Tutoriel pour apprendre les principes avancés de conception objet

Par Duy Anh PHAM

Date de publication : 6 mars 2018

La POO, ou programmation orientée objet, permet de représenter un concept, une idée ou une entité du monde physique par des entités appelées objets ayant des propriétés intrinsèques et exposant des opérations publiques pour les manipuler. Ce tutoriel a pour objectif de vous apprendre les principes de la programmation orientée objet.

Commentez

I - Introduction

Les objets peuvent être vus comme des briques rendant des services aux autres objets et donc réutilisables. L'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctions attendues. La conception est donc une étape importante pour modéliser les éléments du monde réel et les transcrire en code.

Cependant, la conception reste difficile dans le développement logiciel, car :

- les principes de base de la POO que sont l'encapsulation, l'héritage et le polymorphisme ne suffisent pas à guider dans la conception ;
- les design patterns qui sont des abstractions de solutions à des problèmes récurrents ne suffisent pas à former un tout cohérent pour la construction de designs complets.

Ce tutoriel présentera quelques principes utiles en matière de conception et les illustrera par des exemples dans le langage Java.

II - Les risques de dégénérescence de l'application

Lorsqu'une application est en PRODUCTION les phénomènes suivants sont observés pendant les activités de développement.

La rigidité

Chaque évolution risque d'impacter d'autres parties de l'application. Le coût de développement augmente et avec l'approche de la date de livraison, la qualité de code est négligée. Il fonctionne, mais le développeur ne prend pas le temps de refactorer, par conséquent le code devient difficile à modifier. Un cercle vicieux s'installe puisque le coût de modification devient élevé et le logiciel a peu de chances d'évoluer au risque d'entraîner des régressions.

La fragilité

Modifier une partie du code entraîne des erreurs dans une autre partie du logiciel qui devient peu robuste au changement avec un coût de maintenance élevé.

L'immobilité

Il est difficile d'extraire une partie de l'application pour la réutiliser. Le développeur a très vite tendance à copier/coller en modifiant les parties qui le concernent.

Les problèmes énoncés sont d'autant plus importants avec la volumétrie de l'application.

III - La source du problème : la gestion des dépendances

Les dégradations tirent leur origine dans la multiplication des dépendances et de leur architecture : les modules, packages et classes finissent par dépendre les uns des autres aboutissant au code spaghetti.

IV - Les objectifs de la conception

Les modifications de code sont inévitables avec l'évolution des besoins. La conception d'un logiciel a pour objectif d'amortir l'impact des dépendances et à aboutir aux qualités recherchées de :

- robustesse : les changements n'introduisent pas de régression ;
- extensibilité : l'ajout de fonctionnalités doit être facile ;

- réutilisabilité : il est possible de réutiliser certaines parties de l'application pour en construire d'autres.

Les principes présentés ci-dessous doivent aider à éviter les phénomènes de rigidité, fragilité et d'immobilité énoncés plus haut. Ainsi l'application sera capable de s'adapter au changement.

V - Principes SOLID pour l'organisation des classes

V-A - Single Responsability Principle : principe de responsabilité unique (S)

Ce principe stipule qu'il ne doit avoir qu'une raison et une seule raison de modifier une classe/un module. Il doit cadrer pour en donner une définition de la responsabilité et les indications sur la taille d'une classe.

Un nom simple donné à une classe est un indicateur du principe de responsabilité. Si le développeur éprouve des difficultés à nommer, alors il est fort probable que la classe endosse trop de responsabilités.

Le respect de ce principe augmente par ailleurs la cohésion de la classe qui est un concept indiquant le degré d'interdépendance entre variables d'instance et méthodes.

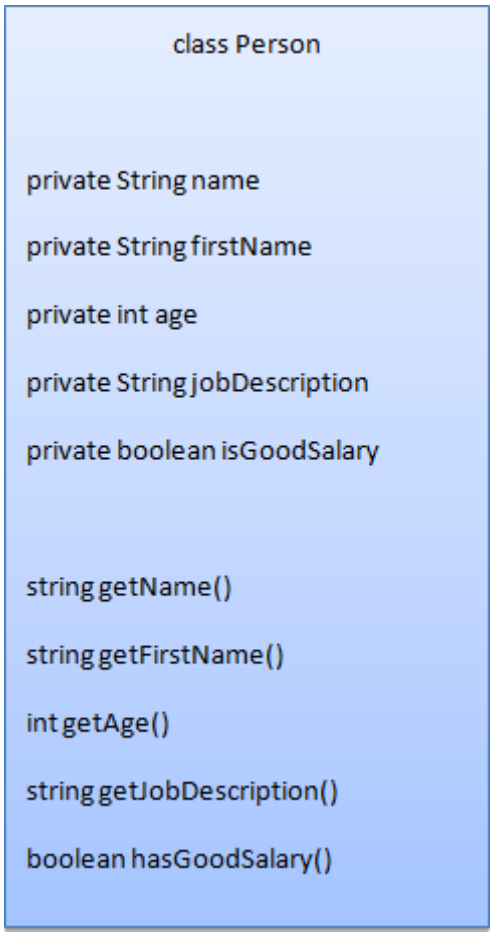
Ce principe simple d'énoncé est pourtant celui qui est le plus transgressé lors du développement logiciel. En effet, lorsqu'une évolution est demandée l'impact logiciel a été analysé et la modification se fait rapidement pour faire fonctionner le logiciel sans considérer un refactoring pour mieux organiser le code et le maintenir propre. À force, le module continue de grossir et augmente en responsabilité.

Quand appliquer : systématiquement.

Le maintien d'une haute cohésion donne un logiciel avec de nombreuses petites classes. Il n'y a pas plus de parties entre un logiciel fait de nombreuses petites classes et un logiciel avec peu de grandes classes puisqu'il y a autant de parties. Le développeur devra chercher s'il existe une fonction dont il a besoin et il est indiscutable qu'il est préférable de chercher dans une boîte à outils avec plusieurs compartiments étiquetés plutôt que dans un sac avec tous les outils en vrac.

Comment appliquer :

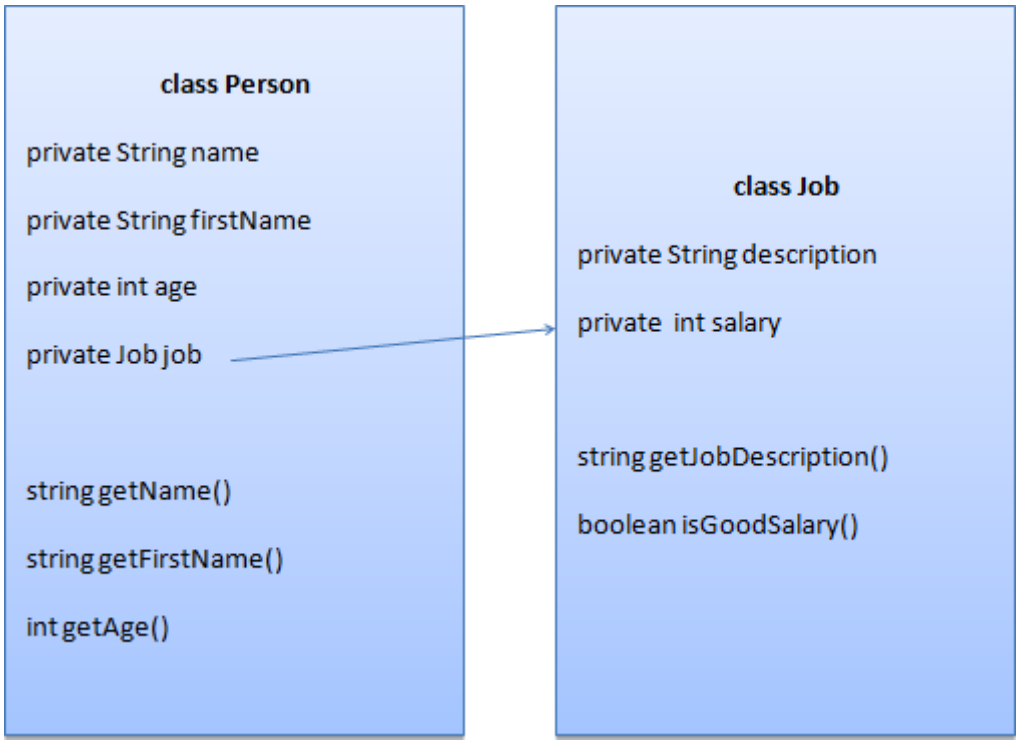
- est-ce que le nom donné à la classe est facile ?
- si cette fonctionnalité est ajoutée dans cette classe existante, est-ce que cela donne de nouvelles responsabilités à la classe si elle doit être modifiée dans le futur ?



