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 recherché est de produire des conceptions extensibles, maintenables et réutilisables. Le problème est que la conception tient beaucoup de l'art. Mais, on peut s'aider du savoir-faire accumulé depuis des années et qui est formalisé sous la forme de :

- **Principes** : Base sur laquelle repose l'organisation de quelque chose et qui en régit le fonctionnement.
- **Règles** : un ensemble de prescriptions de conception à respecter.
- **Patrons** : des modèles de solutions à 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

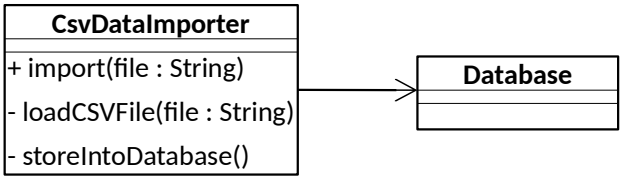
- **Un module (classe, méthode, paquet, etc.) devrait n'avoir qu'une responsabilité unique**.

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

L'exemple est celui d'un composant qui 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.

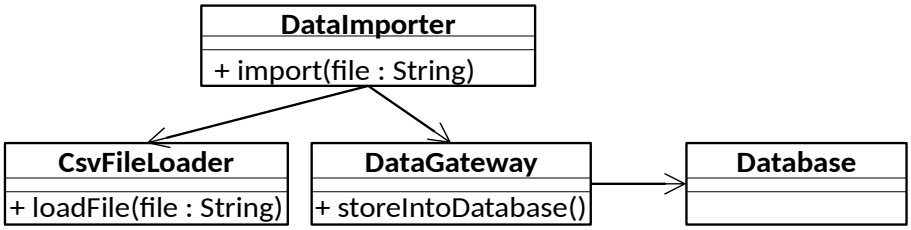


Cependant, pour un développeur, cette modélisation est incorrecte. Il est évident que la classe `CsvDataImporter` possède plus d'une responsabilité. En effet, elle réalise 2 fonctions de nature complètement différente :

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

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 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 un autre moyen de stockage.

Comment augmenter la cohésion ? Il faut séparer le chargeur de fichier et la passerelle de stockage. On aboutit à la modélisation suivante. Elle est 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.

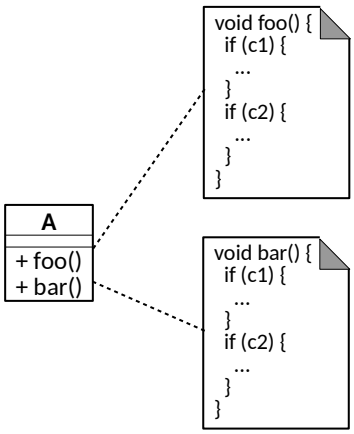
Péril majeur d'un code fermé.

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

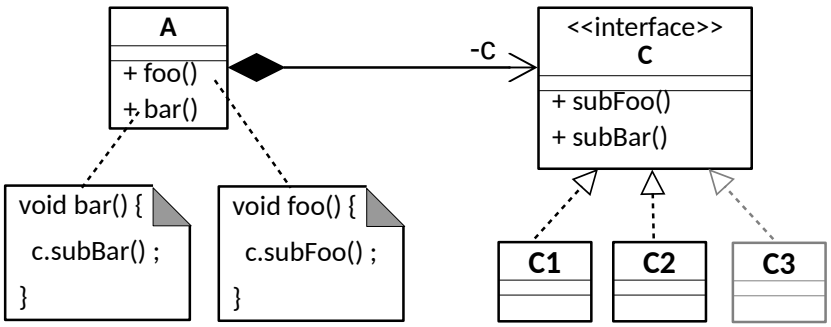
2.2.1. Exemple

- Considérons la classe A ci-à-droite avec deux méthodes 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 la rendre ouverte aux extensions et fermée aux modifications ?
    - Composition, abstraction et polymorphisme. 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 programme ne peut être ouvert à 100 %. Par exemple, dans l'exemple précédent, il y a quelque part quelqu'un qui doit choisir entre les classes C1, C2 ou C3. Dans ce cas, une seule méthode ou une seule classe dans le système doit connaître l'ensemble des alternatives ; cf. les patrons de conception Fabrique et Fabrique abstraite.

2.3. Principe 3 : Substitution de Liskov

- **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.

