Chapitre 2 : Principes avancés de conception objet

« Ce n'est pas l'espèce la plus puissante qui survit, mais celle qui s'adapte le mieux au changement. » ***Charles Darwin***

1. Introduction

Le but du génie logiciel est de produire des conceptions extensibles, maintenables et réutilisables. La difficulté est que la conception tient beaucoup d'un art. Toutefois, on peut s'aider du savoir-faire accumulé depuis des années et qui est formalisé sous la forme de :

- **Principes** : notions importantes desquelles dépend la qualité d'une conception.
- **Règles** : ensemble de prescriptions de conception à respecter.
- **Patrons** : modèles de solutions pour des problèmes récurrents.

2. Principes de conception

Ils sont connus sous l'acronyme **SOLID :**

- **[S]**ingle Responsibility Principle : *Principe de responsabilité unique*
- **[O]**pen-Closed Principle : *Principe d'ouverture / fermeture*
- **[L]**iskov Substitution Principle : *Principe de substitution de Liskov*
- **[I]**nterface Segregation Principle : *Principe de ségrégation des interfaces*
- **[D]**ependency Inversion Principle : *Principes d'inversion des dépendances*

2.1. Principe 1 : Responsabilité unique

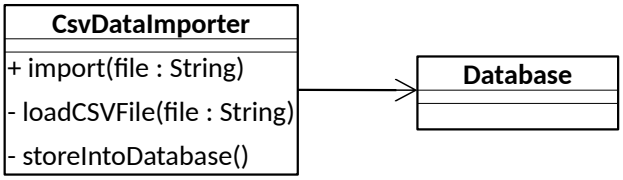
- **Un module (classe, méthode, paquet, etc.) devrait n'avoir qu'une responsabilité unique**.

La responsabilité unique doit s'entendre comme une seule raison de changer. Donc, la phrase précédente peut être reformulée en : un module ne devrait jamais avoir plus d'une raison de changer.
Le but est évidemment d'augmenter la cohésion.
Péril majeur d'un module à responsabilités multiples :

- **Immobilité** : Il est impossible de ne réutiliser qu'une partie du module sans réutiliser le module entier.

2.1.1. Exemple

Un composant permet d'importer des données d'un fichier CSV dans une base de données. La modélisation proposée ci-dessous semble tout à fait correcte. La classe `CsvDataImporter` détient les méthodes pour importer des données issues d'un fichier au format CSV et les stocker dans la base de données associée.



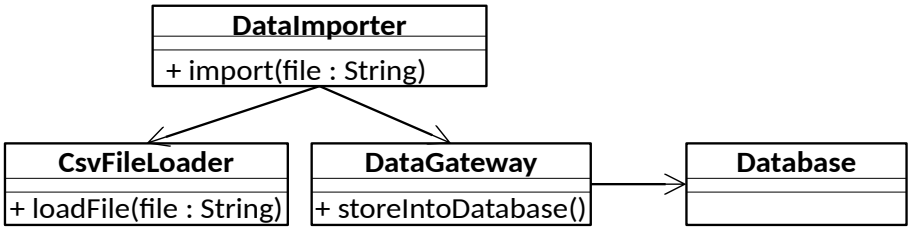
Cependant, pour un développeur, cette modélisation est incorrecte. Il est évident que la classe `CsvDataImporter` possède plus d'une responsabilité. En effet, elle réalise 2 fonctions de nature complètement différente :

1. Lire un fichier CSV et transformer les données en tableaux.
2. Importer ces tableaux dans une base de données locale.

Il y a donc clairement deux raisons pour que la classe change dans un futur proche. La première est le changement du format de sérialisation des données. Il faudra modifier le code de la méthode `loadFile()` si demain les données sont issues d'un fichier XML ou JSON. La seconde concerne le moyen de stockage. La méthode `import()` devra être réécrite pour l'adapter vers une sauvegarde dans un cloud par exemple.

Comment augmenter la cohésion ?

Il faut séparer le chargeur de fichier et la passerelle de stockage. On aboutit à la modélisation suivante, construite à partir de 3 petites classes faciles à changer, faciles à étendre et faciles à tester.



