

# 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

4 TDD

5 Les mesures

6 Mid-level architecture

7 High Level Desgin (HLD)

8 Service-Oriented Architecture

9 Les architectures des échanges du web

10 Conclusion

# 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

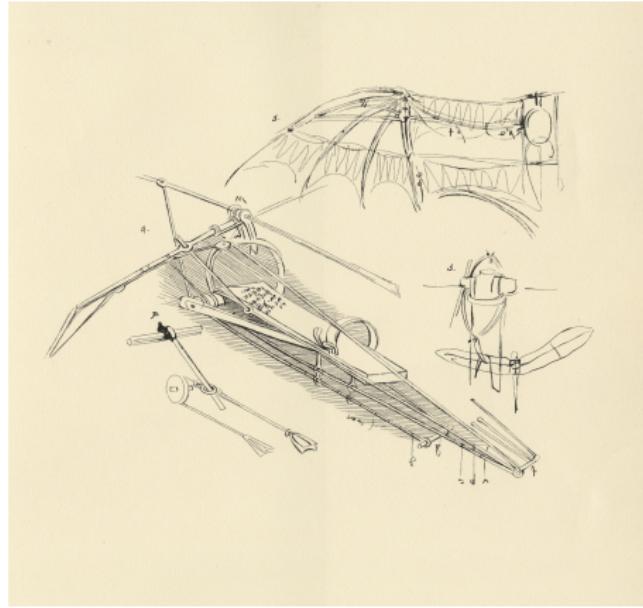
Christophe Brun, conseil en développement informatique

- 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>

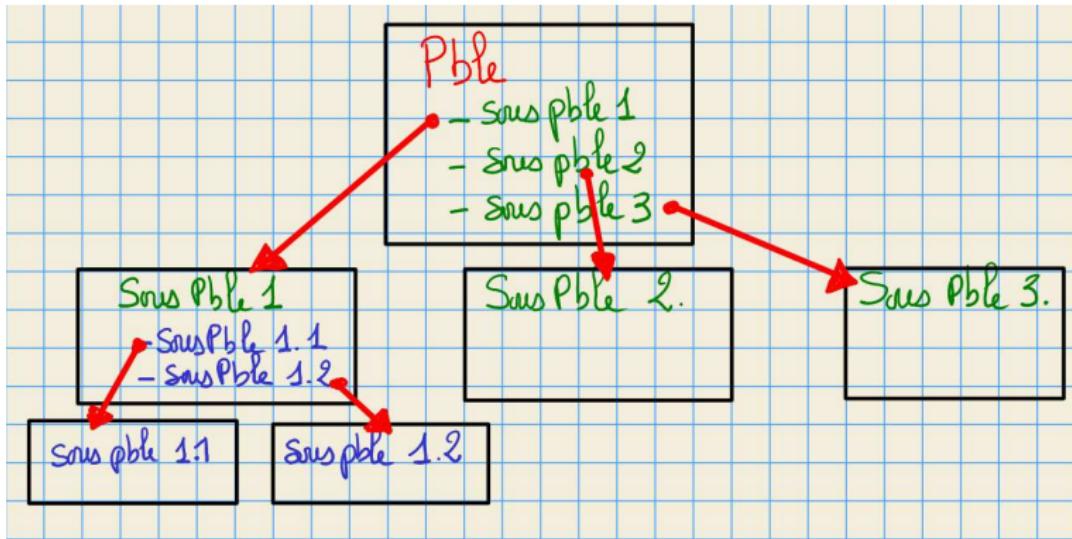
- Programmation structurée (sous-ensemble de la programmation impérative), où la brique est le module de code, i.e., des programmes et sous-programmes. Il contient les instructions de control de flux (if, else, for, while, etc.).
- Programmation orientée objet, où la brique est la classe.
- Programmation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

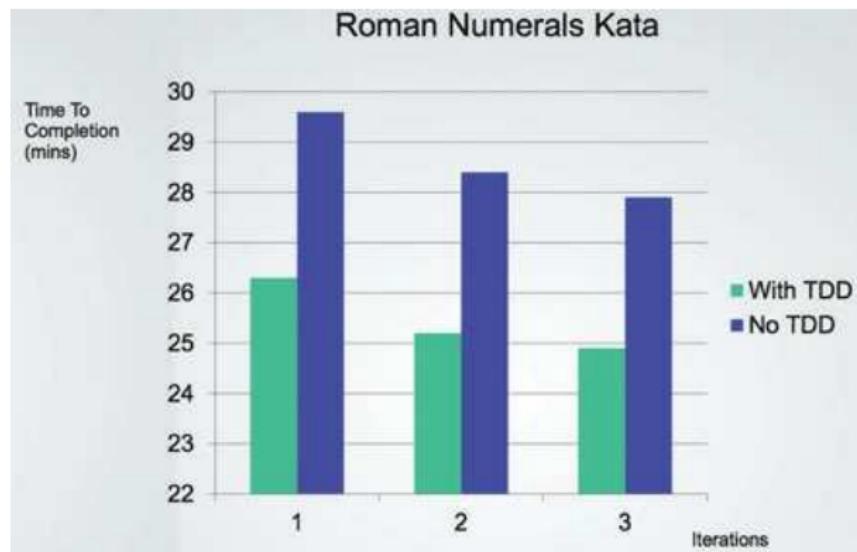
- <https://github.com/facebook/Haxl/>
- [https://github.com/gurupratap-matharu/  
Bike-Rental-System/blob/master/main.py](https://github.com/gurupratap-matharu/Bike-Rental-System/blob/master/main.py)
- <https://github.com/papit-fr/fsma/>

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.

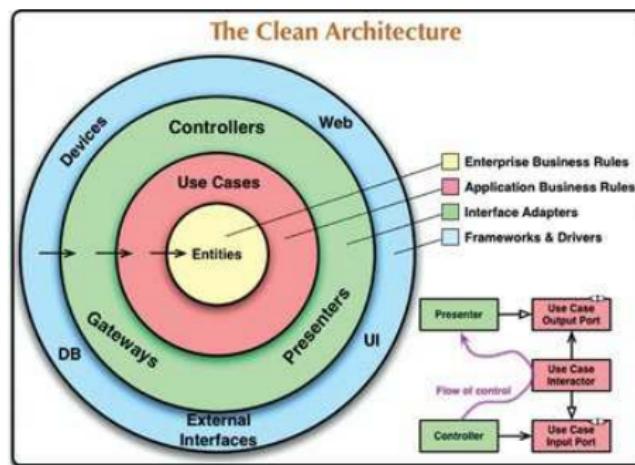


# Le TDD

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> :



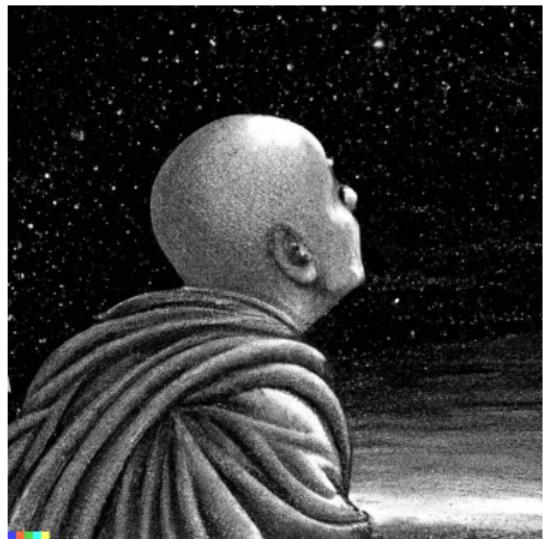
# Le TDD

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 l'un des plus classique d'entre eux, le "FizzBuzz", <https://github.com/gabbloquet/entraînement-au-tdd/tree/master/src/main/java/io/github/gabbloquet/tddtraining/FizzBuzz> .

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 :

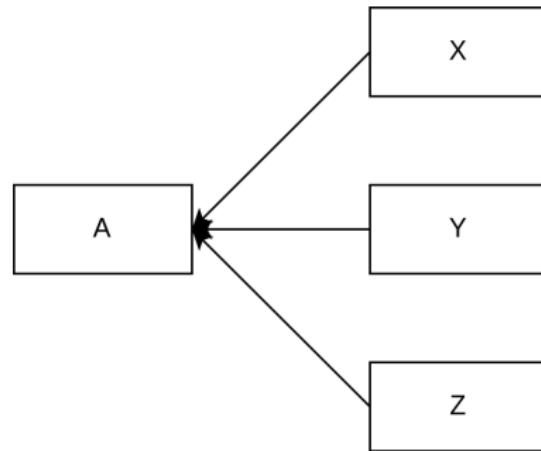
- $A$  est “abstractness”.
- $Ca$  est le nombre de classe(s) abstraite(s).
- $Cc$  est le nombre de classe(s) concrète(s).

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 :



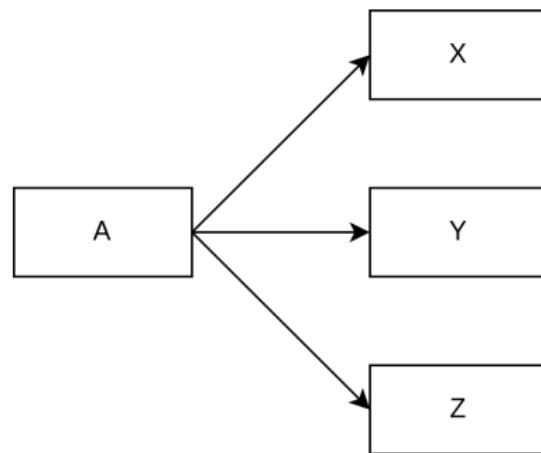
Ici  $C_a$  est 3.

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.