Nous pouvons voir dans la classe Person qu'elle possède deux responsabilités : des données et opérations propres à une personne, mais aussi pour un emploi.

Nous pourrions imaginer que la classe Person contienne aussi des informations/méthodes sur la maison où elle vit (avec une adresse, rue, ville, pays), une voiture qu'elle possède (marque, modèle. numéro de plaque d'immatriculation). Tout ceci alourdirait la classe Person et ferait supporter trop de responsabilités.

Il est plus efficace de refactorer de sorte qu'il y ait deux classes distinctes.



V-B - Open/close principle : principe d'ouverture/fermeture

La rigidité et la fragilité vues précédemment viennent de l'impact d'un changement d'une partie de l'application sur d'autres parties avec des effets indésirables. Bertrand Meyer, créateur du langage orienté objet Eiffel, a stipulé que tout module (package, classe, méthode) doit être :

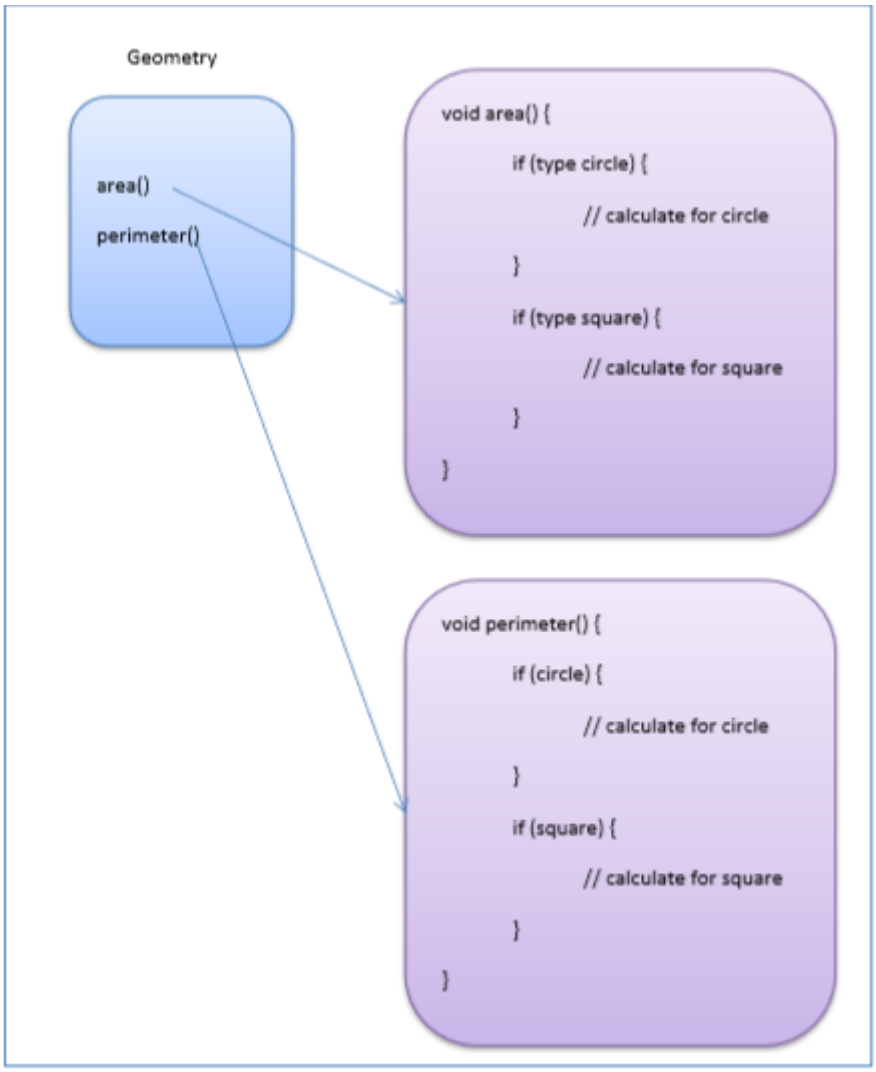
- ouvert aux extensions : on peut ajouter des fonctionnalités non prévues à la création ;
- fermé aux modifications : les changements introduits ne modifient pas le code existant.

Autrement dit, l'extensibilité se traduit par de l'ajout de code uniquement. Une fois le code produit, testé unitairement, qualifié par des procédures et enfin livré en production, le seul moyen de modifier est d'étendre le code pour s'assurer que le code existant ne sera pas altéré au risque d'entraîner des régressions.

L'abstraction et polymorphisme sont les moyens pour y parvenir en faisant reposer le code stable sur une abstraction d'une entité variable pouvant être amenée à évoluer.

Utilisation de la délégation abstraite

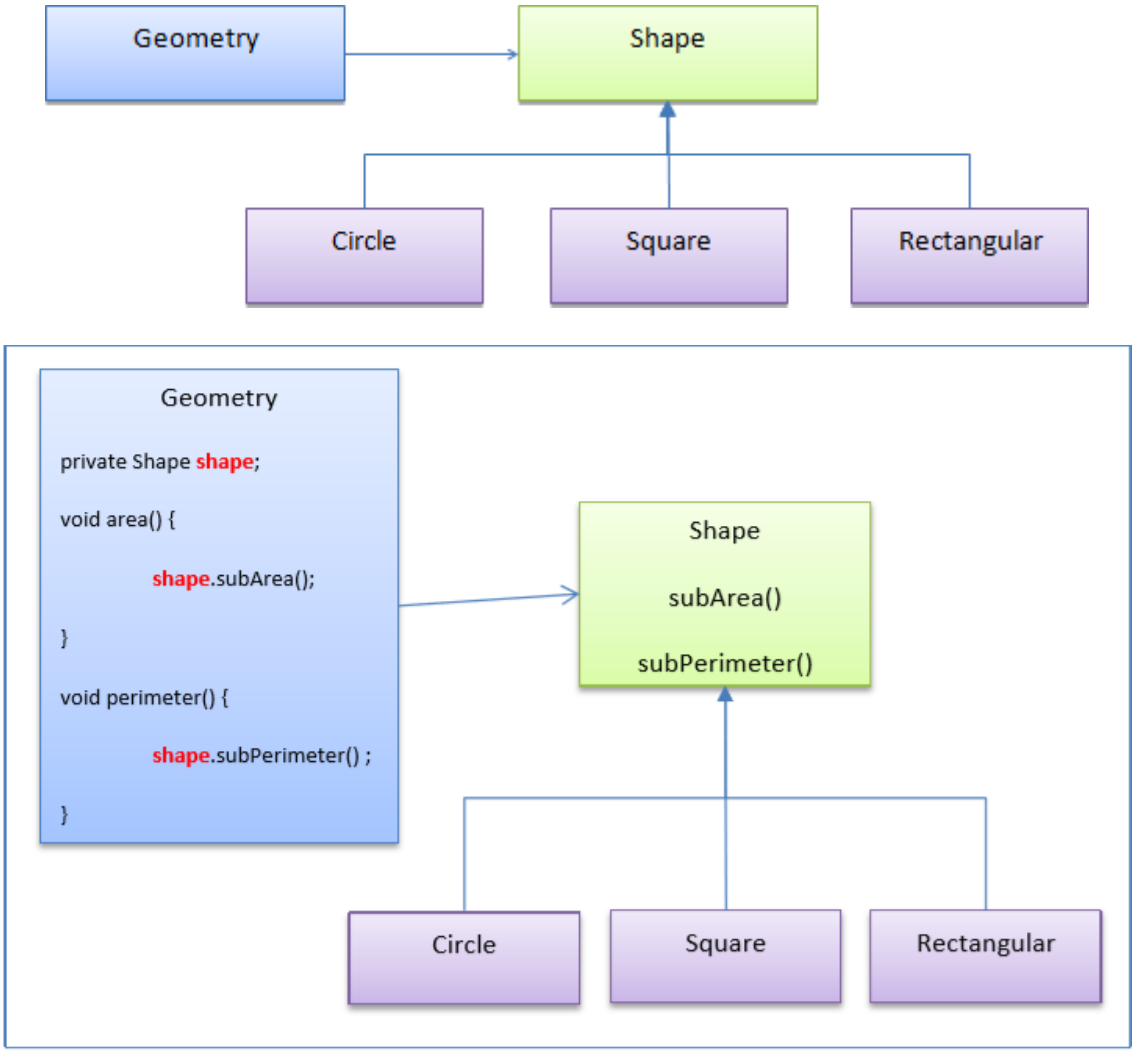
Soit une classe Geometry qui permet de calculer l'aire et le périmètre d'une figure géométrique comme dans le schéma suivant et pseudo-code ci-dessous :




```
1. class Geometry {
2.     void area(Shape shape) {
3.         if (shape instanceof Circle) {
4.             // calculate for circle
5.         }
6.         else if (shape instance of Square) {
7.             // calculate for square
8.         }
9.     }
10.
11.     void perimeter (Shape shape) {
12.         if (shape instanceof Circle) {
13.             // calculate for circle
14.         }
15.         else if (shape instanceof Square()) {
16.             // calculate for square
17.         }
18.     }
19. }
```

La classe Geometry gère les deux cas des formes Circle et Square. Si une nouvelle forme doit être ajoutée, il faut modifier Geometry en ajoutant un 3e bloc de test if sur la nouvelle forme, ce qui violerait le principe d'OCP. Or il est préconisé d'ajouter du code.