2.2. Principe 2 : Ouverture / Fermeture

- **Un module doit être ouvert aux extensions, mais fermé aux modifications**.

Nous devons pouvoir ajouter une nouvelle fonctionnalité en ajoutant du nouveau code et non en éditant du code existant.

Péril majeur d'un code fermé.

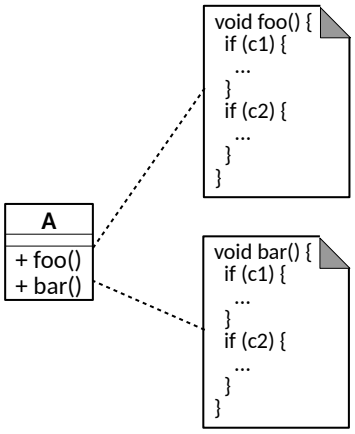
- **Fragilité** : la modification de code existant entraîne une régression en introduisant de nouvelles erreurs dans les parties préexistantes.

2.2.1. Exemple

Considérons la classe A ci-à-droite avec deux méthodes `foo()` et `bar()` qui dépendent des deux attributs `c1` et `c2`.

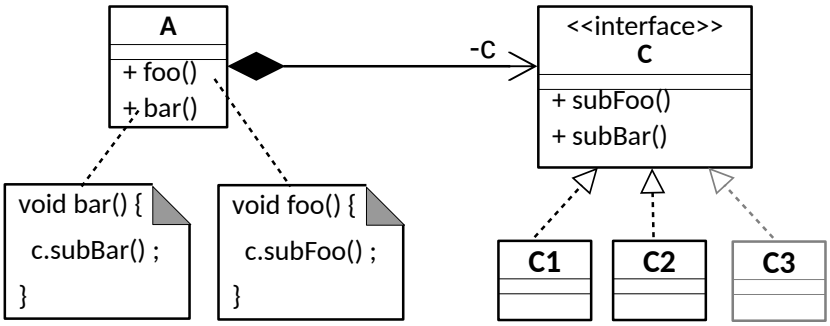
Quelle est la faiblesse de cette conception ?

- Elle est fermée à l'introduction d'une nouvelle alternative pour un attribut `c3`. L'ajout nécessite d'éditer le code des méthodes `foo()` et `bar()`.



Comment rendre la classe A ouverte aux extensions et fermée aux modifications ?

- Composition, abstraction et polymorphisme. Le code des méthodes `foo()` et `bar()` est externalisé et éclaté dans des classes spécialisées pour chacun des cas `c1` et `c2`. Cette fois, il est facile d'ajouter la variation sur l'attribut `c3`, simplement en ajoutant une classe `C3` sous l'interface `C`. Ceci se fait sans modifier de code existant.



2.2.2. Corollaire : le principe de choix unique

Aucun module ne peut être ouvert à 100 %. Par exemple, dans l'exemple précédent, il y a quelque part du code qui permet de choisir entre les classes `C1`, `C2` ou `C3`. Dans ce cas, ce code doit être regroupé dans une seule méthode ou une seule classe qui connaît l'ensemble des alternatives ; cf. les patrons de conception Fabrique et Fabrique abstraite pour l'implémentation.

2.3. Principe 3 : Substitution de Liskov[1]

- **Les objets de classes dérivées doivent être substituables aux objets de la classe de base**.

Les objets de la classe dérivée doivent se comporter d'une manière compatible avec les promesses faites dans le contrat de la classe de base.

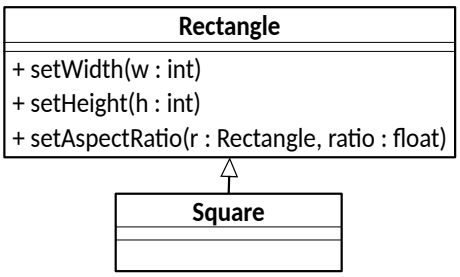
- Si S est un sous-type de T, alors tous les objets de T doivent être substituables par tout objet de S. Par exemple, lorsque T est utilisé comme paramètre d'une fonction, il doit être substituable par un objet de S.
- Cela revient à dire que la classe de base est une interface exportée par toutes ses sous-classes.

