CONCEPTION ET PROGRAMMATION OBJET AVANCEES

Les Principes SOLID de Robert Cecil Martin, dit 'Uncle Bob'

SOLID est un acronyme représentant cinq principes dont la mise en œuvre lors d'un développement orienté objet, vise à **améliorer la cohésion**, **diminuer le couplage** et **favoriser l'encapsulation**, afin que l'application résultante soit fiable et robuste.

- La **cohésion** traduit à quel point les pièces d'un seul composant sont en relation les unes avec les autres. Un module est cohésif lorsqu'au haut niveau d'abstraction il ne fait qu'une seule et précise tâche. Plus un module est centré sur un seul but, plus il est cohésif. Une bonne conception vise à **améliorer la cohésion**.

- Le **couplage** est une métrique qui mesure l'interconnexion des modules. Deux modules sont dits couplés si une modification dans l'un des deux modules demande une modification dans l'autre module. Une bonne conception vise à **diminuer le couplage** (faible couplage).

- L'**encapsulation** vise à intégrer à un objet tous les éléments nécessaires à son fonctionnement, que ce soient des fonctions ou des données. Le corolaire est qu'un objet 'devrait' – et non pas 'doit' comme c'est souvent expliqué – masquer la cuisine interne de la classe, pour exposer une interface propre, de façon à ce que ses clients puissent manipuler l'objet et ses données sans avoir à connaitre le fonctionnement interne de l'objet. Une bonne conception vise à **favoriser l'encapsulation**.

Les cinq principes SOLID sont :

S **Responsabilité unique (S**ingle Responsibility Principle**) SRP**
Une classe doit avoir une et une seule responsabilité

O **Ouvert/fermé (O**pen/closed principle**) OCP**
Une classe doit être ouverte à l'extension, mais fermée à la modification

L **Substitution de Liskov (L**iskov substitution Principle**) LSP**
Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme

I **Ségrégation des interfaces (I**nterface segregation principle**) SIP**
Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale

D **Inversion des dépendances (D**ependency Inversion Principle**) DIP**
Il faut dépendre des abstractions, pas des implémentations

S **Responsabilité unique (S**ingle Responsibility Principle**) SRP**

Signification : Une classe doit avoir une et une seule responsabilité

Trop de responsabilités sur une même classe cause des problèmes de lisibilité, de maintenance et d'utilisation.

Avantages

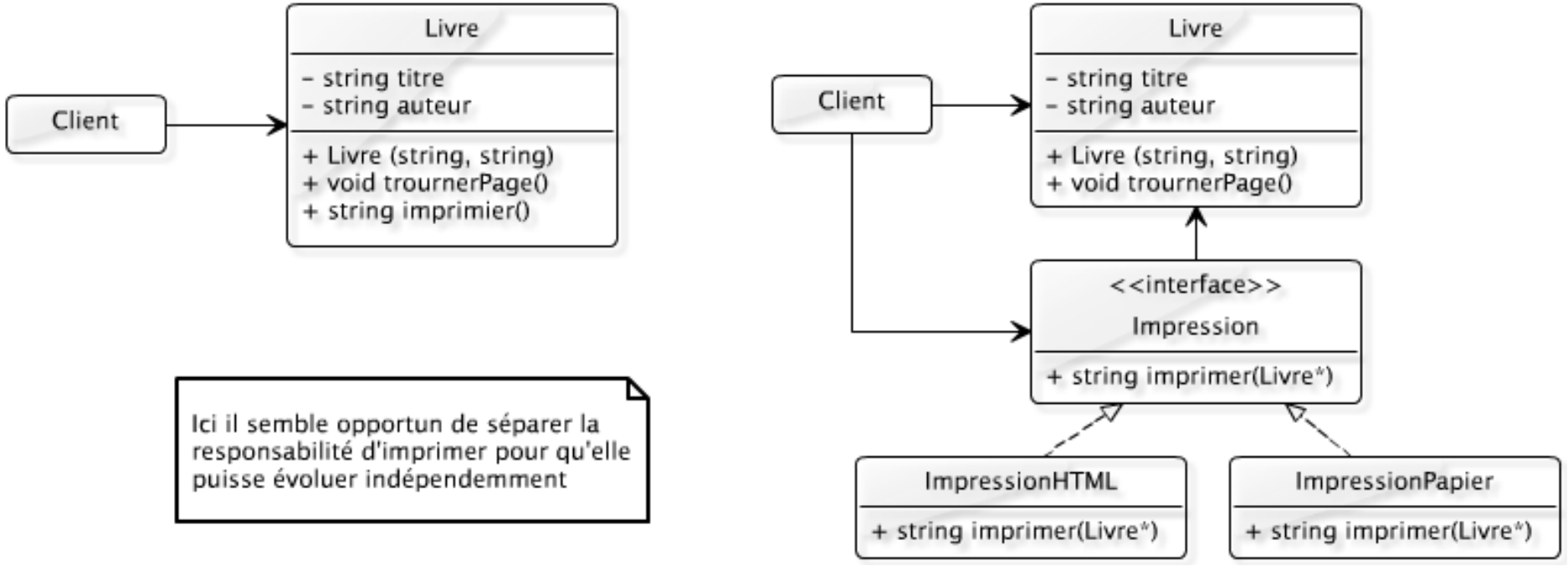
Ce principe recommande de répartir les responsabilités et donc, qu'une classe ne change que pour une seule raison, ce qui :

