# Cours 7 – Conception

MODULE INTRODUCTION AU GÉNIE LOGICIEL

MOSTEFAI MOHAMMED AMINE – BATATA SOFIANE

# Objectifs du Cours

Présenter l'activité de conception

Faire le lien entre la conception haut niveau et la conception bas niveau

Présenter les classes, les interfaces et les composants de conception

Souligner la différence entre les diagrammes comportementaux d'analyse et de conception

Découvrir les principes d'une bonne conception

# Plan du Cours

# Introduction

SECTION 1

# Définition

- Le modèle d'analyse définit les fonctionnalités (le ***quoi***) du système à développer
- La conception s'intéresse à ***comment*** ces fonctionnalités seront implémentées
- La conception se base sur le ***modèle de besoins*** et le modèle d'analyse
- Les solutions proposées par la conception repose sur le ***domaine métier*** et le ***domaine technique***

# Domaine métier et domaine technique

- Le domaine métier concerne les actions relatives au ***métier*** du logiciel
- Le domaine technique inclut les ***actions techniques*** à utiliser par la conception (base de données, techniques de persistance,...etc.)
- Par exemple, un logiciel de facturation. La validation de facturation fait partie du domaine métier. La bibliothèque permettant l'enregistrement des données dans une BDD fait partie du domaine technique.

# Produits de la conception

Composants (sous-systèmes)

Classes

Interfaces

Diagrammes comportementaux

Diagrammes de déploiement

# Conception et analyse

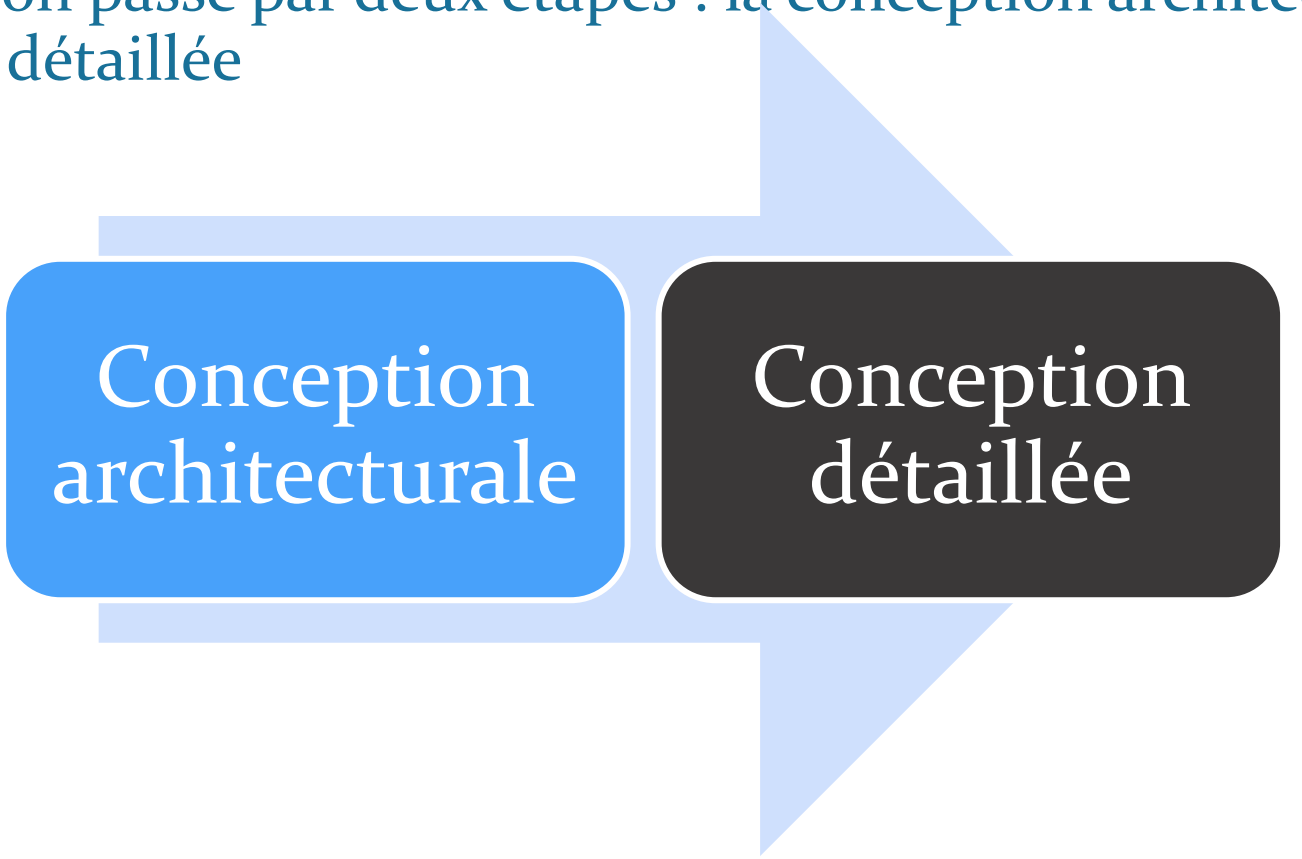
| Elément | Analyse | Conception |
|---|---|---|
| Sous-systèmes | - | Composants définissant la composition du système |
| Classes | La classe d'analyse représentent des concepts métier. Niveau bas de détail (signature de méthodes, types, ...etc.) | Les classes de conception sont plus détaillées et précises. Elles préparent le terrain aux développeur pour l'implémentation. En plus des concepts métier, elles contiennent les classes du domaine technique. |

# Conception et analyse

| Elément | Analyse | Conception |
|---|---|---|
| Interface | - | Représente une « façade » du système indépendante de l'implémentation |
| Diagrammes comportementaux | Implique les instances de classes d'analyse, orientées métier | Implique toutes les classes, orientées technique |
| Diagrammes de composants et déploiement | - | Structure et architecture du système |

# Processus

- La conception passe par deux étapes : la conception architecturale et la conception détaillée



# Processus – Suite

- Activité assurée par les architectes
- La conception architecturale concerne une ***vision descriptive*** des éléments de la conception : sous-systèmes (composants), classes et interfaces
- La conception détaillée a pour but de ***donner les détails précis*** des éléments produits dans la conception architecturale et préparer au mieux l'implémentation

# Caractéristiques d'une bonne conception

## Répond avec précision aux besoins

- La conception doit permettre de fournir une solution technique ***répondant aux attentes du client***. Ne pas faire plus que ce qu'attend le client.

## Séparation des fonctionnalités SoC (Separation of Concerns)

- Une bonne conception engendre un haut niveau de ***modularité*** où chaque module a des fonctionnalités précises et indépendantes. De tels systèmes sont plus simples et plus faciles à maintenir

## Simplicité

- Quand plusieurs solutions s'offrent pour la résolution du même problème, choisir la plus simple. Plus une conception est simple, plus il est facile de la comprendre et de la maintenir.

## Facilité de maintenance

- Une bonne conception permet ***d'isoler*** rapidement les erreurs et des les ***réparer*** efficacement

# Introduction

SECTION 1, DÉBAT 05 MNS

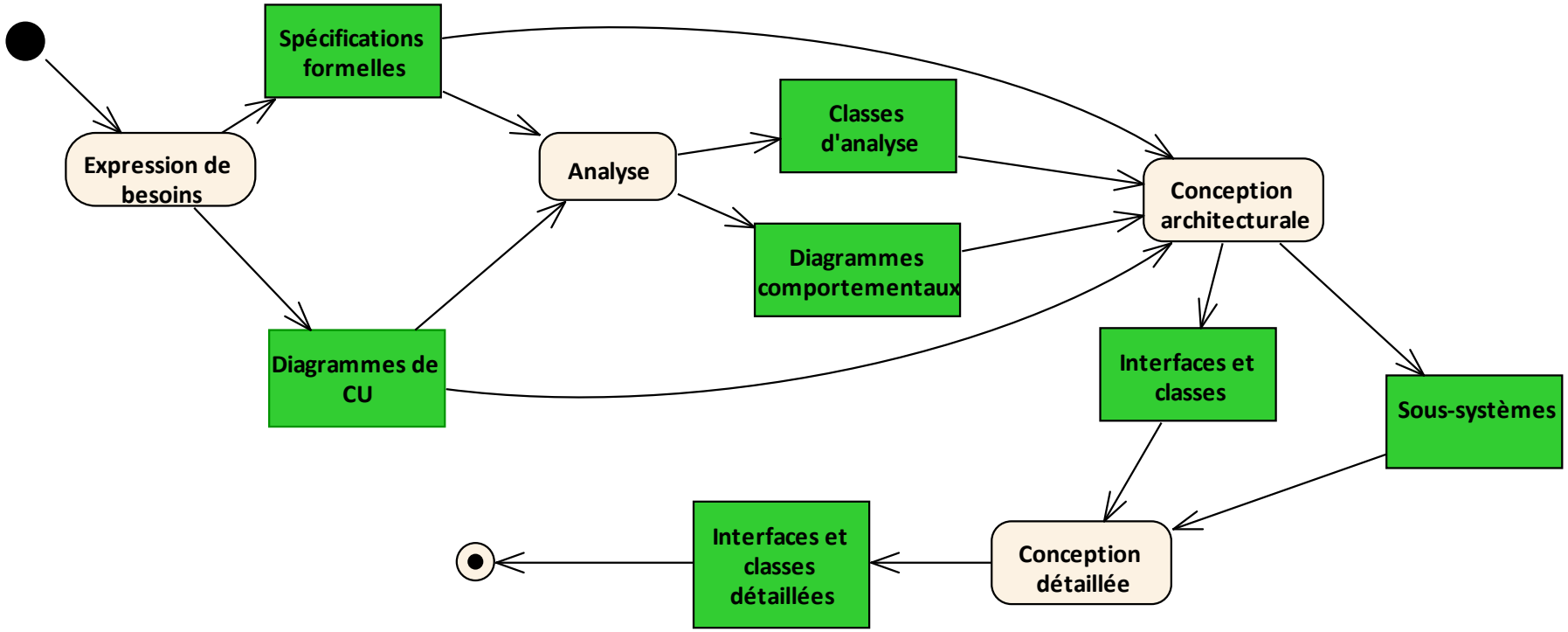
# Les Classes de Conception

SECTION 2

# Classe de conception

- La classe est l'élément principal de la ***conception orientée objet***
- La conception architecturale produit des classes et des interfaces sans trop aller dans le détail
- La conception détaillée finalise le contenu des classes et des interfaces (***classes détaillées***)
- Une classe détaillée est le commencement de l'opération ***d'implémentation*** (codage)