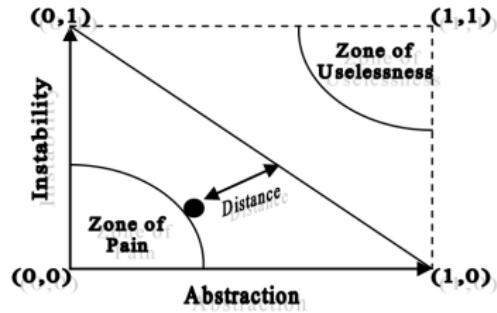
# Les mesures du couplage

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

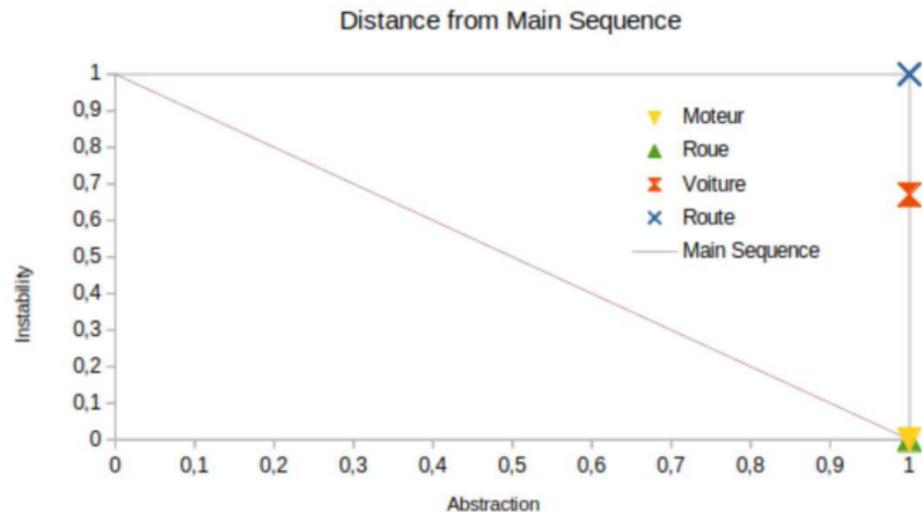
Calculer les valeurs de  $C_a$ ,  $C_e$ ,  $A$ ,  $I$  et  $D$  pour chaque classe du source  
[https://github.com/St-Michel-IT/architecture-application/  
blob/main/coupling.py](https://github.com/St-Michel-IT/architecture-application/blob/main/coupling.py).

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

# Les mesures du couplage

Résultat de l'exercice précédent

Classe	Ca	Ce	A	I	D
Formula	#N/A	#N/A	=SUM(B3:B6)/SUM(C3:C6)	=ROUND(C3/(C3+B3);2)	=ROUND(ABS(D3+E3-1);2)
Moteur	1	0		1	0
Roue	1	0		1	0
Voiture	1	2		1	0,67
Route	0	1		1	1
Main Sequence				0	1
Main Sequence				1	0