- Diminue la complexité du code
- Augmente la lisibilité de chaque classe
- Améliore l'encapsulation et la cohésion

En pratique

Pour une classe de taille importante, lister toutes les méthodes et regrouper celles dont le nom ou les actions semblent être de la même famille. Si plusieurs groupes apparaissent dans une classe, c'est un bon indicateur pour que la classe soit reconsidérée. De même, si un petit groupe de méthodes appellent la base de données, ou un sous-ensemble de méthodes utilisent une API spécifique, ou encore si certains membres sont appelés uniquement par une fonction, ou par un sous-ensemble de fonctions, alors ces situations correspondent vraisemblablement à des responsabilités spécifiques qu'il convient de regrouper dans des classes spécifiques.

Exemple illustratif



Note

Attention à ne pas aller trop loin dans la séparation des responsabilités et tomber dans l'excès inverse. A savoir, qu'au lieu de placer N méthodes dans une classe, définir N classes, chacune en charge d'une seule desdites méthodes.

O Ouvert/fermé (**O**pen/closed principle**) OCP**

Signification : Une classe doit être ouverte à l'extension, mais fermée à la modification

L'idée est qu'une fois qu'une classe a été approuvée via des revues de code, des tests unitaires et d'autres procédures de qualification, elle ne doit plus être modifiée mais seulement étendue. Ainsi, on pourra ajouter une nouvelle fonctionnalité :

- En ajoutant des sous classes (cf. Ouvert à l'extension)
- Mais sans modifier le code de la classe existante (cf. Fermé à la modification)

Avantages

- Le code existant n'est pas modifié, ce qui garantit sa fiabilité pour les appelants/usagers
- Les classes ont plus de chance d'être spécialisées
- L'ajout de nouvelles fonctionnalités est simplifié

En pratique

Le principe ouvert/fermé oblige à faire bon usage de l'abstraction et du polymorphisme. Du coup on préfèrera manipuler des objets de classes abstraites (plutôt que de classes concrètes) et on utilisera le polymorphisme. Ainsi, cela diminuera le couplage avec moulte (sous-)classes.