Pour respecter l'ouverture/fermeture, Geometry va s'appuyer sur une interface Shape avec une implémentation correspondant à chaque forme. Geometry délègue à l'interface Shape le traitement.



Puisque Geometry dépend de Shape, il devient alors possible d'ajouter une nouvelle forme Rectangular sans modifier Geometry en créant une classe implémentant Shape.

Le principe se retrouve dans quelques design patterns :

- Strategy : le code qui doit être ouvert/fermé travaille avec plusieurs algorithmes possibles sans impacter le code client ;
- Abstract factory : une classe qui sert de fabrique d'instances d'un certain type ;
- Template method : la structure générale d'une méthode est fermée, mais certaines sous-parties peuvent être ouvertes pour des implémentations spécifiques.

Quand appliquer

L'OCP est incontournable pour rendre le code flexible. L'erreur classique consisterait à ouvrir/fermer systématiquement toutes les classes de l'application. En effet, tout n'est pas sujet à la flexibilité et cela complexifierait le code. Ce qui rend néfaste d'autant plus que la flexibilité recherchée n'est pas entièrement exploitée. Il convient d'étudier les points d'ouverture/fermeture :

- quand des algorithmes divers interviennent et sont exprimés par le client ;
- en fonction des besoins de flexibilité pressentis par le développeur ;
- à mesure des changements répétés constatés au cours du développement.

Comment appliquer

Utiliser les design patterns cités plus haut pour faire une délégation abstraite.

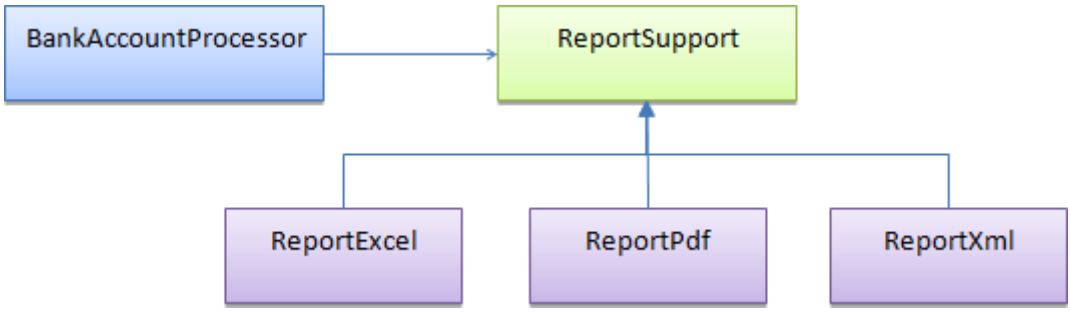
Exemple de cas concret

On peut imaginer un traitement de génération de rapport après extraction des données. Concrètement une application est amenée à produire de tels rapports pour aider les décideurs dans la prise de décision : rapport Excel, PDF, fichiers plats, XML.

L'algorithme peut être schématisé comme suit :

```
1. class BankAccountProcessor {
2.     public void extractAndGenerateReport(Form form) {
3.         Result res = search(form);
4.         if (form.isExcel()) {
5.             generateExcel(res);
6.         } else if (form.isPdf()) {
7.             generateExcel(res);
8.         } else if (form.isXml()) {
9.             generateXml(res) ;
10.         }
11.     }
12. }
```

Dans l'exemple ci-dessus, trois types de rapports sont implémentés. Ajouter un autre type impliquerait donc de modifier la méthode extractAndGenerateReport en ajoutant un autre bloc de test. Il est judicieux d'identifier que le type de rapport doit être une abstraction par exemple ReportSupport avec une méthode generate(Result res) à laquelle on déléguera la suite avec trois implémentations possibles Excel, PDF et XML.



Code de la classe de service de génération de rapport

```
1. class BankAccountProcessor {
2.     private ReportSupport delegate;
3.
4.     public void extractAndGenerateReport(Form form) {
5.         Result res = search(form);
6.         delegate = getExtractor(form);
7.         delegate.generate(res);
8.     }
9.
10.     private ReportSupport getExtractor(Form form) {
11.         // find the right implementation according to support choice
12.     }
13. }
```

Code de la classe abstraite et des trois implémentations

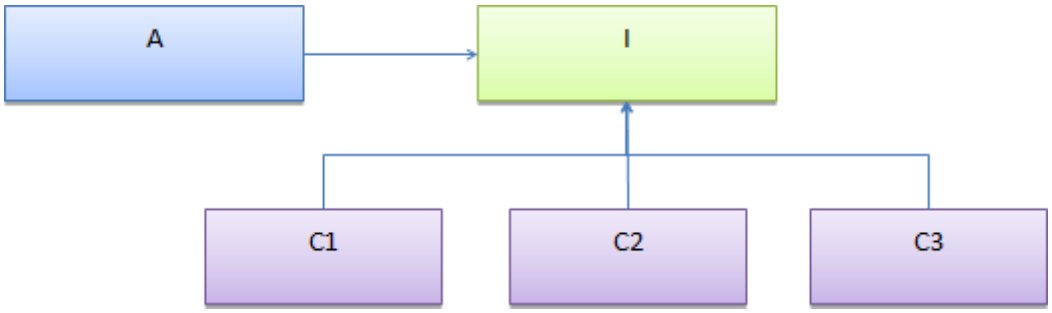
```
1. abstract class ReportSupport {
2.     public abstract void generate(Result res);
3. }
4.
5. class ReportExcel extends ReportSupport {
6.     public void generate(Result res) {
7.         // generate in Excel
8.     }
```

Code de la classe abstraite et des trois implémentations

```
9. }
10.
11. class ReportPdf extends ReportSupport {
12.     public void generate(Result res) {
13.         // generate in PDF
14.     }
15. }
16.
17. class ReportXml extends ReportSupport {
18.     public void generate(Result res) {
19.         // generate in XML
20.     }
21. }
```

V-C - Liskov substitution principle : principe de substitution de Liskov

L'OCP montre l'importance de l'héritage en séparant la partie commune et variable qui est extensible. De manière générale, l'abstraction avec l'utilisation des interfaces permet de découpler les classes en faisant reposer une classe sur une abstraction de classes.



La technique présentée ci-dessus joue un rôle primordial dans la modularité des applications et n'est efficace que si l'abstraction est parfaitement identifiable. L'interface I doit fournir une bonne abstraction des classes Ci et ces dernières doivent s'y conformer. C'est précisément le sens du principe de substitution de Liskov.

C'est Barbara Liskov qui a donné la définition du sous-typage :

> What is wanted here is something like the following substitution property: if for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.
> Barbara Liskov, Data Abstraction and Hierarchy, *SIGPLAN Notices*, 23,5 (May, 1988).

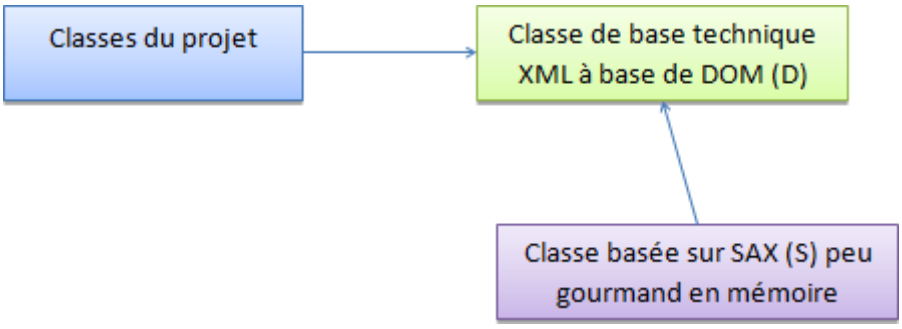
Autrement dit, un sous-type S doit être substituable à son type de base T dans toute l'application où T est utilisé sans causer de comportement non désiré dans le programme.

Exemple 1

Une équipe technique développe une bibliothèque de fichiers XML destinée aux projets Java de toute l'entreprise en exposant une classe D avec les opérations de lecture et écriture :

- read()
- write()

Les projets ont le choix de l'implémentation parmi DOM (D) et SAX (S) proposées par la bibliothèque comme sous-classes.