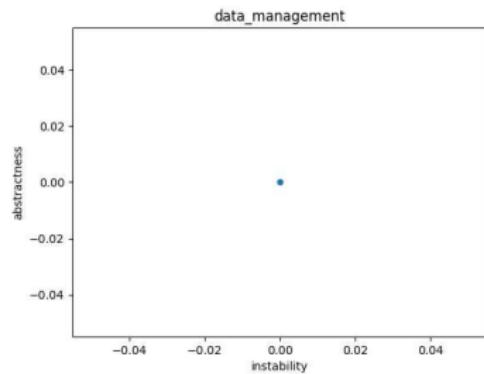
# Les mesures du couplage

## 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;
}
```

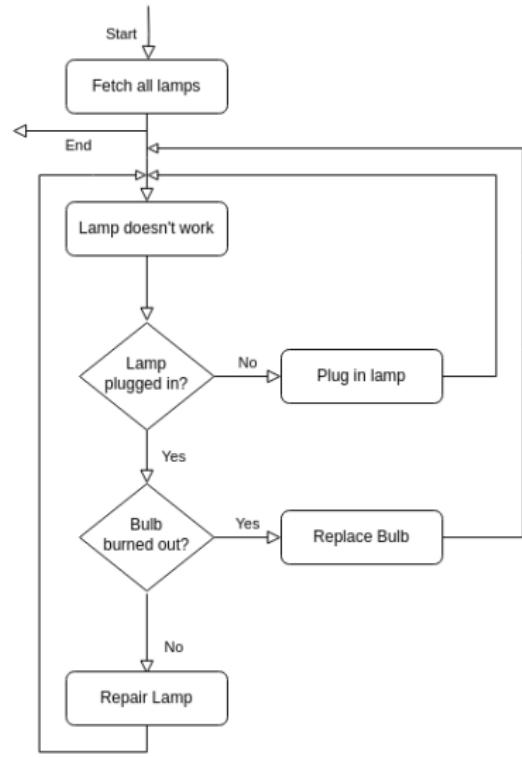
Ici  $E$  est 2 et  $N$  est 3, donc CC est 1.

# Les mesures de la complexité cyclomatique

## Exercice 4

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 5

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 à 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 :

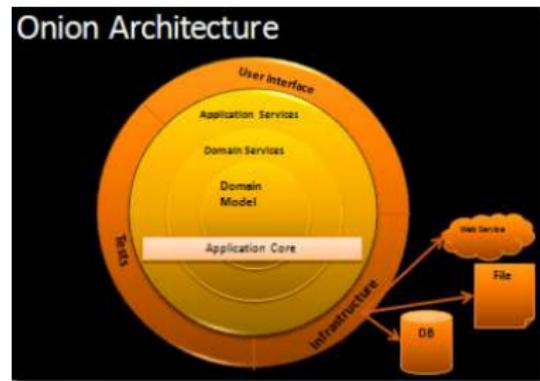
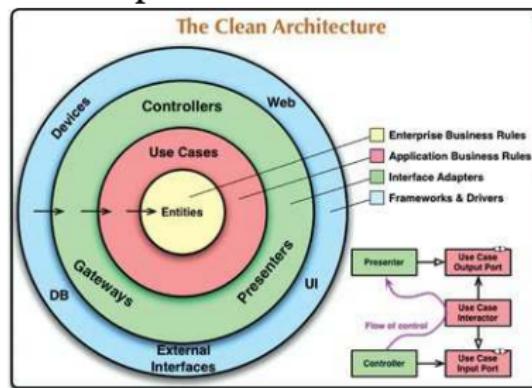
- **SOLID** : Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.
- **Hexagonal Architecture** : A.K.A Ports and Adapters.
- **Onion Architecture** : Une couche intérieur ne sait rien de celle plus extérieur.
- Many more...

# “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> :



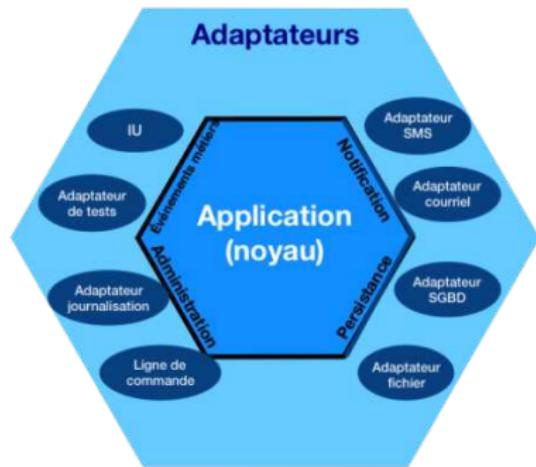
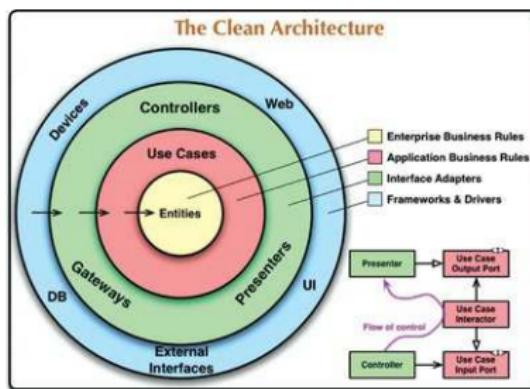
<sup>11</sup>The Onion Architecture : part 1,

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

# “Mid-level” architecture

Comment agencer les briques

De même, l'architecture hexagonale semble similaire 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>

Démarrés dans les années 80, ils sont finalisés en 2004 par Robert C. Martin. 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 de la "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 classes 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)

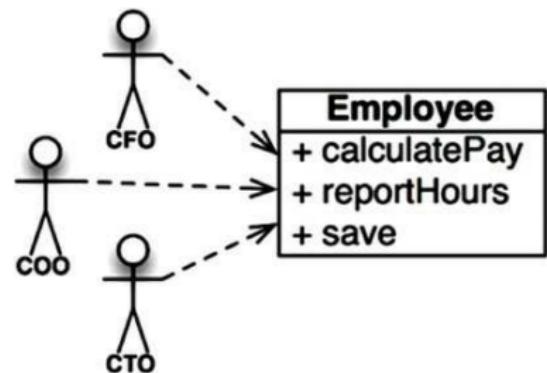
Définition<sup>1</sup>

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 un seul acteur.

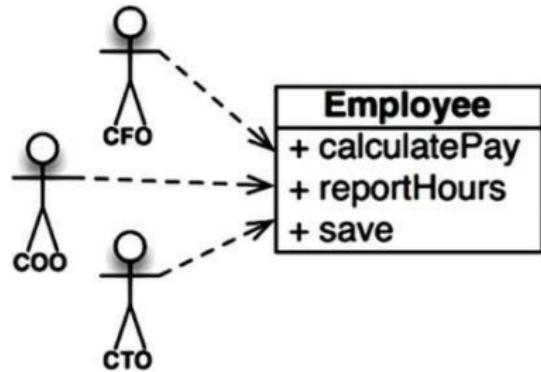


A éviter

# Single Responsibility Principle

Exercice 6 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(s) pour qu'elles respectent le SRP .
- Ajouter les attribues nécessaire(s) et au moins 4 méthodes à la classe `Employee` tout en respectant le SRP .
- Répondre sous forme d'un schéma (en utilisant Draw.io par exemple) avec les différents et classe(s), méthodes.

# Single Responsibility Principle

Exercice 7 de 30 minutes

Coder la solution de manière simpliste.

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

A quoi ressemble l'arborescence du projet ?

Committer dans un dépôt Git, le schéma issu 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'autres 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 8 de 30 minutes

Refactoriser le code

<https://github.com/St-Michel-IT/architecture-application/blob/main/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 issu de l'exercice précédent, le code et les tests.

Le travail de certains étudiants sera évalué.



# Open-Closed Principle

Exercice 9 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 10 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'implémentent pour divers pays.

Committer dans un dépôt Git, le schéma issu 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 11 de refactoring de 15 minutes

Refactoriser le code <https://github.com/St-Michel-IT/architecture-application/blob/main/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, rédiger les tests unitaires avant de coder.

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

Le travail de certains étudiants sera évalué.



# Liskov Principle

Exercice 12 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 et que `use_it` fonctionne avec les deux classes.

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

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

Le travail de certains étudiants sera évalué.

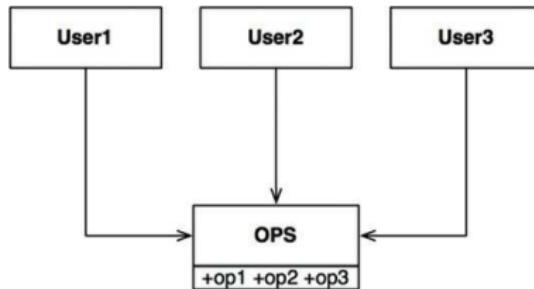


# Interface Segregation Principle (ISP)

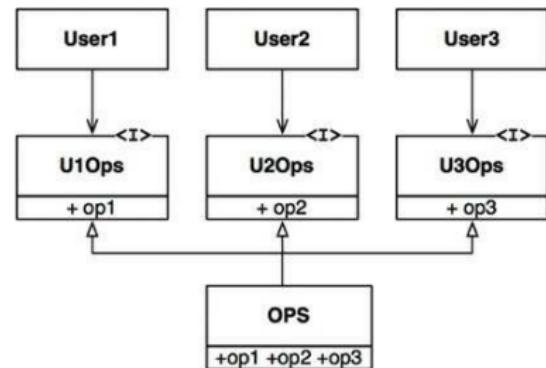
Définition<sup>1</sup>

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.



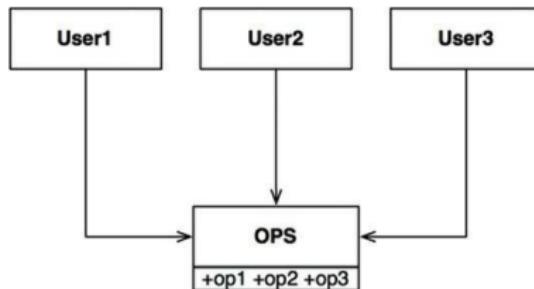
A éviter



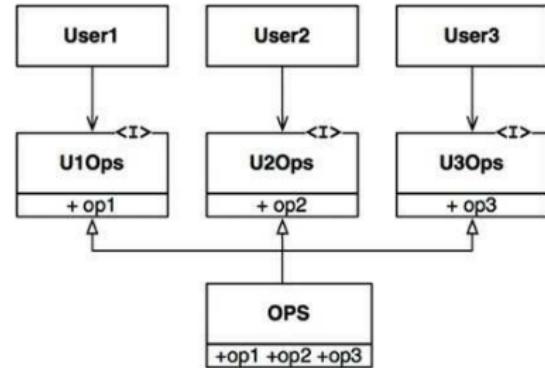
A appliquer

# 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 13 de refactoring de 30 minutes

Dans le code

<https://github.com/St-Michel-IT/architecture-application/blob/main/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 issu 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 certaines classes concrètes sont extrêmement stables comme `String` de `java.lang.string`.

# Dependency Inversion Principle

Exercice 14 de refactoring de 15 minutes

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

- il y a couplage entre la classe PowerSwitch et LightBulb. Alors qu'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, rédiger les tests unitaires avant de coder.

Committer dans un dépôt Git, le schéma issu 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”  $A$  mesurée au précédent chapitre section 5.



# Event-Driven Architecture

Définition<sup>4, 13</sup>

On parle d'Architecture Orientée Événement, ou d'Architecture Évènementielle en français. L’“Event-Driven Architecture”, comme l'architecture de service (SOA) est un style d'architecture logicielle.

Elle vise également à casser la complexité des systèmes en les découplant en plusieurs parties, à diminuer le couplage entre les composants, les dépendances entre ces derniers, en les dissociant.

En effet les composants producteurs d'événements ne ciblent pas spécifiquement un consommateur. Ils envoient l'évènement à un broker.

Un évènement est un changement d'état, pertinent pour l'applicatif, du logiciel ou du matériel...

On peut dans certaines technos prioriser les événements pour faciliter le passage à l'échelle.

---

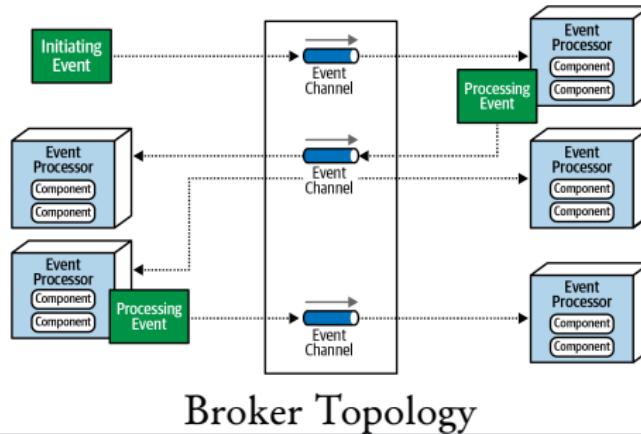
<sup>13</sup>]Une architecture orientée événements, qu'est-ce que c'est ?, <https://www.redhat.com/fr/topics/integration/what-is-event-driven-architecture>