Exemple illustratif

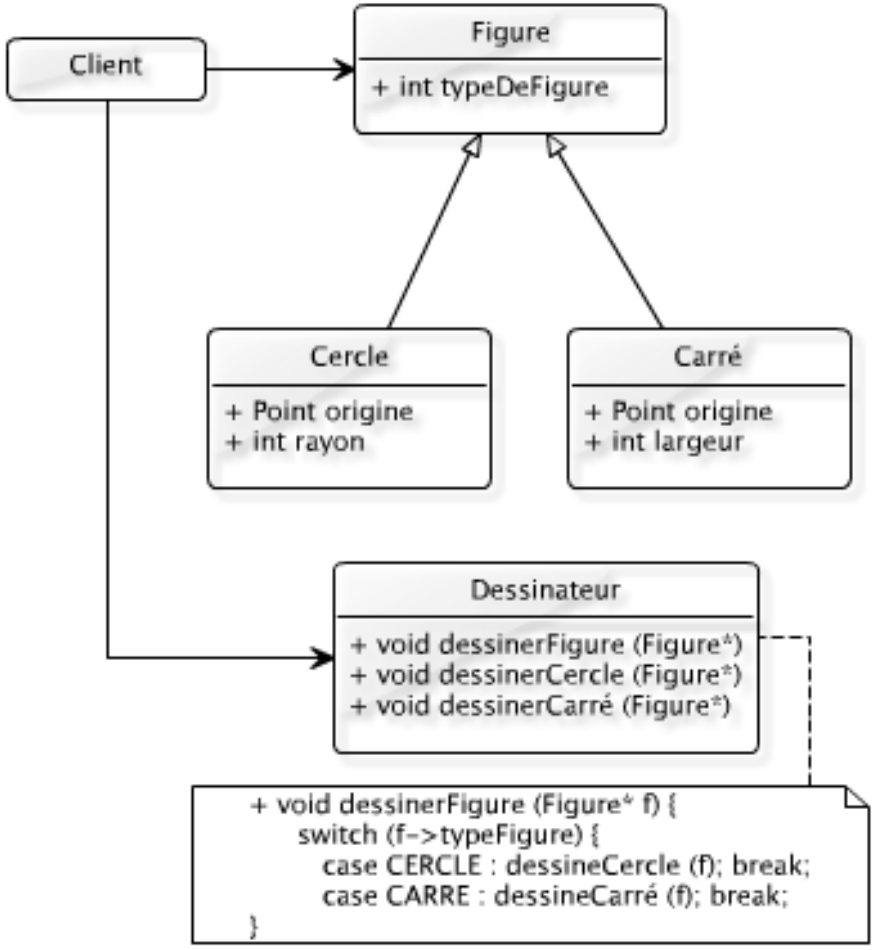


Schéma 1 : ne respecte pas le principe Ouvert/Fermé

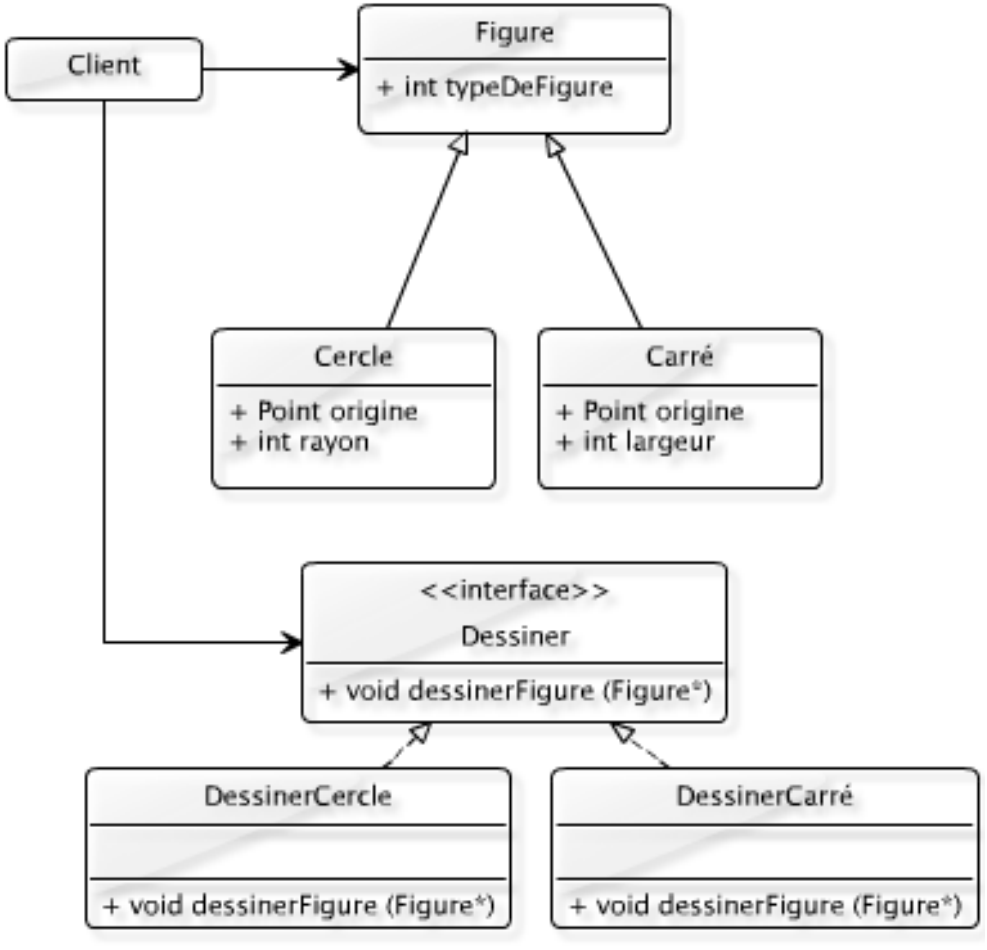


Schéma 2 : respecte le principe Ouvert/Fermé

La classe Dessinateur assume une unique responsabilité (schéma 1) de même que la classe Dessiner (schéma 2). Par contre, le schéma 1 ne respecte pas le principe Ouvert/Fermé. Pour une nouvelle figure, il faudra reprendre le code de la méthode **`Dessinateur::dessinerFigure()`**. Alors que, dans le schéma 2 la classe **`Dessiner`** définit une interface abstraire de comportement et diverses implémentations sont possibles par héritage / polymorphisme y compris pour une nouvelle figure.

L Substitution de Liskov (Liskov substitution Principle) LSP

Signification **Une instance de type T doit pouvoir être remplacée par une instance de type S, tel que S est sous-type de T, sans que cela ne modifie la cohérence du programme**

Les instances d'une classe doivent être remplaçables par des instances de leurs sous-classes sans altérer le programme.

Avantages

- Il n'est pas nécessaire de redéfinir dans les sous-classes ce qui est déjà défini dans les super classes.
- Définition d'une hiérarchie de classe, ce qui permet de considérer les différents enfants comme étant du type parent.

En pratique

Ce principe est étroitement relié à la méthodologie de programmation par contrat :