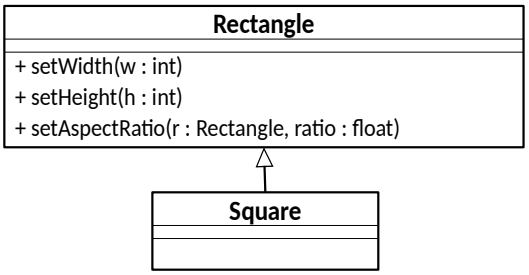
- Si S est un sous-type de T, alors tous les objets de T doivent être substituables par tout objet de S.
- Par exemple, lorsque T est utilisé comme paramètre d'une fonction, il doit être substituable par un objet de S.
- Cela revient à dire que la classe de base est une interface exportée par toutes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

2.3.1. Exemple

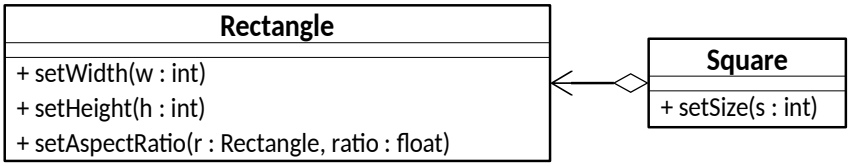
Soit la modélisation suivante que l'on trouve comme exemple d'héritage dans beaucoup de manuels de conception orientée objet : un carré est un rectangle particulier.



Pourquoi cette modélisation est fausse ?

- Il y a certes une petite perte de mémoire du au stockage de deux valeurs de dimension pour le Carré, 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 ?



- Cette fois :
  - Le carré n'hérite plus de rectangle.
  - Le carré utilise le rectangle par composition.
- Cette fois le carré n'est pas substituable au rectangle.

2.4. Principe 4. Ségrégation d'interface

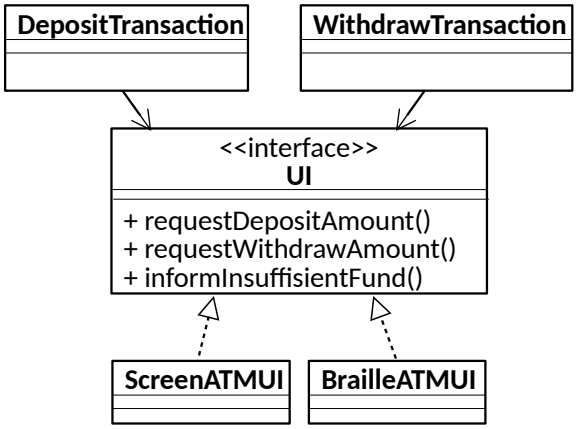
- **La dépendance d'une classe à une autre devrait être restreinte à l'interface la plus petite possible**.

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

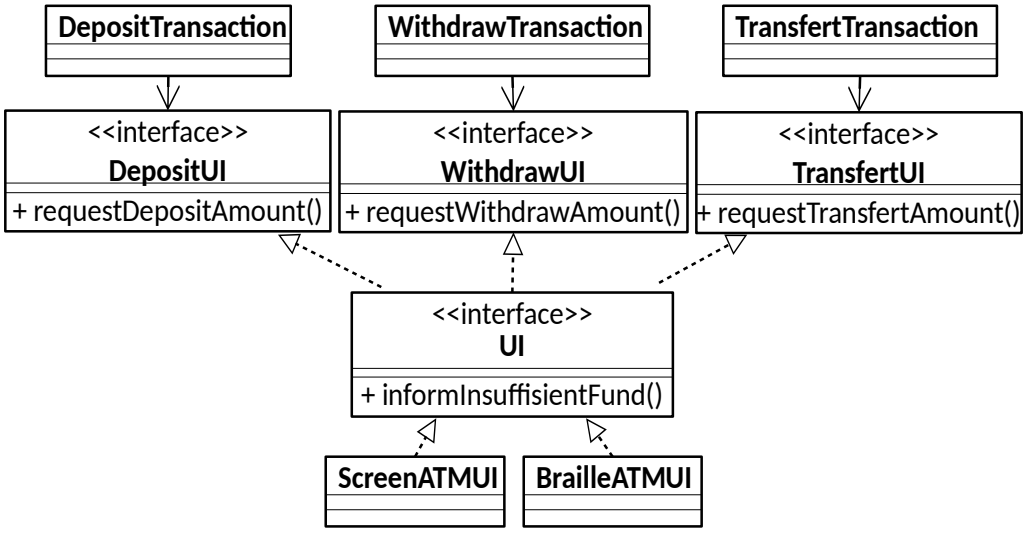
L'écran utilisateur d'un Guichet Automatique Bancaire (GAB) doit être déclinable en deux versions : en écran tactile et en braille. Les services de transaction représentés par une classe chacun, ici de dépôt et de retrait d'argent, interagissent avec l'interface qui référence toutes les formes de sollicitation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



