CrossbowLabs

Principes avancés de conception objet

Régis Medina

03/2008

Table des matières

1. Introduction

En matière de développement logiciel, on constate aujourd'hui que la conception reste principalement une affaire de style personnel et d'expérience :

- Les principes de base de l'objet que sont l'encapsulation, l'héritage et le polymorphisme ne suffisent pas à guider la conception au quotidien.
- Les Design Patterns définissent des référentiels de plus haut niveau, mais ils ne forment pas un tout suffisamment cohérent pour guider clairement la construction de designs complets.

Il existe pourtant des principes extrêmement utiles en matière de conception. Ces principes ont été définis par des "hommes de l'art" tels que Bertrand Meyer et Robert Martin. Ils ont été présentés par Robert C. Martin dans une série d'articles parus dans C++ Report (accessibles dans la section "Publications / Articles" du site d'Object Mentor). Ce dossier présente à la fois une synthèse et une discussion de ces principes.

Ce dossier s'adresse à des lecteurs ayant déjà une bonne connaissance des principes de base de la programmation objet. On suppose en particulier que les notions d'encapsulation, d'héritage et de polymorphisme sont maîtrisées.

2. Les objectifs de la conception

2.1. Eviter les phénomènes de dégénérescence de l'application

Du point de vue de la conception, trois risques principaux pénalisent les activités de développement :

La rigidité : chaque évolution est susceptible d'impacter de nombreuses parties de l'application.

- Le développement est de plus en plus coûteux, ce qui introduit des risques au cours même du développement (ironiquement, c'est au moment où les échéances de livraison approchent et où la pression monte sur le projet que l'application devient la plus difficile à modifier).
- Le coût des modifications étant élevé, le logiciel a peu de chances d'évoluer après sa mise en production.

La fragilité : la modification d'une partie de l'application peut provoquer des erreurs dans une autre partie de l'application.

- Le logiciel est peu robuste, et le coût de maintenance reste élevé.
- Les modifications étant de plus en plus risquées, le logiciel a peu de chances d'évoluer après sa mise en production.

L'immobilité : il est difficile d'extraire une partie de l'application pour la réutiliser dans une autre application.

- Le coût de développement de chaque application reste élevé puisqu'il faut repartir de zéro à chaque fois.

Bien sûr, ces problèmes sont d'autant plus sensibles que l'application est volumineuse. Ils sont déjà perceptibles pour des applications de quelques milliers de lignes de code.

2.2. La clé du problème : la gestion des dépendances

Les problèmes ci-dessus trouvent le plus souvent leur source dans une multiplication des dépendances : tous les modules (classes, packages) de l'application finissent par dépendre de tous les autres modules, ce qui aboutit progressivement au fameux "spaghetti effect"[1]. Tous les principes qui suivent permettent de contrôler les dépendances des modules de l'application. Un contrôle strict des dépendances est en effet indispensable pour aboutir aux qualités recherchées de :

- **Robustesse** : les changements n'introduisent pas de régressions.
- **Extensibilité** : il est facile d'ajouter de nouvelles fonctionnalités.
- **Réutilisabilité** : il est possible de réutiliser certaines parties de l'application pour construire d'autres applications.

[1] Voir à ce propos l'article "Big Ball of Mud" de Brian Foote et Joseph Yoder

3. Index des principes

3.1. Gestion des évolutions et des dépendances entre classes

Principe d'ouverture/fermeture Open-Closed Principle (**OCP**)	Un module doit être ouvert aux extensions mais fermé aux modifications.
Principe de substitution de Liskov Liskov Substitution Principle (**LSP**)	Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir.
Principe d'inversion des dépendances Dependency Inversion Principle (**DIP**)	A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions." B. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.
Principe de séparation des interfaces Interface Segregation Principle (**ISP**)	"Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas."

3.2. Organisation de l'application en modules

Principe d'équivalence livraison/ réutilisation Reuse/Release Equivalence Principle (**REP**)	La granularité en termes de réutilisation est le package. Seuls des packages livrés sont susceptibles d'être réutilisés.
Principe de réutilisation commune Common Reuse Principle (**CRP**)	Réutiliser une classe d'un package, c'est réutiliser le package entier.
Principe de fermeture commune Common Closure Principle (**CCP**)	Les classes impactées par les mêmes changements doivent être placées dans un même package.

3.3. Gestion de la stabilité de l'application

Principe des dépendances acycliques Acyclic Dependencies Principle (ADP)	Les dépendances entre packages doivent former un graphe acyclique.
Principe de relation dépendance/stabilité Stable Dependencies Principle (SDP)	Un package doit dépendre uniquement de packages plus stables que lui.
Principe de stabilité des abstractions Stable Abstractions Principle (SAP)	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.

4. Open-Closed Principle

Tout module (package, classe, méthode) doit être ouvert aux extensions mais fermé aux modifications.

Open-Closed Principle - **OCP**

4.1. Evolution sans modification

Les problèmes de rigidité et de fragilité évoqués précédemment sont liés à une même cause : l'impact des changements sur de nombreuses parties de l'application. Chacun de ces changements oblige à modifier du code existant, ce qui est à la fois coûteux et risqué. Pour éviter cela, le principe d'ouverture/fermeture énoncé par Bertrand Meyer stipule que tout module (package, classe, méthode) doit être à la fois :

- **ouvert aux extensions** : le module peut être étendu pour proposer des comportements qui n'étaient pas prévus lors de sa création.
- **fermé aux modifications** : les extensions sont introduites sans modifier le code du module.

En d'autres termes, l'ajout de fonctionnalités doit se faire en ajoutant du code et non en éditant du code existant.