# Event-Driven Architecture

Définition<sup>4, 13</sup>

On distingue donc 3 types de composants principaux :

- **Producteur d'événements** : Il génère les événements.
- **Broker** : Il reçoit les événements et les distribue aux consommateurs.
- **Consommateur d'événements** : Il traite les événements du “topic” auquel il est abonné. Garder en tête que les consommateurs peuvent être des producteurs également (probablement pas du même type d'event sinon on boucle...).



# Event-Driven Architecture

Définition<sup>4, 13</sup>

Ces événements sont aussi appelés “messages”.

Et le broker est souvent appelé “message broker” ou Message Queue (MQ). Autres synonymes concernant les consommateurs, on parle parfois d’“event processor”. De pub-sub ou de publish-subscribe.

Tout comme les architectures de services, les consommateurs et producteurs de messages sont suffisamment indépendants pour être développés dans des langages différents et déployés indépendamment.

Quelques exemples de message broker open source :

- **Apache Kafka** écrit en Java et ses clients en Go, .NET, Python, C/C++.
- **RabbitMQ** avec son serveur développé en Erlang et ses clients en Erlang, Java et C#.
- **IBM MQ** en Java ou Go et ses clients dans la plupart des langages de programmation modernes.

# Event-Driven Architecture

## Caractéristiques<sup>4</sup>

La communication des events peut être synchrone pour plus de fiabilité, avec un acknowledgement de la part du consommateur.

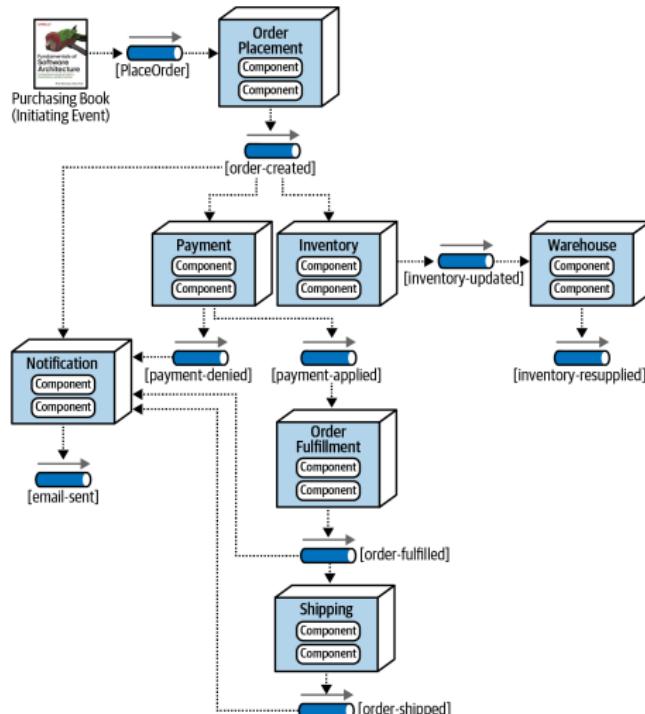
Elle peut être asynchrone pour plus de rapidité, on parle parfois de “fire-and-forget”.

En cas de crash, pour éviter la perte d'événement, la MQ peut être configurée pour sauvegarder les messages de manière permanente sur disque. Les messages peuvent être commités en base.

Perdre des messages c'est dangereux mais traiter deux fois ou plus le même message peut être dangereux aussi. Dans le cas d'une opération financière par exemple, un virement peut être effectué deux fois.

# Event-Driven Architecture

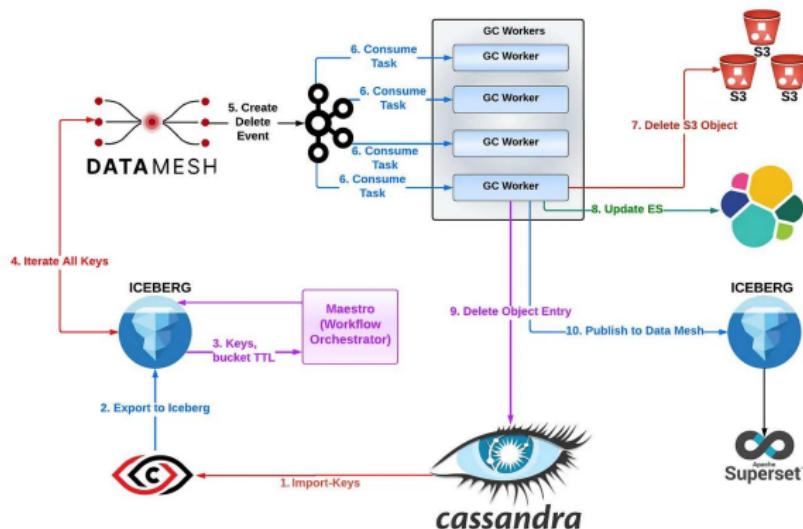
Exemple d'une marketplace<sup>4</sup>



Achat d'un livre sur l'Architecture logicielle

# Event-Driven Architecture

Exemple de Netflix<sup>14</sup>

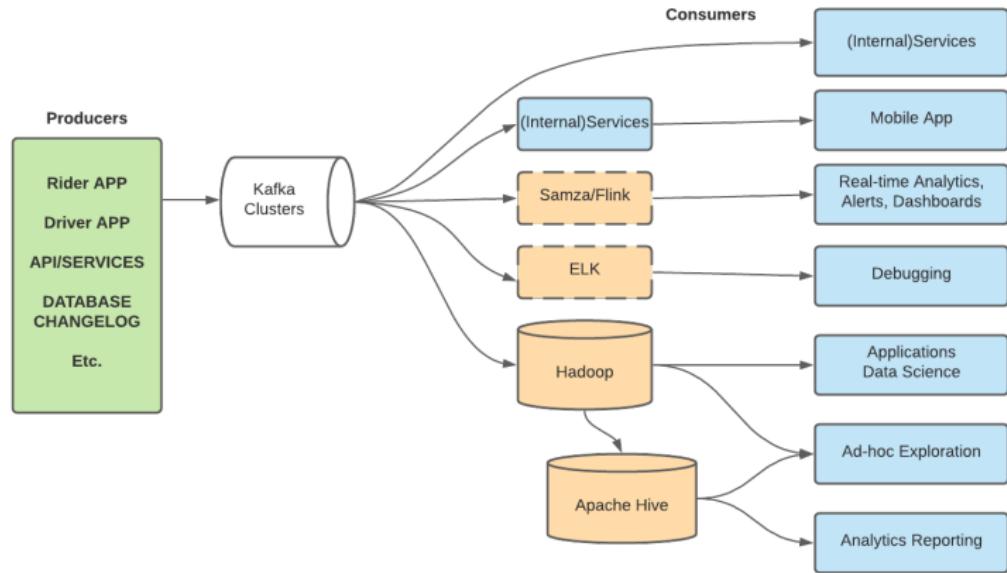


## High-Level Architecture

<sup>14</sup>Navigating the Netflix Data Deluge: The Imperative of Effective Data Management,  
<https://netflixtechblog.medium.com/navigating-the-netflix-data-deluge-the-imperative-of-effective-data-management>

# Event-Driven Architecture

Exemple de Uber<sup>15</sup>



## Kafka Ecosystem at Uber

<sup>15</sup>Enabling Seamless Kafka Async Queuing with Consumer Proxy, <https://www.uber.com/en-FR/blog/kafka-async-queuing-with-consumer-proxy/>

# Event-Driven Architecture

Le protocol MQTT<sup>16</sup>

Inventé en 1999 par IBM, MQTT signifie “Message Queuing Telemetry Transport”.

C'est un protocol de messagerie léger, standardisé par OASIS et ISO, basé sur le principe de pub/sub. Orienté vers les applications IOT et les réseaux à faible bande passante.

Quel est l'intérêt d'un standard ?

---

<sup>16</sup>MQTT, <https://mqtt.org/>

# Event-Driven Architecture

## Le protocol MQTT<sup>16</sup>

Inventé en 1999 par IBM, MQTT signifie “Message Queuing Telemetry Transport”.

C'est un protocol de messagerie léger, standardisé par OASIS et ISO, basé sur le principe de pub/sub. Orienté vers les applications IOT et les réseaux à faible bande passante.

Quel est l'intérêt d'un standard ?

L'interopérabilité entre de multiples implémentations de ce standard. Et l'indépendance vis-à-vis des fournisseurs. La plupart des cloud providers proposent des services MQTT .

---

<sup>16</sup>MQTT, <https://mqtt.org/>

# High Level Desgin (HLD)

Définition<sup>17</sup>

Le HLD est une description de l'architecture, une vue d'ensemble d'un système qui peut être un produit entier.

On peut entre autre utiliser plusieurs diagrammes pour décrire le HLD :

- UML avec le diagramme de séquence.
- UML avec le diagramme d'activité.
- UML avec “Use case”.
- Diagramme “Data Flow”.

Le HLD est une vue d'ensemble, il ne rentre donc pas dans les détails. Ces derniers sont dans les diagrammes de la mid-level architecture.

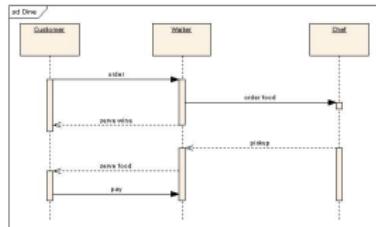
---

<sup>17</sup>Carnegie Mellon University, Philip Koopman,

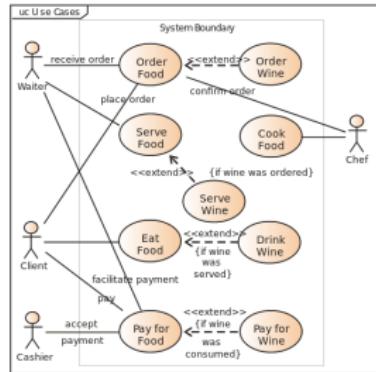
[https://course.ece.cmu.edu/~ece642/lectures/12\\_Architecture\\_HLD.pdf](https://course.ece.cmu.edu/~ece642/lectures/12_Architecture_HLD.pdf)

# High Level Desgin (HLD)

Exemples de diagrammes



UML, diagramme de séquence



UML, diagramme “Use case”

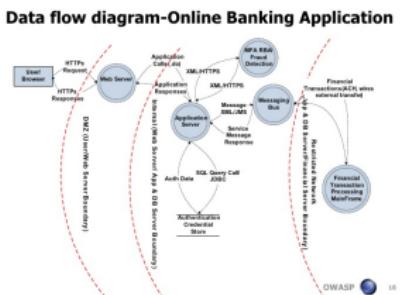
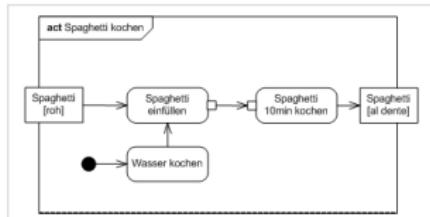


Diagramme Data Flow



UML, diagramme d'activité

# High Level Desgin (HLD)

## Bonnes pratiques

- C'est un ou plusieurs diagrammes
- S'applique à tous les composants du système, soft, hard, réseau, etc.
- S'applique à toutes les interfaces entre ces composants.
- Évolue avec les spécifications.
- Ne comprend pas trop de détails qui compliquent la compréhension.  
D'autres diagrammes sont là pour ça.
- Utiliser des outils dédiés, comme Draw.io ou PlantUML pour les diagrammes UML .

# Architectures Orientées Service

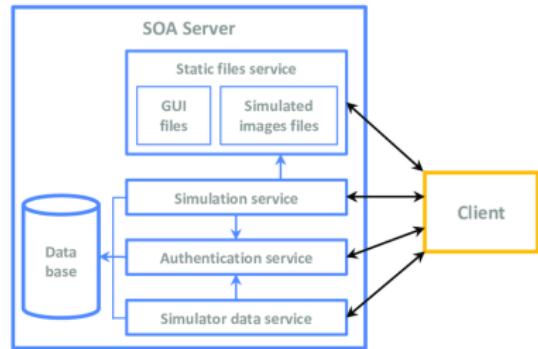
## Définition<sup>4</sup>

“Service-Oriented Architecture” (SOA) en anglais, est un style d’architecture logicielle qui vise à concevoir des applications comme un ensemble de services.

Vu de l’extérieur, un service expose une interface qui permet d’interagir avec lui. Il y a de plusieurs services, chacun dédié à une tâche spécifique. Ils communiquent tous avec la même base de données.

