

# Architectures d'Applications - Bachelor CSI

Christophe Brun

Campus Saint-Michel IT

03 avril 2024



# Table des matières

- ① Programme du module
- ② Généralités
- ③ Les briques
  - Généralités
  - Programmation structurée
  - Programmation fonctionnelle
  - Programmation orientée objet
  - Exercice
- ④ TDD
- ⑤ Les mesures
- ⑥ Mid-level architecture
  - Généralités
  - S.O.L.I.D
  - Single Responsibility Principle
  - Open-Closed Principle

# Architectures d'Applications

Compétence acquise au cours des 3 jours du module

Compétence :

- “Concevoir une architecture d'applications.” ???

# Architectures d'Applications

Compétence acquise au cours des 3 jours du module

Compétence :

- “Concevoir une architecture d'applications.” ???

Reformulation possible :

- Concevoir une application architecturée.



# Architectures d'Applications

Le programme officiel des 3 jours du module

## ① Les différentes architectures d'une application

### ② L'architecture REST

- Architectures Orientées Services
  - Besoins de la SOA
  - Notion de service
  - Introduction aux Architectures Orientées Services
- Vers les Architectures Orientées Services
  - Les architectures Client-Serveur
  - Les architectures Web
- Les Web Services
  - Appel de procédure
  - World Wide Web
  - Formats d'échange textuels
  - Vers la notion de Web Service
- Web Service de type SOAP et REST
- Guidelines API-REST
  - Gestion des actions et des URLs
  - Recherche, Tri, Filtre et Pagination
  - Gestion des erreurs

# Evaluation

- Bons points tout au long du module.
- 25 % x 3 sur chaque séance de travaux dirigés évalués en fin de séance.  
Et une application orientée web REST ou SOAP .
- 25 % sur une évaluation écrite finale

# Intervenant sur le module architecture d'Application

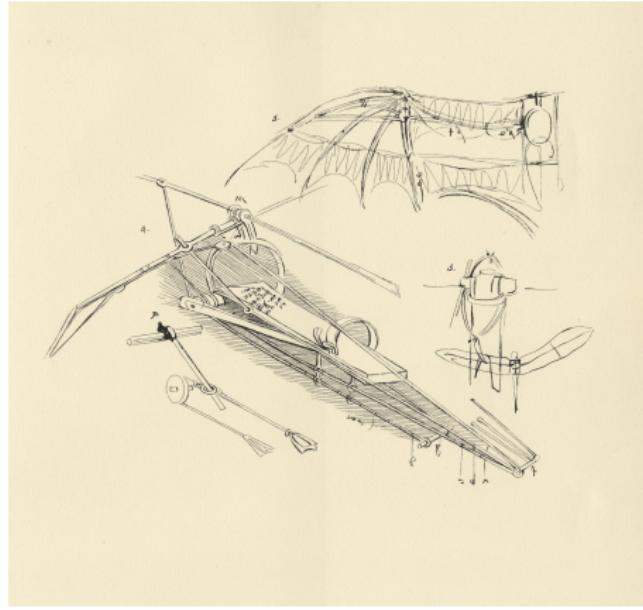
Christophe Brun, conseil en développement informatique

- 1<sup>ère</sup> année d'intervenant à Saint-Michel 😊.
- 7 ans de conseil en développement au sein d'SSII .
- 7 ans de conseil en développement à mon compte [PapIT](#).
- Passionné !



# Pourquoi le design ou architecture logicielle ?

“The goal of software architecture is to minimize the human resources required to build and maintain the required system.”<sup>1</sup>



L. de Vinci par C. G. Gerli en 1789.

<sup>1</sup>Clean architecture, Robert C. Martin

# Éloge de la simplicité

Léonard de Vinci, 1452-1519 : “La simplicité est la sophistication suprême.”<sup>2</sup>

Albert Einstein, 1879-1955 : “Si vous ne pouvez expliquer quelque chose simplement, c'est que vous ne l'avez pas bien compris.”

Trouver et implémenter une architecture logicielle doit être simple. C'est compliqué il y a un souci.

La modularité est en autre permise par la dissection d'un système complexe en sous-systèmes plus simples. Ces systèmes plus simples n'ont qu'une seule responsabilité (Comme en programmation fonctionnelle ?).

---

<sup>2</sup>La Pause Philo, <https://lapausephilo.fr/2015/09/15/simplicite-sophistication-supreme-leonard-de-vinci/>

# Définition

Architecture ?, Design ?, les deux ?

Encore une fois, pour faire simple, nous allons présumer que l'architecture et le design sont la même chose dans le monde du logiciel.<sup>1</sup> Le “Design patterns”<sup>3</sup> est un classique en développement logiciel.

“The only way to go fast is to go well.”<sup>1</sup> Qu'est-ce que cela veut dire ?

“The blueprint of the system.”<sup>4</sup>

Le mot système revient souvent. C'est un terme vague et récursif. C'est-à-dire qu'un système est composé de systèmes plus simples. L'architecture système est d'ailleurs un domaine en charge de l'architecture de l'entreprise, des produits et des logiciels.

---

<sup>3</sup>Design patterns, <https://refactoring.guru/design-patterns>

<sup>4</sup>Fundamentals of Software Architecture, Mark Richards et Neal Ford

# Définition

## Le couplage

C'est un des termes les plus importants en architecture logicielle. Il faut comprendre comment l'éviter tant que possible.

“It is a result of putting a variable, constant, or function in a temporary convenient, thought inappropriate, location. This is lazy and careless”<sup>5</sup>

---

<sup>5</sup>Robert C. Martin, Clean Code

# Les briques

## Les briques en architecture

Probablement hérité du jargon de l'architecture des bâtiments, le mot brique est aussi souvent utilisé en architecture logicielle.

Beaucoup d'industries utilisent des briques pour construire des systèmes plus complexes. Parfois appelée "conception modulaire", c'est par exemple, dans l'industrie automobile, réutiliser une même pièce sur plusieurs modèles de voitures.

Quelles sont les briques en architecture logicielle ?

1 des 4 règles pour la direction de l'esprit de Descartes : "Diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre.".