Quels sont les problèmes potentiels de cette modélisation ?

- La modification ou l'ajout d'une méthode impacte toutes les classes dépendantes même si elles n'utilisent pas la méthode. Si on ajoute une classe 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 est normal. Par contre, le problème c'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'interfaces.



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

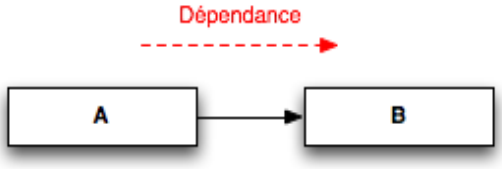
- **La relation de dépendance conventionnelle que les modules de haut niveau (aspect métier) ont par rapport aux modules de bas niveau (aspect implémentation), est inversée dans le but de rendre les premiers indépendants des seconds.**

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

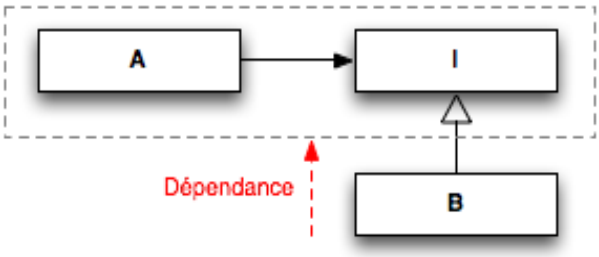
- Périls majeurs de dépendances conventionnelles
    - **Rigidité :** Les modules fonctionnels sont impactés par les changements des modules techniques.
    - **Immobilité** : Les modules génériques ne peuvent pas être réutilisés parce qu'ils sont liés à une implémentation particulière.

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.

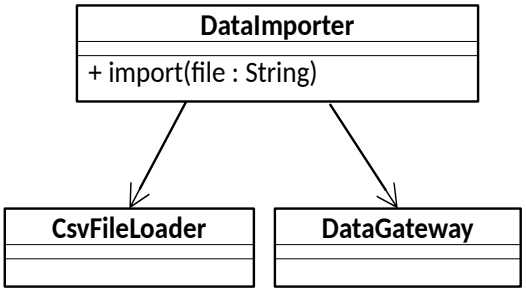


- Inversion de la dépendance : Les modules de bas niveau doivent être conformes aux interfaces définies par des modules de haut niveau. Cette fois A (avec l'interface I) est réutilisable sans B.



2.5.2. Exemple

La classe `DataImporter` est dépendante du chargeur de fichier et de la passerelle de stockage.



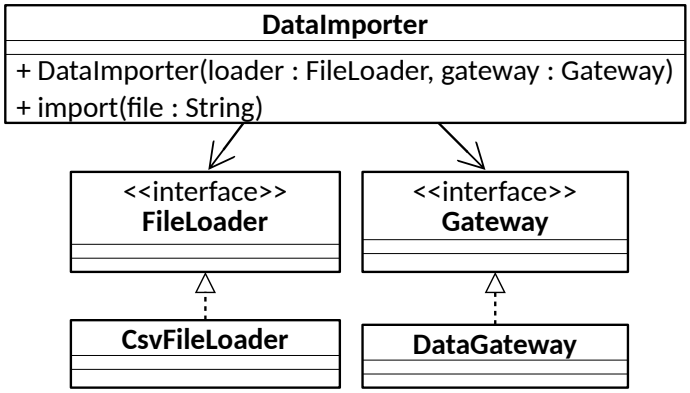
Quel est le problème ?

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

2.5.3. Exemple (refactoring)

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

- Inverser les dépendances avec des interfaces.