Cette une approche considérée comme l’opposé de l’approche “monolithique”.

Le client peut-être un navigateur web, une application mobile, un autre service, etc.



# Architectures Orientées Service

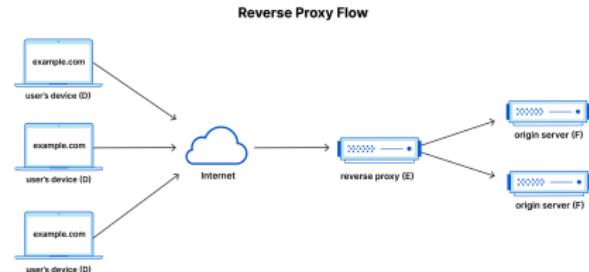
## Définition<sup>4</sup>

Cette approche est hybride entre le monolithique et les microservices.

Elle est considérée comme plus souple que le monolithique et plus simple que les microservices.

La meilleure flexibilité vient entre autre du fait que les services peuvent être développés dans des technologies différentes. Ils peuvent être déployés indépendamment.

Un “Reverse Proxy” peut-être utilisé pour rediriger les requêtes vers le bon service tout en donnant l'impression de n'avoir qu'un seul serveur.



# Architectures Orientées Service

Définition<sup>4</sup>

Quel est l'intérêt d'avoir une ou des bases de données communes à tous les services ?

# Architectures Orientées Service

## Définition<sup>4</sup>

Quel est l'intérêt d'avoir une ou des bases de données communes à tous les services ?

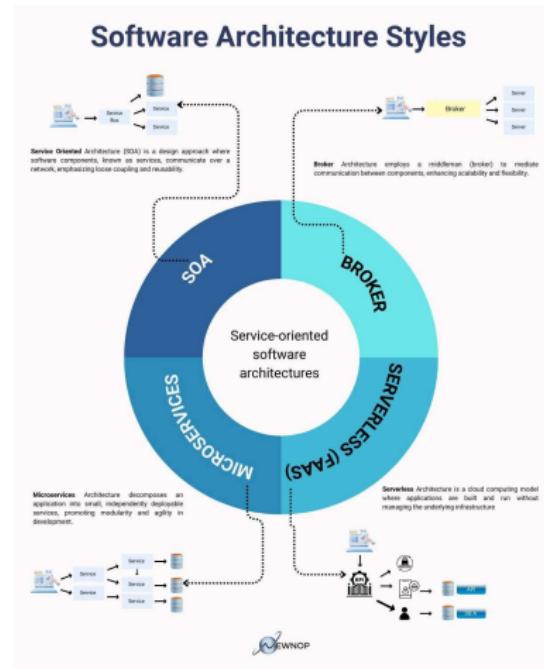
- Éviter la redondance de données.
- Éviter les incohérences de données.
- Éviter les problèmes de synchronisation.
- Éviter les problèmes de sécurité.
- Éviter un impact sur les performances lié à la duplication et à la com'.

■ SOA n'est pas un standard ou un protocole, c'est un style d'architecture.

# Introduction aux Architectures Orientées Services

## Introduction aux Architectures Orientées Services<sup>18</sup>

Plusieurs variantes de SOA ont vues le jour :



<sup>18</sup>A Guide to Service-Oriented Architecture (SOA) Styles, [https://www.newnop.com/blog/a-guide-to-service-oriented-architecture-\(soa\)-styles](https://www.newnop.com/blog/a-guide-to-service-oriented-architecture-(soa)-styles)

# Introduction aux Architectures Orientées Services

## Introduction aux Architectures Orientées Services<sup>18</sup>

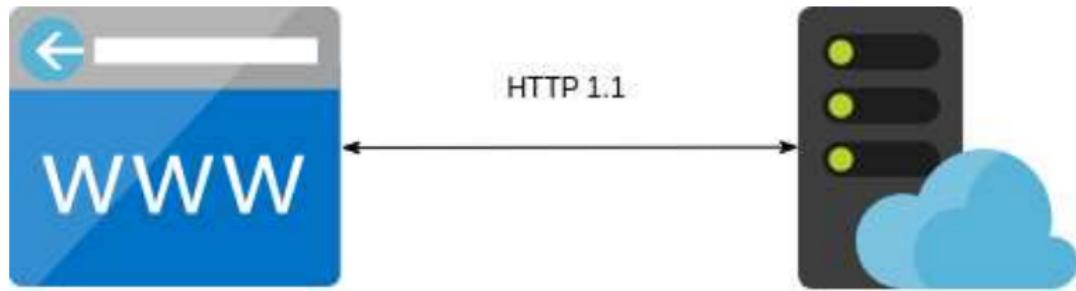
- **broker** : En s'appuyant sur les fondations de l'architecture orientée services (SOA), le style Broker introduit une entité centrale appelée "broker" pour gérer la communication entre différents services. Imaginez ce "broker" comme un contrôleur de trafic expérimenté, routant habilement les messages et assurant un flux de communication fluide.
- **Microservice** : En poussant le concept de modularité à un niveau supérieur, l'architecture des microservices est devenue un choix populaire pour la construction d'applications logicielles modernes. Ce style préconise de décomposer les applications en services encore plus petits, indépendants et autonomes. Pensez à ces services comme à des microcosmes individuels au sein de l'écosystème applicatif plus large, chacun accomplissant une tâche spécifique.
- **Serverless** : le style architectural "serverless", une approche native du cloud qui redéfinit la manière dont les applications sont construites et déployées. Au lieu de provisionner et de gérer des serveurs, les développeurs écrivent des fragments de code appelés « fonctions » qui sont déclenchés par des événements spécifiques. La gestion d'un maximum de l'infrastructure est laissée à une équipe dédiée - c'est l'essence du "serverless". Quel(s) produit(s) cloud "serverless" réputé connaissez-vous ?

Quels sont les avantages et inconvénients de ces styles ?

# Les architectures Client-Serveur

## Les protocoles entre navigateur et serveur

Si le client est un navigateur web, les protocoles seront le HTTP et les protocoles construits au dessus.

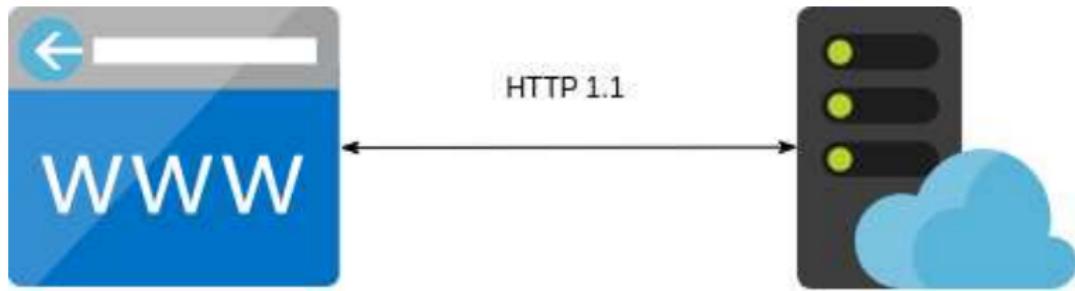


Que peut-on échanger au dessus de HTTP entre le navigateur et le serveur ?

# Les architectures Client-Serveur

## Les protocoles entre navigateur et serveur

Si le client est un navigateur web, les protocoles seront le HTTP et les protocoles construits au dessus.



Que peut-on échanger au dessus de HTTP entre le navigateur et le serveur ?

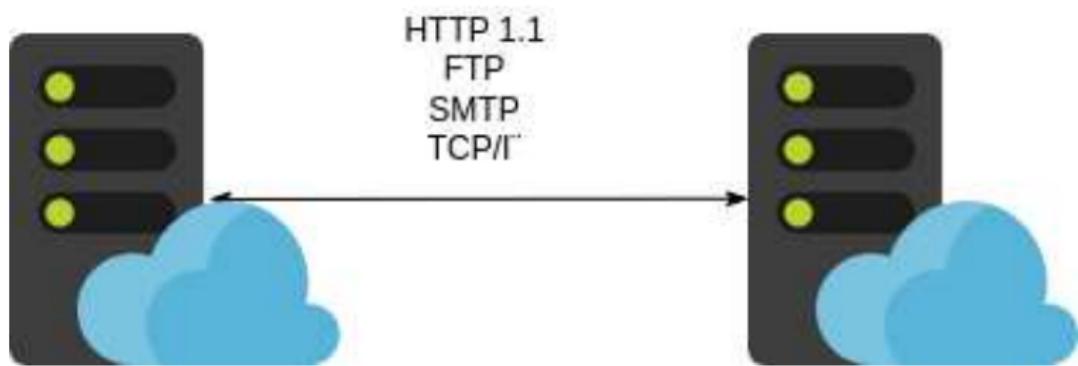
Des données brutes aux formats JSON, XML, HTML, etc.

Potentiellement dans un autre protocole comme JSON-RPC, OpenID Connect, SOAP over HTTP, REST, etc.

# Les architectures Client-Serveur

## Les protocoles entre serveurs

Si le client est une autre machine, beaucoup plus de protocoles sont disponibles.

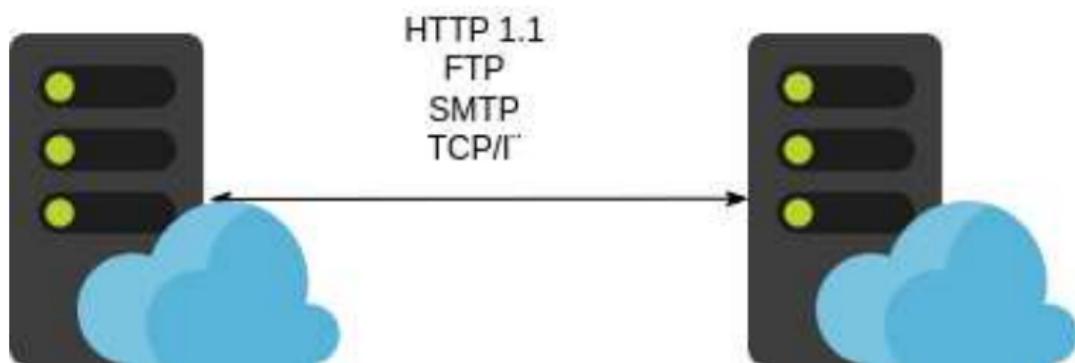


Que peut-on échanger au dessus de HTTP entre serveurs ?

# Les architectures Client-Serveur

## Les protocoles entre serveurs

Si le client est une autre machine, beaucoup plus de protocoles sont disponibles.



Que peut-on échanger au dessus de HTTP entre serveurs ?

Des données brutes aux formats JSON, XML, etc.

Potentiellement dans un autre protocole comme SOAP over SMTP, SOAP over FTP, SOAP over HTTP, etc.

# Appel de procédure

## Définition

Couramment appelé RPC pour “Remote Procedure Call” en anglais.

Ensemble de protocoles qui permettent à un programme client de demander à un serveur d'exécuter une fonction/procédure/programme. Le retour de cette procédure, i.e., la réponse est renvoyée au client.

Les premières standardisations dans une RFC datent de 1976.

C'est globalement une communication inter-process.

Ces appels à des méthodes et les réponses en retour sont souvent sérialisés en JSON ou XML . Ces données sérialisées sont ensuite envoyées en HTTP, le plus souvent.

Fort de leur standardisation, on trouve leur implémentation côté client et serveur dans tous les langages de programmation courants.

# Appel de procédure

1998 le XML-RPC<sup>19</sup>

Cette appel de procédure distant utilise le HTTP pour le transport et le XML pour la sérialisation.

Exemple d'appel :

