La conception Objet

Rezak AZIZ
rezak.aziz@lecnam.net

CNAM Paris

Principes d'architecture orientée objet

Pourquoi une architecture orientée objet ?

- Organiser le code pour qu'il soit **simple, clair et réutilisable**.
- Concevoir des classes qui représentent bien les **éléments du monde réel**.
- Apprendre comment les classes **coopèrent entre elles**.
- Découvrir les **règles de conception** (SOLID, délégation, découplage, etc.).

Références utiles

- Eric Evans, *Domain Driven Design*, 2003
- Martin Fowler, *UML Distilled*, 2003 ;
- Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* 1994
- Barbara Liskov, *Program Development in Java,* 2000
- Robert C. Martin *Clean Architecture*, 2018.
- Bertrand Meyer, *Object Oriented Software Construction,* 1997
- Emmanuel Puybaret, *Les Cahiers du Programmeur Swing,* 2006
- https://martinfowler.com/
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Les buts d'une bonne architecture

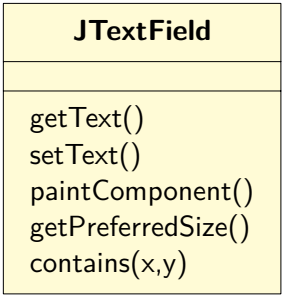
Un bon logiciel doit être :

- **Fiable** : il fait ce qu'on attend.
- **Facile à développer** : chaque classe est testable.
- **Facile à modifier** : une petite évolution ne casse pas tout.
- **Réutilisable** : on peut reprendre une partie du code ailleurs.

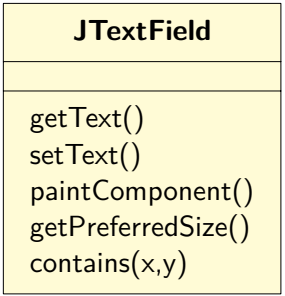
Une classe bien définie, c'est quoi ?

- Une classe doit représenter **un seul concept** et **le représenter bien**.
- Mieux vaut plusieurs petites classes simples qu'une seule énorme.
- Moins une classe dépend des autres, plus elle est **facile à comprendre et tester**.
- C'est comme une table bien normalisée en base de données.

Exemple : une classe qui fait trop de choses



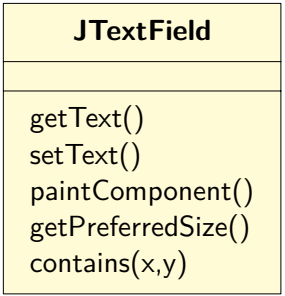
Exemple : une classe qui fait trop de choses



Problème : `JTextField` dans Java gère :

- le contenu du texte ;
- l'affichage ;
- la gestion des événements.

Exemple : une classe qui fait trop de choses



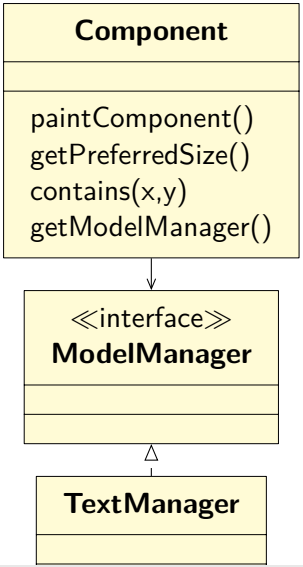
Problème : `JTextField` dans Java gère :

- le contenu du texte ;
- l'affichage ;
- la gestion des événements.

Solution : séparer les rôles :

- `Component` : gère l'affichage ;
- `TextManager` : gère le contenu.

Une classe = une seule



Quelques outils et principes

Type Abstrait de Données (TAD)

Définition :

Un Type de données Abstrait décrit **ce qu'un type peut faire**, sans dire **comment il le fait**. Cela revient à :

- Définir les opérations
- Définir les propriété formelles sur ces opérations(préconditions, post conditions et invariant).

On décrit le comportement attendu, pas la façon de l'implémenter.

Exemple: définition abstraite d'une tableau de double

Opérations

- creer(taille: entier) : tableau de double
- taille() : entier
- get(i: entier) : double
- set(i: entier, v: double)

Spécifications

- `t= creer(a)` a comme précondition $a \geq 0$ et comme postconditions: $t.taille() = a \wedge \forall i, (0 \leq i < a) \Rightarrow t.get(i) = 0$
- v= t.get(i) a comme préconditions $0 \leq i < t.taille()$
- t.set(i,v) a comme précondition $0 \leq i < t.taille()$ et comme postcondition $t.get(i) = v$

Pourquoi séparer l'abstraction et l'implémentation ?