2.5.4. Implications

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

2.6. Loi de Déméter

- **Un objet A ne doit pas utiliser un objet B pour accéder à un troisième objet C pour requérir ses services**.
    - Faire cela signifierait que A a une connaissance plus grande que nécessaire de la structure interne de B (ie, que B est composé de C).
    - Au lieu de cela, B pourrait être modifié si nécessaire pour que A puisse faire la requête directement à B, et B propagera la requête au composant ou sous-composant approprié. La classe B a la connaissance de C.
- Avantage
    - Accroître la maintenabilité et la robustesse du logiciel.
- Désavantage
    - Requière un grand nombre de petites méthodes « wrapper » dans B pour propager les appels des méthodes de C.

3. Des règles de conception

- 4 règles :
    - 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.

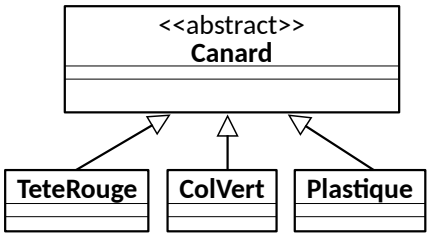
3.1. Règle 1. Réduire l'accessibilité des membres de classe

- **L'accès direct aux données membres d'une classe devrait être limité à la classe elle-même.**
    - Éviter d'exposer les détails de mise en œuvre pour faciliter l'évolution future sans aucune conséquence sur la classe.
- Solution : encapsulation.

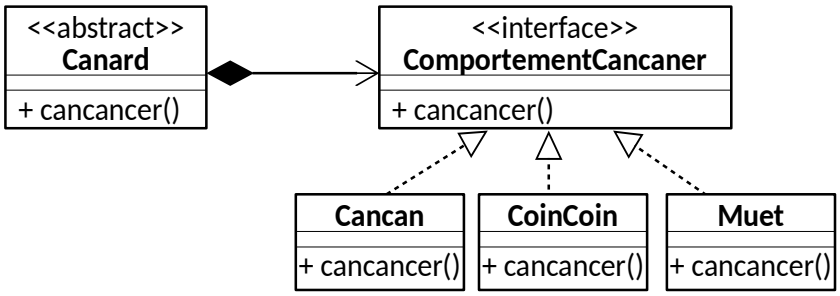
- Faire des attributs privés.
- Réduire l'utilisation des accesseurs et mutateurs. Leur nécessité est souvent révélatrice d'une mauvaise répartition des responsabilités.
    - Par exemple, faire des traitements dans une classe avec des valeurs d'attributs d'une autre classe, plutôt que de faire les traitements dans la classe qui possède les attributs.

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.**
    - Variation sur un concept :



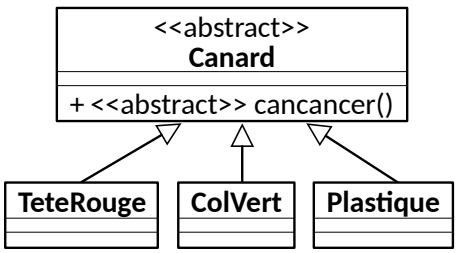
    - Variation sur une méthode :



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

- **Il faut programmer avec des supertypes (interfaces ou classes abstraites) au lieu d'instances. Les supertypes sont faciles à modifier alors que leur implémentation est difficile à changer.**

Par exemple, soit la modélisation suivante :



- Programmer pour une implémentation :

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

- Programmer pour une interface :

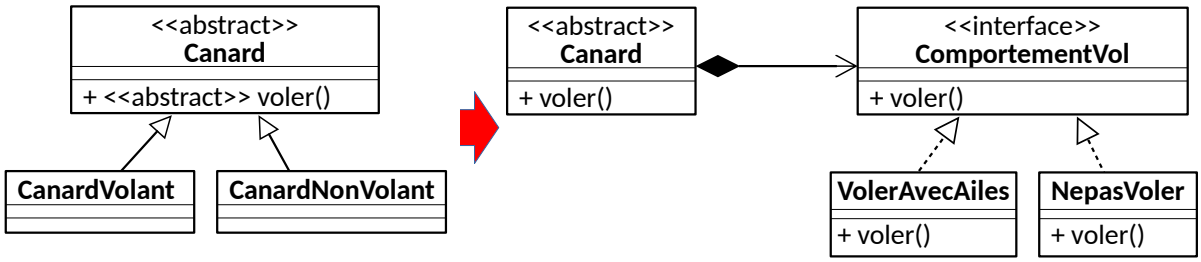
```
Canard a = new ColVert();
a.cancane();
```

- Ce qui permet des évolutions 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 du système 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 boîte blanche*).
  - La composition est définie dynamiquement (*création boîte noire*).