Péril d'une hiérarchie non substituable.

- **Fragilité** : le code créé pour une classe devient inapproprié pour une de ses sous-classes quelque part dans le logiciel. Les effets indésirables ne se révéleront qu'à l'exécution.

2.3.1. Exemple

Soit la modélisation ci-à-droite que l'on trouve comme exemple d'héritage dans beaucoup de manuels de conception orientée objet : un carré est un rectangle particulier. L'objectif visé est la réutilisation de la majorité du code du rectangle pour le carré.



Pourquoi cette modélisation est fausse ?

- Il y a certes une petite perte de mémoire due au stockage

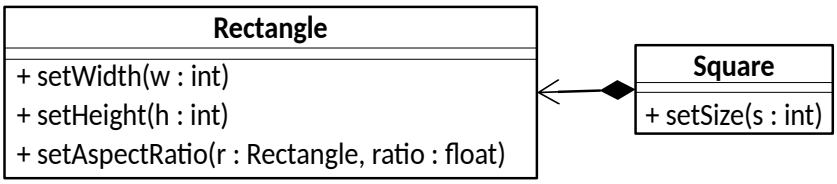
1 Barbara Liskov, prix Turing 2008

des deux valeurs de dimension pour le Carré alors qu'il n'en utilise qu'une, mais ce n'est pas le plus grave.

- Le carré ne respecte pas tout le contrat de *Rectangle*. Par exemple, la méthode *setAspectRatio()* (*4/3*, *16/9*, *A4*, *etc*) n'a certainement pas le comportement attendu avec un carré.

Comment modifier la conception pour éviter le problème de substitution mais sans dupliquer le code commun ?
La solution passe par l'utilisation de la règle « préférer la composition à l'héritage ».

- Le carré n'hérite plus de rectangle.
- Le carré utilise le rectangle par composition.



Cette fois le carré n'est pas substituable au rectangle et le contrat de chacune des classes est respecté sans duplication de code.

2.4. Principe 4 : Ségrégation d'interface

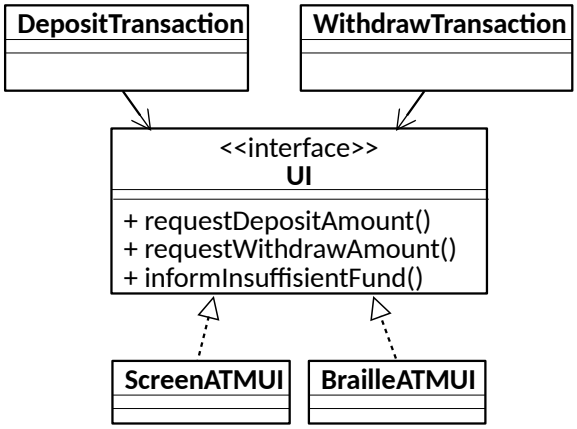
- **La dépendance d'une classe à une autre devrait être restreinte à l'interface la plus petite possible**.

Le client d'une classe ne doit pas être forcé de dépendre de méthodes qu'il n'utilise pas.
Péril majeur d'interfaces complexes :

- **Rigidité :** le client se trouve affecté par des changements dans des méthodes qu'il n'utilise pas.