# Classe de conception - Suite



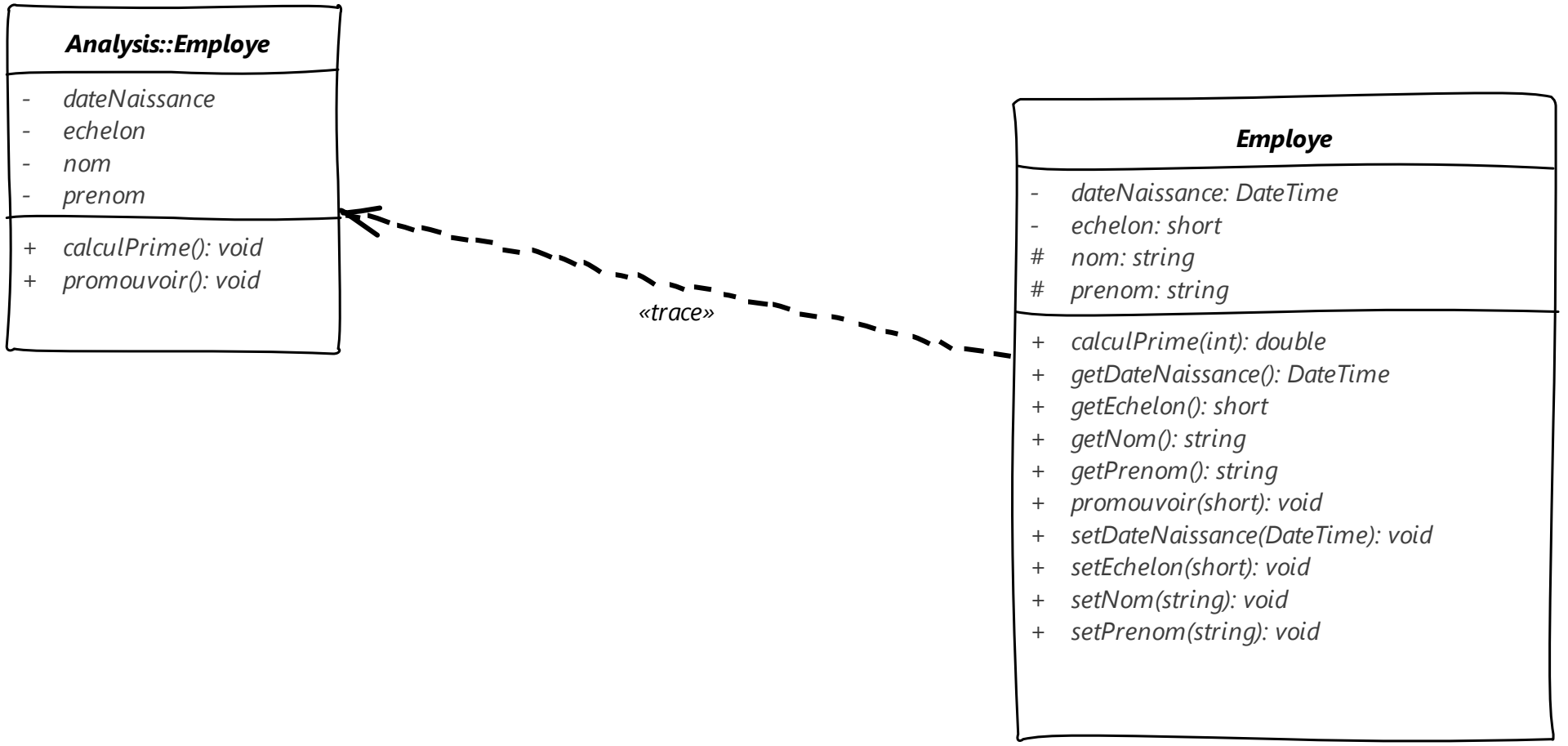
# Provenance des classes de conception

- Les classes de conception peuvent provenir de deux domaines : le ***domaine métier*** et le ***domaine technique***
- Les classes du domaine métier sont une évolution des ***classes d'analyse***
- Les classes du domaine technique font partie des ***bibliothèques*** ou ***frameworks*** faisant partie de la solution technique (bibliothèques, GUI, …), par exemple (Spring, Hibernate, Nhibernate, …)

# Caractéristiques d'une classe de conception

- Une classe d'analyse définit un ***aperçu*** de la classe sans s'occuper des détails techniques
- La classe de conception (domaine métier) est une ***évolution*** de la classe d'analyse en tenant en compte tous les détails (visibilité, paramètres, ..)
- La classe de conception contient la ***liste complète*** des attributs et des opérations que devrait avoir la classe

# Analyse vers Conception : Exemple



# Qu'est-ce qu'une bonne classe de conception ?

***Complétude***

- la classe doit fournir tout ce que l'on attend d'elle

***Suffisance***

- la classe doit fournir ce qu'on attend d'elle et pas plus

***Primitivité***

- Une classe ne doit pas fournir une multitude de façons de réaliser le même but

***Haute cohésion***

- une classe doit avoir un objectif sémantique unique. Les opérations et les attributs de cette classe doivent être relatifs à cet objectif.

***Faible couplage***

- la classe doit être liée au minimum de classes possibles

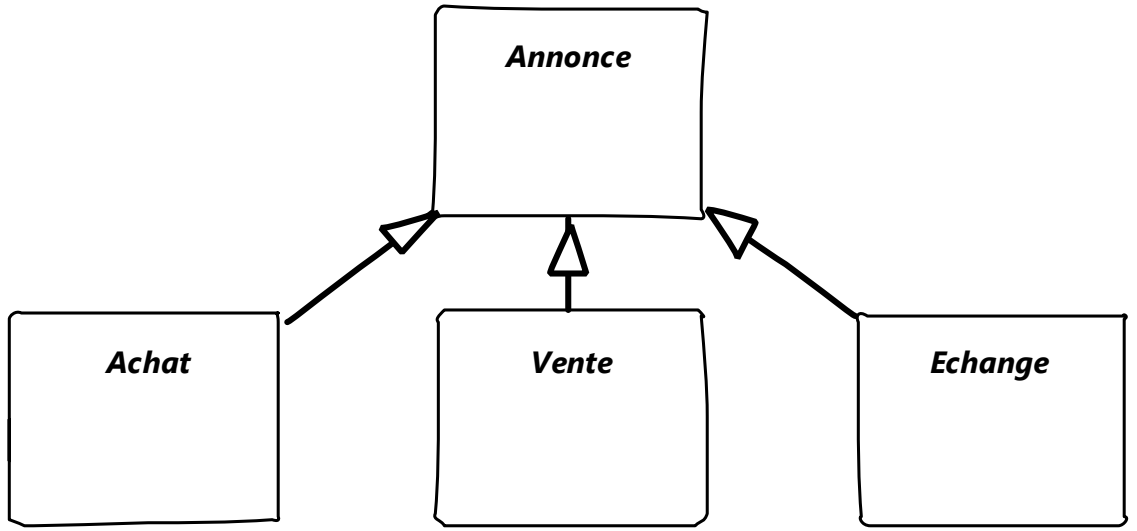
# L'Héritage

L'héritage est un excellent moyen de définir des relations entre les classes mais les développeurs ***ont tendance à en abuser***. L'héritage a les inconvénients suivants :

- L'héritage impose un ***très grand couplage*** entre la classe parent et la classe enfant
- L'encapsulation est ***très faible*** entre les classes appartenant à la même hiérarchie
- La hiérarchie est ***inflexible***, elle ne peut pas changer pendant l'exécution
- Une bonne conception doit bien définir les relations de hiérarchie

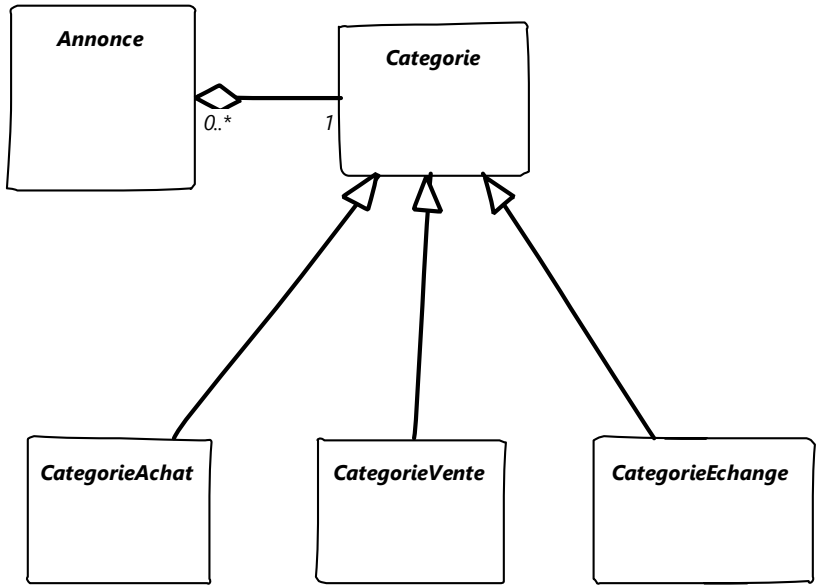
# Héritage versus agrégation

Soit un site d'annonce, le site est composé d'annonces de ventes, d'achat ou d'échange.




Problème : si on veut transformer une annonce d'échange en vente ?

# Héritage versus agrégation



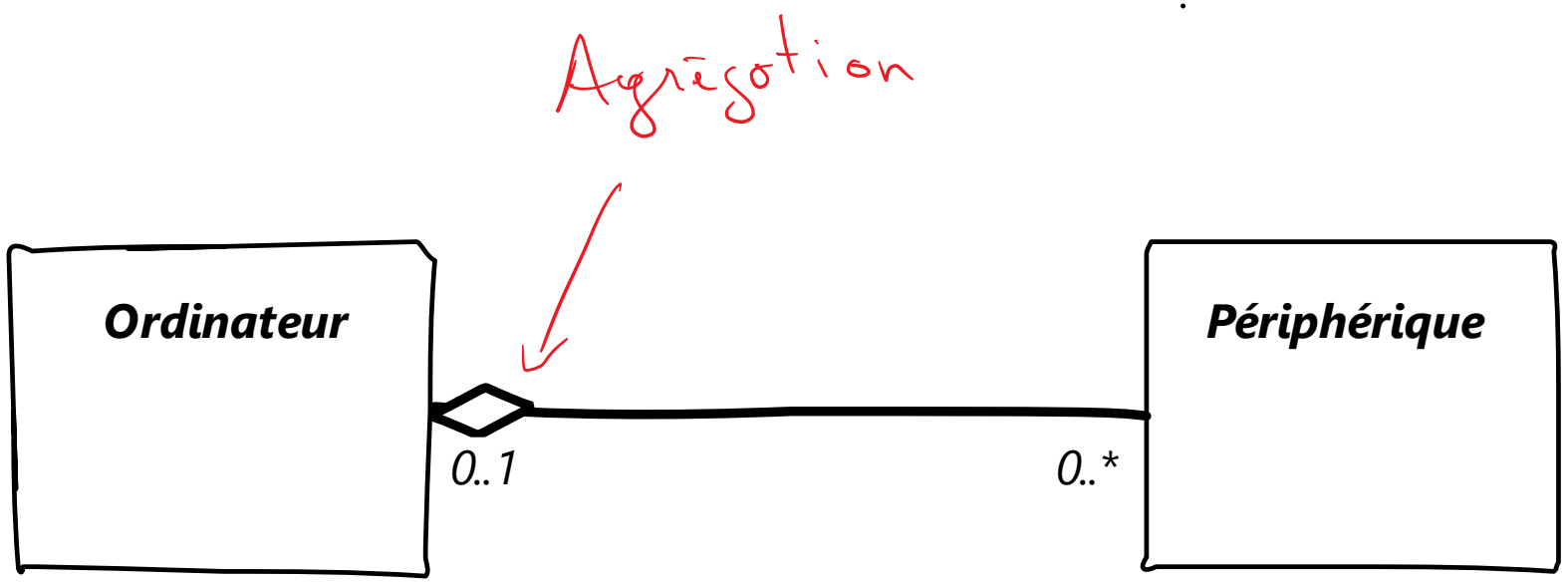
# Agrégation et composition

- Pour les classes du domaine métier, les associations entre les classes d'analyse peuvent évoluer en ***agrégations*** et ***compositions***
- Une agrégation représente une relation ***faible*** entre des objets
- Une composition représente une relation ***forte*** entre des objets

