

Formation :

Conception et Design Patterns

(2 jours)

Formateur

Dr. Bassem Seddik
Maître Technologue en informatique
Consultant en IT, entrepreneur
Bassem.seddik@gmail.com

Plan de formation

- ▶ **I- Présentation générale**
 - 1. Rappel rapide sur la POO et UML
 - 2. Introduction aux Design Patterns
 - 3. Quel DP choisir?
 - 4. Vous convaincre d'utiliser les DP
 - 5. Familles de patterns (GoF, Grasp)
 - 6. Le Refactoring
- ▶ **II- Les principes fondamentaux**
 - 1. Les Couplages
 - 2. Single Responsibility Principle (SPR)
 - 3. Open-Closed Principle (OCP)
 - 4. Liskov Principle (LSP)
 - 5. Interface Segregation Principle (ISP)
 - 6. Dependency Inversion Principle (DIP)
- ▶ **III- Patterns de création**
 - 1. Singleton
 - 2. Factory method et Abstract Factory
- ▶ **IV- Patterns de structuration**
 - 1. Adapter
 - 2. Facade
 - 3. Composite
 - 4. Decorator
 - 5. Proxy
- ▶ **V- Patterns comportementaux**
 - 1. Template
 - 2. Observer
 - 3. State
 - 4. Command
 - 5. Chain of responsibility
- ▶ **VI- Patterns d'architecture**
 - 1. MVC
 - 2. MVVM
- ▶ **VII- Démarche et principes GRASP**
 - 1. Faible couplage
 - 2. Forte Cohésion
 - 3. Règle de l'expert
 - 4. Utiliser le créateur
 - 5. Gérer le contrôleur
 - 6. Appliquer le polymorphisme
 - 7. Ne pas parler aux inconnus
 - 8. Indirection, Pure fabrication et Points de variation
 - 9. Autres principes
 - 10. Odeurs de codes

Présentation du formateur

Bassem Seddik:

- ▶ Docteur en informatique,
- ▶ 15 ans d'expérience professionnelle
 - ▶ 10 ans comme enseignant universitaire
 - ▶ Spécialisé en Développement Web et Multimédia 2D/3D
- ▶ Formateur et consultant IT



Objectifs pédagogiques

1. → Catégoriser les patterns
 2. → Connaître les principaux patterns
 3. → Comprendre la philosophie des Design Patterns (DP)
 4. → Savoir transiter les patterns depuis UML vers le code et inversément
 5. → Mettre en oeuvre les Design Patterns sous des exercices pratique en langage Java.
 6. → S'aligner avec les **best-practices** afin de Concevoir et créer des solutions algorithmiques résolvant un problème donné
-
- ▶ **Public concerné**
 - ▶ Architectes, ingénieurs concepteurs, responsables informatiques, chefs de projets, décideurs informatiques et développeurs objet (Java, .Net).

I.2- Introduction aux Design Patterns (DP)

Prérequis:

- ▶ Ce cours sur les **patrons de conception** ou **Design Patterns** nécessite
 - ▶ La conception des diagrammes UML,
 - ▶ La Programmation Orientée Objet (POO).

Notions fondamentales

- ▶ Vous allez pouvoir progresser régulièrement sur:
 - ▶ Les diagrammes UML relatifs (classe ou séquence)
 - ▶ La mauvaise façon de faire les choses 😞
 - ▶ La solution apportée par le Design Pattern 😊

Travaux pratiques

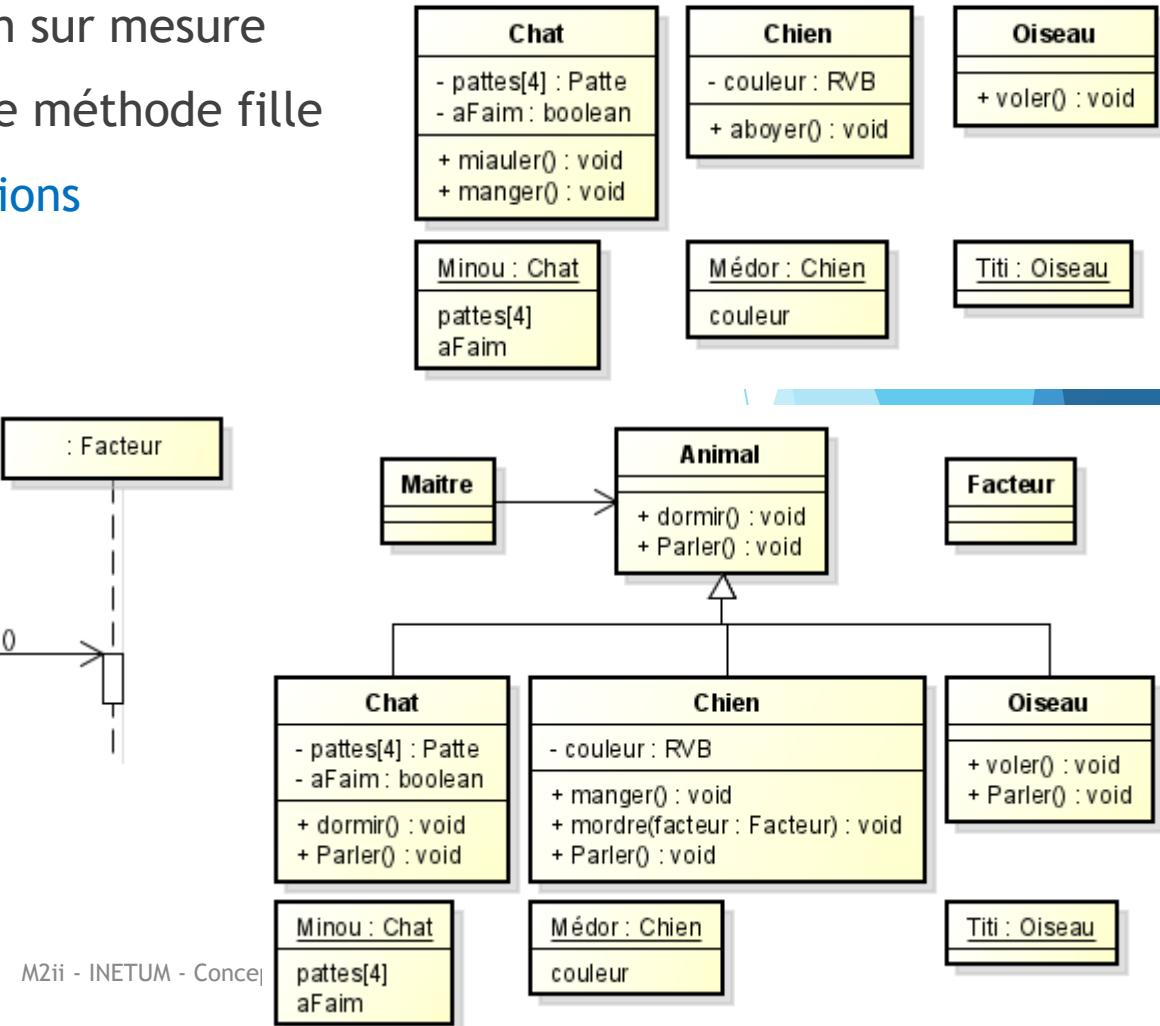
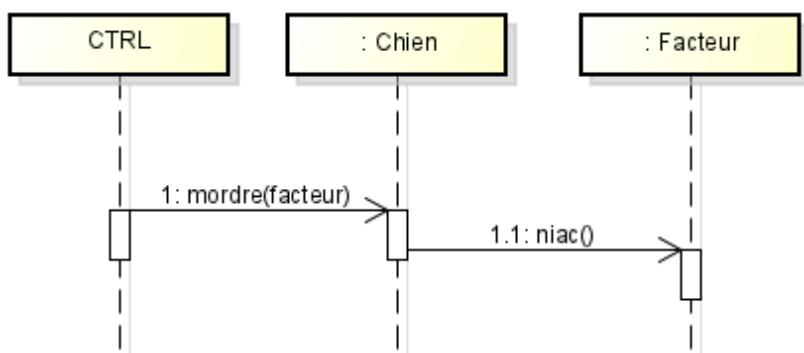
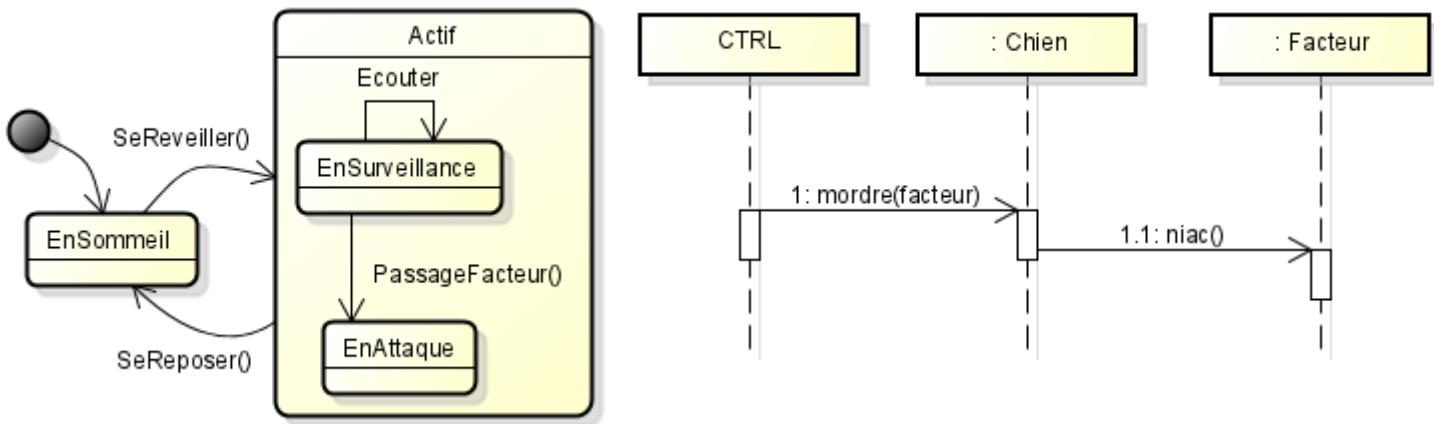
- ▶ Nos codages se feront sous la langage Java,
 - ▶ Mais le mêmes notions sont applicables sous C#, C++, Python3, JavaScript (ES.6), ...

I-Présentation générale

1. Rappel rapide sur la POO et UML
2. Introduction aux Design Patterns (DP)
3. Quel DP choisir?
4. Vous convaincre d'utiliser les DP
5. Familles de patterns (GoF, Grasp)
6. Le Refactoring

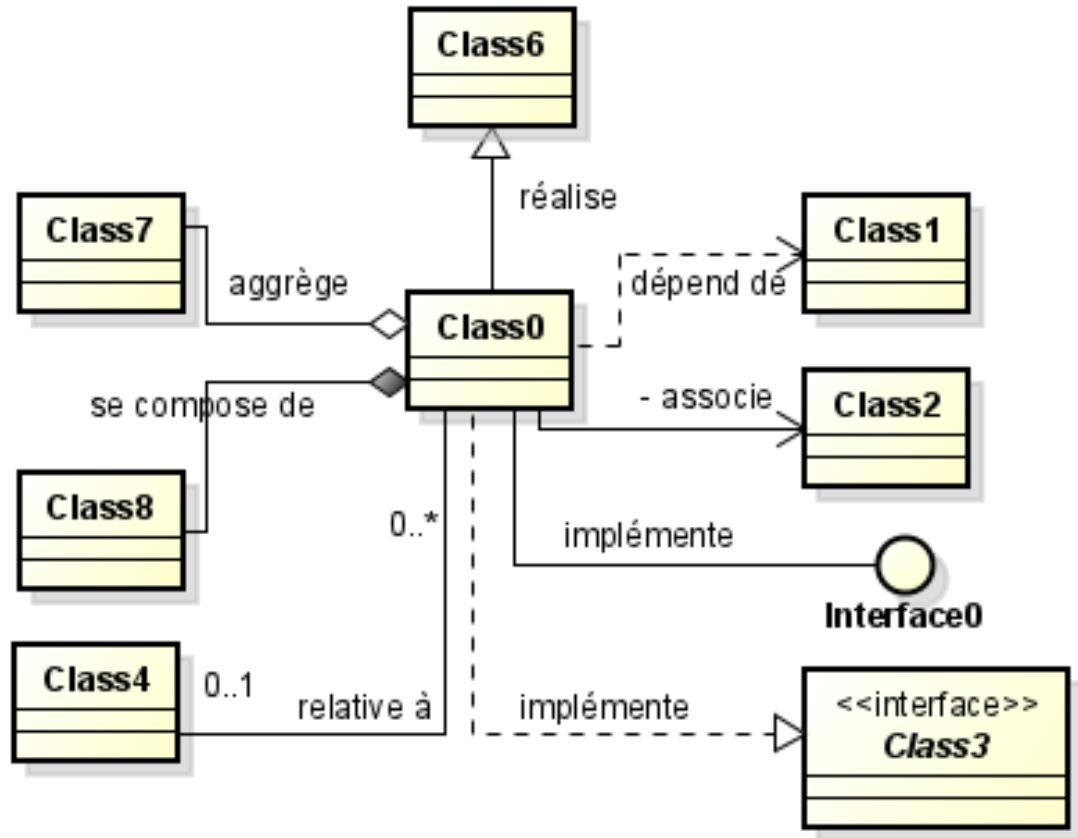
I.1- Rappel rapide sur la POO et UML

- ▶ Un **objet** (jouet plastique) est la concrétisation d'une **classe** (moule métal)
- ▶ L'**encapsulation**: Des niveaux de visibilité/protection sur mesure
- ▶ Le **polymorphisme**: Un appel automatique à la bonne méthode fille
- ▶ Envoi de messages par appels de **méthodes** ou **fonctions**
- ▶ Un objet peut être sous différents **états**



I.1- Rappel rapide sur la POO et UML (suite)

- ▶ Les différentes relations possibles entre les classes:
 - ▶ Généralisation
 - ▶ Dépendance
 - ▶ Association unidirectionnelle
 - ▶ Association bidirectionnelle
 - ▶ Implémentation
 - ▶ Agrégation
 - ▶ Composition



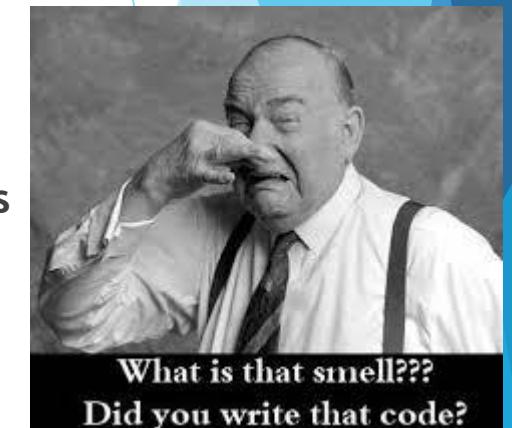
Voir Quiz de départ d'évaluation des pré-requis

I.2- Introduction aux Design Patterns (DP)

- ▶ En génie logiciel, vous rencontrerez souvent le même problème de conception à plusieurs reprises.
 - ▶ Il existe tant de façons de faire face à ces problèmes récurrents
 - ▶ au fil du temps, certaines solutions sont préférées à d'autres, parce qu'elles sont plus flexibles ou réutilisables.
- ▶ Un design pattern est une solution pratique éprouvée à un problème récurrent de conception.
 - ▶ Au lieu de résoudre un problème logiciel particulier en plongeant dans la programmation des principes de base à chaque fois,
 - ▶ Repenser à chaque fois quels objets utiliser?
 - ▶ Comment les objets doivent se relier et ainsi de suite?
- ▶ Vous pouvez utiliser des solutions précédemment décrites que les développeurs experts ont souvent utilisées.
 - ▶ Les DP ne sont pas des propositions théoriques d'intérêt académique seulement,
 - ▶ Ce sont des solutions réelles qui sont utilisées dans les logiciels industriels.

I.2- Intro.: Qu'est-ce qu'on fera avec les DPs ?

- ▶ Dans ce cours, vous étendrez vos connaissances sur la conception et l'analyse orientée objet:
 - ▶ Les problèmes de conception d'applications peuvent être résolus grâce à des patrons de conception couramment appliqués par les **experts**
 - ▶ Vous pourriez appliquer des modèles de conception pour résoudre des situations très **couramment rencontrées** en développement logiciel.
 - ▶ Grâce à une étude des patrons de conception établis, vous obtiendrez une base solide pour "attaquer" des applications logicielles plus **complexes**.
 - ▶ Vous pourriez également découvrir plusieurs **nouveaux principes** de conception pour faire des solutions un peu plus réutilisables, flexibles et maintenables.
 - ▶ Enfin, nous allons apprendre à identifier les mauvaises pratiques de conceptions de logiciels en référençant un catalogue **d'odeur de codes**.



I.2- Intro.: Les DPs VS recettes de cuisine

- ▶ Pensez aux **design patterns** comme recettes de cuisine.
 - ▶ Il y a des plats que les gens ont expérimentés et cuisinés encore et encore.
 - ▶ Après un certain temps, une certaine façon de cuisiner un plat particulier serait préférable parce qu'elle offre le meilleur gout.
 - ▶ Au fil des ans, les développeurs ont expérimenté de nombreuses solutions de conception différentes.

- ▶ Et ces **design patterns** esquissent des solutions qui créent souvent le meilleur résultat.
- ▶ Il y a 23 **design patterns** identifiés dans un célèbre livre appelé « *Design Patterns, Elements of Reusable Object-Oriented Software* »
 - ▶ Par les auteurs Gamma, Helm, Johnson, et Vlissides (la bande de quatre)
 - ▶ Ce cours offrira une sélection parmi ces modèles.



I.3- Quel DP choisir? Tel un jeu d'échec!

- ▶ Il existe de nombreux **design patterns** disponibles et parfois un **DP** particulier peut sembler applicable, mais il peut ne pas être réellement le bon ajustement au problème à portée de mains.
- ▶ Utiliser les **design patterns**, c'est un peu comme jouer à une partie d'échecs.
- ▶ Il existe de nombreuses façons de gagner en mettant votre adversaire en checkmate.
 - ▶ Un joueur d'échecs débutant ne peut connaître que les mouvements de base de chaque pièce.
 - ▶ Un joueur d'échecs expert cherchera des modèles appropriés sur le plateau.
- ▶ Il existe des modèles d'échecs établis appelés
 - ▶ le roi exposé
 - ▶ ou le pion isolé, par exemple.
- ▶ Les experts veulent exploiter ces modèles tout en jouant le jeu.
- ▶ Grâce à l'expérience, ils sont mieux en mesure de juger quels « patterns » utiliser ou créer dans une situation particulière pour gagner la partie.



I.3- Quel DP choisir? Des joueurs d'échecs aux programmeurs

- ▶ Comme aux échecs, en tant que développeurs de logiciels débutants, vous connaissez les éléments de la syntaxe linguistique dans les logiciels de programmation.
- ▶ Vous connaissez également des éléments de programmation: la façon de créer une boucle pour, une méthode, etc.
- ▶ Mais avec plus de pratique et d'expérience en tant que développeurs de logiciels,
 - ▶ Vous écrirez plus de **code idiomatique** qui suit les conventions communes.
 - ▶ Vous pouvez également devenir des experts en conception qui connaissent les **design patterns** à utiliser pour résoudre des problèmes particuliers de conception de logiciels.

Quelles sont les situations dans lesquelles un design pattern serait le mieux utilisé?

- A. Résoudre un problème commun de conception de logiciels qui peut avoir été rencontré avant.
- B. Résoudre un problème très unique pour une application particulière.
- C. Fixation du « code spaghetti » -- par exemple, code source qui n'a pas de structure ou ayant un flux emmêlé d'instructions.

I.4- Pour vous convaincre d'utiliser les design patterns

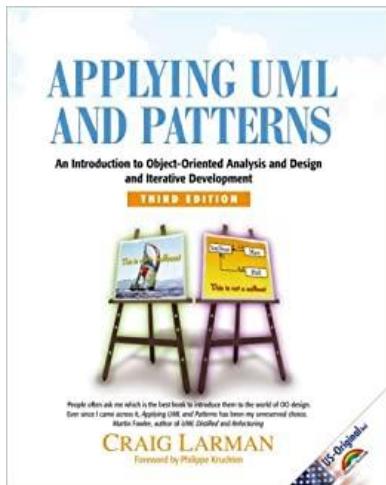
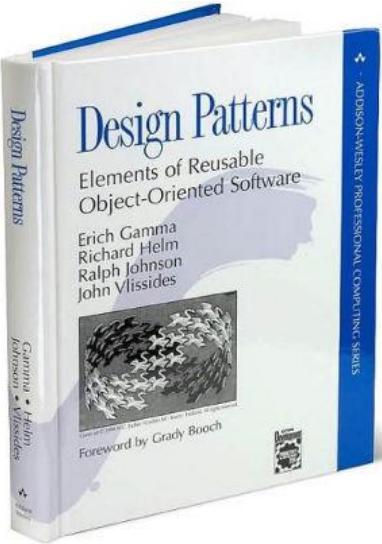
- ▶ Les **design patterns** aident les développeurs de logiciels afin de:
 - ▶ Ne pas tout construire à partir de zéro,
 - ▶ Avoir un guide qui les aide à résoudre les problèmes de conception dans la façon dont les experts le font.
- ▶ Un athlète débutant peut avoir le talent naturel et les connaissances de base du sport qu'il joue.
- ▶ Mais un **entraîneur avec des années d'expérience** sera en mesure d'aider cet athlète à devenir meilleur et utiliser son talent pour toucher à des performances plus intéressantes.
- ▶ Les **design patterns** sont déjà prouvés par les experts, et ils ont traversé les long chemins de validation et de consensus.
- ▶ Vous pouvez éviter les long essais que les précédents développeurs ont déjà traversés, et passer directement à la création de logiciels bien écrits.



I.4- Vous convaincre: Les DPs = VOCABULAIRE des experts

- ▶ Les **DP** aident par le simple fait de donner un nom à chaque modèle et en rendant la communication plus facile entre les développeurs
- ▶ Plutôt que d'expliquer les détails d'une solution de conception encore et encore, vous pouvez simplifier la discussion en indiquant votre DP utilisé.
- ▶ Example: Un problème de conception très commun dans les logiciels est le suivant :
 - ▶ vous avez deux objets et le premier objet dépend du deuxième objet.
 - ▶ Si le deuxième objet change, le premier objet doit être notifié.
- ▶ **Vous pouvez simplement indiquer que vous avez utilisé le DP « observer » ici!**
- ▶ Une connaissance préalable de ce DP particulier décrit une solution et explique clairement la situation.
 - ▶ moins de place pour une mauvaise compréhension
 - ▶ La description de toute personne de cette situation et de cette solution aurait pu être vague
- ▶ **Alors maintenant, il est temps de commencer à concevoir comme un expert.**

I.5- Familles de patterns (GoF, Grasp)

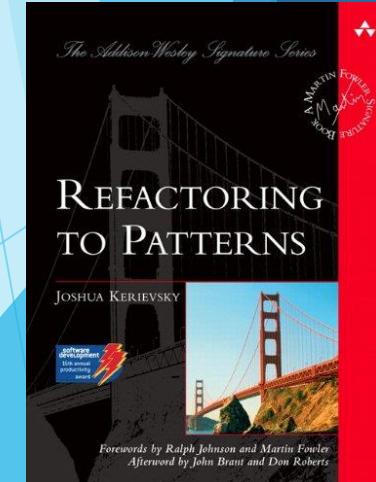


- ▶ Dans les années 90, 4 personnes consacrèrent leurs recherches aux design patterns :
 - ▶ Gamma, Helm, Johnson et Vlissides éditérent un livre qui fait toujours référence «Design Patterns : Elements of reusable object oriented software» en 1994.
 - ▶ Les 23 Patterns fondamentaux indiqués sont appelés de **GoF** pour « Gang of Four ».
- ▶ En parallèle, un autre Monsieur bien connu dans le monde du SCRUM et du développement AGILE: *Craig Larman*, avait proposé **GRASP**:
 - ▶ General Responsibility Assignment Software Patterns (or Principles) ou “patterns généraux d'affectation des responsabilités”
 - ▶ Décrit des modèles de conception intuitifs dans le livre « Applying UML and Patterns: An Introduction to Object-Oriented Analysis & Design » en 1997
- ▶ Dans cette formation, nous allons profiter des **principes fondamentaux de bon sens** et couvrir un maximum de DP depuis les **exemples concrets de DP**.



I.6- Le Refactoring : Réécrire le code

- ▶ Peu importe la façon dont vous concevez votre code, il y aura toujours des modifications à apporter.
- ▶ **Le Refactoring** aide à gérer cela. C'est un processus de modification du code afin que la structure interne soit améliorée, sans que les comportements externes ne soient pas modifiés.
- ▶ Il est mis en œuvre à travers des changements incrémentaux de la structure du code, avec idéalement :
 - ▶ des tests fréquents pour s'assurer que ces changements n'ont pas modifié le comportement du code.
 - ▶ Des changements réalisés lorsque de nouvelles fonctionnalités sont rajoutées, et non pas lorsque le code est terminé. Cela permet de gagner du temps et facilite l'ajout des fonctionnalités.
- ▶ Les modifications sont nécessaires dans le code lorsque le mauvais code émerge.
 - ▶ Tout comme des DP peuvent émerger comme candidats, du mauvais code peut émerger aussi.
 - ▶ Ceux-ci sont connus sous le nom de **anti-patterns** ou **code smells**.
- ▶ Le livre **Refactoring** de Martin Fowler a identifié bon nombre de ces **anti-patterns**.
 - ▶ Les **Code smells** aident à “sentir l’odeur” de ce qui est mauvais dans le code.
 - ▶ Fournit également des refactorisations afin de transformer le code, et “améliorer l’odeur”!



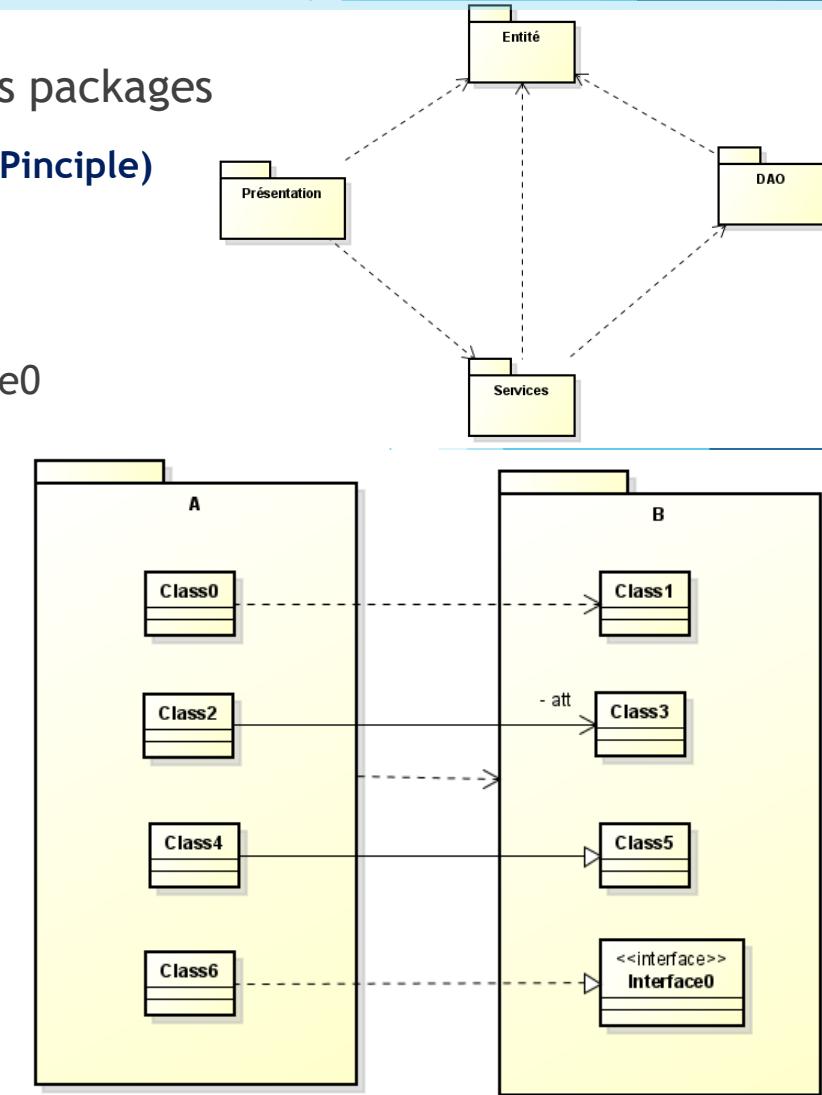
II- Les principes Fondamentaux

1. Les Couplages
2. Single Responsibility Principle (SPR)
3. Open-Closed Principle (OCP)
4. Liskov Principle (LSP)
5. Interface Segregation Principle (ISP)
6. Dependency Inversion Principle (DIP)

Les principes fondamentaux: Couplage

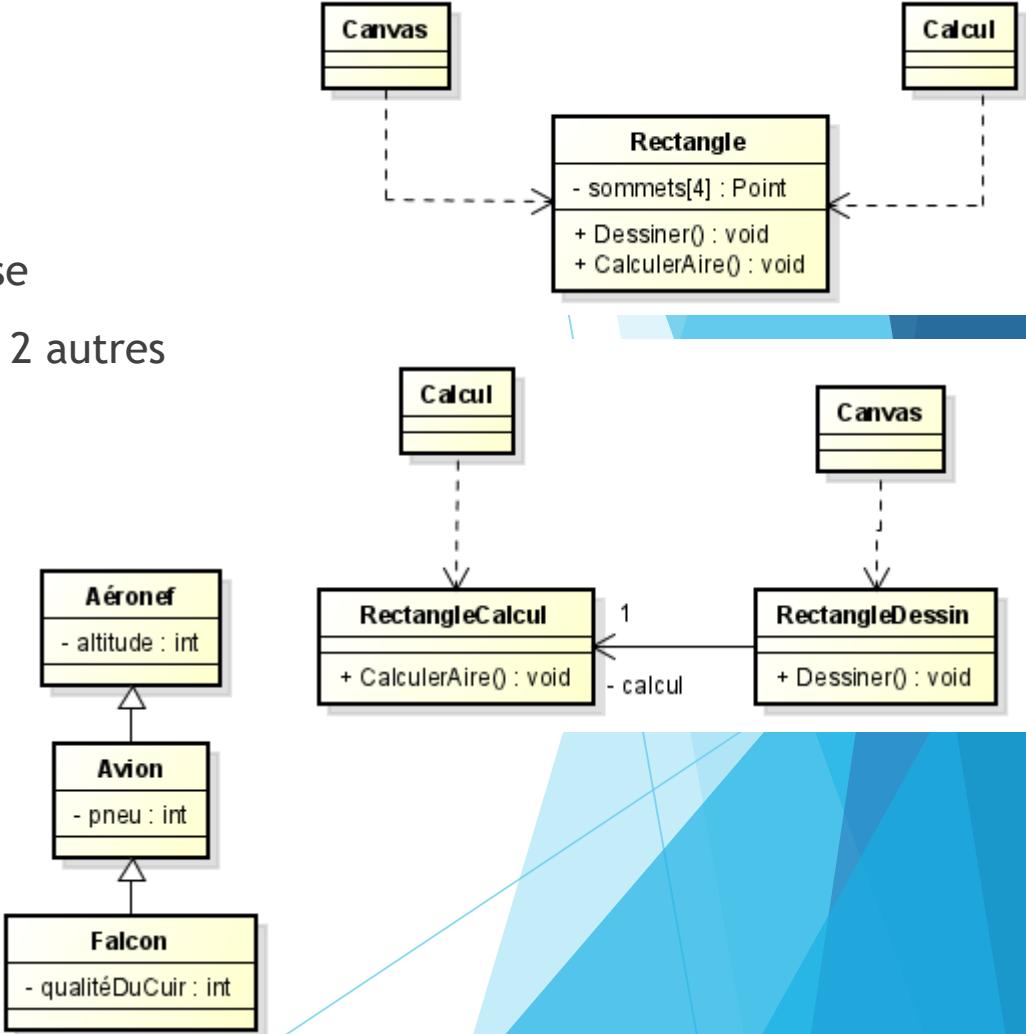
Les couplages : Les liaisons de dépendance qui existent entre les différents packages

- ▶ On cherche à respecter au maximum le principe **ACP (Acyclic dependency Principle)**
 - ▶ Pas de dépendance cyclique entre les packages
- ▶ Les origines du couplage sont causés par les sous-classes présentes
 - ▶ Une **dépendance**: Une variable/paramètre local de type classe1 sous la classe0
 - ▶ Une **association**: La modification d'une classe impacte un **attribut** de l'autre
 - ▶ Un **héritage** (généralisation): Encore plus fort, c'est un couplage structurel très difficile à modifier par la suite
 - ▶ Une **interface**: équivalente à un héritage avec la différence d'indication que c'est une classe considérée comme stable
- ▶ Selon le cas, on peut avoir une application:
 - ▶ **fragile**: la modification d'un élément impacte tout le reste
 - ▶ **Immobile**: Elle dépend de tellement de choses, qu'elle est non réutilisable
 - ▶ **Visqueuse**: Elle présente une complexité énorme qu'elle ne peut évoluer sans se voir virer à coté



Les principes fondamentaux: SRP

- ▶ **Single Responsibility Principle (SRP)**
→ Une classe ne doit avoir qu'une seule responsabilité
- ▶ **Le mauvais exemple**
 - ▶ On partage entre deux classes très différentes une même classe
 - ▶ Cette classe devrait fournir des services compatibles avec les 2 autres
- ▶ **Le bon exemple1**
 - ▶ Chaque classe a une responsabilité unique par **délégation**
 - ▶ On découpe alors la classe Rectangle en 2 classes
 - ▶ Ces classes peuvent communiquer entre elles si on le veut
- ▶ **Bon exemple 2**
 - ▶ à base **d'héritage**
 - ▶ Une séparation par isolation verticale
(la précédente pourrait être vue comme horizontale)



Les principes fondamentaux: OCP

► Open-Closed Principle (OCP)

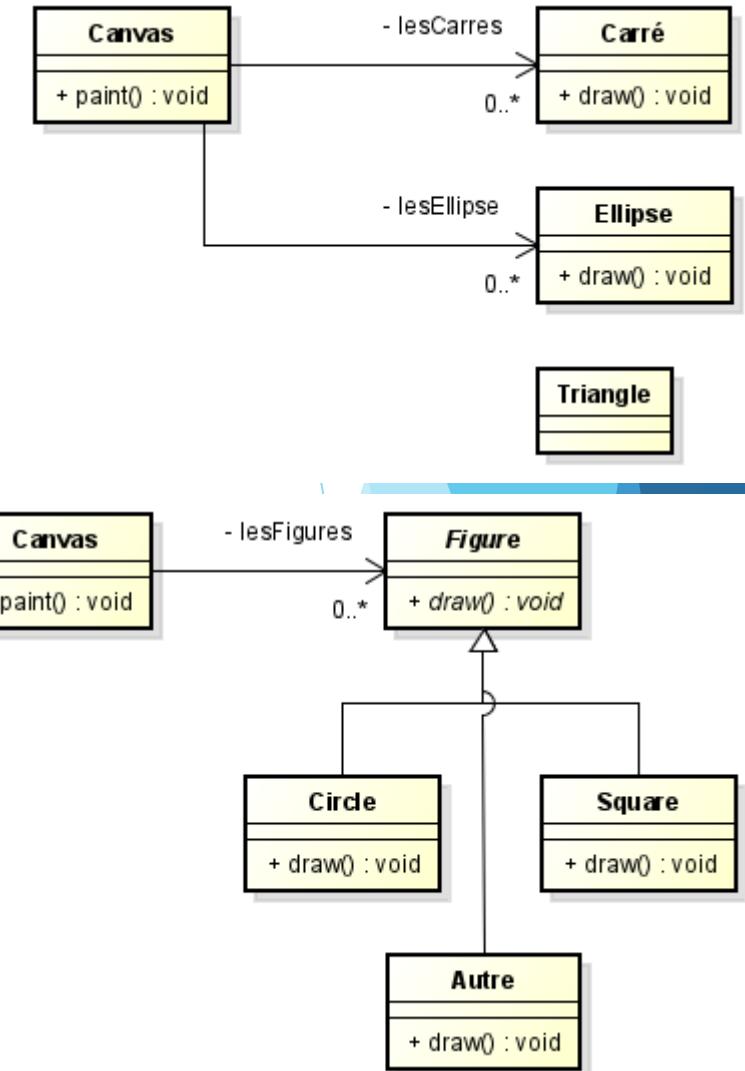
- ▶ Prévoir un modèle ouvert avec des points d'extension bien définis
- ▶ Un objet doit être ouvert à l'extension, mais fermé à la modification
 - ▶ On pourra spécifier explicitement les points d'extension
 - ▶ Mais, certainement, forcer d'autres classe à être de type Final, par exemple.
Donc non extensibles

► Le mauvais exemple

- ▶ Forcément on devra copier/coller du code redondant
- ▶ Besoin de modification de la classe Canvas pour évoluer le projet

► Le bon exemple1

- ▶ Le Canvas ne connaît que la classe Figure
- ▶ La nouvelle classe dispose de
- ▶ On profite ainsi de l'héritage et du polymorphisme
- ▶ Le point d'extension est aussi spécifié comme classe abstraite (*en italique*)



Les principes fondamentaux: LSP

► Liskov Principle (LSP)

- ▶ Ici la classe fille doit se comporter à 100% comme sa classe mère
- ▶ Eviter de se retrouver chez le client avec un code conditionnel
 - ▶ Eviter de se retrouver avec des tests de types sur les données
- ▶ Donc une classe fille est une **extension** conforme au comportement initial de sa mère

► Le mauvais exemple

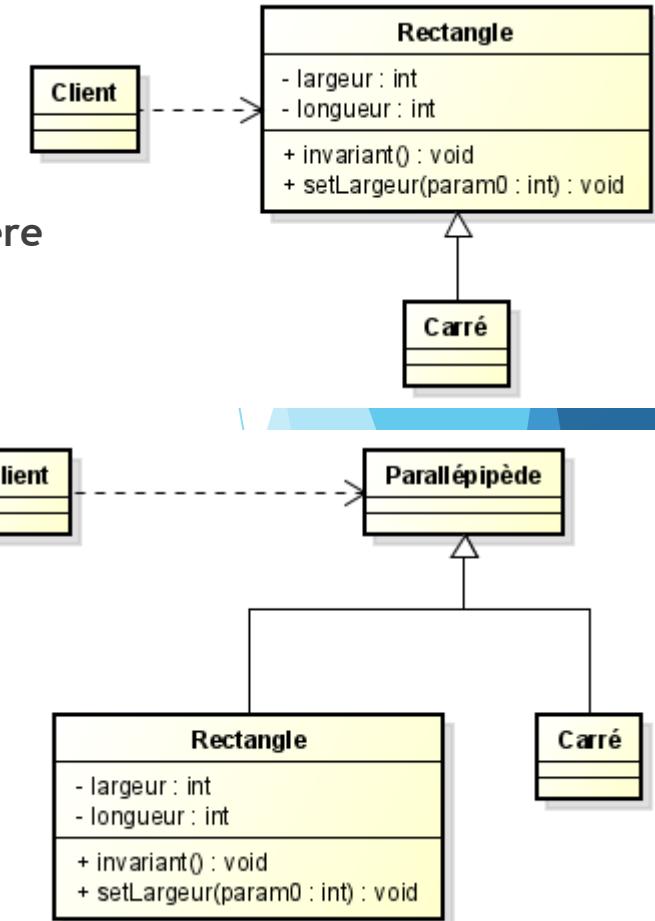
- ▶ Le carré est bien évidemment un rectangle
- ▶ Mais est-ce correcte du point de vue de comportement
- ▶ Le client peut modifier la largeur et hauteur et donc pose problème au carré

► Le bon exemple1

- ▶ Selon Liskov, ici le carré est un parallélépipède avec des nuances
- ▶ Le Rectangle est indépendant ainsi du Carré

► Exceptions existent chez certains Frameworks:

- ▶ Afin d'unifier les méthodes, des tests de types sur les classes filles sont pratiqués



Les principes fondamentaux: ISP

► Interface Segregation Principle (ISP)

- ▶ On utilise des interfaces pour réduire les dépendances au strict utile
- ▶ Recommande d'avoir **des interfaces** plus petites et plus spécifiques afin que les clients ne soient pas dépendants de méthodes qu'ils n'utilisent pas.

► Le mauvais exemple

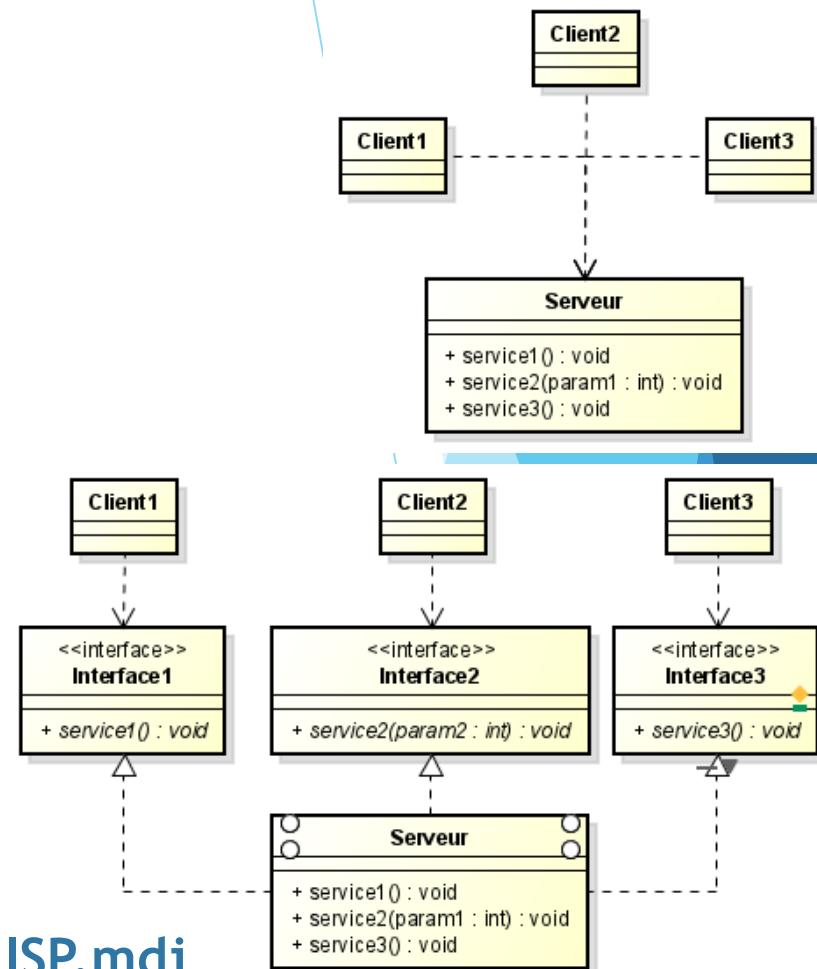
- ▶ Chacun des client dépend des services des autres clients
- ▶ Les flèches de dépendances sont à la cause de ce comportement puisque la même classe du serveur est appelée par le client

► Le bon exemple1

- ▶ Prévoir des interface qui éliminent les dépendances structurelles
- ▶ Les interface sont la clé magique pour obtenir le découplage
- ▶ Le client ne dépend que des services dont il a besoin

► Exercice pratique1 sous Star UML:

- ▶ Proposez votre modification appropriée pour le fichier ISP.mdj



Les principes fondamentaux: DIP

► Dependency Inversion Principle (DIP)

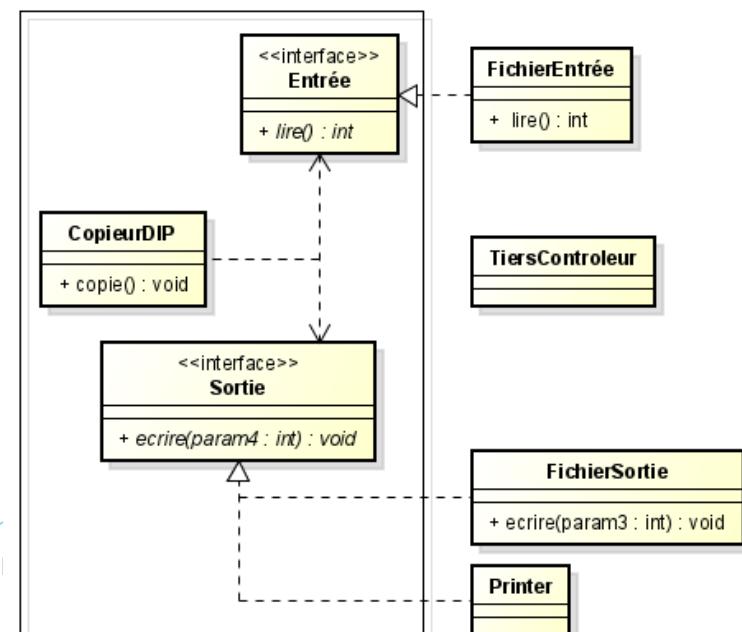
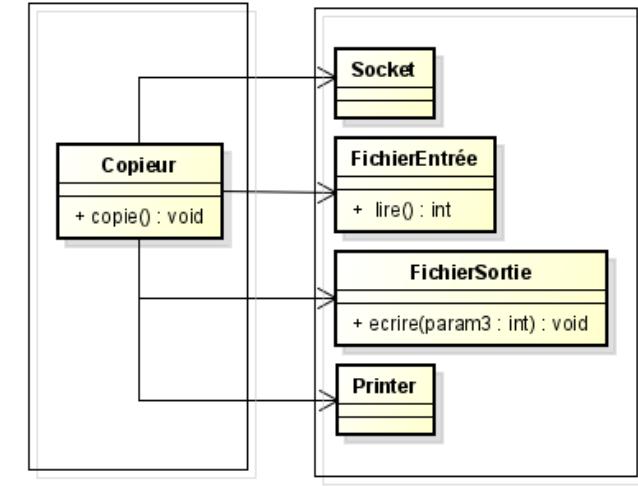
- ▶ C'est aux classes concretes de devoir dépendre des classes abstraites
- ▶ Viser à des dépendances de haut niveau basées sur des superclasses abstraites
- ▶ Alors, on inverse les dépendances (par héritage ici)

► Le mauvais exemple

- ▶ Tout d'abord un copieur depuis fichier d'entrée vers fichier de sortie
 - ▶ Ensuite, succès et on nous demande de rajouter l'usage des entrée socket
 - ▶ Ensuite, succès, on nous demande de rajouter sortie sur printer, ...
- ▶ Cela fait plein d'objets concrets, donc instables
 - ▶ Alors que la copie est la partie abstraite et stable qui fait toujours la même chose
- ▶ Chaque modification du système force la modification (voir sens de flèches) de la partie sensée être stable

► Le bon exemple1

- ▶ Les entrée/sortie sont bien spécifiées sous des interfaces
- ▶ Attention, cela prend plus de temps à coder (version précédente très rapide).
 - ▶ Besoin d'injection de dépendances depuis un contrôleur qui créera les objets pour nous
 - ▶ Donc appliquer la règle de trois: 1) fait sale → 2) fait sale → 3) fait propre



III- Patterns de création

1. Singleton

2. Factory method et Abstract Factory

Note: Le langage de mise en œuvre (Java, JS, ou C#, ...) peut fortement influencer le DP à adopter.