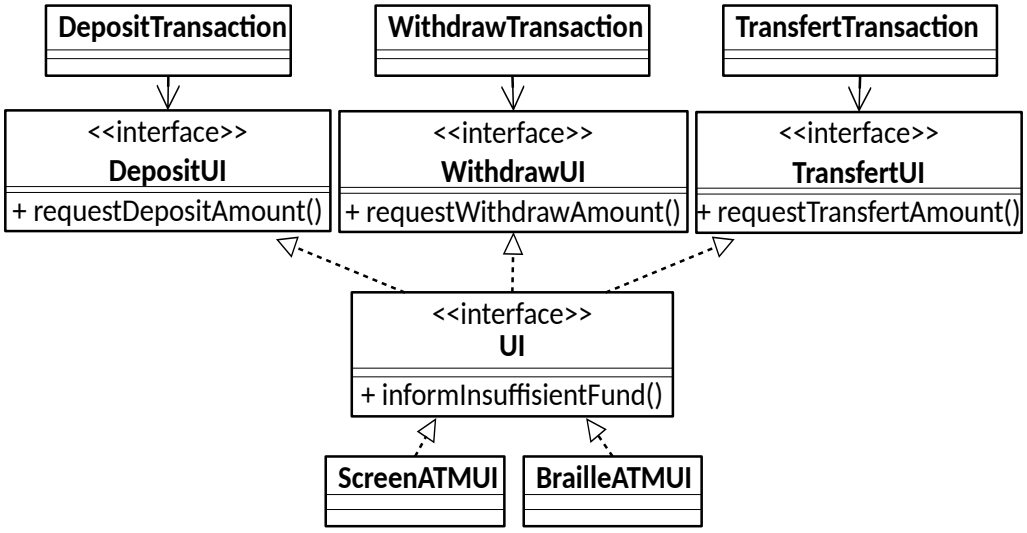
2.4.1. Exemple

L'écran utilisateur d'un Guichet Automatique Bancaire (GAB) doit être déclinable en deux versions : tactile et braille. Les services de transaction sont représentés par une classe chacun, ici dépôt et retrait d'argent. Les classes interagissent avec l'interface qui référence toutes les formes de sollicitation de l'écran. Ainsi, le service de retrait utilise la méthode `RequestDepositAmount()` qui sera effectuée par écran tactile ou écran en braille.



Quels sont les problèmes potentiels de cette modélisation ?

- Si on ajoute une classe `TransfertTransaction` pour effectuer des transactions de compte, on devra ajouter une nouvelle méthode `requestTransfertAmount()` dans l'interface `UI` et donc dans les classes qui l'implémentent, ce qui semble normal. Mais, le problème est que l'on va devoir recompiler toutes les classes qui dépendent de l'interface `UI` alors qu'elles n'utilisent pas cette nouvelle méthode. Sentez-vous le relent de pourrissement ?
- Comment restructurer la conception pour n'avoir que des interfaces cohérentes ?
    - Ségrégation par héritage d'interface.



Cette fois, chaque classe de transaction n'est liée qu'à une interface minimale, sous-ensemble de l'interface intégrale construite par héritage.

2.5. Principe 5 : Inversion des dépendances

- **La relation de dépendance conventionnelle que les modules de haut niveau (aspect métier) ont par rapport aux modules de bas niveau (aspect implémentation) est inversée dans le but de rendre les premiers indépendants des seconds.**

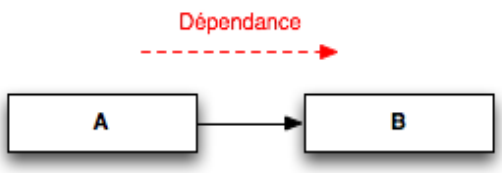
Les abstractions ne doivent pas dépendre de détails. Ce sont les détails qui doivent dépendre des abstractions.

Périls majeurs de dépendances conventionnelles :

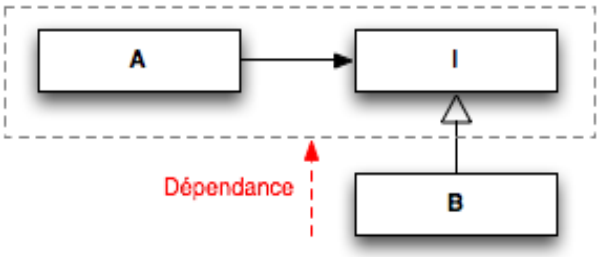
- **Rigidité :** les modules fonctionnels sont impactés par les changements des modules techniques.
- **Immobilité** : les modules génériques ne peuvent pas être réutilisés parce qu'ils sont liés à une implémentation particulière.

2.5.1. L'abstraction comme technique d'inversion des dépendances

- Relation conventionnelle : la classe A est liée à la classe B et ne peut pas être réutilisée sans B.

