

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

Architectures d'Applications

Compétence acquise au cours des 3 jours du module

Compétences :

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

Architectures d'Applications

Compétence acquise au cours des 3 jours du module

Compétences :

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

Reformulation :

- 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. 2 architectures agnostiques à une application et une orientée web REST ou SOAP.
- 25 % sur une évaluation écrite finale

Intervenant sur le module Architecture d'Application

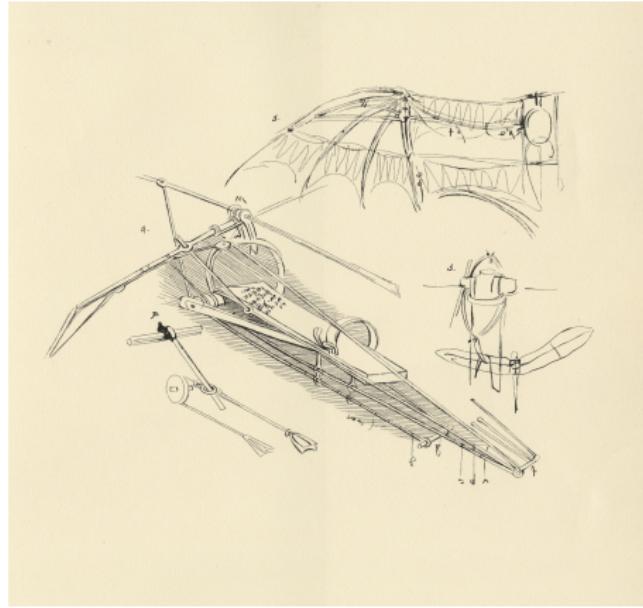
Christophe Brun, conseil en développement informatique

- 1^{ère} 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.”¹



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

¹Clean Architecture, Robert C. Martin

Eloge de la simplicité

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

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 ?).

²La 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.¹ Le “Design patterns”³ est un classique en développement logiciel.

“The onlye way to go fast is to go well.”¹

“The blueprint of the system.”⁴

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

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

⁴Fundamentals of Software Architecture, Mark Richards et Neal Ford

Définition

Le couplage

C'est un des termes les plus importants en Architecture logicielle. Il faut le comprendre 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”⁵

⁵Robert 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 briques¹

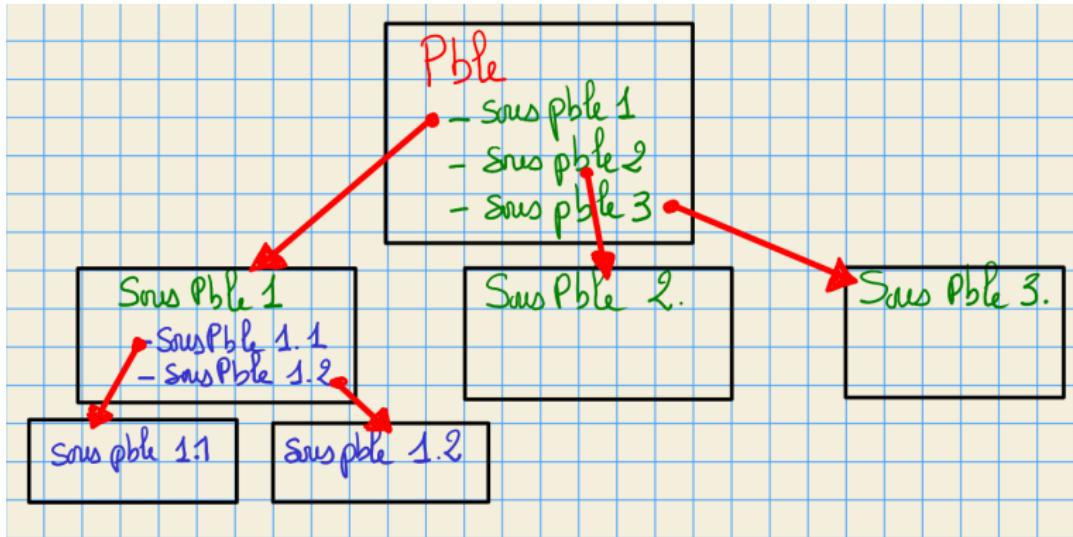
- Programmation structurée (sous-ensemble de la programmation impérative), où la brique est le module de code, i.e., de programmes et sous-programmes. Il contient les instructions de control de flux (if, else, for, while, etc.).
- Programmation orientée objet, où la brique est la classe.
- Programmation 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-programmes⁶.