```
<?xml version="1.0"?>
<methodCall>
<methodName>Teststatut</methodName>
<params>
<param>
<value><i4>10</i4></value>
</param>
</params>
</methodCall>
```

Exemple de réponse :

```
<?xml version="1.0"?>
<methodResponse>
<params>
<param>
<value><string>Statut : OK</string></value>
</param>
</params>
</methodResponse>
```

---

<sup>19</sup>Simple cross-platform distributed computing, based on the standards of the Internet.,  
<http://1998.xmlrpc.com/>

# Appel de procédure

1998, le XML-RPC

Quel est l'avantage du XML-RPC ?

---

<sup>20</sup>Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures.,  
<https://datatracker.ietf.org/doc/html/rfc8610>

# Appel de procédure

1998, le XML-RPC

Quel est l'avantage du XML-RPC ?

Les XML ont une grammaire vérifiable grâce à un schéma XSD . Le JSON de base n'a pas de grammaire. Il faut définir un schéma JSON avec CDDL<sup>20</sup> par exemple.

Quel est l'inconvénient du XML par rapport au JSON ?

---

<sup>20</sup>Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures.,  
<https://datatracker.ietf.org/doc/html/rfc8610>

# Appel de procédure

1998, le XML-RPC

Quel est l'avantage du XML-RPC ?

Les XML ont une grammaire vérifiable grâce à un schéma XSD . Le JSON de base n'a pas de grammaire. Il faut définir un schéma JSON avec CDDL<sup>20</sup> par exemple.

Quel est l'inconvénient du XML par rapport au JSON ?

Le XML est plus verbeux que le JSON, donc plus difficile à lire et plus long à transférer.

“Rappel” : On peut parser du XML en JS dans le navigateur

[https://developer.mozilla.org/en-US/docs/Web/XML/Parsing\\_and\\_serializing\\_XML](https://developer.mozilla.org/en-US/docs/Web/XML/Parsing_and_serializing_XML)

---

<sup>20</sup>Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures.,  
<https://datatracker.ietf.org/doc/html/rfc8610>

# Appel de procédure

Le JSON-RPC 1 et 2<sup>22</sup>

Le JSON se popularise au milieu des années 2000 mais est standardisé seulement en 2013<sup>21</sup>.

La dernière version de JSON-RPC, la 2.0, celle à utiliser, date de 2013 également.

Exemple d'appels et réponses :

```
// Avec des "named parameters"
{"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3} // Appel -->
{"jsonrpc": "2.0", "result": 19, "id": 3} // Réponse <--
// Équivalent aux "positional parameters"
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1} // Appel -->
{"jsonrpc": "2.0", "result": 19, "id": 1} // Réponse <--
// Une notification n'a pas de "id" et donc pas de réponse
{"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]} // Appel -->
{"jsonrpc": "2.0", "method": "foobar"} // Appel -->
// Batch de commandes, les mêmes commandes sont dans un tableau, possible pour les notifications également
[{"jsonrpc": "2.0", "method": "sum", "params": [1,2,4], "id": "1"}, {"jsonrpc": "2.0", "result": 19, "id": "2"}] // Appel -->
[{"jsonrpc": "2.0", "result": 7, "id": "1"}, {"jsonrpc": "2.0", "result": 19, "id": "2"}] // Réponse <--
```

<sup>21</sup>The JavaScript Object Notation (JSON) Data Interchange Format,

<https://www.rfc-editor.org/rfc/rfc7158>

<sup>22</sup>JSON-RPC, A light weight remote procedure call protocol. It is designed to be simple !,

<https://www.jsonrpc.org/>

# Appel de procédure

## JSON-RPC

Quel sont les avantages du JSON-RPC ?

# Appel de procédure

## JSON-RPC

Quel sont les avantages du JSON-RPC ?

- Le JSON est plus léger que le XML, donc plus vite transféré.
- Le JSON est plus facilement lisible que le XML .
- Facile à implémenter en JS dans le navigateur ou à utiliser avec une librairie.

Quel est l'inconvénient du JSON par rapport au XML ?

# Appel de procédure

## JSON-RPC

Quel sont les avantages du JSON-RPC ?

- Le JSON est plus léger que le XML, donc plus vite transféré.
- Le JSON est plus facilement lisible que le XML .
- Facile à implémenter en JS dans le navigateur ou à utiliser avec une librairie.

Quel est l'inconvénient du JSON par rapport au XML ?

Pas de schéma du JSON de la charge utile!

Voir l'exemple de serveur qui [expose le BLE sur le HTTP over JSON-RPC](#) .

# Appel de procédure

gRPC<sup>23</sup>

gRPC est un RPC open source développé par Google qui l'a libéré en 2015.

Il est basé sur le protocole HTTP/2 et utilise le format de sérialisation de données Protobuf A.K.A Protocol Buffer. Protobuf est un format binaire défini par un schéma.

Pourquoi un format binaire ?

```
>>> from sys import getsizeof # La taille en octets d'un objet
>>> from struct import pack " Struct pour parser ou générer du binaire
>>> print("Compare with string")
>>> struct_object = pack(">H", 0xffff) # H == uInt16, unsigned short
>>> print(0xffff)
>>> print("lentgh of the string:", getsizeof("65535"))
>>> print("lentgh of the struct:", getsizeof(struct_object))
Compare with string
65535
lentgh of the string: 54
lentgh of the struct: 35
```

<sup>23</sup>gRPC, A high performance, open source universal RPC framework,

<https://grpc.io/>

# Appel de procédure

Pourquoi Protobuf dans le RPC

Un schéma réduit les bugs possibles.

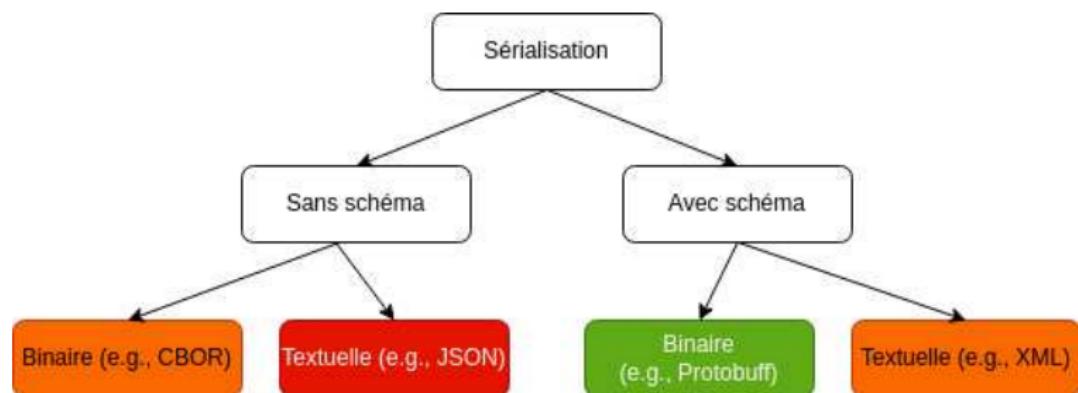
- Comme par exemple un format de donnée incohérent, une balise ou une clé en JSON qui ne respecte pas la hiérarchie attendue.
- Type de donnée incohérent
- Valeur incohérente, comme 6000000 dans un champ qui attend un entier 16 bits.
- Sûrement d'autres...

Toutes les erreurs ci-dessus vont induire un crash lors de la sérialisation des données. Les données erronées ne peuvent donc pas être envoyées au client ou au serveur. D'où la diminution des bugs possibles.

# Appel de procédure

Pourquoi Protobuf dans le RPC

Rappel sur la sérialisation des données, les grandes familles :



Échelle des problèmes potentiels du vert au rouge

# Appel de procédure

## gRPC

Quels sont les avantages du gRPC ?

# Appel de procédure

## gRPC

Quel sont les avantages du gRPC ?

- Un schéma de données qui réduit les bugs possibles.
- Le HTTP/2 est plus rapide que le HTTP/1.1.
- Le format binaire est plus léger que le JSON .

Quel sont les inconvénients du gRPC ?

# Appel de procédure

## gRPC

Quel sont les avantages du gRPC ?

- Un schéma de données qui réduit les bugs possibles.
- Le HTTP/2 est plus rapide que le HTTP/1.1.
- Le format binaire est plus léger que le JSON .

Quel sont les inconvénients du gRPC ?

- Il y a des étapes supplémentaires d'encodage et de décodage des données.
- Plus difficile à utiliser sans librairie et intégrer dans le navigateur.
- Impossible de lire la charge utile sans décoder.

# Appel de procédure

## Conclusion

Les 3 types d'appels de procédure vus sont très standardisés. Les bibliothèques pour les implémenter sont disponibles dans tous les langages de programmation courants. Souvent depuis des années et donc robustes.

On peut les intégrer dans de nombreux systèmes, serveurs, navigateurs, applications mobiles, etc.

L'`id` dans la requête et la réponse permet d'envoyer un batch de requêtes et d'y associer les réponses.

Ils sont reconnus pour leur simplicité d'utilisation, c'est presque aussi simple d'appeler une méthode sur une machine distante qu'en local. Vu comme cela on comprends par contre qu'il y a un très fort couplage entre le client et le serveur. Et donc un manque de flexibilité.

# API REST

## Définition<sup>24</sup>

Les API REST aussi parfois appelées API RESTful sont des API qui respectent les contraintes de l'architecture REST (Representational State Transfer) de Roy Fielding.

Ces contraintes sont :

- Client-serveur : Le client et le serveur sont séparés.
- Stateless : Chaque requête du client doit contenir toutes les informations nécessaires pour être traitée par le serveur.
- Cacheable : Les réponses doivent être explicitement ou implicitement marquées comme cacheable ou non.
- Layered system : Le client ne doit pas savoir si il communique avec le serveur ou un intermédiaire.
- Code on demand (optional) : Le serveur peut envoyer du code au client pour être exécuté.
- Une interface uniforme entre les composants qui permet un transfert standardisé des informations : Les ressources sont identifiées par des URI, les opérations sont standardisées (GET, POST, PUT, DELETE, etc), les réponses sont auto-descriptives.

<sup>24</sup>Une API REST, qu'est-ce que c'est ?,

<https://www.redhat.com/fr/topics/api/what-is-a-rest-api>

# API

## Définition

Aucune contrainte donc sur le langage de programmation, le système d'exploitation, le type de donnée (HTML, XML, protobuf, YAML mais plus souvent JSON), etc.

De nombreuses libraries existent pour implémenter des API REST dans tous les langages de programmation courants.

Par exemple dans un serveur Flask avec [Flask-RESTful](#) :

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class Hello(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(Hello, '/hello')

if __name__ == '__main__':
    app.run(debug=True)
```