# Les briques

3 paradigmes de programmation représentant par des briques<sup>1</sup>

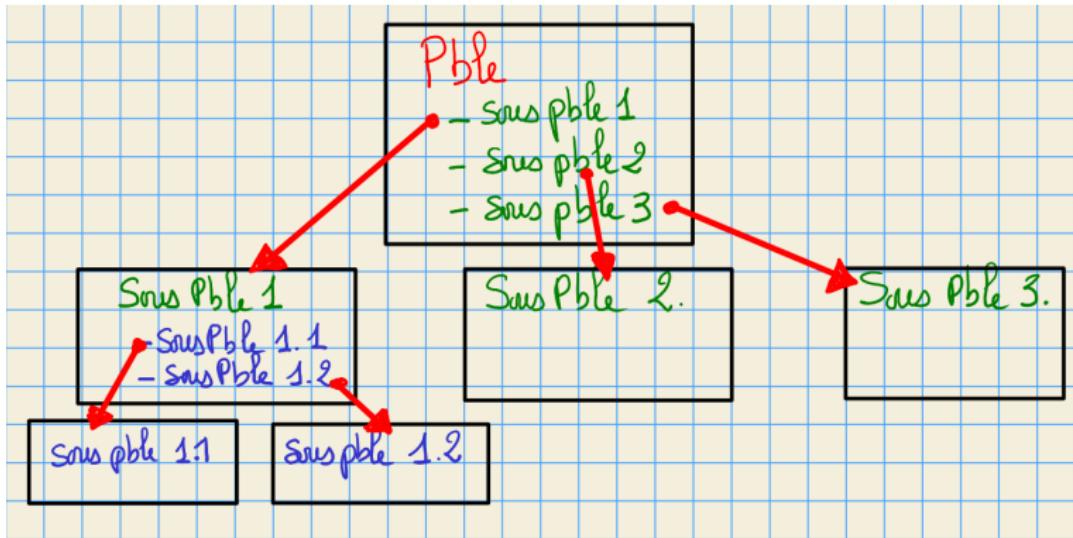
- Programmation structurée (sous-ensemble de la programmation impérative), où la brique est le module de code, i.e., des programmes et sous-programmes. Il contient les instructions de control de flux (if, else, for, while, etc.).
- Programmation orientée objet, où la brique est la classe.
- Programmation fonctionnelle, où la brique est la fonction.

La plupart des langages de programmation modernes supportent ces 3 paradigmes, JS, Python, Java, C/C++.

# Les briques

Exemple de programmation structurée

Utilisation de tout le control flow classique.



Analyse top-down pour division des tâches en sous-programmes<sup>6</sup>.

<sup>6</sup> Programmation structurée, [https://perso.univ-lyon1.fr/marc.buffat/COURS/BOOK\\_INPROS\\_HTML/CHAP5/COURS\\_ALGORITHMIQUE.html](https://perso.univ-lyon1.fr/marc.buffat/COURS/BOOK_INPROS_HTML/CHAP5/COURS_ALGORITHMIQUE.html)

# Les briques

Exemple de programmation fonctionnelle<sup>1</sup>

Les briques sont de simples fonctions, des fonctions dites pures. Les données ne sont modifiées uniquement que par ces fonctions.

Des exemples de langages fonctionnels : Haskell, Lisp, Erlang, Scala, F#.

D'autres langages supportent la programmation fonctionnelle comme Python, JS, Java en autres et ont des outils/librairies dédiées à la programmation fonctionnelle. Souvent nommées fonctions lambda ou "arrow functions", map, reduce, filter, etc.

```
>>> is_even = lambda x: x % 2 == 0 # No return needed?
>>> list(map(is_even, [1, 2, 3, 4]))
[False, True, False, True]
>>> list(filter(is_even, [1, 2, 3, 4])) # Kept only if True
[2, 4]
>>> from functools import reduce
>>> from operator import mul
>>> reduce(mul, [1, 2, 3, 4]) # Function applied to all elements, 1 * 2 * 3 * 4
24
```

# Les briques

Exemple de programmation fonctionnelle

3 modules dédiés à la programmation fonctionnelle dans la “standard library” Python<sup>7</sup> :

- `functools`, “Higher-order functions and operations on callable objects”.
- `itertools`, “Functions creating iterators for efficient looping”.
- `operator`, “Standard operators as functions”.

L’équivalent en JS, sans aucune librairie à importer :

```
> numbers = new Array(1, 2, 3, 4, 5)
[ 1, 2, 3, 4, 5 ]
> const isEven = x => x % 2 === 0;
undefined
> numbers.map(isEven);
[ false, true, false, true, false ]
> numbers.reduce(isEven);
true
> numbers.filter(isEven);
[ 2, 4 ]
> numbers.reduce((x, y) => y * x);
120
```

---

<sup>7</sup>Functional Programming Modules,

# Les briques

Exemple de programmation orientée objet<sup>1</sup>

(OOP) La brique de base en programmation orientée objet est la classe.

En OOP on parle de polymorphisme, c'est la capacité d'un objet, i.e., d'une classe, à prendre plusieurs formes. Une évolution dans une classe fille n'implique pas de modification dans la classe mère. Une évolution dans une classe mère impacte les classes filles. Si ces classes sont dans des modules différents, il n'est même pas nécessaire de recompiler tous les modules. Pouvoir contrôler quelle classe dépend de quelle autre est un atout majeur de cette architecture logicielle.

L'encapsulation des données et des méthodes dans une classe permet un design simple de ce qui appartient à un objet, une instance de classe, et ce qui est accessible depuis l'extérieur.

L'encapsulation et l'héritage qui permet le polymorphisme sont les 2 atouts principaux de la programmation orientée objet.

# Les briques

Exemple de programmation orientée objet

Polymorphisme par héritage d'une classe.

```
>>> class Oiseau:  
    def __init__(self, name):  
        self.name = name  
  
>>> class Pigeon(Oiseau):  
    def __init__(self, name):  
        Oiseau.__init__(self, name)  
    def mange_un_filtre_cigarette(self):  
        print("{} mange un filtre de cigarette!!!".format(self.name))  
  
>>> class Cygne(Oiseau):  
    def __init__(self, name):  
        Oiseau.__init__(self, name)  
    def plonge(self):  
        print("{} plonge dans le lac!!!".format(self.name))  
  
>>> goelan = Oiseau("Jonathan Livingston")  
>>> cygne = Cygne("Juste Leblanc") # C'est un Oiseau aussi !  
>>> pigeon = Pigeon("Casimir") # Même le pigeaoen est un Oiseau !  
>>> all(map(lambda x: isinstance(x, Oiseau), [cygne, pigeon, goelan]))  
True
```

