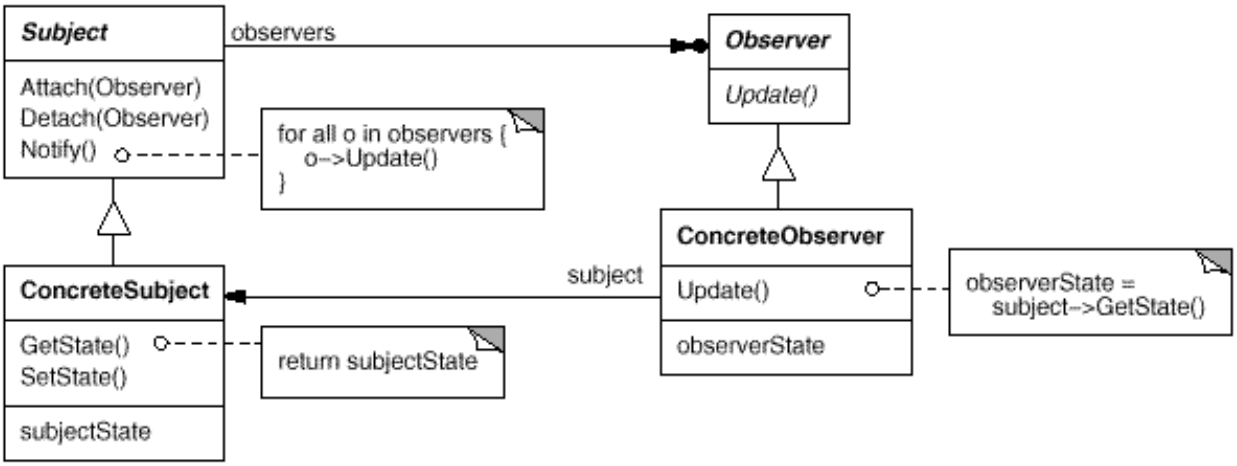
- les composites reçoivent leurs composants à la construction et la liate n'est pas mutable

8.7 Design Patterns "behavioural"

8.7.1 Observer

- Coupler des classes
  - Mais sans dépendances !
- En IR1/IG1/L3, on codait directement (sans réfléchir ?)
- Puis, on a appris à utiliser la délégation
- L'observer va encore plus loin :
  - Je ne délègue plus une opération
  - Je "préviens" "quelqu'un" de quelque chose
  - Je ne sais pas QUI je préviens
  - Je ne sais donc pas CE qu'il fera