# API

Quelques exemples d'API REST avec des POST et aussi des arguments grâce à la librairie webargs sur leur [GitHub](#).

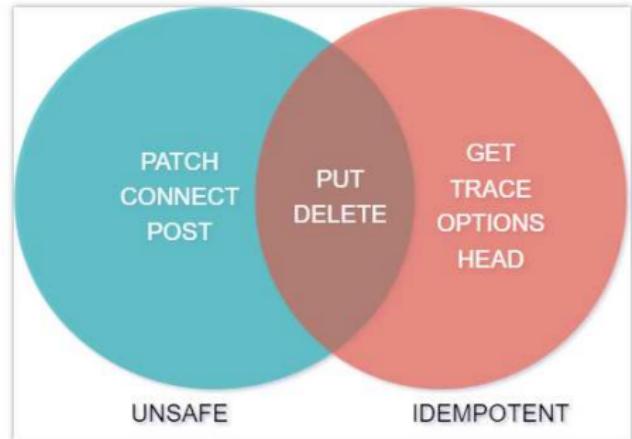
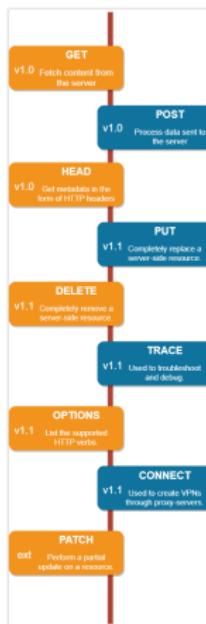
```
class DateAddResource(Resource):
    dateadd_args = {
        "value": fields.Date(required=False),
        "addend": fields.Int(required=True, validate=validate.Range(min=1)),
        "unit": fields.Str(
            load_default="days", validate=validate.OneOf(["minutes", "days"])
    },
}

@use_kwargs(dateadd_args)
def post(self, value, addend, unit):
    """A date adder endpoint."""
    value = value or dt.datetime.utcnow()
    if unit == "minutes":
        delta = dt.timedelta(minutes=addend)
    else:
        delta = dt.timedelta(days=addend)
    result = value + delta
    return {"result": result.isoformat()}

# This error handler is necessary for usage with Flask-RESTful
@parser.error_handler
def handle_request_parsing_error(err, req, schema, *, error_status_code, error_headers):
    """webargs error handler that uses Flask-RESTful's abort function to return
    a JSON error response to the client.
    """
    abort(error_status_code, errors=err.messages)
```

# Guideline API REST

## Les méthodes<sup>25</sup>



<sup>25</sup>HTTP request methods, <https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/HTTP-methods>

# Guideline API REST

La pagination dans les URLs<sup>26</sup>

La plus classique est la pagination par “offset” et “limit” :

```
GET /api/v1/users?offset=10&limit=10
```

Il existe aussi la “Page based” avec les paramètres “page” et “size” :

```
GET /api/v1/users?page=2&size=10
```

D’autres méthodes existent, comme la pagination par “cursor” ou “keyset” mais semblent anecdotiques.

Le succès de la pagination avec “offset” et “limit” est peut-être dû à sa simplicité et à sa facilité d’implémentation. Il suffit passer ces paramètres à une requête SQL .

---

<sup>26</sup>A guide to REST API pagination,  
<https://www.merge.dev/blog/rest-api-pagination>

# Guideline API REST

## Les URLs<sup>27</sup>

- Utilisez uniquement des lettres minuscules dans les URLs des API RESTful.
- Pour les espaces, utilisez kebab-case, et non snake\_case ou des espaces.
- Construisez les URIs avec des noms, pas des verbes.
- Utilisez les méthodes HTTP appropriées pour effectuer une opération.
- Les appels d'API REST qui retournent une collection doivent être au pluriel.
- Une URL qui retourne un résultat unique doit être au singulier.
- N'incluez pas d'extensions de fichier.
- Utilisez les en-têtes pour garder les URIs propres.
- N'identifiez pas les opérations de création, lecture, mise à jour et suppression (CRUD) dans l'URL .
- Structurez les URIs librement selon la hiérarchie de votre modèle de données.
- Utilisez des paramètres de requête pour le filtrage et la recherche.
- Ne dévoilez pas le fonctionnement interne de votre architecture.
- Faites des URLs courtes, intuitives et lisibles.
- Protégez-vous contre les injections SQL .
- Incluez la version de l'API REST à la base de l'URI .

---

<sup>27</sup>Top REST API URL naming convention standards, <https://www.theserverside.com/video/Top-REST-API-URL-naming-convention-standards>

# Guideline API REST

## Les URLs

Donner un exemple d'URL pour les cas suivants :

- Récupérer la liste des utilisateurs.
- Récupérer un utilisateur.
- Créer un utilisateur.
- Mettre à jour un utilisateur.
- Supprimer un utilisateur.
- Récupérer 20 commentaires d'un utilisateur.
- Récupérer 20 commentaires d'un utilisateur, la 3eme page.
- Rechercher les utilisateurs dont le nom contient "John".

# Guideline API REST

## Les URLs

Donner un exemple d'URL pour les cas suivants :

- Récupérer la liste des utilisateurs.
- Récupérer un utilisateur.
- Créer un utilisateur.
- Mettre à jour un utilisateur.
- Supprimer un utilisateur.
- Récupérer 20 commentaires d'un utilisateur.
- Récupérer 20 commentaires d'un utilisateur, la 3eme page.
- Rechercher les utilisateurs dont le nom contient "John".

Corriger les URLs :

- GET /api/v1/user/1/update
- GET /api/v1/user/1/delete

# Guideline API REST

## Les URLs

Donner un exemple d'URL pour les cas suivants :

- Récupérer la liste des utilisateurs.
- Récupérer un utilisateur.
- Créer un utilisateur.
- Mettre à jour un utilisateur.
- Supprimer un utilisateur.
- Récupérer 20 commentaires d'un utilisateur.
- Récupérer 20 commentaires d'un utilisateur, la 3eme page.
- Rechercher les utilisateurs dont le nom contient "John".

Corriger les URLs :

- GET /api/v1/user/1/update
- GET /api/v1/user/1/delete

Qu'est-ce qu'une injection SQL et comment éviter une injection SQL ?

# Guideline API REST

Les codes de retour HTTP<sup>28</sup>

## Codes retour HTTP :

- Niveau 100 (Informationnel) – le serveur accuse réception d'une requête.
- Niveau 200 (Succès) – le serveur a complété la requête comme prévu.
- Niveau 300 (Redirection) – le client doit effectuer des actions supplémentaires pour compléter la requête.
- Niveau 400 (Erreur client) – le client a envoyé une requête invalide.
- Niveau 500 (Erreur serveur) – le serveur n'a pas pu exécuter une requête valide en raison d'une erreur du serveur.

## Par exemple, les classiques :

- 400 Requête incorrecte – le client a envoyé une requête invalide, comme l'absence du corps de la requête ou d'un paramètre requis
- 401 Non autorisé – le client n'a pas réussi à s'authentifier auprès du serveur
- 403 Interdit – le client est authentifié mais n'a pas la permission d'accéder à la ressource demandée
- 404 Non trouvé – la ressource demandée n'existe pas
- 412 Précondition échouée – une ou plusieurs conditions dans les champs d'en-tête de la requête se sont avérées fausses
- 500 Erreur interne du serveur – une erreur générique s'est produite sur le serveur
- 503 Service indisponible – le service demandé n'est pas disponible

---

<sup>28</sup>Best Practices for REST API Error Handling,

<https://www.baeldung.com/rest-api-error-handling-best-practices>

# Swagger

La spécification OpenAPI et le format Swagger

La spécification OpenAPI est une spécification pour décrire des API JSON .  
Donc les API RESTful entre autre.

Cette définition au format JSON ou YAML, dans un fichier appelé “Swagger”.

Grâce à ce fichier de définition et des outils open source comme `swagger-codegen`, on peut générer en local ou dans le CI :

- La documentation de l’API .
- Les clients pour les langages de programmation courants.
- Les serveurs pour les langages de programmation courants.
- Les tests.
- Une web UI pour tester l’API .
- Many more...

# Swagger

La spécification OpenAPI et le format Swagger

Pour écrire le “Swagger” directement dans le code, il existe des librairies comme `flasgger` qui permettent de générer la documentation de l’API à partir des docstrings des méthodes.

Voir par exemple <https://github.com/flasgger/flasgger>.