# Les briques

Exemple de programmation orientée objet

Polymorphisme par héritage d'une classe abstraite.

```
>>> from abc import ABC, abstractmethod

>>> class Oiseau(ABC): # ABC -> ABstract Class
    def __init__(self):
        pass
    @abstractmethod
    def vole(self): # Polymorphisme au niveau de cette méthode !
        raise NotImplementedError

>>> class Poule(Oiseau):
    def __init__(self):
        Oiseau.__init__(self)
    def vole(self): # Chacun vole à sa manière, son implémentation
        print("Je vole quelques mètres")

>>> cocote = Poule()

>>> cocote.vol()
Je vole quelques mètres
```

# Les briques

Exemple de programmation orientée objet

Contrainte d'implémentation d'une méthode abstraite `vole`.

```
>>> class Dodo(Oiseau):
    def __init__(self):
        Oiseau.__init__(self)

>>> Aussie = Dodo()
-----
TypeError Traceback (most recent call last)
Cell In [10], line 1
----> 1 Aussie = Dodo()

TypeError: Can't instantiate abstract class Dodo with abstract method vole
```

# Les types de briques

## Exercice 1 de 5 minutes

Trouver quel type de paradigme de programmation est utilisé dans les exemples suivants :

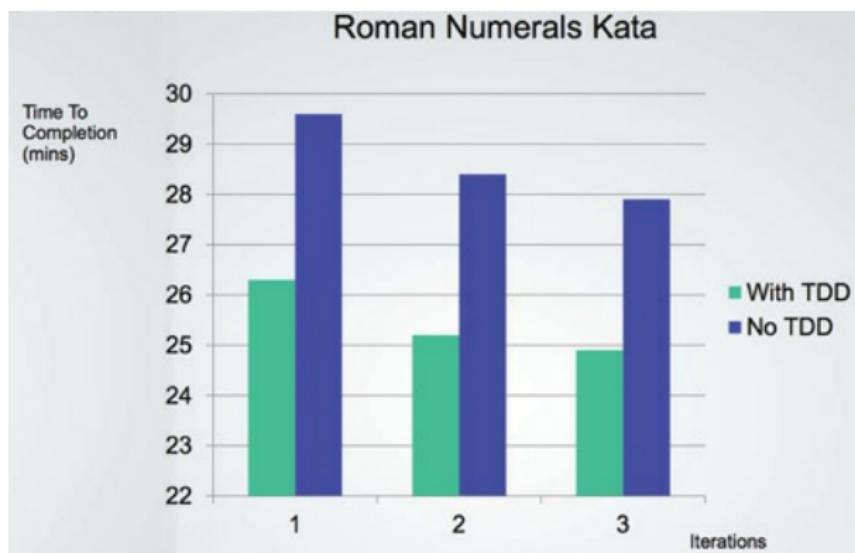
- <https://github.com/facebook/Haxl/>
- [https://github.com/gurupratap-matharu/  
Bike-Rental-System/blob/master/main.py](https://github.com/gurupratap-matharu/Bike-Rental-System/blob/master/main.py)
- <https://github.com/papit-fr/fsma/>

Tirage au sort d'un étudiant pour chaque exemple, il doit expliquer pourquoi il pense que c'est ce type de programmation.

# Le TDD

## Test Driven Development

Vouloir aller plus vite sans faire de test semble être une mauvaise idée.  
Ce qui pratiquent le TDD vont plus vite dès le début d'un projet<sup>1</sup> !  
Entre autre, les tests unitaires font réduire la taille des classes, functions et  
des méthodes, ce qui améliore architecture.

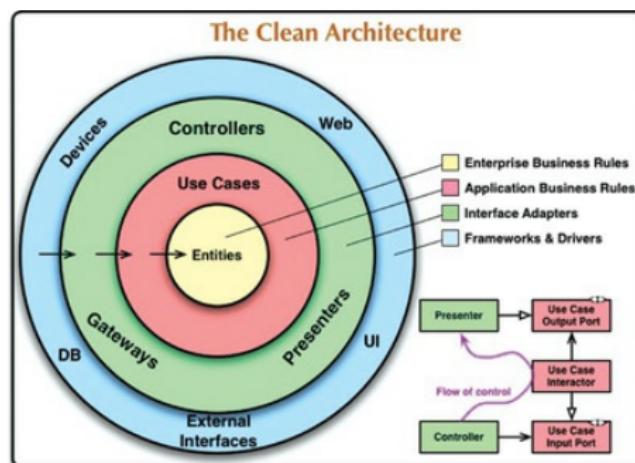


# Le TDD

Quel type(s) de test ?

Tous les types de tests possibles sont à écrire dès que possible avant même de commencer à coder. Les tests unitaires sont en général les plus simples à écrire et les plus rapides à exécuter.

Les éléments centraux, les règles de gestions, les “business rules” doivent être testables sans les éléments périphériques<sup>1</sup> :



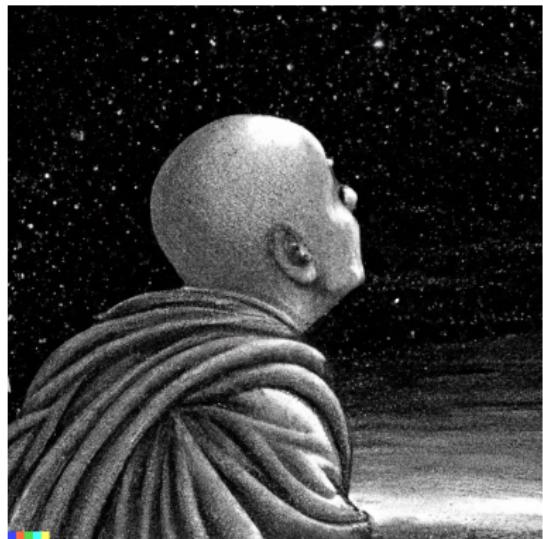
# Le TDD

Quel types de test ?

Edsger Dijkstra : “Testing shows the presence, not the absence of bugs.”

Autrement dit, l'univers des tests est infini, on n'a jamais la certitude de suffisamment tester.

Même le coverage est une mauvaise mesure.



# Le TDD

## Exercices 2

De nombreux exercices de TDD sont disponibles sur le web comme par sur GitHub, <https://github.com/gabbloquet/entraînement-au-tdd>.