⁶ Programmation structurée, https://perso.univ-lyon1.fr/marc.buffat/COURS/BOOK_INPROS_HTML/CHAP5/COURS_ALGORITHMIQUE.html

Les briques

Exemple de programmation fonctionnelle¹

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 autre ont des outils orientés programmation fonctionnelle. Souvent nommées fonctions dites 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”⁷ :

- `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
```

⁷Functional Programming Modules

Les briques

Exemple de programmation orientée objet¹

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., 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 pigeaoon 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 de 5 minutes

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

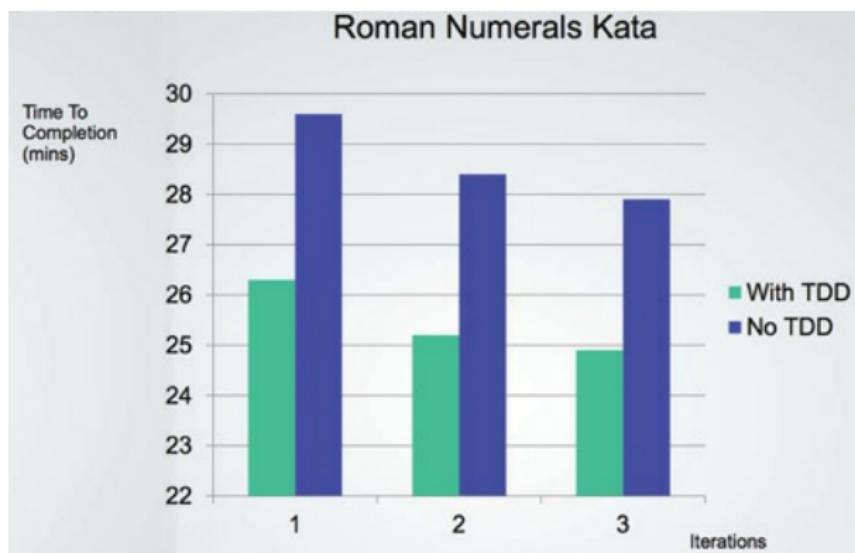
- <https://github.com/facebook/Haxl/>
- <https://github.com/gurupratap-matharu/Bike-Rental-System/blob/master/main.py>
- <https://github.com/papit-fr/fsma/>

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 projet¹ !
Entre autre, les tests unitaires font réduire la taille des classes, functions et
des méthodes ce qui améliore architecture.

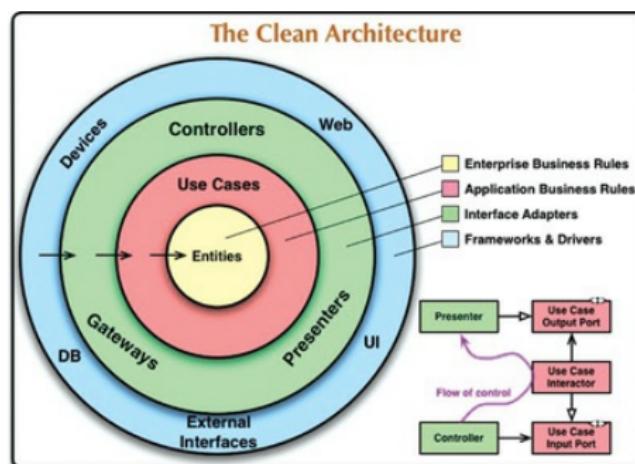


Le TDD

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 périphériques¹ :



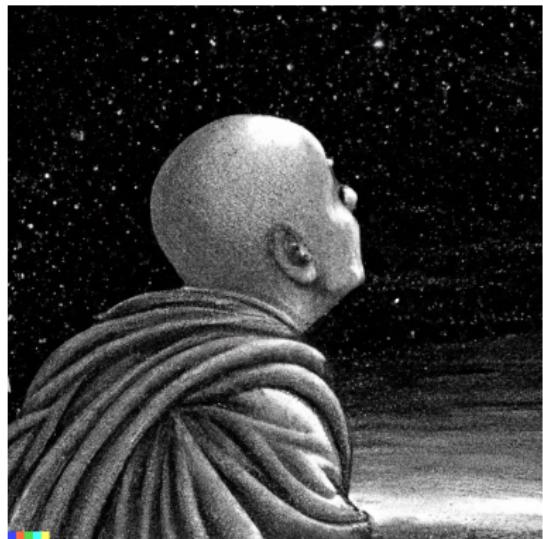
Le TDD

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.



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 mesurer l'architecture 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 des CK (Chidamber & Kemerer) metrics qui peuvent ressembler, etc.

Les mesures du couplage

L’“abstractness” A^4

“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 :