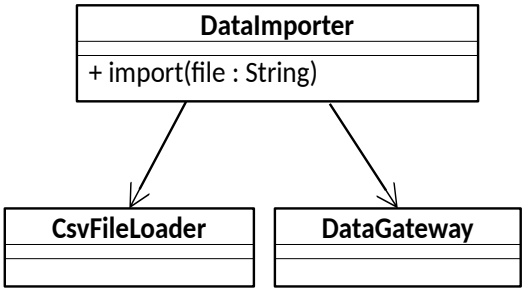


- Inversion de la dépendance : les modules de bas niveau doivent être conformes aux interfaces définies par les modules de haut niveau. Cette fois A (avec l'interface I) est réutilisable sans B.



2.5.2. Exemple

En reprenant la solution de l'exemple de la section 2.1.1, la classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage.

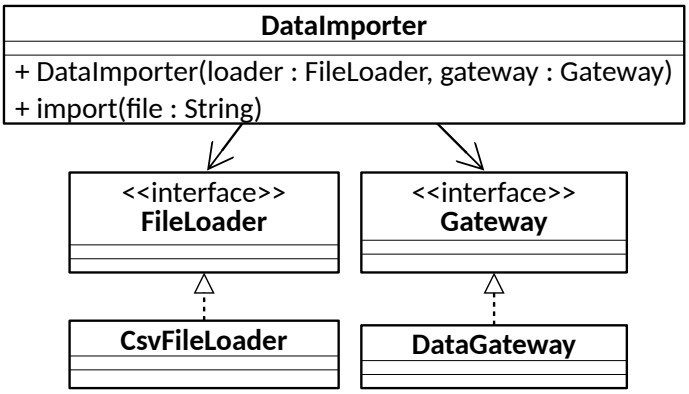


Quel est le problème de cette solution ?

- On ne peut pas réutiliser la classe d'importation `DataImporter` sans réutiliser le chargeur de fichier CVS et la passerelle de stockage.

Comment rendre `DataImporter` indépendant des implémentations du chargeur du fichier CVS et de la passerelle ?

- Il faut inverser les dépendances en utilisant des interfaces comme dans le schéma ci-dessous.



2.5.3. Implications

On peut tirer trois implications pour réaliser une conception :

- Ne pas dériver d'une classe concrète.

- Ne pas agréger une classe concrète.
- Ne pas dépendre d'une classe concrète.

3. Règles de conception

En utilisant le paradigme orienté objet, 5 règles fondamentales régissent la conception :

- Règle 1. Réduire l'accessibilité des membres de classe.
- Règle 2. Encapsuler ce qui varie.
- Règle 3. Programmer pour une interface et non pour une implémentation.
- Règle 4. Privilégier la composition à l'héritage.

Nous ajoutons une règle plus particulière :

- Règle 5. Loi de Démeter.

3.1. Règle 1 : Réduire l'accessibilité des membres de classe

- **L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même.**

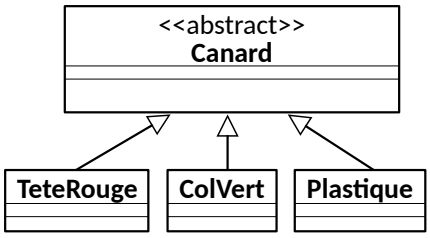
Il s'agit d'éviter d'exposer les détails de mise en œuvre de la classe pour faciliter l'évolution future sans aucune conséquence sur les autres classes qui l'utilise. Cette règle est la mise en pratique du principe d'encapsulation et consiste à :

- faire des attributs privés,
- réduire l'utilisation des accesseurs et mutateurs. Leur nécessité est souvent révélatrice d'une mauvaise répartition des responsabilités qui conduit à réaliser un traitement dans une classe avec des valeurs d'attributs d'une autre classe, plutôt que de déléguer le traitement à la classe qui possède les attributs ou au moins la partie du traitement qui utilisent ces attributs. Dans ce cas, la classe contenant les attributs est considérée comme une simple structure de données, ce qui est contraire aux principes de la conception orientée objet.