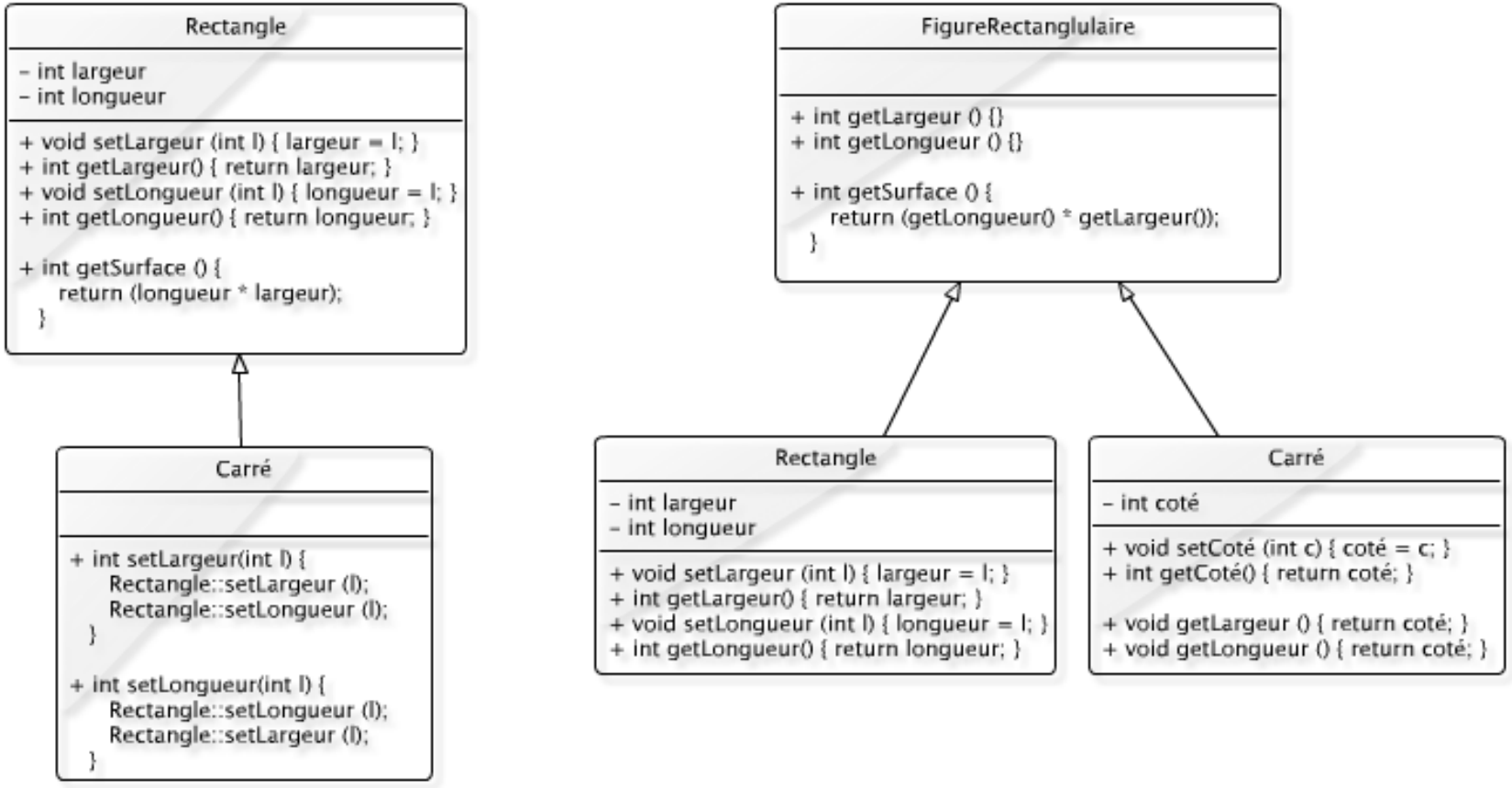
- Les préconditions ne peuvent pas être renforcées dans une sous-classe. Cela signifie que l'on ne peut pas avoir une sous-classe avec des préconditions plus fortes que celles de sa superclasse.
- Les postconditions ne peuvent pas être affaiblies dans une sous-classe. Cela signifie que l'on ne peut pas avoir une sous-classe avec des postconditions plus faibles que celles de sa superclasse.

De plus, ce principe fait que des exceptions d'un nouveau type ne peuvent pas être levées par les méthodes des sous-classes, sauf si ces exceptions sont elles-mêmes des sous-types des exceptions définies dans les méthodes de la superclasse.

Note

Une fonction utilisant la connaissance de la hiérarchie de classe viole le principe car elle ne devrait utiliser que la référence à la classe de base. Càd que `is_instance_of()` ou instructions similaires, sont à proscrire. En effet, l'usage d'un appel de la sorte dans une méthode viole le principe ouvert/fermé car ladite méthode doit être modifiée quand on créera une classe dérivée de la classe de base.