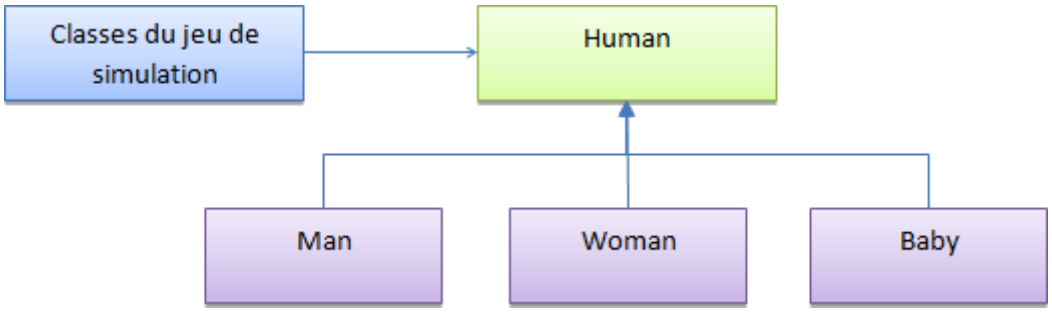
La classe s'appuyant sur SAX hérite de la classe de base pour réutiliser du code de journalisation des événements lors de la lecture et écriture, ce qui est tout à fait louable puisque la duplication de code est éliminée.

Une équipe choisit DOM pour la facilité de l'emploi des objets et poursuit le développement de son application jusqu'au jour où un test de charge provoque une erreur de mémoire de type OutOfMemoryError à la lecture d'un fichier XML volumineux. Une demande de support à l'équipe technique leur permet de résoudre leur problème en changeant DOM par SAX qui est peu consommateur en mémoire.

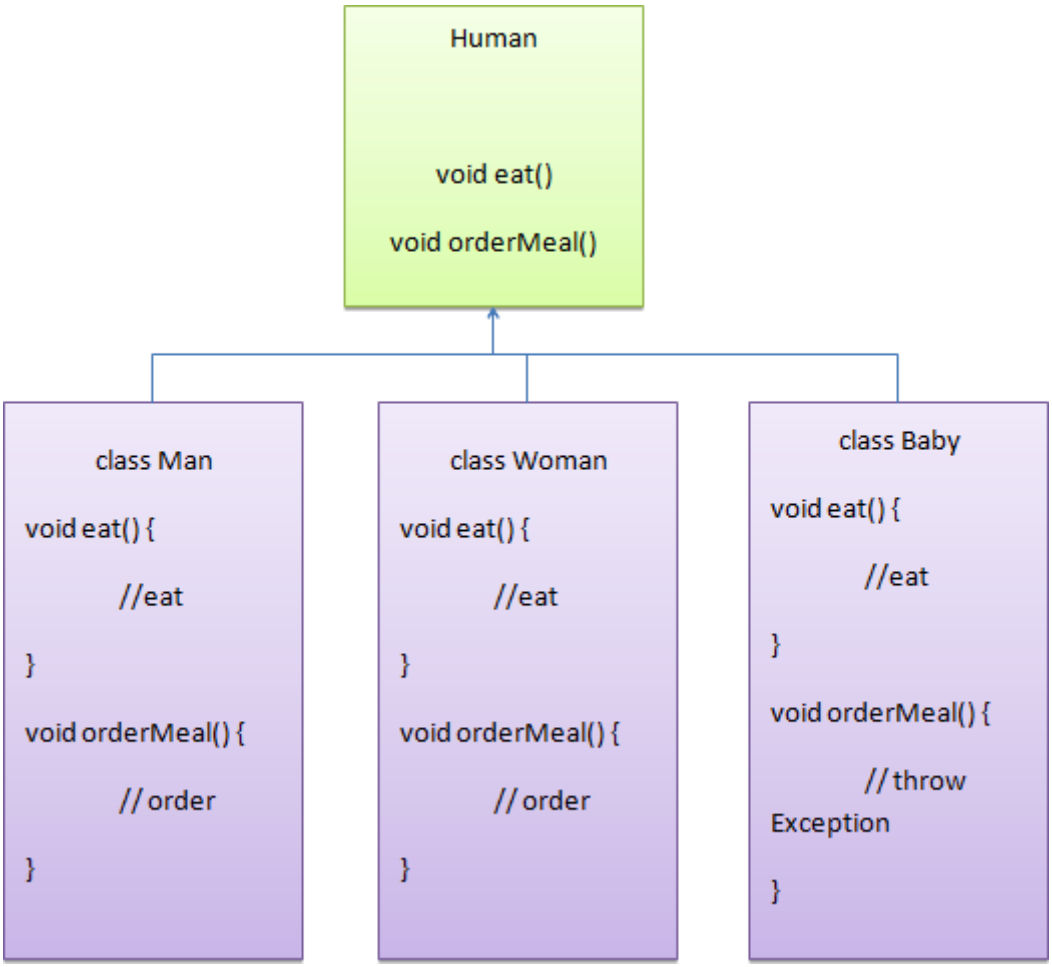
L'ennui à présent est que l'écriture repasse en mode DOM puisque SAX est une API de parsing uniquement. Le projet était dans l'impossibilité de générer des fichiers XML volumineux pour les envoyer à leurs différents partenaires. S n'est pas substituable à D.

Exemple 2

Un éditeur souhaite créer un jeu vidéo simulant l'activité humaine. Il sous-traite le développement de l'homme par une équipe tierce pour se focaliser sur l'infrastructure d'une ville et les activités urbaines. Une classe de base Human leur est fournie sous forme de bibliothèque avec plusieurs sous-classes comme Man, Woman, et Baby.



L'interface Human propose des méthodes comme eat() et orderMeal().



L'éditeur intègre dans la bibliothèque et l'utilise pour simuler l'activité humaine dans la ville.

Un test est réalisé en chargeant la partie dans un restaurant pour qu'un humain se nourrisse. Le test est concluant pour un adulte comme Man ou Woman, mais pour un Baby le test se solde par une erreur. En effet, un bébé n'est pas autonome pour être capable de commander un plat.

La méthode orderMeal() renvoyait une exception UnsupportedOperationException. Baby n'est pas substituable à Human dans ce cas d'utilisation.

Ces exemples montrent que pour que la substitution soit efficace, il est important que les opérations de toutes les sous-classes respectent bien un contrat établi.

Conception par contrat

La définition met en exergue l'importance du rôle de la classe de base qui se présente comme une offre de service fournie par chaque sous-classe. En langage objet, cela signifie que la classe de base A est une interface exposée que les implémentations doivent respecter.

Ce concept rejoint celui du « Design by contract » de Bertrand Meyer, l'interface représentant un véritable contrat passé entre chaque sous-classe et les classes susceptibles de l'utiliser.

En s'appuyant sur le mécanisme d'héritage et plus particulièrement le polymorphisme, le principe de substitution s'oppose à une pratique très répandue dans laquelle l'héritage permet de factoriser du code dans la classe de base pour être réutilisé par plusieurs sous-classes. Du point de vue langage, ceci est tout à fait « légal » puisque le

code compile. Il est cependant plus efficace d'utiliser la composition, en externalisant dans un module dédié à cette responsabilité (voir SRP).

Jusqu'où peut aller la substitution ?

Selon le principe, la substitution est parfaite tant que le contrat est respecté. À mesure que l'application évolue, tôt ou tard, une classe finira par ne pas respecter le contrat établi par l'interface et deux choix s'offrent alors au développeur :

- l'interface reste fermée et impose donc de recourir au downcast (utilisation de instanceof en Java avec cast) entraînant une violation de l'OCP ;
- l'interface est étendue pour couvrir ce cas particulier et impose donc aux autres classes qui l'implémentent une partie qui ne les concerne pas. Ce sont typiquement des méthodes sans corps ou lançant une exception de type UnsupportedOperationException. Ceci entraîne cette fois une violation du LSP.

La 1re solution est de loin préférable pour respecter l'intégrité du contrat. En d'autres termes, un downcast isolé pour implémenter un cas particulier est préférable plutôt que de corrompre l'interface et surtout masquer le problème. Dans le cas du jeu de simulation évoqué plus haut, un test avec instanceof Baby doit être fait pour éviter un plantage dans la partie de jeu au restaurant.