- On peut **changer le code interne** sans casser ce qui l'utilise.
- Le code devient **plus stable et réutilisable**.
- En POO, on manipule les objets via leur **interface**, pas leur intérieur.

Une abstraction est comme un contrat : "Je te promets ce que je sais faire, pas comment je le fais."

Les principes SOLID

- **S** : Single Responsibility — une seule responsabilité
- **O** : Open/Closed — ouvert à l'extension, fermé à la modification
- **L** : Liskov Substitution — une sous-classe doit pouvoir remplacer sa super-classe
- **I** : Interface Segregation — petites interfaces ciblées
- **D** : Dependency Inversion — dépendre des abstractions, pas des implémentations

S — Single Responsibility Principle

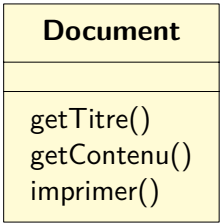
Definition

A class should have one, and only one, reason to change (Uncle Bob)

- Une classe doit avoir une seule responsabilité.
- Si elle gère plusieurs choses, c'est qu'il faut la découper.

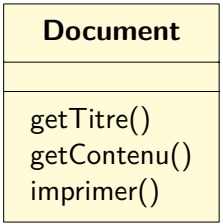
Single Responsability Principle

Exemple



Single Responsability Principle

Exemple

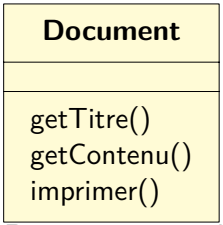


Deux responsabilités :

- Représenter un document ;
- Imprimer celui-ci ;

Single Responsability Principle

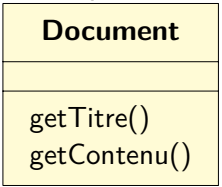
Exemple

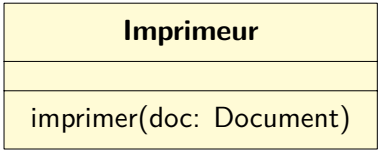


Deux responsabilités :

- Représenter un document ;
- Imprimer celui-ci ;

Devrait être plutôt :





Open/Closed Principle

Principe Ouvert/Fermé

Definition

- une classe doit être ouvert à l'extension: on doit pouvoir lui rajouter des fonctionnalités (par héritage, composition, modification du code...)

Open/Closed Principle

Principe Ouvert/Fermé

Definition

- une classe doit être ouvert à l'extension: on doit pouvoir lui rajouter des fonctionnalités (par héritage, composition, modification du code...)
- une classe doit être fermé à la modification de son *contrat* : le code extérieur qui utilise la classe doit *continuer à fonctionner sans modifications* même si on modifie la classe.

Open/Closed Principle

Principe Ouvert/Fermé

Definition

- une classe doit être ouvert à l'extension: on doit pouvoir lui rajouter des fonctionnalités (par héritage, composition, modification du code...)
- une classe doit être fermé à la modification de son *contrat* : le code extérieur qui utilise la classe doit *continuer à fonctionner sans modifications* même si on modifie la classe.

Donc :

- bien définir l'interface publique d'une classe ;
- ne pas laisser traîner de détails d'implémentation dans la partie publique ;
- **quand c'est pertinent,** prévoir des mécanismes d'extension.
  On ajoute, on ne casse pas.

L — Liskov Substitution Principle

- tout ce qu'on peut faire sur un objet de classe A, on peut le faire sur B ;
- une méthode de B qui étend une méthode de A :
  - ne doit pas imposer de restrictions supplémentaires sur ses arguments.
  - peut éventuellement s'imposer des limitations sur le type des données retournées (Notion de cours ?

L — Liskov Substitution Principle

- tout ce qu'on peut faire sur un objet de classe A, on peut le faire sur B ;
- une méthode de B qui étend une méthode de A :
  - ne doit pas imposer de restrictions supplémentaires sur ses arguments.
  - peut éventuellement s'imposer des limitations sur le type des données retournées (Notion de cours ? **(covariance)**).
- donc, si la classe Oiseau a une méthode voler(), on a un problème pour faire descendre Autruche de Oiseaux !

REVOIR COURS PRECEDENT

Interface Segregation Principle

Definition

Créer des interfaces qui sont spécifiques aux clients. Avec uniquement les méthodes nécessaires.