Exemple illustratif



La solution de gauche ne respecte pas le principe de Liskov. Le test ci-dessous ne peut pas être appelé pour un objet **`Carré`** :

```
void testRectangle (Rectangle* r) {
   r->setLargeur (2); r->setLongueur (3);
   if (r->getSurface() != 3*2)
         cout << ”bizarre!”;
}
```

En fait la solution de gauche n’est pas bien modélisée, car un **`Carré`** n’est pas un **`Rectangle`**. Les tests seraient à dissocier.

```
void testRectangle (Rectangle* r) {
     r->setLargeur (2); r->setLongueur (3);
     if (r->getSurface() != 3*2) cout << ”bizarre!”;
}

void testCarré (Carré* c) {
     c->setCoté (2);
     if (r->getSurface() != 2*2) cout << ”bizarre!”;
}
```

I Ségrégation des interfaces (Interface segregation principle) ISP

Signification Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale

Les interfaces sont les clefs de la pluggabilité, et une classe peut implémenter diverses interfaces (une interface est un sous ensemble des méthodes implémentées par une classe). Le principe incite à avoir des interfaces petites pour ne pas forcer des classes à implémenter les méthodes qu'elles ne veulent pas.

En pratique

- Découper les interfaces en responsabilités distinctes (SRP)
- Quand une interface grossit, se poser la question du rôle de l'interface
- Eviter de devoir implémenter des services qui n'ont pas à être proposés par la classe qui implémente l'interface
- Limiter les modifications lors de la modification de l'interface