En 30 minutes, sans assistant du type ChatGPT, réaliser un des plus classique d'entre eux, le “FizzBuzz”, <https://github.com/gabbloquet/entraînement-au-tdd/blob/master/src/main/java/io/github/gabbloquet/tddtraining/FizzBuzz/FizzBuzz.java>. Respecter l'esprit du TDD qui veut que l'on écrive les tests avant le code :

- ① Écrire un test par requirement.
- ② Écrire l'algorithme.
- ③ Refactoriser.
- ④ Revenir à l'étape 2 jusqu'à ce que tous les test passent.

# Les mesures du couplage

## L'importance des données

L'architecture c'est le monde du compromis, des possibilités infinies.

L'intuition, l'expérience, les connaissances et le bon sens, nous guiderons. Mais il est possible de qualifier l'architecture avec des mesures et donc d'avoir une approche "Data Driven".

Nous verrons dans cette section les différentes des métriques classiques, mais pas toutes, car elles sont nombreuses.

On parle parfois aussi des CK (Chidamber & Kemerer) metrics qui peuvent ressembler, etc.

# Les mesures du couplage

L’“abstractness”  $A^4$

L’“abstractness” une mesure de l’abstraction par rapport aux implémentations concrètes. Elle fut introduite par Robert C. Martin.

$$A = \frac{\sum Ca}{\sum Cc} \quad (1)$$

Dans l’équation 1 :

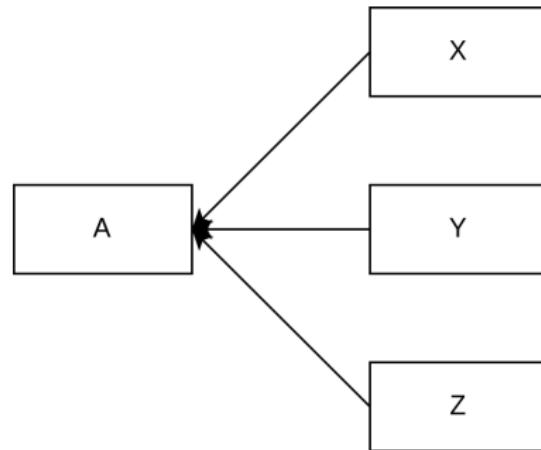
- $A$  est “abstractness”.
- $Ca$  est le nombre de classe(s) abstraite(s).
- $Cc$  est le nombre de classe(s) concrète(s).

Les valeurs de  $A$  sont entre 0 et 1. Les extrêmes proches de 0 ou 1 sont à éviter, elles représentent des architectures trop concrètes ou trop abstraites.

# Les mesures du couplage

## Couplage afférent

Le couplage afférent est le nombre total de classe(s) qui dépend(ent) de la classe A :



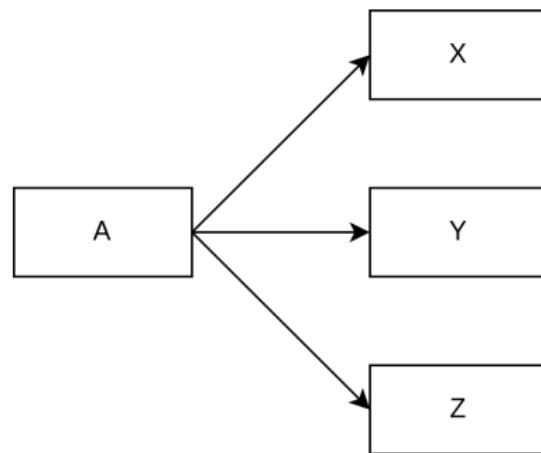
Ici  $C_a$  est 3.

Un  $C_a$  élevé impacte principalement la portabilité. Car ce code viendra forcément avec beaucoup de code en dépendance.

# Les mesures du couplage

## Couplage efférent $C_e$

Le couplage efférent est le nombre total de classe(s) dont dépend la classe A :



Ici  $C_e$  est 3.

Plus le  $C_e$  est élevé, plus le code est difficile à reuse et à maintenir.

# Les mesures du couplage

L’“instability”  $I^4$

L’“instability” est une autre mesure qui en découle. Elle détermine la volatilité du code, l’effort à fournir pour modifier une partie du code sans devoir en modifier une autre.

$$I = \frac{Ce}{Ce + Ca} \quad (2)$$

Dans l’équation 2 :

- $I$  est “instability”
- $Ca$  couplage afférent, objet ou package en entrée (“a” en premier d’où “entrée”).
- $Ce$  couplage efférent, objet ou package en sortie (“e” comme “exit”).

Quand elle est trop élevée, qu’elle tend vers 1, le code casse vite à cause d’un fort couplage. Elle a donc un impact négatif sur le reuse, les correctifs, la maintenance et la portabilité.

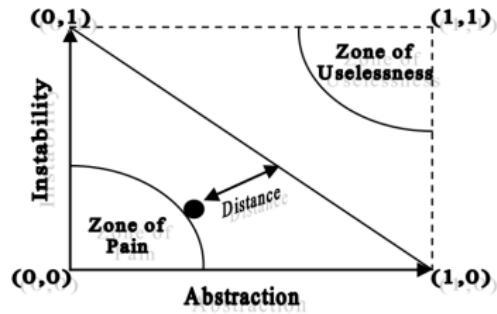
# Les mesures du couplage

La "Distance from main sequence"  $D^4, 8$

Une cinquième métrique introduite par Robert C. Martin est la "Distance from main sequence".

Elle définit une relation entre  $A$  et  $I$  comme suit :

$$D = |A + I - 1| \quad (3)$$



$D$  doit s'approcher de 0 pour une architecture souhaitable. Si  $D$  est élevé, on s'éloigne de compromis acceptable.

<sup>8</sup>A Study on Robert C.Martin's Metrics for Packet Categorization Using Fuzzy Logic,  
Gurpreet Kaur and Deepak

Sharma [https://gvpress.com/journals/IJHIT/vol8\\_no12/15.pdf](https://gvpress.com/journals/IJHIT/vol8_no12/15.pdf)

# Les mesures du couplage

Exercice 3 de 30 minutes

Calculer les valeurs de  $C_a$ ,  $C_e$ ,  $A$ ,  $I$  et  $D$  pour chaque classe du source  
[https://github.com/St-Michel-IT/architecture-application/  
blob/main/coupling.py](https://github.com/St-Michel-IT/architecture-application/blob/main/coupling.py).