Décorateur (à voir après)

Adaptateur (à voir après)

Façade (à voir après)



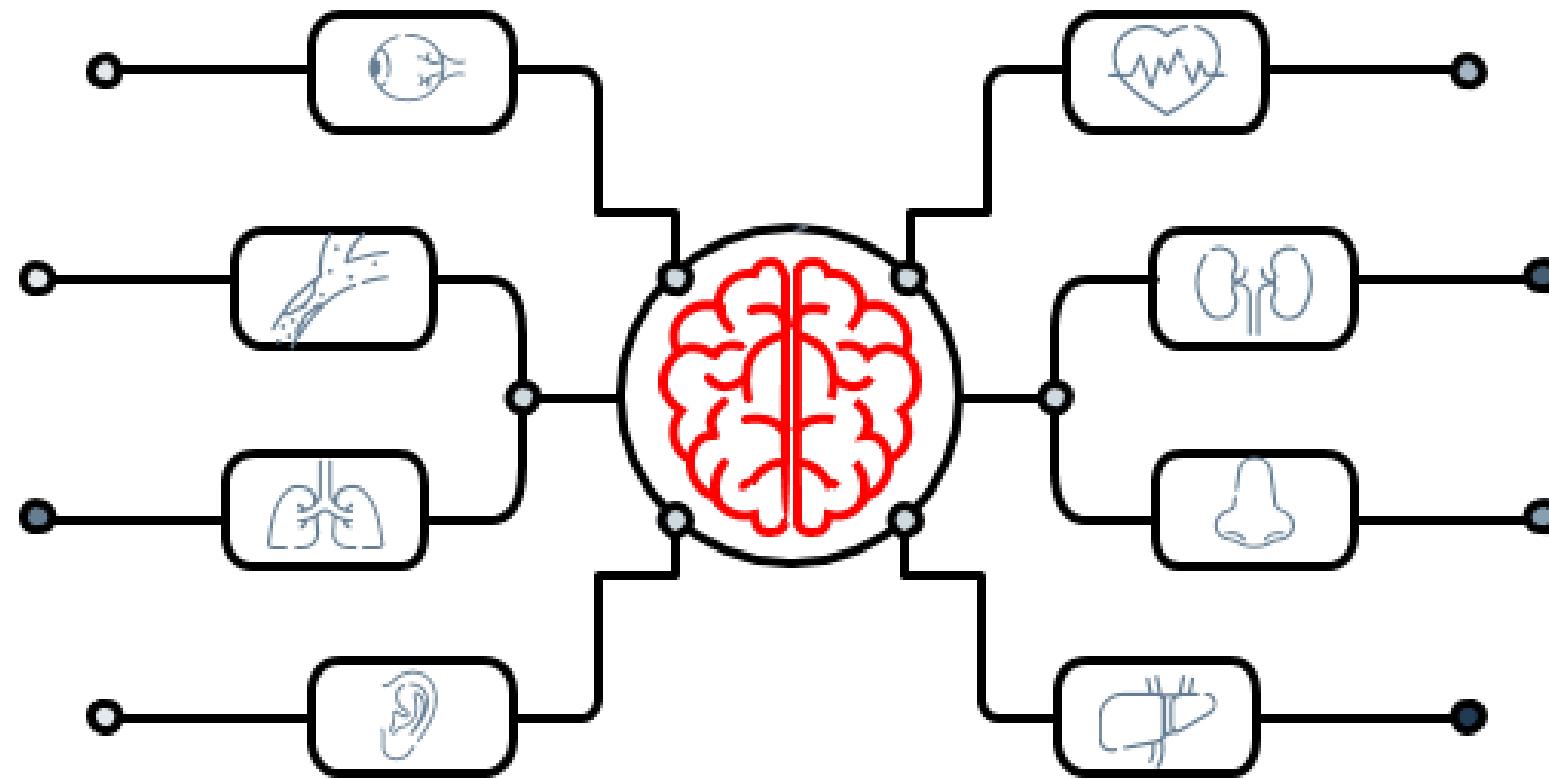
Les patterns de création (Creational patterns):

C'est les DPs qui permettent la création ou le clonage des objets.

Ils gèrent la façon dont vous gérez la création des nouveaux objets dans le code.



- ▶ **Singleton** - un DP qui applique une seule instantanéité d'une classe qui est accessible à l'échelle globale.
- ▶ **Méthode d'usine (Factory Method)** - un DP qui délègue l'instanciation concrète à une méthode de sous-classe.
- ▶ **Décorateur** - un DP permettant un comportement supplémentaire grâce à une enveloppe à base de composants.
- ▶ **Façade** - un DP qui consiste à encapsuler un système complexe dans une interface simple.
- ▶ **Adaptateur (Adapter)** - un DP qui relie et adapte deux systèmes incompatibles en fournissant une interface compatible avec les deux.



III.1- Singleton DP

Le DP Singleton

- ▶ Singleton est un DP de création qui décrit une façon de créer un objet.
- ▶ Le modèle singleton est l'un des exemples les plus simples parmi les DP, mais c'est une technique très puissante
- ▶ **Le modèle Singleton se réfère à n'avoir qu'un seul objet instance d'une classe.**
- ▶ Considérez un jeu de poker mobile:
 - ▶ Ce jeu a une classe de préférences pour définir où tous les paramètres sont stockés pour l'utilisateur.
 - ▶ Ces préférences incluent :
 - ▶ Les éléments visuels, tels que la couleur de la surface de jeu,
 - ▶ la conception sur le dos des cartes
- ▶ Imaginez si plus d'un exemple de la classe de préférences existait.
 - ▶ Il sera difficile de choisir les préférences à choisir et à appliquer
 - ▶ avoir un autre objet de préférences pourrait conduire à des conflits ou des incohérences.
- ▶ **Un autre objectif du modèle de conception Singleton est que l'objet unique soit st accessible à l'échelle globale dans le cadre du programme**



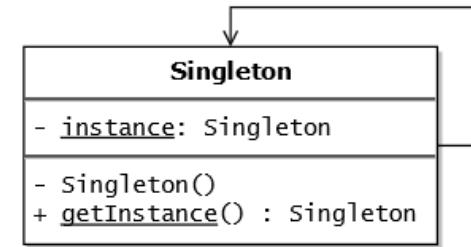
Implémentation du DP Singleton

- ▶ Si vous travaillez sur votre propre projet, vous pouvez instancier une classe, et faire une note mentale pour vous-même afin ne pas créer une autre instance
 - ▶ et le problème est résolu
 - ▶ Si vous oubliez et créez une autre instance, alors vous pourriez vous causer des problèmes à l'exécution du code
- ▶ Donc, ce n'es pas une solution réalisable dans les grands projets, ou des projets demandant la collaboration de plusieurs développeurs.
- ▶ La solution est de construire **cette seule et unique instance unique** dans la classe elle-même,
 - ▶ Afin que la création d'une autre instance d'une classe Singleton soit impossible.
- ▶ Comment faire cela ?
 - ▶ Si vous avez une classe avec un constructeur public,
 - ▶ Vous ne pouvez **PAS** empêcher le logiciel de créer beaucoup d'autres objets instances de cette classe.
- ▶ **Au lieu de cela, donnez à la classe un constructeur privé de sorte qu'il ne peut pas être appelé en dehors de la classe elle-même**

```
public class NotSingleton {  
    public NotSingleton() {  
        ...  
    }  
}
```

Implémentation du DP Singleton (suite)

- ▶ Vous donnez à la classe un constructeur privé,
 - ▶ Ce constructeur ne peut pas être appelé de l'extérieur de la classe.
- ▶ En apparence, cela donne un impression de contradiction:
 - ▶ Le constructeur est privé, il ne peut donc pas être appelé en dehors de sa classe!
 - ▶ Alors, comment créer un objet de cette classe sans constructeur public? A quoi ça sert alors?!



Il y a deux éléments clés pour contourner cela:

1. La première est de déclarer une variable privée de la classe appelée, par exemple, **uniqueInstance**.
 - ▶ de sorte qu'elle ne peut être modifiée qu'au sein de la classe
 - ▶ Elle permettra de contenir la seule instance de votre classe Singleton.

```
public class ExampleSingleton { // lazy construction
    // the class variable is null if no instance is
    // instantiated
    private static ExampleSingleton uniqueInstance = null;

    private ExampleSingleton() {
        ...
    }
```

Implémentation du DP Singleton (suite)

2. La deuxième étape consiste à créer une **méthode publique** dans la classe

Elle créera une instance de cette classe, mais seulement si elle n'existe pas déjà.

- ▶ Nous pouvons appeler cette méthode `getInstance()` et elle peut être appelée en dehors de la classe et être accessible à la population entière des autres classes de notre projet
- ▶ Cette méthode vérifiera d'abord si la variable unique d'instance est nulle. S'elle l'est, alors notre méthode fera une instance de la classe et définira cette variable pour référencer l'objet instance
- ▶ Si la variable `uniqueInstance` fait actuellement référence à un objet, ce qui signifie déjà qu'un objet de cette classe existe, alors la méthode retournera simplement cet objet.

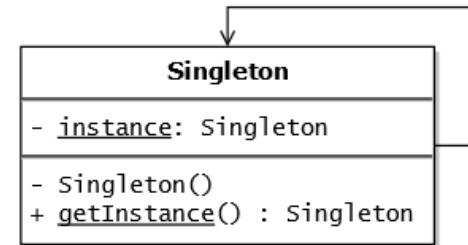
```
public class ExampleSingleton { // lazy construction
    // the class variable is null if no instance is
    // instantiated
    private static ExampleSingleton uniqueInstance = null;

    private ExampleSingleton() {
        ...
    }
```

```
// lazy construction of the instance
public static ExampleSingleton getInstance() {
    if (uniqueInstance == null) {
        uniqueInstance = new ExampleSingleton();
    }

    return uniqueInstance;
}

...
}
```



Singleton: Avantages et inconvénients

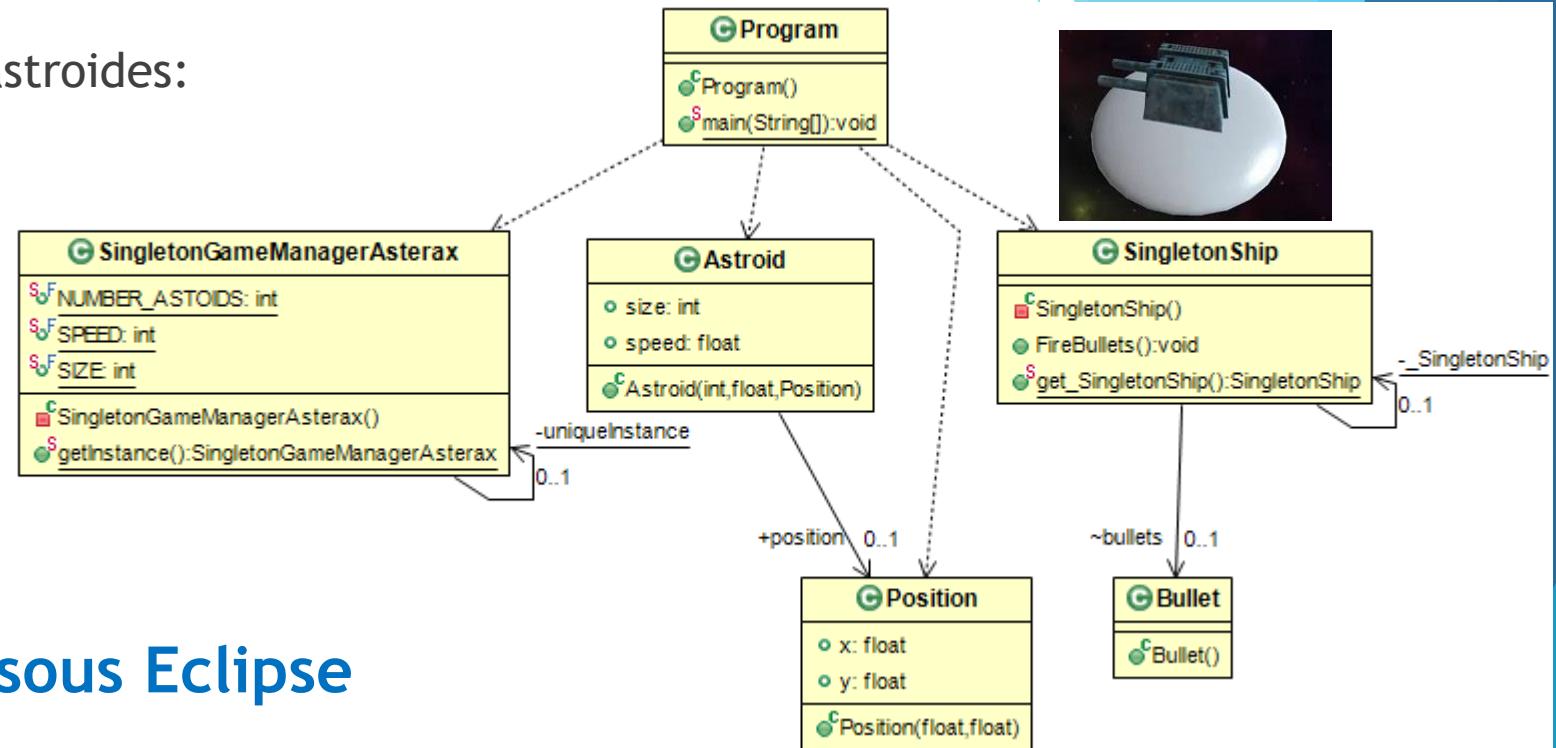
- ▶ Un avantage d'une classe Singleton est que l'objet n'est pas créé jusqu'à ce qu'il soit vraiment nécessaire.
- ▶ C'est ce qu'on appelle une création paresseuse (**lazy creation**). Et, cela peut être utile si l'objet est large.
 - ▶ Il n'est pas créé tant que la méthode `getInstance()` n'est pas appelée, ce qui est plus efficace.
- ▶ Un contre-compromis à l'utilisation du modèle de conception Singleton est:
 - ▶ S'il y a plusieurs threads parallèles en cours d'exécution, il peut y avoir des problèmes causés par les threads essayant d'accéder à l'objet unique partagé.
- ▶ Pour décider si approprié d'utiliser le Singleton DP ou non, il est utile d'avoir une bonne idée de la façon dont il fonctionne
- ▶ Ainsi, vous pouvez voir des variations de la façon dont le DP Singleton est réalisé,
 - ▶ Mais leur objectif est toujours le même : Fournir un accès global à une classe limitée à une seule instance.

En général, cela se fait en ayant un constructeur privé avec une méthode publique qui instanciera la classe, si elle n'est pas déjà instantanée.

Singleton: Exemple de démonstration d'usage

Asterax est un jeu de destruction d'astroides:

- ▶ Il y a un unique GameManager,
- ▶ une seule SpaceShip, reforçable,
- ▶ 3 Astroides divisables, prédefinis,
- ▶ des cartouches (bullets) variées,
- ▶ ...



Voir Exemple du Singleton sous Eclipse



Game Over

Final Level:
Final Score: 800

Complexité grandissante → Design Patterns 😊

AsteraX - Components and Classes Diagram

by Jeremy Gibson Bond

Notes:

- A solid black line denotes a parent/child relationship in the Unity Hierarchy pane.
- A solid line with a semi-circle on the end represents one GameObject instantiating another GameObjects as needed.
- A light purple arrow denotes a class pulling info from or utilizing another class.
- Boxes stacked together or touching each other are multiple Components on the same GameObject.
- The AsteraX script has relationships with so many classes that they are denoted with these lines. The open circle is data going from AsteraX, & the closed circle is data going to AsteraX.

Common Components:

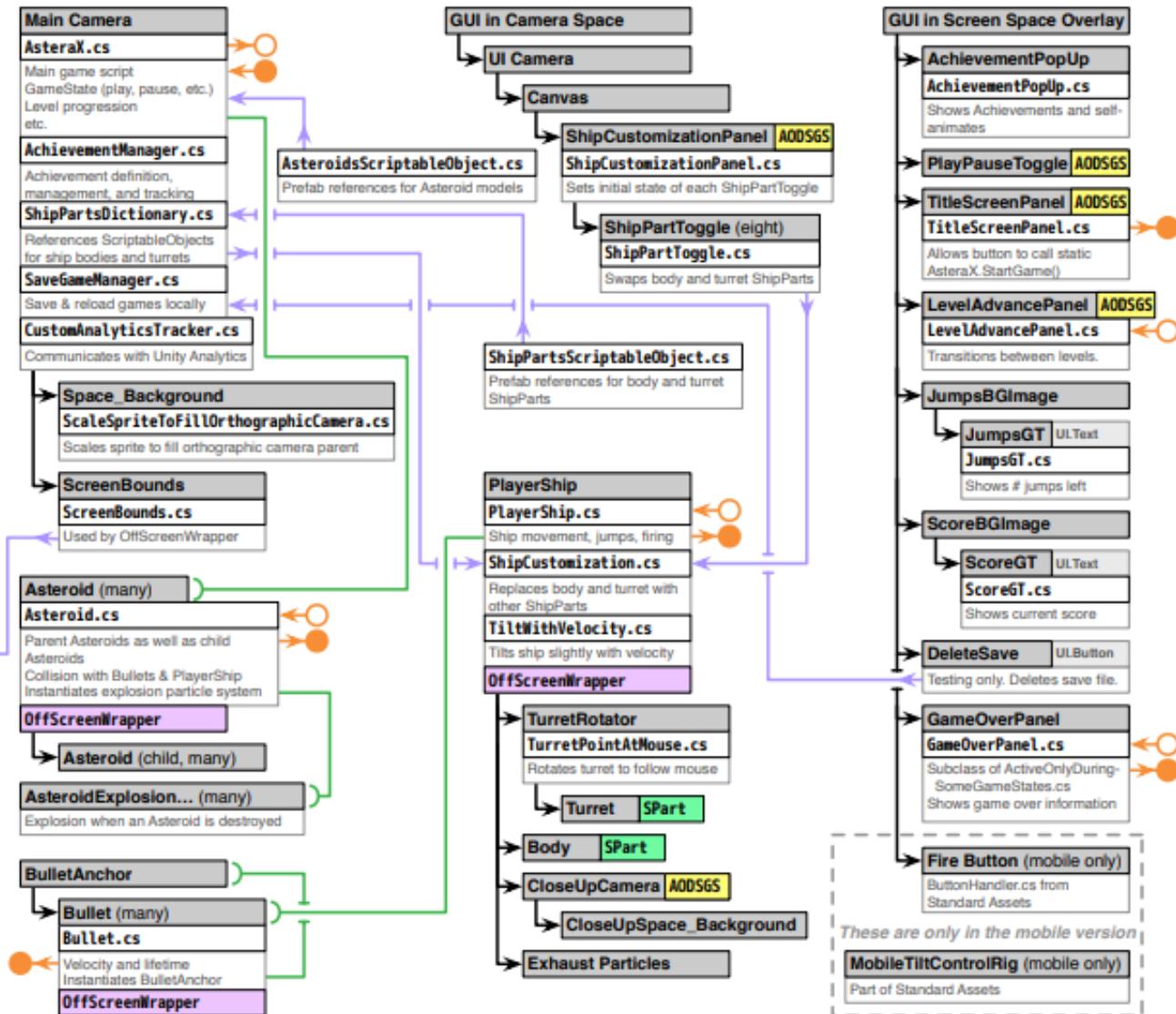
These are attached to several different GameObjects

AOOSGS	ActiveOnlyDuringSomeGameStates.cs
	Sets GameObject.active based on AsteraX.GAME_STATE.
SPart	ShipPart.cs
	Script for either a body or turret customizable part
OSW	OffScreenWrapper.cs
	Wraps an object around the screen when it exits ScreenBounds

Additional Scripts:

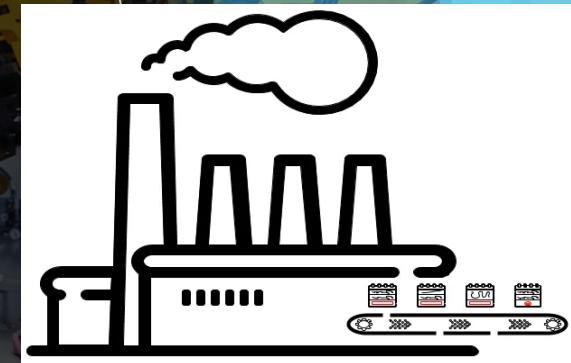
These scripts don't have a place in the diagram. I do not expect you to have these in your diagram.

Vector3Extensions.cs
An extension to the Vector3 class that implements a method named ComponentDivide() that divides each component of one Vector3 by its counterpart in another Vector3.
EnumFlagsAttributePropertyDrawer.cs
A script posted to the Unity forum by Aqbsadiq that allows any enum with the System.Flags attribute to be edited in the Unity Inspector using MaskField editor (the same pop-up with checkboxes editor that is often shown for Physics Layers).



III.2- Les DPs “Factory/Factory method” & “Abstract Factory”

Usine d'objets/Méthode d'usine et
Usine Abstraite





DP “Factory (usine d'objets)”

- ▶ Notre objectif est le modèle de « usine abstraite» (Abstract Factory). Mais tout d'abord, nous allons voir le DP « méthode d'usine » (Factory-Method) ou **Factory** tout court.
- ▶ Tout comme une usine dans le monde réel, le but de ces usines en POO est de créer des objets.
- ▶ Utiliser des « usines », rend votre logiciel plus facile à entretenir et à mettre à jour puisque la création d'objets se produit dans des usines qui peuvent être spécialisées!
- ▶ Considérons que vous avez une solution logicielle qui implémente une boutique en ligne:
 - ▶ Vous voulez créer des objets à vendre: Un magasin de **couteaux** par exemple.
 - ▶ Il existe de nombreux types de couteaux,
 - ▶ couteaux à steak
 - ▶ couteaux de chefs, etc
- ▶ Vous avez une superclasse, **couteau**, avec sous-classes, **SteakKnife** et **ChefsKnife**.
 - ▶ Vous écrivez une méthode pour commander un couteau,
 - ▶ créera d'abord un de ces objets de type couteau, puis le préparera à l'expédition.





DP “usine d’objets” (problematic exp.)

- ▶ Disons que les préparatifs à effectuer sur un couteau sont:
 - ▶ Aiguiser (*sharpen*), polir (*polish*) et emballer (*package*).
- ▶ Voici le code de déclaration requis (STEP1):
 - ▶ La méthode déclare d’abord une variable `knife`, qui se référera à l’objet couteau (`Knife`) à créer.
 - ▶ Les conditions déterminent quelle sous-classe de couteau est effectivement instanciée.
 - ▶ L’acte d’instanciation d’une classe pour créer un objet d’un type spécifique est appelé instantiation concrète (**concrete instantiation**).
 - ▶ L’instanciation concrète est faite avec l’opérateur `new`.

L’instanciation concrète est en fait l’objectif principal de nos DP usines

- ▶ (STEP2) Appelez quelques méthodes qui sont communes aux différents types d’objets de couteau : aiguiser, polir, et emballer.
 - ▶ Ces méthodes ne se soucient pas de quel type de couteau il s’agit.
 - ▶ Tout ce qu’ils veulent, c’est un couteau (`knife`) pour travailler dessus

```
Knife orderKnife(String knifeType) {  
    Knife knife;  
    // create Knife object - concrete instantiation  
    if (knifeType.equals("steak")) {  
        knife = new SteakKnife();  
    } else if (knifeType.equals("chefs")) {  
        knife = new ChefsKnife();  
    }  
  
    // prepare the Knife  
    knife.sharpen();  
    knife.polish();  
    knife.package();  
    return knife;  
}
```



DP “usine d’objets” (problematic exp.)

- ▶ Imaginez maintenant que votre magasin ajoute de plus en plus de types de couteaux à mesure que vos ventes s’améliorent.
 - ▶ De nouvelles sous-classes sont ajoutées au besoin:
 - ▶ Couteau à pain,
 - ▶ Couteau d’épluchage, etc.
- ▶ Dans cet exemple, la liste des conditionnels s’allonge et s’allonge à mesure que de nouveaux types de couteaux sont ajoutés.
- ▶ Notez que ce que nous faisons avec le couteau après sa création ne change pas.
 - ▶ Tous ces couteaux doivent être aiguisés, polis et emballés.
 - ▶ Tout cela devient assez compliqué.

[Voir Exemple de l’usine \(à l’ancienne!\) sous Eclipse](#)

```
Knife orderKnife(String knifeType) {  
    Knife knife;  
  
    // create Knife object - concrete instantiation  
    if (knifeType.equals("steak")) {  
        knife = new SteakKnife();  
    } else if (knifeType.equals("chefs")) {  
        knife = new ChefsKnife();  
    } else if (knifeType.equals("bread")) {  
        knife = new BreadKnife();  
    } else if (knifeType.equals("paring")) {  
        knife = new ParingKnife();  
    }  
  
    // prepare the Knife  
    knife.sharpen();  
    knife.polish();  
    knife.package();  
  
    return knife;  
}
```



DP “usine d’objets” (solution)

- ▶ Tout comme dans le monde réel, les objets sont fabriqués dans des **usines**.
 - ▶ Nous pouvons créer une entité d’usine dont le rôle est de créer des objets de produits particuliers.
- ▶ Ainsi, le sharpening, le polissage et l’emballage resteront là où il sont dans la méthode **orderKnife**, mais on **délèguera** la responsabilité de la création d’objets de produit à un autre objet: une usine de couteaux.
 - ▶ Nous allons déplacer le code pour décider quel type de couteau créer, et le code pour décider quelle sous-classe de couteau à instancier dans la classe usine.
- ▶ La classe pour un **KnifeFactory** est montrée ici avec une méthode **createKnife()**.
 - ▶ Presque exactement le même code que l’exemple précédent.
 - ▶ Sauf que ce même code vient d’être déplacé dans une méthode d’une nouvelle classe appelée le **KnifeFactory**

```
public class KnifeFactory {  
    public Knife createKnife(String knifeType) {  
        Knife knife = null;  
  
        // create Knife object  
        if (knifeType.equals("steak")) {  
            knife = new SteakKnife();  
        } else if (knifeType.equals("chefs")) {  
            knife = new ChefsKnife();  
        }  
  
        return knife;  
    }  
}
```



DP “usine d’objets” (solution (suite))

- ▶ Maintenant, cette `knifeFactory` peut être utilisée par la classe `knifeStore`
- ▶ Donc, le `knifeStore` est maintenant un **client** de `knifeFactory`. Exemple:
 - ▶ l’objet `knifeFactory` à utiliser est passé dans le constructeur pour la classe `KnifeStore`
 - ▶ La méthode `orderKnife` est très similaire à celle précédente.
 - ▶ Cependant, au lieu d’effectuer l’instanciation concrète elle-même, elle **délègue** cette tâche à l’objet usine.
- ▶ Gains:
 - ▶ `knifeStore` et sa méthode `orderKnife` peuvent ne pas être le seul client de notre `knifeFactory`.
 - ▶ D’autres clients peuvent utiliser `knifeFactory` pour créer des couteaux à d’autres fins.
 - ▶ Au lieu de traquer plusieurs extraits de codes d’instanciations similaires, nous pouvons tout simplement ajouter ou supprimer les sous-classes particulières à instancier dans notre classe usine.

```
public class KnifeStore {  
    private KnifeFactory factory;  
    // require a KnifeFactory object to be passed to  
    // this constructor:  
    public KnifeStore(KnifeFactory factory) {  
        this.factory = factory;  
    }  
    public Knife orderKnife(String knifeType) {  
        Knife knife;  
        // use the create method in the factory  
        knife = factory.createKnife(knifeType);  
        // prepare the Knife  
        knife.sharpen();  
        knife.polish();  
        knife.package();  
        return knife;  
    }  
}
```

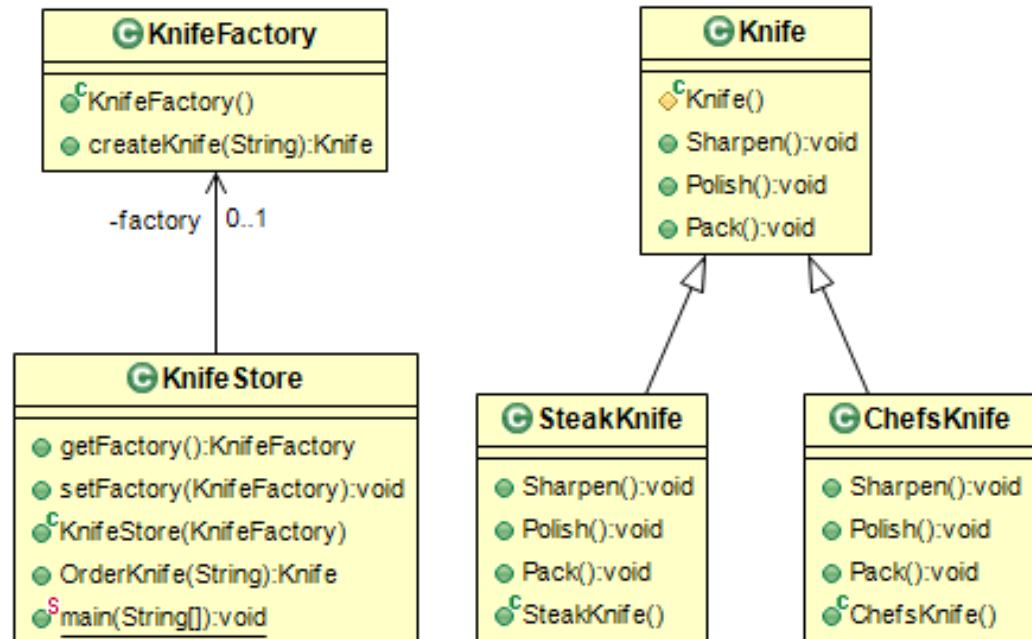
DP “Factory method”: Quand y penser

- ▶ Si il y a plusieurs clients qui veulent instancier le même ensemble de classes,
- ▶ alors utiliser un objet de type Usine,
 - ▶ vous isolez ainsi le code redondant et rendez votre code nettement plus facile à modifier.

Voir Exemple du Factory Method sous Eclipse

ASTUCE: Sous Eclipse JAVA IDE, rajouter le plugin ObjectAid:

- ▶ Help->Install New Software
- ▶ Name: ObjectAid UML Explorer
- ▶ Location: <http://www.objectaid.com/update/current>





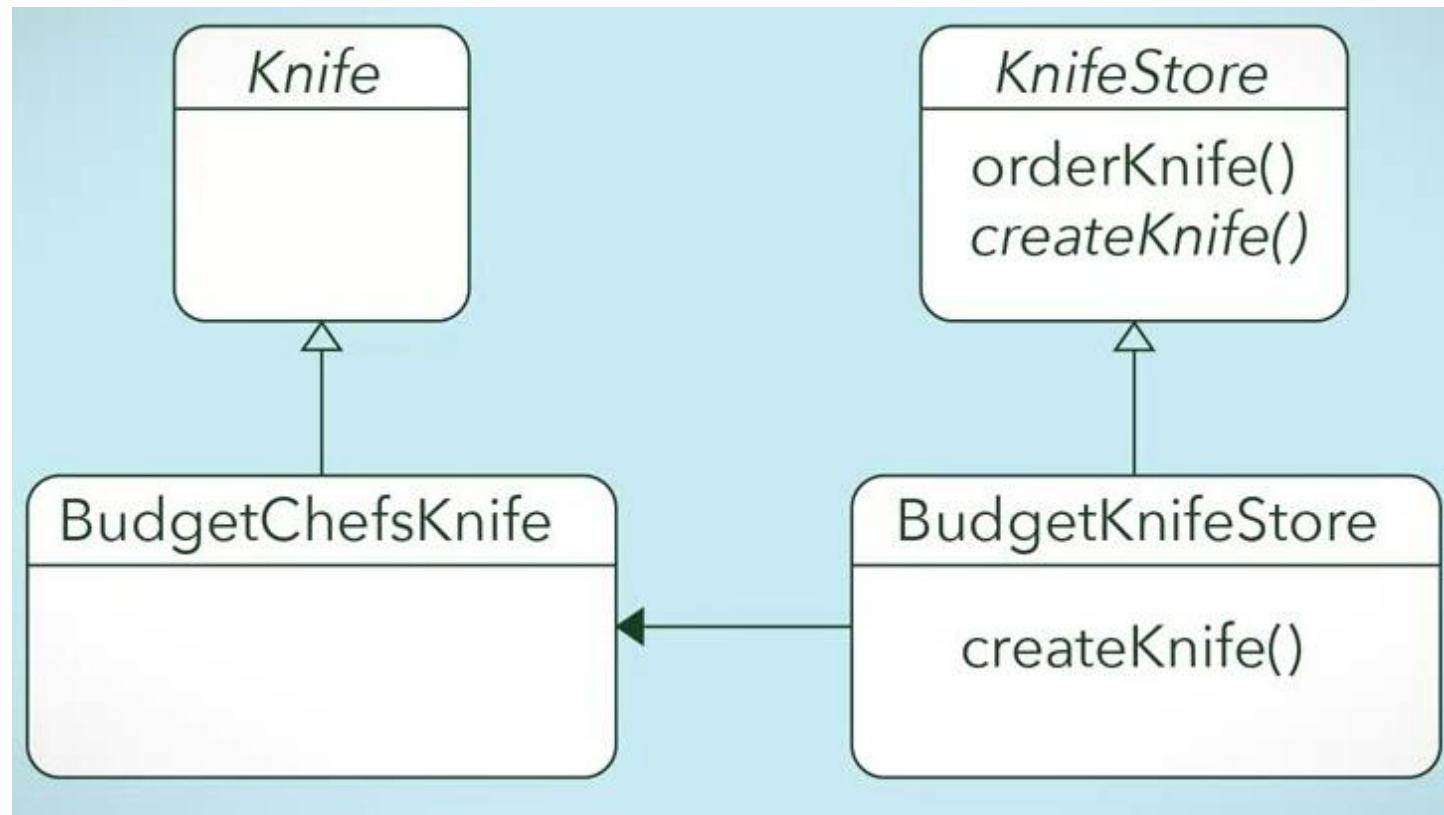
Extension: DP “Abstract Factory”:

- ▶ Un DP créational qui utilise des méthodes d'usine pour:
 - ▶ Traiter le problème de la création d'objets sans avoir à spécifier leur classe exacte
 - ▶ On pourrait ne pas être sûr de tous les types et détails que nous allons avoir dans notre classe
- ▶ Réalisé en créant des objets qui appellent une méthode d'usine qui pourrait être l'un de ces 2 cas:
 1. spécifiée dans une **interface** et **implémentée** par des classes filles,
 2. implémentée dans une **classe de base** et optionnellement redéfinie (**override**) par les classes dérivées
- ▶ Fournit des objets plus faciles à implémenter, à modifier, à tester et à réutiliser
 - ▶ utilisé au lieu du constructeur ordinaire de la classe
 - ▶ découpe la construction d'objets des objets eux-même
- ▶ Avantages:
 - ▶ Permet la construction de classes avec un composant d'un type qui n'a pas été prédéterminé, mais seulement défini dans une « interface » ou une « superclasse »



Extension: DP “Abstract Factory”: Schéma UML

- ▶ Exemple pour les couteaux (Knives):
 - ▶ Knife est une **abstraction du produit**
 - ▶ BudgetChefsKnife est notre **produit concret**
- ▶ KnifeStore est notre **abstract Factory**
- ▶ BudgetKnifeStore est la realization de notre factory par un **client concret**

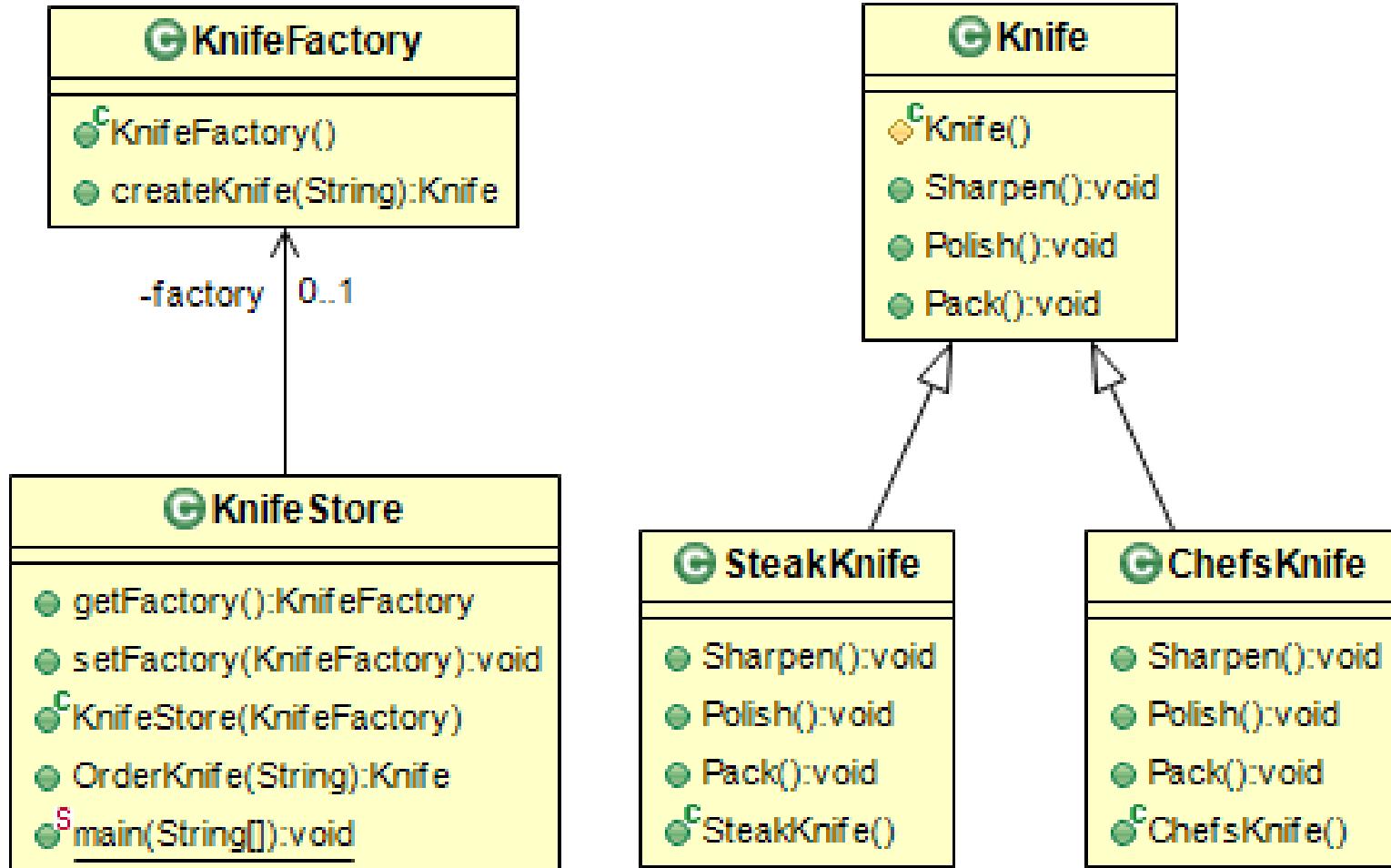




DP “Abstract Factory”: Exemple de code source

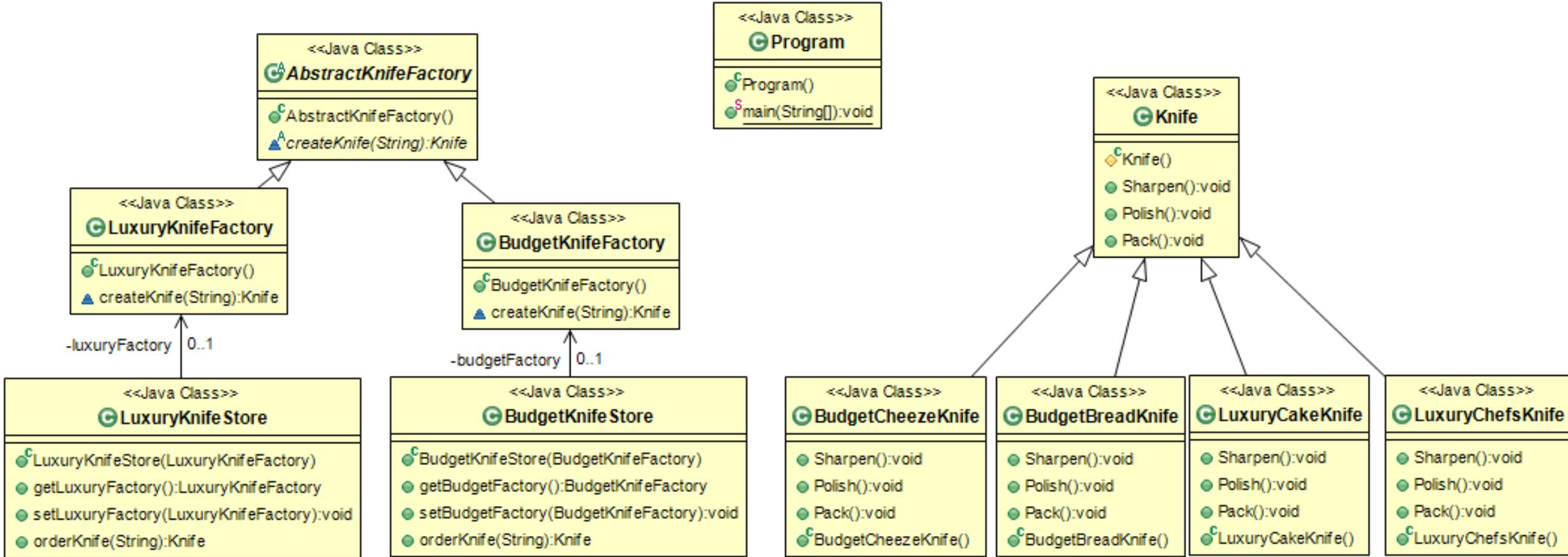
```
public abstract class KnifeStore {  
    public Knife orderKnife(String knifeType) {  
  
        Knife knife;  
        // now creating a knife is a method in  
        // the class  
        knife = createKnife(knifeType);  
        knife.sharpen();  
        knife.polish();  
        knife.package();  
  
        return knife;  
    }  
    abstract Knife createKnife (String type);  
}
```

```
public BudgetKnifeStore extends KnifeStore {  
    //up to any subclass of KnifeStore to define  
    //this method  
    Knife createKnife(String knifeTYpe) {  
  
        if (knifeType.equals("steak")) {  
            return new BudgetSteakKnife();  
        } else if (knifeType.equals("chefs")) {  
            return new BudgetChefsKnife();  
        }  
        //.. more types  
        else return null;  
    }  
}
```



Exemple N°2 complet à tester sous eclipse

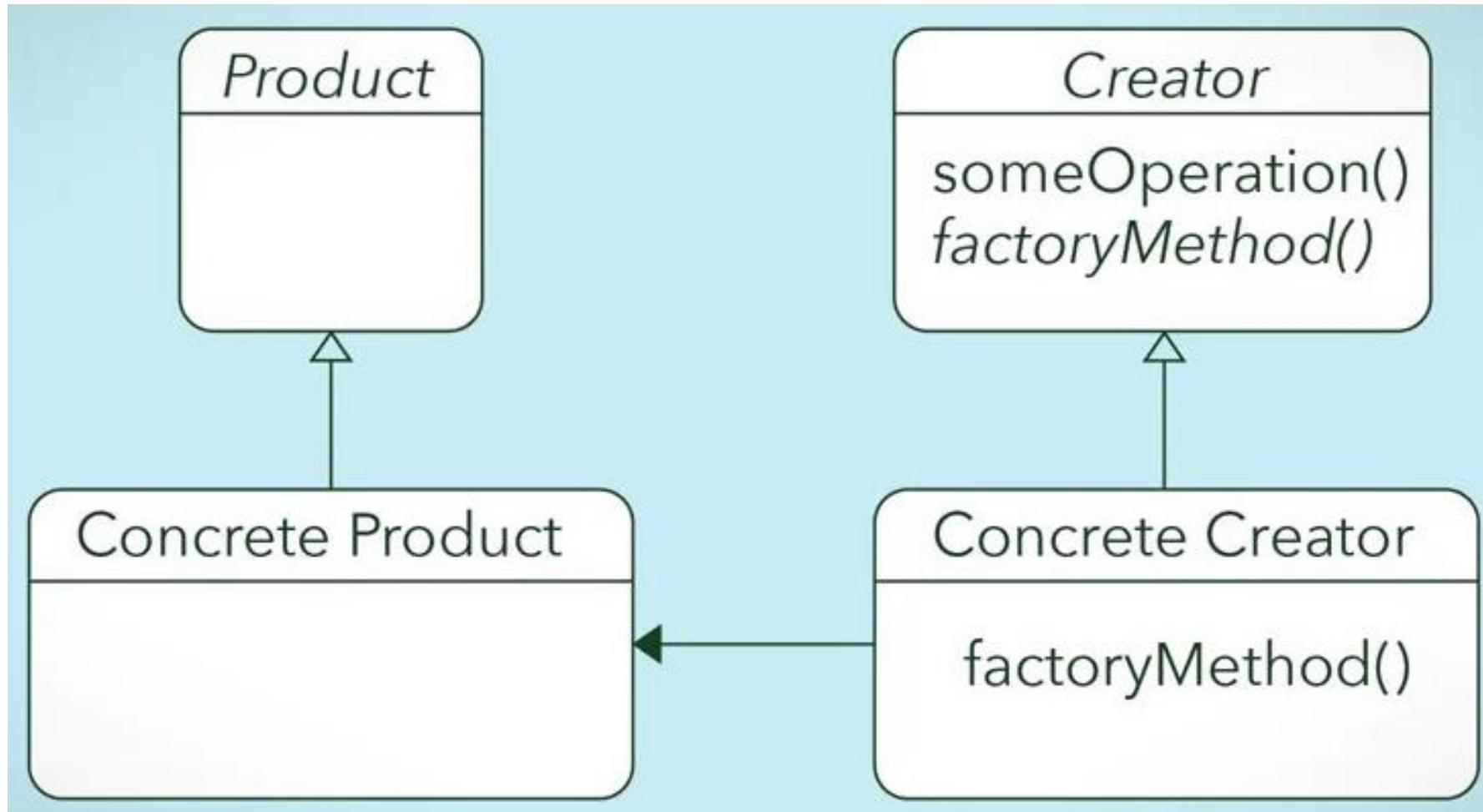
Voir Exemple du Abstract Factory sous Eclipse





Généralisation: DP “Abstract Factory”: UML

- ▶ Forme générale



IV- Patterns de structuration

(Structural Patterns)

1. Façade
2. Adapter
3. Composite
4. Decorator
5. Proxy

Glossaire des Design Patterns (suite)

Les patterns de Structuration (Structural patterns):

C'est des modèles de conception qui décrivent comment les objets sont connectés les uns aux autres.

Pensez-y comme un plat alimentaire qui est composé de différents ingrédients et saveurs:

- ▶ certains ingrédients disparaissent dans le goût global des aliments
 - ▶ d'autres sont toujours là et se distinguent sur la table
-
- ▶ **Décorateur** - (**Decorator**) un DP permettant un comportement supplémentaire grâce à une enveloppe à base de composants.
 - ▶ **Façade** - un DP qui consiste à encapsuler un système complexe dans une interface simple.
 - ▶ **Adaptateur (Adapter)** - un DP qui relie et adapte deux systèmes incompatibles en fournissant une interface compatible avec les deux.
 - ▶ **Proxy** - tel un proxy sous réseaux, ce DP offre une enveloppe de sécurité et simplification d'accès pour les classes clientes
 - ▶ **Composition (Composite)** - un modèle de conception pour composer des structures imbriquées d'objets et traiter ces objets uniformément.

Association

Aggregation

Composition

Inheritance

Interface



IV.1- Le Design Pattern Façade

- ▶ À mesure que les systèmes (ou leurs parties) grandissent, ils deviennent également plus complexes.
- ▶ Pas forcément une mauvaise chose!
 - ▶ Si un problème est important, il peut nécessiter une solution complexe.
 - ▶ Les classes de clients, cependant, fonctionnent mieux avec une interaction plus simple.



- ▶ Le modèle de conception de **Façade** tente de résoudre ce problème, en fournissant une interface unique et simplifiée pour que les classes client interagissent avec un sous-système. Il en est ainsi un DP **structurel**.

IV.1- Le Design Pattern Façade

- ▶ Dans le monde réel, une façade pour un magasin est ce qui attire les clients et leur donne des conseils sur les services qui les attendent dans un endroit donné.
 - ▶ Imaginez que vous marchez dans la rue et à la recherche d'un endroit pour avoir le repas. Que cherchez-vous ?
 - ▶ Naturellement, un panneau sur la face d'un bâtiment qui indique que cet espace peut vous servir de repas.
 - ▶ Le même concept s'applique aux sites d'achats en ligne, par exemple,:
 - ▶ Nous sommes à la recherche de points d'entrée dans l'interface du site: L'endroit où commencer nos achats (la devant du magasin)
 - ▶ **(i) Quand un serveur prend et passe votre commande,**
 - ▶ **ou (ii) quand une plate-forme de vente en ligne envoie votre commande pour être traitée, ils agissent tel une façade en cachant tout le travail supplémentaire à faire.**



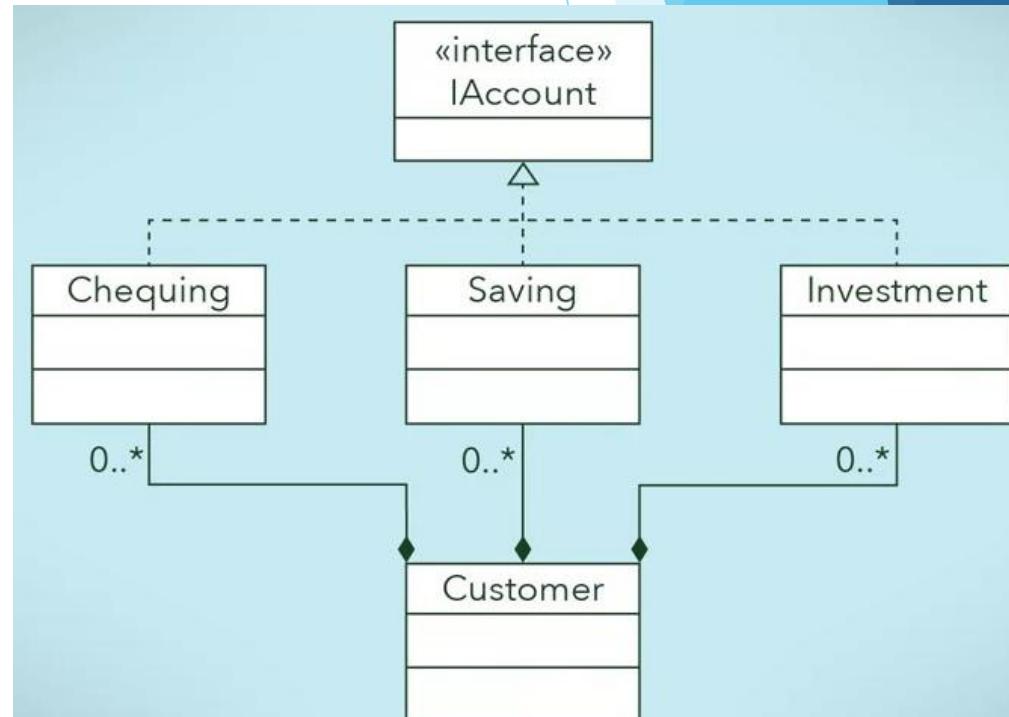
Quiz 1 sur le DP Façade

Quelles sont les deux conditions requises pour utiliser le DP façade?