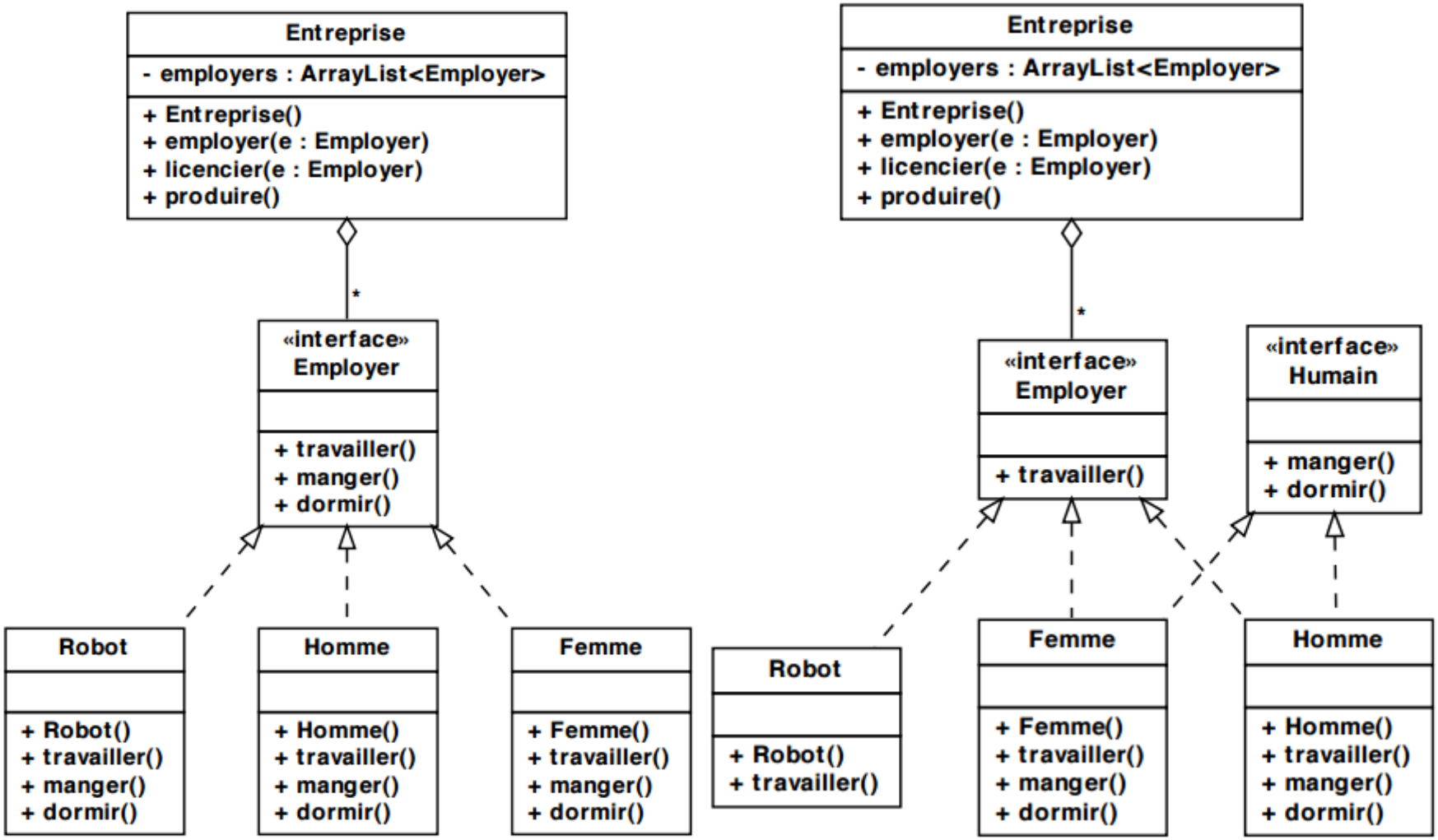
Avantages

- Le code existant est moins modifié ⇒ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Note

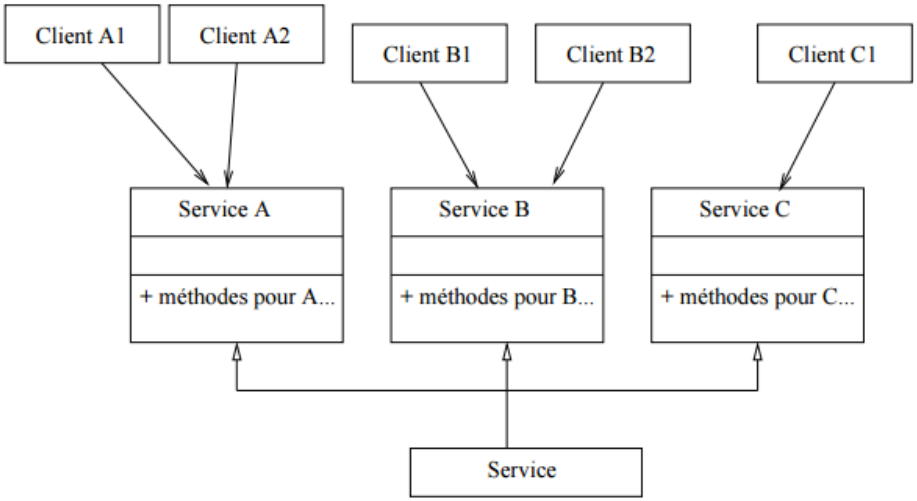
Peut amener à une multiplication excessive du nombre d'interfaces. **A l'extrême une interface avec une méthode ! ☹**