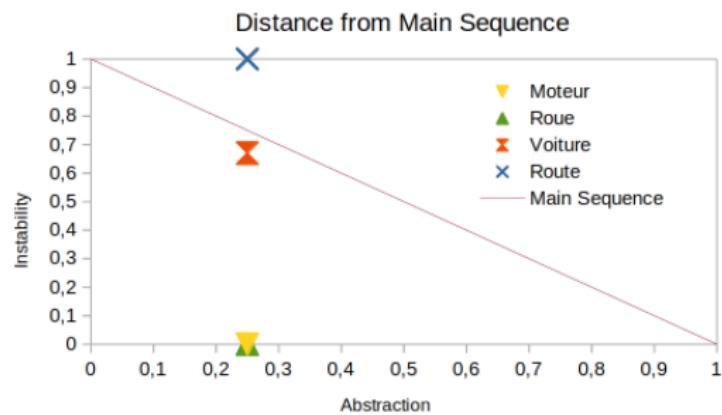
Restituer le résultat dans un tableau Excel et un graphique avec les  $D$  et la droite “main sequence”.

# Les mesures du couplage

Exercice 4 de 30 minutes

Résultat de l'exercice :

Classe Formula	Ca #N/A	Ce #N/A =1/4	A =ROUND(C3/(C3+B3);2)	I =ROUND(ABS(D3+E3-1);2)	D
Moteur	1	0	0,25	0	0,75
Roue	1	0	0,25	0	0,75
Voiture	1	2	0,25	0,67	0,08
Route	0	1	0,25	1	0,25
Main Sequence			0	1	
Main Sequence			1	0	



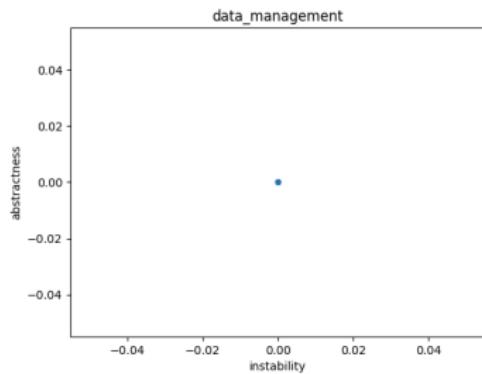
# Les mesures du couplage

## Les outils

Certains outils comme JCAT<sup>9</sup> en Java ou `module_coupling_metrics`<sup>10</sup> en Python permettent de calculer ces métriques.

```
$ module_coupling_metrics data_management.py
```

```
  Component : Instability : Abstractness : Distance :  
  data_management.py : 0.000000 : 0.000000 : 0.000000
```



Attention à bien comprendre ces métriques et comment les interpréter.  
`module_coupling_metrics` ne calcule le couplage qu'entre modules.

<sup>9</sup>Tool for Measuring Coupling in Object-Oriented Java Software,  
<https://shorturl.at/npDR2>

<sup>10</sup>[https://github.com/0az/module\\_coupling\\_metrics](https://github.com/0az/module_coupling_metrics)

# Les mesures de la complexité cyclomatique

## La théorie

Établie en 1976, elle est devenue le standard de mesure de complexité du code.

Elle se calcule pour une méthode ou fonction avec l'équation suivante :

$$CC = E - N + 2 \quad (4)$$

Où :

- $CC$  est la complexité cyclomatique.
- $E$  comme “edge”, est le nombre d'arêtes (les `return`, `yield`, `exit`) du graphe de contrôle de flux.
- $N$  comme “node”, est le nombre de nœuds, les `if`, `while`, `for`, du graphe de contrôle de flux.

Les valeurs au-dessous de 5 sont bonnes, au-dessus de 10 il y a danger<sup>4</sup>.

Une CC trop élevée est un code smell de Sonar Qube.

# Les mesures de la complexité cyclomatique

## Exemple de fonction

```
uint64_t fsma(uint64_t base, uint64_t exp, uint64_t mod) {
    uint64_t res = 1;
    while (exp > 1) {
        // If the exponent digit is 1, then multiply
        if (exp & 1) {
            res = (res * base) % mod;
            // If an intermediate result is zero, we can return 0.
            // The result will be zero anyway
            if (res == 0) {
                return 0;
            }
        }
        // If the exponent digit is 0, then square
        base = (base * base) % mod;
        exp >>= 1;
    }
    return (base * res) % mod;
}
```

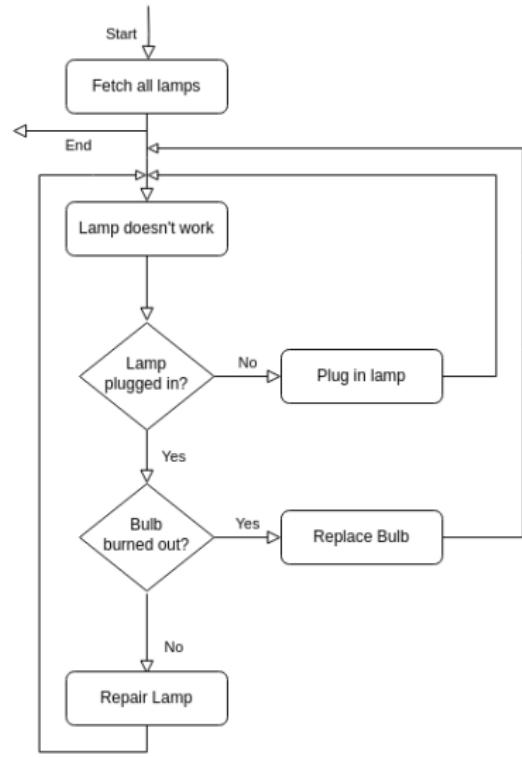
Ici  $E$  est 2 et  $N$  est 3, donc CC est 1.

# Les mesures de la complexité cyclomatique

## Exercice 5

Cette métrique peut se calculer plus facilement avec un graphe de contrôle de flux.

Calculer la complexité cyclomatique de la fonction décrite à droite.



# Les mesures de la complexité cyclomatique

## Exercice 6

Calculer la complexité cyclomatique des fonctions de l'implémentation de l'algorithme de hachage MD5 du noyau Linux.

Le code est sur Github :

<https://github.com/torvalds/linux/blob/master/crypto/md5.c>

Rendre un tableau des valeurs de CC pour chaque fonction et en tirer une conclusion.