4.2. L'abstraction comme moyen d'ouverture/fermeture

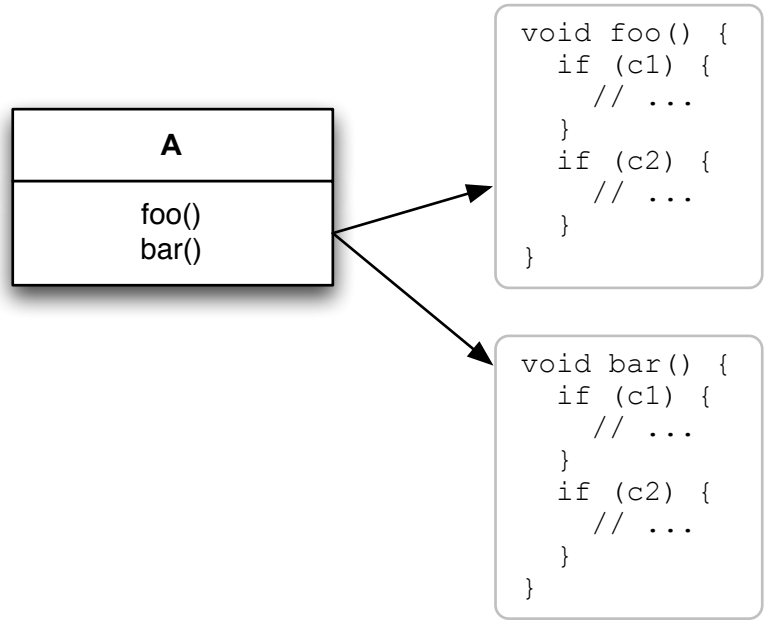
L'ouverture/fermeture se pratique en faisant reposer le code "fixe" sur une abstraction du code amené à évoluer. En d'autres termes, l'OCP consiste à séparer le commun du variable, en faisant reposer le commun sur une définition stable du variable.

Deux mécanismes principaux permettent de mettre en place l'abstraction préconisée par l'OCP :

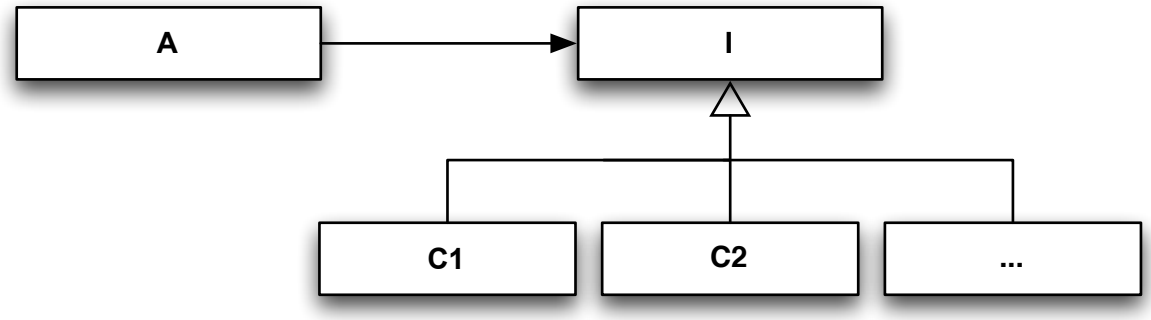
- En programmation objet, l'abstraction repose sur l'utilisation de classes d'interface (classes abstraites en C++, interfaces en Java) ,
- En programmation générique, l'abstraction repose sur l'utilisation de templates.

4.2.1. Utilisation de la "délégation abstraite"

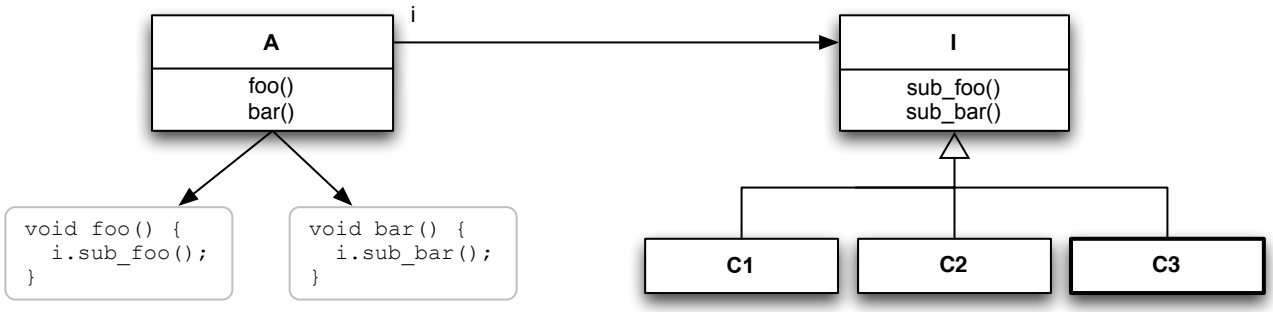
Dans le schéma suivant, A gère les cas c1 et c2. Si un nouveau cas c3 doit être géré, il faut modifier le code de A en conséquence :



Le code de A peut être ouvert/fermé aux modifications en introduisant une classe d'interface I dont dérivent des classes C1 et C2 correspondant aux cas c1 et c2 :



Puisque A repose uniquement sur l'interface I, il devient possible d'ajouter un nouveau cas c3 sous la forme d'une classe C3 dérivée de I, sans avoir à modifier A :



Ce mécanisme d'abstraction consiste à extraire la partie variable du code que l'on souhaite ouvrir/fermer : on introduit une sorte de "plug" dans le code pour lui permettre d'accueillir les évolutions futures.

4.2.2. Utilisation des templates

Les templates permettent de définir des abstractions de type, comme dans l'exemple suivant ou il est possible d'utiliser la méthode foo avec n'importe quelle nouvelle classe du moment que celle-ci possède une méthode `bar` :

```
template <class T>
void foo(T& t)
{
  ...
  t.bar();
  ...
}
```

Note: Ceci n'est cependant qu'un exemple de mise en oeuvre de l'OCP en programmation générique.

On retrouve ainsi de nombreux exemples d'ouverture/fermeture dans la Standard Template Library du C++, que l'on peut même considérer par nature comme un cas d'ouverture/fermeture :

- les conteneurs sont ouverts/fermés au traitement de nouveaux types,
- les algorithmes sont ouverts/fermés au traitement de nouveaux ensembles.

4.3. L'OCP SE RETROUVE DANS DE NOMBREUX DESIGN PATTERNS

De nombreux Design Patterns peuvent être vus comme des cas de mise en pratique de l'OCP. Par exemple :

- Strategy : le code initial est ouvert/fermé à l'ajout de nouveaux algorithmes en ayant recours à une classe d'interface commune à tous les algorithmes. Cela permet d'ajouter de nouveaux algorithmes sans changer le code client.
- Abstract factory : le code client est ouvert/fermé à la création de nouveaux objets.

- Template method : on ferme la structure générale d'une méthode, en ouvrant certaines sous-parties à de nouvelles implémentations. Cela permet de changer les sous-parties sans modifier la structure de base.
- Visitor : la structure parcourue est ouverte/fermée à l'ajout de nouveaux algorithmes de traitement.