Exemple illustratif

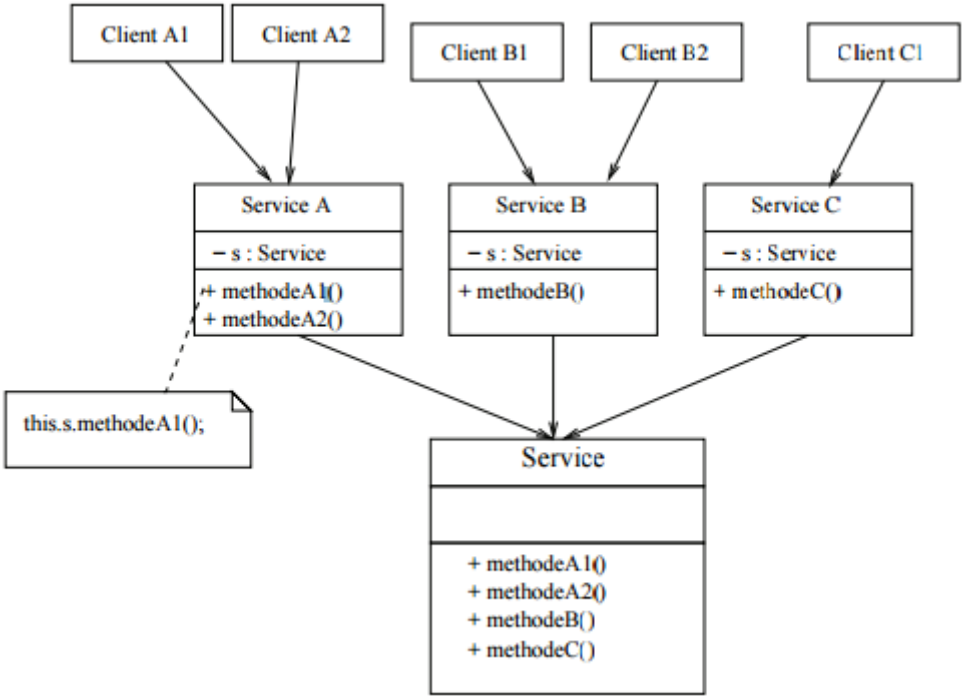


Sur le schéma de gauche, le principe de Ségrégation d'Interface n'est pas respecté. En effet, le **`Robot`** ne va pas associer de code aux méthodes **`manger()`** et **`dormir().`** Dans le schéma de droite, des fonctions ont été isolées dans une interface spécifique aux **`Humain`**, différenciées de celle(s) spécifique(s) aux **`Employé`**

Exemple de solution générique



Autre exemple de solution générique basé sur le design pattern **Adaptateur**



D Inversion des dépendances (**D**ependency Inversion Principle**) DIP**

Note : *Le principe d'inversion des dépendances est un peu « un principe secondaire ». En effet, il résulte d'une application stricte de deux autres principes, à savoir les principes Ouvert/Fermé (OCP) et Substitution de Liskov (LSP).*

Signification : Il faut dépendre des abstractions, pas des implémentations

Les deux bases de ce principe sont :

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

En pratique

- Les aspects "bas niveau" sont représentés pour le composant de haut niveau par des interfaces
- Les composants "bas niveau" doivent implanter les interfaces requises (on peut utiliser le pattern Adaptateur pour ce faire)
- Découpler les différents modules de votre programme
- Les lier en utilisant des interfaces
- Décrire correctement le comportement de chaque module

Avantages

- Les modules sont plus facilement réutilisables
- Permet de remplacer un module par un autre module plus facilement
- Simplification de l’ajout de nouvelles fonctionnalités
- L’intégration est rendue plus facile

L'idée est que chaque point de contact entre deux modules soit matérialisé par une abstraction.

Sur le schéma de gauche, le haut niveau Classe A, utilise classe B de bas niveau. Dans le schéma de droite, une interface est venue s'intercaler entre les deux modules. Dépendre d'interfaces et de classes abstraites plutôt que de classes de bas niveau

Utiliser l'abstraction, encore et toujours !!!!

Webographie

https://code.tutsplus.com/series/the-solid-principles--cms-634
https://fr.wikipedia.org/wiki/SOLID_(informatique)
http://philippe.developpez.com/articles/SOLIDdotNet/
http://pageperso.lif.univ-mrs.fr/~bertrand.estellon/java/cours.pdf
http://www.fil.univ-lille1.fr/~routier/enseignement/licence/coo/cours/conception_et_patterns-4parpage.pdf
http://live.yworks.com/graphity/

Bibliographie

« Agile Software Development, Principles, Patterns, and Practices » Robert C. Martin, 2002