- A. Vous avez besoin d'une classe pour agir comme interface entre votre sous-système et une classe client.
- B. Vous devez simplifier l'interaction avec votre sous-système pour les classes clients.
- C. Vous avez besoin d'une classe qui pourra instancier d'autres classes au sein de votre système et fournira ces instances à une classe de clients.
- D. Vous avez besoin d'une classe pour traduire des messages entre deux sous-systèmes existants parce que l'un d'entre eux s'attend à une interface spécifique à utiliser, mais est fourni avec une interface qui est non compatible.

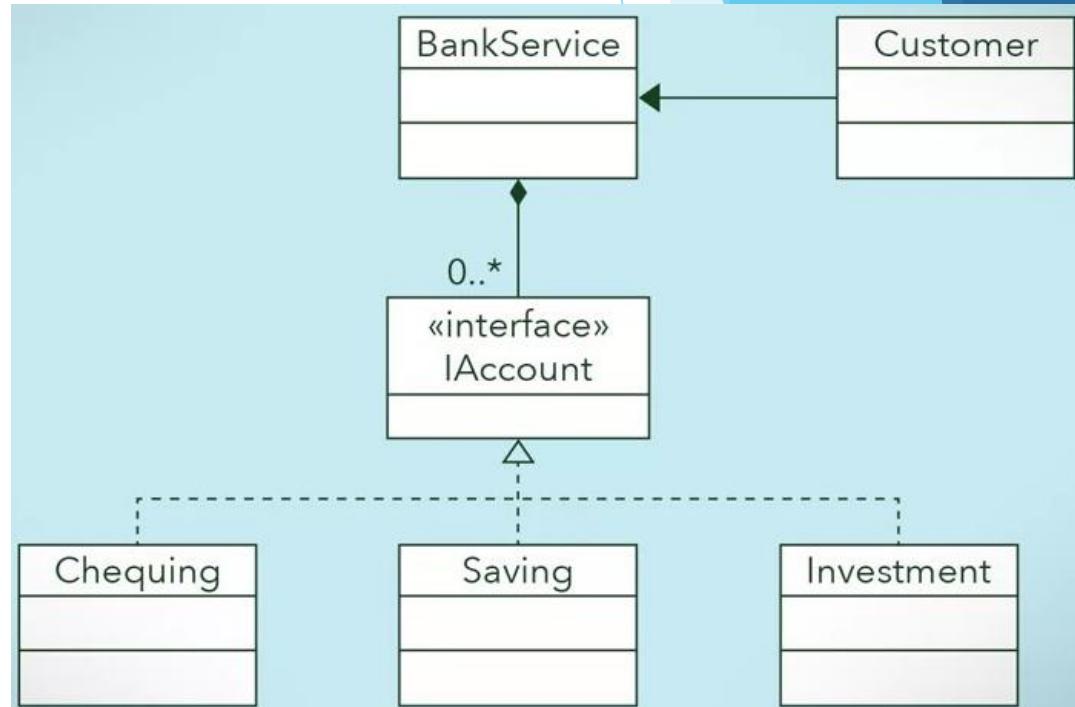
Si vous ne faites pas appel à une Façade! 😞

- ▶ Sous logiciel, le DP Façade fait exactement ce qu'un serveur ou vendeur ferait dans la vraie vie.
- ▶ Une façade est une classe d'emballage qui enveloppe le sous-système afin de cacher sa complexité.
 - ▶ Cette classe permettra à une classe client d'interagir avec le sous-système à travers une façade.
- ▶ Considérons **un système bancaire simple**:
- ▶ Sans façade, la classe **Client** contiendrait des instances des classes de Chèques (**Chequing**), d'Epargnes (**Saving**) et d'Investissements (**Investment**).
- ▶ Cela signifie que le client est responsable d'instancier chacune de ces classes constituantes et connaît tous ses différents attributs et méthodes.
- ▶ C'est comme gérer tous vos comptes financiers dans la vie réelle, (ce qui pourrait être très complexe avec beaucoup de comptes), au lieu de laisser une institution financière le faire pour vous.



Obtenir l'aide d'une Façade ☺

- ▶ Nous introduisons la classe de service bancaire pour agir comme une façade:
 - ▶ Pour les classes **Checking**, **Saving** et **Investment**.
- ▶ La classe client n'a plus besoin de gérer l'instanciation ou de faire face à l'une des complexités de la gestion financière.
- ▶ Les trois comptes différents implémentent l'interface **IAccount**,
 - ▶ la classe **BankService** enveloppe (*wrapper*) efficacement les classes interfaçant avec le compte,
 - ▶ présentant un front plus simple à utiliser pour la classe cliente (*Custumer*).
- ▶ Le modèle de conception de façade est simple à appliquer dans notre exemple de comptes bancaires.
 1. Il combine l'implémentation de l'interface par une ou plusieurs classes,
 2. Puis le tout est enveloppés par la classe façade.



Mise en œuvre de la façade en 4 étapes

Step 1: Design de l'Interface

- ▶ Créer une interface qui sera implémentée par les différentes classes de compte,
- ▶ Elle ne sera pas connue de la classe Client.

```
public interface IAccount {  
    public void deposit(BigDecimal amount);  
    public void withdraw(BigDecimal amount);  
    public void transfer(BigDecimal amount);  
    public int getAccountNumber();  
}
```

Step 2: Implémenter l'interface avec une ou plusieurs classes:

- ▶ Implémenter l'interface avec les classes qui seront enveloppées avec la classe façade,
- ▶ Notez que dans cet exemple simple, une seule interface est implémentée et cachée, mais dans la pratique, une classe de façade peut être utilisée pour envelopper toutes les interfaces et classes pour un sous-système.

```
public class Chequing implements IAccount { ... }  
public class Saving implements IAccount { ... }  
public class Investment implements IAccount { ... }
```

/!\

- ▶ Rappelez-vous que les interfaces permettent la création de sous-types
- ▶ Ceci vient dire ici que, *Chequing*, *Saving*, et *Investment* sont des sous-types de *IAccount*, et donc sont attendus à se comporter comme des types de *account*.

Mise en œuvre de la façade en 4 étapes (suite)

Step 3: Créer la classe façade class et envelopper les classes qui implémentent l'interface

- ▶ La classe `BankService` est la façade. Ses méthodes publiques sont simples à utiliser et ne montrent aucune indication de l'interface sous-jacente et de la mise en œuvre des classes.
- ▶ Le principe de *information hiding* est utilisé ici pour empêcher les classes de clients de « voir » les objets de compte, et comment ces comptes se comportent,
- ▶ Noter que les accès sont privé (`private`) pour chaque référence à un `Account`.

```
public class BankService {  
    private Hashtable<int, IAccount> bankAccounts;  
    public BankService() {  
        this.bankAccounts = new Hashtable<int, IAccount>;  
    }  
    public int createNewAccount(String type, BigDecimal initAmount) {  
        IAccount newAccount = null;  
        switch (type) {  
            case "chequing":  
                newAccount = new Chequing(initAmount);  
                break;  
            case "saving":  
                newAccount = new Saving(initAmount);  
                break;  
            case "investment":  
                newAccount = new Investment(initAmount);  
                break;  
            default:  
                System.out.println("Unknown account type");  
                break;  
        }  
        if (newAccount != null) {  
            this.bankAccounts.put(newAccount.getAccountNumber(),  
                newAccount);  
            return newAccount.getAccountNumber();  
        }  
        return -1;  
    }  
    public void transferMoney(int to, int from, BigDecimal amount) {  
        IAccount toAccount = this.bankAccounts.get(to);  
        IAccount fromAccount = this.bankAccounts.get(from);  
        fromAccount.transfer(toAccount, amount);  
    }  
}
```

Mise en œuvre de la façade en 4 étapes (suite)

Step 4: Utilisez la classe façade pour accéder au sous-système:

- ▶ Avec la classe façade en place, la classe client peut accéder aux comptes par les méthodes de la classe BankService.
- ▶ La classe BankService indiquera au client quel type d'actions il pourra appeler, puis elle pourra déléguer (delegate) cette action à l'objet Account bien approprié.

```
public class Customer {  
  
    public static void main (String args []) {  
        BankService myBankService = new BankService();  
  
        int mySaving = myBankService.createNewAccount("saving", new BigDecimal(500.00));  
        int myInvestment = myBankService.createNewAccount("investment", new BigDecimal(1000.00));  
  
        myBankService.transferMoney(mySaving, myInvestment, new BigDecimal(300.00));  
    }  
  
}
```

Quiz 2 sur le DP Façade

Quels sont les principes de conception clés utilisés pour mettre en œuvre le DP façade?

- A. Encapsulation, implémentation d'interfaces, dissimulation d'informations, héritage
- B. Héritage, dissimulation d'informations, séparation des préoccupations, généralisation
- C. Encapsulation, dissimulation d'informations, séparation des préoccupations
- D. Dissimulation d'informations, encapsulation, généralisation, couplage

Points à retenir à propos du DP Façade

Tout est une question de client!

- ▶ Il est destiné à **cacher la complexité** d'un sous-système en l'encapsulant derrière un wrapper unificateur appelé classe façade.
- ▶ Élimine la nécessité pour les classes clients de gérer elles-mêmes un sous-système, ce qui entraîne **moins de couplage** entre le sous-système et les classes clients.
- ▶ Gère **l'instantanéation et la réorientation** des tâches vers la classe appropriée au sein du sous-système.
- ▶ Fournit aux classes client une **interface simplifiée** pour le sous-système.
- ▶ Agit simplement comme un **point d'entrée** à un sous-système et n'ajoute pas plus de fonctionnalités au sous-système.

En fournissant une interface plus simple à travers une façade, les classes clients auront plus de facilité à utiliser avec succès notre logiciel.



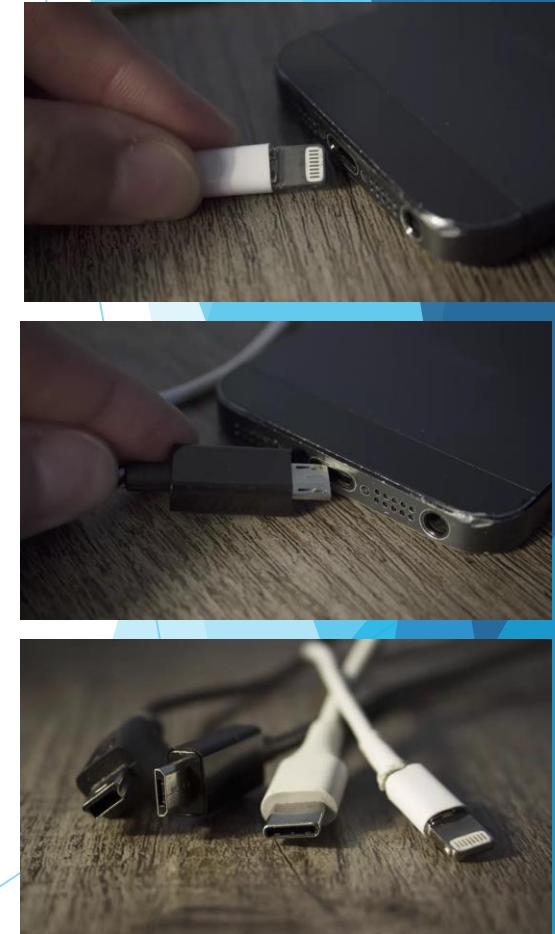
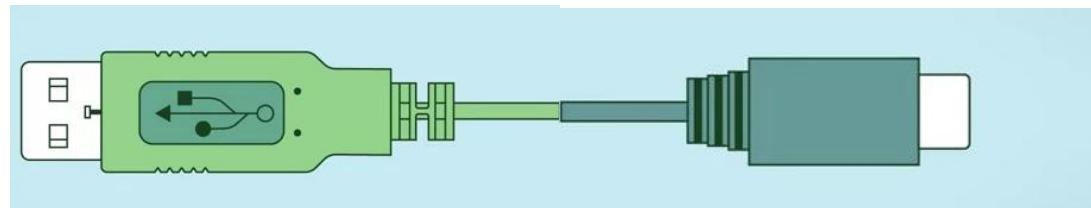
IV.2- Le DP Adaptateur

Une situation courante que vous avez peut-être:

- ▶ Lorsque vous avez un appareil qui s'attend à une sorte de connecteur,
 - ▶ mais vous avez quelque chose qui fournit un connecteur différent.
 - ▶ En raison de l'inadéquation, vous ne pouvez pas faire ce lien.
 - ▶ Ainsi, vous aurez besoin d'un **adaptateur** pour cela,
-
- ▶ Les adaptateurs sont fréquemment utilisés parce que différents fournisseurs auront des connecteurs différents et évolutifs pour leurs appareils.
 - ▶ Par exemple, un connecteur commun pour un périphérique, comme un clavier, était autrefois PS2.
 - ▶ Mais les ordinateurs au milieu des années 1990 ont commencé à prendre en charge uniquement le port USB, alors, les gens avaient besoin d'un adaptateur pour les nouveaux ordinateurs.

Le DP Adaptateur

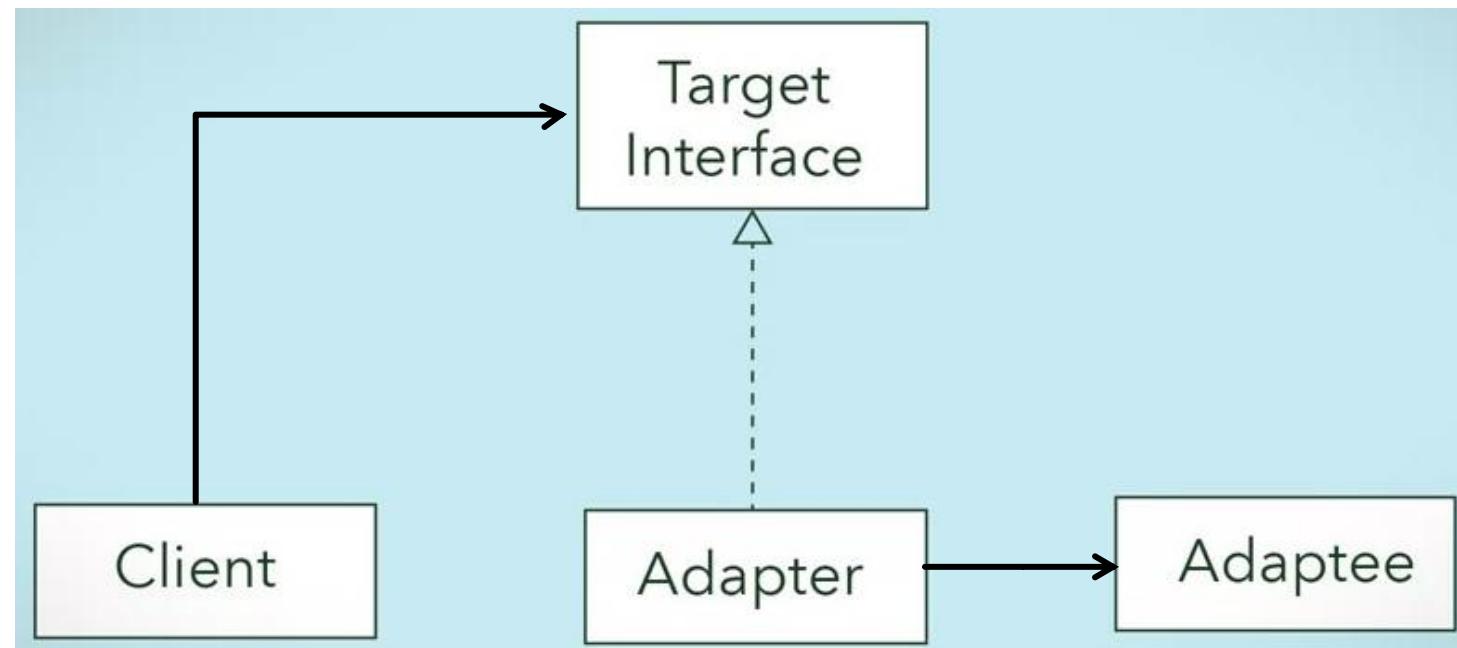
- ▶ Physiquement, un adaptateur est un dispositif qui est utilisé pour connecter des pièces d'équipement qui ne peuvent pas être connectées directement
- ▶ Les systèmes logiciels peuvent faire face à des problèmes similaires:
 - ▶ Les systèmes peuvent avoir des interfaces logicielles incompatibles
 - ▶ la sortie d'un système peut ne pas être conforme à l'apport attendu d'un autre système
- ▶ Cela se produit fréquemment lorsqu'un système préexistant doit intégrer des bibliothèques tierces ou doit se connecter à d'autres systèmes..
- ▶ Le **DP structurel d'Adaptateur** facilite la communication entre deux systèmes existants en fournissant une interface compatible



Composition du DP Adaptateur

Le modèle de conception de l'adaptateur se compose de 4 pièces différentes:

- ▶ **Une classe de clients.** Cette classe est la partie de votre système qui veut utiliser une bibliothèque tierce ou des systèmes externes.
- ▶ **Une classe Adaptee.** This class is the third-party library or external system that is to be used.
- ▶ **Une classe Adapter.** Cette classe se situe entre le client et l'adapté.
 - ▶ L'adaptateur est conforme à ce que le client s'attend à voir, en implémentant une interface cible.
 - ▶ Il traduit également la demande du client en un message que l'adapté comprendra,
 - ▶ Une fois la traduction terminée, elle enverra la demande traduite à l'adapté (Adaptee)
- ▶ **Une interface cible (ou Target).** Ceci est utilisé par le client pour envoyer une demande à l'adaptateur.



Quiz 1

Quel type de classe l'adaptateur est approprié à l'adaptateur?

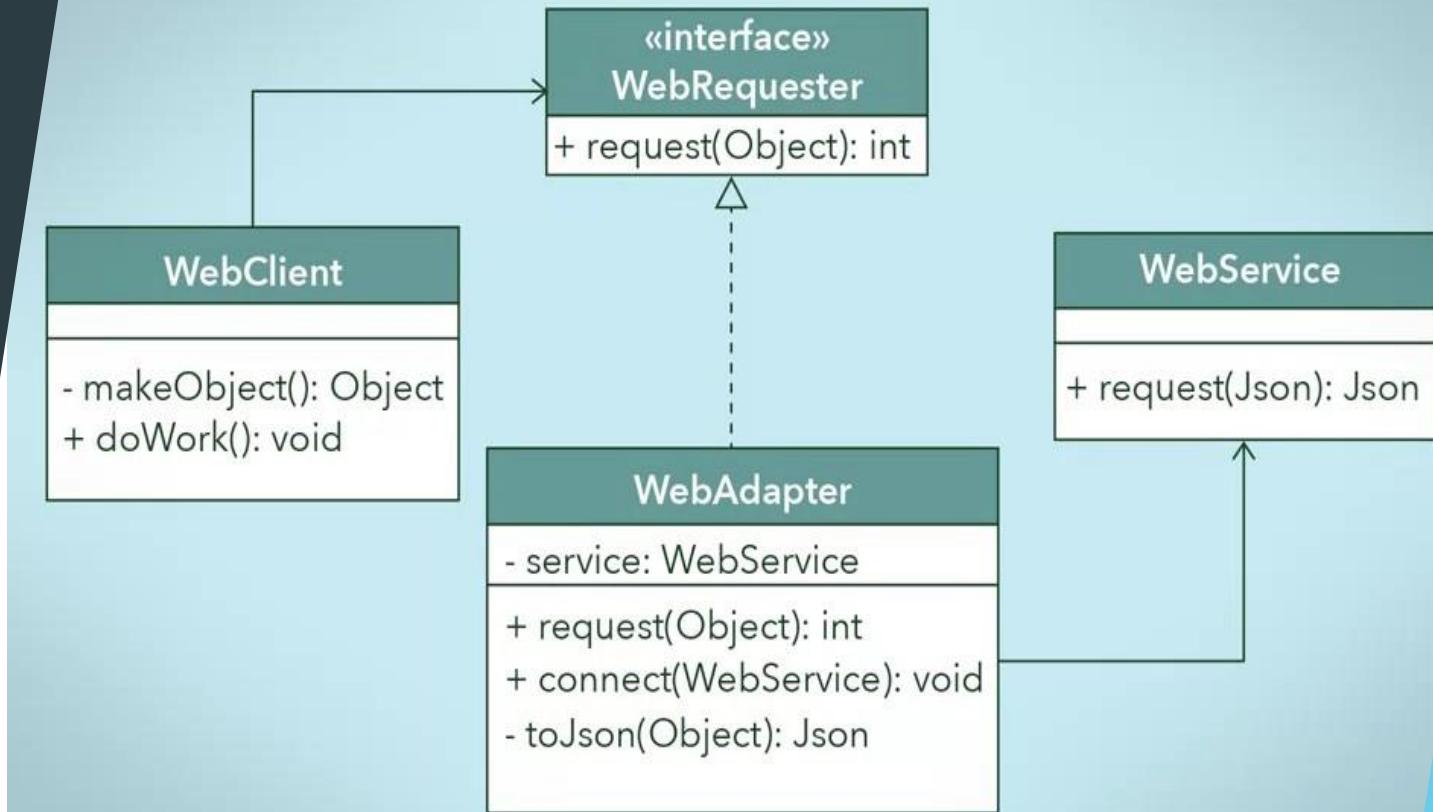
- A. Classe abstraite
- B. Superclasse
- C. Classe Enveloppe (Wrapper)
- D. Interface



Exemple de DP Adaptateur

Voici le diagramme UML pour cette situation:

- ▶ Imaginez qu'il existe un **client Web** que nous aimerais faire interagir avec un nouveau **service Web**.
- ▶ Mais que le service ne prend en charge que les objets **JSON**, et un **adaptateur** serait nécessaire pour convertir notre objet de demande en un objet **JSON**.



Exemple de DP Adaptateur (suite 2/4)

Step 1: Concevoir l'interface cible:

- ▶ Tout d'abord, créez l'interface cible que votre classe d'adaptateur implémentera pour votre classe client:

```
public interface WebRequester {  
    public int request(Object);  
}
```

Step 2: Implémenter l'interface cible avec la classe adaptateur

- ▶ La classe **Adapter** fournit les méthodes qui vont prendre l'objet de la classe **Client** et le convertir en un objet JSON.
- ▶ L'adaptateur doit convertir n'importe quelle instance d'une classe que le client peut créer et l'envoyer ensuite dans une requête.
- ▶ La classe **Adapter** transfère aussi la requête translatée vers le **Adaptee**.
- ▶ La classe **Client** a alors seulement besoin de savoir l'interface cible (**Target**) qui permettra l'accès à l'adaptateur.

```
public class WebAdapter implements WebRequester {  
    private WebService service;  
    public void connect(WebService currentService) {  
        this.service = currentService;  
        /* Connect to the web service */  
    }  
    public int request(Object request) {  
        Json result = this.toJson(request);  
        Json response = service.request(result);  
        if (response != null)  
            return 200; // OK status code  
        return 500; // Server error status code  
    }  
    private Json toJson(Object input) { ... }  
}
```

Exemple de DP Adaptateur (suite 3/4)

Step 3: Envoyer la demande du client à l'adaptateur à l'aide de l'interface cible

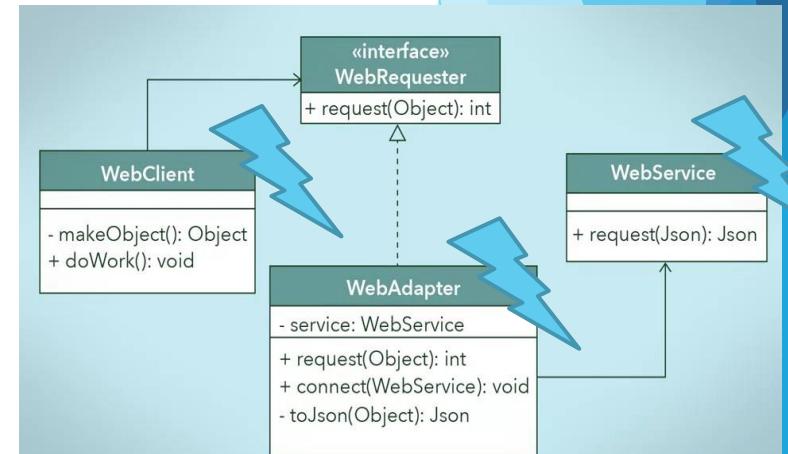
- ▶ Le client web renvoie normalement un objet au client.
- ▶ La méthode **doWork()** méthode ne doit pas être modifiée pour ne pas perturber d'autres parties du système.
- ▶ Au lieu des changements, le client Web devrait effectuer ce comportement comme d'habitude, et rajouter une méthode d'envoi de message, où l'on indique le Adapter, le WebService, et le message à envoyer.

```
public class WebClient {  
    private WebRequester webRequester;  
    public WebClient(WebRequester webRequester) {  
        this.webRequester = webRequester;  
    }  
    private Object makeObject() { ... } // Make an Object  
    public void doWork() {  
        Object object = makeObject();  
        int status = webRequester.request(object);  
        if (status == 200) {  
            System.out.println("OK");  
        } else {  
            System.out.println("Not OK");  
        }  
        return;  
    }  
}
```

Exemple de DP Adaptateur (suite 4/4)

- ▶ Dans le programme principal, vous devez instancier
 - ▶ la classe `WebService`,
 - ▶ la classe `WebAdapter`,
 - ▶ et la classe `WebClient`.
- ▶ Le client Web traite avec l'adaptateur via l'interface `WebRequester` pour envoyer une demande.
 - ▶ Le client Web ne devrait pas avoir besoin de savoir quoi que ce soit sur le service Web, comme son besoin d'objets JSON.
 - ▶ L'Adapté est caché au `Client` par la classe `Adaptateur`.

```
public class Program {  
    public static void main(String args[]) {  
        String webHost = "Host: https://google.com\n\r";  
        WebService service = new WebService(webHost);  
        WebAdapter adapter = new WebAdapter();  
        adapter.connect(service);  
        WebClient client = new WebClient(adapter);  
        client.doWork();  
    }  
}
```



Une réflexion sur l'importance des adaptateurs logiciels?

- ▶ Bien qu'il puisse être tentant de penser qu'une solution est simplement de changer une interface de sorte qu'elle soit compatible avec une autre, ce n'est pas toujours faisable, surtout si l'autre interface est à partir d'une bibliothèque tierce ou d'un système externe.
- ▶ D'autre part, changer votre système pour correspondre à l'autre système n'est pas toujours une solution nonplus, parce qu'une mise à jour par les fournisseurs vers les systèmes extérieurs peut briser une partie de notre système.
 - ▶ Ce n'est pas parce qu'une interface n'est pas conforme à ce que votre système attend que votre système doit changer.

Quiz 2:

Quelles sont les principales caractéristiques du DP Adaptateur?

- A. Les classes **Client** et celle adaptée (**Adaptee**) ont des interfaces incompatibles.
- B. Un adaptateur est une classe d'emballage qui enveloppe l'adapté en le cachant au client.
- C. Le client envoie indirectement des demandes à l'adapté en utilisant l'interface cible de l'adaptateur.
- D. L'adaptateur traduit la demande envoyée par la classe client en une demande à laquelle s'attend la classe adaptée.



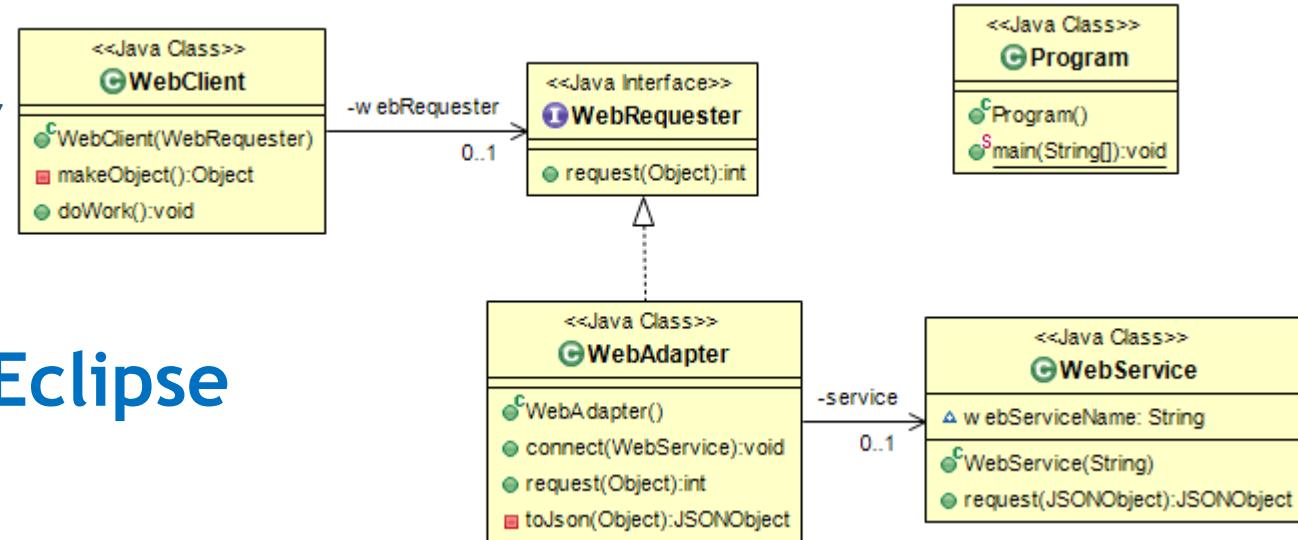
Point à se rappeler à propos du DP Adapteur:

En résumé, un adaptateur est destiné à:

- ▶ Envelopper la classe adaptée (**Adaptee**) et exposer une interface cible au client.
- ▶ Modifier indirectement l'interface de la classe adaptée (**Adaptee**) en une interface à laquelle s'attend le client, et ce en **implémentant une interface cible**.
- ▶ Traduire indirectement la demande du client en une demande que la classe adaptée (**Adaptee**) comprend.
- ▶ Réutiliser une classe adaptée existante avec une interface incompatible.

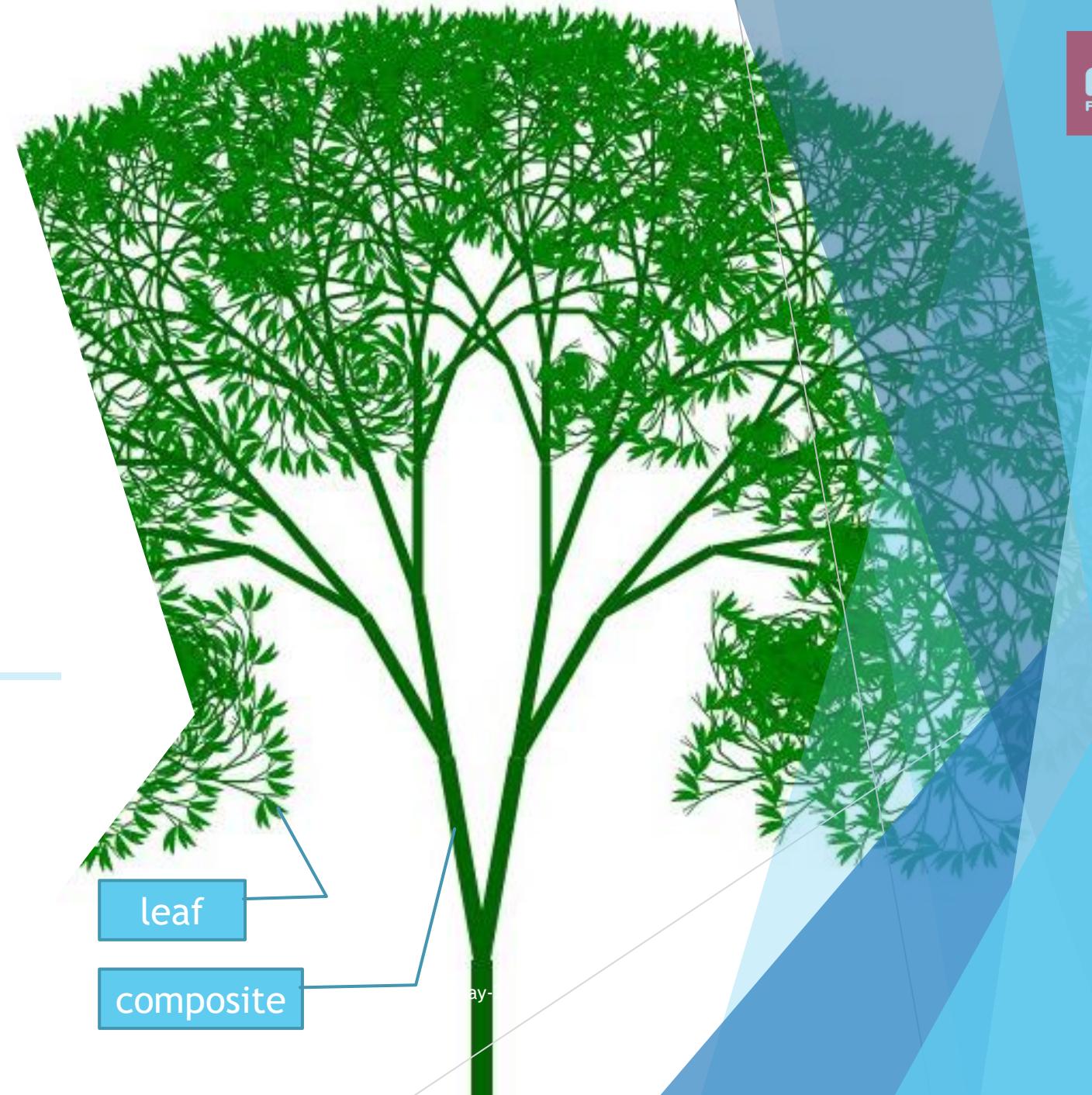
Cela vous permettra de continuer à utiliser vos systèmes existants et d'y intégrer des sources externes additionnelles.

Voir Exemple du Adapter sous Eclipse



IV-3. Le DP de Composition

(Composite DP)

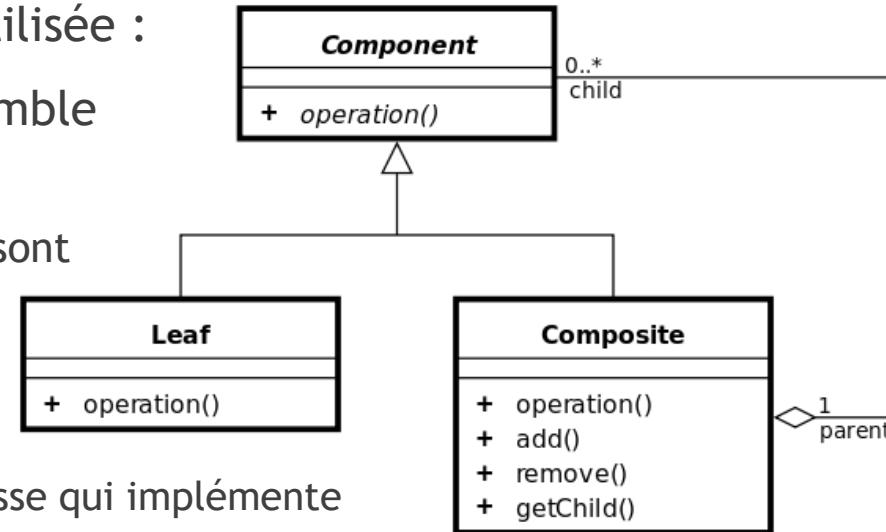


IV.3- Le DP de Composition

- ▶ Le DP de **Composition** fournit une conception pour une hiérarchie, dans laquelle les nœuds ayant des enfants diffèrent dans leur comportements des nœuds sans enfants.
 - ▶ Un exemple peut être l'affichage d'une hiérarchie, comme un système de fichiers, où les dossiers peuvent contenir des fichiers.
 - ▶ Les dossiers seraient affichés différemment des fichiers et auraient du contenu (les fichiers)
- ▶ Ce modèle se compose de trois classes:
 - ▶ La classe **Composite** (le nœud qui peut avoir des enfants),
 - ▶ La classe **Leaf** (Feuille: donc sans enfants),
 - ▶ Et la classe **Component**, qui est une superclasse étendue à la fois par les classes composite Cet la Leaf.
- ▶ Le composite a une collection de composants, de sorte que la classe Composite peut boucler à travers ces composants sans obligation de savoir si le composant est un composite ou une feuille.
- ▶ Le Composite dispose également d'une méthode **addComponent()** afin que des composants puissent être ajoutés à son contenu.
- ▶ **Pourquoi un composant (Component)?** Sans l'abstraction de la super classe composant, le Composite devrait maintenir des listes différentes pour chaque type d'élément dans son contenu,
 - ▶ Il faudrait alors fournir des méthodes individuelles pour ajouter du contenu et afficher du contenu pour chaque type de contenu.

Diagramme UML pour le DP de Composition

- ▶ Il s'agit d'un DP structurel. La conception de base suivante est utilisée :
- ▶ Une interface appelée **Component** sert de supertype pour l'ensemble des classes.
 - ▶ En utilisant le **polymorphisme**, toutes les classes de mise en œuvre sont conformes à la même interface, ce qui leur permet d'être traitées uniformément.
- ▶ Une classe nommée **Composite** est également présente.
 - ▶ Cette classe est utilisée pour regrouper (agrégation) n'importe quelle classe qui implémente l'interface de composants (componenents).
 - ▶ Elle vous permet de « **traverser** » et « **potentiellement manipuler** » les objets composants que l'objet composite contient.
- ▶ Une classe **leaf (feuille)** représente un type non composite. Il n'est pas composé d'autres composants.

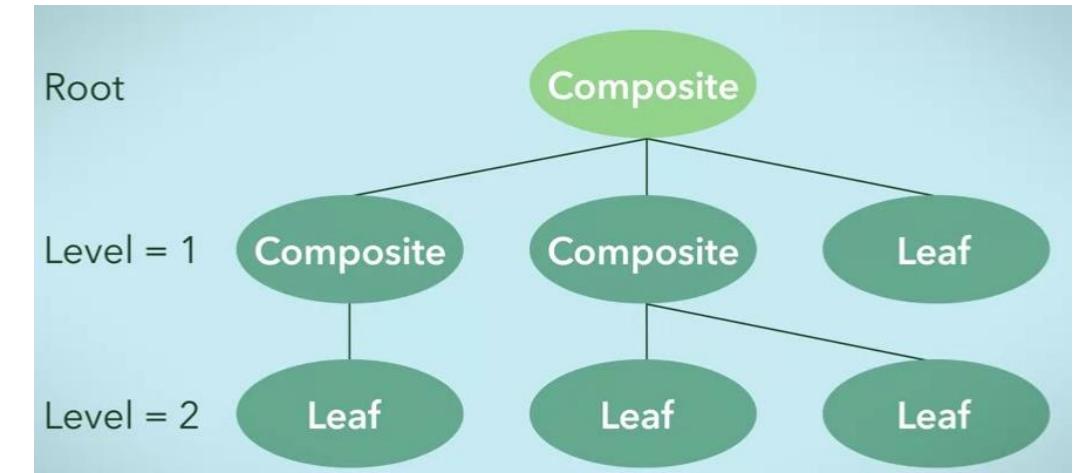


Note: Une superclasse abstraite peut également être utilisée à la place d'une interface, car les deux permettent le polymorphisme:

- ▶ Dans cette implémentation, nous exploitons les interfaces.

Comprendre le DP Composite

- ▶ La classe **Leaf** et la classe **Composite** implémentent tous les deux l'interface **Component**, les unifiant ainsi sous un seul type.
 - ▶ Cela nous permet de traiter les objets non composites et composites uniformément - la classe Leaf et la classe Composite sont maintenant considérées comme des sous-types de composants (Component).
 - ▶ Vous pouvez avoir d'autres classes "composites" ou "feuilles" dans la pratique, mais il n'y aura qu'une seule interface globale ou une superclasse abstraite de tous les composants
- ▶ Un autre concept important à noter est qu'un objet composite peut contenir d'autres objets composites, puisque la classe composite est un sous-type de composant.
 - ▶ C'est ce qu'on appelle la composition récursive.
 - ▶ D'où le nom de composite design pattern.
- ▶ Ce DP a une classe à caractère cyclique,
 - ▶ ce qui peut rendre difficile la visualisation.
- ▶ Pour simplifier, considérez-le comme **un arbre**



Le modèle de conception composite est utilisé pour résoudre deux problèmes:

- ▶ Comment utilisons-nous différents types d'objets pour construire une structure en arbre? et
- ▶ Comment pouvons-nous traiter les différents types d'objets uniformément sans vérifier leurs types?

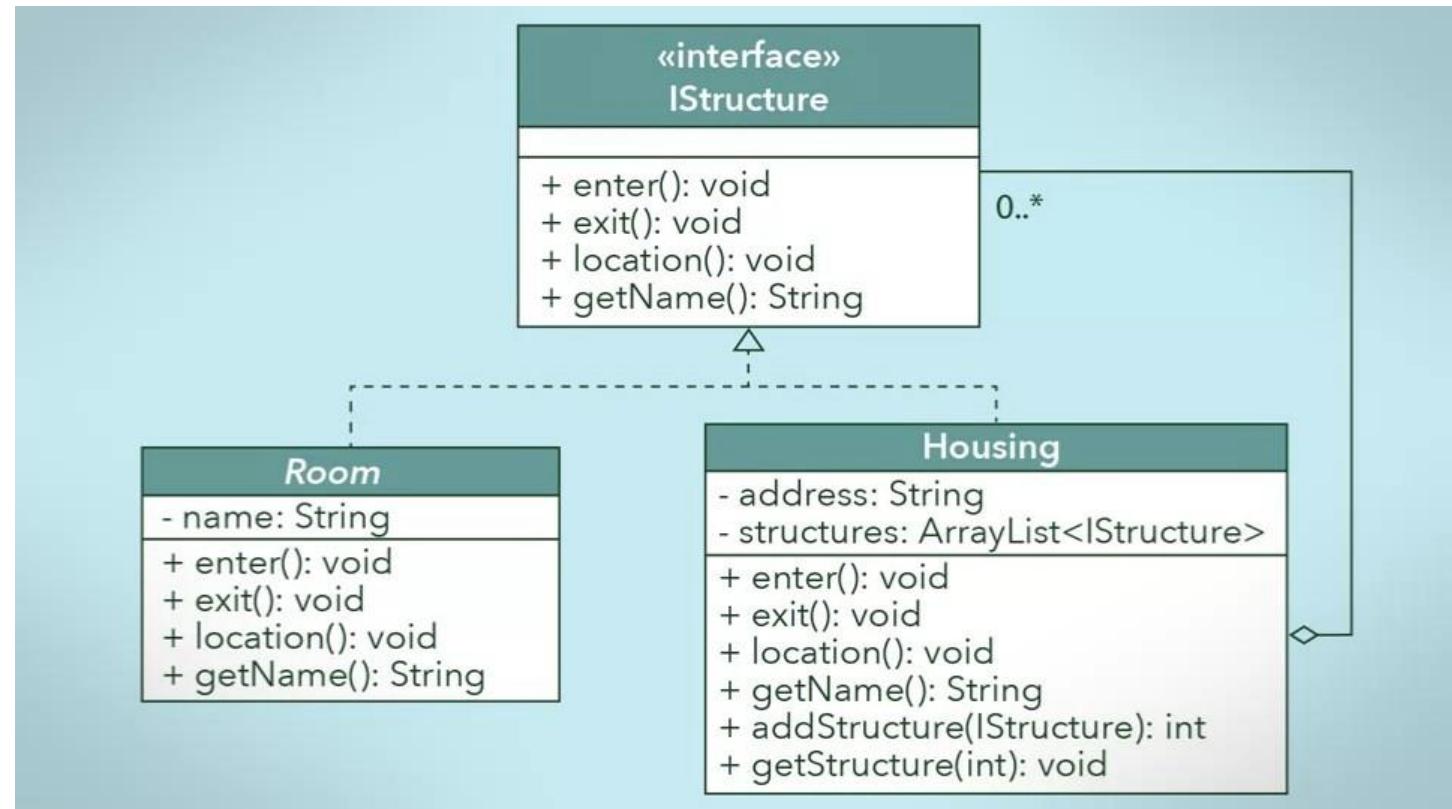
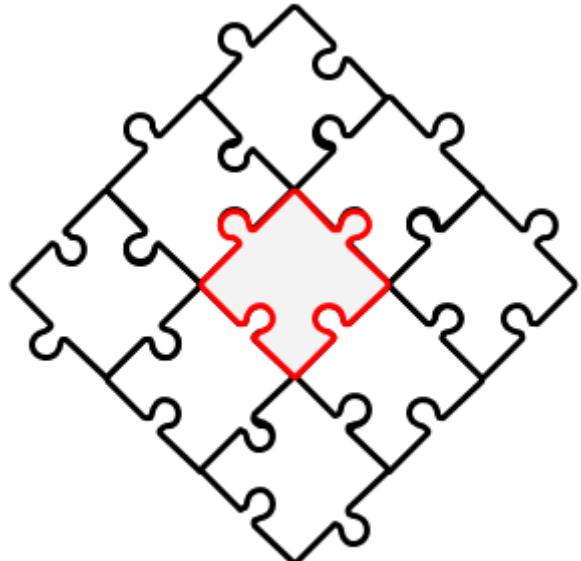
Comment ces deux questions sont-elles traitées par le DP de Composition?

- A. La classe « **Composite** » est capable d'agréger les classes de composants, ce qui créera une structure en forme d'arbre.
- B. Nous pouvons utiliser des instructions ***if-else*** pour déterminer le type d'objet.
- C. La classe « **Feuille** » hérite de la classe Composite. Ceci permet à la classe de feuilles d'avoir les comportements de la classe composite.
- D. Chaque classe est un sous-type d'une interface (ou une superclasse). Elle sera donc en conformité à un ensemble de comportements communs.