4.4. L'application de l'OCP est un choix stratégique

L'OCP est un principe incontournable lorsque l'on parle de flexibilité du code. Par contre, une erreur classique consisterait à vouloir ouvrir/fermer toutes les classes de l'application en vue d'éventuels changements. Cela constitue une erreur dans la mesure où la mise en oeuvre de l'OCP impose une certaine complexité qui devient néfaste si la flexibilité recherchée n'est pas réellement exploitée.

Il convient donc d'identifier correctement les points d'ouverture/fermeture de l'application, en s'inspirant :

- des besoins d'évolutivité exprimés par le client,
- des besoins de flexibilité pressentis par les développeurs,
- des changements répétés constatés au cours du développement.

La mise en oeuvre de ce principe reste donc une affaire de bon sens, sachant que la meilleure heuristique reste la suivante : **on n'applique l'OCP que lorsque cela simplifie la conception**.

4.5. L'OCP s'applique également dans une démarche itérative

En préconisant d'anticiper le changement, ce principe semble s'opposer à certains principes d'Extreme Programming ("*You ain't gonna need it*" et "*Do the simplest thing that could possibly work*"). La contradiction n'est cependant qu'apparente :

- L'OCP a pour but de réduire le coût du changement dans le logiciel : il facilite donc en cela l'approche itérative/incrémentale.
- L'ouverture/fermeture du code n'est pas obligatoirement faite dans un design "up-front" : elle peut (doit) au contraire n'être mise en place qu'au cours de l'activité permanente de refactoring, lorsque les cas concrets d'extension se présentent.

Article original : http://www.objectmentor.com/publications/ocp.pdf

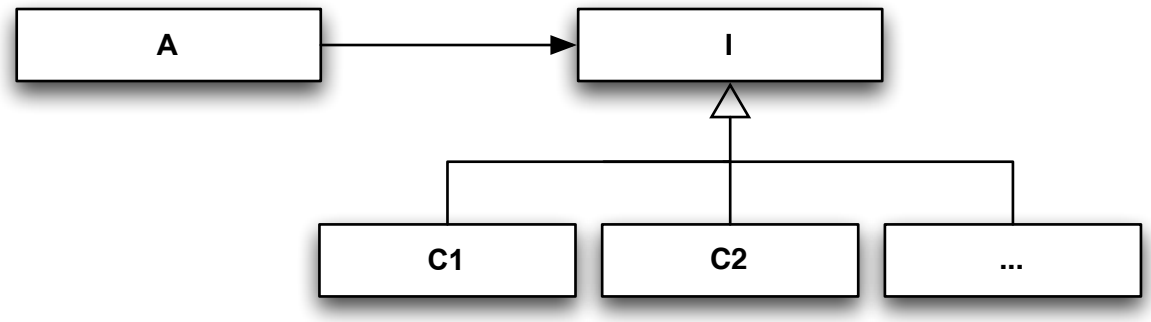
5. Principe de Substitution de Liskov

**Les méthodes qui utilisent des objets d'une classe
doivent pouvoir utiliser des objets dérivés de cette classe
sans même le savoir.**

Liskov Substitution Principle - **LSP**

5.1. L'OCP REPOSE SUR LA LÉGITIMITÉ DES ABSTRACTIONS

La présentation du principe d'ouverture/fermeture a permis de montrer le rôle de l'héritage dans la séparation du commun et du variable. Plus généralement, l'introduction de classes d'interface permet de briser des dépendances directes entre classes en faisant reposer une classe sur une abstraction d'autres classes :



Cette technique, qui joue un rôle primordial dans la modularité des applications objet, n'est cependant efficace que dans la mesure où une abstraction est réellement identifiable. En d'autres termes la classe d'interface I doit fournir une bonne abstraction des classes Ci, et à l'inverse les classes Ci doivent se conformer correctement à l'interface I. C'est précisément le sens du principe de substitution de Liskov.

5.2. Le principe de substitution de Liskov

Barbara Liskov donne la définition suivante 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 (1988).

En clair, pour prétendre à l'héritage, une sous-classe doit être conçue de sorte que ses instances puissent se substituer à des instances de la classe de base partout où cette classe de base est utilisée.

5.3. L'héritage comme offre de service

Cette définition de l'héritage place l'accent sur la classe de base, qui se présente comme une veritable offre de service affichée par chaque sous-classe. En langage objet, cela revient à dire que **la classe de base est une interface exportée par toutes ses sous-classes**.

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 insistant sur cette approche de l'héritage, le principe de substitution s'oppose à une pratique répandue dans laquelle l'héritage est mis en oeuvre pour factoriser du code entre plusieurs classes. Cette pratique, bien que parfaitement "légale" du point de vue du langage, représente une utilisation parfois abusive du mécanisme d'héritage à laquelle les "puristes" préfèrent l'héritage privé ou la composition.

5.4. La substitution parfaite ?

Selon ce principe, l'héritage est donc d'autant plus efficace que la substitution est parfaite. Comme toujours cela est plus facile à dire qu'à faire : tôt ou tard apparaît une sous-classe qui ne "rentre pas dans l'interface", et là deux solutions s'offrent au développeur :

- soit il reste ferme sur l'interface et impose aux utilisateurs de la sous-classe de recourir au "downcast" pour traiter le cas particulier, ce qui entraîne une violation de l'OCP,
- soit il élargit l'interface pour couvrir ce cas particulier, mais il impose alors aux autres sous-classes une partie d'interface qui ne leur correspond pas (typiquement sous la forme de méthodes définies dans l'interface mais dont l'implémentation restera vide dans les classes dérivées), avec cette fois-ci une violation du LSP.

Ce problème n'a pas de solution simple, mais la première solution nous semble préférable dans la mesure où l'intégrité du "contrat" est respectée. En d'autres termes, dans les cas où l'interface ne convient pas on préfèrera recourir au downcast et laisser les problèmes où ils apparaissent plutôt que de tenter de masquer le problème en corrompant l'interface.

5.5. Le LSP ne se réduit pas à l'héritage

Pour reprendre le vocabulaire employé par Jim Coplien dans son livre "Multi-Paradigm Design with C++", l'interface exprime une commonalité, et chaque sous-classe représente une variabilité particulière liée à cette commonalité. "Cope" distingue les variabilités positives, c'est-à-dire celles qui respectent la commonalité, des variabilités négatives, c'est-à-dire celles qui la rompent.