3.4. La règle de Peter Coad

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.

3.4.1. Exemple

Soit une classe utilitaire nommée `Statistics` qui calcule la moyenne, l'écart-type pour un signal. Nous voulons créer une classe `Butterworth` pour faire un filtrage sur un signal en profitant des méthodes de la classe `Statistics`. La solution simple qui consiste à faire dériver `Butterworth` de `Statistics` pour récupérer les méthodes de `Statistics` 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 une `Statistics`.

4. III. 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

Dans ce cours, nous nous limitons :

1. Patrons d'architecture
2. Patrons de conception

Toutefois, au préalable, nous présentons brièvement d'autres formes de réutilisation.

4.1. Idiome

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

- 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 ;) { // fin de la chaîne avec '\0'
    function(*s++);
  }
}
```

- En Java

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

4.2. Bibliothèque de classes

Collection de classes et de fonctions.

- Souvent des classes concrètes.
- Classes connexes mais indépendantes.
- Sans comportement par défaut.
- Ne prescrit aucune méthode de conception spécifique.
- Spécifique d'un langage de programmation.

Exemples

- X11, JavaFX, Qt
- C++ STL, Boost

- Java SE, Guava

4.3. Framework

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

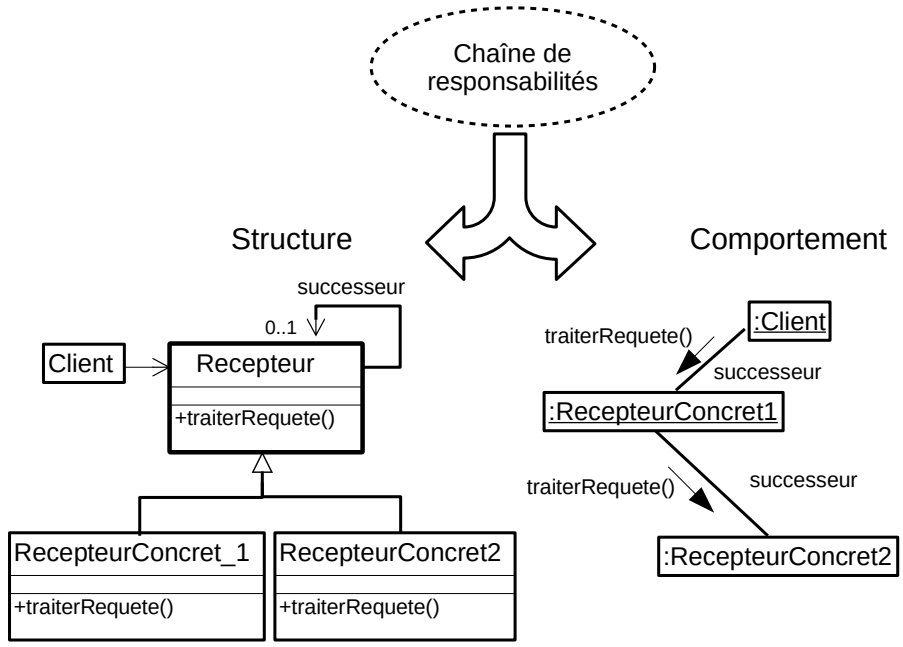
- Framework boîte blanche
  - Basé sur l'héritage.
  - Prêt à l'emploi par dérivation de classe.
- Framework boîte noire
  - Basé sur la composition.
  - Prêt à l'emploi par composition des classes entre elles.

Exemples

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

4.4. Patron (*Pattern*)

Un pattern décrit un problème récurrent et une solution pour ce problème. Il modélise une façon de faire, d'arriver à une solution. Il est basé sur l'expertise de précurseurs en la matière.



4.5. 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.

Il y en a de deux catégories :

- Solution d'un problème qui conduit à un échec.
- Comment se sortir d'une mauvaise situation et continuer à partir de là vers une bonne solution.

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 dans 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 !