# Agrégation

- Une agrégation est une relation entre une ***classe d'ensemble*** et des ***parts***
- La classe d'ensemble ***contrôle*** les parts
- La part expose ses ***service*** à la classe d'ensemble
- La part peut ***ne pas être au courant*** de son appartenance à une classe d'ensemble
- Une part peut exister ***en dehors*** de la classe d'ensemble
- Une part peut faire partie de ***plusieurs*** classes d'ensemble

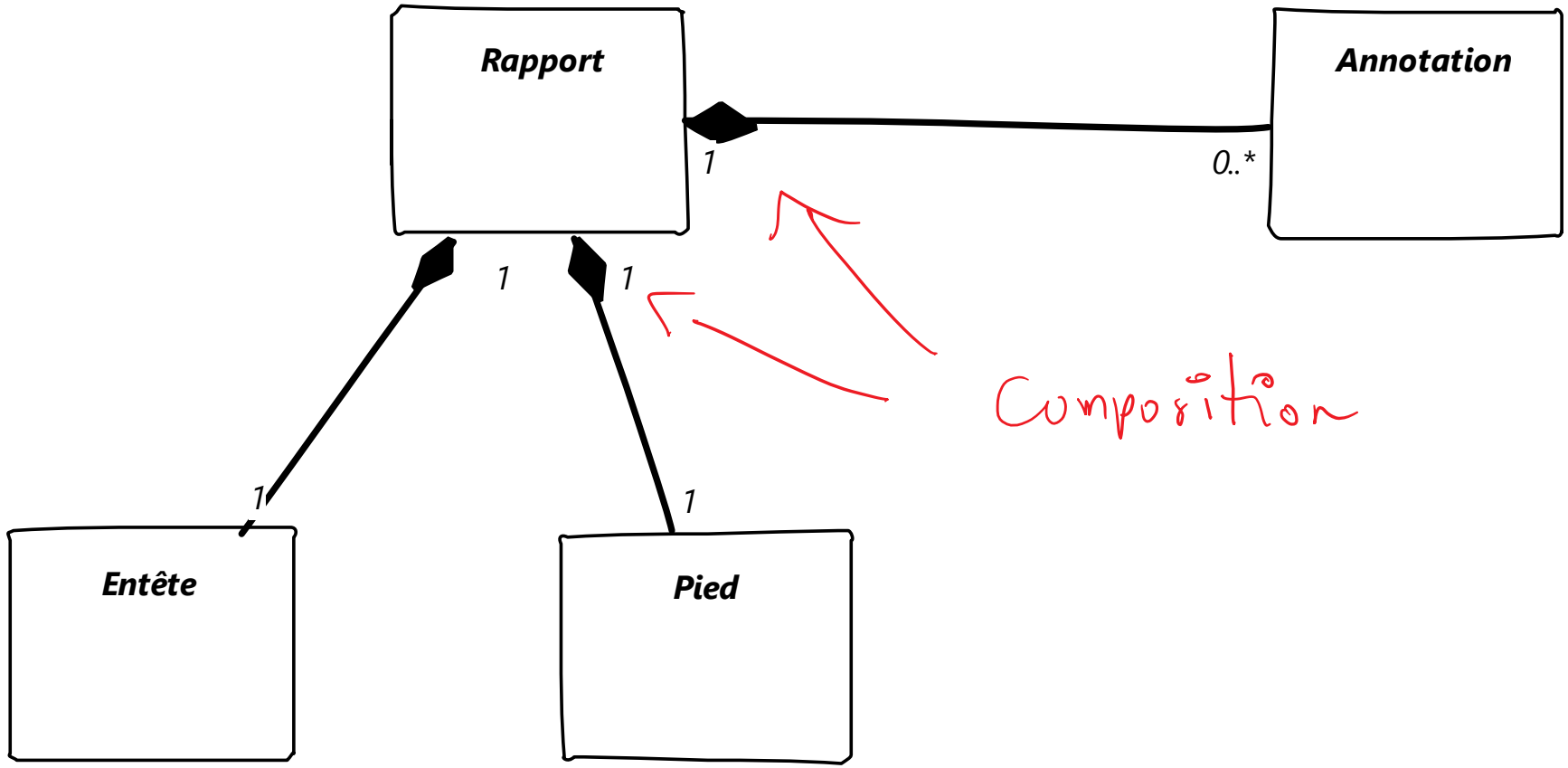
# Agrégation



# Composition

- La composition est similaire à l'agrégation sauf que la composition est sémantiquement ***plus forte***
- Une part ne peut pas exister ***en dehors*** de la classe d'ensemble
- Une part ne peut appartenir qu'à ***une seule*** classe d'ensemble
- La destruction de la ***classe d'ensemble*** conduit à la destruction de ses parts

# Composition



# Les Classes de Conception

SECTION 2, DÉBAT 05 MNS

# Interfaces et Composants

SECTION 3

# Introduction

- La conception passe par ***diviser*** un système en sous-systèmes représentés par des ***composants***
- L'interaction et la médiation entre ces sous-systèmes se fait en utilisant les interfaces

# Qu'est-ce qu'une interface ?

- Une interface est ***un ensemble d'opérations*** liées sémantiquement qui permettent de séparer la spécification de l'implémentation
- Une interface ***ne contient pas d'implémentation***. Les classes se chargent d'effectuer cette implémentation. Ce lien s'appelle ***réalisation***.
- Les interfaces sont supportées par Java et C#. En C++ le concept d'interface équivaut à une classe abstraite.
- Par convention, les interfaces commencent par I et utilisent un objectif (par exemple, IComparable)

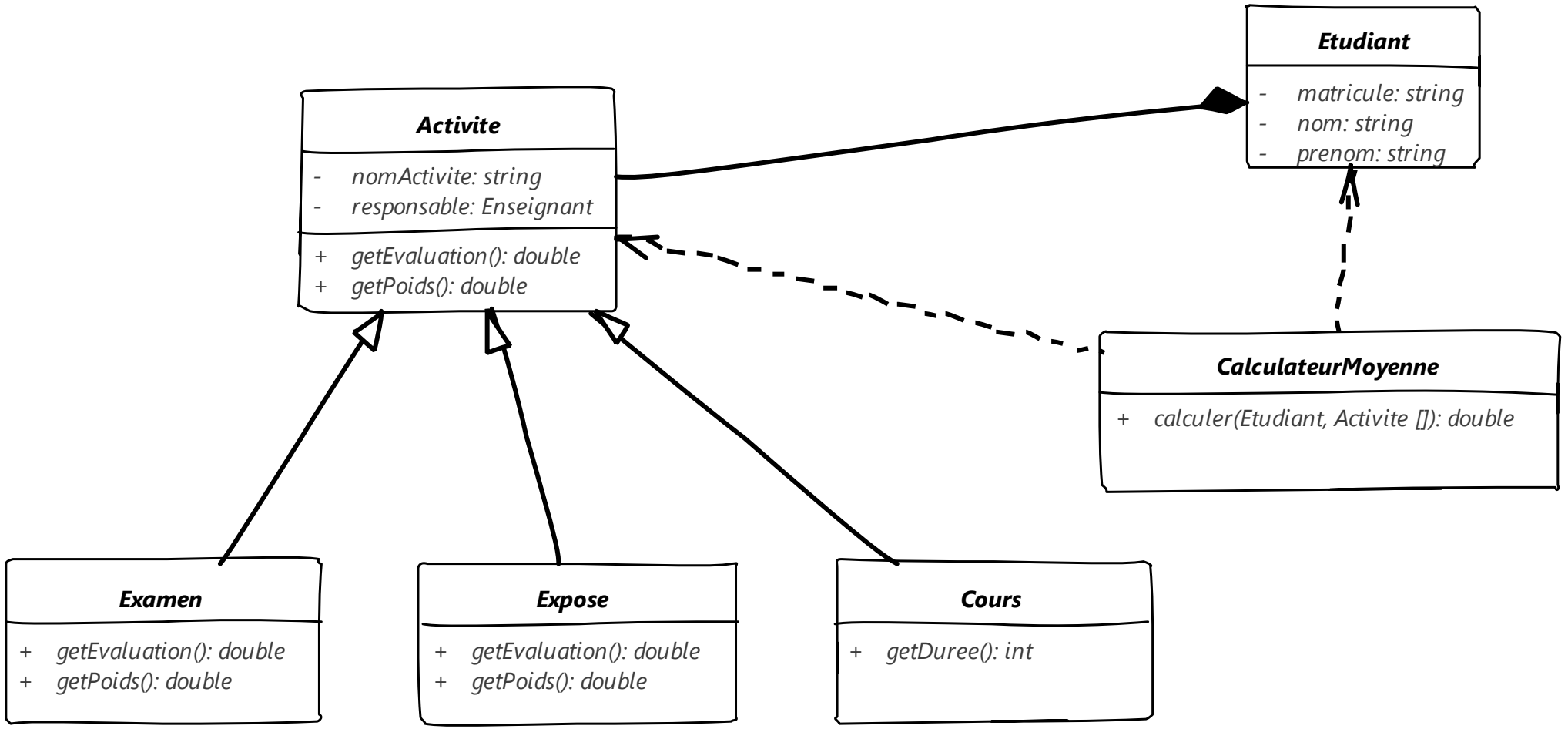
# Interface fournie et interface requise

- Une interface fournie représente les fonctionnalités ***assurées*** par une classe ou par un sous-système
- Une interface requise représente les fonctionnalités ***nécessaires*** à une classe
- L'assemblage de deux composants sur la base d'une interface fournie / requise est appelé ***contrat***

# Exemple

Imaginons une gestion de scolarité où l'élève suit un certain nombre d'activités. Certaines de ces activités sont évaluables comme les examens et les exposés et d'autres ne le sont pas comme les cours.