# “Mid-level” architecture

Comment agencer les briques

Les briques vues précédemment sont nécessaires mais pas suffisantes pour construire un système. Si elles ne sont pas correctement conçues il est inutile de les intégrer un design élaboré, le système risque fortement de ne pas fonctionner

Pour qu'un système fonctionne, il faut que les briques soient correctement agencées. C'est la “mid-level architecture”.

De nombreux principes ou patterns existent pour agencer les briques.  
Comme par exemple :

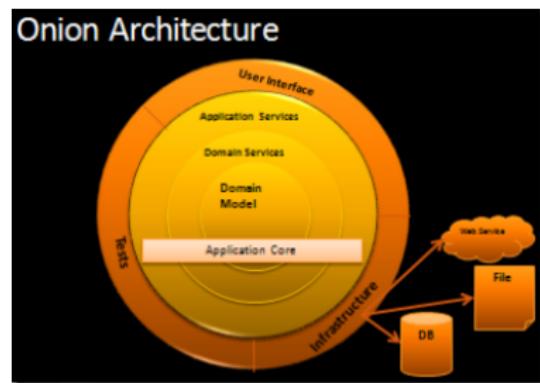
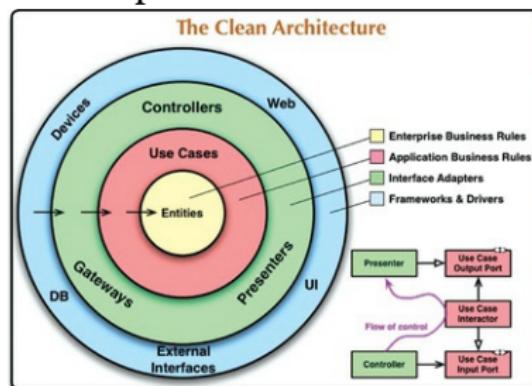
- **SOLID** : Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.
- **Hexagonal Architecture** : A.K.A Ports and Adapters.
- **Onion Architecture** : Une couche intérieur ne sait rien de celle plus extérieur.
- Many more...

# “Mid-level” architecture

Comment agencer les briques

De manière générale ces différents patterns ne sont pas concurrents. Ils ne se contredisent pas. Ils peuvent même être utilisés ensemble.

La coexistence des ces patterns permet de visualiser un même problème sous différents point de vue<sup>1, 11</sup> :



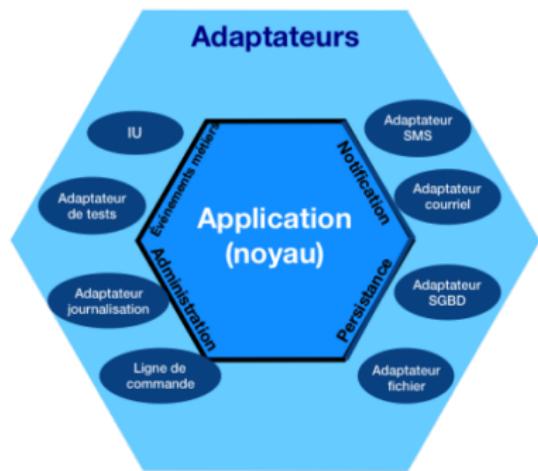
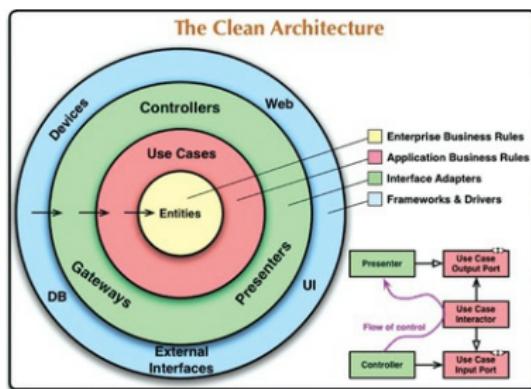
<sup>11</sup>The Onion Architecture : part 1,

<https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/>

# “Mid-level” architecture

Comment agencer les briques

De même l'architecture hexagonale semble proche elle aussi<sup>1, 12</sup> :



<sup>12</sup>Architecture hexagonale, Wikipédia,

[https://fr.wikipedia.org/wiki/Architecture\\_hexagonale](https://fr.wikipedia.org/wiki/Architecture_hexagonale)

# Les principes S.O.L.I.D

## Histoire et définition<sup>1</sup>

Démarré dans les années 80, ils sont finalisés en 2004 par Robert C. Martin, le S.O.L.I.D est un acronyme pour 5 principes de design.

Les buts de ces principes sont :

- Tolérer le changement.
- Être facile à comprendre.
- Ils sont les composants de base qui peuvent être utilisés dans tout logiciel.

C'est du "Mid-level architecture", car même avec des briques bien conçues, agencées selon ces principes, il manque encore l'architecture de plus haut niveau.

# Les principes S.O.L.I.D

## Les 5 principes résumés<sup>1</sup>

- **Single Responsibility Principle** : Une classe ne doit avoir qu'une seule responsabilité.
- **Open-Closed Principle** : Une classe doit être ouverte à l'extension mais fermée à la modification.
- **Liskov Substitution Principle** : Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme. Cela garantit que les sous-classes peuvent être utilisées de manière interchangeable avec leurs classes de base.
- **Interface Segregation Principle** : Les clients ne doivent pas être forcés d'implémenter des interfaces qu'ils n'utilisent pas. Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale. Cela évite aux classes de dépendre de méthodes dont elles n'ont pas besoin, réduisant ainsi les couplages inutiles.
- **Dependency Inversion Principle** : Il faut dépendre des abstractions, pas des implémentations. Cela favorise la modularité, la flexibilité et la réutilisabilité en réduisant les dépendances directes entre les modules. Le code des règles de haut-niveau ne doit pas dépendre du code de détail bas-niveau. Mais les détails peuvent dépendre des règles haut-niveau.

# Single Responsibility Principle (SRP)

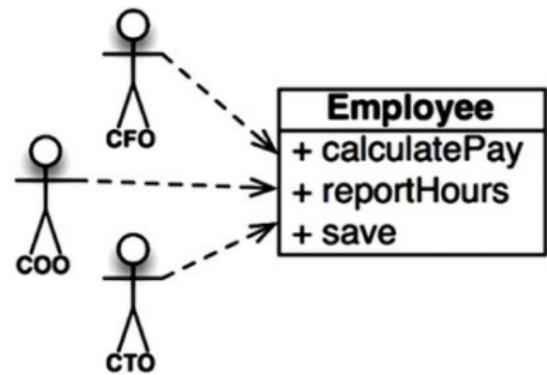
Définition<sup>1</sup>

Ne pas confondre avec le refactor d'une méthode ou d'une fonction en méthode ou fonction plus atomique.