On remarque qu’on ne peut intégrer les deux libraries `Flask-RESTful` et `flasgger` ensemble dans un même serveur Flask. Il faut écrire la définition dans un fichier “Swagger” à part, “à l’ancienne”.

La standardisation c'est propre mais c'est compliqué...

# JSON-LD

JSON Linked-Data<sup>30</sup>

Inspiré du web sémantique et de la sémantique des données, le JSON-LD est un format de données structurées qui permet de lier des données entre elles.  
“JSON-LD is a lightweight Linked Data format. It is easy for humans to read and write.”

Exemple de retour d'une API JSON-LD :

```
{  
  "@context": "https://json-ld.org-contexts/person.jsonld",  
  "@id": "http://dbpedia.org/resource/John_Lennon",  
  "name": "John Lennon",  
  "born": "1940-10-09",  
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"  
}
```

Le @contex est une URL qui pointe vers un fichier JSON-LD qui définit les relations entre les données<sup>29</sup>.

---

<sup>29</sup>JSON-LD contexts,

<http://niem.github.io/json/reference/json-ld/context/>

<sup>30</sup>JSON for Linking Data, <https://json-ld.org/>

# JSON-LD

## La pagination<sup>31</sup>

Une “Collection” de données peut être paginée pour ne pas surcharger le client avec une trop grosse réponse :

```
{  
  "@id": "http://api.example.com/an-issue/comments?whatever=3",  
  "@type": "Collection",  
  "first": "/an-issue/comments",  
  "previous": "/an-issue/comments?whatever=2",  
  "next": "/an-issue/comments?whatever=4",  
  "last": "/an-issue/comments?whatever=498",  
  "member": [ ... ]  
}
```

---

<sup>31</sup>Pagination, <https://www.w3.org/community/hydra/wiki/Pagination>

# API SOAP

## Définition

SOAP pour “Simple Object Access Protocol” est un protocole de communication qui permet à des applications de communiquer entre elles. Il est basé sur XML et est donc plus verbeux que le JSON . Le plus utilisé est over HTTP mais il peut également être utilisé over SMTP, FTP, etc.

La WSDL pour “Web Services Description Language” est un fichier XML qui décrit les méthodes, les types de données, les protocoles de communication, etc.

```
<message name="getTermRequest">
<part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
<part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
<operation name="getTerm">
<input message="getTermRequest"/>
<output message="getTermResponse"/>
</operation>
</portType>
```

# API SOAP

## Définition<sup>32</sup>

Les standards de conformité intégrés incluent la sécurité, l'atomicité, la cohérence, l'isolement et la durabilité (ACID), un ensemble de propriétés qui permet d'assurer des transactions de base de données fiables. Voici les principales spécifications de ces services web :

- **WS-Security (Web Services Security)** : spécification qui standardise la manière dont les messages sont sécurisés et transférés via des identifiants uniques appelés jetons.
- **WS-ReliableMessaging** : spécification qui standardise la gestion des erreurs entre les messages transférés par le biais d'une infrastructure informatique non fiable.
- **WS-Addressing (Web Services Addressing)** : spécification qui ajoute les informations de routage des paquets en tant que métadonnées dans des en-têtes SOAP, au lieu de les conserver plus en profondeur dans le réseau.
- **WSDL (Web Services Description Language)** : décrit la fonction d'un service web ainsi que ses limites.

---

<sup>32</sup>REST et SOAP : quelle est la différence ?, <https://www.redhat.com/fr/topics/integration/whats-the-difference-between-soap-rest>

# API SOAP

## La requête

Des librairies existent pour générer les requêtes SOAP, par exemple en Python zeep :

```
from zeep import Client
from requests import Session
from zeep.transports import Transport

wsdl = "https://wsvc.cdiscount.com/MarketplaceAPIService.svc?wsdl"

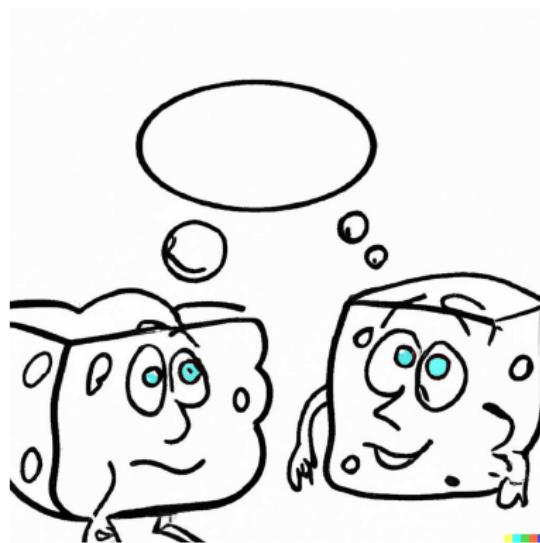
session = Session()
session.auth = HTTPBasicAuth(<username>, <password>

#An additional argument 'transport' is passed with the authentication details
client = Client(wsdl, transport=Transport(session=session))
request_data = {"account": 1, "new_balance": 100}
response = client.service.sendData(**request_data)
```

# API SOAP

## Le serveur

Des librairies existent pour transformer un serveur web Python classique en API SOAP, par exemple `spyne`. Voir leur exemple avec le module [https://github.com/arskom/spyne/blob/master/examples/helloworld\\_soap.py](https://github.com/arskom/spyne/blob/master/examples/helloworld_soap.py).



# SOA

## Exercice 15

Désigner et développer une architecture de services qui distribue les données de l'[Index EgaPro](#). Respecter les exigences suivantes :

- Travailler par groupe de 3 maximum.
- Créer un repo GitHub dédié et le communiquer à l'enseignant.
- Chaque étudiant sera évalué selon ses commits sur le repo. Il doit développer au moins un service.
- Documenter l'architecture avec un diagramme de séquence.
- Documenter l'API REST avec un Swagger.
- Documenter l'API SOAP avec un WSDL .
- Intégrer ces schémas à un README détaillé.
- Développer 3 services minimum :
  - Distribution des données EgaPro en RPC .
  - Distribution des données EgaPro par API REST .
  - Distribution des données EgaPro par API SOAP .
- Tous doivent avoir une base de donnée commune.
- Distribuer les données par SIREN suffit, voir  
[https://github.com/St-Michel-IT/testing/blob/master/egapro\\_api.py](https://github.com/St-Michel-IT/testing/blob/master/egapro_api.py).
- Configurer un Reverse Proxy (Nginx par exemple) pour avoir tous les services derrière le même domaine et port.

Le dernier commit de la journée sera évalué.

# Conclusion

## L'architecture, un Art

L'étymologie d'architecture semble ne pas avoir de rapport l'art mais plutôt *arkhè*, le commencement.

On constate bien que ce n'est pas une "science exacte". Beaucoup de concepts similaires reviennent dans différents styles d'architecture sous des appellations différentes. De nombreux termes sont utilisés de manière interchangeable, ce qui ne facilite pas la communication. C'est bien un art, dans le monde du logiciel également.

C'est le monde du compromis, aucune application complexe ne permettra de suivre à la lettre toutes les recommandations d'architecture.

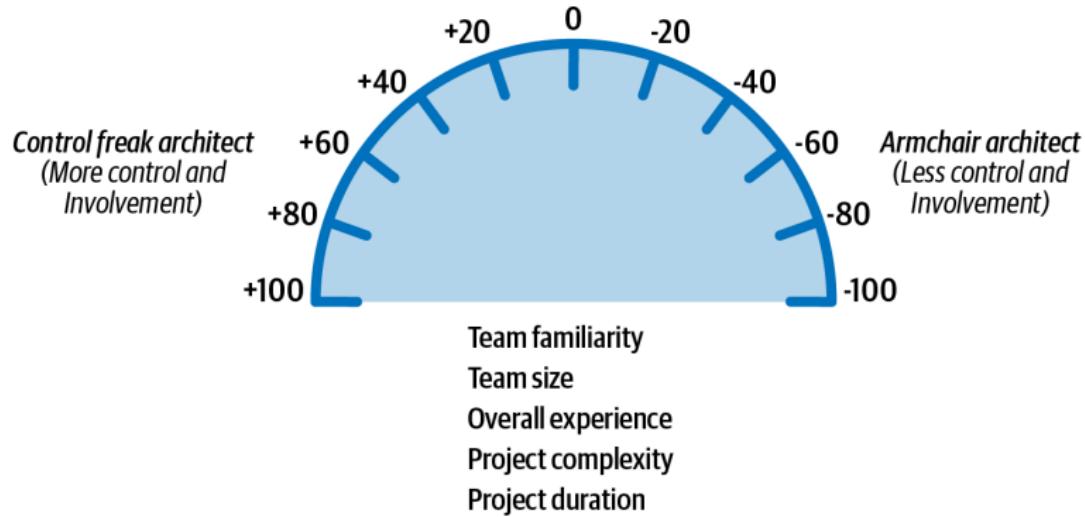


# Conclusion

## L'architecture en équipe<sup>4</sup>

L'humain est beaucoup plus présent dans l'architecture logicielle que dans le développement.

Le rôle d'architecture implique souvent du management et de la communication.



# Conclusion

## L'architecture en équipe

Si vous n'aimez pas communiquer, partager vos choix, peut-être que l'architecture n'est pas faite pour vous.

La communication doit se faire avec de nombreuses parties prenantes :

- Les développeurs : doivent comprendre les choix et les implémenter correctement.
- Les managers : doivent être convaincus des choix pour les défendre.
- Les specifiers : partager les business rules et les contraintes.
- Many more...

C'est un art mais pour argumenter un choix, les données chiffrées restent la meilleure arme !

Sortir des KPI sur les temps de maintenance, les coûts de développement, le nombre de bugs, etc.

# Conclusion

Le mot de la fin

Ne pas essayer de paraître malin en utilisant des termes et des technos compliqués. Même si c'est à la mode en ce moment...

K.I.S.S. “Keep It Simple, Stupid” !