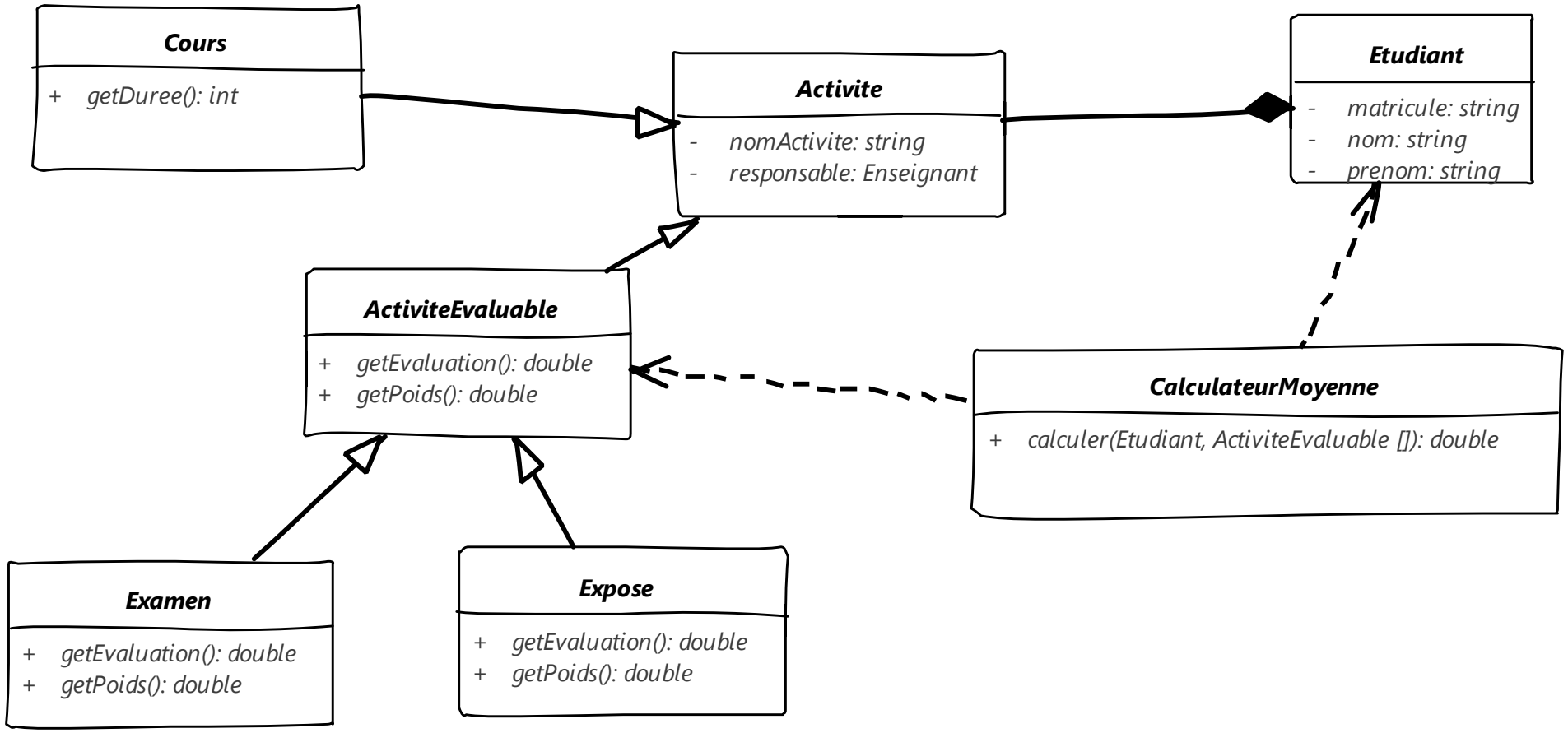
# 1ère solution



# 1ère solution

Conceptuellement, la 1ère solution n'est pas bonne. Les classes descendantes doivent redéfinir le mécanisme d'évaluation, chose qui n'est pas applicable sur les cours.

# 2ème solution



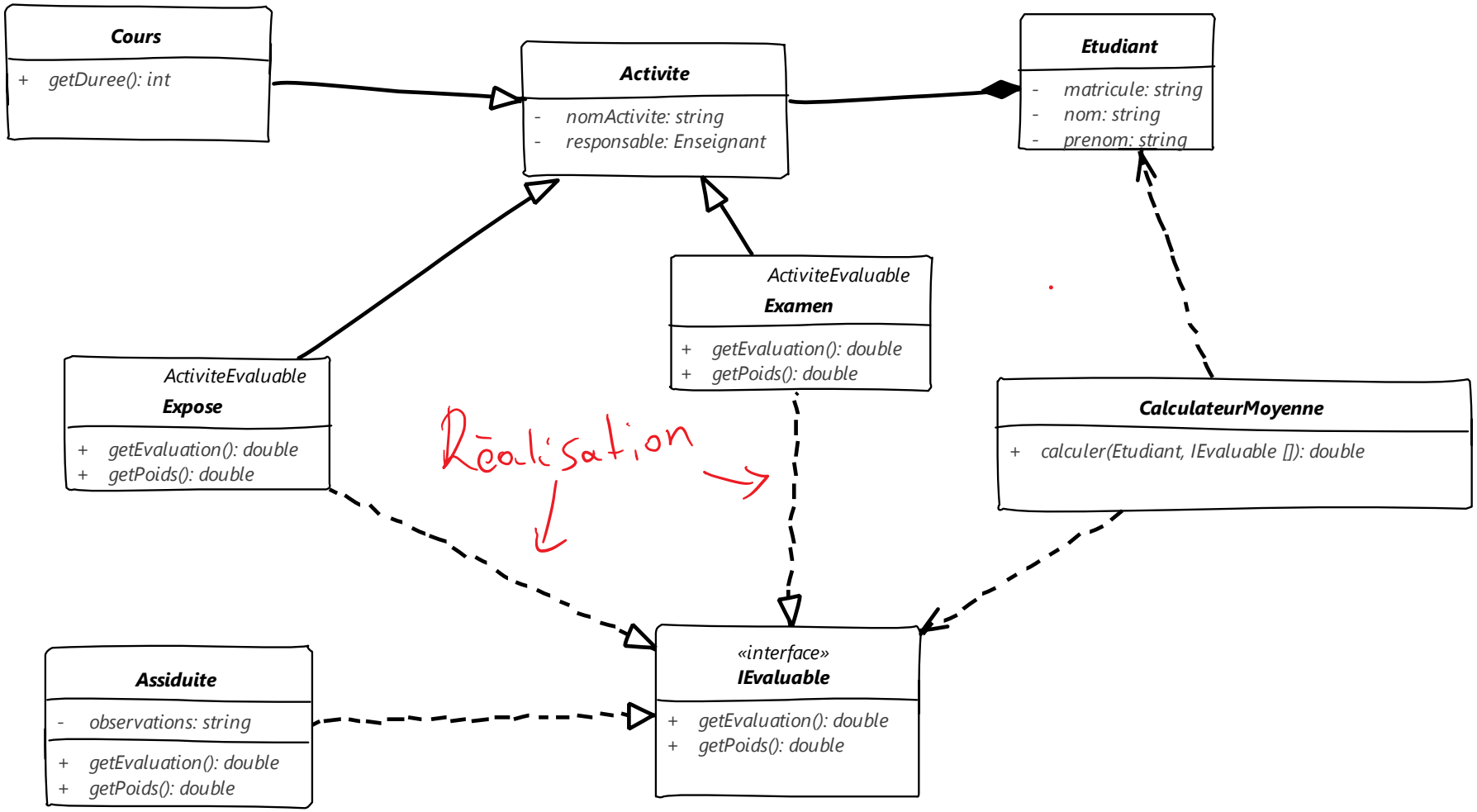
# 2ème solution

La 2ème solution est meilleure que la 1ère. Supposons qu'on veuille intégrer l'assiduité de l'étudiant.

Problème 1 : l'assiduité n'est pas une activité pédagogique sémantiquement.

Problème 2 : Si la classe « Assiduité » hérite d'activité pédagogique, elle héritera de toutes les propriétés (par exemple « Responsable ») ce qui n'aura pas de sens.

# Solution 3

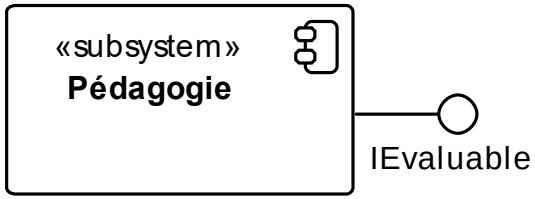


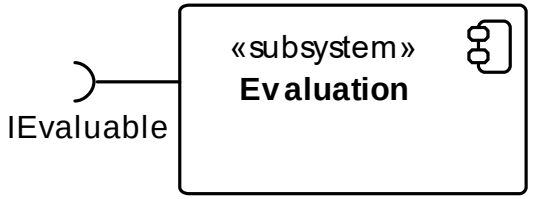
# 3ème solution

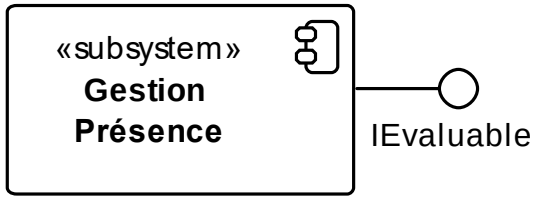
La 3ème solution résout les problèmes précédents en ajoutant un très grand degré d'abstraction.

En plus elle facilite la conception du système global, le sous-système d'évaluation a pour interface requise « IEvaluable » et d'autres sous-systèmes comme la gestion des examens, ont cette interface comme fournie.

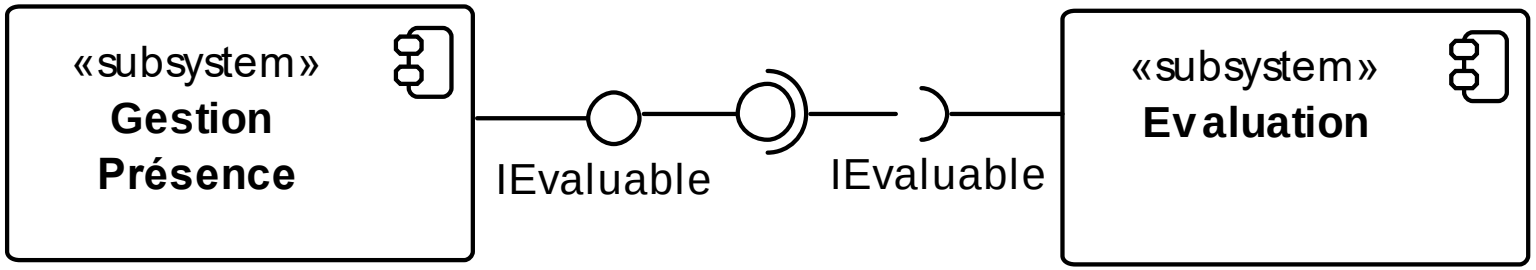
# Sous-systèmes



# Assemblage



# Trouver les interfaces

Examiner chaque association, vérifier si une association doit être figée ou flexible

Examiner les opérations ayant un lien sémantique et voir s'il est bénéfique de les factoriser en une interface

Trouver les opérations qui se trouvent dans plusieurs classes et dont l'héritage ne convient pas pour les faire lier

Trouver les classes qui jouent le même rôle dans le système. Ce rôle peut être factorisé en une interface.