Le DP de Composition: Exemple de Logement (Housing)

Exprimer cela en Java (ou tout autre langage de P.O.O.) pourrait être décomposé en 3 étapes:

1. Concevoir l'Interface qui définit le type global.
2. Implémenter la classe Composite.
3. Implémenter la classe Feuille



Step 1: Concevoir l'interface qui définit le type global

- ▶ Commencer par définir l'interface que les classes composites et feuilles implémenteront. Cela force le polymorphisme pour les classes de composants (**Component**) et de feuilles (**Leaf**).

```
public interface IStructure {
    public void enter();
    public void exit();
    public void location();
    public String getName();
}
```

Step 2: Implémenter la classe **Composite**

- ▶ Implémentez l'interface dans la classe composite. Cela donnera à la classe de logement (**Housing**) son propre comportement lorsque le code client y fera appel.

```
public class Housing implements IStructure {
    private ArrayList<IStructure> structures;
    private String address;
    public Housing (String address) {
        this.structures = new ArrayList<IStructure>();
        this.address = address;
    }
}
```

```
public String getName() {
    return this.address;
}
public int addStructure(IStructure component) {
    this.structures.add(component);
    return this.structures.size() - 1;
}
```

Step 2: Implémenter la classe Composite (suite)

```
public IStructure getStructure(int componentNumber) {  
    return this.structures.get(componentNumber);  
}  
public void location() {  
    System.out.println("You are currently in " + this.getName() + ". It has ");  
    for (IStructure struct : this.structures)  
        System.out.println(struct.getName());  
}  
/* Print out when you enter and exit the building */  
public void enter() { ... }  
public void exit() { ... }  
}
```

Step 3: Implémenter la classe de feuilles (Leaf)

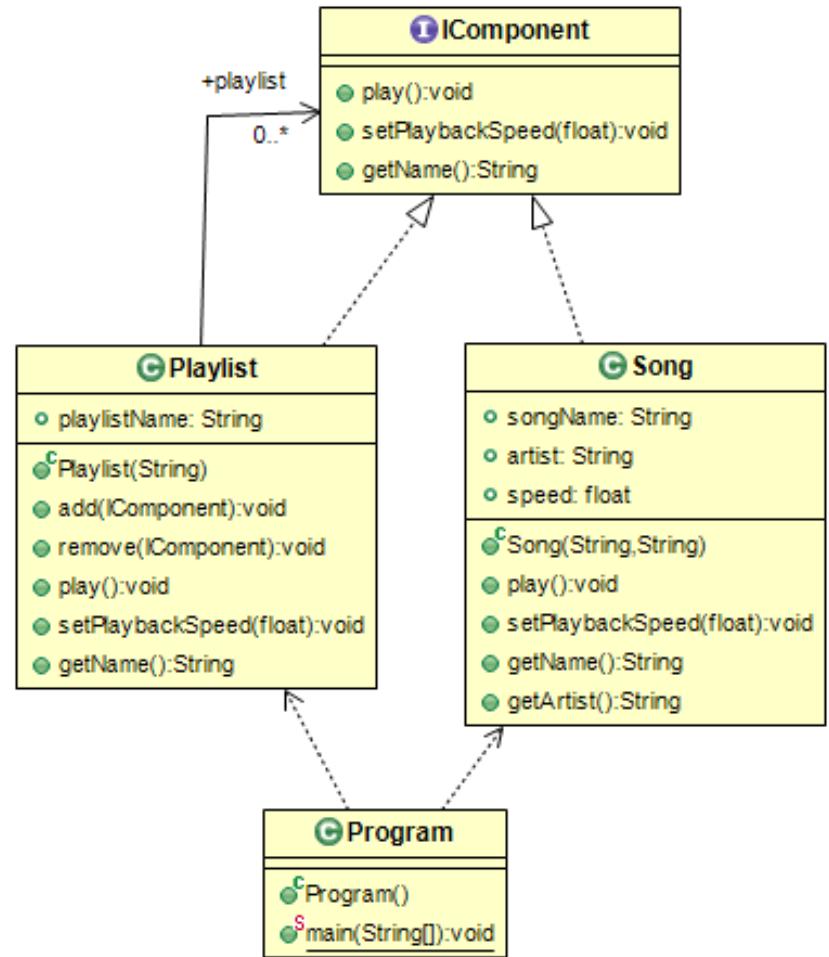
- ▶ La dernière étape consiste à implémenter la classe de feuilles. La classe de feuilles ne contient aucun composants, et n'a pas besoin d'avoir une collection de composants ajoutés à elle, ou des méthodes pour gérer une telle collection.
- ▶ Au lieu de cela, les méthodes de l'interface iStructure doivent simplement être implémentées.

```
public abstract class Room implements IStructure {  
    public String name;  
    public void enter() {  
        System.out.println("You have entered the " +  
this.name);  
    }  
    public void exit() {  
        System.out.println("You have left the " + this.name);  
    }  
    public void location() {  
        System.out.println("You are currently in the " +  
this.name);  
    }  
    public String getName() {  
        return this.name;  
    }  
}
```

Programme principal : Dans cet exemple, l'immeuble dispose d'un étage avec une salle commune (common) et deux salles de bain (washroom). Voir le code ci-dessous

Voir Exemple de DP Composite sous Eclipse

```
public class Program {
    public static void main(String args[]) {
        Housing building = new Housing("123 Street");
        Housing floor1 = new Housing("123 Street - First Floor");
        int firstFloor = building.addStructure(floor1);
        Room washroom1m = new Room("1F Men's Washroom");
        Room washroom1w = new Room("1F Women's Washroom");
        Room common1 = new Room("1F Common Area");
        int firstMens = floor1.addStructure(washroom1m);
        int firstWomans = floor1.addStructure(washroom1w);
        int firstCommon = floor1.addStructure(common1);
        building.enter(); // Enter the building
        Housing currentfloor = building.getStructure(firstFloor);
        currentFloor.enter(); // Walk into the first floor
        Room currentRoom = currentFloor.getStructure(firstMens);
        currentRoom.enter(); // Walk into the men's room
        currentRoom = currentFloor.getStructure(firstCommon);
        currentRoom.enter(); // Walk into the common area
    }
}
```



Points à retenir à propos du DP de Composition

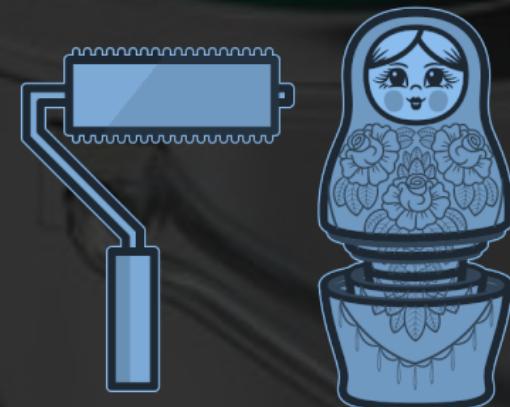
Les objets composites peuvent être construits rapidement et facilement:

- ▶ Chaque composant devrait avoir le même ensemble de comportements sans avoir besoin de vérification de type

En somme, le DP de Composition nous permet de :

- ▶ construire une **structure d'objets en arbre**, et de traiter les différents types de ces objets **uniformément**. Pour ce faire,:
 - ▶ Force le **polymorphisme** dans chaque classe en implémentant une interface (ou en héritant d'une superclasse).
 - ▶ En utilisant une technique appelée **composition récursive** qui permet aux objets d'être composés d'autres objets de type commun.
- ▶ Les DP composites appliquent les principes de conception de **décomposition** et de **généralisation**.
- ▶ Ils cassent l'ensemble en parties, tout en veillant que l'ensemble et les pièces qui le composent sont toutes conformes à un type commun.
 - ▶ Des structures complexes peuvent être construites à l'aide d'objets composites et d'objets feuilles appartenant à un type de composant uniifié.
 - ▶ Il est ainsi plus facile de comprendre et de manipuler une structure donnée.

IV.4- Le DP Décorateur



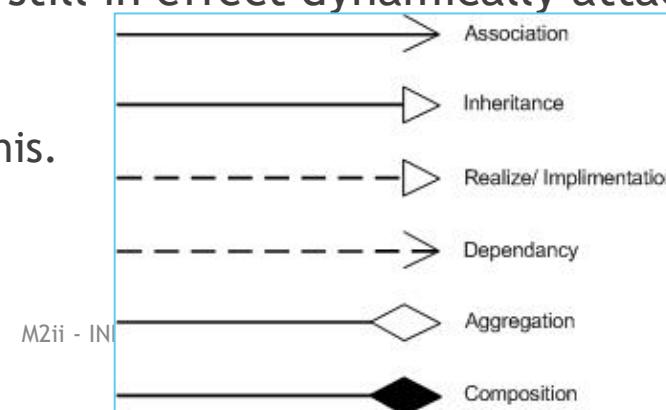
Le DP Décorateur (conçu comme du Café!)

- ▶ Beaucoup d'entre nous consomment du café. Et, oui, je suis un accro de café!
 - ▶ La plupart du temps, j'aime boire une tasse ordinaire de « café noir ».
 - ▶ Parfois, je peux passer à « cappuccino ». Je peux aussi essayer d'autres saveurs et goûts selon les choix disponibles et les humeurs.
 - ▶ Le café peut également être mélangé avec d'autres ingrédients comme « lait », « chocolat », « crème aromatisée », etc.
 - ▶ Il peut également venir dans différents « Degrés Celsius » de « chaud » à « froid ».
 - ▶ Avec tous ces choix différents, un élément est toujours nécessaire : « la caféine » provenant des noix de cacao.
- ▶ Quel que soit le type de café que nous buvons, les effets de chaque ingrédient ajouté ne changent pas l'essence initiale du café: **Il agira toujours comme un stimulant.**
- ▶ Chaque ingrédient détient ses propres caractéristiques, et en les combinant, on crée une nouvelle boisson qui pourrait être plus agréable qu'un simple café basique.
- ▶ Le contenu du café est **ajouté dynamiquement** sur le tas, comme surplus à la première composante de café noir.



“Ajout dynamique” 😊 VS “beaucoup de classes” 😞

- ▶ Dans les logiciels, comme la combinaison d'une nouvelle boisson, il est bénéfique d'avoir des combinaisons flexibles de comportements globaux.
- ▶ But we encounter an issue **trying to do this dynamically at runtime**, that's because the behavior of an object is defined by its class, **but the notion of a class and relationships like inheritance are static**.
 - ▶ The class and its contents definition happens at compile time.
 - ▶ So, we cannot make changes to classes while our program is running.
 - ▶ We need to create a **new class** in order to achieve any **new behavior** combinations.
- ▶ As a result, having lots of new combinations would lead to lots of classes and **we don't want that 😞!**
- ▶ Given that an object has a certain behavior, can we still in effect dynamically attach additional behaviors or responsibilities to it?
 - ▶ Luckily, there is the **Decorator DP** to achieve this.
 - ▶ It uses **aggregation** to combine behaviors at **runtime**.



L'agrégation nous sauve

- ▶ We can use that "**has a**" relationship to build a stack of objects where each level of the stack contains an object that knows about its own behavior and augments the one underneath it in the stack.
- ▶ The aggregation relationship **is always one-to-one** in the Decorator DP in order to build up the **stack** so that one object is on top of another.
- ▶ This is what an aggregation stack looks like:
 - ▶ **Object A** is a base object because it doesn't contain another object. It will have its own set of behaviors.
 - ▶ **Object B** aggregates object A allowing object B to, augment the behaviors of object A.
 - ▶ We can continue to add objects to the stack in such a way that **objects C** aggregates object B to augment the behaviors of object B.
- ▶ To achieve an overall combination of behaviors for the stacked objects, we call:
 1. The top element which is **object C**. Object C would call upon **object B**. And object B would call upon **object A**.
 2. Object A **responds** with its behavior, then object B **adds** its incremented behavior, and then object C adds its **incremented** behavior.

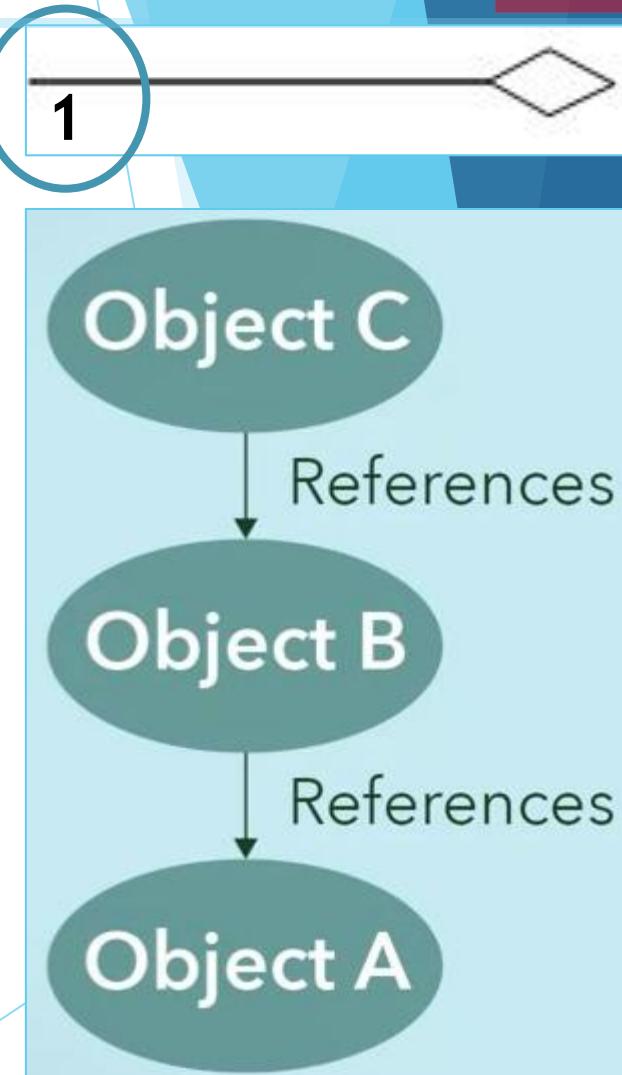


Diagramme UML du DP Décorateur

- ▶ This design pattern makes use of both interfaces and inheritance so that the classes conform to a common type, whose instances can be stacked up in a coherent and compatible way that builds up.
1. The component interface is used to define the common type for all the classes.
 - ▶ A client class will expect the same interface across all the component classes.
 2. A concrete component class implements the component interface and can be instantiated.
 - ▶ An instance of this class can be used as a base object in the stack
 3. Decorator is an abstract class.
 - ▶ Just like the concrete component class the also implements the component interface.
 - ▶ The main differences are that decorator aggregates other types of components
 - ▶ Allows to stack components on top of each other,
 - ▶ Decorators serve as the abstract superclass of concrete decorator classes that will provide an increment of behavior.

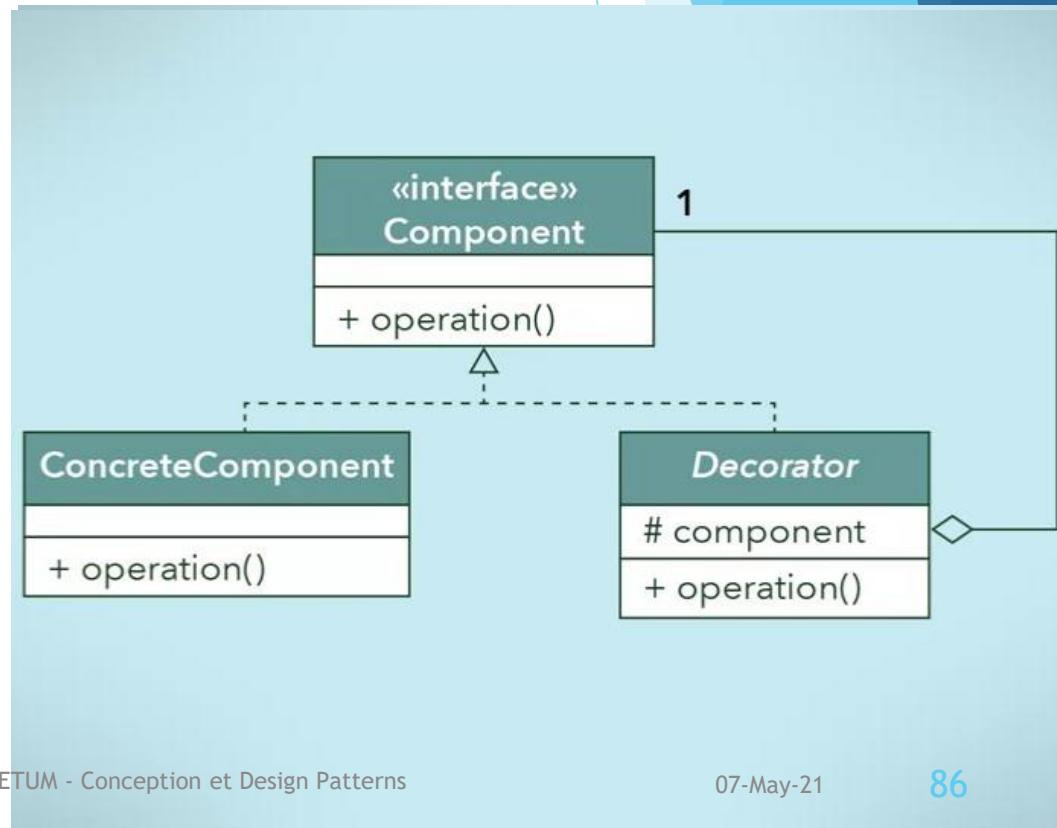
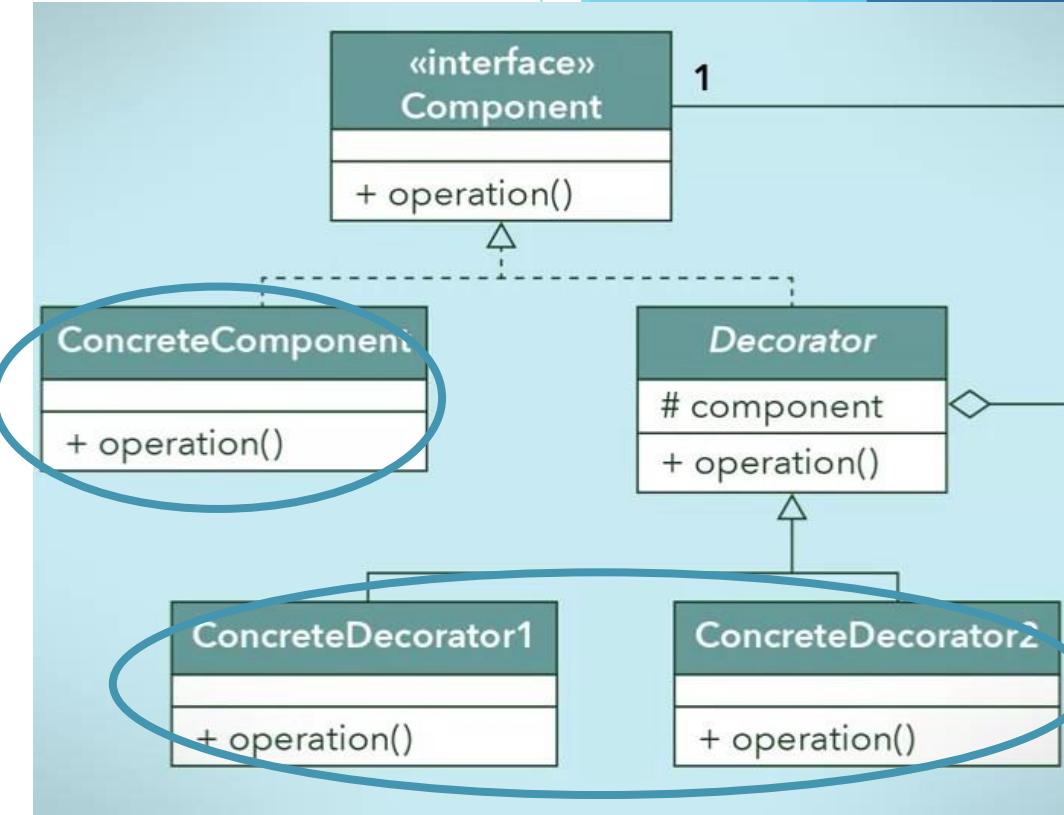


Diagramme UML du DP Décorateur (suite)

4. We build the stack of components starting with:
 - ▶ an instance of the **concrete component** class
 - ▶ and continuing with **instances of subclasses** of the decorator abstract class.
- ▶ In terms of our precedent coffee analogy:
 - ▶ the concrete component would be a **black coffee**.
 - ▶ Decorators for our coffee would be **milk, sugar, hot chocolate**, and any other ingredient you could add to a coffee
- ▶ Let's now look at an example that is more realistic to what you would see in a software system.



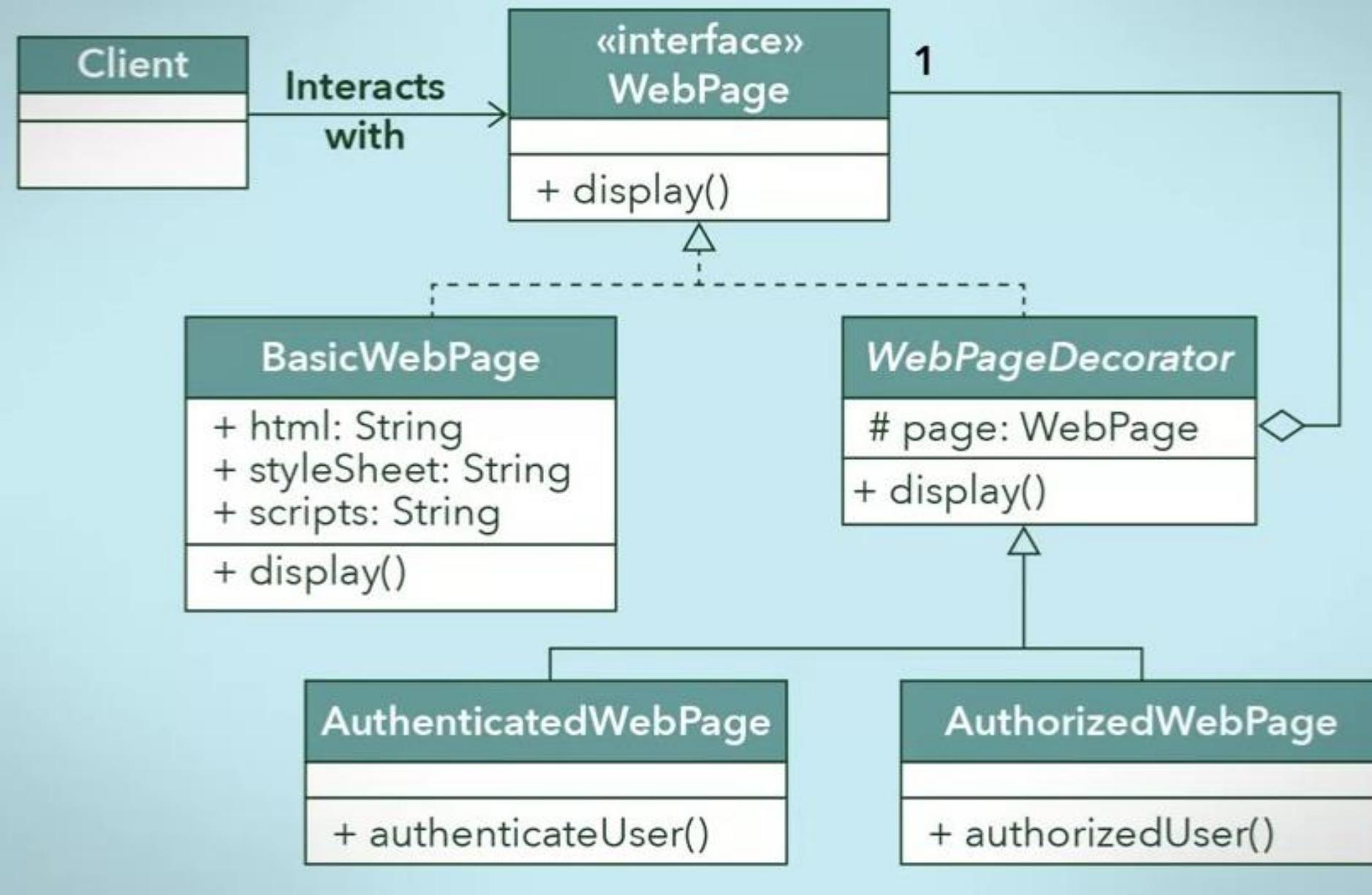
Quelles sont les raisons de l'utilisation du modèle de conception de décorateur?

- A. Le modèle de conception de décorateur permet aux objets d'ajouter dynamiquement des comportements aux autres.
- B. Il vous permet de construire une structure d'objets en forme d'arbre qui peut être traitée comme un seul type d'objet uniforme.
- C. Pour cacher un objet complexe et exigeant en ressources jusqu'à ce qu'il soit nécessaire.
- D. Pour réduire le nombre de classes nécessaires pour offrir une combinaison de comportements.

Exp. de cas d'utilisation du DP Décorateur

- ▶ Let us examine the decorator design pattern using an example of a webpage.
- ▶ A web page might display complex behavior, such as:
 - ▶ only allowing access to authorized users,
 - ▶ or splitting search results across multiple pages, etc.
- ▶ Every possible combination of web page permission, pagination, or caching behaviors does not need to be written as different web page types.
 - ▶ The Decorator DP can be used to create one class for each type of behavior and build specific combinations for a web page that you want at run time.
 - ▶ For simplicity, the **BasicWebPage's HTML**, stylesheet, and scripts can be represented as **Strings**.
 - ▶ Any number of additional behaviours can be used to augment the basic web page,
 - ▶ this example is limited to the behaviours of adding **user authorization**, and **authentication** to make sure the user is who they claim to be.
- ▶ When expressed as a UML diagram, the example outlined before looks as:





Implémentation de l'exemple de page Web

Comme dans les DP précédents, nous suivrons les 4 étapes suivantes:

1. Concevoir l'interface des composants (**Component**)
2. Implémenter l'interface avec votre classe concrète (**ConcreteComponent**) de composants de base
3. Implémenter l'interface avec votre classe de décorateur(**AbstractDecorator**) abstrait
4. Hériter du décorateur abstrait et implémentez l'interface des composants avec des classes de décorateurs concrets (**ConcreteDecorator**)



Implémentation de l'exemple de page Web (suite)

Step 1: Concevoir l'interface des composants :

- ▶ First, define the interface that the rest of the classes in the design pattern will be subtypes of.
- ▶ This interface will define the **common behaviors** that your basic web page and decorators will have.

```
public interface WebPage {  
    public void display();  
}
```

Step 2: Implémenter l'interface avec votre classe concrète (**ConcreteComponent**) de composants de base :

- ▶ Next, implement the base concrete component class with the interface.
 - ▶ The concrete component class will be the base building block for all web pages objects during run time.
- ▶ The concrete component class will implement how it displays itself by using standard HTML markup and page element styling defined in the cascading style sheet.
 - ▶ This example web page will also run some basic JavaScript.

```
public class BasicWebPage : WebPage {  
    public string html = ...;  
    public string styleSheet = ...;  
    public string script = ...;  
    public void display() {  
        /* Renders the HTML to the stylesheet,  
        and run any embedded scripts */  
    }  
}
```

Implémentation de l'exemple de page Web (suite)

*Step 3: Implémenter l'interface avec votre classe de décorateur(**AbstractDecorator**) abstrait*

- ▶ Next, the abstract decorator class should be implemented with the interface.
 - ▶ This implementation is important.
- ▶ The web page decorator only **has one instance** of web page.
 - ▶ This allows decorators to be stacked on top of the basic web page, and on top of each other.
- ▶ Each type of web page is responsible for its own behavior and will recursively invoke the next web page on the stack to execute its behavior.
 - ▶ The **constructor** allows different subtypes of web pages to be linked together in a stack. All that must be done is to indicate what instance of web page subtype should be stacked upon.
- ▶ In stacking, **the order** in which you build the stack **matters**.
 - ▶ The basic web page must be the **first** one in the stack.
 - ▶ The rest of the ordering will depend on the design of your system and which augmenting behaviors you want executed first.

```
public abstract class WebPageDecorator : WebPage {  
    protected WebPage page;  
    public WebPageDecorator (WebPage webpage) {  
        this.page = webpage;  
    }  
    public void display() {  
        this.page.display();  
    }  
}
```

- ▶ The abstract decorator simply delegates the display behavior to the web page object that it aggregates.
 - ▶ This allows you to combine the display behavior down the stack of web pages

Step 4: Hériter du décorateur abstrait et implémentez l'interface des composants avec des classes de décorateurs concrets (ConcreteDecorator**)**

- ▶ The final step is to inherit from the abstract decorator and implement the component interface with the concrete decorator classes.
- ▶ In this example, the constructors use the abstract superclass' constructor, as it allows decorators to be stacked together.
 - ▶ The abstract web page decorator class handles the aggregation of the concrete decorator classes.
 - ▶ Further, each decorator has its own responsibilities. These are implemented within the appropriate classes so that they can be invoked.

```
public class AuthorizedWebPage extends WebPageDecorator {  
    public AuthorizedWebPage(WebPage decoratedPage) {  
        super(decoratedPage); }  
    public void authorizedUser() {  
        System.out.println("Authorizing user");  
    }  
    public display() {  
        super.display();  
        this.authorizedUser();  
    }  
}
```

```
public class AuthenticatedWebPage extends WebPageDecorator {  
    public AuthenticatedWebPage(WebPage decoratedPage) {  
        super(decoratedPage); }  
    public void authenticateUser() {  
        System.out.println("Authenticating user");  
    }  
    public display() {  
        super.display();  
        this.authenticateUser();  
    }  
}
```

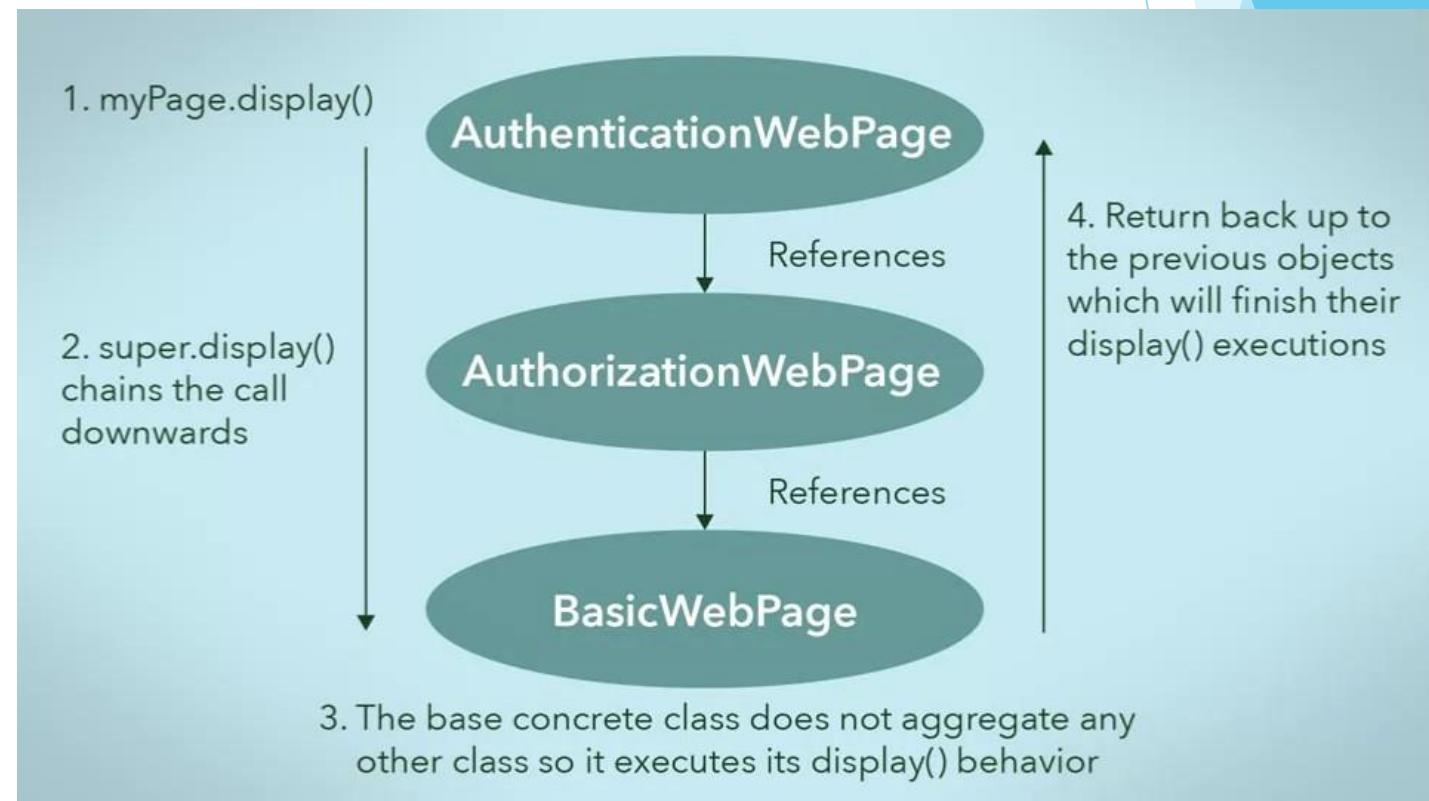
Example: Programme principal (Main)

- ▶ Noter que pour appeler récursivement le comportement d'affichage, les décorateurs concrets invoquent la méthode d'affichage de la superclasse.
 - ▶ Puisque la superclasse abstraite de décorateur facilite l'agrégation de divers types de conception de pages web, l'appel à **basic.display()** provoquera que la page Web suivante dans la pile exécute sa version d'affichage jusqu'à arriver à la page Web de base.
- ▶ L'appel récursif se terminera ici parce que la page Web de base est le composant concret, qui n'agrège aucun autre type de pages Web.
 - ▶ Cela relie les appels d'affichage tout le long du chemin vers le fond, et fait bouillonner l'exécution dans le chemin de retour. La page Web de base doit ainsi être construite avant que des comportements puissent y être ajoutés.
- ▶ Examinons maintenant le DP de décorateur en action:

```
public class Program {  
    public static void main(String args[]) {  
        WebPage myPage = new BasicWebPage();  
        myPage = new AuthorizedWebPage(myPage);  
        myPage = new AuthenticatedWebPage(myPage);  
        myPage.display();  
    }  
}
```

Example: La pile

- ▶ As explained above, the basic web page is built first. The authorization behaviour is added next. The web page is completed by adding the authentication behaviour last.
 - ▶ When the `display()` method is called, it will link the method calls down to the basic web page.
- ▶ The basic web page will display itself and then the display call will move back up the links to the authorized behaviour first and the authentication behaviour last. As an aggregation stack diagram, this would look as follows:
- ▶ Any decorator could be added to the basic web page to create a different combined behaviour.
- ▶ The basic web page's behaviour can be dynamically built up in this way.

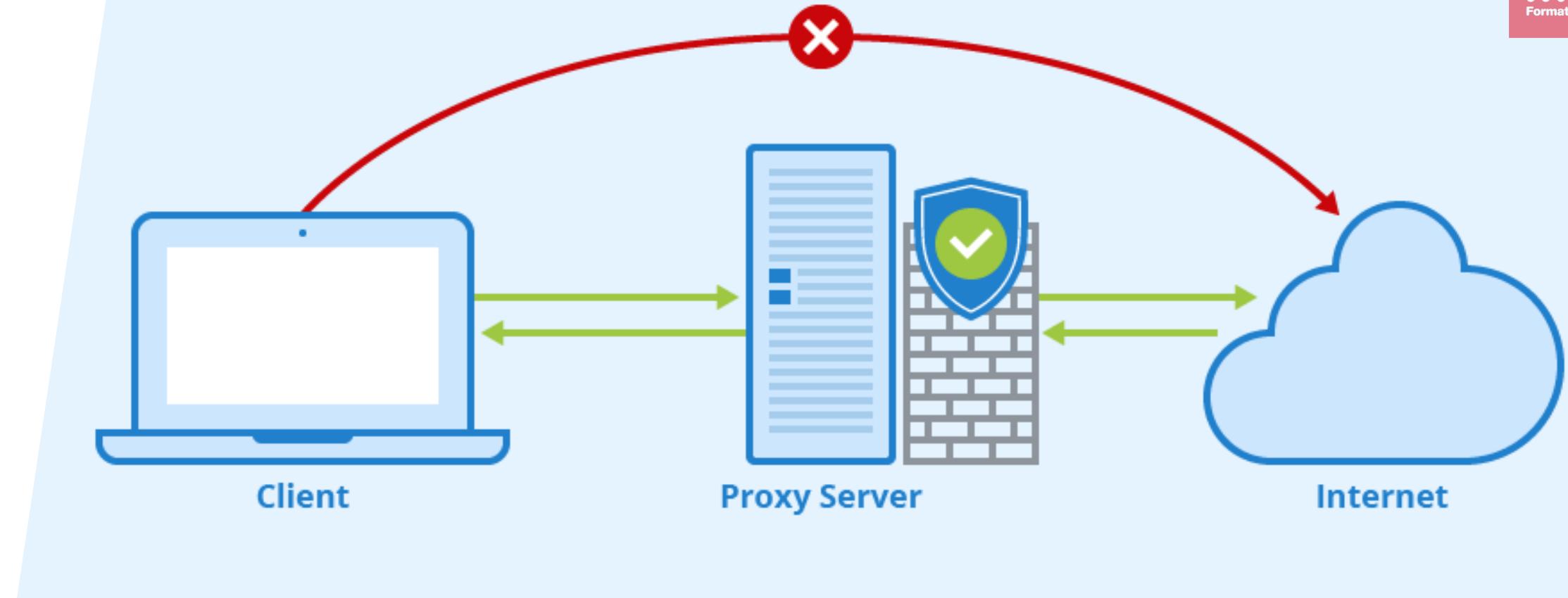


Décorateur: Résumé

Les principales caractéristiques d'un modèle de conception Décorateur sont:

- ▶ Nous pouvons **ajouter**, en effet, n'importe quel nombre de comportements dynamiquement à un objet à l'heure d'exécution en utilisant **l'agrégation** comme une alternative à l'héritage pure
- ▶ **Le polymorphisme** est atteint par l'implémentation d'une **unique interface**.
- ▶ L'agrégation nous permet de créer une **pile** d'objets.
- ▶ Chaque objet décorateur dans la pile est agrégé dans une relation en **un-à-un** avec l'objet en dessous dans la pile. Et,
- ▶ En combinant agrégation et polymorphisme, nous pouvons invoquer de façon récursive le même comportement dans la pile et faire exécuter le comportement vers le haut à partir de l'objet composant (**Component**) concret.

L'utilisation du DP décorateur aide à créer un logiciel complexe, sans payer un surplus de complexité dans le projet.



IV.5- Le Design Pattern Proxy

Le Design Pattern Proxy

- ▶ Un **proxy** est quelque chose qui agit comme une version simplifiée, ou légère, de l'objet original.
- ▶ Le DP proxy permet à une classe proxy de représenter une véritable classe de « sujet ».
- ▶ Un objet proxy peut effectuer les mêmes tâches qu'un objet original, mais peut aussi **déléguer** des tâches spécifiques à l'objet original pour les réaliser.
- ▶ Dans ce DP, la classe proxy enveloppe la classe de sujets réelle.
 - ▶ Une référence à une instance de la classe de sujet réel est cachée par la classe proxy.
 - ▶ L'objet réel habituellement:
 - ▶ Contient des informations sensibles et fait partie du système logiciel,
 - ▶ serait gourmand en ressources pour qu'il ne soit instancié.
 - ▶ Comme la classe proxy est une enveloppe représentative, les classes clientes interagissent avec elle au lieu de la classe de sujet réel. Exemples:



3 scénarios les plus courants pour un Proxy

▶ Pour agir comme un proxy virtuel:

- ▶ C'est lorsqu'une classe proxy est utilisée à la place d'une classe de sujet réel qui serait gourmand en ressources si instancié.
- ▶ Couramment utilisé sur les images dans les pages Web ou les éditeurs graphiques, comme une image haute définition pourrait être extrêmement lourde à charger

▶ Agir comme un proxy de protection:

- ▶ Une classe proxy est utilisée pour contrôler l'accès à la classe de sujets réels.
- ▶ Par exemple, un système utilisé par les étudiants et les instructeurs pourrait limiter les accès à certains services en fonction des rôles.

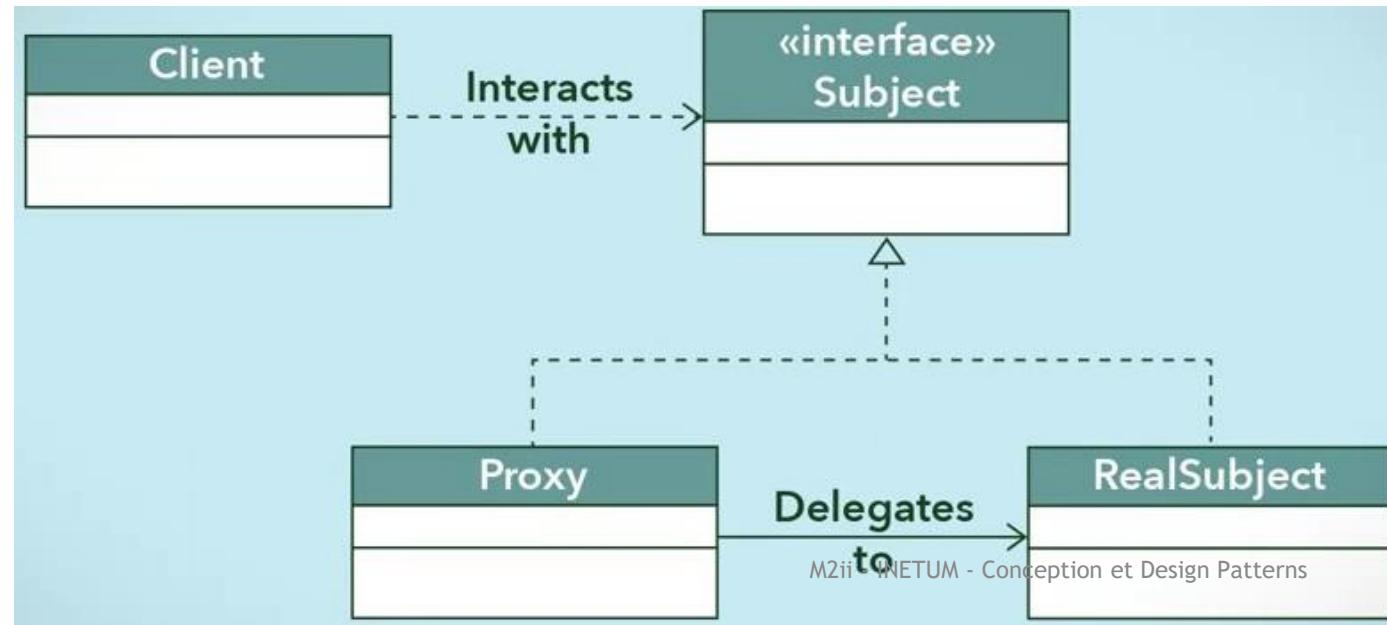
▶ Pour agir comme un proxy distant (remote):

- ▶ Lorsqu'une classe proxy est locale et que la classe de sujets réels existe à distance.
- ▶ Google docs faire usage de cela, où les navigateurs Web possèdent tous les objets dont ils ont besoin localement. Des objets qui existent également sur un serveur Google distant ailleurs.



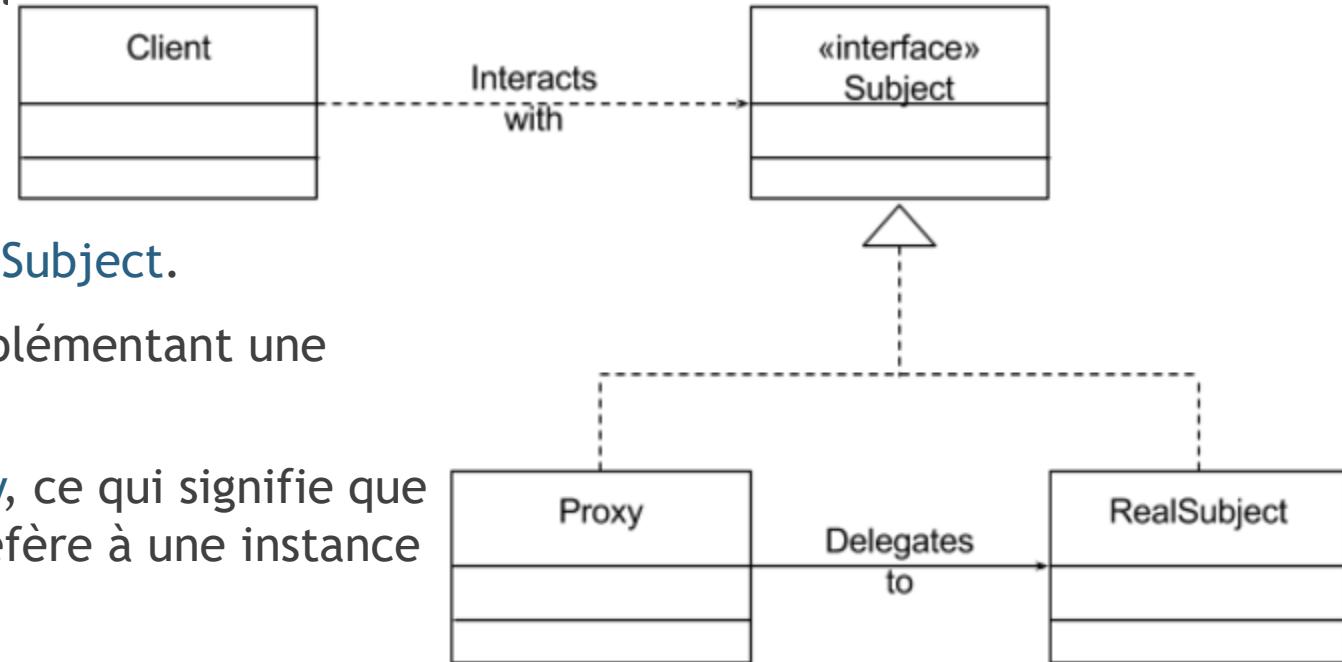
Proxy DP. UML diagram

- ▶ The proxy class wraps and may delegate, or redirect, calls upon it to the real subject class.
 - ▶ Not all class contents are delegated, as the proxy class can handle some its lighter responsibilities.
 - ▶ Only substantive requests are sent to the real subject class.
 - ▶ Because of this, the proxy class must offer the same methods.
- ▶ Both classes implement a common interface allowing for polymorphism
- ▶ A client class can interact with the proxy, which will have the same expected interface as the real subject.



Quiz

- Quelles conclusions peuvent être tirées à propos du D Proxy en se basant sur sa structure montrée dans ce diagramme de classe UML?



- A. Proxy et RealSubject sont des sous-types de Subject.
- B. Le DP Proxy réalise le polymorphisme en implémentant une interface Subject.
- C. La classe RealSubject connaît la classe Proxy, ce qui signifie que la classe RealSubject *a un attribut* qui se réfère à une instance de la classe Proxy.
- D. Le DP Proxy ne réalise pas de polymorphisme parce que les classes Proxy et RealSubject implémentent toutes deux la classe Subject.

Example of a Proxy use case

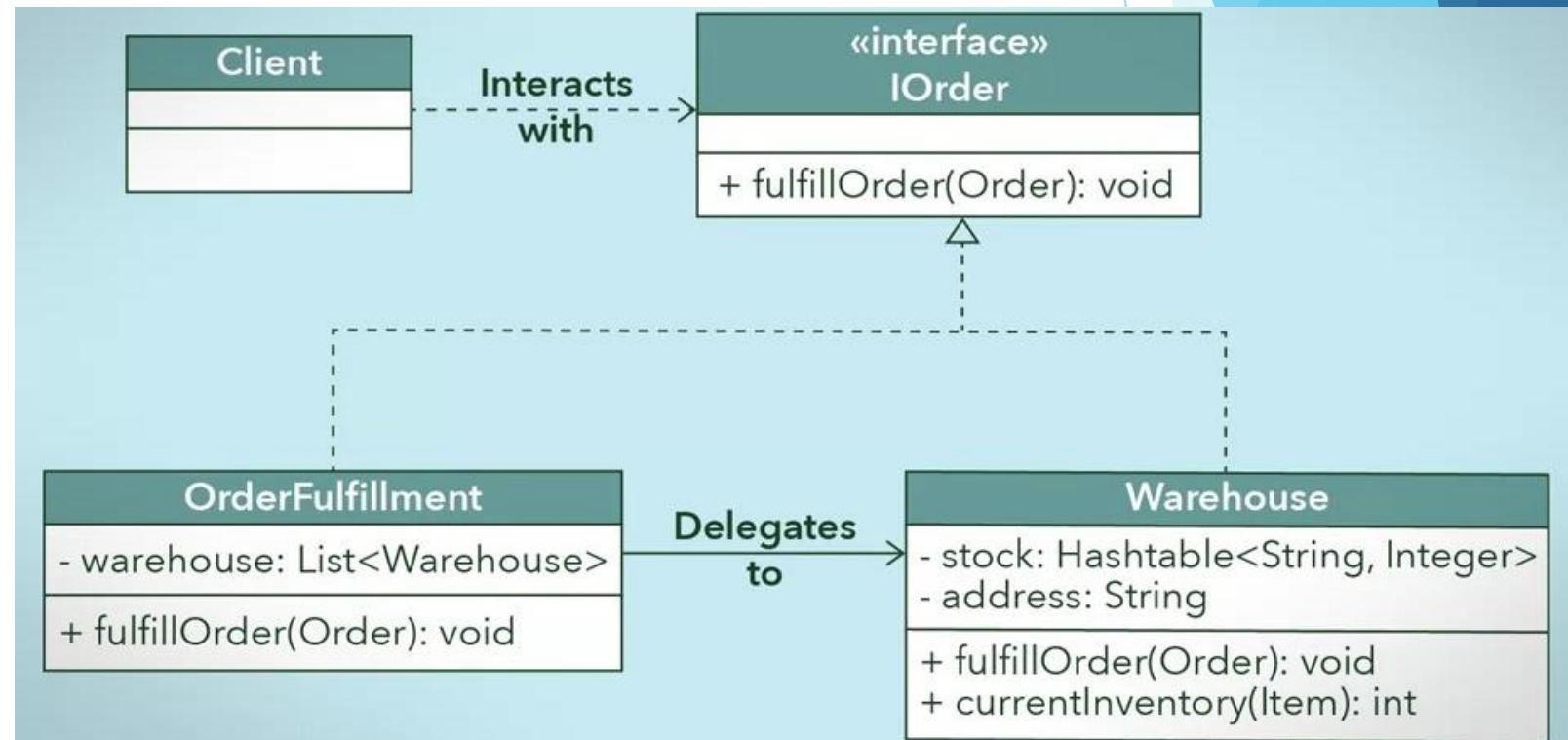
- ▶ Lets consider **an online retail store** (magasin de détail) with **global distribution** and **warehousing** (stockage).
 - ▶ In this scenario, you need to determine which warehouse to send orders to.
 - ▶ A system that routes orders for fulfillment to an appropriate warehouse will **prevent your warehouses from receiving orders that they cannot fulfill**.
 - ▶ A proxy may protect your real subject, the warehouses, from receiving orders if the warehouses do not have enough stock to fulfill an order.
- ▶ A proxy **may protect your real subject**, the warehouses, from receiving orders if the warehouses do not have enough stock to fulfill an order
- ▶ In **The Order Fulfillment class** is the proxy in this example.
- ▶ Clients interact with the system by using an **interface**



Solution UML proposée

L'Implémentation de ce DP en C# peut être décomposée en 3 étapes.