L'argument central de Coplien dans ce livre est que le traitement des commonalités et des variabilités forme le fondement du design, et que son implémentation ne se réduit pas à la mise en oeuvre du paradigme objet. Le principe de substitution s'applique également selon lui aux cas de spécialisation de templates ou de surcharge de fonctions.

_____
Article original : http://www.objectmentor.com/publications/lsp.pdf

6. Principe d'inversion des dépendances

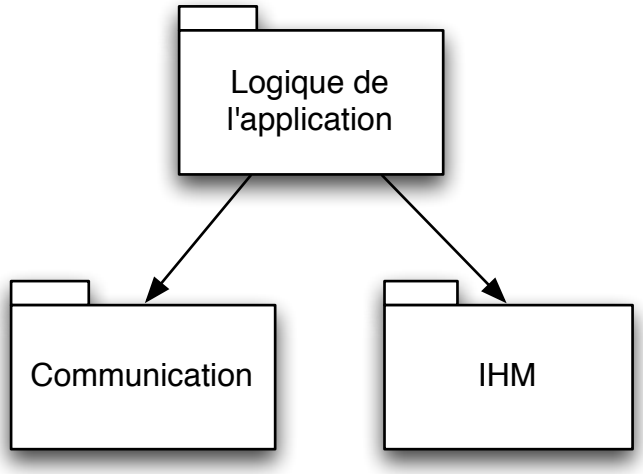
A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions.

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

Dependency Inversion Principle - **DIP**

6.1. Les architectures classiques ne permettent pas la réutilisation des modules "métier"

Dans la plupart des applications, les modules de haut niveau (ceux qui portent la logique fonctionnelle de l'application ou les aspects "métier") sont construits directement sur les modules de bas niveau (par exemple les librairies graphiques ou de communication) :

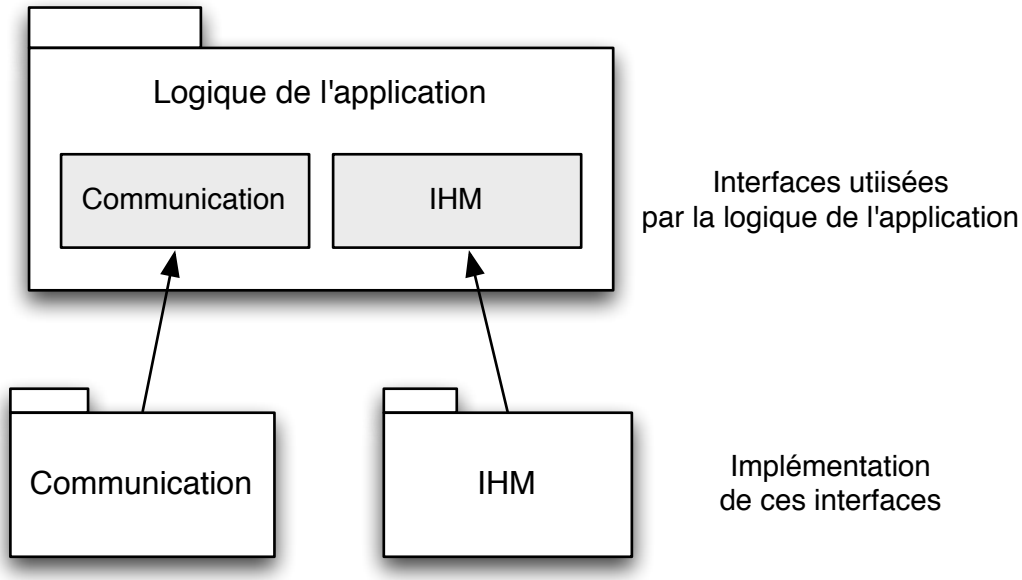


Cela paraît naturel au premier abord mais pose en réalité deux problèmes essentiels :

- Les modules de haut niveau doivent être modifiés lorsque les modules de bas niveau sont modifiés.
- 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.

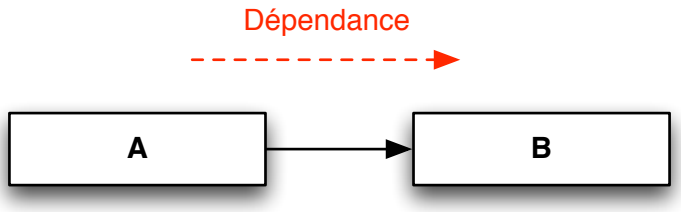
6.2. L'INVERSION DES DÉPENDANCES

Selon ce principe, la relation de dépendance doit être inversée : **les modules de bas niveau doivent se conformer à des interfaces définies et utilisées par les modules de haut niveau**.

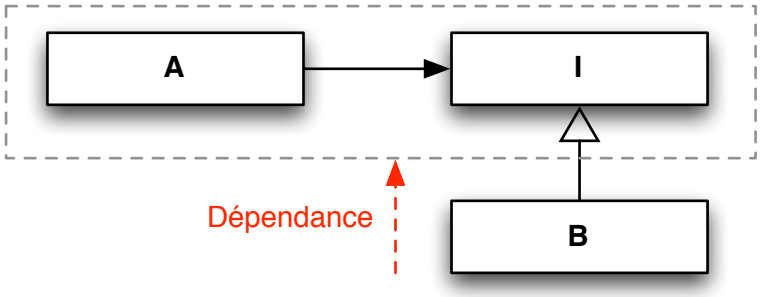


6.3. L'ABSTRACTION COMME TECHNIQUE D'INVERSION DES DÉPENDANCES

Considérons deux classes A et B, A utilisant B comme indiqué sur le schéma ci-dessous :



Pour inverser la dépendance de A vers B, on introduit une classe d'interface I dont derive B comme suit :



Dans ce nouveau schéma B dépend désormais du couple (A,I), la dépendance est donc inversée. Le couple (A,I) forme une sorte d'extension de la classe A dans laquelle on inclut une représentation abstraite des dépendances de A.

Remarque : On reconnaît ici le mécanisme de "délégation abstraite" présenté dans le Principe d'Ouverture/Fermeture. L'ouverture/fermeture est en effet obtenue en inversant la dépendance entre A et B, de sorte que A n'est plus impactée par des changements de B.