Selon Robert C. Martin, “A module should be responsible to one, and only one, actor”.

Module est souvent synonyme de fichier source.

Donc autrement dit, dans un fichier source, on trouve les fonctions ou les structures de données responsables face à seul acteur.

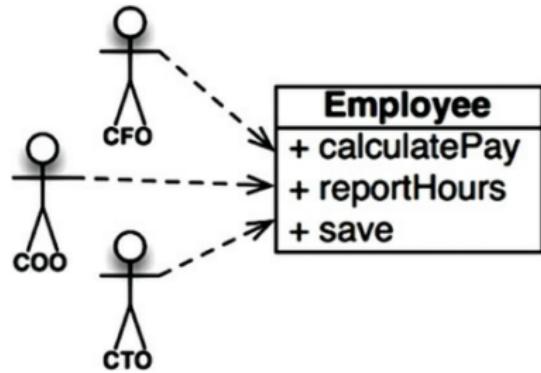


A éviter

# Single Responsibility Principle

Exercice 7 de 15 minutes

Quelles solutions pour éviter ce pattern anti-SRP ?



Les exigences :

- Refactoriser des fonctions `calculatePay`, `reportHours` et `save` avec le(s) éventuel(s) argument(s) nécessaire pour qu'elles respectent le SRP .
- Ajouter les attribues nécessaire et au moins 4 méthodes à la classe `Employee` tout en respectant le SRP .
- Répondre sous formes d'un schéma (en utilisant Draw.io par exemple) avec les différents et classe(s), méthodes.

# Single Responsibility Principle

Exercice 8 de 30 minutes

Codez la solution de manière simpliste.

Appliquer le TDD, rédigez les tests unitaires avant de coder.

A quoi ressemble l'arborescence du projet ?

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Open-Closed Principle (OCP)

Définition<sup>1</sup>

OCP veut dire : “A software artifact should be open for extension but closed for modification”.

En d'autre termes, un code doit être extensible, on peut ajouter du code et des fonctionnalités sans modifier le code existant. Encore une fois, cela permet de tolérer le changement en limitant les risques de bug et en facilitant la maintenance.

Une des solutions est de séparer tout ce qui est susceptible de changer pour des raisons différentes dans le futur.

# Open-Closed Principle

Exercice 9 de 30 minutes

Refactoriser le code <https://github.com/St-Michel-IT/architecture-application/open-close-bank-customer.py>. Le code souhaité doit être extensible sans modification pour pouvoir retourner les informations actuellement printées dans stdout en HTML .  
Appliquer le TDD, rédigez les tests unitaires avant de coder.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Open-Closed Principle

Exercice 10 de 30 minutes avec les interfaces

En Java comme dans beaucoup de langage il existe des interfaces. Pour rappel, une interface est une structure, une liste de méthodes que les classes qui l'implémentent doivent respecter. Comme les classes abstraites, une implementation, i.e., une classe en hérite, elles ne peuvent pas être instanciée directement.

```
public class UserAgeValidator {  
    public boolean isOldEnoughToDrinkAlcohol(int age) {  
        return age >= 18;  
    }  
}
```

Cette classe ne respecte pas le OCP, si on l'applique à plusieurs pays.

```
public class UserAgeValidator {  
    public boolean isOldEnoughToDrinkAlcohol(int age, String stateCode) {  
        return stateCode.equalsIgnoreCase('FR') && age >= 18  
            || stateCode.equalsIgnoreCase('US') && age >= 21  
    }  
}
```

# Open-Closed Principle

Exercice 11 de 30 minutes avec les interfaces

En utilisant les interfaces, refactoriser la classe `UserAgeValidator` pour qu'elle respecte le OCP et créer les classes qui l'implémente pour divers pays.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Open-Closed Principle

Exercice 12 de 30 minutes avec les interfaces

En utilisant les interfaces, refactoriser la classe `UserAgeValidator` pour qu'elle respecte le OCP et créer les classes qui l'implémente pour divers pays.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Liskov Principle

## Définition<sup>1</sup>

Une classe enfant doit pouvoir être utilisée à la place de sa classe mère sans que cela ne modifie le comportement du programme.

Pour rappel, évidemment, une classe enfant peut être substitué par un autre enfant de la même classe mère.

Autrement dit une classe fille ne doit pas casser pas le code de la classe mère.  
Ne jamais surcharger une méthode de la classe mère.

```
# EXEMPLE À NE PAS SUIVRE !!!
class Bird:
    def fly(self):
        pass

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostriches cannot fly!")

bird = Bird()
bird.fly() # Output: (no implementation)

ostrich = Ostrich()
ostrich.fly() # Raises NotImplementedError
```

# Liskov Principle

Exercice 13 de refactoring de 15 minutes

Refactoriser le code <https://github.com/St-Michel-IT/architecture-application/liskov-rectangle.py>. La fonction `use_it` ne peut pas être utilisée avec la classe `Square` sans modification. Refactoriser cette dernière pour qu'elle le puisse et donc qu'elle respecte le Liskov Principle.

Appliquer le TDD, rédigez les tests unitaires avant de coder.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Liskov Principle

Exercice 14 de refactoring de 15 minutes

Refactoriser le code <https://github.com/St-Michel-IT/architecture-application/liskov-rectangle.py>. La fonction `use_it` ne peut pas être utilisée avec la classe `Square` sans modification. Refactoriser les classes `Square` et `Rectangle` pour qu'elles héritent d'une classe abstraite à développer pour que `use_it` fonctionne avec les deux classes.

Appliquer le TDD, rédigez les tests unitaires avant de coder.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.