1. Concevoir l'interface Subject
2. Implémenter la classe RealSubject
3. Implémenter la classe Proxy



Quelles sont les principales responsabilités de la classe Proxy?

- A. Elle reçoit ses tâches depuis la classe de sujet réel (RealSubject).
- B. Elle protège la classe de sujets réels en vérifiant la demande du client et en contrôlant l'accès à la classe de sujets réels.
- C. Elle agit comme une classe d'enveloppe pour la classe du sujet réel.
- D. Elle fournit au client une nouvelle interface, différente de l'interface du sujet réel, pour interagir avec le système.

Step 1: Design the subject interface

- ▶ First, create the interface that client software will use to interact with your system.

```
public interface IOrder {  
    public void fulfillOrder(Order);  
}
```

Step 2: Implement the real subject class

- ▶ Implement the real subject class. In this case, the Warehouse class knows how to process an order for fulfillment, and it knows how to report the current stock of items.
- ▶ It does not need to check if it has enough stock to fulfill an order, as an order should only be sent to a warehouse if it can be fulfilled.

```
public class Warehouse implements IOrder {  
    private Hashtable<String, Integer> stock;  
    private String address;  
    /* Constructors and attributes here */  
    ...  
}
```

```
public void fulfillOrder(Order order) {  
    for (Item item : order.itemList)  
        this.stock.replace(item.sku, stock.get(item.sku)-1);  
    /* Process the order for shipment and delivery */  
    ...  
}  
  
public int currentInventory(Item item) {  
    if (stock.containsKey(item.sku))  
        return stock.get(item.sku).intValue();  
    return 0;  
}
```

Step 3: Implement the proxy class

- ▶ In this case, the **Order Fulfillment** class checks warehouse inventory and ensures that an order can be completed before sending requests to the warehouse.
 - ▶ it asks each warehouse if it has enough stock of a particular item.
 - ▶ If a warehouse does, then the item gets added to a new Order object that will be sent to the Warehouse.
 - ▶ The Order Fulfillment class also lets you separate order validation from the order fulfillment by separate them into two pieces.
 - ▶ This improves the overall rate of processing an order, as the warehouse does not have to worry about the validation process or about re-routing an order if it cannot be fulfilled.
- ▶ The Order Fulfillment class can be improved with other functionalities, such as prioritizing sending orders to warehouses based on proximity to the customer.

```
public class OrderFulfillment implements IOrder {  
    private List<Warehouse> warehouses;  
    /* Constructors and other attributes would go here */  
    public void fulfillOrder(Order order) {  
        /* For each item in a customer order, check each warehouse to see if it is in stock.  
        If it is then create a new Order for that warehouse. Else check the next warehouse. Send the all the  
        Orders to the warehouse(s) after you finish iterating over all the items in the original Order. */  
        for (Item item: order.itemList) {  
            for (Warehouse warehouse: warehouses) {  
                ...  
            }  
        }  
        return;  
    }
```

If it is then create a new Order for that warehouse. Else check the next warehouse. Send the all the Orders to the warehouse(s) after you finish iterating over all the items in the original Order. */

Points à retenir sur le DP Proxy

- ▶ Le DP proxy permet de :
 1. **Reporter** la création d'objets à forte intensité de ressources jusqu'à ce que nécessaire,
 2. **Contrôler l'accès** à des objets spécifiques, ou lorsque qu'il y a besoin de quelque d'un intermédiaire pour agir comme une représentation locale d'un système distant.
 3. Rendre les systèmes **plus sûrs** et moins gourmands en ressources.
- ▶ Les principales caractéristiques du DP proxy sont:
 - ▶ To use the proxy class to **wrap the real subject class**.
 - ▶ To have a **polymorphic design** so that the client class can expect the same interface for the proxy and real subject classes.
 - ▶ To use a **lightweight** proxy in place of a resource intensive object until it is actually needed.
 - ▶ To implement some form of **intelligent verification** of requests from client code in order to determine if, how, and to whom the requests should be forwarded to.
 - ▶ To present a **local representation** of a system that is not in the same physical or virtual space.

07-May-21

V- Les DP Comportementaux

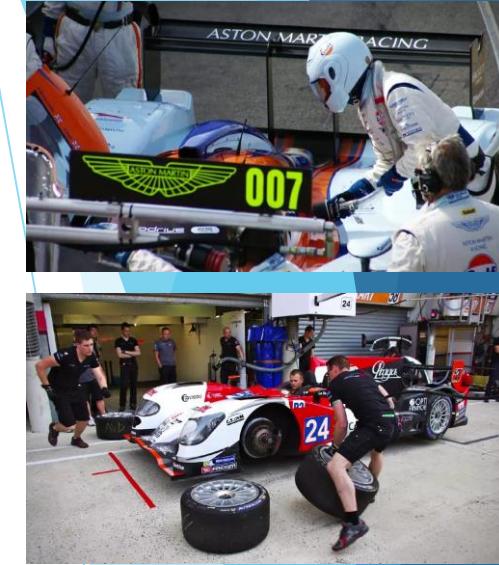
Behavioral DPs

Les patterns de Comportement (Behavioural patterns)

C'est des modèles de conception qui mettent l'accent sur la façon dont les objets distribuent le travail.

Il met également l'accent sur la façon dont chaque objet indépendant contribue à l'objectif final.

Pour chaque objet, afin de fonctionner correctement, il doit avoir un objectif préétabli dès le départ.



Les DP suivants définissent la façon de répartir les tâches entre les différents objets.

- ▶ **Le DP Chaîne de responsabilités (Chain of Responsibility)** - Permet à plusieurs entités de traiter les demandes.
- ▶ **Le DP Commande (Command)** - Encapsule les demandes en tant qu'objets.
- ▶ **Le DP Observateur (Observer)** - Gère les événements.
- ▶ **Le DP Etat (State)** - Traite des demandes en tenant compte de l'état actuel de l'objet.
- ▶ **Le DP Méthode de modèle (Template Method)** - Permet l'héritage d'un comportement similaire par plusieurs sous-classes.

Pourquoi avoir des DP comportementaux?

- ▶ Lorsque vous concevez un logiciel, il est important de reconnaître comment vos différents objets travaillent ensemble vers un objectif commun:
 - ▶ Chaque objet que vous faites est un morceau d'une solution plus grande.
 - ▶ Pour que chacun fasse son travail efficacement, il doit avoir un objectif prédéfini.
- ▶ Pensez-y comme chaque personne travaillant dans une entreprise:
 - ▶ Si les gens de l'entreprise n'avaient pas de rôles prédéfinis, il n'y aurait aucun moyen de s'assurer que chacune de leurs fonctions était exécutée..
 - ▶ L'entreprise serait moins efficace et pas aussi organisée.

Cette situation est comme un **DP comportemental (Behavioral DP)**.

- ▶ Les DP du type **Behavioral** se concentrent sur la façon dont les objets indépendants fonctionnent et collaborent en vue d'un objectif commun.
- ▶ Il existe de nombreuses façons pour définir des comportements aux objets.
 - ▶ Dans cette partie du cours, nous verrons différents modèles comportementaux que vous pouvez utiliser pour concevoir des logiciels.

Pour vous donner une meilleure idée de la façon dont les modèles comportementaux peuvent s'appliquer dans le monde réel, examinons quelques exemples.



Example 1

- ▶ Rappelez-vous les instructions pour mélanger une boisson à partir d'ingrédients en poudre d'un produit donné:
 - ▶ Chocolat en poudre
 - ▶ Serelak, etc.
- ▶ Une proposition des étapes à faire inclus:
 1. Obtenez un conteneur.
 2. Vider les ingrédients dans le récipient,
 3. ajouter de l'eau,
 4. remuer et préparer.
- ▶ Entre les différents produits, les étapes sont ordonnées de la quasi même manière, avec des étapes identiques et d'autres légèrement différentes dans la mise en œuvre .



Example 2

- ▶ Vous êtes un chef exécutif dans une grande chaîne de restaurants qui servent des pâtes.
 - ▶ Vous voulez que les plats soient cohérents dans tous les restaurants de la chaîne,
 - ▶ Alors, vous fournissez les instructions pour préparer identiquement chaque plat.
- ▶ Vos deux plats les plus populaires sont:
 1. Spaghetti avec sauce tomate et boules de viandes, et
 2. Penne noodles avec sauce Alfredo (sauce blanche) et poulet.
- ▶ Les deux plats exigent les tâches suivantes:
 1. faire bouillir l'eau,
 2. Cuire les pâtes,
 3. ajouter la sauce,
 4. ajouter la protéine,
 5. et garnir l'assiette.



Example 2 (suite)

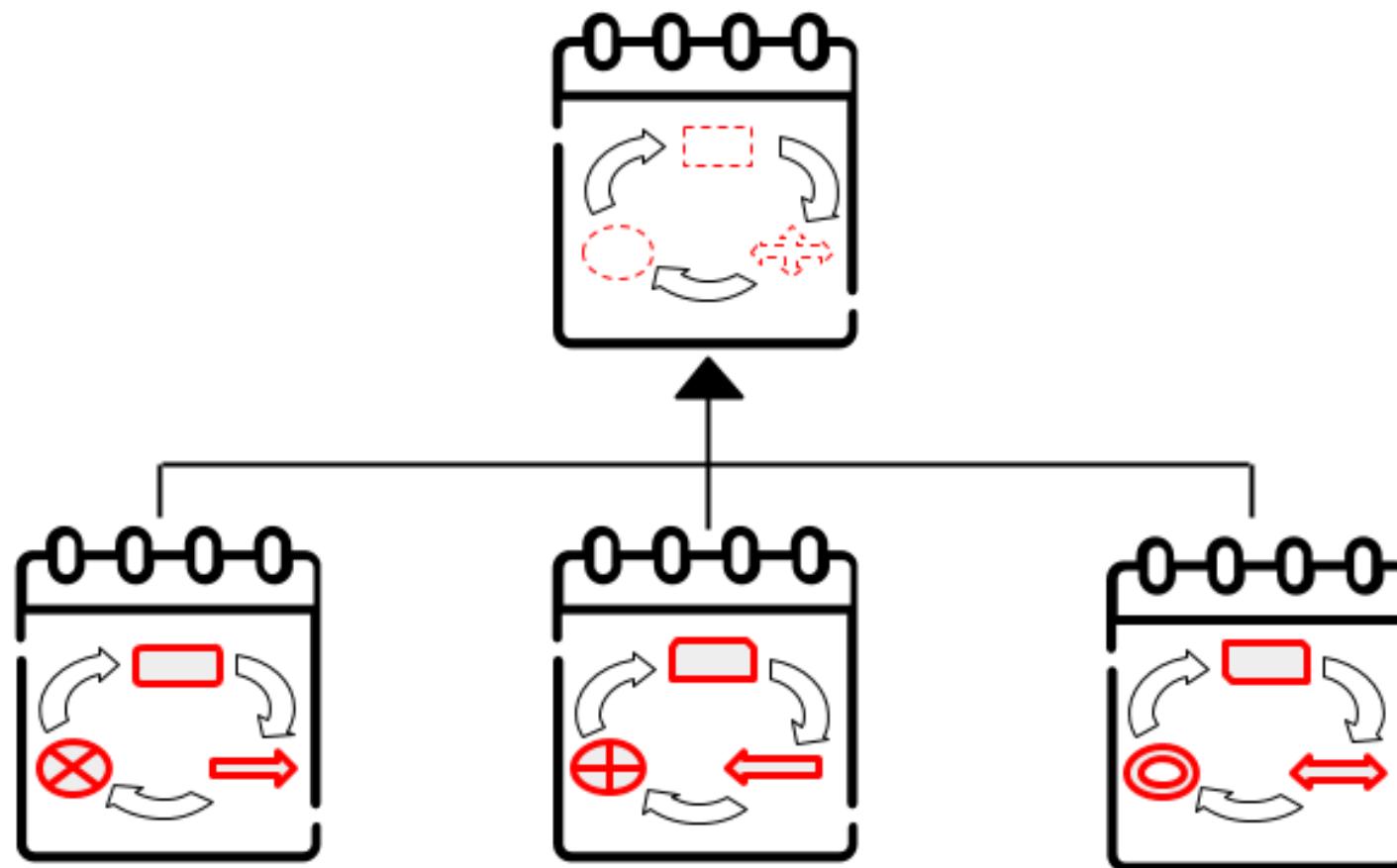
- ▶ Certaines de ces étapes sont mises en œuvre différemment selon le plat que vous faites:
 - ▶ Chaque plat dispose de différentes protéine, sauce, et garniture.
 - ▶ Ainsi, ceux-ci seraient mis en œuvre d'une manière légèrement différentes à chaque fois.
- ▶ D'autres étapes seraient mises en œuvre **de la même manière pour les deux plats.**
 - ▶ Les deux plats ont besoin de faire bouillir de l'eau.
 - ▶ L'eau ne peut être bouillie que de façon singulière quel que soit le type de plat.
- ▶ Vous pouvez modéliser cette situation avec une classe **PastaDish** ayant la méthode **MakeRecipe()**:
 - ▶ Elle sert de recette pour chaque sous-classe: **SpaghettiWithMeatballs** et **PenneAlfredo**.
 - ▶ La méthode **MakeRecipe()** connaît l'ensemble général des étapes pour faire l'un ou l'autre des plats, et ce depuis l'eau bouillante .. Jusqu'à l'ajout de la garniture.
 - ▶ Les étapes qui sont spéciales à un plat, comme le **rajout de la sauce**, sont implémenté dans sa sous-classe.

Tous ces comportements sont des éléments du design pattern **Template-method**.

- ▶ C'est le premier DP que nous verrons ici. Il s'intéresse à l'attribution des responsabilités,
- ▶ Le DP comportemental **Template-method** définit les étapes sommaires d'un algorithme, et délègue l'implémentation de certains étapes aux Sous-classes.

Maintenant, c'est à votre tour de trouver une **Template-method**.





V.1- Le DP Template-method

Question

Maintenant, c'est à votre tour de trouver une **template method**.

- ▶ Nommer un possible **template method** pour une classe de véhicules (voiture/moto) autonomes?

Le nom de votre méthode de modèle de véhicule autonome pourrait éventuellement être **DriveToDestination()**

- ▶ Les deux classes **SelfDrivingCar** et **SelfDrivingMotocycle** auraient une telle fonction pour atteindre leur destination.
- ▶ Ils auraient besoin de passer par toutes les étapes suivantes:
 1. Accélération,
 2. Direction,
 3. Freinage et
 4. Vérification s'il y a une autre destination.



- ▶ Les appels à toutes ces méthodes seront disposés à l'intérieur d'une **template method**.
- ▶ On l'appellerait de la même façon pour les deux sous-classes. Bien que certaines des méthodes appelées diffèrent.
 - ▶ Par exemple, la façon dont vous dirigez une voiture est différente de la façon dont vous dirigez une moto.

Conseils sur les utilisations du DP Template method

- ▶ Le DP template method est le mieux utilisé lorsque vous pouvez **généraliser entre deux classes** dans une nouvelle super classe.
- ▶ Pensez-y comme une autre technique à utiliser lorsque vous remarquez que vous avez **deux classes distinctes (ou plus)** avec des fonctionnalités très similaires **dans l'ordre des opérations**.
- ▶ Après avoir utilisé la généralisation, vous pouvez réutiliser plus efficacement les objets en utilisant **l'héritage**, vous pouvez partager les fonctionnalités entre les classes et permettre un code plus clair et **plus explicite**.

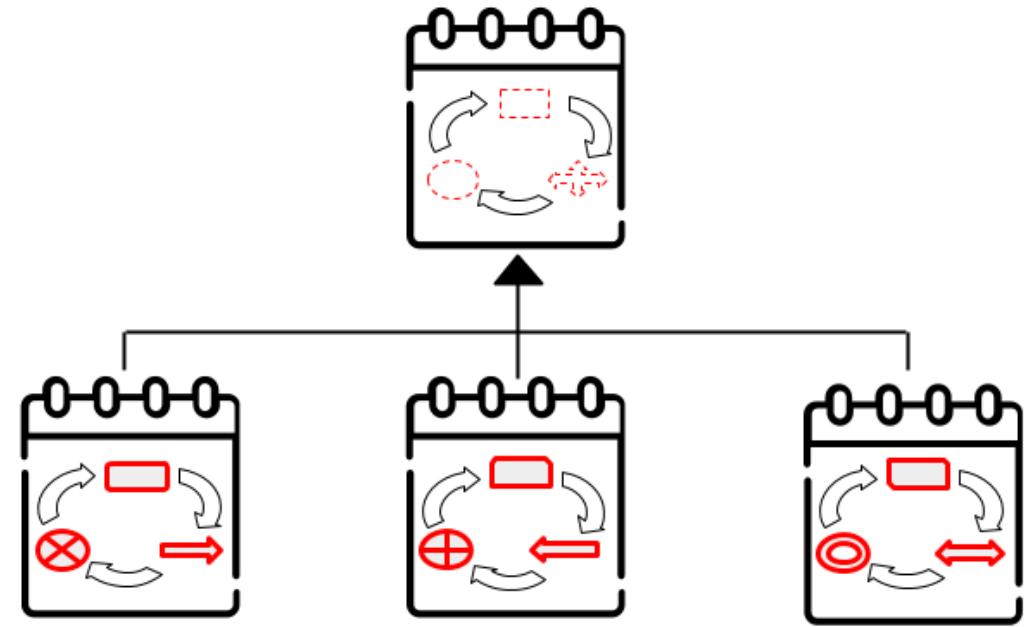
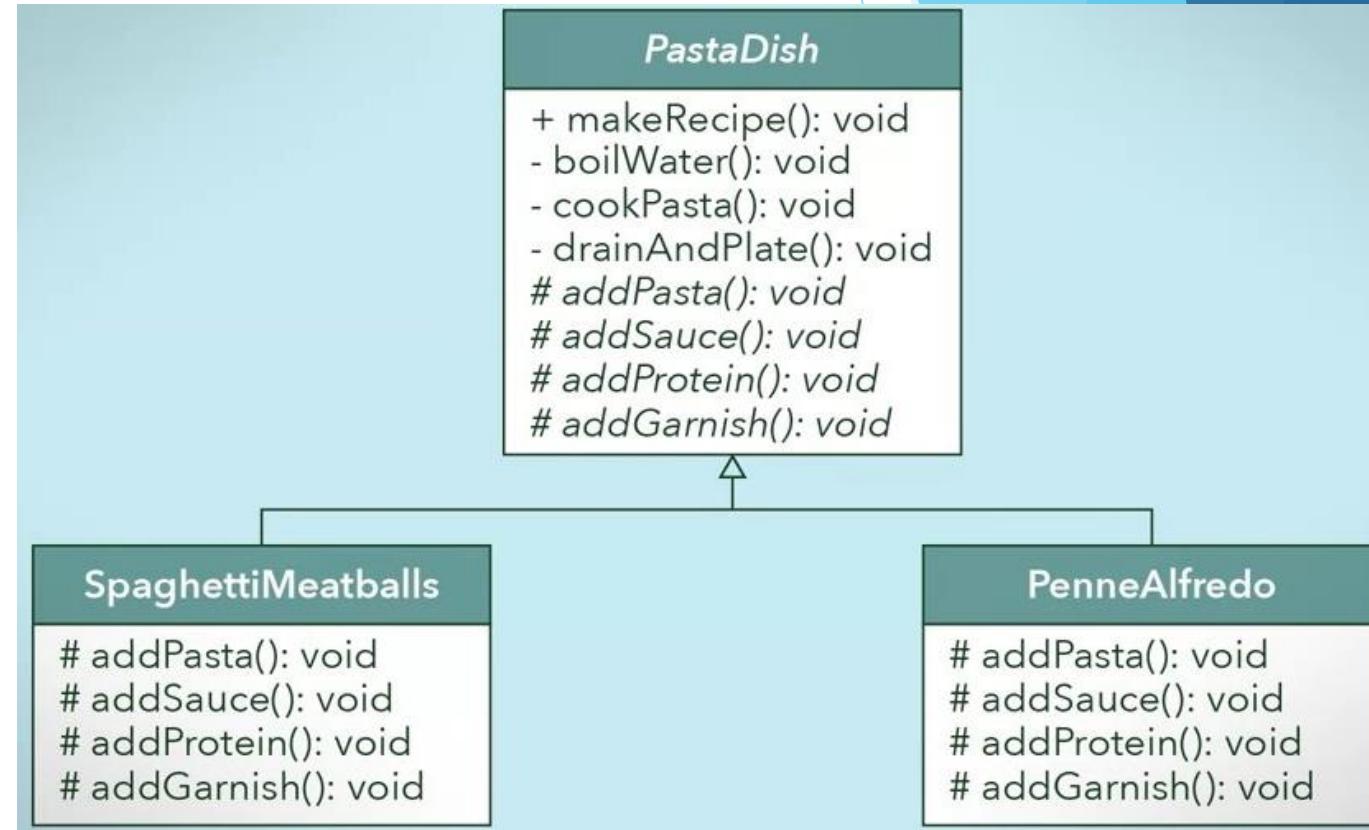


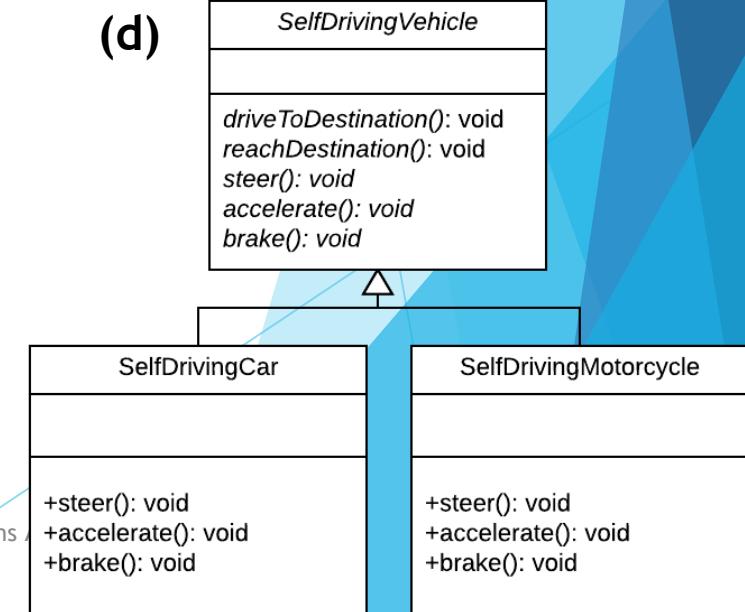
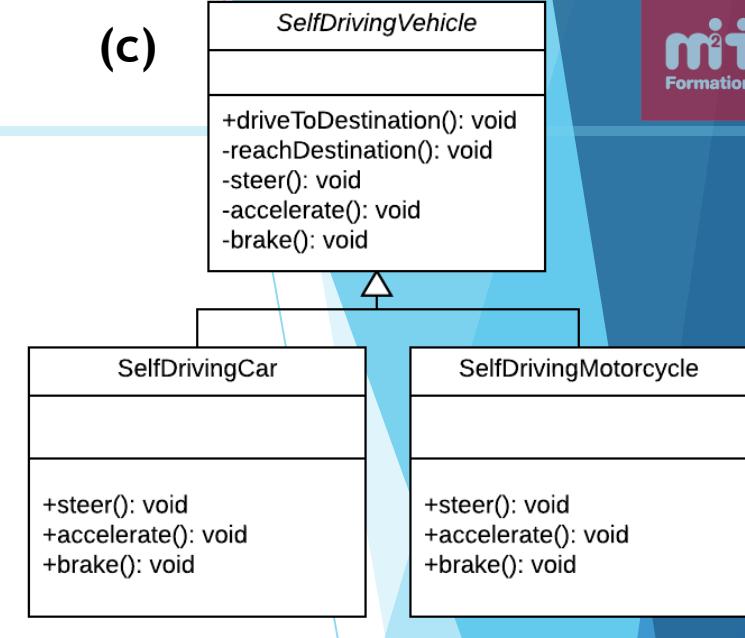
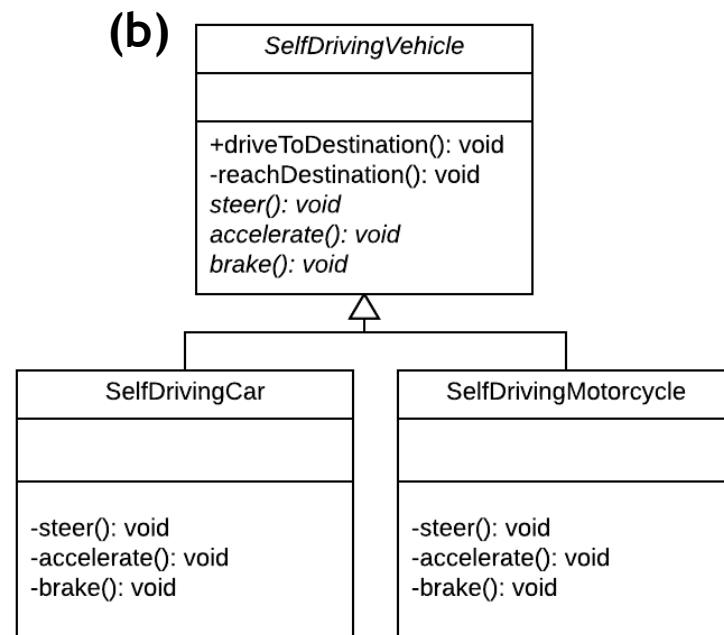
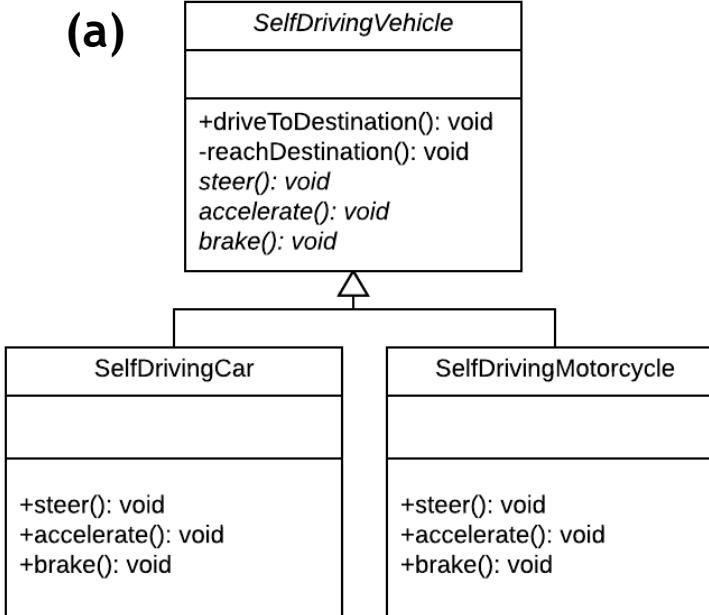
Diagramme de classe UML pour le DP Template Method

- ▶ Revenons à notre exemple de pâtes.
- ▶ Nous avons la superclasse **PastaDish**
 - ▶ Avec la template method, **makeRecipe()** qui appelle d'autres méthodes pour les étapes de la recette.
 - ▶ Certaines étapes sont communes à travers des plats spécifiques comme l'eau bouillante, de sorte que **boilWater()** est une méthode de **PastaDish**.
- ▶ Cependant, certaines étapes sont spécifiques aux plats comme l'ajout de la sauce.
 - ▶ Alors **addSauce** est une **abstract method** dans **PastaDish**, (noter le signe #)
 - ▶ il sera à la sous-classe de plat de pâtes de fournir le corps de la méthode **addSauce()** pour ajouter la bonne sauce.
- ▶ **SpaghettiMeatballs** et **PenneAlfredo** sont des sous-classes de la classe **PastaDish**.
 - ▶ Ils doivent fournir leurs propres versions de :
 - ▶ **addPasta()**, **addSauce()**, **addProtein()**, et **addGarnish()**.



Question d'évaluation uml

- ▶ Choisissez la bonne conception UML pour
“un véhicule autonome basé sur template method”



Code Source pour le DP Template-method

- ▶ Examinons maintenant comment cela pourrait être implémenté dans le code Java:
- ▶ Tout d'abord, examinons la superclasse **PastaDish**

```
public abstract class PastaDish {  
    final void makeRecipe() {  
        boilWater();  
        addPasta();  
        cookPasta();  
        drainAndPlate();  
        addSauce();  
        addProtein();  
        addGarnish();  
    }  
    abstract void addPasta();  
    abstract void addSauce();  
    abstract void addProtein();  
    abstract void addGarnish();  
    private void boilWater() {  
        System.out.println("Boiling water");  
    }  
    //...  
}
```

- ▶ PastaDish doit être une classe abstraite:
 - ▶ Vous ne pouvez pas créer un PastaDish concret,
 - ▶ vous ne sauriez pas quelles pâtes, protéines, sauce, et garniture ajouter.
 - ▶ Donc, ces méthodes sont abstraites.
- ▶ La méthode **makeRecipe()** est notre méthode template.
 - ▶ Vous remarquerez que cette méthode est marquée comme **final**.
 - ▶ . En Java, le mot clé final signifie que la méthode déclarée ne peut pas être surchargée par des sous-classes
 - ▶ Cela signifie qu'aucune sous-classe de plat spécifique ne peut avoir sa propre version de **makeRecipe()**.
 - ▶ Cela assure la cohérence dans les étapes de fabrication des plats et réduit le code redondant.

Code Source pour le DP Template-method (suite)

- ▶ Dans makeRecipe(), les autres méthodes sont appelées.
- ▶ Par exemple, on appelle boilWater()
 - ▶ boilWater est fait la même manière pour toutes les sous-classes.
 - ▶ Et donc sa mise en œuvre est placée dans la superclasse.
- ▶ Les méthodes addPasta, addProtein, addSauce, et addGarnish
 - ▶ vont être laissés aux sous-classes, comme SpaghettiMeatballs or PenneAlfredo, pour les détailler.

```
public abstract class PastaDish {  
    final void makeRecipe() {  
        boilWater();  
        addPasta();  
        cookPasta();  
        drainAndPlate();  
        addSauce();  
        addProtein();  
        addGarnish();  
    }  
    abstract void addPasta();  
    abstract void addSauce();  
    abstract void addProtein();  
    abstract void addGarnish();  
    private void boilWater() {  
        System.out.println("Boiling water");  
    }  
    //...  
}
```

Exercice:

- ▶ Avant de continuer, il est temps pour écrire une superclasse de type Template method.
- ▶ Créez la superclasse pour un véhicule autonome: _____
- ▶ Votre réponse peut différer légèrement de la solution décrite ci-dessous. Mais dans globalement, voici à quoi votre classe de véhicule autonome devrait ressembler:

```
public abstract class SelfDrivingVehicle {  
  
    public final void driveToDestination() {  
        accelerate();  
        steer();  
        brake();  
        reachDestination();  
    }  
  
    abstract void accelerate();  
  
    abstract void steer();  
  
    abstract void brake();  
  
    void reachDestination() {  
        System.out.println("Made it to destination.");  
    }  
}
```

- ▶ Votre réponse doit contenir 2 composantes principales:
 1. Une Template method qui offre différentes étapes: **driveToDestination()** dans ce cas
 2. Les étapes définies par les sous-classes (accelerate, steer, et brake, ici) sont définies de façon **abstraite**!
- ▶ Votre réponse peut aussi avoir quelques étapes qui sont communes à toutes les sous-classes. Celles-ci seront alors définies dans la superclasse: **reachDestination()** dans cet exemple

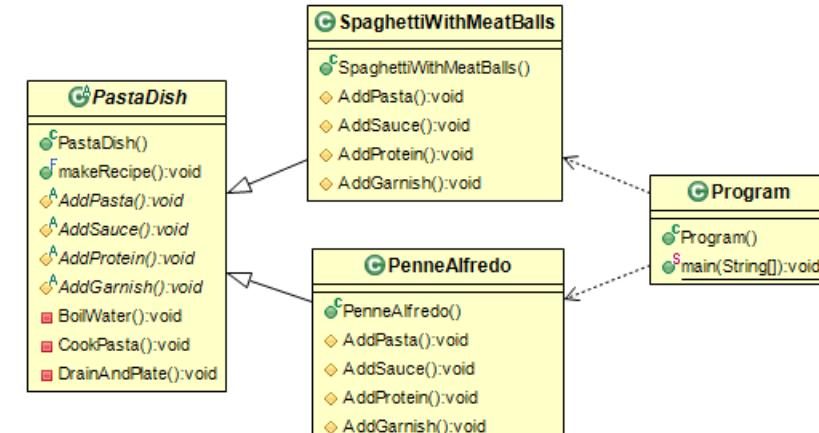
Sous-classes de notre Template-method

- ▶ A quoi ressemblent nos sous-classes **SpaghettiMeatballs** et **PenneAlfredo** :
 - ▶ Les deux étendent la classe **PastaDish**.
 - ▶ Elles doivent chacune impémenter les méthodes abstraites: **addPasta**, **addProtein**, **addSauce**, et **addGarnish** pour fournir les ingrédients appropriés au plat.
- ▶ Les méthodes **addPasta**, **addProtein**, **addSauce** et **addGarnish** sont appelées dans **makeRecipe()** template method sous la superclasse **PastaDish**.
- ▶ La template method est héritée dans les sous-classes et se comporte comme prévu pour réaliser un plat avec les bons ingrédients.

[Voir exemple de Template DP sous Eclipse](#)

```
public class SpaghettiMeatballs extends PastaDish {
    protected void addPasta() {
        System.out.println("Add spaghetti");
    }
    protected void addProtein() {
        System.out.println("Add meatballs");
    }
    protected void addSauce() {
        System.out.println("Add tomato sauce");
    }
    protected void addGarnish() {
        System.out.println("Add Parmesan cheese");
    }
}

public class PenneAlfredo extends PastaDish {
    protected void addPasta() {
        System.out.println("Add penne");
    }
    protected void addProtein() {
        System.out.println("Add chicken");
    }
    protected void addSauce() {
        System.out.println("Add Alfredo sauce");
    }
    protected void addGarnish() {
        System.out.println("Add parsley");
    }
}
```



Exercice: Essayez d'implémenter vos sous-classes

Maintenant, vous pouvez essayer de mettre en œuvre vos sous-classes de template method.

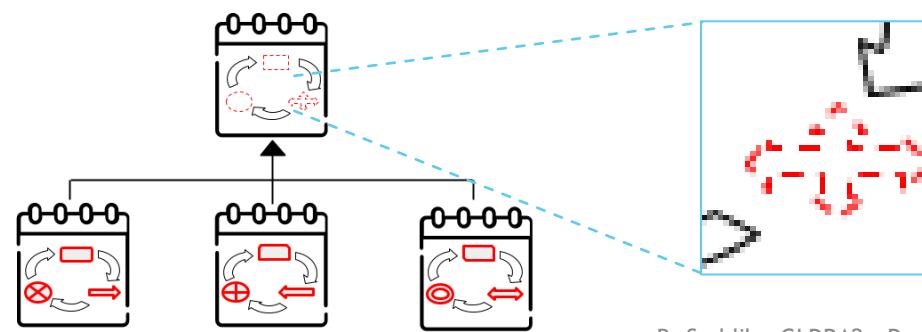
- ▶ Ecrire l'une des sous-classes qui met en œuvre les méthodes abstraites que nous avons décris dans la superclasse **SelfDrivingVehicle**:

```
public class SelfDrivingMotorcycle extends SelfDrivingVehicle {  
    public void steer() {  
        System.out.println("Turning handlebars");  
    }  
  
    public void brake() {  
        System.out.println("Pull brake levers");  
    }  
  
    public void accelerate() {  
        System.out.println("Twist the throttle");  
    }  
}  
  
public class SelfDrivingCar extends SelfDrivingVehicle {  
    public void steer() {  
        System.out.println("Turning steering wheel");  
    }  
  
    public void brake() {  
        System.out.println("Push brake pedal");  
    }  
  
    public void accelerate() {  
        System.out.println("Push the accelerating pedal");  
    }  
}
```

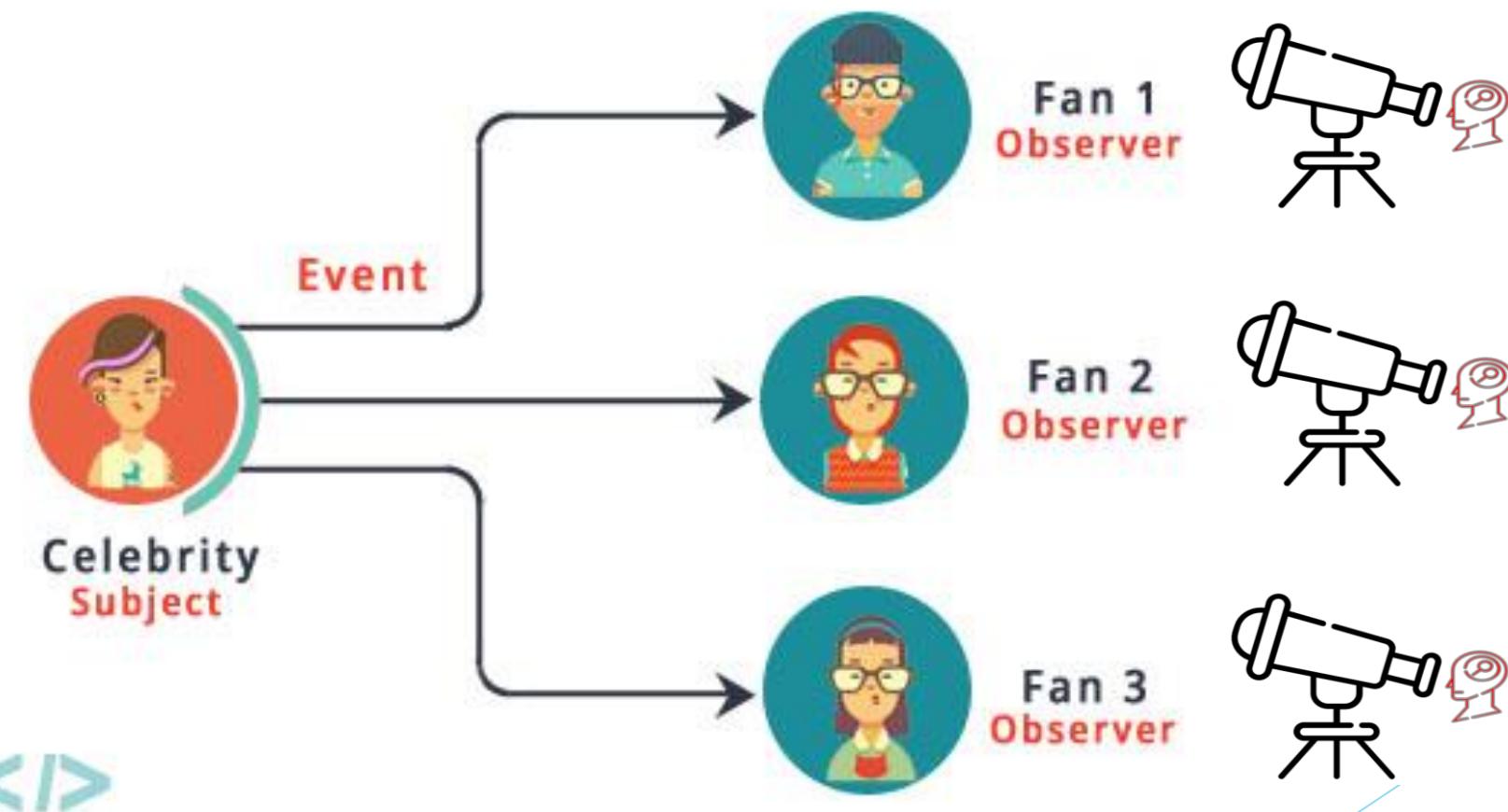
- ▶ Dans cette question, vous devez mettre en œuvre le bouillon, freiner et accélérer les méthodes pour les sous-classes.
- ▶ Deux sous-classes possibles sont les **SelfDrivingCar** et **SelfDrivingMotorcycle**.
 - ▶ Pour chacune de ces sous-classes, vous devez implémenter les méthodes abstraites, de façons assez différentes.

Points à retenir pour le DP Template-method

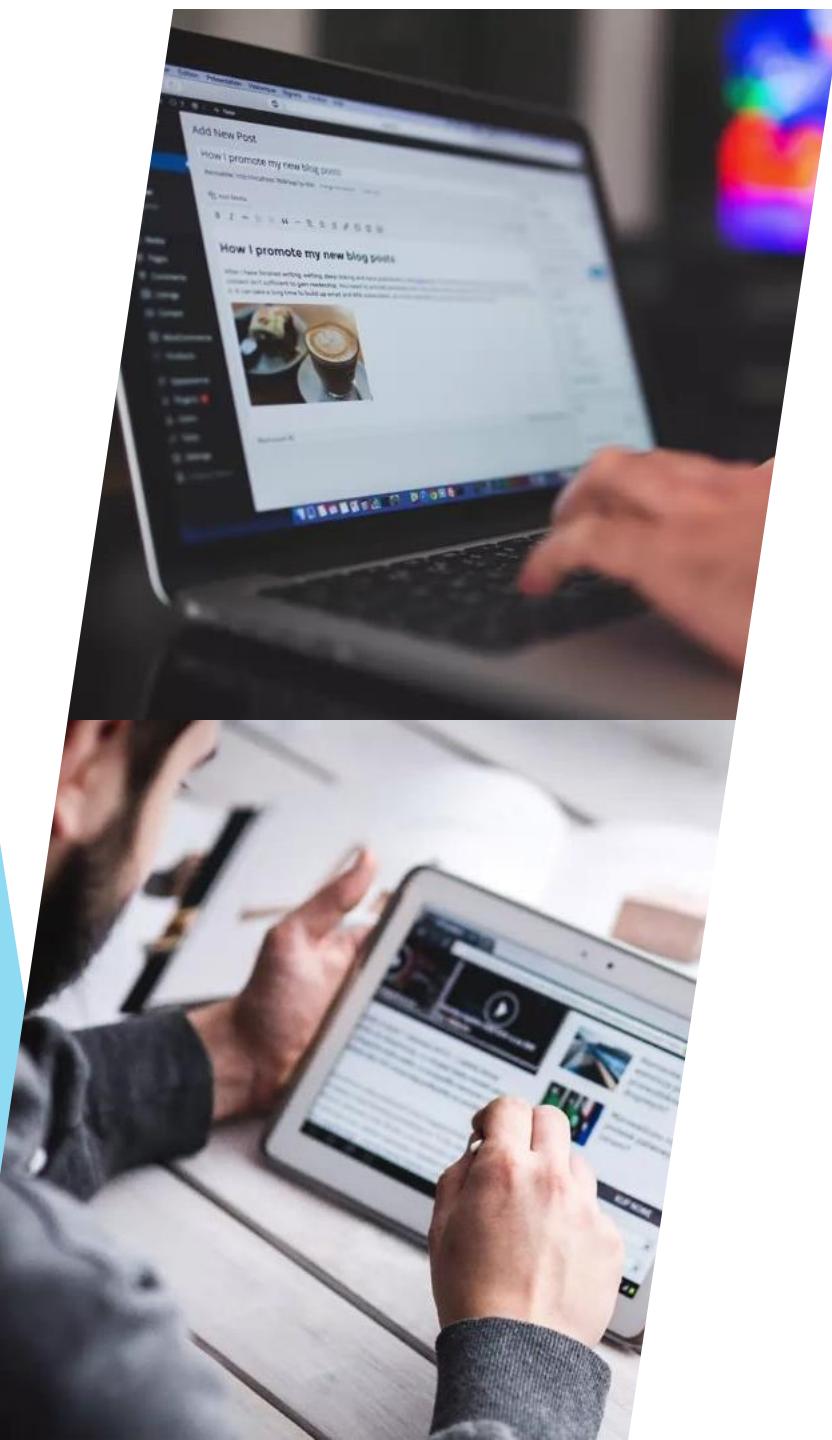
- ▶ Le DP Template method est une application pratique de la **généralisation** et de **l'héritage**.
 - ▶ peut être utile si vous avez deux classes avec des fonctionnalités similaires.
 - ▶ Lorsque vous remarquez deux classes avec un ordre d'opérations très similaire, vous pouvez y penser.
- ▶ Lors de l'écriture d'un logiciel, vous remarquerez peut-être deux classes distinctes qui partagent des similitudes de méthodes avec un algorithme assez similaire.
 - ▶ Plutôt que d'apporter des modifications à ces algorithmes en deux endroits, vous pouvez **consolider** les algorithmes dans un seul endroit dans une méthode Template sous une superclasse.
 - ▶ Vous **généralisez** depuis deux méthodes distinctes en une seule méthode Template au sein d'une superclasse qui sera héritée par les deux classes.



V.2- Le DP Observer



Le besoin pour un Observateur

- 
- ▶ Imaginez que vous avez un blog préféré.
 - ▶ Chaque jour, vous le visitez plusieurs fois par jour pour vérifier les nouveaux billets de blog.
 - ▶ Après un certain temps, vous en avez assez de cette routine. Il doit y avoir une meilleure pour les mises à jour.
 - 1. **La première solution** qui vous traverse l'esprit est d'écrire un script pour vérifier le blog pour de nouveaux messages chaque fraction de seconde.
 - ▶ vous vous rendez compte que la plupart des sites n'apprécieraient pas ce bourrage de demandes, et bloquerait votre adresse IP.
 - ▶ Pour éviter cela, vous écrivez plutôt un script pour vérifier le blog une fois par heure.
 - 2. **Mais à votre grande déception**, cela signifie que maintenant vous passez à côté des post intéressants donnant les meilleures nouveautés .
 - 3. **Une meilleure solution à ce problème est :**
 - ▶ C'est au blog de vous avertir chaque fois qu'un nouveau poste est ajouté.
 - ▶ **Vous vous abonnez au blog**, chaque fois qu'un nouveau message a été publié, le blog informerait chaque abonné, y compris vous.

C'est beaucoup mieux ainsi, pour tous les tiers.

Ce qui est nécessaire pour un DP Observer

- ▶ Le sujet dans notre exemple: le **Blog**, garderait une **liste de Observers (observateurs)**.
 - ▶ Nous pourrions considérer ces observateurs comme des abonnés au blog.
 - ▶ Les observateurs comptent sur le blog pour les informer de tout changement, exp.: Si un nouveau poste est ajouté.

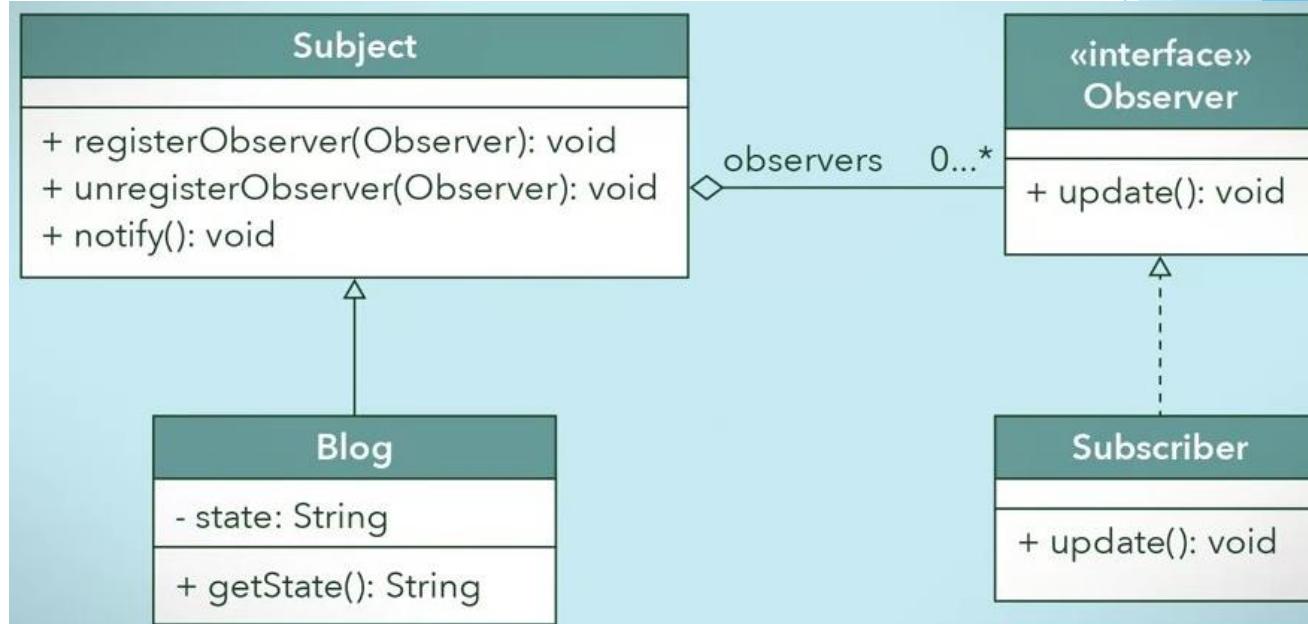
Explorons comment nous pourrions appliquer cette idée à notre exemple:

- ▶ Tout d'abord, nous aurons une superclasse **Subject** qui définit trois méthodes
 1. permettre à un nouvel observateur de s'abonner,
 2. permettre à un observateur actuel de se désabonner,
 3. et notifier tous les observateurs d'un nouveau blog.
- ▶ Cette superclasse pourrait également :
 - ▶ Disposer d'une variable (un attribut) pour garder trace de tous les observateurs (tableau, liste, ...),

Diagramme de classe UML pour le DP Observer

- ▶ La superclasse **Subject** a 3 méthodes: **registerObserver**, **unregisterObserver**, et **notify**.
 - ▶ Tout cela est essentiel pour qu'un sujet se relie à ses observateurs.
 - ▶ La sous-classe **Blog** hériterait de ces méthodes pour abonner, désabonner et notifier ses souscrits.
- ▶ L'interface **Observer** possède seulement la méthode **update()**.
 - ▶ Un observateur doit toujours avoir un moyen de se mettre à jour, dès qu'il est notifié, il se met à jour.
 - ▶ La classe **Subscriber** implemente l'interface **Observer**, en donnant le corps de la méthode **update()**.