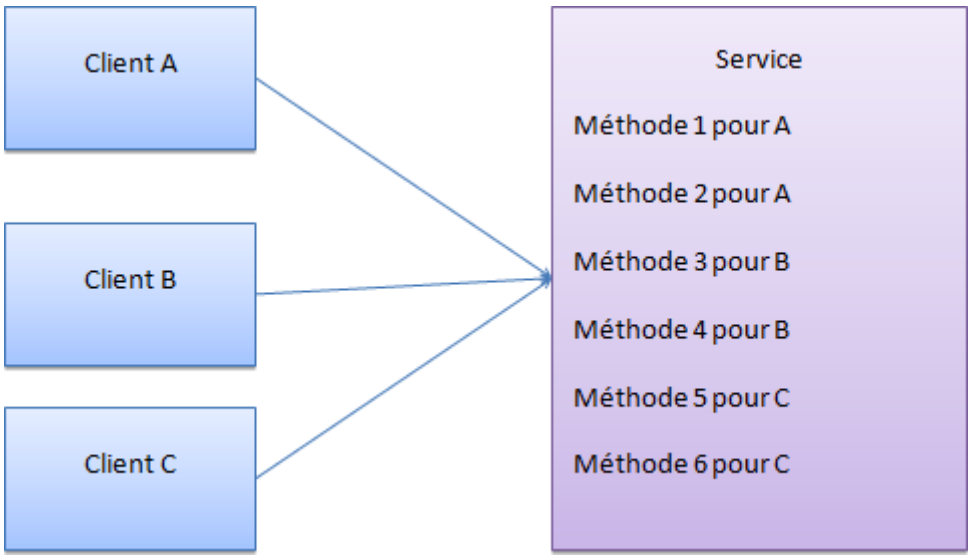
Sur des projets de grande envergure, il n'est pas rare de voir de nombreuses classes implémentant des méthodes héritées sans corps, ce qui rend le refactoring d'autant plus difficile, car le développeur n'a pas forcément connaissance de la méthode d'invocation sur ces méthodes (réflexion, composant distribué…).

V-D - Interface Segregation Principle : principe de ségrégation des interfaces

Le principe stipule que le client ne doit voir que les services dont il a besoin.

Autrement dit, la dépendance d'une classe vers une autre doit être restreinte à l'interface la plus petite possible.

Lorsque le principe de responsabilité unique est respecté avec un rôle bien défini et une forte cohésion, elle peut être utilisée par différents clients sans que ces derniers utilisent des fonctions communes.



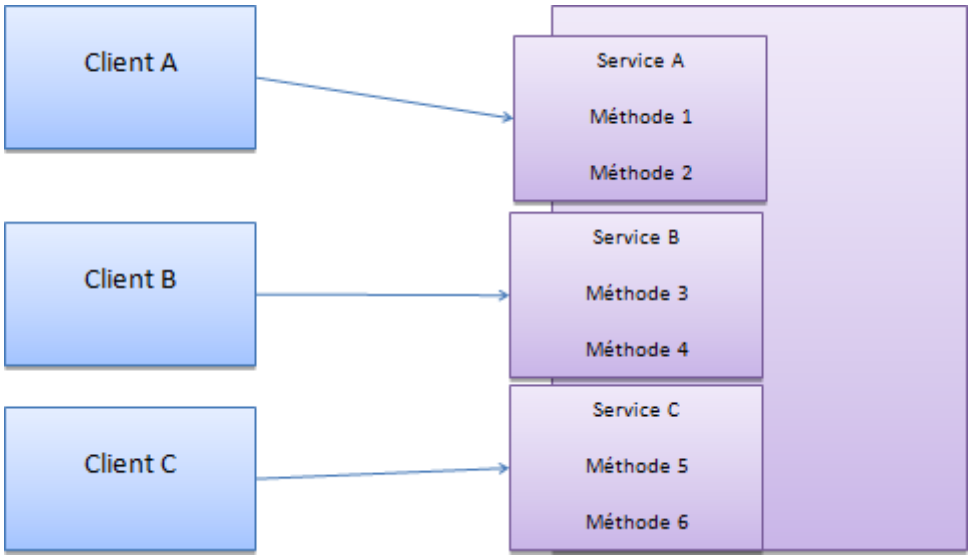
Quand appliquer

L'inconvénient est que tous les clients voient les opérations exposées par le service :

- le client n'utilise qu'une partie de l'interface qui l'intéresse ;

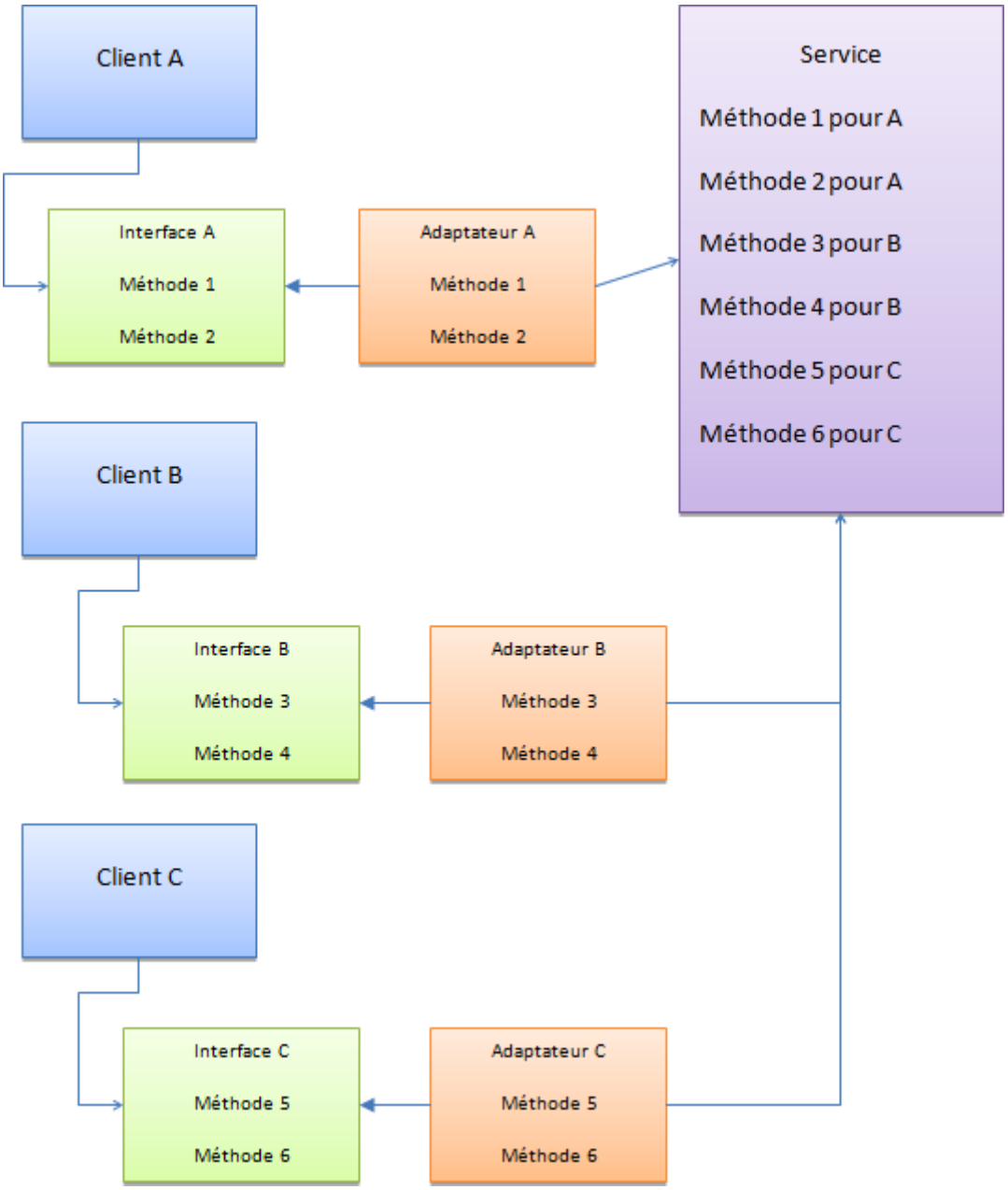
- chaque client peut être impacté par les changements d'une interface qu'il n'utilise pas.

Pour y remédier, il faut donc séparer l'interface en autant d'interfaces pour chaque client.