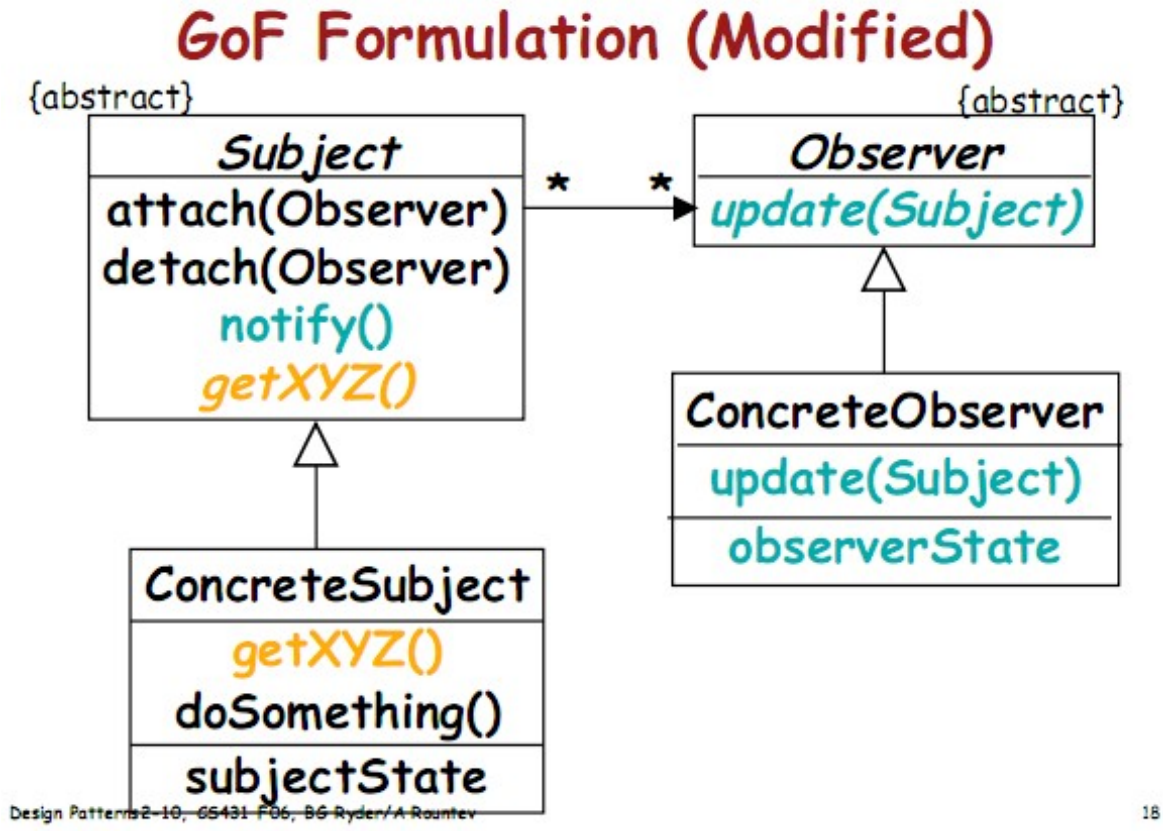
Schéma GoF



observer-gof

Schéma plus courant

- Souvent, l'observer ne connaît pas la classe concrète du sujet
- Souvent, l'observer reçoit l'objet concerné lors de l'appel des méthodes de notifications



observer-std

Les apports

> Super moyen pour découpler !
>
> Open Close !

- Je peux interagir avec un objet déjà existant
- En l'écoutant
- Alors qu'il ne me connaît pas ! Ni mon interface !

Les difficultés

L'implémentation de base d'un observer est triviale ... mais :

- Attention au mode de notification
  - Push ou pull
  - Impact important sur les performances
- Attention à la sémantique de notification :
  - Interface unique ou pas
  - Granularité des méthodes de notification
  - Impact important sur les performances

8.7.2 Visitor

Un des Design Patterns un peu dur à appréhender.

> Mais qu'il est indispensable de comprendre. Il est très régulièrement utilisé.

Contexte et besoins

Nous avons une hiérarchie d'objets, utilisés dans une structure complexe (Arbre, graphe, ...)

- Les clients veulent pouvoir exécuter un code qui dépend du type réel de l'objet
- On ne veut pas avoir plein de if et/ou instanceof !
- Nous ne connaissons ni ne voulons connaître le code du client !

> impossible d'utiliser l'appel de méthode virtuel directement car cela suppose d'inclure le code du client dans les objets de la hiérarchie

Mais on veut tout de même avoir l'efficacité de l'appel virtuel, Et que le code puisse changer en fonction du client

Comment

On utilise une implémentation qui effectue un *double dispatch* :

- on crée une interface Visitor qui contient une méthode *visit(X x)* différente par type concret X de la hiérarchie.
- le type le plus haut dans la hiérarchie a une méthode

```
abstract accept(Visitor v)
```

- les types concrets de la hiérarchie implémentent cette méthode de la manière suivante :

```
void accept(Visitor v) {
   v.visit(this);
}
```

Pour utiliser le visiteur, plutôt que d'appeler *visit* sur l'objet de la hiérarchie, on appelle *accept* de l'objet sur le visiteur

```
visitor.visit(object) —› object.accept(visitor)
```

Même dans les méthodes *visit*, on n'appelle pas directement les méthodes *visit* sur les objets liés, mais bien *accept* de l'objet sur visiteur (i.e. this)

```
visit(object) —› object.accept(this)
```

> La surchage est static, pas le polymorphisme !