6.4. Vers des frameworks métier ?

Ce principe conduit à des applications dans lesquelles la logique "métier" est parfaitement réutilisable. Cette partie métier forme une sorte de "framework", qui permet de développer une même application dans des contextes techniques différents.

Article original : http://www.objectmentor.com/publications/dip.pdf

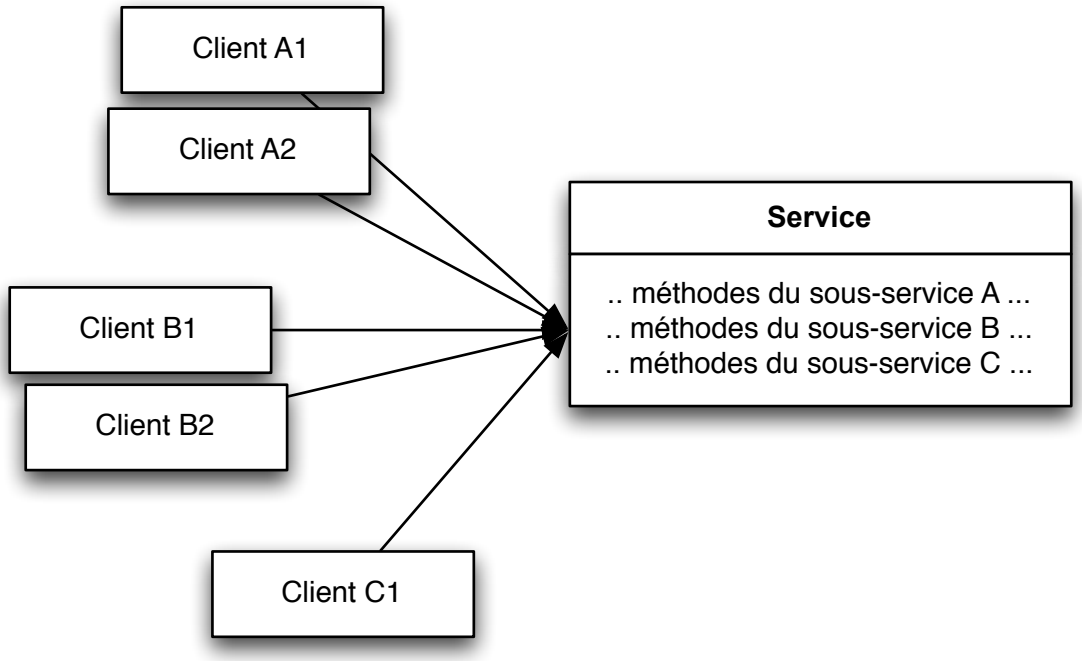
7. Principe de ségrégation des interfaces

Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.

Interface Segregation Principle - **ISP**

7.1. Pollution d'interface par aggrégation de services

On retrouve dans la plupart des designs quelques classes qui rendent plusieurs services simultanément, comme l'illustre le schéma ci-dessous :

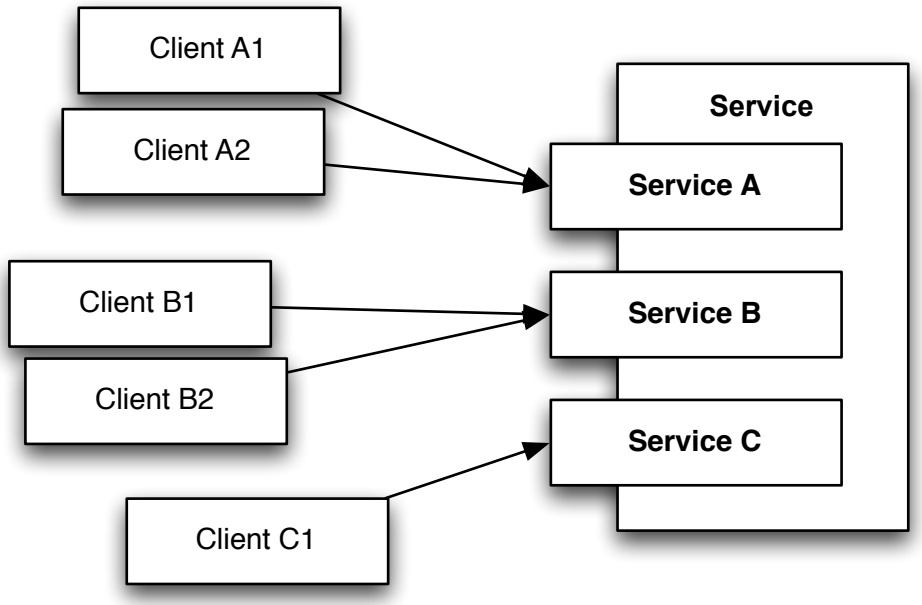


L'inconvénient de ce genre de cas est que tous les clients ont une visibilité sur tous les services rendus par la classe :

- Chaque client voit une interface trop riche dont une partie ne l'intéresse pas,
- Chaque client peut être impacté par des changements d'une interface qu'il n'utilise pas.

7.2. Solution : séparation des services de l'interface

Le principe de séparation des interfaces stipule que chaque client ne doit "voir" que les services qu'il utilise réellement :