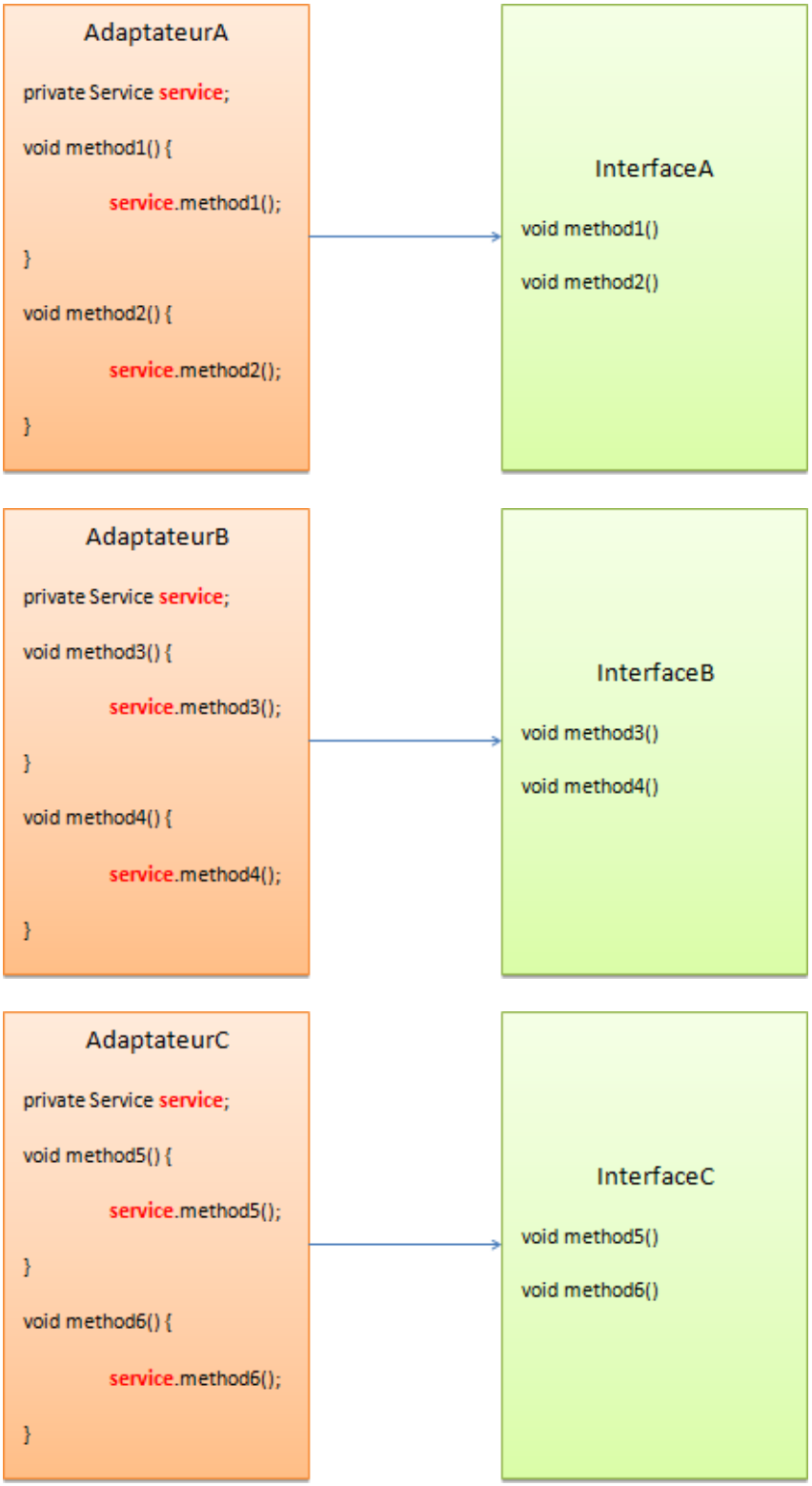
Comment appliquer

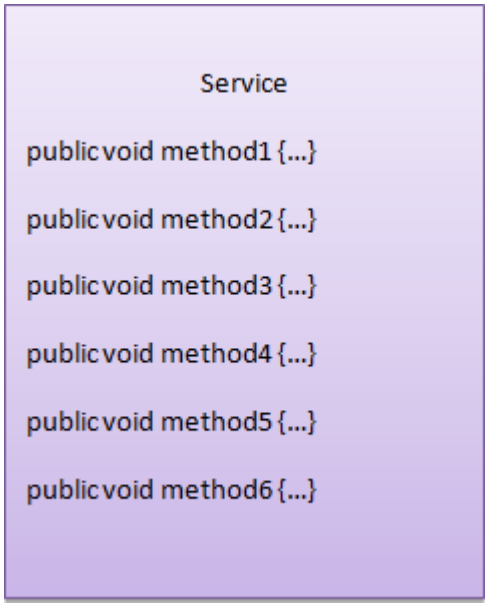
Utiliser le pattern adaptateur permet d'exposer uniquement les opérations d'un service pour le client et encapsule la classe concrète en déléguant l'appel.



Chacun des clients utilise l'interface dont il a vraiment besoin et l'implémentation associée utilise le même service sous-jacent pour effectuer le traitement demandé.

Code des interfaces et adaptateurs respectifs





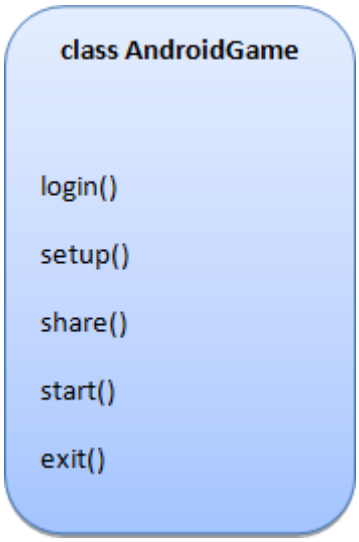
V-E - Dependency Inversion Principle : principe d'inversion des dépendances

Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstraction.

Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstraction.

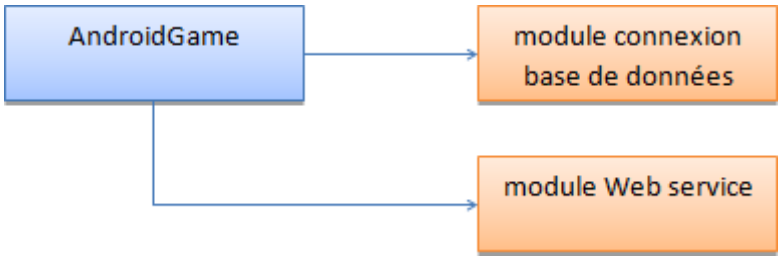
Problèmes des architectures monolithiques

Par exemple, une classe AndroidGame peut être créée avec les opérations suivantes :



Lorsque nous jouons à un jeu sur un smartphone Android (ceci est vrai également pour un jeu sur iPhone), celui-ci propose une identification pour sauvegarder la partie et partager les exploits de deux manières :

- avec un identifiant de l'éditeur du logiciel. L'utilisation d'un module de connexion aux bases de données est nécessaire pour effectuer les opérations transactionnelles ;
- avec un compte Facebook, avec un module de communication vers une application extérieure de type Web service.



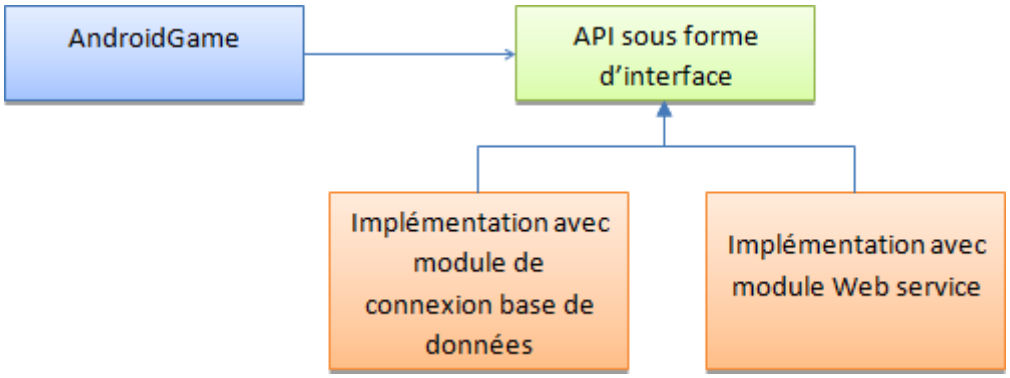
Il est souhaité que les modules métier soient les plus réutilisables possible. Or ces derniers sont construits sur d'autres modules de bas niveau et ceci pose deux problèmes :

- les modules de haut niveau sont impactés lorsqu'un module de bas niveau est modifié ;
- il n'est pas possible de réutiliser les modules de haut niveau indépendamment de ceux de bas niveau. En d'autres termes, il n'est pas possible de réutiliser la logique d'une application en dehors du contexte technique dans lequel elle a été développée.

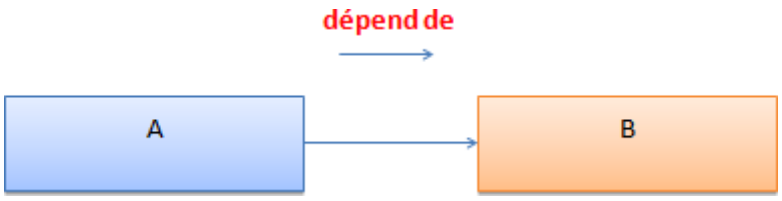
Comment appliquer

L'inversion des dépendances par emploi de l'abstraction.