Pourquoi ça marche

Dans chaque type concret X, *visitor.visit(this)* appelle la méthode *visit(X x)* car this est typé statiquement X

> rappel : le choix de la méthode parmi les surcharges se fait en fonction du type déclaré

Quand on appelle *object.accept(visitor)*, la méthode *accept* choisie est celle du type réel de object

> Polymorphisme classique : cette implémentation appelle la bonne méthode *visit* du visiteur

9 Annexes

9.1 Bonnes pratiques POO

9.1.1 La base

- Responsabilité
- 1 classe == 1 responsabilité
- Encapsulation
- champs private !
- Intégrité
- Un seul constructeur exigeant, tester les arguments !
- Pas de getter et surtout setter inutiles !
- Évolutivité
- Les méthodes publiques prennent et retournent des interfaces
- Interface explicite et adaptée au besoin
  - Noms corrects, verbe d'action pour les méthodes
  - Pas d'interface si pas d'utilisation commune de 2 classes
  - Pas de méthodes inutiles (- de 10 méthodes SVP)

9.1.2 Hiérarchie de types

- Pas de classes inutiles
- Ne pas se précipiter pour construire des hiérarchies
- Héritage est souvent une fausse bonne idée
  - Couplage trop fort entre les classes
  - Privilégier la délégation qui est plus souple
  - Héritage: lien entre classes à la compilation
  - Délégation: lien entre objets à l'exécution

9.1.3 Open / Close

Close them all !

Pas mal de design patterns permettent d'utiliser/tourner autour du fait qu'une classe peut être fermée

9.2 Tests unitaires

9.2.1 Définition

Un des tests parmi

- Les tests d'intégration,
- test utilisateur,
- test de recette,

- test de performance,
- ...

C'est un test automatisé local à une classe ou un petit nombre de classes qui marchent ensemble

Sans dépendances, sans se soucier des autres classes. pas de base de donnée,

> Notion de Conformité (couverture du code)
>
> Aide à la Non-Régression

9.2.2 Savoir quoi tester

- On teste les cas qui marchent
  - Renvoie bien les bonnes valeurs
- On teste les cas qui ne marchent pas
  - Valeur indiquant une erreur, exceptions si ...
- On teste les cas limites
  - Cas qui risque de ne pas marcher (expérience !)

9.2.3 Exemple : chercher un caractère dans une chaîne

- On teste les cas qui marchent
  - On cherche 'a' dans 'tard''
- On teste les cas qui ne marchent pas
  - On cherche 'b' dans 'tot''
  - On cherche 'c' dans null
- On teste les cas limites
  - On cherche en première et dernière position
  - On cherche avec plusieurs occurrences ... (vérifier le contrat ...)

9.2.4 JUnit 5

Ecrire un premier test

Consultez le guide JUnit 5 !

Structure d'un test

On écrit une classe FooTest si la classe a tester s'appelle Foo

On utilise l'annotation *@Test* pour indiquer qu'il s'agit d'une méthode de test

```
public class Foo {
  @Test
  public void myFirstTest()  {
    ...
  }
```

```
}
```

Vérifier des propriétés

On utilise des méthodes assert* pour vérifier des propriétés

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class Foo {
  @Test
  public void myFirstTest()  {
    assertEquals(3, 2 + 1);
  }
}
```

Les principales possibilités :

- *assertEquals(expected, value)* ok si .equals()
- *assertSame(expected, value)* ok si ==
- *assertTrue/assertFalse(value)*

9.2.5 Test & Conception

Pourquoi

- Pas de test, pas de confiance sur le code
- Pas de test, pas de refactoring
- Pas de test, pas d'évolution du design
- Pas de test, pas de chocolat

Règles d'or

1. Ne pas modifier un test qui échoue
2. Les tests n'expliquent pas pourquoi cela plante
3. Les tests ne garantissent pas à 100% que cela marche

9.2.6 La pratique

Ecrire un test unitaire en pratique

Un test unitaire doit être simple et "bourrin" :

- Pas d'algo

- Pas de ruse

Un test unitaire doit tester une seule et unique chose

Ne pas regarder le détail de l'implémentation pour écrire un test :

> Se concentrer sur le contrat.

Quand écrire un test en pratique

- Si on a un bug pas couvert par les tests,
  - on commence pas écrire un test, qui doit planter !
  - On reproduit le bug
- Si on veut écrire une API :
  - *TDD*
- Dès que l'on commence à avoir les idées assez claires sur le design et le code
  - Pas trop tôt,
  - pas trop tard
  - *expérience !*

9.2.7 Test Driven Development

Pratique de dévelopement qui consiste à écrire les tests d'abord

- Garantit que les tests sont exhaustifs
- On s'oblige à penser à tous les cas de test
- Se concentrer sur le contrat et pas sur le comment
- Le danger, on ne s'occupe pas du comment

A écouter : TDD is Dead.

[^2] : Effective Java, Second Edition, by Joshua Bloch. Une des références des développeurs Java.