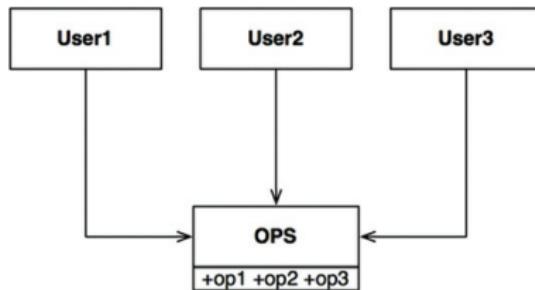


# Interface Segregation Principle (ISP)

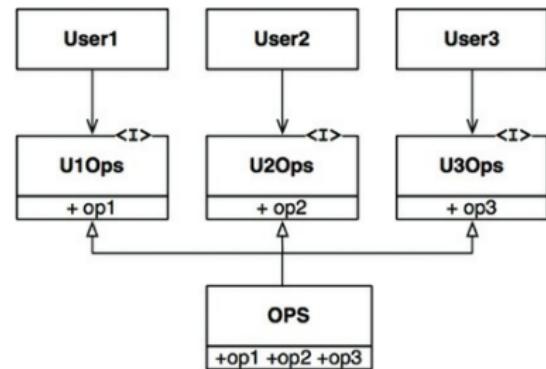
Définition<sup>1</sup>

Dépendre de quelque qui a un bagage que l'on utilise pas peut être une source de bugs inattendus.

Pour éviter ce risque, il faut multiplier les interfaces plus petites pour mieux isoler l'impact d'une modification.



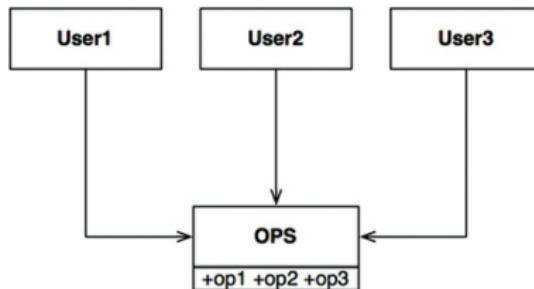
A éviter



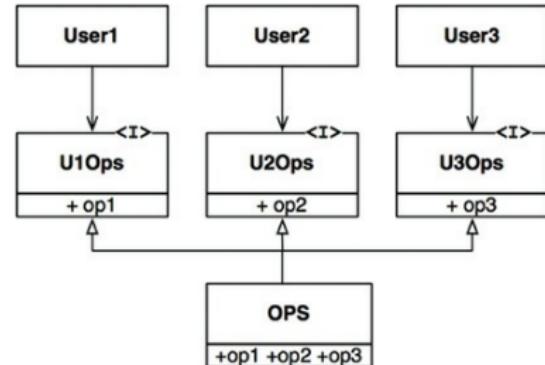
A appliquer

# Interface Segregation Principle

Définition<sup>1</sup>



A éviter



A appliquer

Dans le cas d'un langage typé statiquement, comme Java, à gauche on a une interface unique, et si il y a une modification dans une classe `UserX`, recompilation, déploiement toutes les classes `UserX` doivent être recompilées et redéployées même si elles ne sont pas modifiées.

Ce problème n'existe pas avec l'approche à droite. `Ops` peut être modifié sans avoir à recompiler les classes `UserX`.

# Interface Segregation Principle

Définition<sup>1</sup>

Question :Pourquoi ce problème n'apparaît pas avec les langages typés dynamiquement ?

# Interface Segregation Principle

## Définition<sup>1</sup>

Question :Pourquoi ce problème n'apparaît pas avec les langages typés dynamiquement ?

Car le type est inféré au moment de l'exécution puisqu'il est dynamique.

Cela ne veut pas dire qu'il ne faut pas respecter l'ISP dans ces langages. Les dépendances inutiles sont toujours à éviter, l'ISP est juste une raison de plus.

Quel rapport avec les dépendances aux packages distants (du web) ?

# Interface Segregation Principle

## Définition<sup>1</sup>

Question :Pourquoi ce problème n'apparaît pas avec les langages typés dynamiquement ?

Car le type est inféré au moment de l'exécution puisqu'il est dynamique.

Cela ne veut pas dire qu'il ne faut pas respecter l'ISP dans ces langages. Les dépendances inutiles sont toujours à éviter, l'ISP est juste une raison de plus.

Quel rapport avec les dépendances aux packages distants (du web) ?

On ne connaît pas le code de ces packages, on ne sait pas ce qu'ils contiennent. Le plus souvent, au mieux, on en connaît les interfaces décrites dans la documentation.

# Interface Segregation Principle

Exercice 15 de refactoring de 30 minutes

Dans le code <https://github.com/St-Michel-IT/architecture-application/interface-segregation-payment.py>.

- Trouver les classes qui ne respectent pas l'ISP .
- Refactoriser le code pour qu'il respecte l'ISP . **Indice :** Utiliser les classes abstraites.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Dependency Inversion Principle (DIP)

Définition<sup>1</sup>

Dépendre des structures les plus stables, i.e., les moins volatiles.

Tout changement dans une classe abstraite implique un changement dans les classes concrètes qui en héritent. L'inverse n'est pas vrai. Un changement dans une classe concrète n'implique pas forcément de modification dans les classes abstraites.

Par conséquent les abstractions et les interfaces sont plus stables que les implémentations.

Autrement dit, le DIP conseil, **tant que faire ce peut**, de dépendre des abstractions et non des implementations. Les `import`, `use`, `require`, `include`, etc, doivent pointer vers des interfaces et non des classes concrètes.

Il n'est pas réaliste de suivre de manière stricte cette règle. Tentez de l'appliquer quand c'est possible sans rajouter de complexité. Gardons en tête que certaine classes concrètes sont extrêmement stable comme `String` de `java.lang.string`.

# Dependency Inversion Principle

Exercice 16 de refactoring de 15 minutes

Dans le code <https://github.com/St-Michel-IT/architecture-application/dip-to-refactor.py>.

- Le couplage entre la classe PowerSwitch et LightBulb. On peut être l'interrupteur d'autre chose qu'une ampoule.
- Refactoriser le code pour qu'il respecte l'DIP . Indice : Ajouter une classe abstraite.
- Ajouter une classe concrète Fan qui implémente la nouvelle classe abstraite comme test.

Appliquer le TDD, rédigez les tests unitaires avant de coder.

Committer dans un dépôt Git, le schéma issue de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# S.O.L.I.D.

## Conclusion

S.O.L.I.D. est un ensemble de principes qui aident à solutionner des problèmes déjà vus aux chapitres précédents.

- Principalement les soucis de couplage trop fort section 2.
- Une des solutions récurrente pour respecter les principes S.O.L.I.D. est de créer des abstractions. Cela rappel à la mesure de l’“Abstractness” A mesurer au précédent chapitre section 5.