Réfléchir en terme d'extensibilité, plus on utilise les interfaces, plus on est extensible

Identifier les interconnexions entre les composants et les sous-systèmes

# Concevoir avec les interfaces

Le patron de conception « Façade » permet de montrer d'un sous-système que la partie intéressante à utiliser

L'interface est un élément parfait pour implémenter la façade

En plus, l'interface facilite la conception du système sous forme de couches

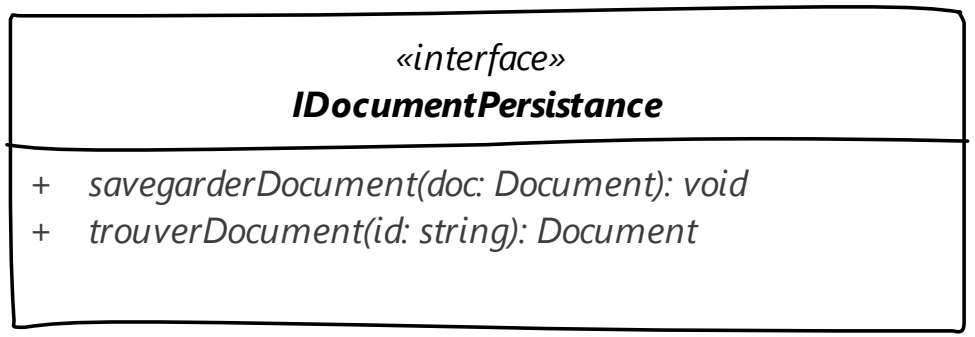
# Exemple

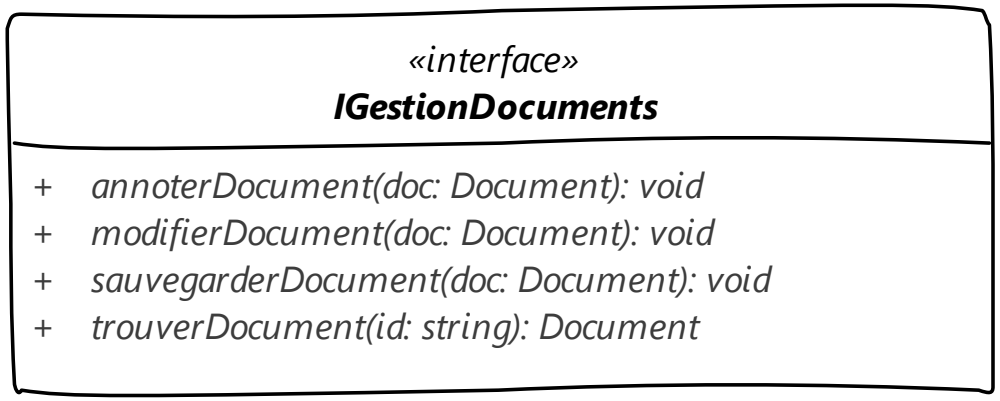
- Soit un système de gestion de documents composé de trois modules : persistance, métier et GUI
- La partie persistance permet d'enregistrer un document. Ce module enregistre le document dans plusieurs entrepôts. Pour récupérer un document, le module cherche la version la plus récente dans les entrepôts et la renvoie.
- La partie métier concerne les opérations sur les documents telles que la modification ou l'annotation.
- La partie GUI concerne l'affichage des informations.

# Exemple - Suite

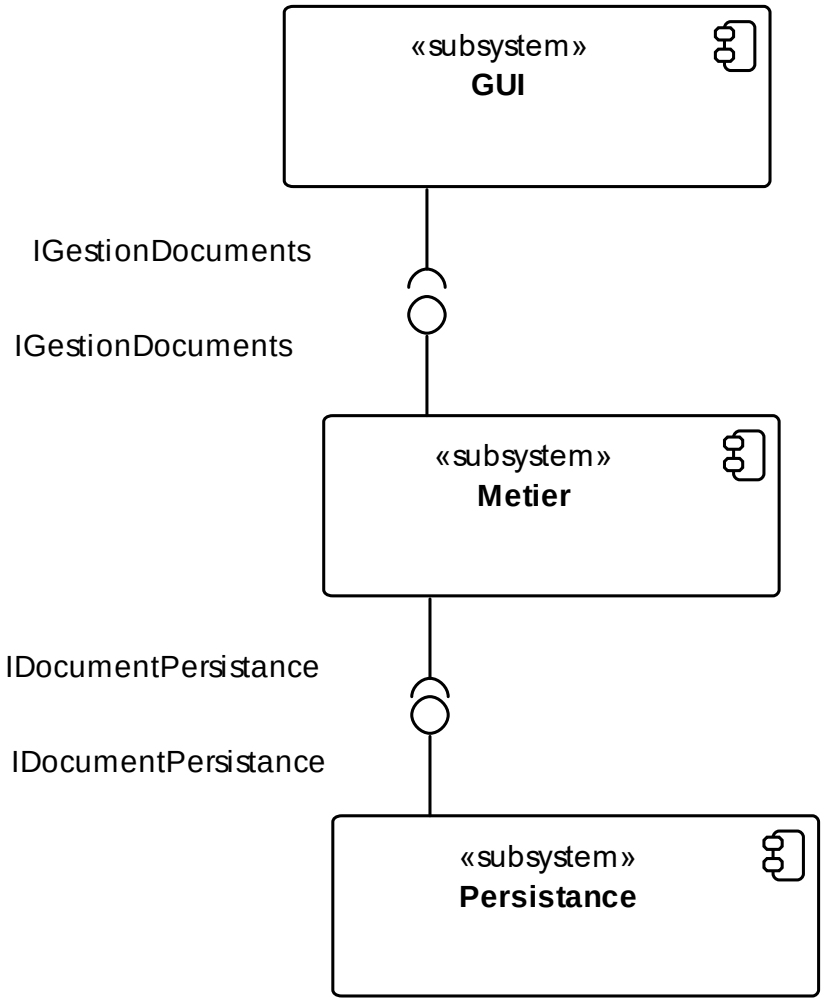
- Le patron de conception « Façade » permet de cacher tous les détails inutiles à l'extérieur. Quel que soit le mécanisme de persistance, on doit être capable de sauvegarder un document ou de trouver un document.

# Exemple - Suite



# Exemple - Suite



# Interfaces et Composants

SECTION 3

# Les Diagrammes Comportementaux

SECTION 4

# Utilisation des Diagrammes Dynamiques

- Pour modéliser certains aspects dynamiques, la conception utilise les mêmes diagrammes que l'analyse (voir cour 5)
- La différences entre les deux est que les diagrammes de conception ont 1) plus d'éléments (classes métier + technique) et 2) adressent éventuellement un aspect technique métier (par exemple séquencement de processus, traitement parallèle,…etc)
- Parmi les livrables de la conception, figurent aussi des diagrammes comportementaux quand c'est nécessaire

# Les Diagrammes Comportementaux

SECTION 4, DÉBAT 05 MNS

# Principes de Conception

SECTION 5

# Principes

Isoler les parties changeantes

SRP (Single Responsibility Principle)

DIP (The Dependency Inversion Principle)

OCP (Open Closed Principle)

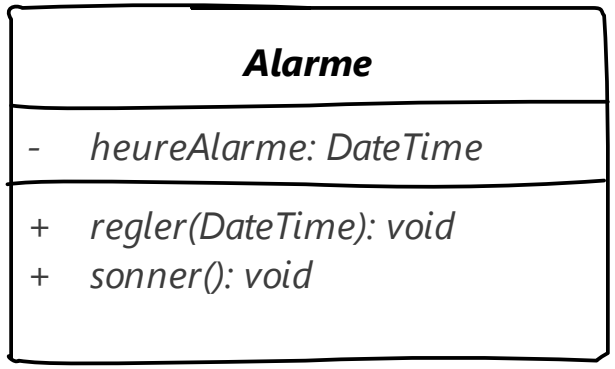
DRY (Don't Repeat Yourself)

LSP (LiskPov Substitution Principle)

ISP (Interface Segregation Principle)

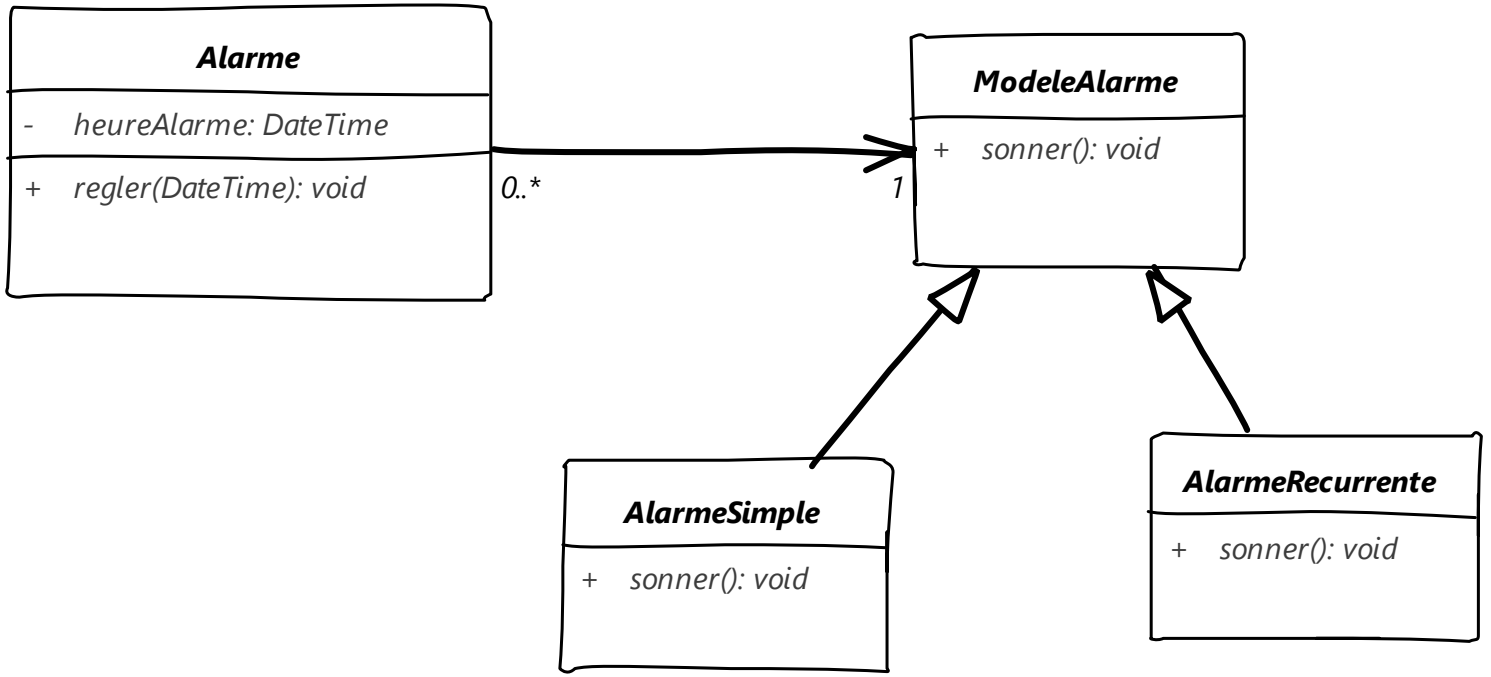
# Isoler les parties changeantes

- Essayer de changer les parties changeantes dans d'autres classes
- Par exemple, dans la classe Alarme, la méthode « Régler » est figée mais « Sonner » dépend du type d'alarme. Cette méthode est aussi appelée à évoluer si d'autres types sont créés