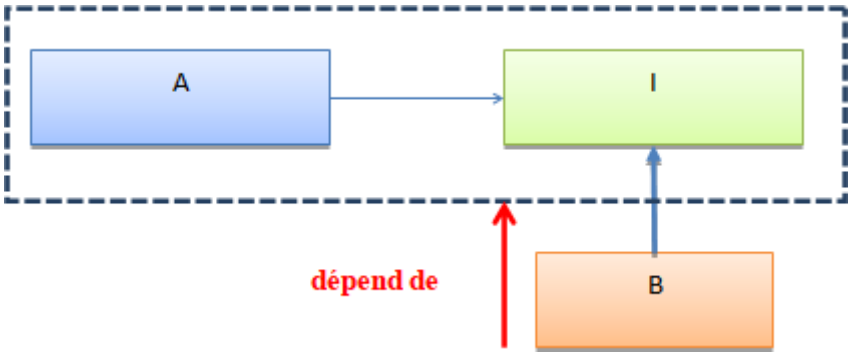
Selon le principe, la relation des dépendances doit être inversée : les modules de bas niveau doivent dépendre d'abstractions qui seront utilisées par les modules de haut niveau.



De manière générale, quand une situation se présente comme suit



l'inversion consiste à introduire une interface I dont A dépendra directement et dont B dépend également et qu'il doit implémenter.



Nous reconnaissons le mécanisme de délégation abstraite employé dans l'OCP. L'ouverture/fermeture est obtenue en inversant les dépendances de sorte que A ne soit plus impacté par les changements dans B.

Les classes A et B dépendent de I pour compiler. Cependant A a besoin de B au runtime, c'est-à-dire à l'exécution, pour fonctionner et remplir pleinement ses fonctions.

Par exemple, dans le monde JAVA/JEE, l'API Servlet d'un projet Web est nécessaire à la compilation pour créer des Servlets, mais au runtime l'implémentation de l'API Servlet est fournie par le serveur d'applications ou conteneur de Servlets sur lequel l'application est déployée.

Vers des frameworks métier

Ce principe conduit à des applications dont la logique métier est réutilisable quel que soit le contexte technique. En effet, la partie métier forme un « framework » qui permet de développer une même application dans plusieurs contextes techniques différents.

VI - Organisation de l'application en modules

VI-A - Principe de réutilisabilité de l'équivalence de livraison

> La granularité en termes de réutilisation est le package. Seuls des packages livrés sont susceptibles d'être réutilisés.
> Reuse-Release Equivalence Principle - REP

Cela signifie que pour réutiliser du code, il doit être livré complet dans une boîte noire. Les utilisateurs de ce code doivent être protégés des changements, car ils doivent être libres de décider quand intégrer le package dans leur code.

Cependant la réutilisabilité n'est efficace que si :

- le code reste la propriété de son auteur qui a la charge de le maintenir et faire évoluer ;
- le code est réutilisé tel quel avec l'API publique.

En effet il est très néfaste de vouloir intégrer le code tiers dans son propre code et de le patcher pour ses besoins avec des effets de bord que seul son auteur maîtrise.

L'ensemble des classes et interfaces de l'API doivent être livrées et versionnées. La granularité adéquate est le package, car c'est le niveau approprié pour livrer cet ensemble.

Par exemple migrer la bibliothèque commons-lang d'Apache, permettant de manipuler des classes de base Java, de la version 2.4 vers 2.6 peut se faire sans aucun risque.

VI-B - Principe de réutilisabilité commune

> Réutiliser une classe d'un package, c'est réutiliser le package entier.
> Common Reuse Principle - CRP

Il est rare d'utiliser une classe seule surtout si le SRP est respecté. Les packages doivent être constitués de classes susceptibles d'être réutilisées ensemble. Dans ce cas, il est plus efficace de les intégrer dans un même package, ce qui facilite l'utilisation de bibliothèques par l'utilisateur.

À l'inverse en termes de dépendances, utiliser une classe d'une bibliothèque revient à utiliser toute la bibliothèque. Il faut veiller à ne pas inclure deux classes totalement indépendantes dans un même package, car l'utilisateur est forcé de dépendre d'une autre classe B dont il n'a pas besoin alors qu'il utilise une classe A.

VI-C - Principe de fermeture commune

> Les classes impactées par les mêmes changements doivent être placées dans un même package.
> Common Closure Principle - CCP

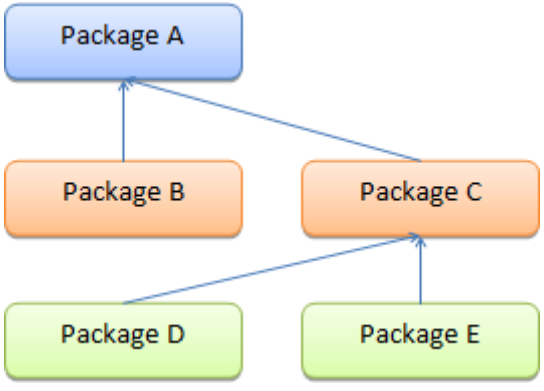
Le principe stipule qu'il faut regrouper dans un même package les classes impactées par un même changement.

VII - Gestion de la stabilité de l'application

VII-A - Principe des dépendances acycliques

> Les dépendances entre packages doivent former un graphe acyclique.
> Acyclic Dependencies Principle - ADP

L'objectif de la décomposition en packages est de limiter la propagation des impacts lorsqu'un package est modifié. Les changements intervenus sur une classe impactent immédiatement les autres classes du même package, mais n'impactent les autres packages qui en dépendent que si une nouvelle version du package est livrée.

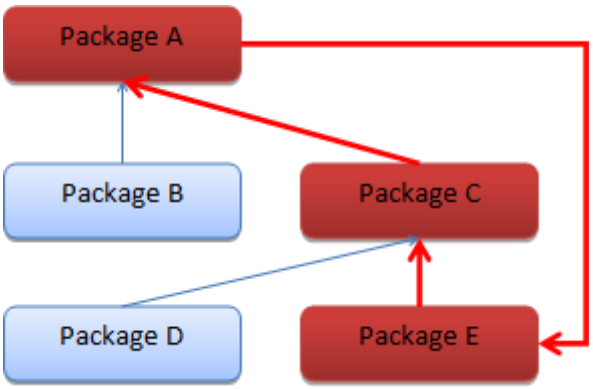


Les modifications d'une classe dans le package A ne sont propagées qu'aux packages B et C que lorsque ceux-ci décident d'utiliser une nouvelle version de A, et aux packages D et E que lorsque ceux-ci décident d'utiliser une nouvelle version de C.

La propagation des changements est guidée par les dépendances entre packages et forme un graphe orienté. L'organisation de ces dépendances forme un élément fondamental dans l'architecture de l'application.

Le principe des dépendances acycliques stipule que les dépendances entre packages doivent former un graphe acyclique orienté.

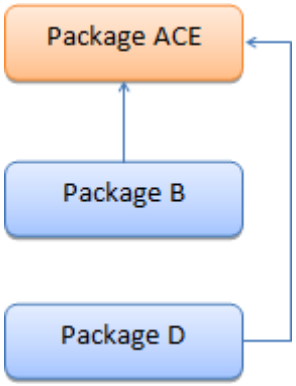
Que se passerait-il si le graphe devient cyclique ?



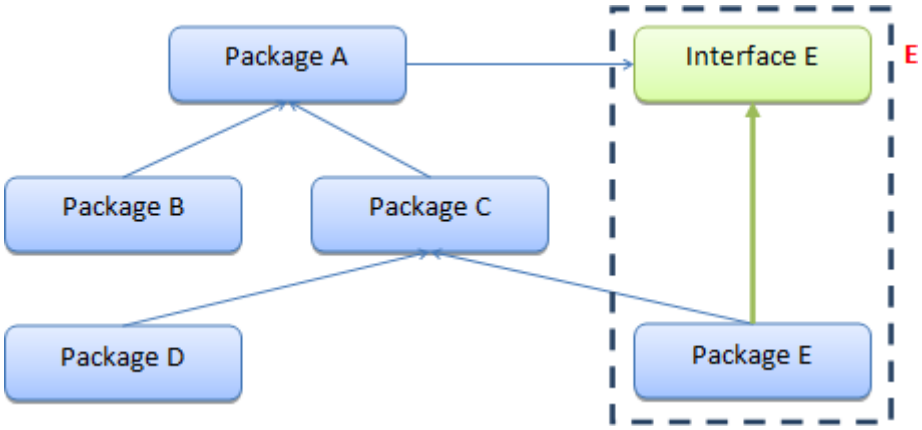
Cette fois le package A dépend du package E.

- A dépend de E pour compiler ;
- E dépend de C pour compiler ;
- C a besoin de A pour compiler.