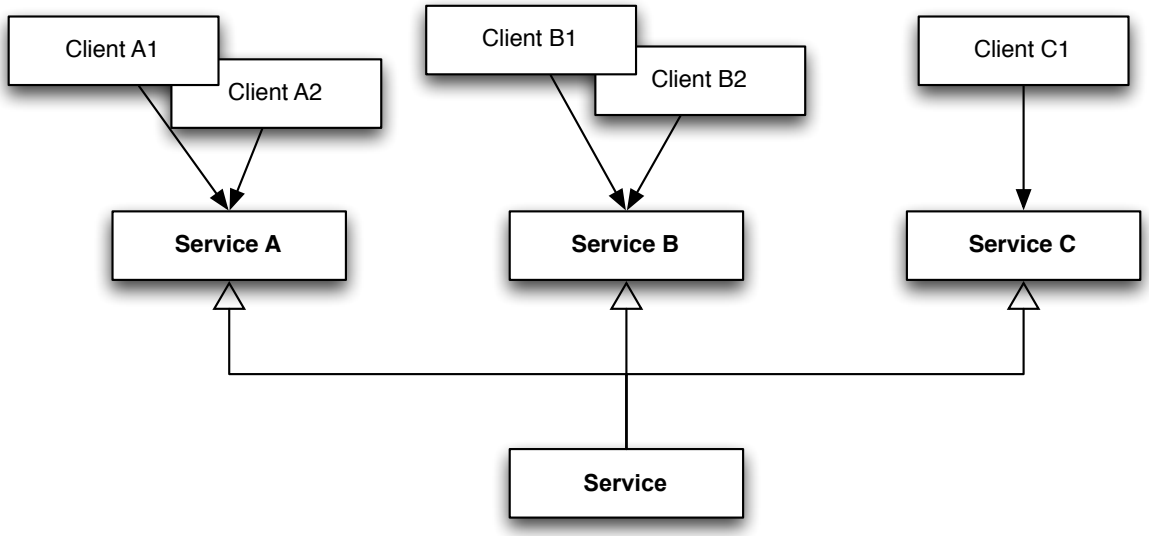
7.3. Techniques de séparation

Il existe deux techniques principales de mise en pratique de l'ISP :

- L'héritage multiple,
  Le Design Pattern "Adapter".

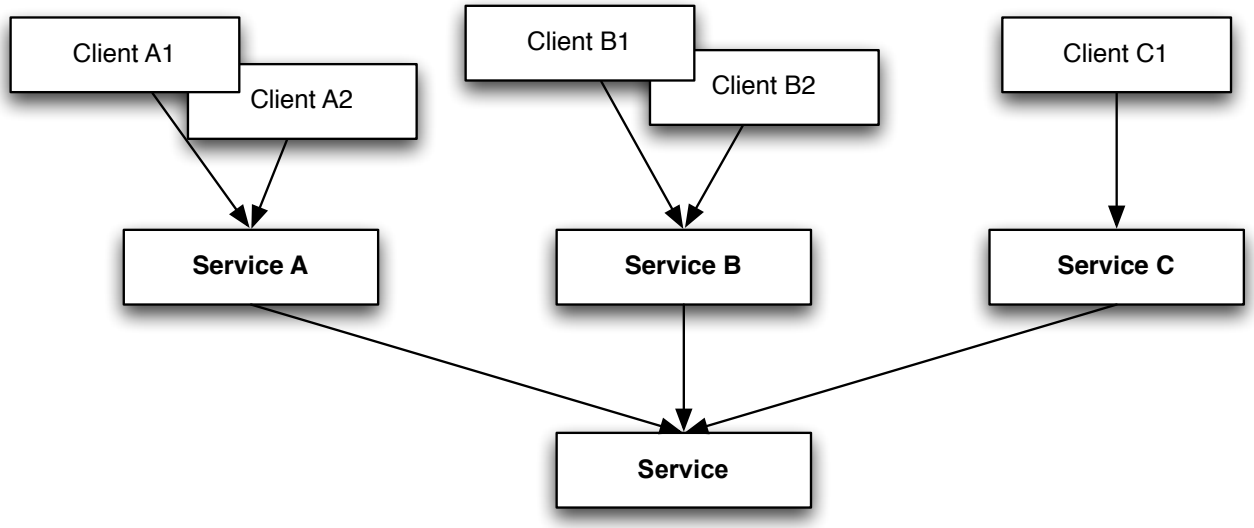
7.3.1. Séparation par héritage multiple

Dans cette approche chaque service est représenté par une classe d'interface dont dérive la classe qui implémente les services. Les clients ne voient les services qu'au travers de ces classes d'interface comme le montre le schéma suivant :



7.3.2. Séparation par Adapter

Lorsque l'héritage multiple n'est pas possible, les services peuvent être représentés par des classes d'adaptation :



Article original : http://www.objectmentor.com/publications/isp.pdf

8. Principe d'équivalence livraison/réutilisation

La granularité en termes de réutilisation est le package.
Seuls des packages livrés sont susceptibles d'être réutilisés.

Reuse-Release Equivalence Principle - **REP**

Le principe d'équivalence livraison/réutilisation stipule que la réutilisation n'est efficace que dans les conditions suivantes :

- **Le code réutilisé reste la propriété de son auteur**, qui garde la charge de le corriger et de le faire évoluer.
- **Le code est réutilisé tel quel**. Celui qui le réutilise doit se contenter de passer par des interfaces minimales, sans avoir à comprendre le fonctionnement interne du code utilisé.

Le code livré/réutilisé doit donc être géré comme un véritable produit, identifié par un système de gestion de versions. Ce mécanisme est trop lourd pour être appliqué au niveau de la classe, il s'applique à l'échelle du package : **la granularité en termes de réutilisation est donc le package**.

Article original : http://www.objectmentor.com/publications/granularity.pdf

9. Principe de réutilisation commune

Réutiliser une classe d'un package, c'est réutiliser le package entier.

Common Reuse Principle - **CRP**

Les packages doivent être constitués de classes susceptibles d'être réutilisées ensemble. En effet

- Si plusieurs classes doivent être réutilisées ensemble, il est plus simple de les intégrer au même package. L'utilisateur n'aura en effet à tirer qu'une seule librairie.
- A l'inverse, en termes de dépendances, réutiliser une classe d'une librairie force à dépendre de toute la librairie. Si l'on place deux classes totalement indépendantes dans un même package, on oblige les utilisateurs d'une classe à dépendre de l'autre classe alors que c'est inutile.

Article original : http://www.objectmentor.com/publications/granularity.pdf

10.Principe de fermeture commune

Les classes impactées par les mêmes changements doivent être placées dans un même package.

Common Closure Principle - **CCP**

Nous avons vu dans la présentation du Principe d'Ouverture/Fermeture que la fermeture complète de l'application était impossible, et que le surcoût de sa mise en oeuvre faisait du choix des endroits à ouvrir/fermer un choix stratégique. Ceci implique qu'il reste obligatoirement dans l'application des endroits impactés par les changements.

Pour réduire l'impact de ces changements et donc réduire les coûts d'évolution et de maintenance, le Principe de Fermeture Commune stipule simplement qu'**il faut regrouper dans un même package l'ensemble des classes impactées par un même changement**.