# Isoler les parties changeantes

- La conception ci-dessous permet d'isoler les changements et en prévoir d'autres



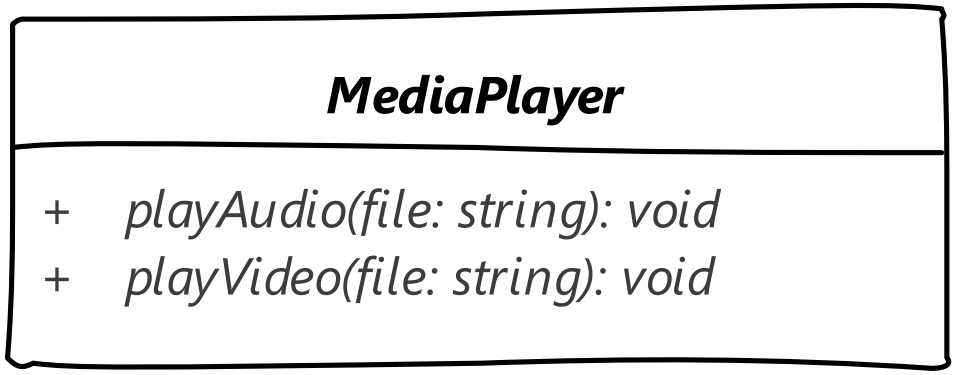
# Single Responsibility Principle

- Single Responsibility Principle = Principe de fonctionnalité unique
- Une classe doit avoir une raison ***unique*** de changer
- Une classe ne doit pas faire « trop » de choses

# Single Responsibility Principle

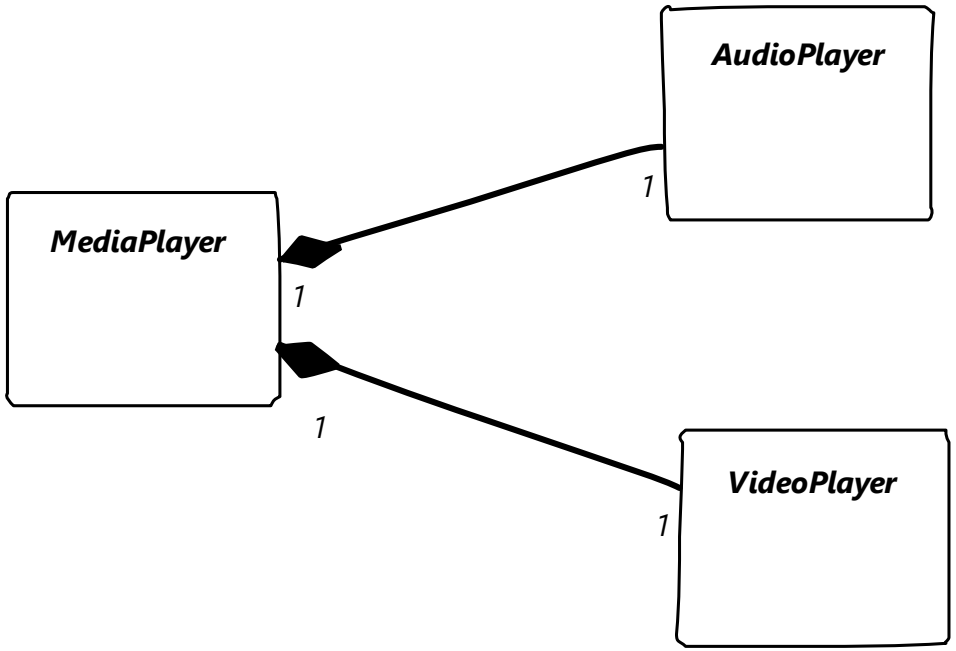
# SRP- Exemple

- La classe MediaPlayer ne respecte pas le SRP car le changement de l'algorithme d'audio ou de vidéo affecteraient la classe



# SRP- Exemple

- Un premier pas consiste à isoler la vidéo et l'audio



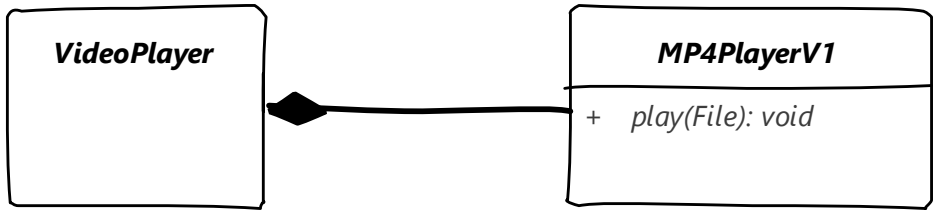
# Dependency Inversion Principle

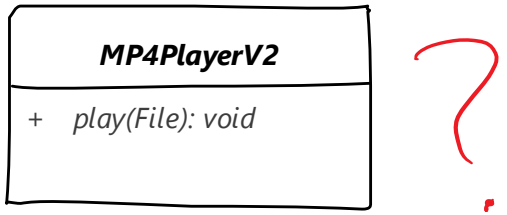
- Dependency Inversion Principle = Principe d'inversion de dépandance
- Les modules ***ne doivent pas*** dépendre des modules de bas niveau. Les deux doivent dépendre d'abstraction
- Les abstraction ***ne doivent pas*** dépendre des détails, c'est les détails qui doivent dépendre des abstractions

# DIP - Exemple

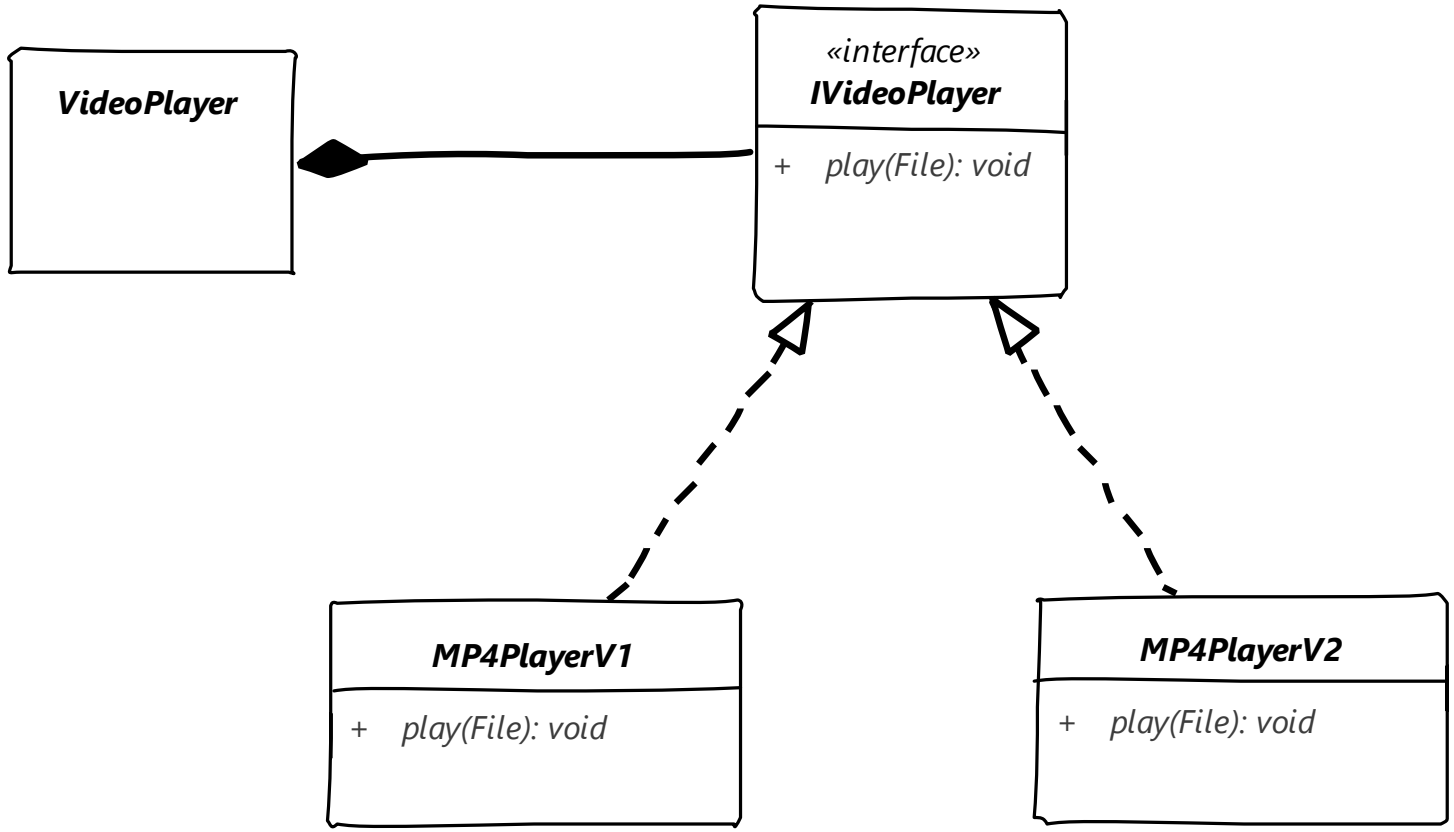
La classe Video dépend de PM4PlayerV1.