Si A est modifié, alors il peut devenir impossible de recompiler ne serait-ce qu'un seul package, car C a besoin de l'ancienne version de A pour compiler. C et E doivent être recompilés, voire modifiés avant de compiler A. Les packages doivent donc évoluer ensemble. Si un graphe cyclique apparaît dans l'application, c'est peut-être le signe que l'ensemble des classes doivent être reconsidérées pour être réunies et livrées dans un même package.



Une autre solution consisterait à utiliser l'inversion des dépendances pour briser le cycle :



Le package A dépend de l'interface E puisqu'il dépendait initialement directement du package E. Ce dernier implémente l'interface E. L'inversion permet de changer le sens d'une dépendance et d'éliminer un cycle.

JDepend est un outil d'analyse des dépendances des packages Java. Le programme existe sous forme de plugin Eclipse pour faciliter l'intégration lors du développement de code.

VII-B - Principe de relation dépendance/stabilité

> Un package doit dépendre uniquement de packages plus stables que lui.
> Stable Dependencies Principle - SDP

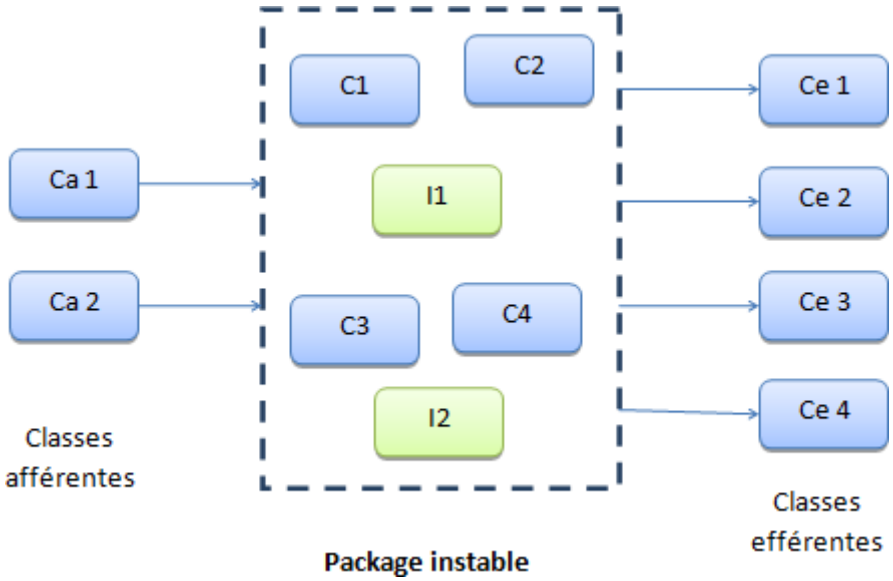
La stabilité ou instabilité est liée au couplage entre deux modules. Elle mesure le degré de fragilité du module si des changements s'opèrent dans les dépendances extérieures.

JDepend permet de calculer cette métrique également et peut être intégré dans le processus de développement pour améliorer le code.

L'instabilité correspond à un ratio entre les couplages efférents (Ce) et afférents (Ca) de telle sorte que $I = \frac{C_e}{C_e + C_a}$. Cette métrique est un indicateur de stabilité par rapport à la mise à jour d'autres packages.

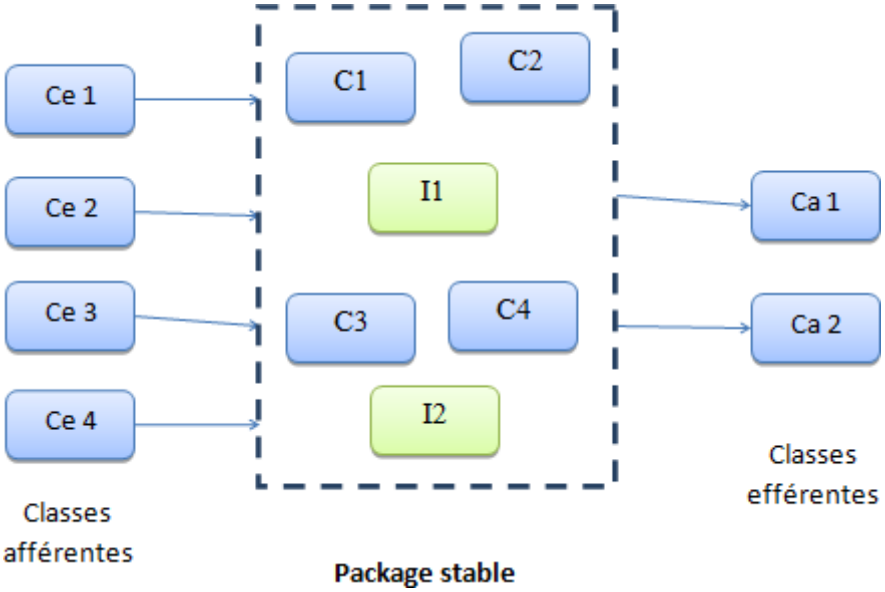
Plus formellement, pour un module donné :

- plus le nombre de modules dont il dépend est grand (couplage efférent), plus il est susceptible d'être impacté par des modifications d'un de ces modules, et donc moins il est stable.



Package instable

- plus le nombre de modules dépendant de ce module est grand (couplage afférent), plus les modifications de ce module sont coûteuses, et donc plus il est stable. Cela traduit la responsabilité du module dans l'ensemble du code.



Package stable

Selon cette définition, la stabilité du module est :

- maximale si le module n'utilise aucun autre module et se trouve lui-même utilisé par un grand nombre de modules (couplage efférent faible et couplage afférent fort) ;
- minimale si le module utilise de nombreux autres modules alors qu'il n'est utilisé lui-même par aucun autre module (couplage efférent fort et couplage afférent faible).

La mesure de l'instabilité est calculée comme suit :

$I = \frac{C_e}{C_e + C_a}$ avec I variant entre 0 et 1.

Dans les configurations suivantes :

- stabilité maximale : Ce = 1 et Ca = infini donc I tend vers 0 ;
- stabilité minimale : Ce = infini et Ca = 1 donc I tend vers 1.

Par exemple dépendre directement du JDK pour manipuler l'API Collection est maximal par rapport à utiliser une bibliothèque tierce. De base tout projet Java dépend du JDK pour compiler.

Le principe de relation dépendance/stabilité stipule qu'un module doit dépendre uniquement de modules plus stables que lui. En effet, l'impact sur les changements dans ces derniers serait amorti maximisant la stabilité globale de l'application.

VII-C - Principe de stabilité des abstractions

> Les packages les plus stables doivent être les plus abstraits.
> Les packages instables doivent être concrets.
> Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.
> Stable Abstractions Principle - SAP

Ce principe stipule que les interfaces et les classes qui l'implémentent doivent être dans des packages différents.

En effet, les interfaces ne contiennent que la signature des méthodes publiques (les méthodes par défaut existent depuis Java 8) avec éventuellement des constantes et sont mises à disposition pour d'autres équipes de développement. Si des bogues existent alors leur correction ne concerne que les implémentations et est ainsi isolée dans un package dédié.

VIII - Conclusion

La conception doit placer le contrôle des dépendances au cœur de son activité pour limiter les impacts des changements. Le coût des modifications engendrées serait alors réduit et les objectifs recherchés d'extensibilité, robustesse et réutilisabilité seraient atteints.

La réutilisabilité s'obtient :

- avec le SRP où chaque classe doit avoir un rôle bien défini et être la plus cohésive possible ;
- les classes qui sont utilisées ensemble doivent être réunies dans un même package lors de livraison pour faciliter la distribution (REP, CRP).

Plusieurs principes montrent le rôle majeur des interfaces en termes d'extensibilité et robustesse :

- elles servent de pare-feu arrêtant la propagation des changements d'un module sur les modules afférents (inversion de dépendance, abstraction et stabilité) ;
- l'héritage doit être davantage considéré comme une implémentation de l'interface plutôt qu'un moyen de factoriser du code. L'interface représente un contrat à respecter pour le code client et les classes qui l'implémentent rendant la substitution opérationnelle grâce au polymorphisme (OCP et principe de substitution de Liskov) ;
- avec le principe de ségrégation, une classe peut implémenter plusieurs interfaces répondant à plusieurs services et par conséquent le client n'utiliserait qu'une seule interface avec uniquement les méthodes nécessaires.

La robustesse s'obtient aussi en regroupant les classes fonctionnant ensemble :

- pour isoler au même endroit les changements induits par les classes efférentes (CCP) ; pour éviter les dépendances cycliques (ADP) ;
- en séparant les interfaces des classes concrètes dans deux packages différents pour (SDP et SAP) la stabilité de l'application est augmentée.

Ces principes constituent un cadre solide pour concevoir une application extensible et robuste.