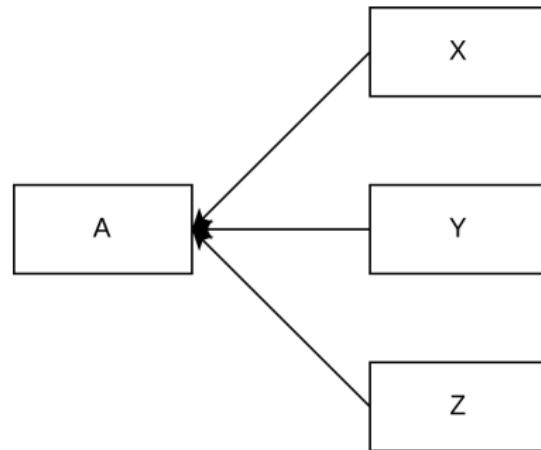
- A est “abstractness”
- Ca est le nombre de classes abstraites
- Cc est le nombre de classes concrètes.

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 classes qui dépendent de la classe A :



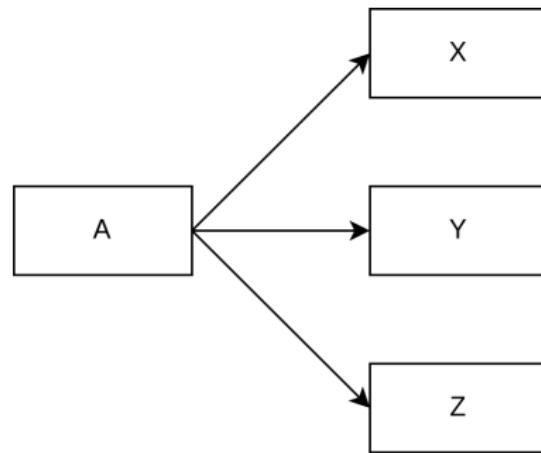
Ici C_a est 3.

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 classes dont dépend de 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.

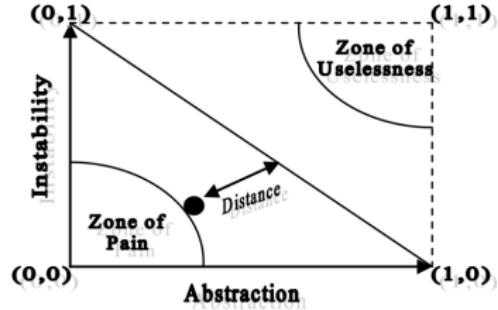
Les mesures du couplage

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.

⁸ 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

Les mesures du couplage

Exercice de 30 minutes

Calculer les valeurs de C_a , C_e , A , I et D pour les 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).

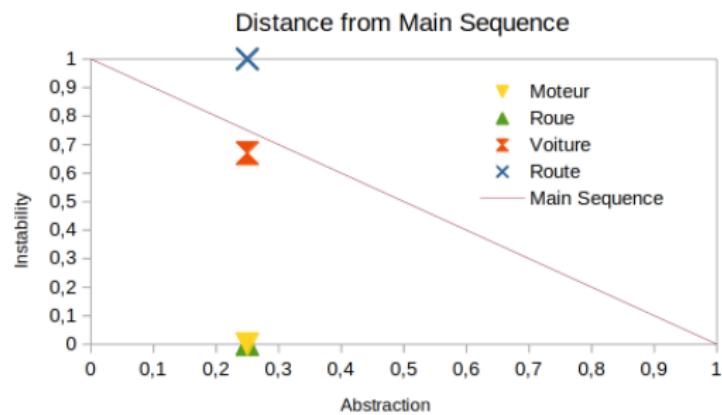
Restituez le résultat dans un tableau Excel et un graphique avec les D et la droite “main sequence”.

Les mesures du couplage

Exercice 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	



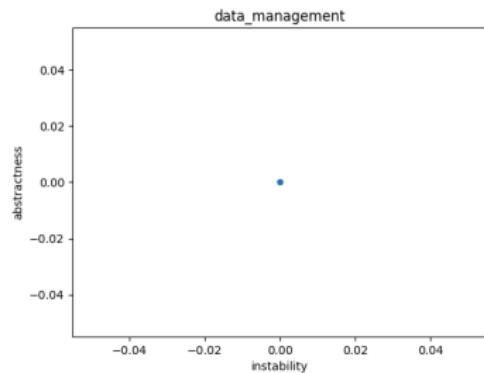
Les mesures du couplage

Les outils

Certains outils comme JCAT⁹ en Java ou module_coupling_metrics¹⁰ 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 est ne calcule le couplage qu'entre modules.

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

¹⁰https://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 complexity du code.

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

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

Ou :

- 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`, du graphe de contrôle de flux.

Les valeurs au-dessous de 5 sont bonnes, au-dessus de 10 il y a danger⁴.

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.