- La conception ne respecte pas DI. Chaque changement de l'algorithme MP3 va impacter VideoPlayer
- Par conséquent, très difficile de faire évoluer



# DIP - Exemple



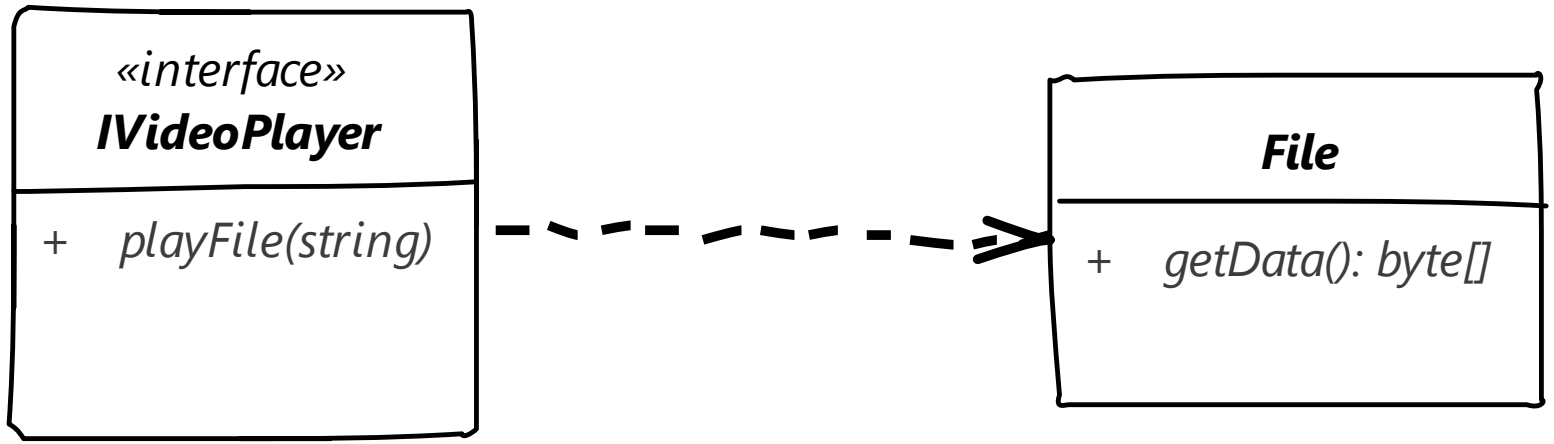
# OCP

- OCP : Open-Closed Principle (Principe Ouvert-Fermé)
- Une classe doit être ***ouverte*** à l'extension et ***fermée*** à la modification
- Trouver les comportements appelés à évoluer et les transformer en abstraction

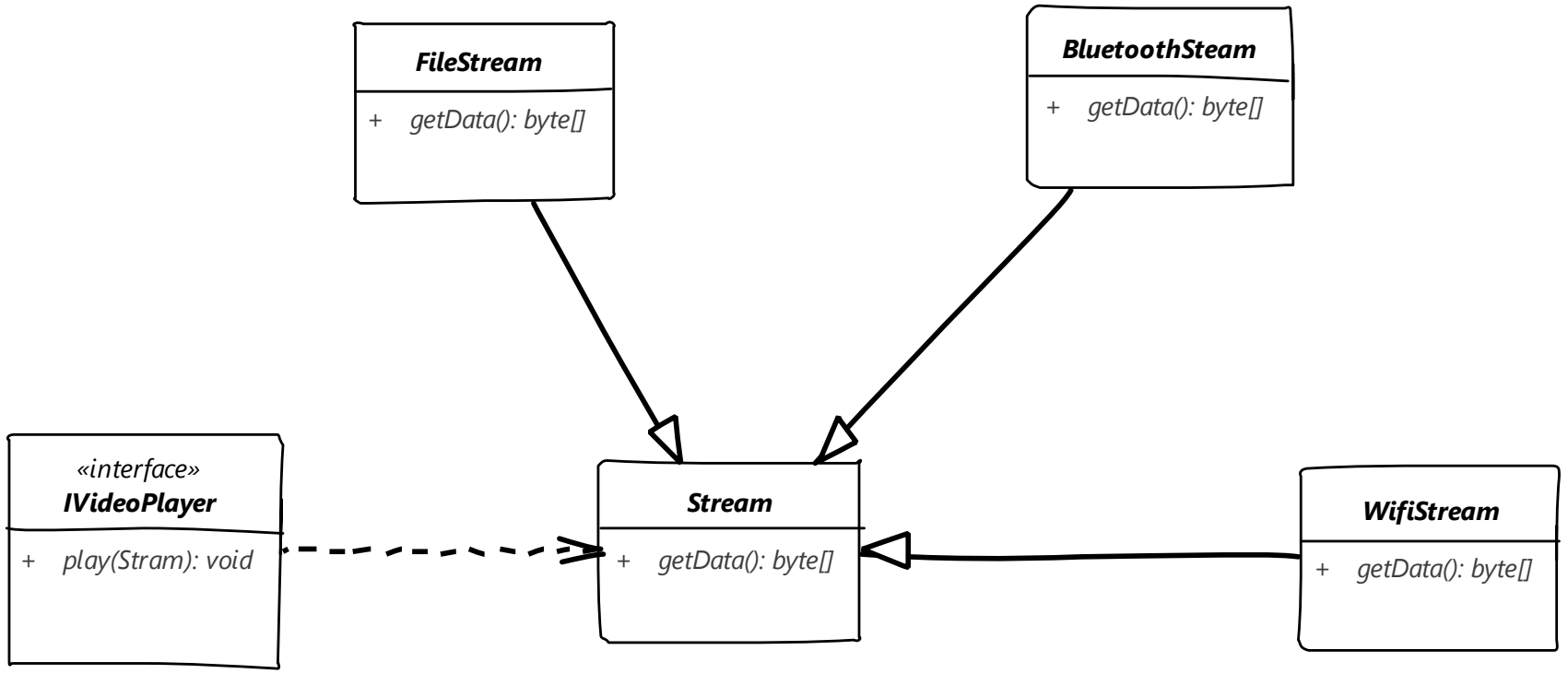
# OCP - Exemple

- L'interface IVideoPlayer pour lire des fichiers audio, utilise la classe File qui renvoie les données sous format d'octets (getData).
- La classe AudioPlayer ne respecte pas OCP car elle se restreint au données provenant de fichiers. Si on veut lire des données depuis internet, streaming, wifi ou bluetooth il faut changer l'interface(ouverte à la modification)

# OCP - Exemple



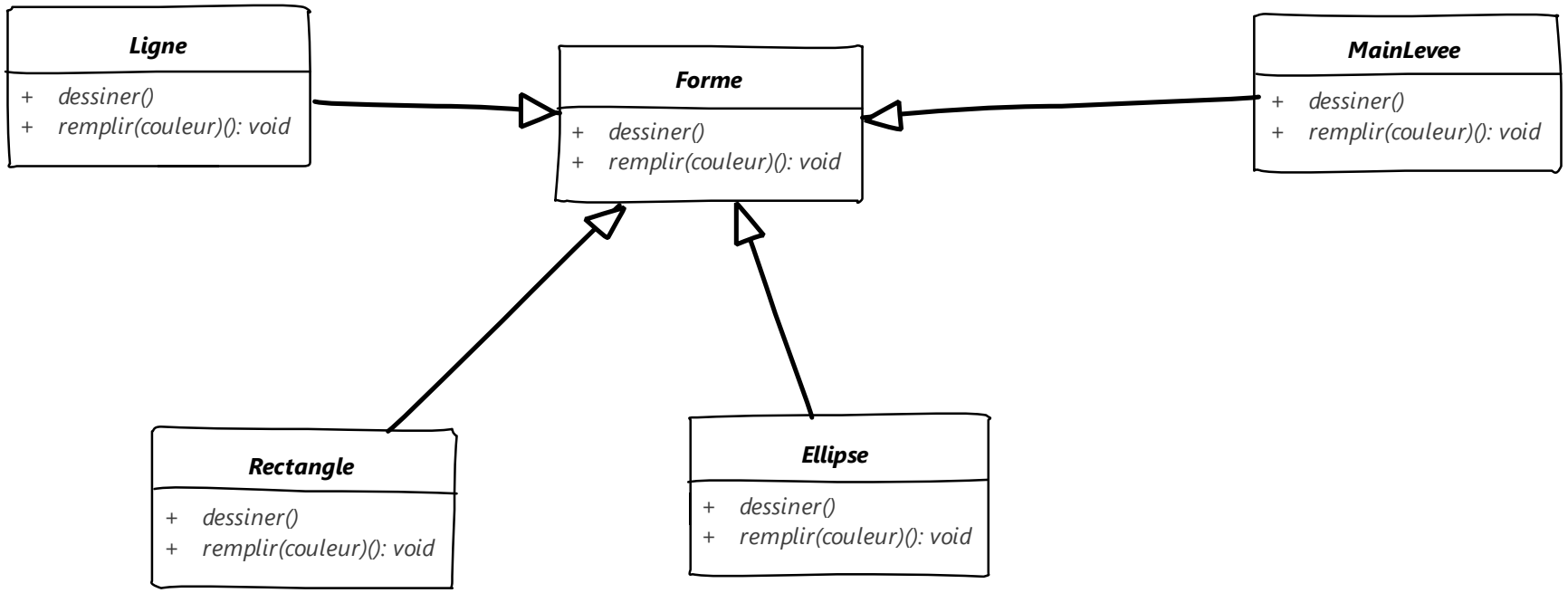
# OCP - Exemple



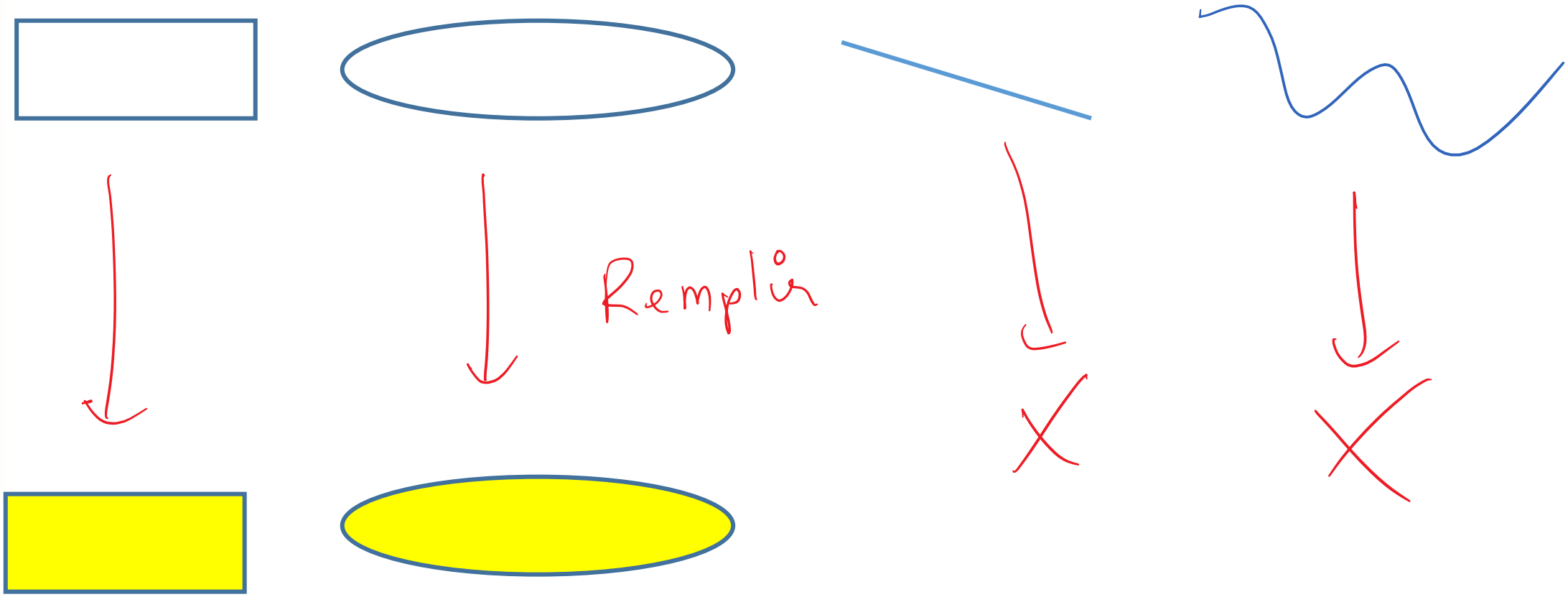
# LSP – Liskov Substitution Principle

- L'héritage est une solution à OCP mais comment réguler l'héritage ?
- Principe : Les sous-types doivent être des substituts pour leur ancêtres