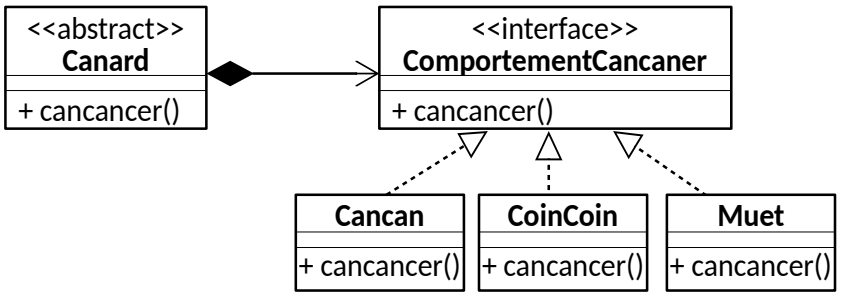
3.2. Règle 2 : Encapsuler ce qui varie

- **Identifier ce qui devrait être variable dans une conception puis encapsuler ce qui varie dans une hiérarchie propre.**

Toutes les formes de variation sont modélisables par une hiérarchie de classes. La variation sur un concept est naturelle en conception orientée objet :



Mais, on peut aussi représenter des variations sur une méthode où une hiérarchie est entièrement composée pour représenter les variations d'une seule méthode :



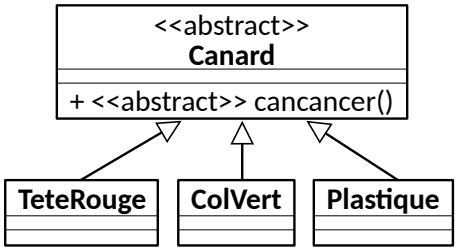
Nous verrons plus tard d'autres formes de variation.

3.3. Règle 3 : Programmer pour une interface et non pour une implémentation

- **Il faut programmer avec des supertypes au lieu de types concrets. Le code qui utilise des supertypes est facilement extensible sans beaucoup de changement, alors que le code qui repose sur des types concrets doit être profondément changé pour être étendu.**

Attention, en conception la notion d'interface fait référence à un supertype qu'il ne faut pas restreindre à la notion d'interface en Java. Une interface en conception se code en Java par une classe abstraite ou par une interface.

En guise d'illustration de l'application de cette règle, considérons la modélisation suivante :



- Programmer pour une implémentation :

```
ColVert c = new Colvert();
c.cancane();
```

Le code est totalement dépendant du type ColVert. Ce code n'est pas adaptable à un autre type de Canard sans changer le code, d'autant plus s'il utilise des méthodes spécifiques de ColVert.

- Programmer pour une interface :

```
Canard a = new ColVert();
a.cancane();
```

Ainsi le code est générique au type Canard, ce qui permet de considérer n'importe quel type de Canard sans rien changer par ailleurs :

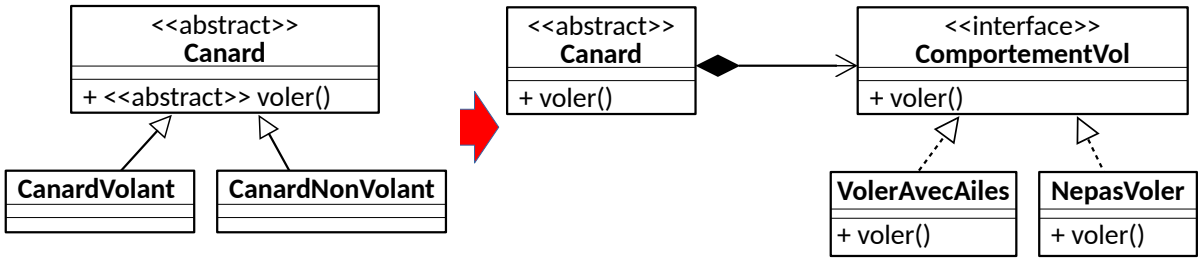
```
Canard a = getCanard();
a.cancane();
```

3.3.1. Règle 4 : Privilégier la composition à l'héritage

- **La conception est simplifiée par l'identification des comportements d'objets dans des interfaces**

séparées au lieu de créer une relation hiérarchique pour répartir les comportements entre les classes métier par héritage.

- L'héritage rompt l'encapsulation (*création de type boîte blanche*).
- Au contraire, la composition est définie dynamiquement (*création de type boîte noire*).