Interface Segregation Principle

Definition

Créer des interfaces qui sont spécifiques aux clients. Avec uniquement les méthodes nécessaires.

- Quand une classe A utilise un objet à travers une interface B, B ne devrait pas montrer à A de méthodes dont celui-ci n'a pas besoin.

Interface Segregation Principle

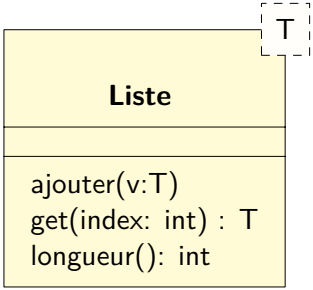
Definition

Créer des interfaces qui sont spécifiques aux clients. Avec uniquement les méthodes nécessaires.

- Quand une classe A utilise un objet à travers une interface B, B ne devrait pas montrer à A de méthodes dont celui-ci n'a pas besoin.
- Par exemple, si je dois simplement consulter un objet, mais pas le modifier, je n'ai pas besoin de voir ses setters.

Interface Segregation Principle

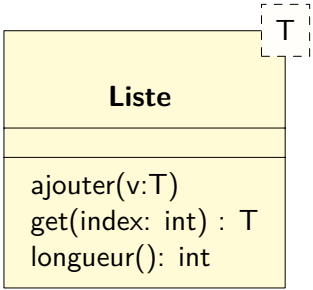
Exemple



En java, une liste non modifiable... est une liste qui lève une exception quand on lui ajoute un élément...

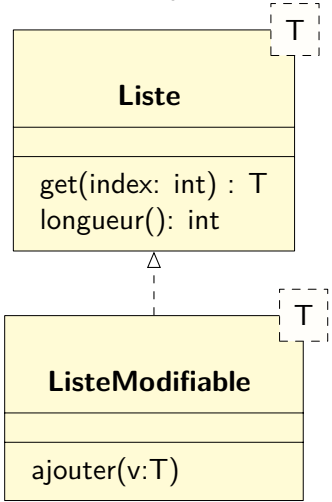
Interface Segregation Principle

Exemple



En java, une liste non modifiable... est une liste qui lève une exception quand on lui ajoute un élément...

Devrait être plutôt :



Dependency Inversion Principle

Definition

Une classe doit faire référence à des abstractions, non à leur implémentation.

Dependency Inversion Principle

Definition

Une classe doit faire référence à des abstractions, non à leur implémentation.

- Quand une classe A dit qu'elle a besoin d'une collection pour travailler, la façon dont la collection est implémentée ne nous intéresse pas ;

Dependency Inversion Principle

Definition

Une classe doit faire référence à des abstractions, non à leur implémentation.

- Quand une classe A dit qu'elle a besoin d'une collection pour travailler, la façon dont la collection est implémentée ne nous intéresse pas ;
- ce qui nous intéresse, c'est la collection des méthodes disponibles, pas la manière dont celles-ci sont écrites ;

Dependency Inversion Principle

Definition

Une classe doit faire référence à des abstractions, non à leur implémentation.

- Quand une classe A dit qu'elle a besoin d'une collection pour travailler, la façon dont la collection est implémentée ne nous intéresse pas ;
- ce qui nous intéresse, c'est la collection des méthodes disponibles, pas la manière dont celles-ci sont écrites ;
- boîtes noires ; interfaces plutôt que classes.

Dependency Inversion Principle

Definition

Une classe doit faire référence à des abstractions, non à leur implémentation.

- Quand une classe A dit qu'elle a besoin d'une collection pour travailler, la façon dont la collection est implémentée ne nous intéresse pas ;
- ce qui nous intéresse, c'est la collection des méthodes disponibles, pas la manière dont celles-ci sont écrites ;
- boîtes noires ; interfaces plutôt que classes.
- En suivant ce principe, une classe java ne devrait pas dépendre d'autres classes java, mais seulement d'**interfaces**.