_____

Article original : http://www.objectmentor.com/publications/granularity.pdf

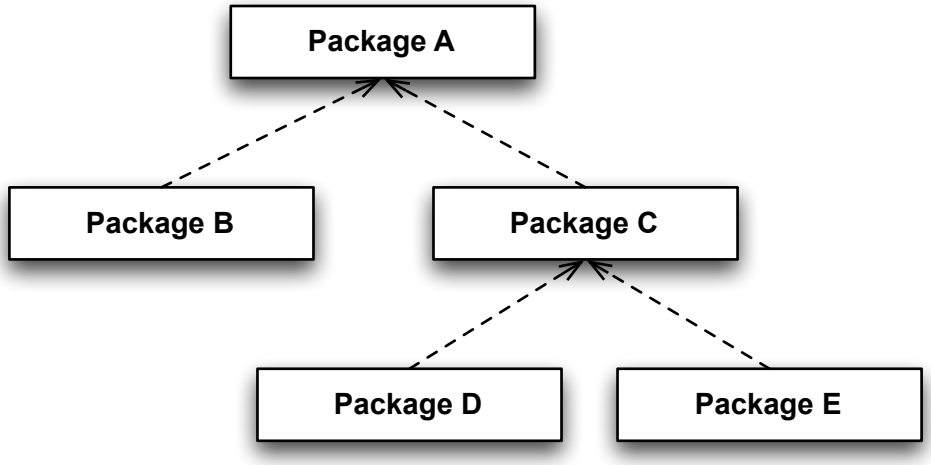
11.Principe des dépendances acycliques

Les dépendances entre packages doivent former un graphe acyclique.

Acyclic Dependencies Principle - **ADP**

11.1.Objectif de la décomposition en packages

Le but de la décomposition en packages est de temporiser la propagation des changements dans l'application : les changements d'une classe impactent immédiatement les autres classes du package, mais elles n'impactent les packages qui en dépendent que lorsqu'une nouvelle version du package est livrée :



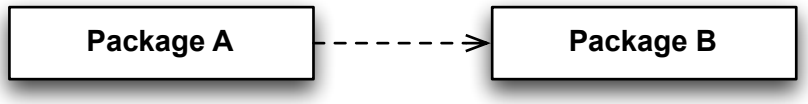
Les modifications d'une classe de A n'est propagée aux packages B et C que lorsque ceux-ci décident d'utiliser une nouvelle version de A, et aux packages D et E que lorsque ceux-ci décident d'utiliser une nouvelle version de C.

Les packages introduisent donc des points de synchronisation des changements dans l'application. La propagation des changements étant guidée par les dépendances entre packages, l'organisation de ces dépendances forme un élément fondamental de l'architecture de l'application.

Effet d'un cycle dans le graphe de dépendances

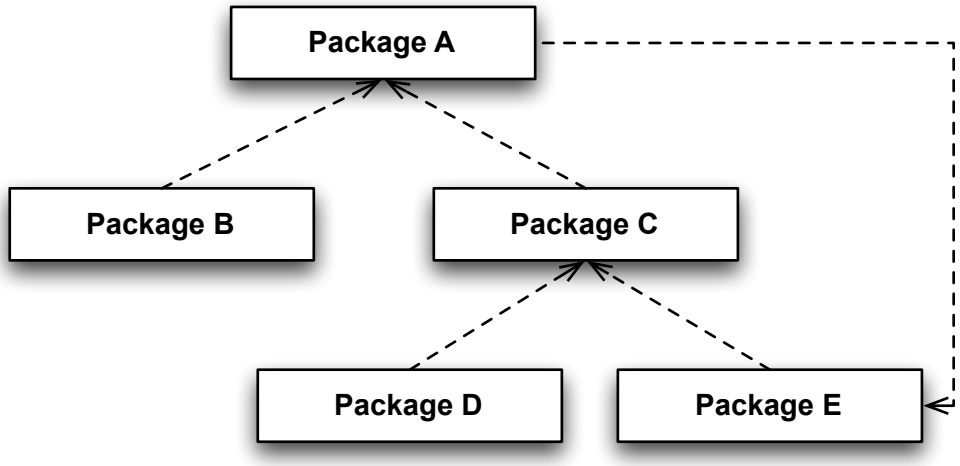
Le principe des dépendances acycliques stipule que les dépendances entre packages doivent former un graphe acyclique orienté.

Considérons en effet l'impact d'un cycle dans le graphe de dépendances :



En cas de rupture d'interface d'une classe de A, la recompilation de A peut être rendue impossible du fait que B utilise l'ancienne version de la classe modifiée. Il faut donc recompiler (ou même modifier) B avant de compiler A, donc A et B doivent évoluer ensemble. On peut alors considérer que A et B forment un unique package.

Le même raisonnement s'applique à des cycles plus larges :

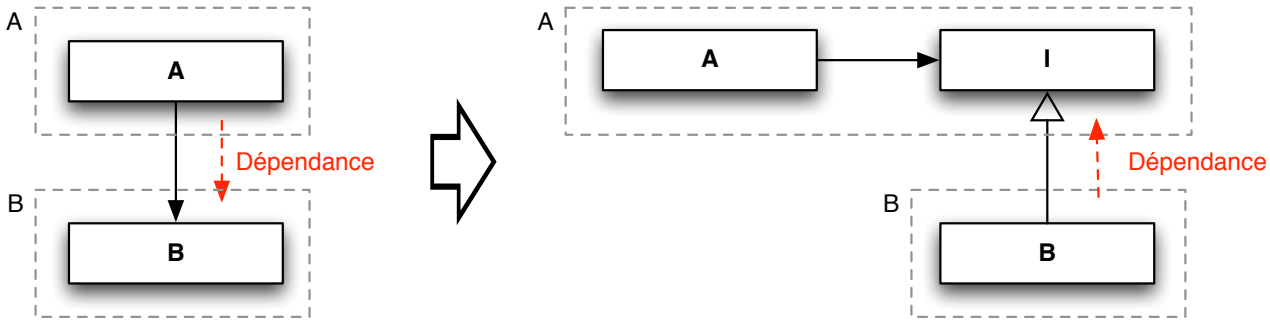


- Pour compiler le package A, il faut utiliser une version de E qui s'appuie sur la version courante de A : A et E doivent donc être compilés en même temps. Le même raisonnement s'applique à C. Résultat : il faut compiler et livrer A, C et E en même temps.
- Le package D utilise le package C, et de proche en proche les packages A et E.
- Les packages A, C et E peuvent donc être considérés comme un seul et même package.