La règle de Peter Coad pour légitimer le recours à l'héritage

Il faut utiliser l'héritage quand tous les critères suivants sont satisfaits :

- La sous-classe représente un « type spécial » de la super-classe et non un « rôle joué » par la super-classe.
- Une instance de la sous-classe ne doit jamais devenir un objet d'une autre sous-classe.
- La sous-classe étend plutôt que redéfinie ou annule les responsabilités de la super-classe.
- La classe de base n'est pas une classe utilitaire qui détient des fonctionnalités qui sont simplement réutilisées dans la sous-classe.

Prenons l'exemple d'une classe utilitaire nommée `Statistics` qui calcule la moyenne et l'écart-type d'un signal donné. Nous voulons créer une classe `Butterworth` pour faire un filtrage du signal en profitant des méthodes de la classe `Statistics`. La solution simple qui consiste à faire dériver `Butterworth` de `Statistics` pour en récupérer les méthodes ne respecte pas la règle de Coad et conduira inévitablement à des problèmes de modélisation plus tard. `Butterworth` ne doit pas hériter de `Statistics`, parce que `Butterworth` n'est pas une `Statistics`. En autres problèmes, elle hérite de méthodes qu'elle n'utilise pas.

3.4. Règle 5 : Loi de Déméter[2]

- **Un objet A ne doit pas utiliser un objet B pour accéder à un troisième objet C pour requérir ses services**, que l'on peut aussi résumer à : « ne parlez qu'à vos amis immédiats ».

Ne pas respecter ce principe signifierait que A a une connaissance plus grande que nécessaire de la structure interne de B (ie, entre autres que B est composé de C).

Au lieu de cela, il est préférable de modifier B pour que A puisse faire la requête directement à B, et B propagera la requête au composant approprié. Ainsi, seul la classe B a la connaissance de C.

En résumé : il faut bannir `b.c.m()` par `b.m()` et `B::m() { c.m(); }`.

Avantage

- accroître la maintenabilité et la robustesse du logiciel.

Désavantage :

2 Nom d'un projet de recherche en 1987.

- requière un grand nombre de petites méthodes « wrapper » dans B pour propager les appels des méthodes de C.

4. Patrons de conception

4.1. Réutilisabilité

Une grande partie de l'activité de développement de logiciels se fait à l'aide de savoir-faire récurrents. Nous procédons par recopie, imitation et réutilisation de solutions qui ont fait leurs preuves d'efficacité dans le passé. La réutilisation concerne tous les niveaux du cycle en V.

- Analyse des besoins
- Spécification fonctionnelle
- Conception architecturale
- Conception détaillée
- Programmation
- Tests

En fonction des niveaux, il existe aussi plusieurs formes de représentation du savoir-faire, de la plus concrète à la plus abstraite.

4.2. Idiome

Un idiome est une construction récurrente dans un langage de programmation particulier.

- Il est non transposable dans un autre langage de programmation.
- Les idiomes sont décrits dans les manuels de programmation avancée.

Exemple : parcours d'une chaîne de caractères

- En C

```
void handleCString( const char * s) {
  for (;*s ;) {        // basé sur la connaissance que la représentation
    function(*s++);    // d'une chaîne en C se termine par '\0'
  }
}
```

- En Java

```
void handleJavaString( String s ) {
  s.chars().forEach(c -> function(c)); // Utilisation des streams
}                                      // qui cachent la représentation
```

4.3. Bibliothèque de classes

Une bibliothèque est une collection de classes et de fonctions. Elle est souvent composée des classes concrètes, connexes mais indépendantes, sans comportement par défaut. La bibliothèque ne prescrit aucune méthode de conception spécifique et elle est spécifique d'un langage de programmation.

Exemples :

- X11, JavaFX, Qt

- C++ STL, Boost
- Java SE, Guava

4.4. Framework

C'est un ensemble de classes qui composent un modèle d'application. Il est constitué de classes abstraites et concrètes. Les classes sont définies pour être utilisées ensemble en fournissant un comportement par défaut. On distingue :

- Framework de type boîte blanche
  - Basé sur l'héritage.
  - Prêt à l'emploi par dérivation de classes du framework.
- Framework de type boîte noire
  - Basé sur la composition.
  - Prêt à l'emploi par assemblage des classes du framework.