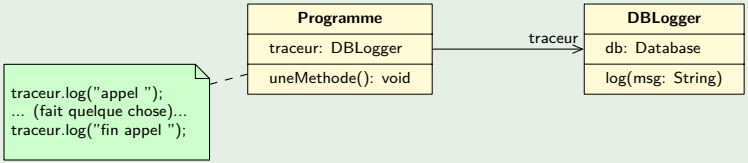
- En s'applique surtout entre différentes couches d'un logiciel ;
- Si `Facture` a besoin d'une `Adresse`, on ne va probablement pas créer une interface pour `Adresse`

Coder pour des interfaces, non pour des classes

dépendre des abstractions, non des implémentations

Exemple

dans la classe `Programme`, on souhaite enregistrer (logger) les actions réalisées par l'utilisateur d'un programme dans une base de données.
On a donc deux classes :

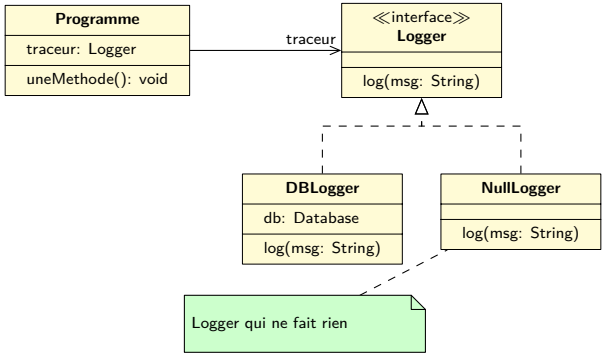


Problèmes

- pas pratique pour les tests, il faut une `Database` ;
- et si on veut envoyer les logs ailleurs ?

Coder pour des interfaces, non pour des classes

Solution: Programme ne connaît que l'interface de `DBLogger`...



- avantage : beaucoup plus souple et facile à tester ;
- la classe Programme *ne dépend plus de la classe DBLogger* !!!
- inconvénient : complexifie le code.

Délégation, découplage

Délégation

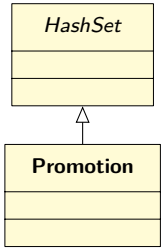
Definition

La délégation, c'est de faire faire le travail par quelqu'un d'autre.
Pour un objet, c'est faire réaliser une opération par un de ses composants.

Exemple (très basique): Une propotion d'étudiants

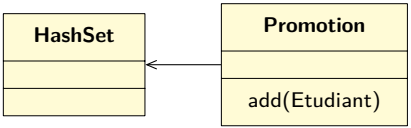
Heritage vs Délégation

Solution 1: Héritage



Promotion *est un* HashSet
Rapide mais rigide (Si je veux changer le hashset ?)

Solution 2 : Délégation



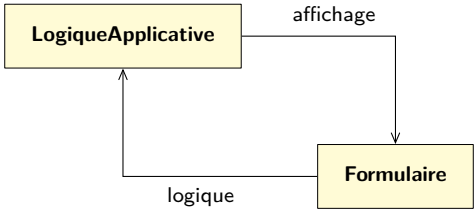
Promotion *contient un* HashSet
Plus souple et modulaire

Le couplage

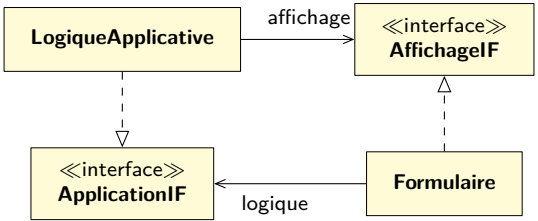
- Deux classes sont couplées quand elles dépendent l'une de l'autre.
- Trop de couplage = code difficile à tester et à maintenir.
- On réduit le couplage en :
  - introduisant des interfaces,
  - supprimant les dépendances circulaires.

Suppression *technique* du couplage

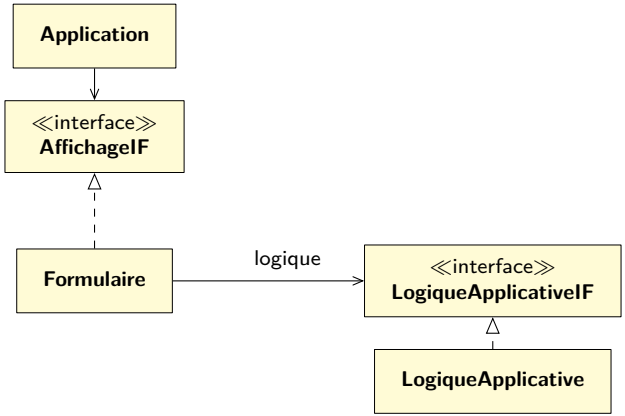
Avant:



Après :



Amélioration (probable) de l'architecture précédente



Récapitulatif général

- Une classe = une seule idée.
- Les classes coopèrent entre elles.
- On conçoit avant de coder.
- Favoriser la simplicité et la clarté.
- Appliquer SOLID avec bon sens.