Notez qu'un sujet peut avoir zéro observateurs ou plus enregistrés à un moment donné.



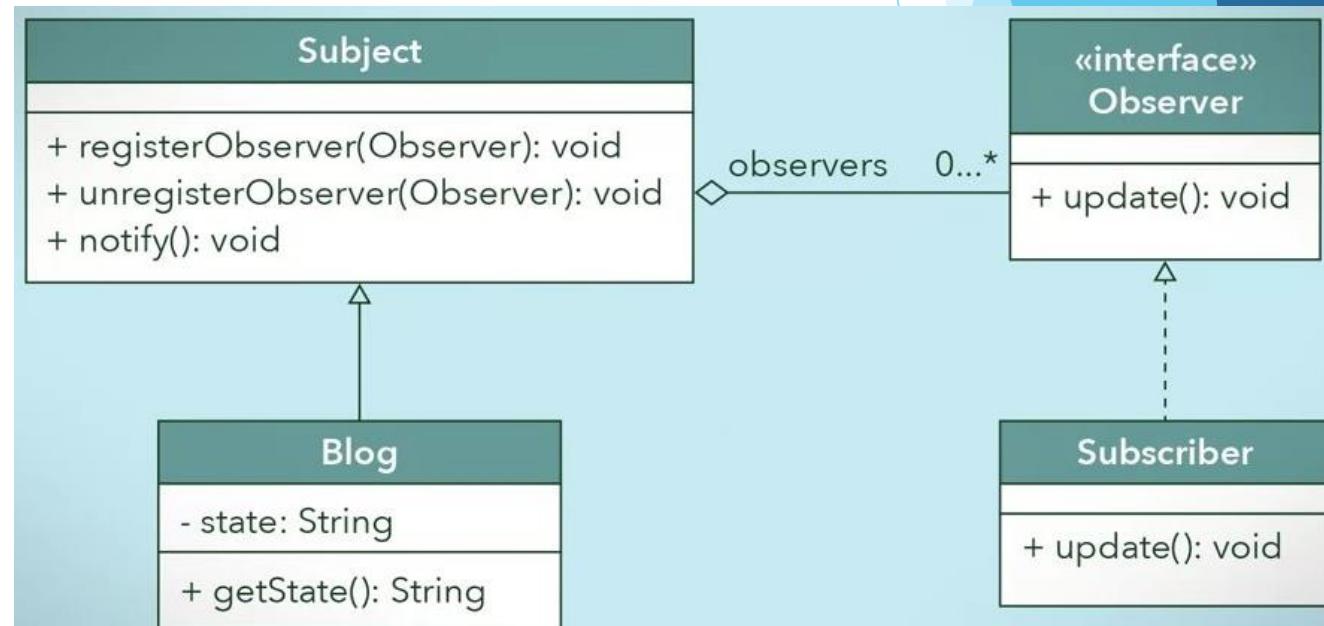
Observer VS Observable

- ▶ Les éléments de conception essentiels à la formation d'une relation entre Subject (sujet observé) ou **Observable**

↔

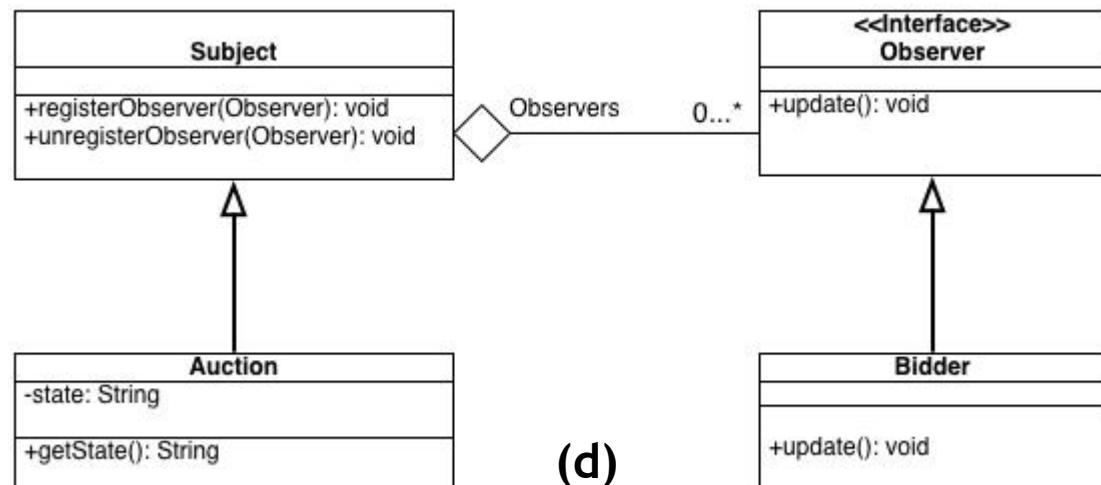
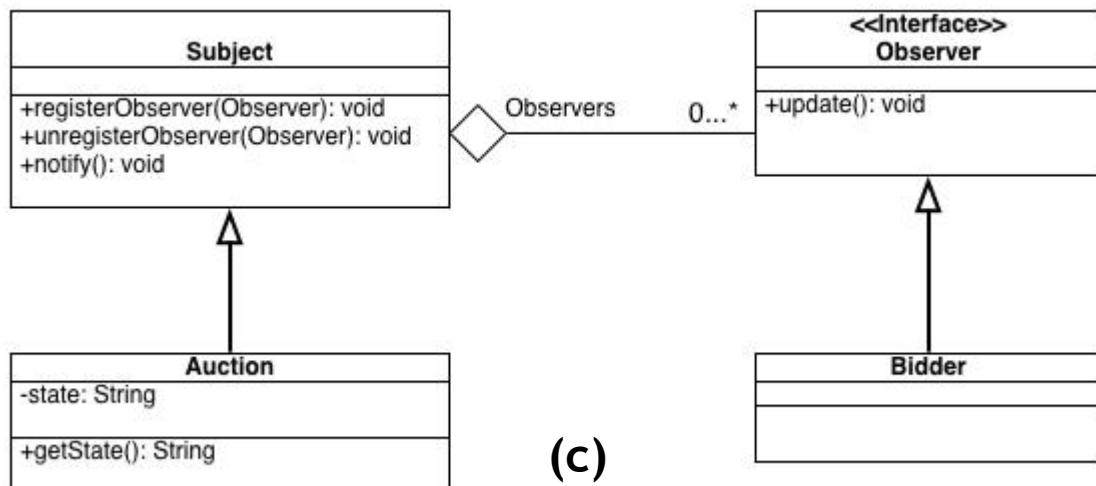
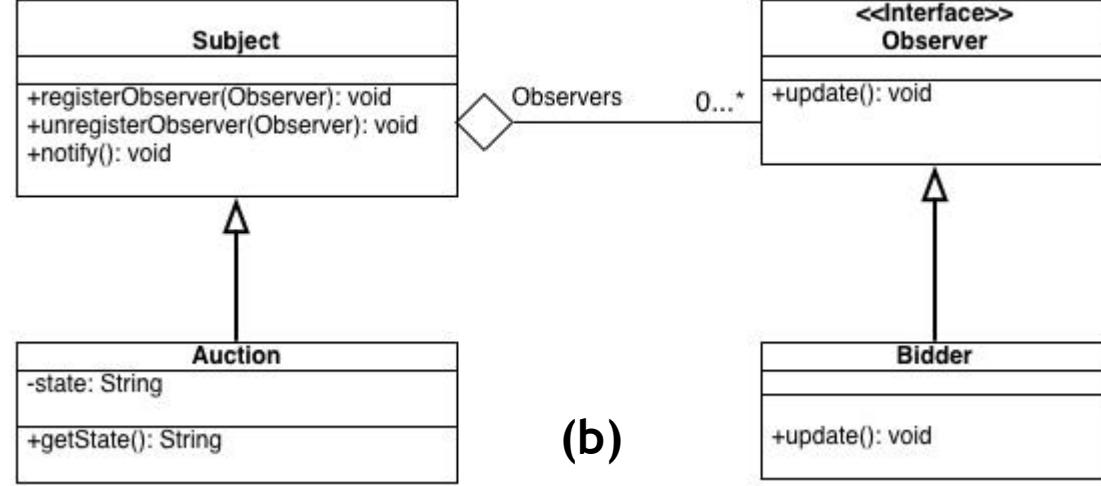
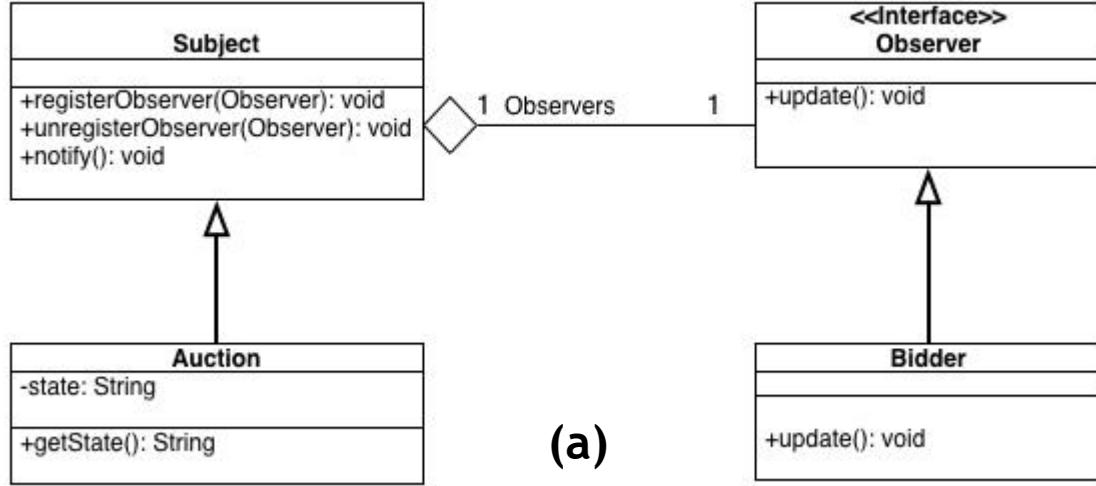
- ▶ **Observer** (nos observateurs abonnés)

- ▶ Cette logique est omniprésente
- ▶ Exploitable via les classes Java prédéfinies
 - ▶ Observer
 - ▶ Observable
- ▶ @Deprecated(since="9"):
 - ▶ Possibilité d'utiliser les Listeners
 - ▶ Même logique: s'abonner et être notifié pour se mettre à jour



DP Observer: Question 2

- Choisissez l'UML correcte du DP Observer entre une vente aux enchères (**Auction**) et les enchérisseurs (**Bidder**)



Observer DP: Question 1

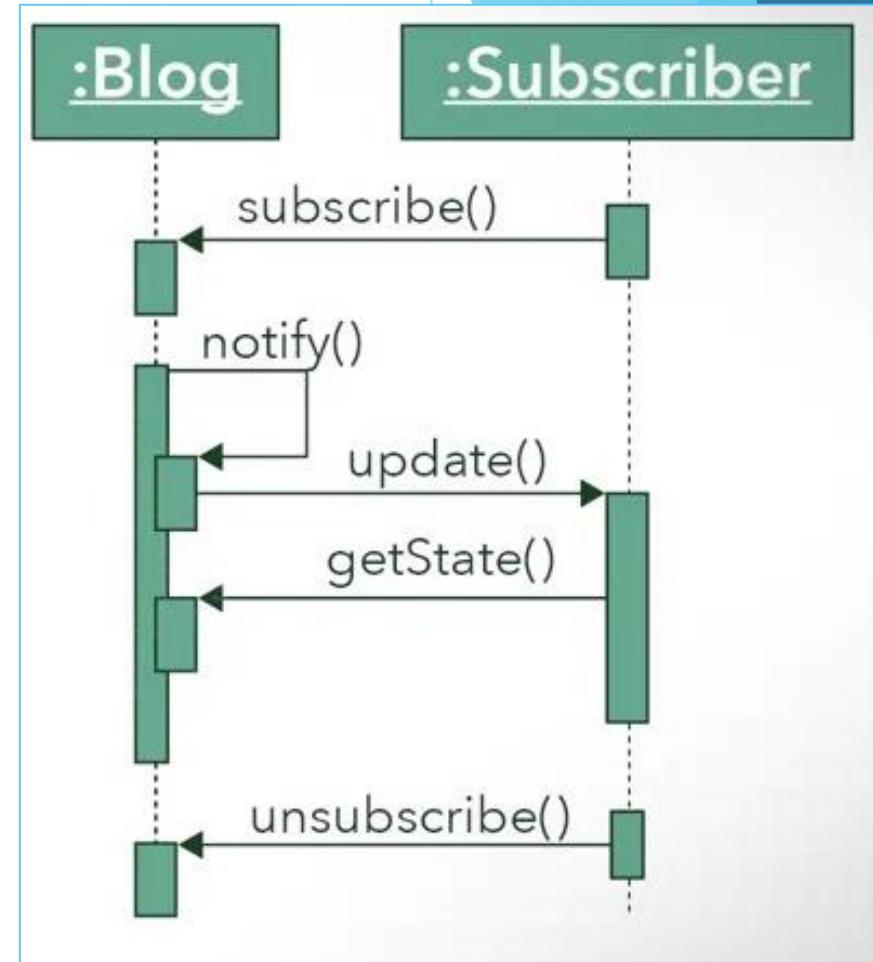
Trouvez un possible cas de **Subject** et **Observer** selon vous:

- ▶ Il existe de nombreux autres exemples possibles, mais vous auriez pu dire:
 - ▶ une vente aux enchères (**Auction**) et un enchérisseur (**Bidder**),
 - ▶ news **Broadcast** et **Viewer**.
- ▶ Dans chacun d'eux, les relations sont généralement 1 Sujet vers * observateurs
 - ▶ Une vente aux enchères est observée par de nombreux enchérisseur.
 - ▶ Une émission est observée par de nombreux téléspectateurs.
- ▶ D'autres exemples possibles peuvent être:
 - ▶ Une **personnalité célèbre** qui fait des mises à jour sur le web d'un côté, puis de l'autre, **ses fans**,
 - ▶ Une **planète** qui se déplace dans le ciel, et de l'autre coté sur terre, les **scientifiques** qui suivent tout changement, etc.



Diagramme de séquence d'un DP Observer

- ▶ Dans un diagramme de séquence qui applique le DP Observer, il y a deux objets majeurs, le Subject (**:Blog**) et le Observer (**:Subscriber**)
 1. Un Subscriber doit appeler() **subscribe()** sur le Blog.
 2. Le blog doit être en mesure de **notify()** ses abonnés des changements
 - ▶ C'est le travail de la fonction de notification de garder tous les abonnés informés.
 - ▶ Cette méthode n'est appelée que lorsque le blog a un changement,
 3. il est temps pour le blog d'informer ses abonnés par le biais d'un appel de **update()**, et les abonnés obtiennent l'état du blog à travers un appel de **getState()**.
 4. Si l'abonné ne se retrouve pas à profiter du contenu du blog, ils auraient besoin d'un moyen d'appeler **unsubscribe()**
 - ▶ Il provient de l'abonné, et permet au blog de savoir que l'abonné doit être retiré de sa liste d'abonnés.



Exemple de Code Source pour le DP Observer

- ▶ La méthode **registerObserver** ajoute un observateur à la liste des observateurs.
- ▶ La méthode **unregisterObserver** supprime un observateur dans la liste,
- ▶ La méthode **notify** appelle **update()** pour chaque observateur dans la liste
- ▶ La classe **Blog** est une sous-classe de **Subject**. Elle hérite de **registerObserver()**, **unregisterObserver()** et **notify()** et dispose des autres responsabilités pour gérer le blog et de poster les messages

```
public class Subject {  
    private ArrayList<Observer> observers = new ArrayList<Observer>();  
  
    public void registerObserver(Observer observer) {  
        observers.add(observer);  
    }  
    public void unregisterObserver(Observer observer) {  
        observers.remove(observer);  
    }  
    public void notify() {  
        for (Observer o : observers) {  
            o.update();  
        }  
    }  
}
```

```
public class Blog extends Subject {  
  
    private String state;  
  
    public String getState() {  
        return state;  
    }  
  
    // blog responsibilities  
    ...  
}
```

DP Observer: Question 3

- ▶ Pensez (oralement) au code source pour la sous-classe **Auction** de la superclasse **Subject** :
 - ▶ Cette classe héritera de :
 - ▶ registerObserver(),
 - ▶ unregisterObserver(),
 - ▶ Et notify (),
 - ▶ et possède ses propres responsabilités pour la gestion de la vente aux enchères.

```
public class Auction extends Subject {  
  
    private String state;  
  
    public String getState() {  
        return state;  
    }  
  
    // auction responsibilities  
    ...  
}
```

Code implementation for Observer DP (suite)

- ▶ The **Observer** interface makes sure all observer objects behave the same way.
 - ▶ Observer classes only need to implement a single method, **update()**.
 - ▶ The update method is called by the subject.
 - ▶ The subject makes sure when a change happens, all its observers are notified to update themselves
- ▶ More specifically, for our example, we have a class **Subscriber** that implements the observer interface.
 - ▶ This update method is called, "when the blog notifies the subscriber of a change.

```
public interface Observer {  
    public void update();  
}
```

```
public class Subscriber implements  
Observer {  
  
    public void update() {  
        // get the blog change  
        ...  
    }  
}
```

Question 4

- ▶ Pensez (oralement) au code source permettant d'implémenter l'interface **Observer** sous la classe **Bidder** observatrice
 - ▶ La classe **Bidder** doit seulement implémenter la méthode `update()`.
 - ▶ La méthode `Update()`
 - ▶ s'assure que le soumissionnaire est informé que l'état de la vente aux enchères a changé.
 - ▶ est appelé de l'intérieur de **Auction**, sous la méthode `notify()` héritée de **Subject**.

```
public class Bidder implements  
Observer {  
  
    public void update(){  
        // make a bid  
        ...  
    }  
}
```

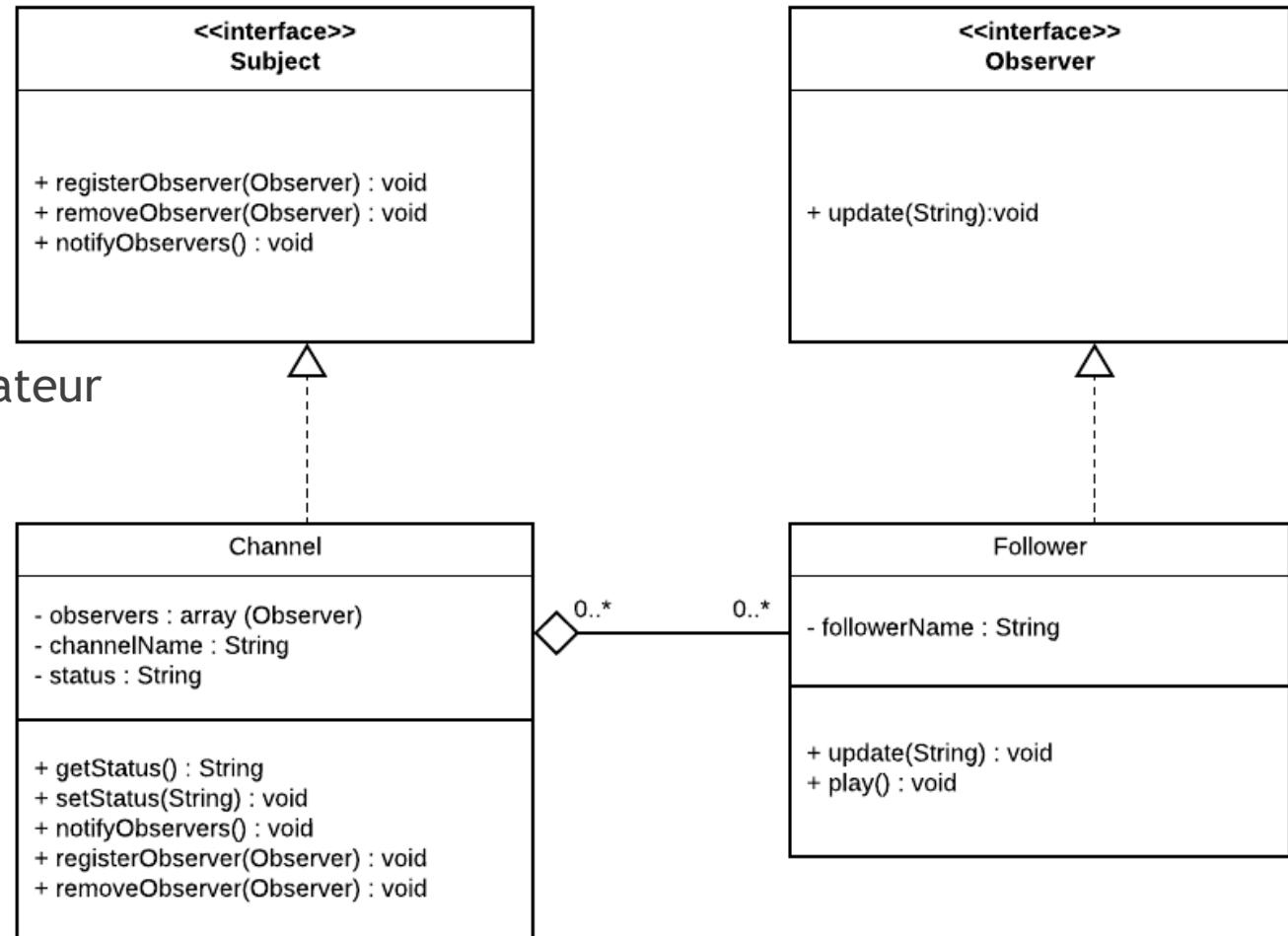
Quelles sont les exigences minimales du modèle Observateur?

Choisissez les trois réponses qui sont correctes.

- A. méthode pour notifier les observateurs
- B. méthode Update() chez les observateurs
- C. méthodes pour l'ajout ou de suppression d'observateurs
- D. une variable d'état pour déterminer si les observateurs ont été avisés.

Exercice à coder sous Eclipse: Youtube Observer

- ▶ **Youtube** permet aux utilisateurs de :
 - ▶ suivre leur stars favorites,
 - ▶ de les notifier lorsque la chaîne est en direct.
- ▶ Cela se fait sur terrain à l'aide du DP observateur
- ▶ Proposez les codes sources en vous guidant par le diagramme de classe UML suivant :



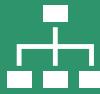
Les points à retenir depuis le DP Observer



Le DP **Observer** permet de gagner du temps lors de la mise en œuvre d'un système.



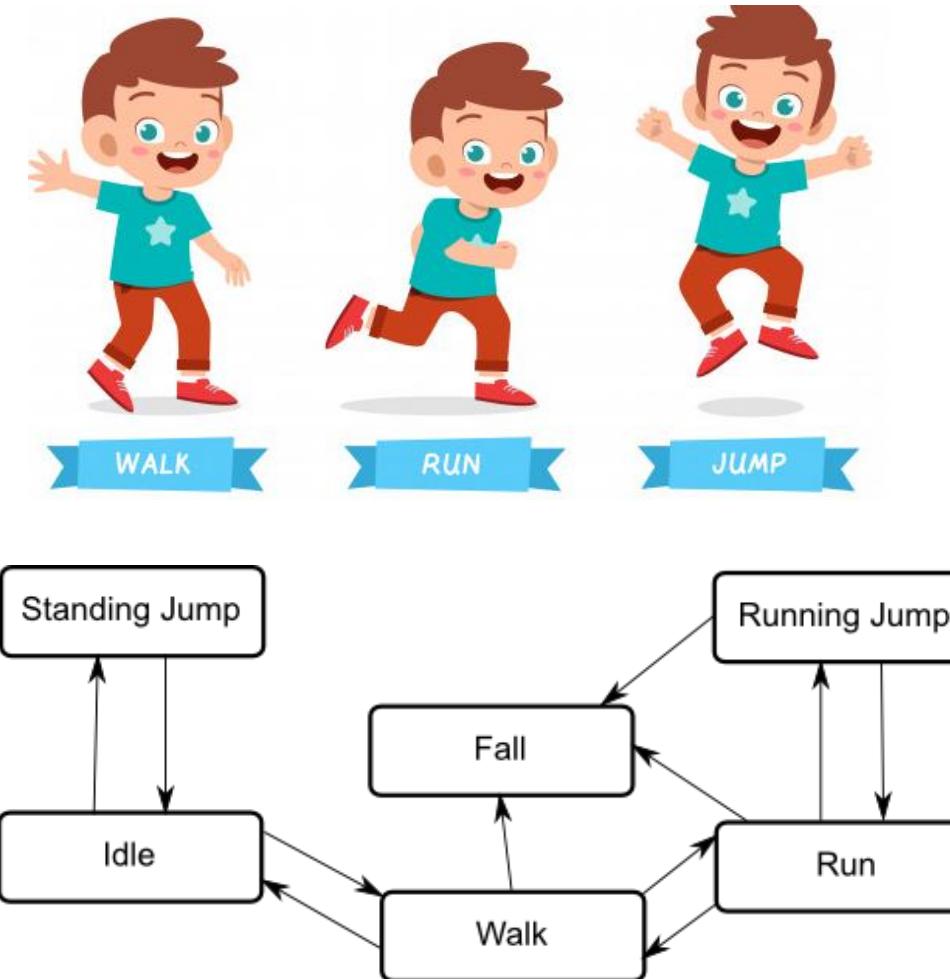
Si de nombreux objets reposent sur l'état d'un, le DP **Observer** a encore plus de valeur.



Au lieu de gérer tous les objets d'observation individuellement, le sujet les gère et s'assure que les observateurs se mettent à jour quand il y a besoin..



Ce DP de comportement est très utilisé pour faciliter la distribution et la gestion des notifications sur les modifications entre les systèmes d'une manière efficace.



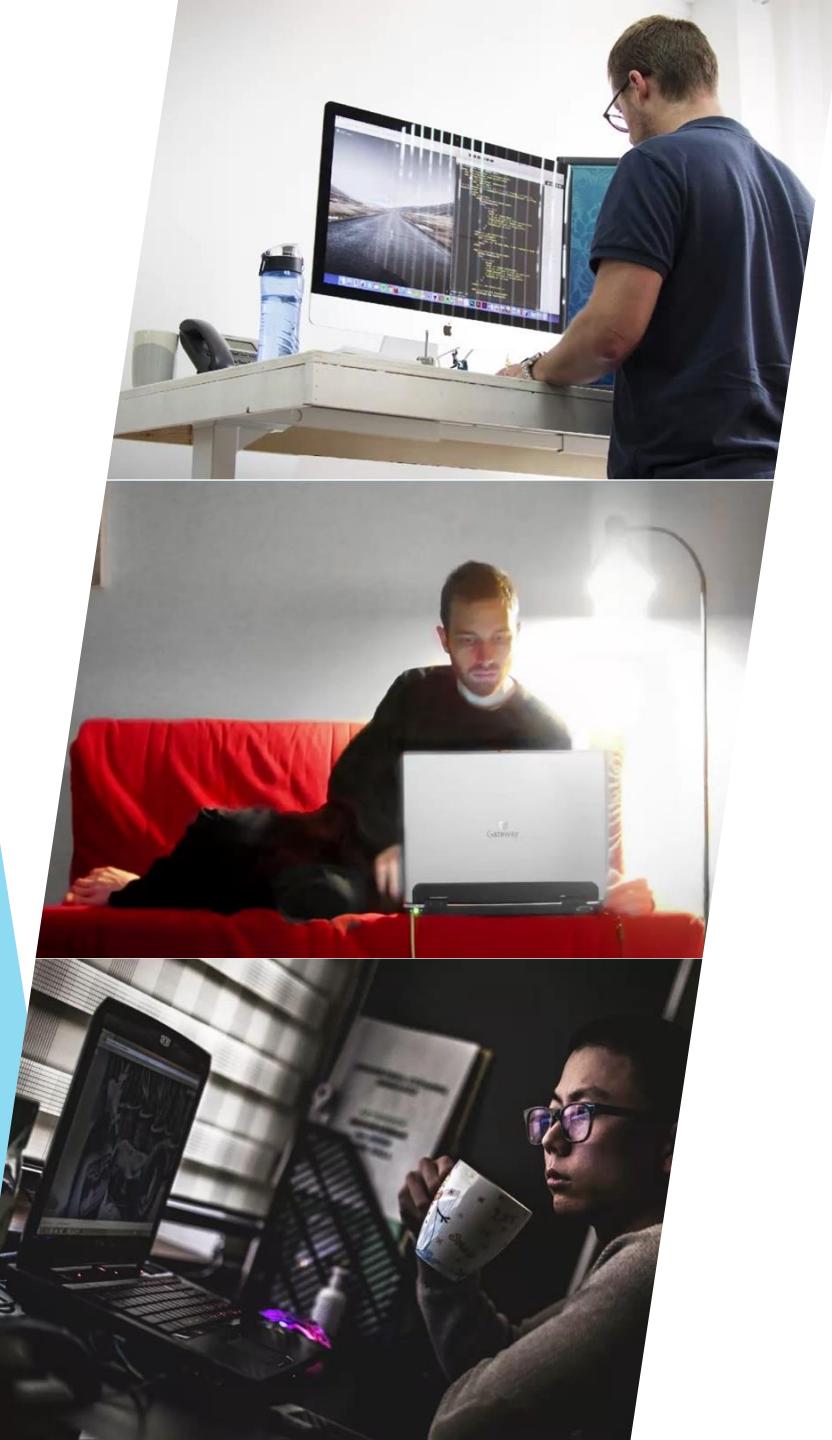
V.3-le DP State

Transiter depuis un état vers un autre de comportement différent

Intro: C'est quoi votre Etat?

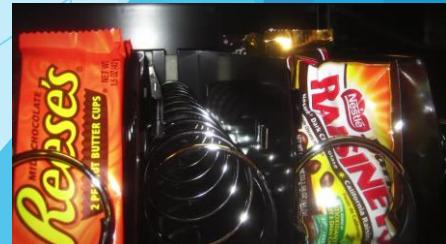
- ▶ Pensez à vous-même lorsque vous êtes devant votre PC à la maison:
 - ▶ Quel est votre état lorsque vous faites cette activité?
 - ▶ debout
 - ▶ allongé(e) sur canapé
 - ▶ Assis(es) sur le bureau, etc.
 - ▶ Si vous où on vous demande d'assister à une session de tutorat en ligne (comme la notre!), alors vous devriez changer votre état pour être plus en mesure de répondre à cette action
- ▶ Dans le DP **State**, puisque les objets de votre code sont **conscients de leur état actuel**, vous pouvez exploiter cette hypothèse dans votre code.
 - ▶ Ils peuvent choisir un comportement approprié en fonction de leur état actuel.
 - ▶ Lorsque leur état actuel change, ce comportement peut être modifié.

Le modèle d'état est principalement utilisé lorsque vous devez modifier le comportement d'un objet en fonction de l'état dans lequel il se trouve en cours d'exécution.



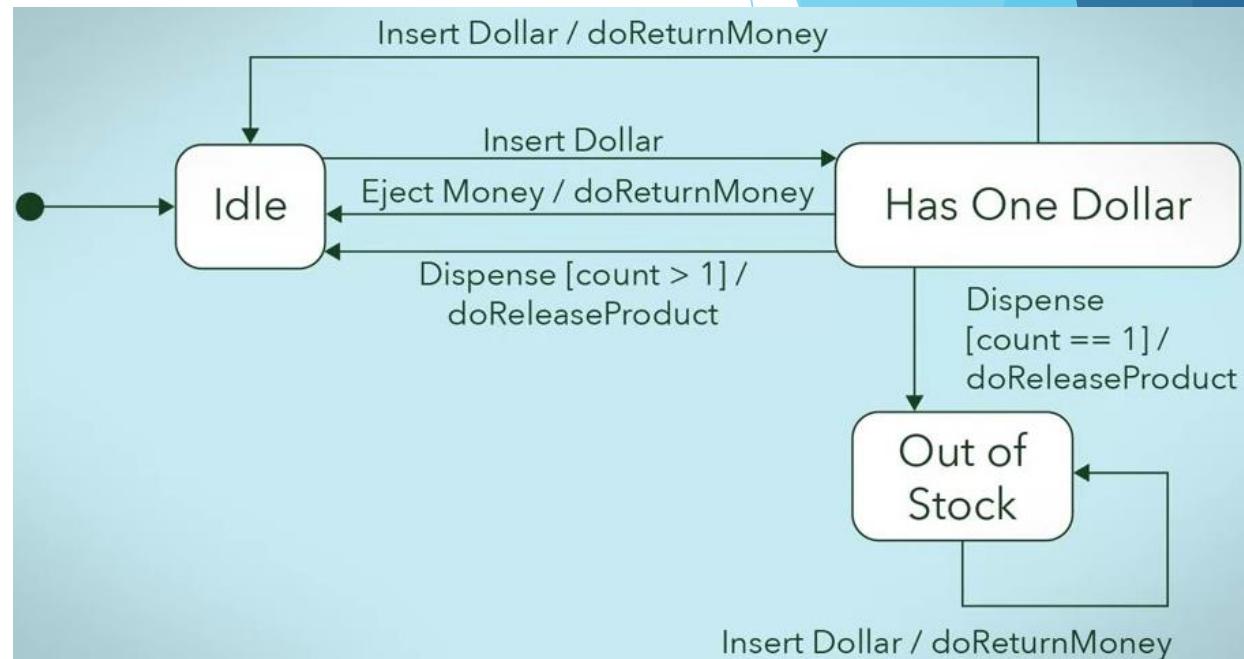
Example: Vending machine

- ▶ Un distributeur automatique représente très bien le DP **State**, car il possède plusieurs états et des actions spécifiques basées sur ces états.
- ▶ Considérez que vous voulez acheter une barre de chocolat à partir d'un distributeur automatique.
 - ▶ La barre de chocolat coûte 2 Dollars (Euros).
 - ▶ Vous marchez jusqu'au distributeur automatique, insérez l'argent et faites votre sélection.
 - ▶ La machine distribue ensuite la barre de chocolat. **Tout va bien!**
- ▶ Mais, que faire si vous avez inséré votre premier morceau d'argent, puis avez décidé que vous ne voulez plus de la barre de chocolat?
 - ▶ Vous appuieriez sur le bouton d'argent "**EJECT**" et la machine retournerait l'argent.
- ▶ Que se passe-t-il si le distributeur est à court de tablettes de chocolat?
 - ▶ Il doit garder une trace de son inventaire et informer les clients quand il n'y a plus de produit particulier à vendre.



UML State diagram for the State DP

- ▶ À partir de ce diagramme, vous pouvez voir qu'il y a trois états:
 - ▶ **Idle**, **HasOneDollar**, et **OutOfStock**.
 - ▶ Le distributeur automatique est dans un état bien particulier à un instant donné.
- ▶ Vous pouvez également noter qu'il y a trois déclencheurs (**Triggers**), ou événements (**Events**) qui sont réalisables par l'usager:
 - ▶ **insert dollar**, lorsque le dollar est inséré,
 - ▶ **eject money**, lorsqu'il y a une demande d'éjection d'argent,
 - ▶ et **dispense**, lorsqu'il y a une demande pour recevoir un produit en sortie de la machine.
- ▶ Il y a deux actions que le distributeur automatique peut faire.
 - ▶ **doReturnMoney**, pour retourner de la monnaie,
 - ▶ ou **doReleaseProduct**, pour expluser le produit.



Question: Pouvez-vous proposer un code qui implémente cette logique?

Analyse du comportement de la machine :

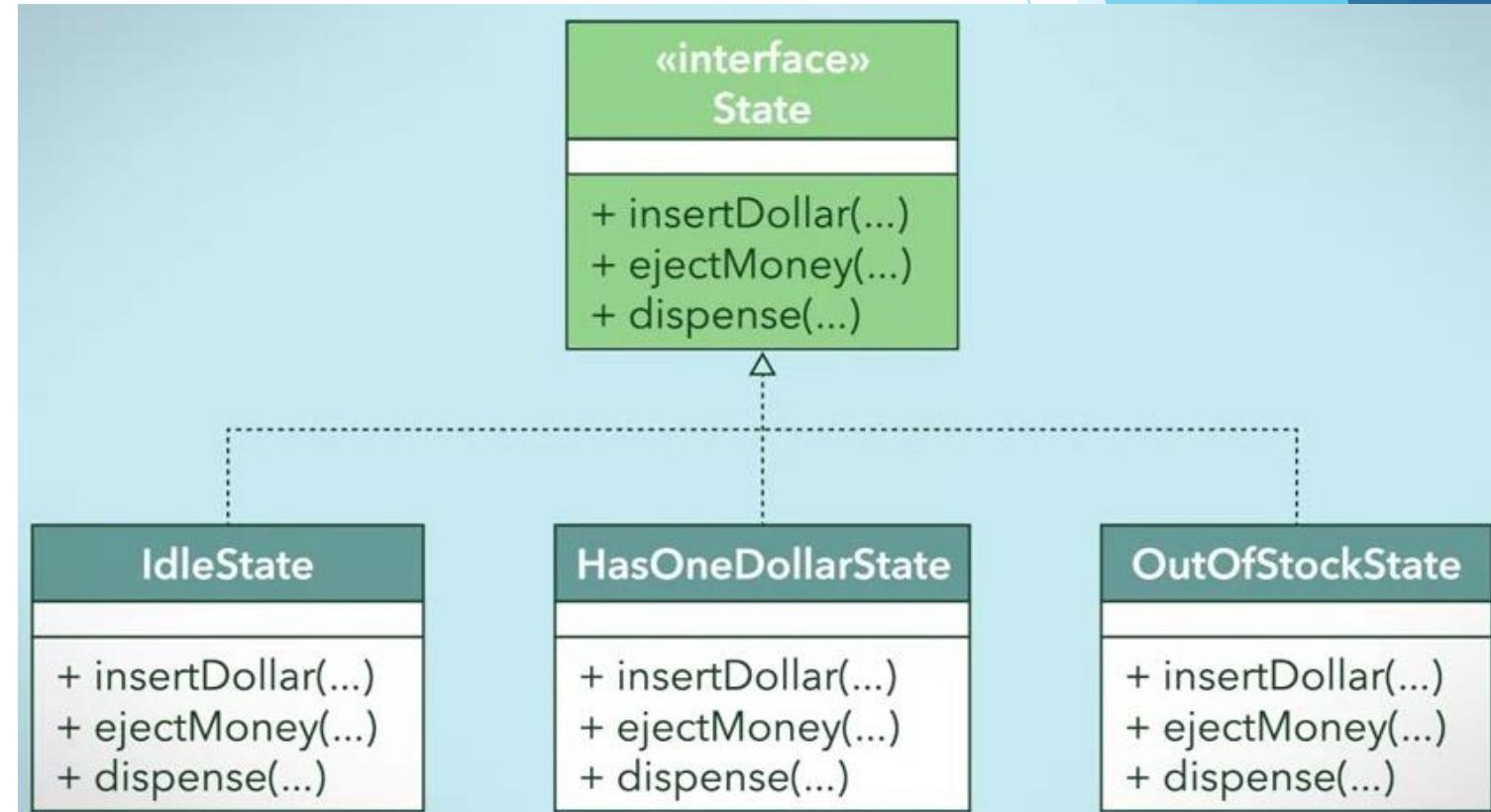
Prenez une seconde et pensez aux différents états que le distributeur automatique pourrait posséder.

- ▶ Avant de vous approcher du distributeur automatique, la machine est dans un état de ralenti,
 - ▶ rien ne se passe.
- ▶ Que se passe-t-il lorsque vous insérez de l'argent?
 - ▶ L'état du distributeur change.
- ▶ Il y a des **événements à écouter** et des **actions à effectuer** qui sont maintenant pertinents pour le distributeur automatique.
 - ▶ Ils pourraient répondre à une demande pour éjecter l'argent → en nous rendant notre argent, et aussi répondre à une demande pour dispenser le produit → faire sortir le produit.
- ▶ Un troisième état de la machine pourrait être en, serait si la machine est en rupture de stock.
 - ▶ Il ne serait plus en mesure de distribuer un produit, de sorte que la réaction de la machine change.

Examinons maintenant comment vous pouvez restructurer cela correctement en utilisant le DP State

Solution UML équivalente

- ▶ Nous définirons une interface State avec une méthode déclencheur à laquelle un État doit répondre.
 - ▶ insertDollar(),
 - ▶ ejectMoney(),
 - ▶ et dispense() .
- ▶ Et nous aurons des classes d'État qui implémenteront l'interface State.
- ▶ Une pour chaque État nécessaire:
 - ▶ IdleState,
 - ▶ HasOneDollarState
 - ▶ et OutOfStockState.



Un State pour tous!

► L'interface **State**:

- ▶ Les classes d'État doivent implémenter les méthodes de cette interface pour répondre à chaque déclencheur.

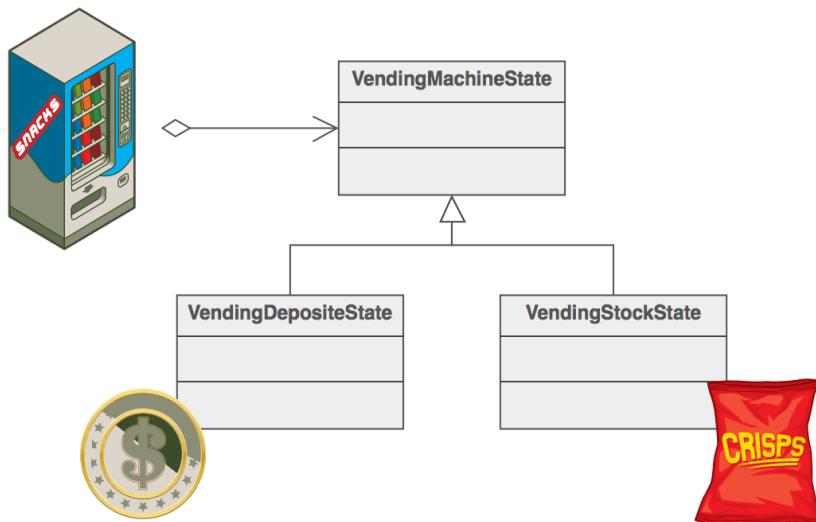
► Notre classe **IdleState** implémente l'interface State.

- ▶ Lorsqu'un dollar est inséré, la méthode **insertDollar()** est appelée,
- ▶ qui appelle alors une méthode **setState()** depuis instance actuelle de **vendingMachine**.
- ▶ Cela modifie l'état actuel de la machine à **HasOneDollarState**

```
public interface State {  
    public void insertDollar( VendingMachine vendingMachine );  
    public void ejectMoney( VendingMachine vendingMachine );  
    public void dispense( VendingMachine vendingMachine );  
}  
  
public class IdleState implements State {  
  
    public void insertDollar( VendingMachine vendingMachine )  
        System.out.println( "dollar inserted" );  
  
        vendingMachine.setState(  
            vendingMachine.getHasOneDollarState()  
        );  
    }  
  
    public void ejectMoney( VendingMachine vendingMachine ) {  
        System.out.println( "no money to return" );  
    }  
  
    public void dispense( VendingMachine vendingMachine ) {  
        System.out.println( "payment required" );  
    }  
}
```

Transitions de States

- ▶ De même, dans la classe **HasOneDollarState**, lorsque la méthode **ejectMoney()** est appelée,
 - ▶ l'argent est retourné et
 - ▶ **setState()** est appelé sur le **vendingMachine** pour changer l'État à la **IdleState**.
- ▶ Voici ce qui se passe si la méthode **dispense()** est appelée sous la classe **HasOneDollarState**.
 - ▶ Le produit est libéré.



```

public class HasOneDollarState implements State {

    public void insertDollar( VendingMachine vendingMachine ) {
        System.out.println( "already have one dollar" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }

    public void ejectMoney( VendingMachine vendingMachine ) {
        System.out.println( "returning money" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }

    // class HasOneDollarState continued

    public void dispense( VendingMachine vendingMachine ) {
        System.out.println( "releasing product" );

        if (vendingMachine.getCount() > 1) {
            vendingMachine.doReleaseProduct();
            vendingMachine.setState(
                vendingMachine.getIdleState()
            );
        } else {
            vendingMachine.doReleaseProduct();
            vendingMachine.setState(
                vendingMachine.getOutOfStockState()
            );
        }
    }
}
  
```

Notre classe VendingMachine

- ▶ Selon que le stock reste après la sortie du produit, l'état actuel de la vendingmachine est défini soit à **idleState** ou bien **outOfStockState**.
- ▶ Le constructeur de la classe Vendingmachine pourra **instantier chacune des sous-classes** d'états.
 - ▶ Et l'état actuel référencé par la variable **currentState** se référera à l'un de ces objets d'État.
- ▶ La classe Vendingmachine pourra aussi avoir des méthodes pour manier les conditions de transition (triggers) comme précédamment,
 - ▶ **Mais cette fois, elle délègue la manipulation à l'objet donnant l'état actuel.**
 - ▶ Toute la "magie" se passe ici!
- ▶ Noter, la taille réduite du code, pas besoin pour de longs test conditionnels dans ces méthodes pour transiter d'un état à un autre.

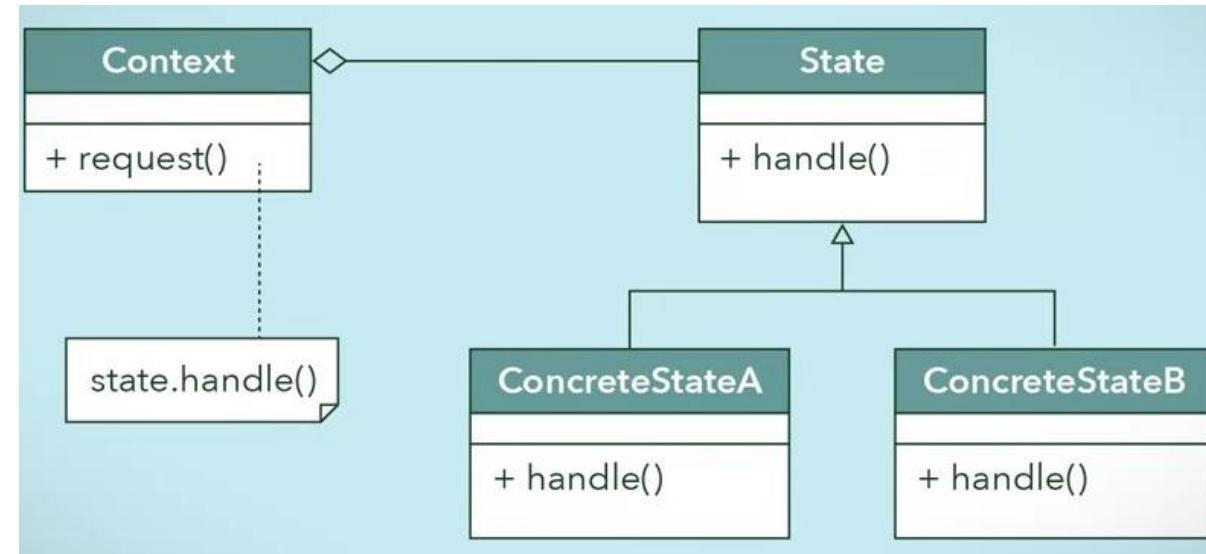
Voir exemple du DP State sous Eclipse

```
public class VendingMachine {  
    private State idleState;  
    private State hasOneDollarState;  
    private State outOfStockState;  
  
    private State currentState;  
    private int count;  
  
    public VendingMachine( int count ) {  
        // make the needed states  
        idleState = new IdleState();  
        hasOneDollarState = new HasOneDollarState();  
        outOfStockState = new OutOfStockState();  
  
        if (count > 0) {  
            currentState = idleState;  
            this.count = count;  
        } else {  
            currentState = outOfStockState;  
            this.count = 0;  
        }  
    }  
}
```

```
public void insertDollar() {  
    currentState.insertDollar( this );  
}  
  
public void ejectMoney() {  
    currentState.ejectMoney( this );  
}  
  
public void dispense() {  
    currentState.dispense( this );  
}
```

Structure générale pour le DP State

- ▶ La structure du modèle d'état ressemble généralement à ceci.
 - ▶ Dans notre exemple, la VendingMachine est la classe de contexte. Elle garde une trace de son état actuel.
- ▶ Lorsqu'un **trigger** sera réalisé et une demande est passée à **l'objet contexte**, il la **délègue à un objet d'État** pour traiter la demande.