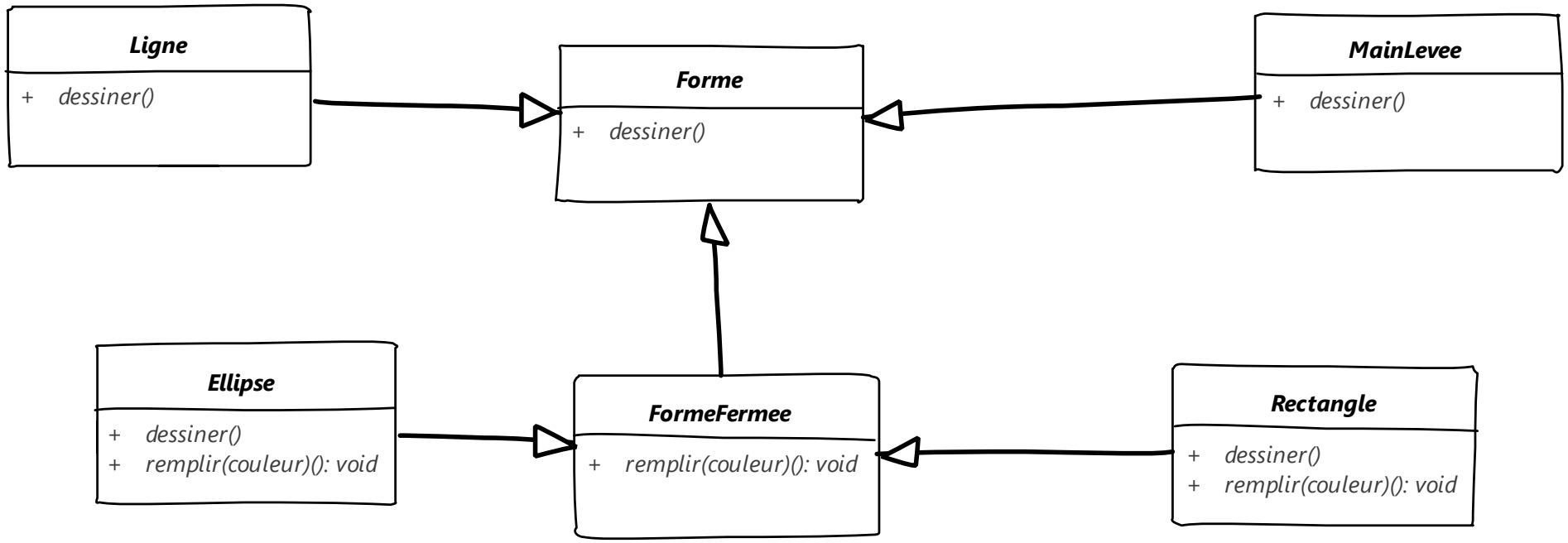
# LSP – Exemple



# LSP – Exemple



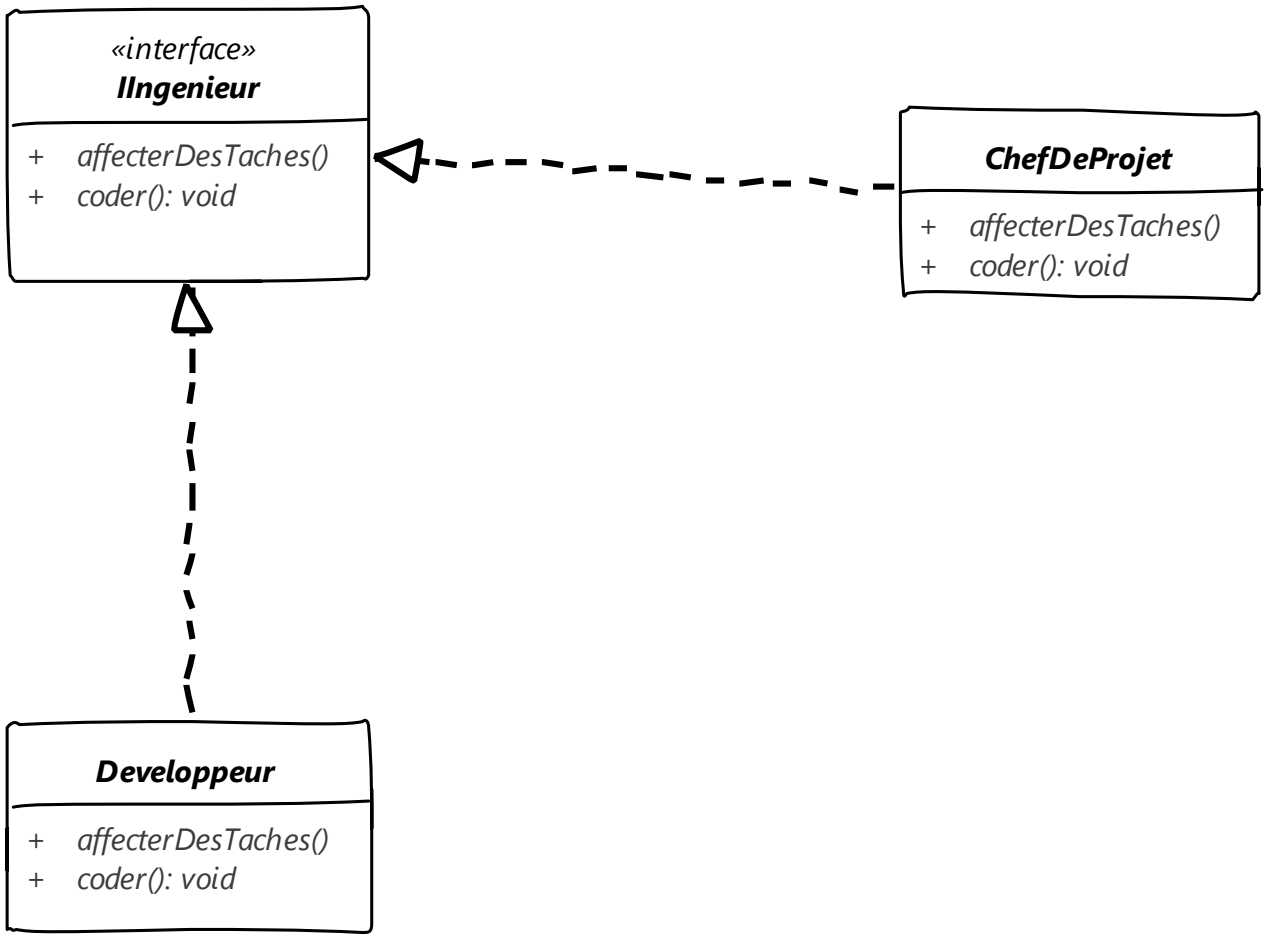
# LSP – Solution



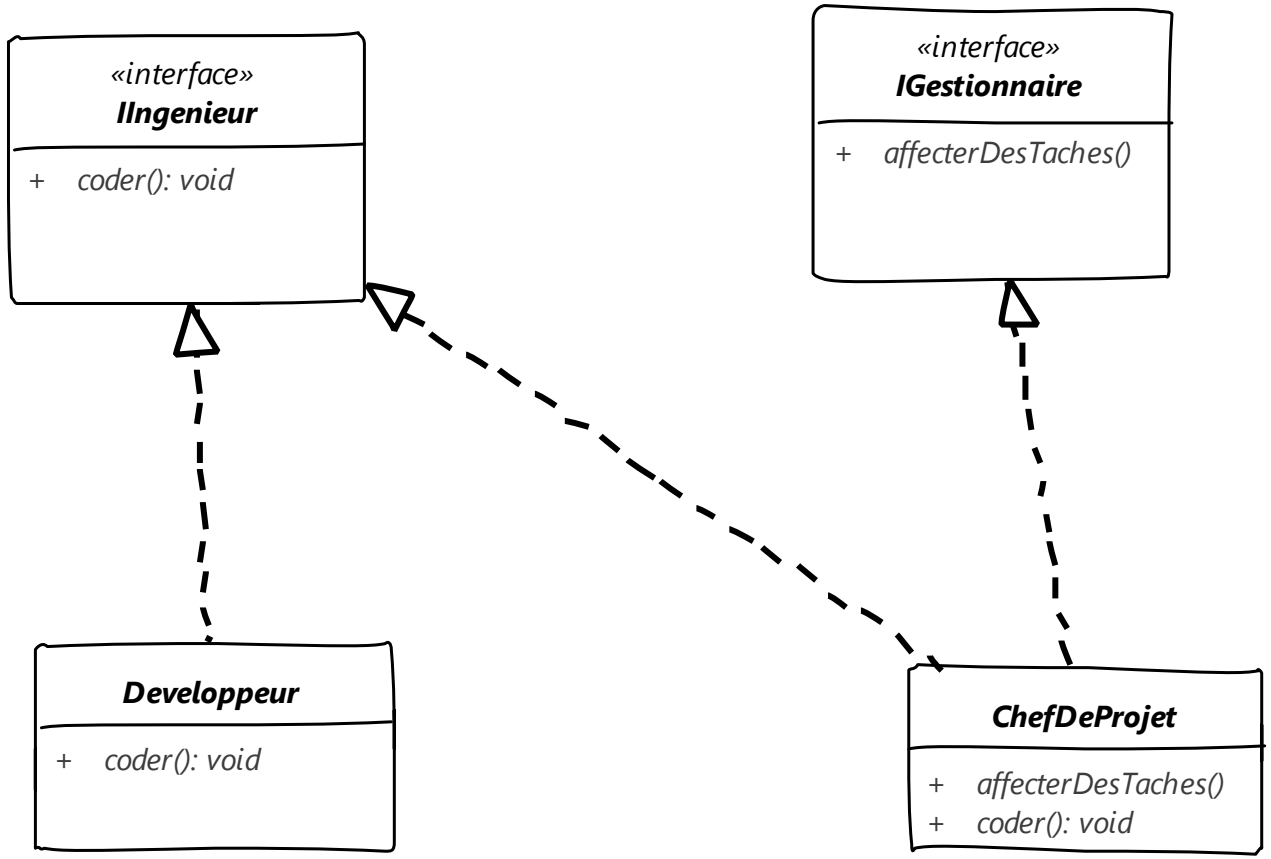
# ISP – Interface Segregation Principle

- Les classes ne sont pas obligées d'implémenter des méthodes qui ne les concernent pas
- Les clients ne doivent pas être forcées à dépendre de méthodes qu'elles n'utilisent pas

# ISP – Problème



# ISP – Solution



# Principes de Conception

SECTION 5, DÉBAT 05 MNS

# Bibliographie


- UML2 and the Unified Process, Second Edition, Addison Wesley 2005
- Software Development and Professional Practice, John Dooley, APress, 2010
- SOLID Principles in C#, By Damodhar Naidu, 2014, http://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/