Pour tirer parti de l'effet de temporisation de la décomposition en packages, il faut donc éviter tout cycle dans le graphe de dépendances.

11.2.TECHNIQUE D'INVERSION DES DÉPENDANCES

La technique d'inversion de dépendances est celle présentée dans le principe d'inversion des dépendances, et repose sur l'introduction d'une classe d'interface :



Dans l'exemple de gauche, une classe A du package A utilise une classe B du package B. Le package A dépend donc du package B. Pour inverser la dépendance, on introduit une classe d'interface I dans le package A, I représentant l'interface de B utilisée par A. B est ensuite définie comme dérivant de I (ou implémentant l'interface I), et A utilise B via son interface I. Résultat : B dérive de I, donc B utilise I, donc le package B dépend du package A.

Il faut par contre ajouter un mécanisme qui permette d'enregistrer l'instance de B auprès de A. Ce mécanisme sera placé dans le package B ou un package tiers qui aura une visibilité sur A et B.

Article original : http://www.objectmentor.com/publications/stability.pdf

12.Principe de relation dépendance/stabilité

Un package doit dépendre uniquement de packages plus stables que lui.

Stable Dependencies Principle - **SDP**

12.1.Une mesure de la stabilité

En première approche, on pourrait définir la stabilité par rapport à une probabilité de changement : telle partie de l'application est définie sur des spécifications floues, et est donc susceptible d'être remise en question par la suite. Cette approche n'est cependant pas suffisamment précise pour qu'on puisse en tirer des principes généraux.

Il existe une manière plus formelle de définir la stabilité d'un module (classe ou package). Pour un module donné :

- Plus le nombre de modules dont il dépend est grand, plus il est susceptible d'être impacté par des modifications d'un de ces modules, et donc moins il est stable.
- Plus le nombre de modules dépendant de ce module est grand, plus les modifications de ce module sont coûteuses, et donc plus il est 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.
- Minimale si le module utilise de nombreux autres modules alors qu'il n'est utilisé lui-même par aucun autre module.

12.2.Intégrer la stabilité dans la conception

Le principe de relation dépendance/stabilité stipule qu'un module donné doit dépendre uniquement de modules plus stables que lui. Cela permet de limiter l'impact des changements les plus fréquents, et maximise donc la stabilité globale de l'application.

Article original : http://www.objectmentor.com/publications/stability.pdf

13.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**

Le Principe d'Ouverture/Fermeture (OCP) a permis de montrer le rôle des classes d'interface dans la stabilisation de certaines parties de l'application. Les parties ainsi stabilisées et leurs interfaces forment les packages les plus stables, mais également les plus abstraits. Les autres packages contiennent les implémentations de ces interfaces, et donc principalement des classes concrètes.

En pratique, et selon le Principe d'Inversion des Dépendances (DIP), ces packages stables portent la logique fonctionnelle de l'application. Ils forment un framework métier qui constitue le squelette de l'application.

Article original : http://www.objectmentor.com/publications/stability.pdf

14.Résumé

Comme précisé en introduction, les principes s'articulent en trois groupes principaux :

1. Gestion des évolutions et des dépendances entre classes,
2. Organisation de l'application en modules,
3. Gestion de la stabilité de l'application.

Voici les principales leçons à retenir pour chacun de ces groupes :

14.1.Gestion des évolutions et des dépendances entre classes

Principe d'ouverture/fermeture
Open-Closed Principle (**OCP**)

Principe de substitution de Liskov
Liskov Substitution Principle (**LSP**)

Principe d'inversion des dépendances
Dependency Inversion Principle (**DIP**)

Principe de séparation des interfaces
Interface Segregation Principle (**ISP**)

Il faut :

- Isoler les parties génériques/réutilisables de l'application en les faisant reposer uniquement sur des classes d'interface,
- Considérer l'héritage comme une implémentation d'interface, la classe dérivée pouvant se "brancher" dans n'importe quel code qui utilise cette interface (l'interface forme alors un contrat entre le code utilisateur et les classes dérivées),
- Utiliser des classes d'interfaces pour créer des pare-feu contre la propagation des changements,
- Construire les parties "techniques" de l'application sur les parties "fonctionnelles", et non l'inverse,
- Utiliser l'héritage multiple pour décomposer les interfaces complexes en interfaces simples correspondant chacune à un service spécifique. Une classe donnée peut ensuite proposer plusieurs services simultanément en implémentant les interfaces correspondantes.

14.2. Organisation de l'application en modules

Principe d'équivalence livraison/réutilisation
Reuse/Release Equivalence Principle (**REP**)

Principe de réutilisation commune
Common Reuse Principle (**CRP**)

Principe de fermeture commune
Common Closure Principle (**CCP**)

Il faut :

- Décomposer l'application en packages pour gérer correctement les versions et permettre une réelle réutilisation,
- Regrouper dans un même package les classes qui sont utilisées ensemble et qui sont impactées par les mêmes changements.

14.3. Gestion de la stabilité de l'application

Principe des dépendances acycliques
Acyclic Dependencies Principle (**ADP**)

Principe de relation dépendance/stabilité
Stable Dependencies Principle (**SDP**)

Principe de stabilité des abstractions
Stable Abstractions Principle (**SAP**)

Il faut :

- Organiser les modules en un arbre de dépendances (en supprimant donc tout cycle dans le graphe des dépendances),
- Placer les packages les plus stables à la base de l'abre,
- Placer les interfaces dans les packages les plus stables.

15.Conclusion

Les principes énoncés ici placent le contrôle des dépendances au coeur de l'activité de conception, dans le but de limiter le coût des modifications et ainsi atteindre les objectifs recherchés d'extensibilité, de robustesse et de réutilisabilité. Ce contrôle repose sur une utilisation efficace des interfaces, qu'il s'agisse d'interfaces entre applications, entre packages ou entre classes.

Ces principes restent bien sûr uniquement des principes : ils n'apportent pas de recettes miraculeuses ou de règles strictes qui rendraient la conception quasiment automatique. Ils constituent par contre un solide cadre de décision et d'évaluation, qui permet d'aboutir à des applications plus facilement extensibles, plus robustes et plus simples.

Cette présentation était volontairement concise et assez abstraite. Pour aller plus loin, nous vous invitons vivement à lire les articles originaux de Robert Martin, les ouvrages recensés dans notre section Livres, ou mieux encore : mettre en pratique nos principes de conception lors de l'un de nos bootcamps !