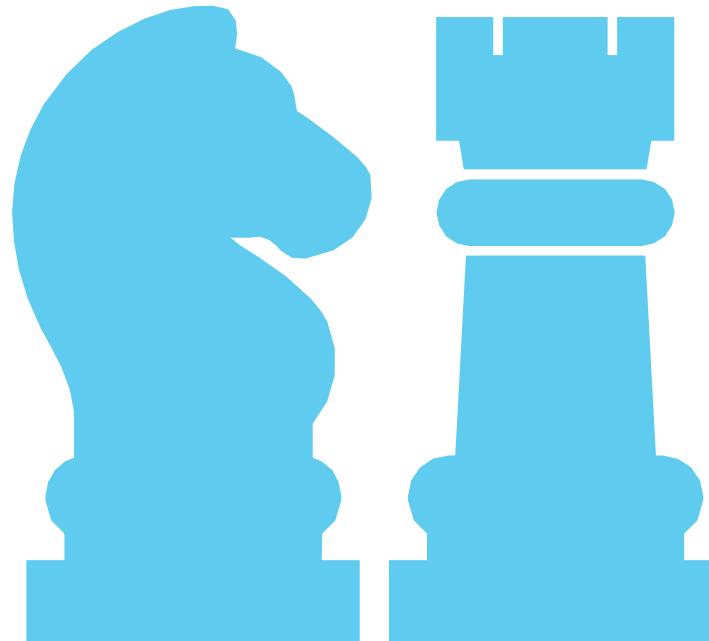


- ▶ Le modèle d'état est utile lorsque vous devez modifier le comportement d'un objet en fonction des modifications apportées à son état interne.
- ▶ Vous pouvez également utiliser le modèle pour simplifier les méthodes qui souffrent de longues conditions dépendantes de l'état de l'objet.

Discussion: Défaire/refaire une action

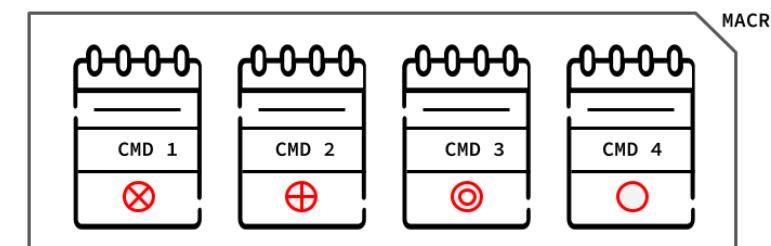
Comment y procéder?

- ▶ Vous écrivez un programme de jeu d'échecs
- ▶ Vous voulez reprendre certains mouvements en arrière.
 - ▶ Vous voudriez peut-être aussi refaire les mêmes mouvements que vous avez repris.
- ▶ Comment penseriez-vous à concevoir ce programme?
 - ▶ Cela s'applique, certainement à tout autre programme (Word, Photoshop, etc.)



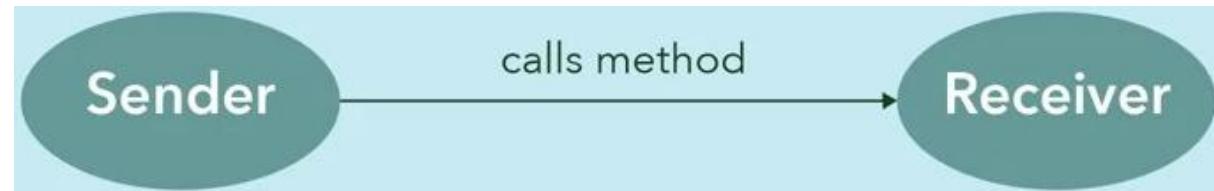
V.4- Le DP Command

Création de pile pour annulation / rétablissement dans notre logiciel



Without VS with Command DP

- ▶ The Command DP encapsulates the request as an object of its own.
- ▶ Usually, when one object makes a request for a second object to do an action, the first object would call a method of the second object and the second object would complete the task.
 - ▶ In this situation, the sender object **directly communicates** with the receiver object.



- ▶ Instead of having these objects directly communicating with each other, the command pattern creates a **command object in between** the sender and receiver.
 - ▶ This way, the sender doesn't need to know about the receiver and the methods to call.



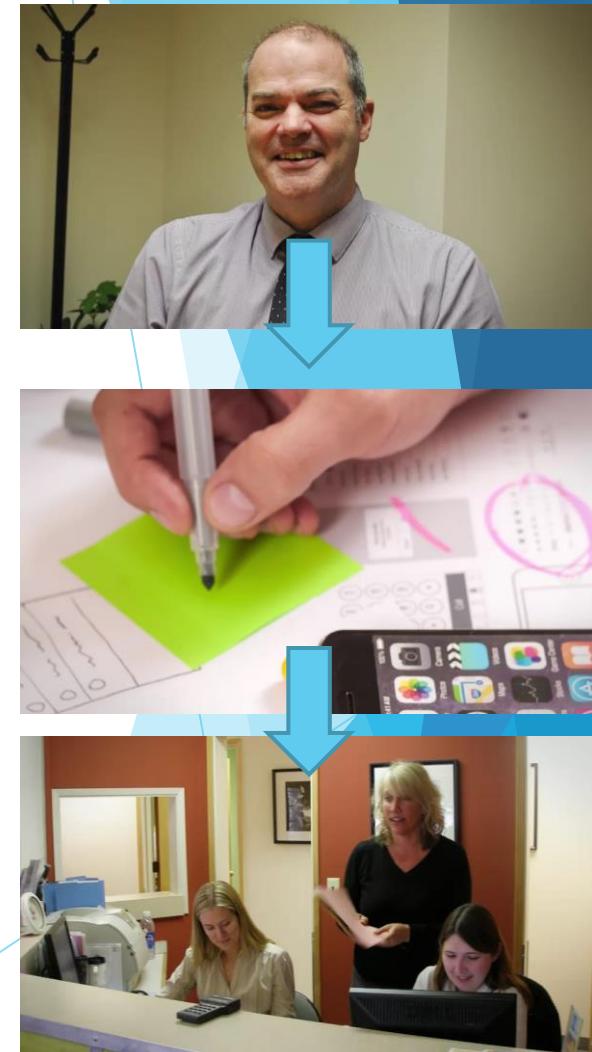
The Command DP is like a Boss' Secretary

- ▶ Think of the Command DP like how a busy boss in a company uses memos to carry out tasks.
 - ▶ If the boss needed a worker to schedule a meeting and also needed another worker to talk to an important client about business, he can use memos.
 - ▶ The boss may not have time to remember which workers do what jobs and won't have time to walk to the different workers.

Instead:

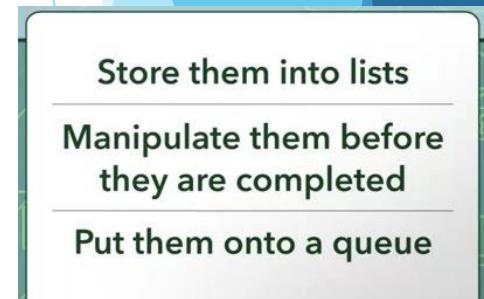
- ▶ the **boss** (commands manager) could
- ▶ just write the tasks down onto a **sticky note** (command)
- ▶ and the **secretary** (invoker) could
- ▶ give these to the **workers** (receivers) that can complete them.
- ▶ So, the boss is encapsulating his commands into memos,

Similarly, requests could be encapsulated into command objects in software.



Command DP usage in software

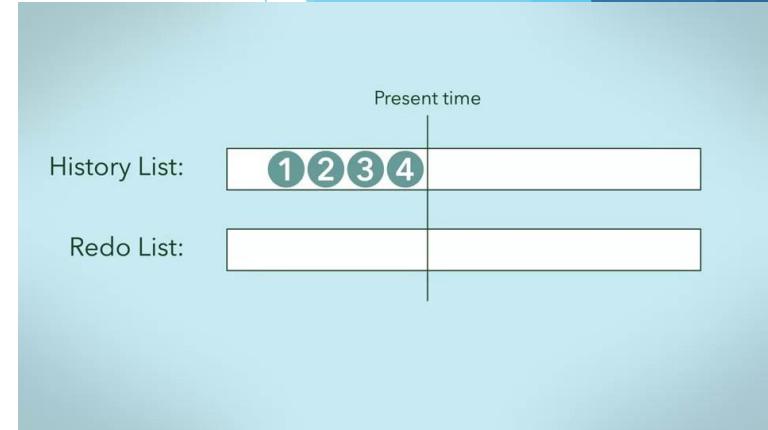
- ▶ In software, a sender object can create a command object.
- ▶ To invoke the specific receiver object to complete the task, an **invoker** object is needed
- ▶ A **command manager** can also be used which basically keeps track of the commands, manipulate and invoke them.
 - ▶ The secretary can keep track of the memos, making sure that the memos are executed at the right time.
- ▶ **USE CASE 1: To store and schedule different requests**
 - ▶ Turning the different requests in our software into **command-objects** allows us to treat them as **objects**.
 - ▶ We can store these commands into lists, manipulate them before they are completed, or put them onto a queue so that we can schedule different commands for different times.
- ▶ Example: Use of the command DP to have an **alarm** ring in a calendar software.
 - ▶ When an event is created in the calendar, a command object could be created to ring the alarm.
 - ▶ This command could be put onto a queue, so that the command can be completed later when the event is actually scheduled to occur.



The Undo/Redo functionality

USE CASE 2: One very important purpose for the command DP

- ▶ Suppose that you are creating text editing software:
 - ▶ Many commands can be executed: delete text, change font, colors, etc.
 - ▶ To achieve redo and undo functionality, our software will need **two lists**:
 - ▶ a **history list** which holds all the commands that have been executed,
 - ▶ and a **redo list** which would be used to put commands that have been undone.



1. When a command is **requested**, a command object is created and executed.
2. When a command is **completed**, that command goes to the **history list**.
3. If we undo a command, the software takes the most recent command in history list
 - ▶ The software would ask this command to undo itself and then put it on the redo list.
4. If we undo another command, then in order to redo it, the software would take the most recent command undone in the redo list and ask the command to execute again.
5. Then move it onto the history list again. The redo list will also need to be emptied every time.

07-May-21

156

The Undo/Redo functionality (suite)

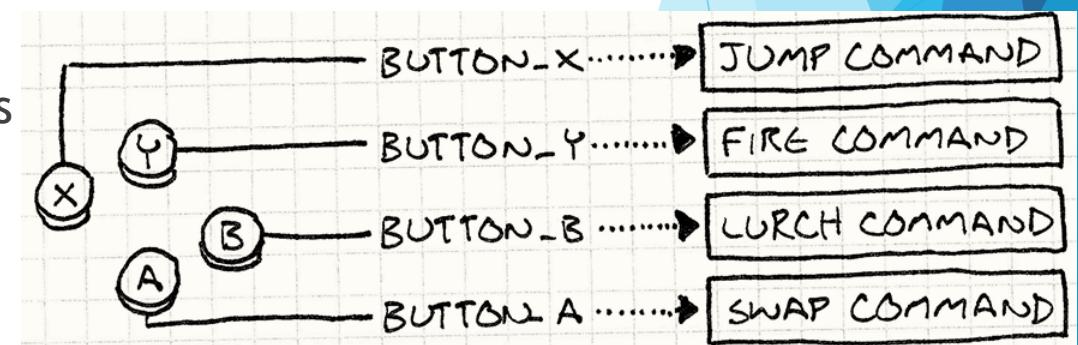
- ▶ This redo/undo functionality doesn't have to be just for text editors.
- ▶ We can use a command pattern to allow redo/undo **on any type of application**.
- ▶ The important thing is that the command pattern lets you do things to request that you wouldn't be able to do if they were simple method calls from one object to the other.

USE CASE 3: You can store these commands in a log list.

- ▶ So that if your software program has an unexpected crash, you can allow your users to redo all the recent commands.
- ▶ Creating these requests as objects allows you to create very useful functionality in your programs.

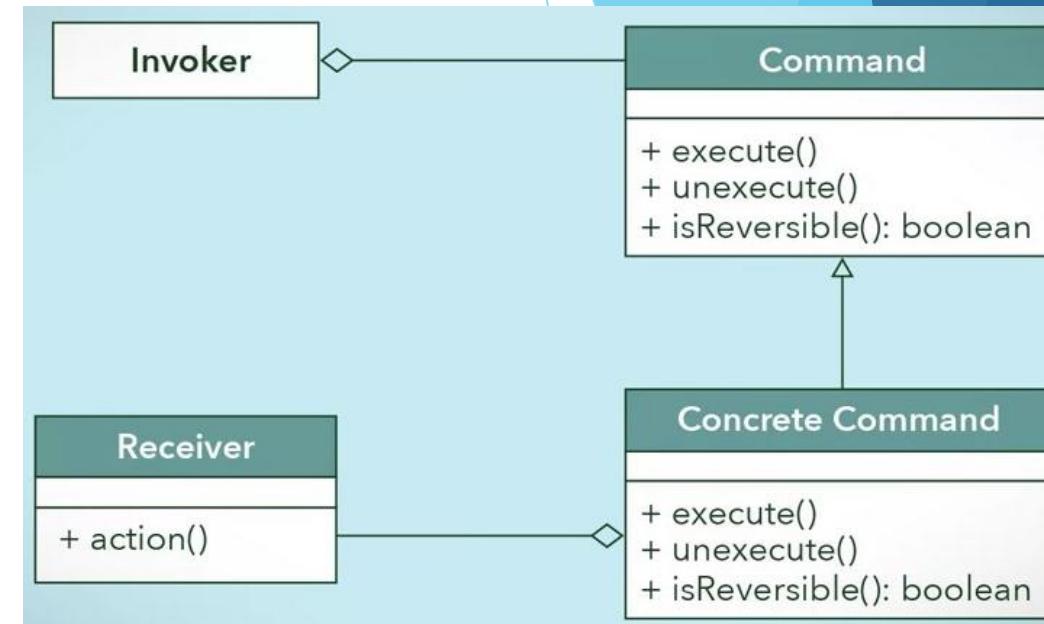
USE CASE 4: control user Inputs in a game

- ▶ Somewhere in every game is a chunk of code that reads in raw user input – button presses, keyboard events, mouse clicks, whatever. It takes each input and translates it to a meaningful action in the game:



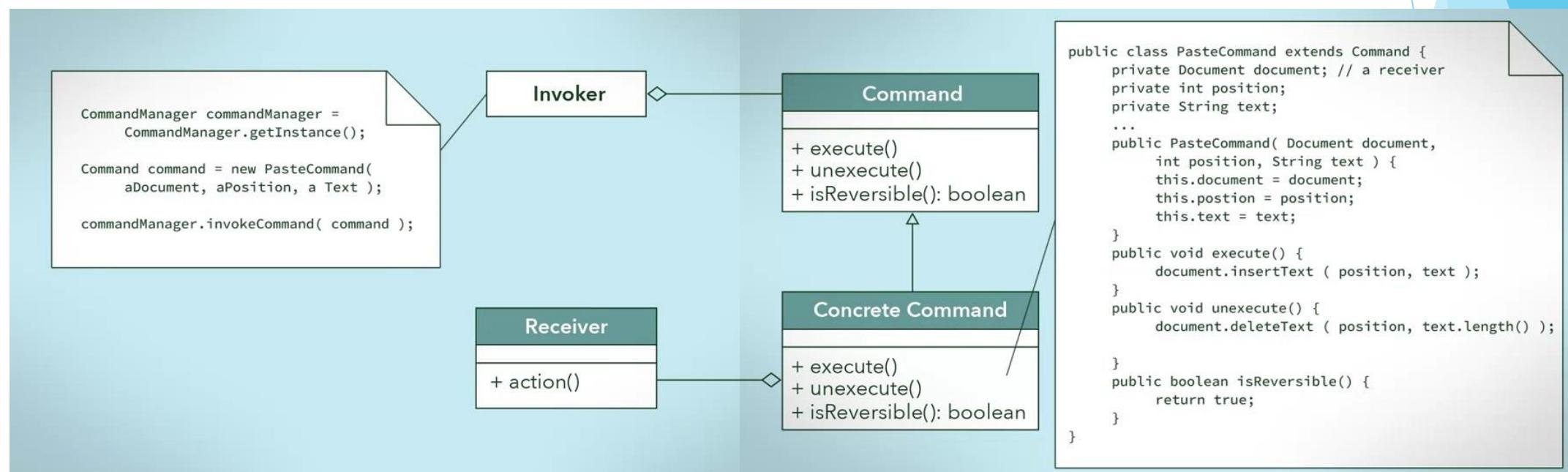
Command DP: UML Diagram

- ▶ The **Command** super-class defines the common behaviors of our commands.
- ▶ All **ConcreteCommands** will be instances of it, have the methods execute, unexecute and isReversible.
 - ▶ The **execute** method does the work that the command is supposed to do,
 - ▶ **unexecute** is for undoing the command,
 - ▶ and **isReversible** determines if the command is reversible,
 - ▶ There could be some commands that can't be undone. For example, save as a command that doesn't really make sense to be undone.
- ▶ These concrete command classes will call on specific **Receiver** classes to deal with the actual work of completing the command.



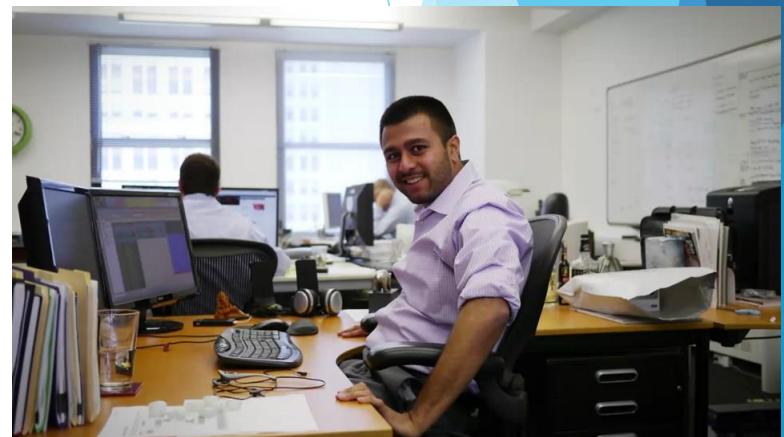
Command DP: UML Diagram (suite)

- ▶ For the **ConcreteCommand** class, for example pasting text:
 - ▶ the **pasteCommand** extends the command super-class and keeps track of where the text will be inserted and what will be inserted.
 - ▶ Command objects must keep track of a **lot of details** on the current state of the document in order for commands to be reversible.
 - ▶ In this case, the paste command is also a **reversible** command because you can delete the object you just pasted.
- ▶ When the execute and unexecute methods are called, this is where the command object actually calls on the receiver,
- ▶ The source code for the **invoker** is done in simple steps:
 - ▶ First, it will need a reference to the command manager, which is the object that manages the history and redo lists.
 - ▶ The invoker then creates the command object with the information needed to complete the command, then calls the command manager to execute the command.



Benefits of the Command DP

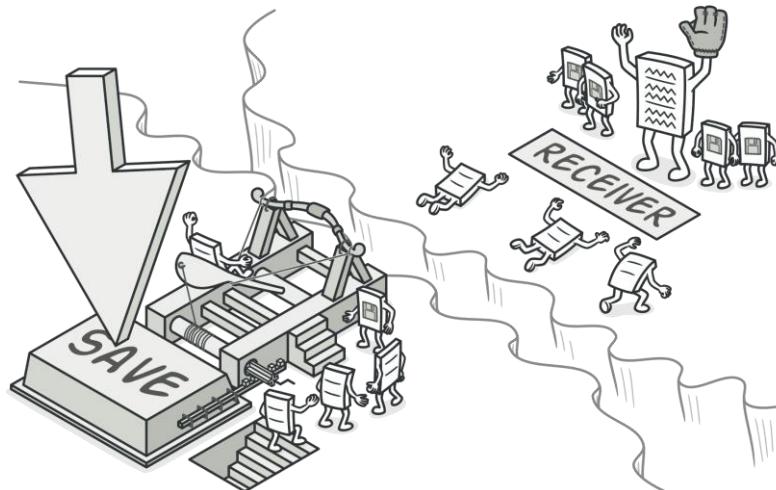
- ▶ Another main benefit of the command pattern is that it **decouples the objects** of your software program.
- ▶ When you have a class that wants to make a request, it doesn't need to know about the other objects in the software system.
- ▶ It can simply just **create a command object** and let the command object deal with the work by invoking the receiver objects,
- ▶ The object doesn't need to know who deals with the request anymore, like how the boss doesn't need to know which worker will deal with the task.



Benefits of the Command DP (suite)

USE CASE 5: The command pattern also allows you to pull out logic from your *user interfaces*.

- ▶ Usually, the code to handle requests is put into the event handlers for the UIs.
 - ▶ However, it doesn't really make sense to have a lot of application logic sitting in your UI classes.
 - ▶ The UI classes should only be dealing with UI issues like getting information to and from the user.
- ▶ Instead, the Command DP creates a new layer which is where these command objects will go.
 - ▶ Every time a button is clicked, it will create a **command** object instead. And that is where the logic will set.
 - ▶ These command objects are independent (like Controllers in MVC) of your user interface.
- ▶ So, it makes adding changes like new buttons to your software's user interface easier and faster.



Takeaways on the Command DP

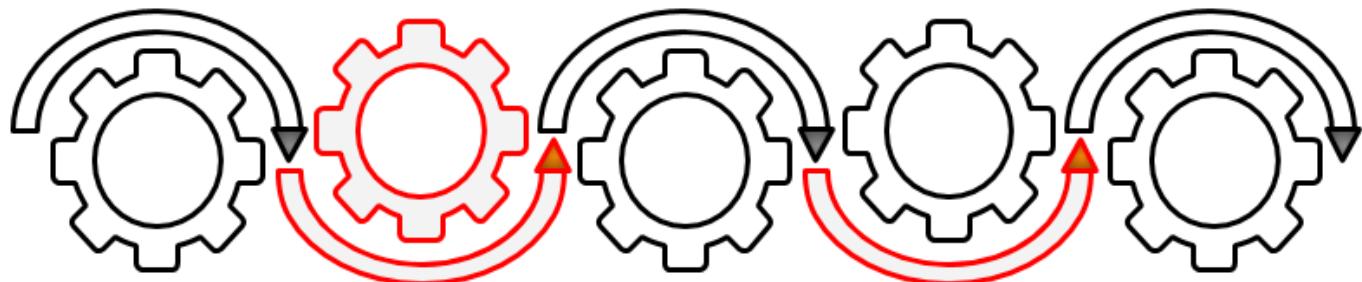
- ▶ It allows to turn each and every service in your system into an object of its own and allow much more flexible functionality.
- ▶ Decouples the different components of our system
- ▶ A very useful DP for many use cases: command queues, undo/redo lists, logs handling, UI controls, etc.
- ▶ Next time you're creating an application, soon you'll see that using this pattern can really make your software programs versatile and easier to maintain.

You'll want to use it on every software you create.
Good luck!



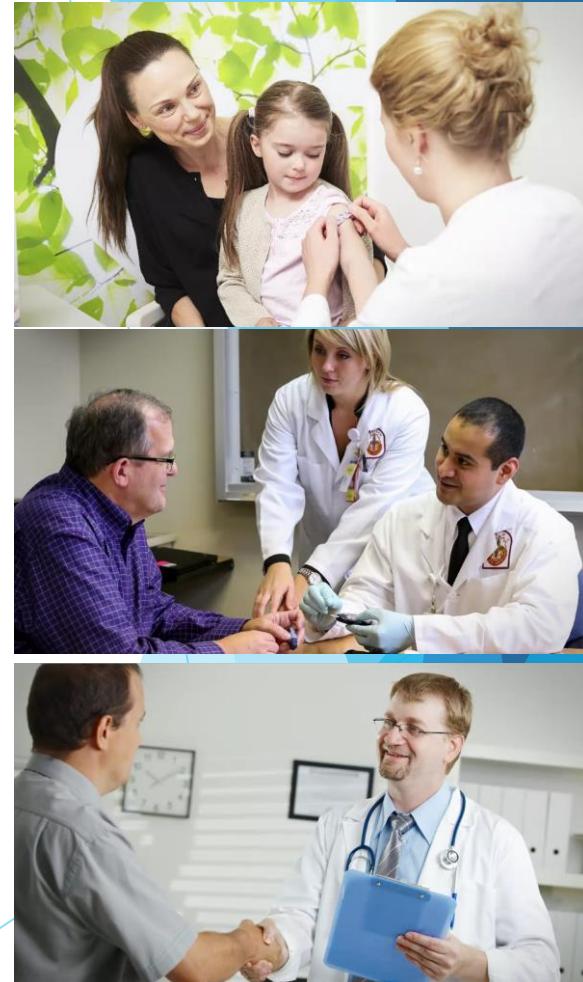
V.5- Le DP Chain of responsibility

Apporter une contribution, chacun de son côté, pour répondre au max du besoin



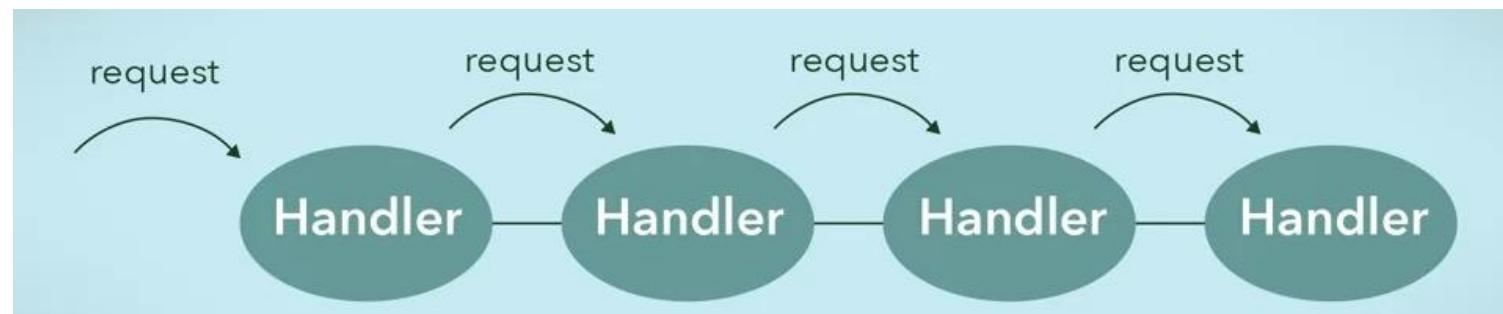
Chain of responsibility analogy to doctors

- ▶ The Chain of Responsibility is what it sounds like:
 - ▶ A chain of objects that are responsible for handling requests
- ▶ Think of it like requesting help on a health problem:
 1. So, you try visiting your doctor first with your quest but your case is unusual, so you're referred to a specialist.
 2. But the specialist is on leave and can't take your request and so you referred to another specialist.
 3. And finally, this specialist ends up dealing with your problem.
- ▶ You don't care who along this chain of health professionals actually helps. You only care that your request is met.



Chain of responsibility in software design

- ▶ In software design, the Chain of Responsibility is a series of handler objects that are linked together.
- ▶ These handlers have methods that are written to handle specific requests.
 1. When a client object sends a request, the first handler in the chain will try to process it.
 - ▶ If the handler can process the request, **the request ends** at this handler.
 2. If the handler cannot deal with the request, the handler will send the request to the next handler in the chain who will try to process the request.
 3. Again, if the handler cannot process the request, it will send the request onto the next handler.
 - ▶ This passing of the request continues until we find the handler that can process the request.
- ▶ If the request goes through the entire chain of handlers and no handler can process it, then **the request is not satisfied**



Ether real-world examples

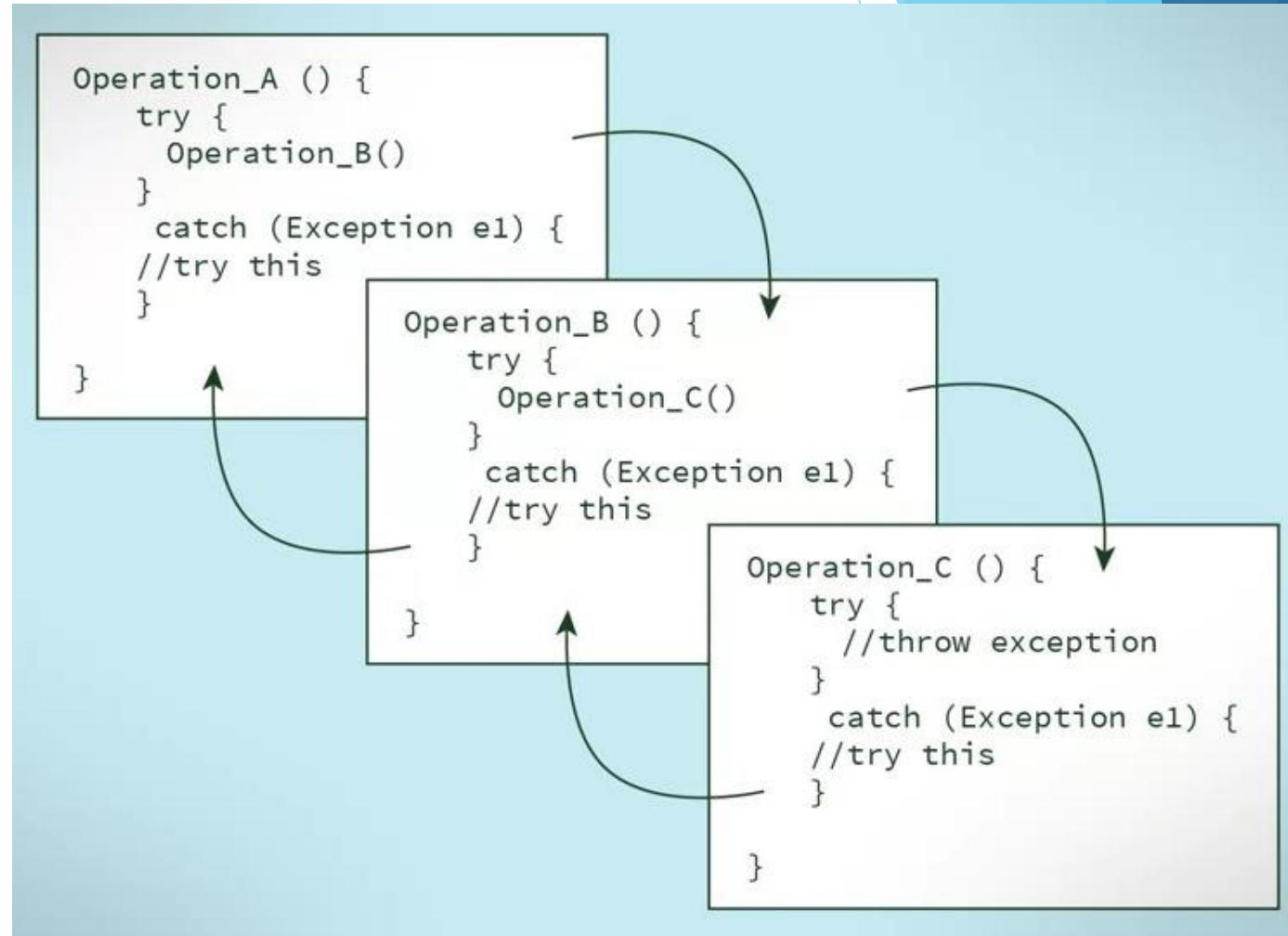
- ▶ Consider that you are fixing a chair so,
 - ▶ You needed a tool to tighten a particular screw.
 - ▶ But, you're not sure which type of tool to use.
- ▶ You have several types of screwdrivers and wrenches. What would you do?
 - ▶ You will probably take each tool and one at a time try it on the screw until one of them works.
- ▶ This is very similar to how the Chain of Responsibility design pattern works.
 - ▶ Each object tries to handle the request **until one is able to** successfully handle it.
- ▶ Another example is when you have a set of key and you are searching, a step at a time, for the right key that could unlock the door



The case of Exception handing

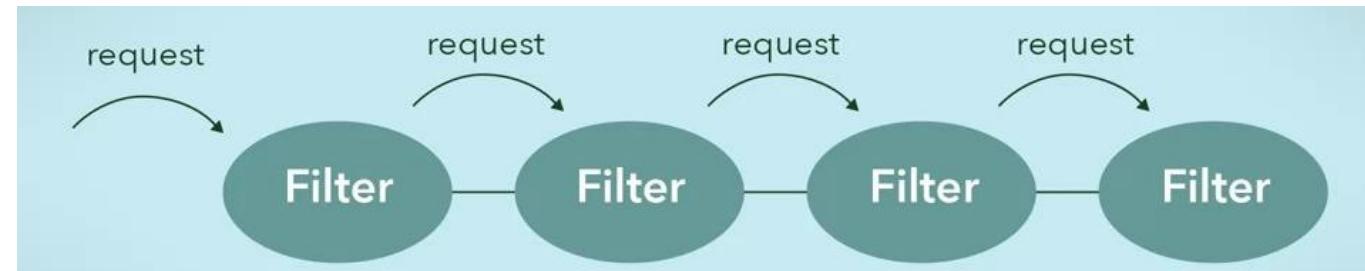
- ▶ You can also think of this pattern as similar to Exception handling in OOP languages such as Java or C#.
- ▶ With Exception handling, you write a series of try-catch blocks in your software to ensure that exceptions are dealt with properly.
- ▶ When an exception occurs, one of the catch blocks is expected to handle it.

So, Where might you use this design pattern in your software?



Example: Email service

- ▶ Suppose you are setting up an email service where there are lots of ways to filter through the emails:
 - ▶ an email message is considered by a series of filters until one is found that applies.
 - ▶ We can create objects that work as the **individual filters**.
- ▶ Each of these filter objects has a method that I will call Handle request.
 - ▶ This method will check if it filters particular rule matches an email messages content.
 1. If the rule matches the content, it will deal with the email.
 - ▶ For example, the filter object can put that email into a spam folder and the request is done.
 2. If the rule doesn't match the content,
 - ▶ then the filter will call the next filter in the chain to handle the request.
- ▶ Using the Chain of Responsibility design pattern is a very common way to set up this kind of behavior.

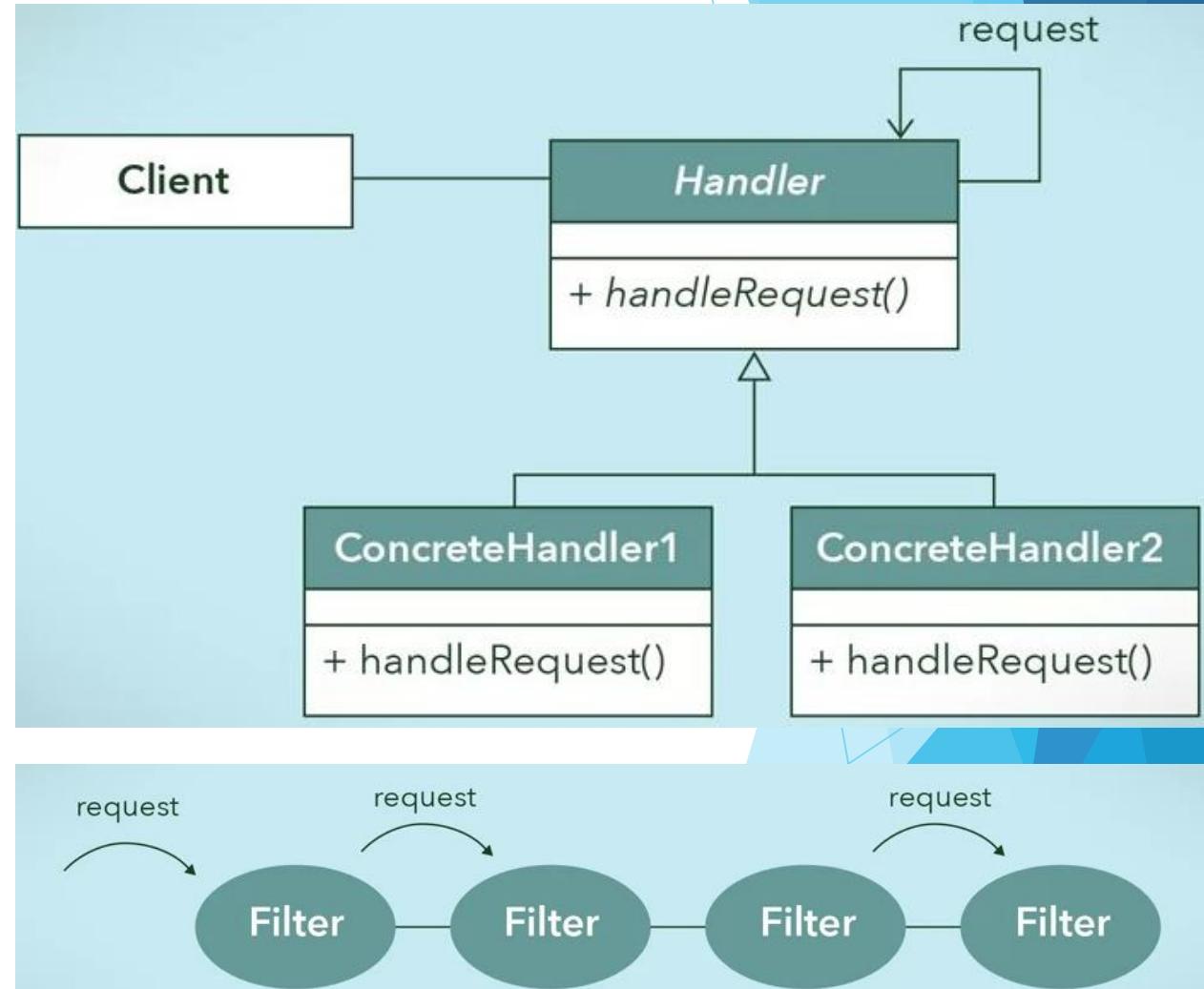


UML Diagram for Chain of Responsibility

- In general, UML class diagram for the Chain of Responsibility DP looks like this.
- All the objects on the chain are handlers that implement a common method :
 - `handleRequest()` declared in an abstract superclass handler.
- These handle objects are connected from one to the next in the chain.
 - There are subclasses of handler to handle requests in their own way.

Can you see an issue with the design?

- What if, there's a mistake in the second filter?
- What if its rule doesn't match but forgets to pass the request onto the next filter?



How to guarantee the Chain?

- ▶ We need an algorithm to make sure that each chain class handles requests in a similar fashion and goes through the following steps:
 1. First, check if a rule matches.
 2. If it does match, do something specific.
 3. If it doesn't match, call the next filter in the list.
- ▶ To achieve this behavior, we can use the **Template Pattern** that we saw earlier lesson
 - ▶ It ensures that each class will handle the request in a similar way following the required steps.
- ▶ Often, design patterns can have their own internal problems and we can use other design patterns to fix those problems.

It is very common in software design to combine patterns.

1. **Check if rule matches**
2. **If it matches, do something specific**
3. **If it doesn't match, call the next filter in the list**



Points à retenir: DP Chain Of Responsibility

- ▶ You can remember this DP as being similar to an **Administration bureaucracy!**
 - ▶ You have to get a paper from an employee, but
 - ▶ He doesn't have the right, so leads you to the director,
 - ▶ The latter does not have the time, so you are directed to the PDG!
 - ▶ Finally, and hopefully, the PDG signs your paper on a hurry and goes!
- ▶ The Chain of Responsibility is a very important pattern to learn in software.
 - ▶ The intent of this design pattern is to **avoid coupling the sender to the receiver** by giving more than one object the chance to handle the request.
- ▶ Whoever sends the request **doesn't have to care** who will handle the request.
 - ▶ It just sends it to the first thing and hopefully someone will take care of it.

This implementation makes it very easy for the sender

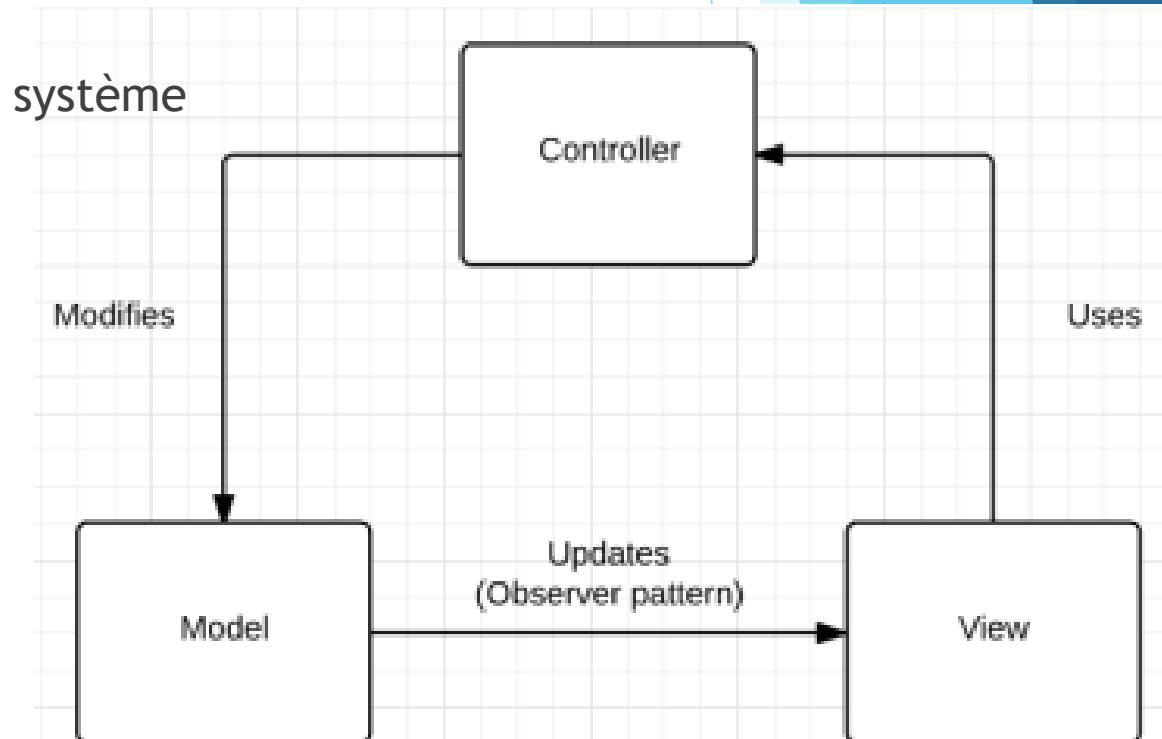
A photograph of a man wearing a yellow hard hat, a brown blazer, and blue jeans. He is looking down at a large sheet of architectural blueprints spread out on a table. In the background, there is a computer monitor and some papers. The image is partially overlaid by a large, semi-transparent graphic of a modern building's structural framework.

VI- Les DP d'Architecture

-
1. MVC
 2. MVVM

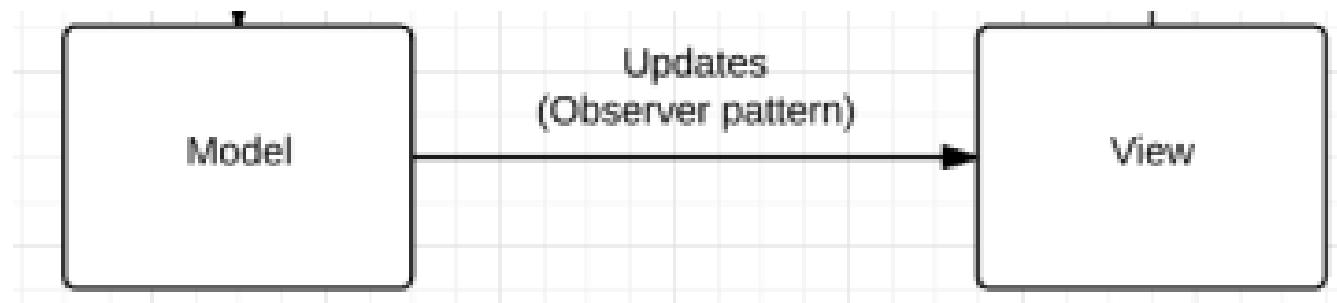
Les patterns d'architecture: MVC

- ▶ Le **DP MVC** - Assemblées de modèle, vue et contrôleur. C'est un DP qui sépare les fonctions de données, d'affichage et de contrôle en rôles distincts.
- ▶ **Model, View, Controller (MVC) patterns** sont à prendre en considération si on a affaire à une utilisation d'interfaces utilisateur.
- ▶ Les modèles MVC divisent les responsabilités d'un système qui offre une interface utilisateur en trois parties
- ▶ Notez l'indication de l'utilisation du **Observer DP**



MVC: Model - View relationship

- ▶ Le **Model** contient les données, l'état et la logique que les utilisateurs veulent voir et manipuler.
 - ▶ Il est similaire au « back end » du logiciel.
 - ▶ Un aspect clé du DP MVC est que le modèle est **autonome et indépendant**.
 - ▶ Il a tous les States,
 - ▶ méthodes et données nécessaires le rendre indépendant en soi.
- ▶ Le **View** permet aux utilisateurs de voir le modèle comme ils s'y attendent, d'y interagir en totalité, ou du moins avec des parties de celui-ci.
 - ▶ C'est le « front end » ou la couche de présentation du logiciel.
 - ▶ Un modèle peut avoir plusieurs **View** qui présentent différentes parties du modèle, ou présentent le modèle de façons différentes.
- ▶ Lorsqu'une valeur change dans le **model**, la **view** doit être **notifiée** afin qu'elle puisse appliquer un **update** adéquat.
 - ▶ Le **observer design pattern** permet que cela se produise.
 - ▶ Ici, la vue est un observateur. Lorsque le modèle change, il informe toutes les vue qui en sont souscrites (**subscribed**).



From View to Controller, then from controller to Model

- ▶ La vue fournit également aux utilisateurs des moyens d'apporter des modifications aux données du modèle.

- ▶ Mais, elle n'envoie pas directement ses demandes au modèle,
- ▶ Au lieu de cela, l'interaction utilisateur est transmise à un **Controller**.

1. Le **controller** est responsable de :

- ▶ L'interprétation des demandes et l'interaction avec les éléments de la vue,
- ▶ Et le changement du modèle,
- ▶ Par conséquent, s'assure que les vues et le modèle sont faiblement couplés,

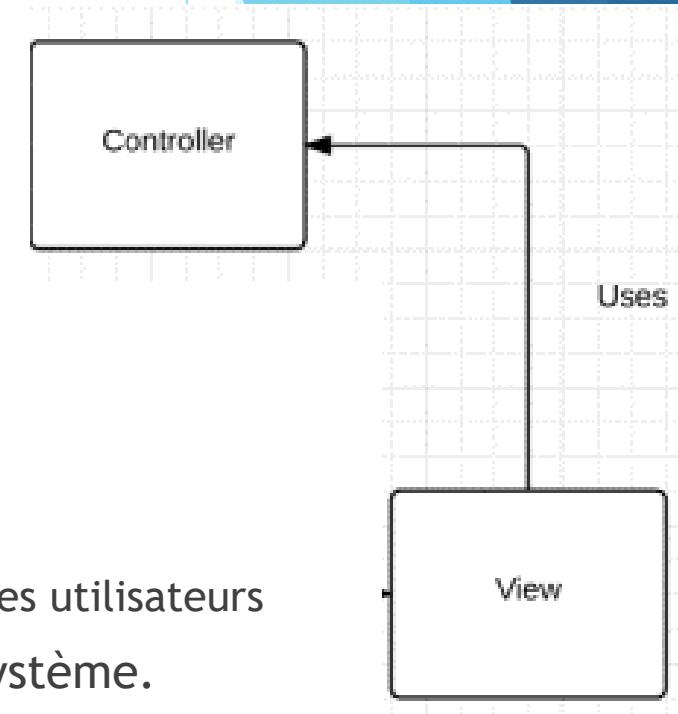
2. La **View** n'est donc responsable que de l'apparence visuelle du système.

- ▶ correspond à un **boundary object** au bord de votre système et qui traite avec les utilisateurs

3. Le **model** se concentre uniquement sur la gestion des informations du système.

- ▶ correspond à des **entity objects** extraits en analysant l'espace du système par conception

Le modèle MVC utilise le principe de **separation of concerns** pour répartir les principales responsabilités dans un système interactif.