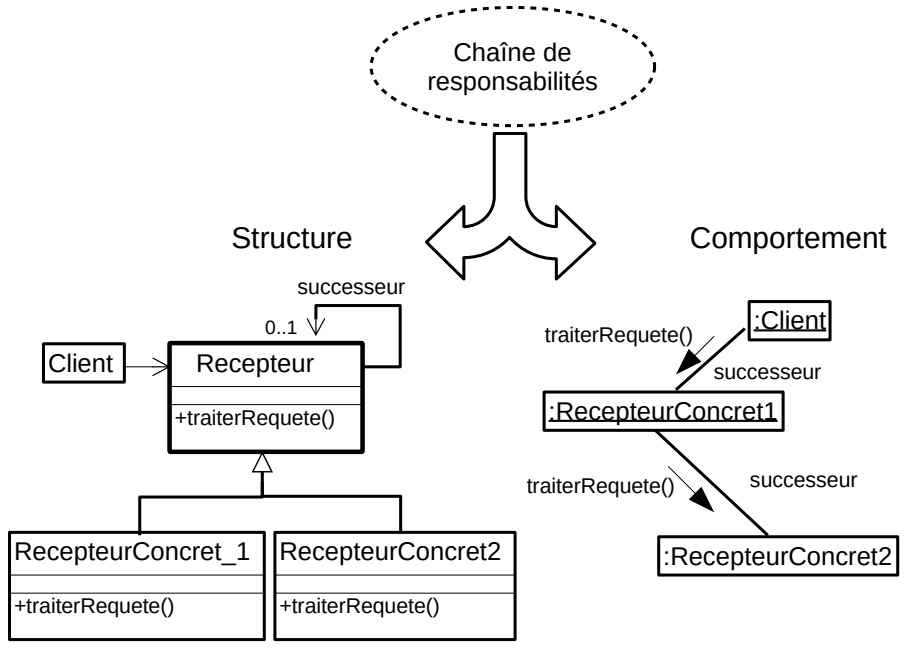
Exemples

- Java EE (Jakarta EE)
- Spring boot
- .NET,
- Struts.

4.5. Patron (*Pattern*)

Un patron décrit un problème récurrent et une solution pour ce problème. Il modélise une façon de faire pour arriver à une solution. Il est basé sur l'expertise de précurseurs en la matière. Il se modélise au niveau UML.

L'exemple ci-après modélise le patron « chaîne de responsabilité » sous la forme de diagrammes UML qui donnent les classes, leur organisation structurelle et le fonctionnement par leurs interactions :



4.6. Anti-Patron (*Anti-Pattern*)

Un anti-patron correspond à un patron qui peut être couramment employé, mais qui est inefficace voire néfaste en pratique. Il représente une leçon apprise. Ils doivent permettre d'identifier des situations courantes pour lesquelles certaines solutions ne doivent pas être utilisées parce qu'elles conduisent à un échec.

5. Conclusion

5.1.1. Que retenir de ce chapitre

- Il faut isoler les parties génériques (réutilisables) de l'application en les faisant reposer sur des interfaces.
- Considérer l'héritage comme une implémentation d'interface ; la classe dérivée pouvant se brancher sur n'importe quel code qui utilise cette interface.
- Utiliser l'héritage multiple pour décomposer les interfaces complexes en interfaces simples correspondant chacune à un service spécifique.
- Construire les parties « techniques » de l'application sur les parties « fonctionnelles », et non l'inverse.

5.1.2. Remarque

Ces principes et ces règles ne fournissent pas de recettes miracles ou des lois absolues qui font de la conception un processus automatique. Il ne faut pas les appliquer pour toutes les conceptions. Par exemple, le principe de responsabilité unique accroît le couplage. Il ne faut les appliquer que quand l'extension et la réutilisation sont des contraintes importantes, par exemple durant le processus de *refonte de code*. Cependant, ces principes et ces règles doivent être une préoccupation constante. Il faut trouver une raison de ne pas les appliquer !