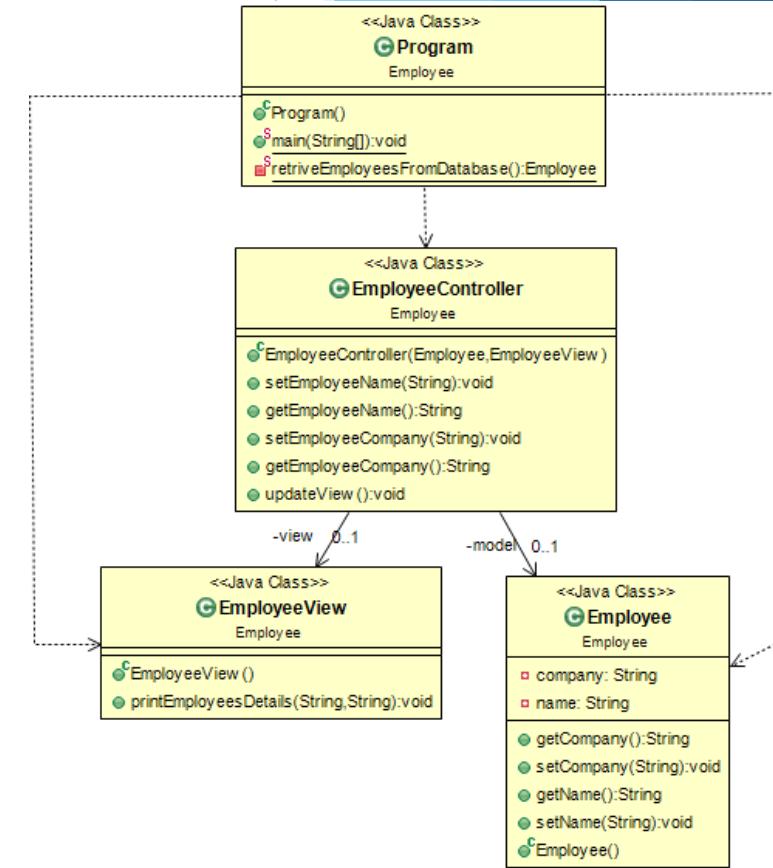


Exemple MVC 1: Employee dans une Company

- Cet exemple est très simple, mais démontre que le contrôleur doit avoir des références à la fois à la vue et au modèle qu'il connecte.

```
public class Employee {
    private String company;
    private String name;
    public String getCompany() {
        return company;
    }
    public void setCompany(String company) {
        this.company = company;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Employee() {
        System.out.println("- new employee -");
    }
}
```

```
public class EmployeeView {
    public EmployeeView() {
        System.out.println("-+ new view created +-");
    }
    public void printEmployeesDetails(String
        employeeName, String employeeCompany) {
        System.out.println("\nEmployee details display:");
        System.out.println("Name: " + employeeName);
        System.out.println("Roll No: " +
            employeeCompany);
    }
}
```



Exemple MVC 1: Employee dans une Company (suite)

```
public class EmployeeController {  
    private Employee model;  
    private EmployeeView view;  
  
    public EmployeeController(Employee model, EmployeeView view){  
        System.out.println("-+| new controller |+-");  
        this.model = model;  
        this.view = view;  
    }  
    public void setEmployeeName(String name) {  
        model.setName(name);  
    }  
    public String getEmployeeName() {  
        return model.getName();  
    }  
    public void setEmployeeCompany(String rollNo) {  
        model.setCompany(rollNo);  
    }  
    public String getEmployeeCompany() {  
        return model.getCompany();  
    }  
    public void updateView() {  
        view.printEmployeesDetails(model.getName(),  
            model.getCompany());  
    }  
}
```

```
public class Program {  
    public static void main(String[] args) {  
        Employee model = retrieveEmployeesFromDatabase();  
        EmployeeView view = new EmployeeView();  
        EmployeeController controller = new  
        EmployeeController(model, view);  
        controller.updateView();  
  
        controller.setEmployeeCompany("INETUM");  
        controller.updateView();  
    }  
  
    private static Employee retrieveEmployeesFromDatabase()  
    {  
        System.out.println("> NOW WE READ DATA <");  
        Employee employee = new Employee();  
        employee.setName("Robert");  
        employee.setCompany("Atlas_Corp");  
        return employee;  
    }  
}
```

Voir Exemple de code sous Eclipse

MVC example code 2

- ▶ Imaginez que vous créez une interface pour une magasin, où les caissiers (**Cashier**) peuvent entrer des commandes (**Orders**), et ils sont affichés.
- ▶ Les clients (**Customer**) et les caissiers (**Cashier**) devrait être en mesure de voir la liste des éléments entrés dans l'ordre avec un scanner de code à barres, et voir le montant total de la facture.
- ▶ Les caissiers devraient également être en mesure d'apporter des corrections si nécessaire.



MVC, Exemple de code pour le Model

- ▶ le modèle, devrait être en mesure d'exister indépendamment, sans vues ou contrôleurs.
- ▶ Puisque la vue va être un observateur (**Observer**), le modèle doit être observable (**Observable**).
 - ▶ L'interface **Observable** peut être utilisée pour implémenter ce comportement.
- ▶ Ici, Observable peut être étendue et the StoreOrder class allows to add views as observers.
 - ▶ This will allow them to update whenever the order is updated.

```
import java.util.*;
public class StoreOrder extends Observable {
    private ArrayList<String> itemList;
    private ArrayList<BigDecimal> priceList;
    public StoreOrder() {
        itemList = new ArrayList<String>();
        priceList = new ArrayList<BigDecimal>();
    }
    public String getItem( int itemNum ) {
        return itemList.get(itemNum);
    }
    public String getPrice( int itemNum ) {
        return priceList.get(itemNum);
    }
    public ListIterator<String> getItemList() {
        ListIterator<String> itemItr =
itemList.listIterator();
        return itemItr;
    }
}
```

```
public ListIterator<BigDecimal> getPriceList() {
    ListIterator<String> priceItr = priceList.listIterator();
    return priceItr;
}
public void deleteItem( int itemNum ) {
    itemList.remove(itemNum);
    priceList.remove(itemNum);
    setChanged();
    notifyObservers();
}
public void addItem( int barcode ) {
    // code to add item (exp. used from a scanner)
    setChanged();
    notifyObservers();
}
public void changePrice( int itemNum, BigDecimal newPrice ) {
    priceList.set(itemNum,newPrice);
    setChanged();
    notifyObservers();
}
}
```

MVC, Exemple de code View

- ▶ Puisque OrderView implemente l'interface **Observer**, une méthode Update() doit être fournie
- ▶ Noter le **downcast** de StoreOrder pour appeler getItemList() et getPriceList()
- ▶ La vue ne peut pas appeler la méthode du modèle,
 - ▶ appelle plutôt les méthodes du contrôleur

```
import java.util.*;
import javax.swing.JFrame; // ..etc.
public class OrderView extends JPanel implements Observer,
ActionListener{
    // Controller
    private OrderController controller;
    // User-Interface Elements
    private JFrame frame;
    private JButton changePriceButton;
    private JButton deleteItemButton;
    private JTextField newPriceField;
    private JLabel totalLabel;
    private JTable groceryList;
    private void createUI() {
        // Initialize UI elements. e.g.:
        deleteItemButton = new JButton("Delete Item");
        add(deleteItemButton);
        //... // Add listeners. e.g.:
        deleteItemButton.addActionListener(this);
        //...
    }
}
```

```
public void update ( Observable s, Object arg ) {
    display(((StoreOrder) s).getItemList(),
    ((StoreOrder) s).getPriceList());
}
public OrderView(OrderController controller) {
    this.controller = controller;
    createUI();
}
public void display ( ArrayList itemList, ArrayList
priceList ) {
    // code to display order
    //...
}
public void actionPerformed(ActionEvent event) {
    if (event.getSource() == deleteItemButton) {
        controller.deleteItem(groceryList.getSelectedRow());
    }
    else if (event.getSource() == changePriceButton) {
        BigDecimal newPrice = new
        BigDecimal(newPriceField.getText());
        controller.changePrice(groceryList.get
        SelectedRow(),newPrice);
    }
}
M2ii
}
```

MVC, Exemple de code pour le Controller

- ▶ Notez également que le contrôleur n'apportera pas de modifications directes à l'état du modèle.
- ▶ Au lieu de cela, il appelle les méthodes du modèle pour apporter des changements

```

public class OrderController {

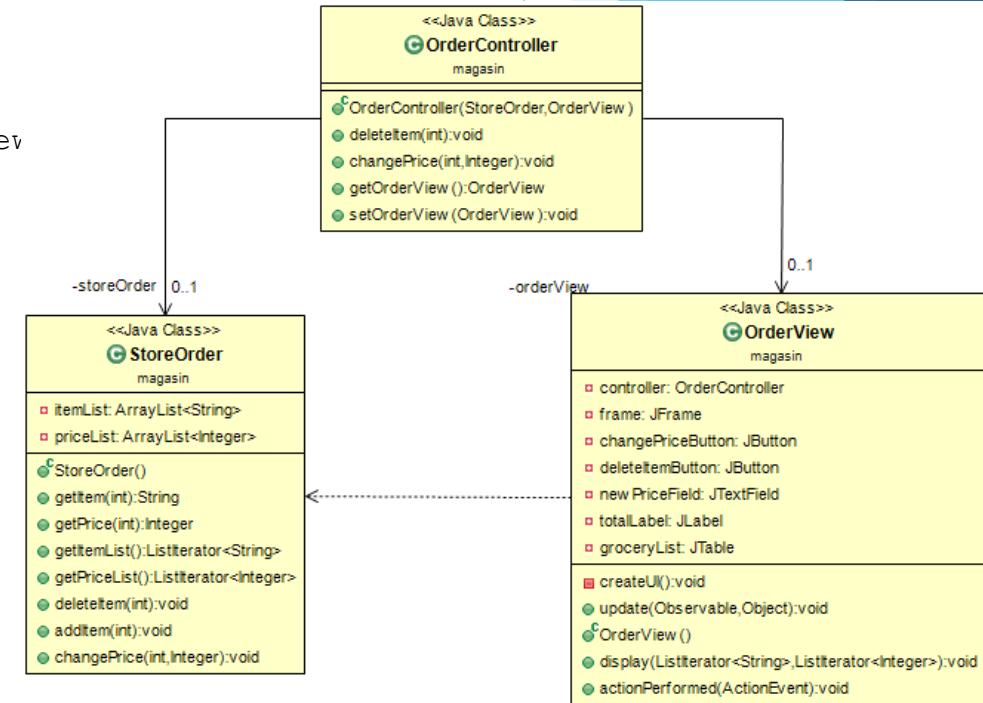
    private StoreOrder storeOrder;
    private OrderView orderView;

    public OrderController(StoreOrder storeOrder, OrderView orderView) {
        this.storeOrder = storeOrder;
        this.orderView = orderView;
    }

    public void deleteItem( int itemNum ) {
        storeOrder.deleteItem( itemNum );
    }

    public void changePrice( int itemNum, BigDecimal newPrice ) {
        storeOrder.changePrice( itemNum, newPrice );
    }
}

```



Points à retenir à propos du MVC

- ▶ Les modèles MVC peuvent être utilisés de différentes façons.
- ▶ La caractéristique déterminante du modèle MVC est la séparation des préoccupations entre .
 - ▶ Le back-end,
 - ▶ le front-end,
 - ▶ et la coordination entre les deux
- ▶ La vue peut se concentrer sur son objectif principal : **présenter l'interface utilisateur**,
- ▶ Le contrôleur prend la responsabilité
 - ▶ **d'interpréter l'entrée** de l'utilisateur et
 - ▶ de **travailler avec le modèle** en fonction de l'entrée fournie.
- ▶ La vue et le modèle sont « **faiblement couplés** » puisque le contrôleur se trouve entre eux.
 - ▶ Les fonctionnalités peuvent être ajoutées au modèle et testées bien avant d'être ajoutées à la vue.

Exercice de synthèse: Depuis description vers UML

- ▶ On vous donne le glossaire suivant des termes décrivant notre application désirée. Vous êtes demandés de proposer un diagramme UML relatif en utilisant, **obligatoirement un Design Pattern (DP)** au moins.
 - ▶ Veuillez indiquer le DP utilisé et d'expliquer votre choix. ([Voici un lien pour travail collaboratif en ligne](#))
- ▶ **utilisateur:** une personne qui utilise ou exploite l'application. Les utilisateurs peuvent aussi être des propriétaires.
- ▶ **propriétaire:** un utilisateur qui possède l'élément.
- ▶ **contact:** Chaque contact est associé à un e-mail et à un nom d'utilisateur unique. Les propriétaires ajoutent les contacts à leur liste de contacts. Les propriétaires peuvent modifier/supprimer des contacts de leur liste de contacts. Un contact est un emprunteur potentiel.
- ▶ **contacts:** une liste de tous les contacts qu'un propriétaire a ajoutés dans l'application. Ces contacts sont tous des potentiels emprunteurs.
- ▶ **emprunteur:** contact qui emprunte un article et le retourne éventuellement au propriétaire de l'article.
- ▶ **élément:** un objet appartenant au propriétaire et pouvant être emprunté par un contact depuis la liste de contacts du propriétaire
- ▶ **état_elément:** l'état actuel de l'élément, qui par défaut est «Disponible» et peut être modifié à tout moment par le propriétaire. Lorsqu'un article est emprunté, le propriétaire change le statut de l'article en «Emprunté» et lorsque l'article est retourné, le propriétaire rétablit le statut en «Disponible».
- ▶ **disponible:** l'article n'est pas emprunté actuellement, il est donc disponible pour être emprunté.
- ▶ **emprunté:** l'article est actuellement emprunté à l'un des contacts du propriétaire.
- ▶ **Liste_eléments:** une liste d'éléments que le propriétaire a actuellement répertoriés, éventuellement filtrée par une condition, c'est-à-dire «Disponible» ou «Emprunté».
- ▶ **nom_utilisateur:** une séquence unique de caractères qui identifie un contact.
- ▶ **titre_elément:** le titre de l'élément.
- ▶ **Fabricant_elément:** le fabricant, la marque ou le créateur de l'élément.

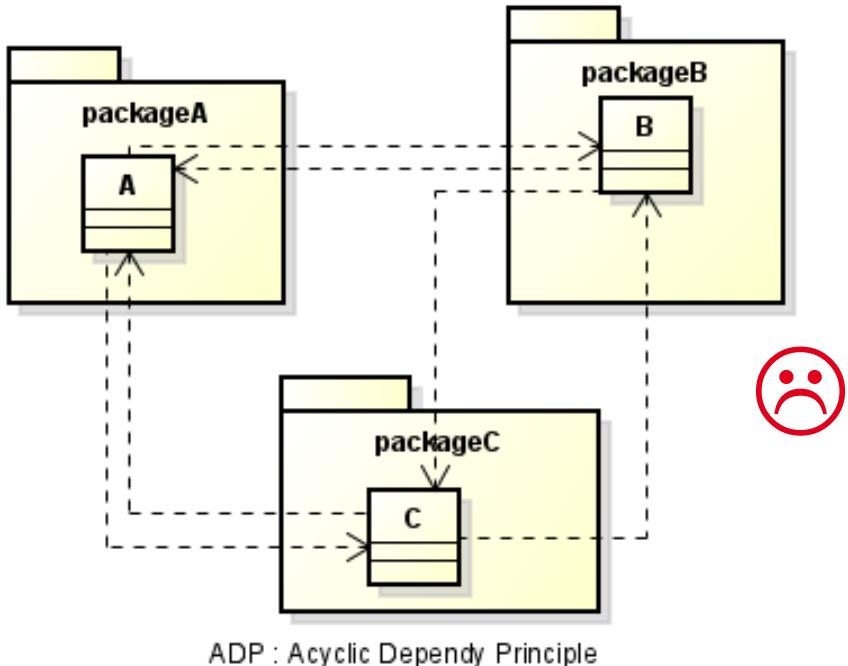
VII- Les principes GRASP

1. Faible couplage
2. Forte Cohésion
3. Règle de l'expert
4. Utiliser le créateur
5. Gérer le contrôleur
6. Appliquer le polymorphisme
7. Ne pas parler aux inconnus
8. Indirection, Pure fabrication et Points de variation
9. Autres Principes
10. Odeurs du code (Code-Smells)

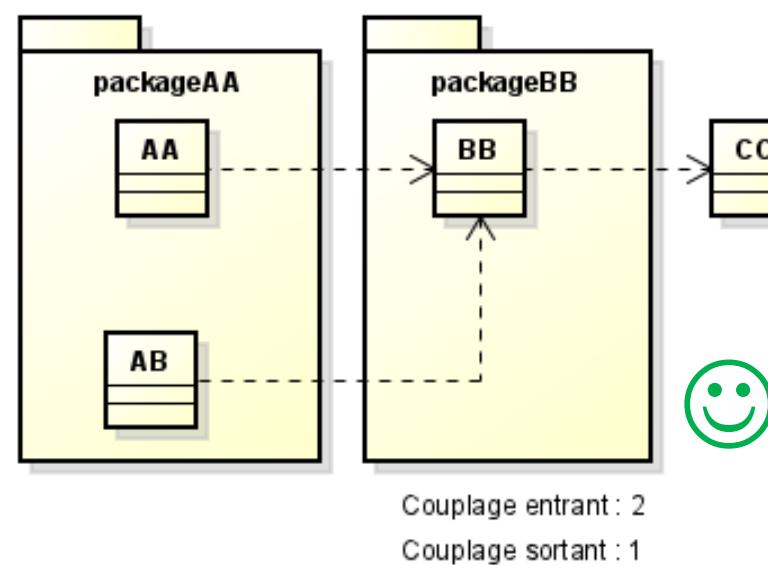
Les 10 Principes du GRASP

1. Faible couplage

- ▶ Ne pas avoir un **plat de spaghetti** de dépendances lorsqu'on ne réfléchit pas assez
- ▶ Ne pas avoir **une usine à gaz**; c'est lorsqu'on réfléchit un peu trop!
- ▶ Ici on ne pas modifier une classe sans impacter les autres



- ▶ On peut même calculer des métriques du niveau de couplage
 - ▶ Nombre de couplages entrants
 - ▶ Nombre de couplages sortant
- ▶ Dans le monde Java, il ya Jdepend qui le permet (Ndepend sous .NET)
- ▶ Et bien sur pas de dépendances cycliques

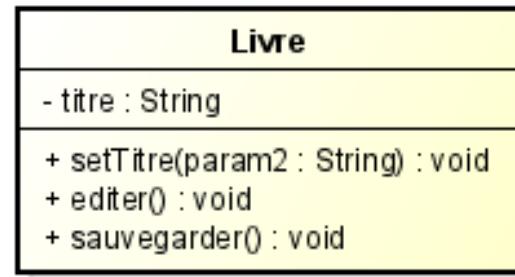


Les 10 Principes du GRASP

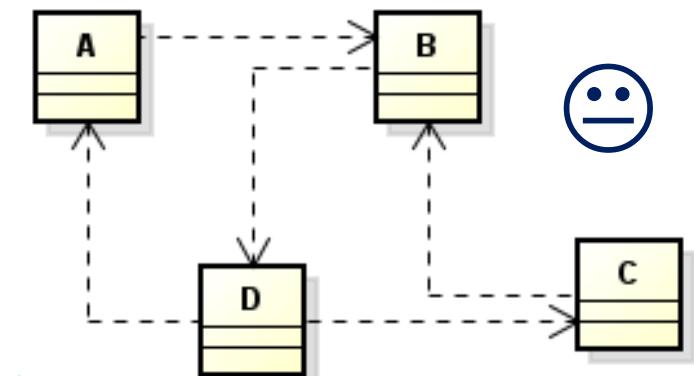
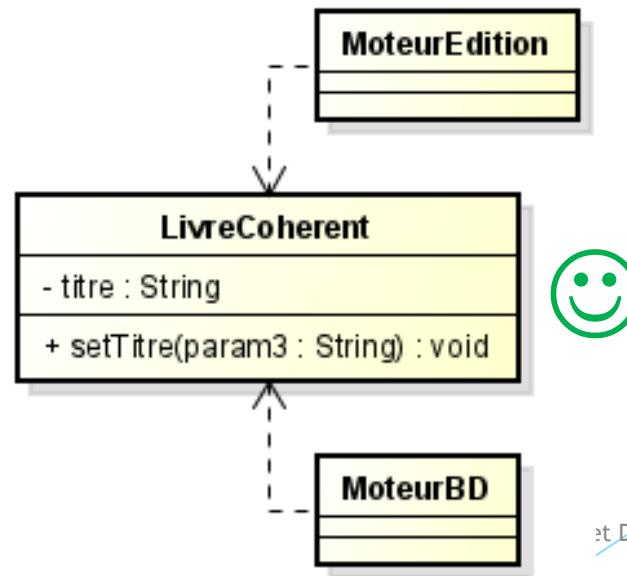
2. Forte Cohésion

- ▶ Equivalent au principe SRP. Ici on est un peu moins stricte dans le sens où Craig Larman nous conseille juste d'éviter à avoir plusieurs responsabilités car la classe concernée sera difficile à gérer, modifier, ...
- ▶ Alors il faudrait améliorer la cohérence et donc notre livre est beaucoup plus cohérent
 - ▶ Les accès aux données sont isolé de la partie édition graphique dans une interface
- ▶ Si on exagère dans la cohérence, alors on tomber dans une assez lourde décomposition basée sur des micro-services (un service atomique par classe)
 - ▶ On risque de se retrouver encore un fois dans des dépendances cycliques

▶ On cherche donc un juste milieu



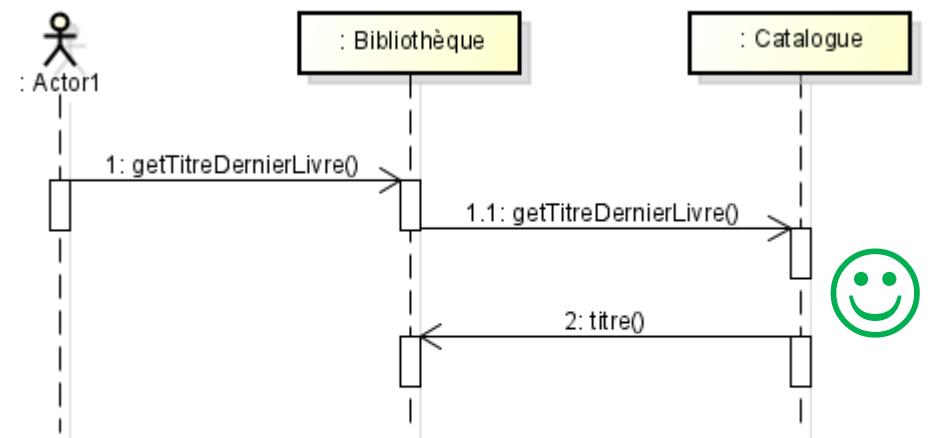
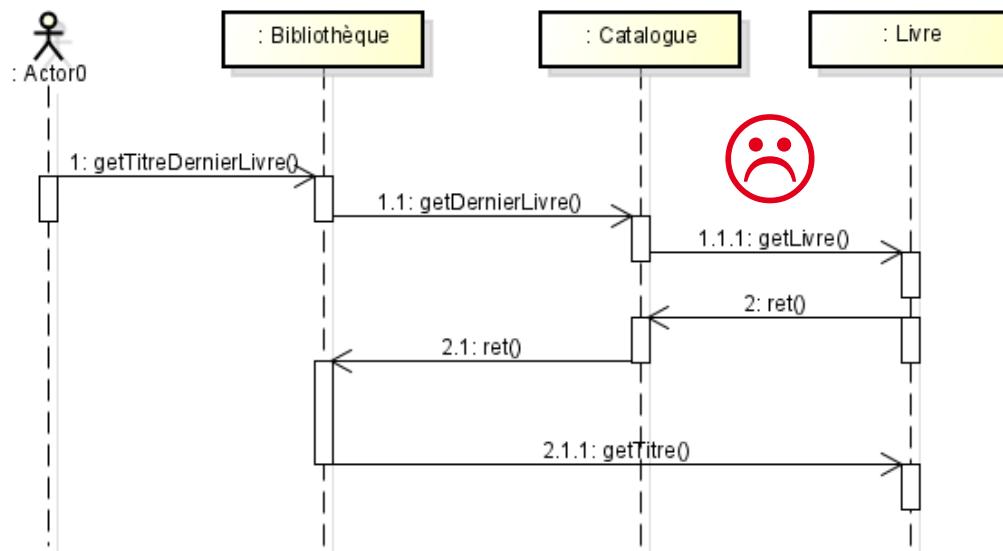
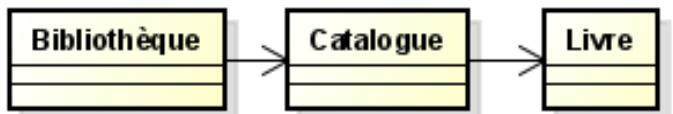
SRP ==



Les 10 Principes du GRASP

3. Règle de l'expert

- ▶ S'applique clairement lorsqu'on fait la dynamique du système
- ▶ **Ne pas respecter la règle de l'expert :** Cela revient à traverser tout le chemin depuis la bibliothèque → Catalogue → Livre via des méthodes présentes, quand une entité de l'extérieur cherche à avoir un livre.
 - ▶ Or celui qui était sensé être expert des Livres, est normalement la classe Catalogue
- ▶ **Respecter l'expert:** On laisse le catalogue (l'objet le mieux placé pour faire le boulot) masquer la façon dont il accède aux livres et de les présenter
 - ▶ Ceci est offert par l'encapsulation (private, protected)

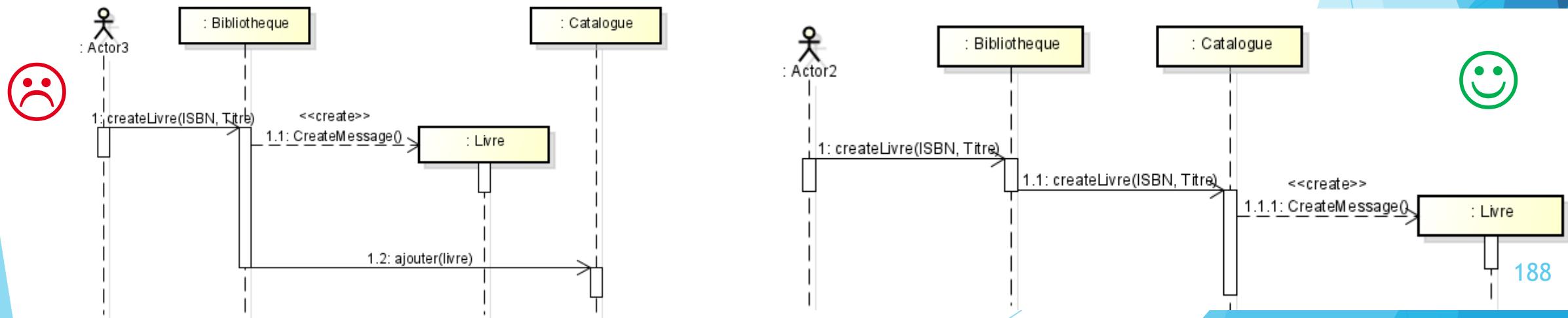
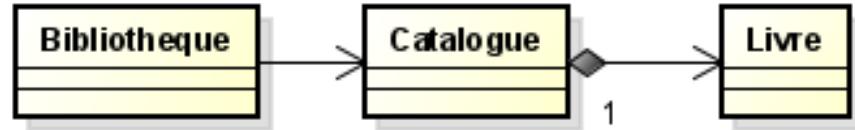


L'objet possède l'attribut
L'objet sait calculer
L'objet est le mieux placé

Les 10 Principes du GRASP

4. Utiliser le créateur

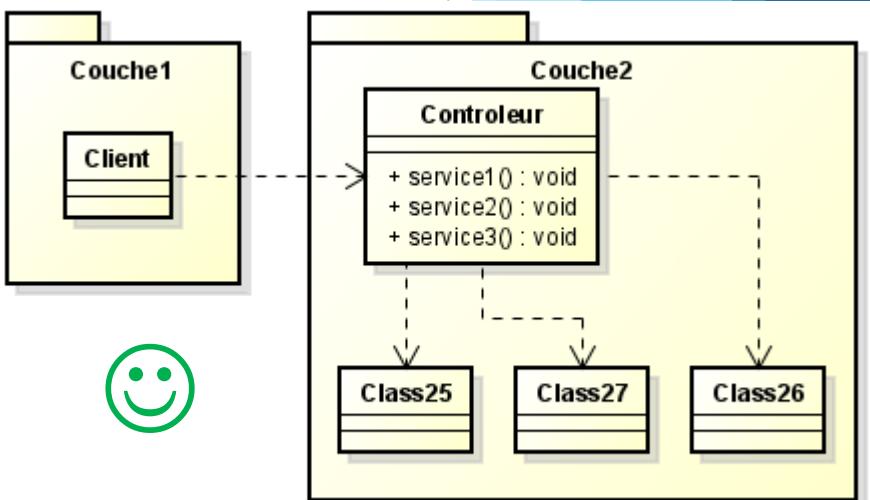
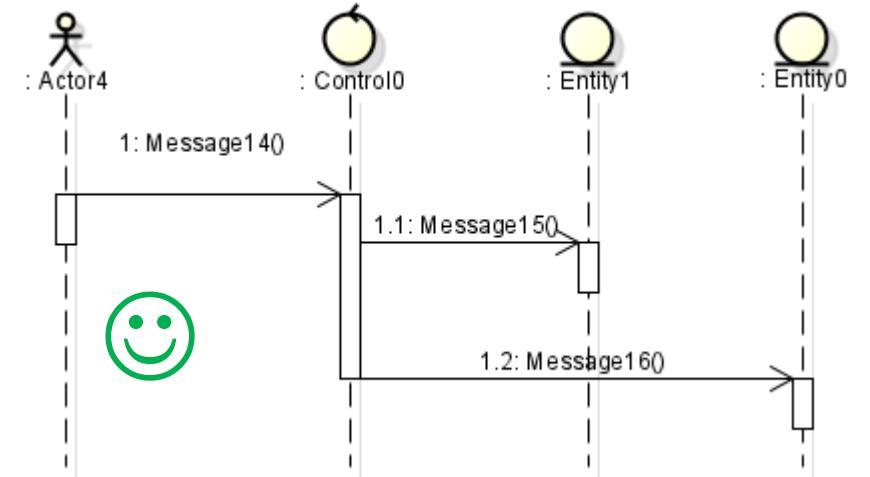
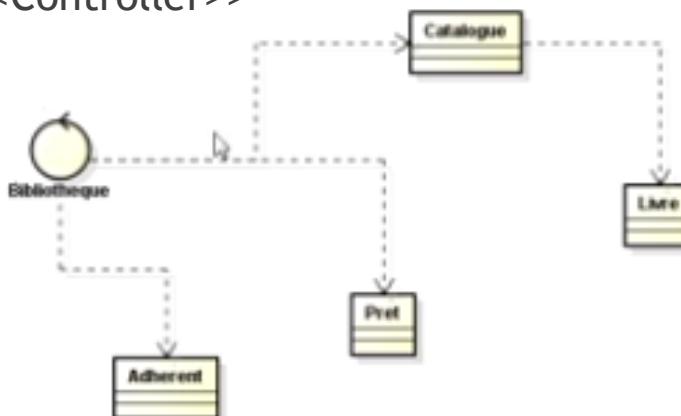
- ▶ Le livre appartient à un seul catalogue
- ▶ quant on supprime le catalogue, on supprime le livre
- ▶ **Ne pas respecter le principe de créateur:** La bibliothèque crée elle-même le livre (avec un new())
 - ▶ puis signale au Catalogue pour le rajouter
- ▶ **Si on respecte la règle de créateur,** quand on crée le livre, la bibliothèque doit s'adresser au Catalogue
 - ▶ Le new() sera dans le Catalogue



Les 10 Principes du GRASP

5. Gérer le contrôleur

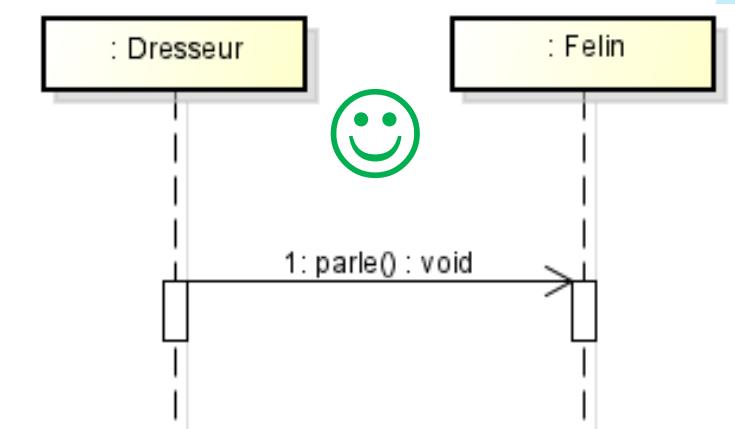
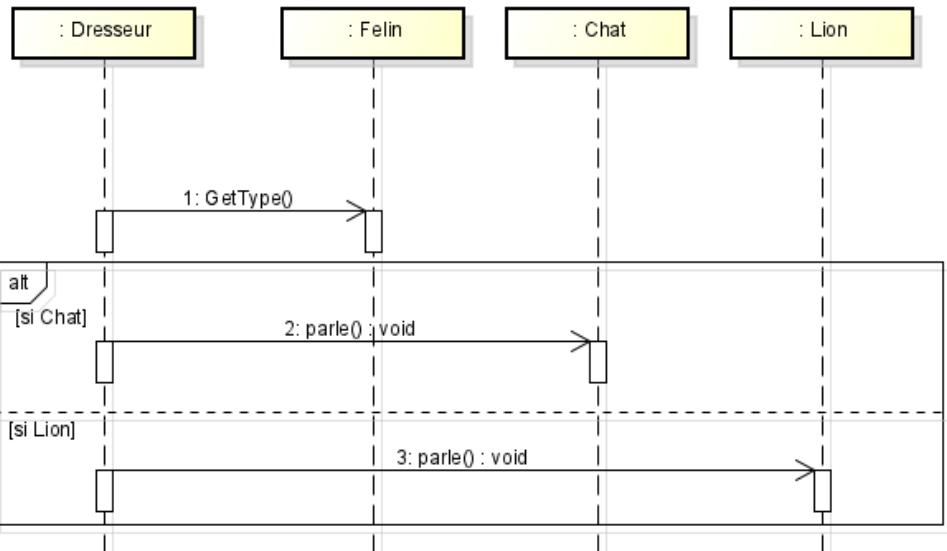
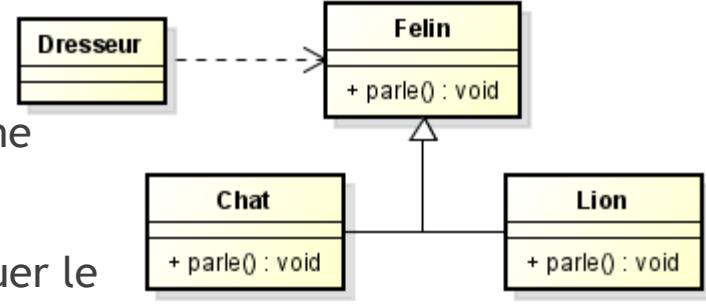
- ▶ Le contrôleur : C'est ce qui **reçoit les demandes de services et orchestre le comportement des entités**
- ▶ Il permet d'éviter qu'une classe dans une couche donnée connaît trop des contenus présents dans une couche
 - ▶ La solution c'est d'intercaler un contrôleur qui connaît très bien les classes internes
- ▶ **Un soucis:** plus le contrôleur est équipé de services, il devient moins cohérent
 - ▶ Donc ne respecte moins le principe GRASP numéro 2 de cohérence
- ▶ **En pratique:** plus une classe donnée va exposer de services depuis notre système, on pourrait décider un instant donné de lui affecter le stéréotype <>Controller<>



Les 10 Principes du GRASP

6. Appliquer le polymorphisme

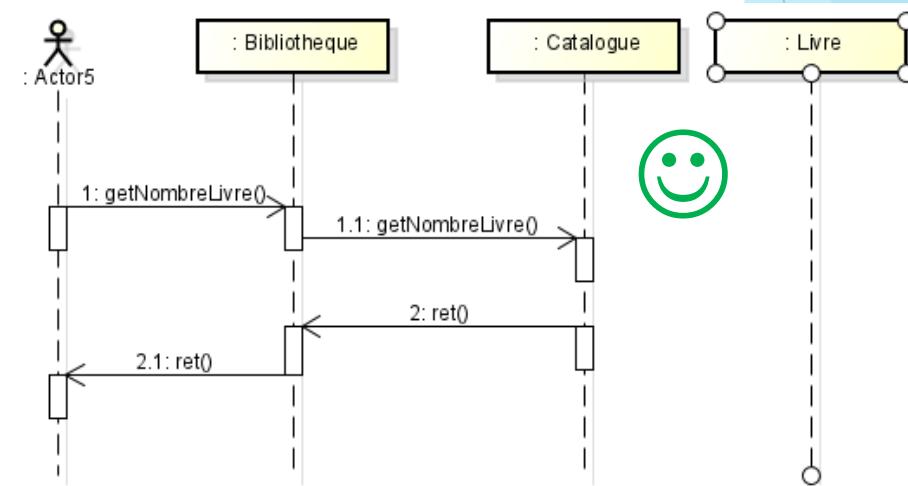
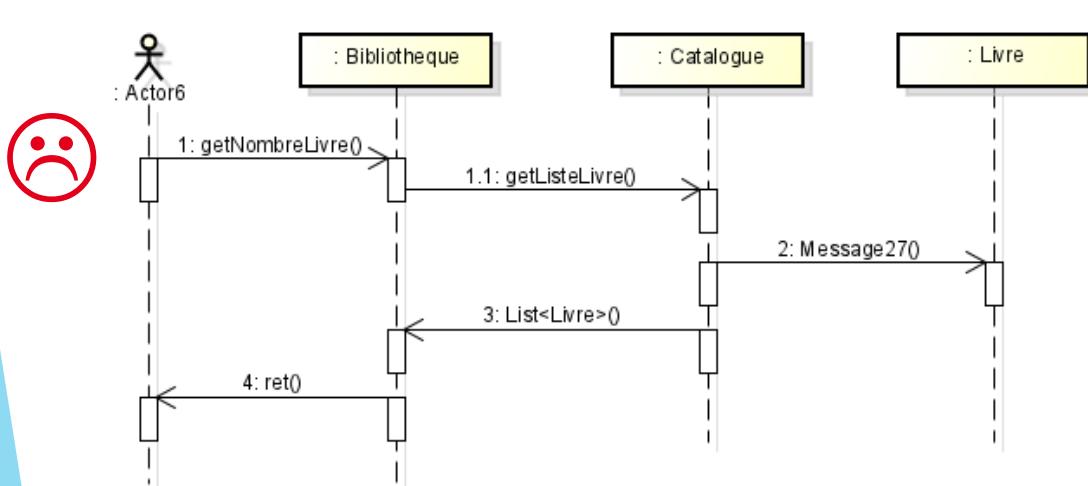
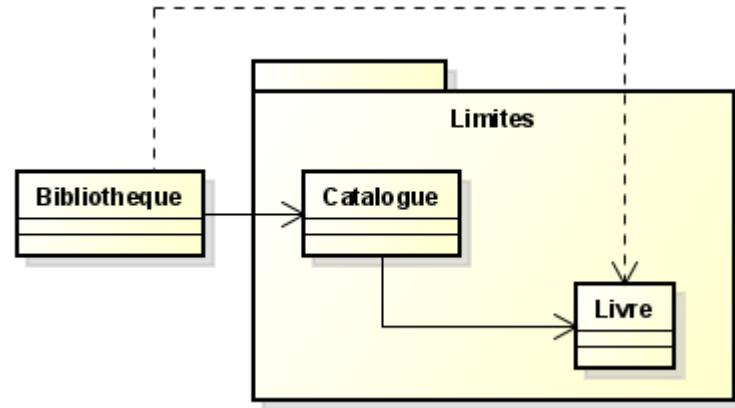
- ▶ Ne pas avoir à tester le type de l'objet mais plutôt exploiter le polymorphisme
 - ▶ Donc une classe externe telle que Dresseur est en contact
- ▶ **Ne pas respecter le principe du polymorphisme:** Tester le type et puis appliquer le traitement spécifique
- ▶ **Respecter le principe du polymorphisme:** C'est tout simplement faire appel au type parent
 - ▶ **Risque:** Un coté indéterminé
 - ▶ On créer un chat et on le présente d'une façon masque au Dresseur sous le Type de la classe mère Félin
 - ▶ Il y a sous Java la le test (`f instanceof Chat`) qui permet de nous aider à vérifier le type



Les 10 Principes du GRASP

7. Ne pas parler aux inconnus

- ▶ Normalement, la bibliothèque étant externe au package, elle ne doit pas “parler” au livre.
 - ▶ C'est une dépendance qui offre un type/entité qu'on aurait du utiliser
- ▶ Tout le monde est d'accord, mais dans la pratique, il est vrai que moins de respect est remarquable



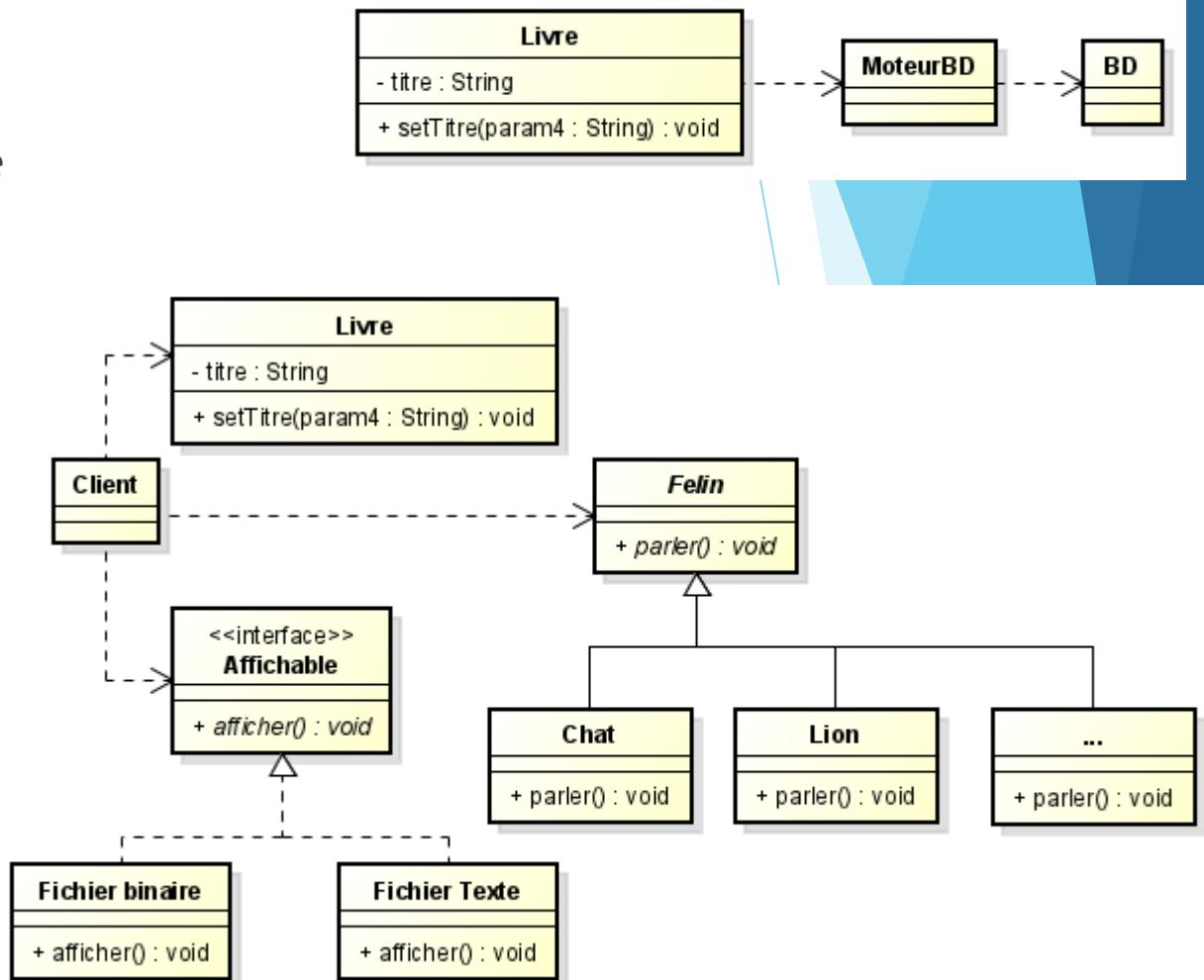
Les 10 Principes du GRASP

8. Indirection, 9. Pure fabrication

- ▶ Pour éviter le couplage entre livre et DB, on place
 - ▶ un **in-directeur ou objet intermédiaire**
- ▶ On crée un “truc” de façon **purement artificielle**
- ▶ Deux règles qui collaborent dans un même sens

10. Points de variation

- ▶ Placer des points de stabilité dans la conception
- ▶ **Respecter l'encapsulation**
 - ▶ Demander le titre du livre depuis le livre
- ▶ **Utiliser l'héritage et le polymorphisme**
 - ▶ Ne pas toucher au sous-types
 - ▶ Donc la classe Félin est hyper-stable → Abstraite
- ▶ **Le must: utiliser les interface**
 - ▶ C'est le meilleur des points de stabilité



Autres principes

- ▶ **Généralisation** - un principe de conception dictant que le comportement doit être généralisé dans la mesure du possible. Les solutions générales peuvent être appliquées plus largement.
- ▶ **information hiding - Dissimulation d'informations** - un principe de conception dictant que l'information qui n'est pas essentielle en dehors d'une classe doit être cachée à d'autres classes. Parfois synonyme d'encapsulation.
- ▶ **Separation of concerns - Séparation des préoccupations** - un principe de conception dictant qu'un programme doit séparer différentes préoccupations ou fonctions en objets ou processus distincts. Permet un entretien plus facile et un couplage moins intence entre les objets.
- ▶ **principle of least knowledge - Principe de la moindre connaissance** - un principe de conception stipulant que les classes devraient connaître et interagir avec le moindre d'autres classes que possible. Aussi appelé la loi de Demeter. Semblable au principe du « besoin de savoir » dans les organisations

Odeurs de code/ anti-modèle

- ▶ **Classe de données** - une très petite classe avec un peu plus que des données.
- ▶ **Touffe de données** - un groupe de données fréquemment associées qui seraient mieux encapsulées comme un objet.
- ▶ **Changement divergent** - les développeurs se retrouvent à changer une classe de différentes façons, souvent contradictoires. Indique une mauvaise séparation des préoccupations.
- ▶ **Code dupliqué** - le même code, parfois avec de petits changements, se trouve à plusieurs endroits dans le logiciel. Indique que le développeur pourrait mieux utiliser la généralisation.
- ▶ **Envie de fonctionnalité** - deux classes qui seraient mieux ensemble, généralement indiquées en invoquant fréquemment les méthodes de l'autre.
- ▶ **Intimité inappropriée** - deux classes qui devraient avoir un autre degré de séparation.
- ▶ **Grande classe** - une classe est très grande. Indique qu'il pourrait être nécessaire d'avoir une meilleure séparation des préoccupations. Ce qui est considéré comme « grand » peut varier considérablement selon la complexité.

Odeurs de code/ anti-modèle (suite)

- ▶ **Méthode longue** - une méthode est très grande. Indique qu'il pourrait être nécessaire d'avoir une meilleure séparation des préoccupations. Ce qui est considéré comme grand peut varier considérablement, par exemple l'initialisation des éléments visuels est généralement lourd en code
- ▶ **Longue liste de paramètres** - une méthode ayant une longue liste de paramètres, rend la méthode difficile à utiliser correctement. Ceci peut parfois être corrigé en passant des objets paramètres qui encapsulent des paramètres communs.
- ▶ **Obsession primitive** - utilisation inappropriée ou étendue des types de données primitifs.
- ▶ **Mandats refusés** - les sous-classes héritent de méthodes dont elles n'ont pas besoin. Ceci pourrait être une indication que la méthode devrait être définie dans les sous-classes
- ▶ **Déclarations "switch"** - utiliser des instructions de type « switch...case » pour dicter un comportement est à remplacer par le polymorphisme. Ceci est une meilleure solution.

References

- ▶ Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994), “*Design Patterns: Elements of Reusable Object-Oriented Software*”. Upper Saddle River, NJ: Addison-Wesley Professional
- ▶ Fowler, M. (1999), *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts: Addison-Wesley
- ▶ Robert Nystrom (2014), "Game Programming Patterns"
- ▶ Craig Larman (1997), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis & Design"
- ▶ Plan de formation M2ii Design Patterns:
 - ▶ <https://www.m2iformation.fr/formation-objet-design-patterns/DES-PT/>
 - ▶ <http://www.vishalchovatiya.com/category/design-patterns/>