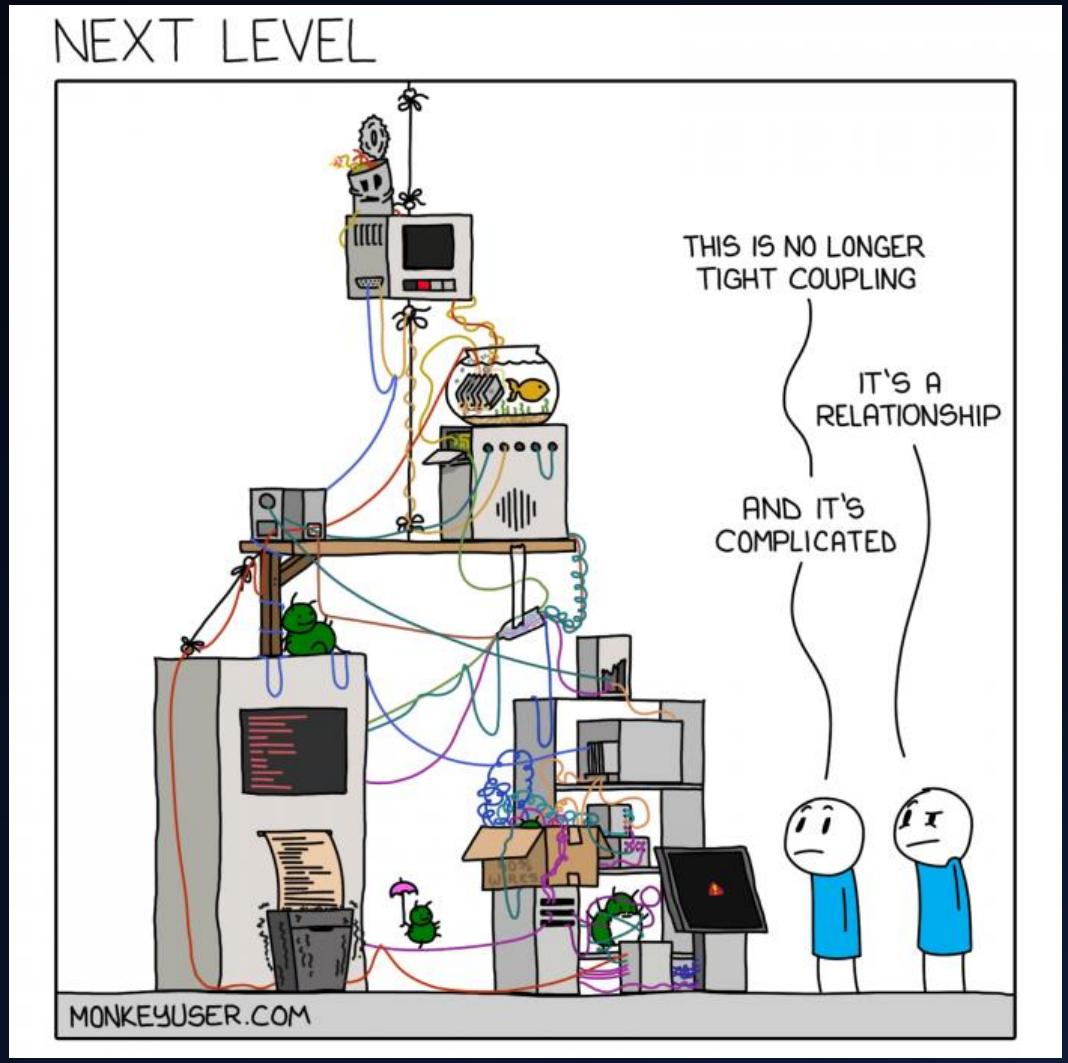
Introduction aux principes SOLID

ECRIRE DU CODE MAINTENABLE DE QUALITÉ

Les défis de la POO et pièges courants

- **Rigidité**
- **Fragilité**
- **Immobilité**



Les défis de la POO et pièges courants

- **Rigidité** : La difficulté de changer un logiciel sans introduire de bugs ailleurs, car les classes sont trop couplées.
- **Fragilité** : Les changements provoquent des régressions dans des parties apparemment non liées du système.
- **Immobilité** : L'incapacité de réutiliser le code d'un projet dans un autre en raison d'un couplage excessif ou de dépendances complexes.

Les principes SOLID :

- **S** - Le Principe de Responsabilité Unique :
  Un module fait une seule chose.
- **O** - Le Principe Ouvert/Fermé :
  On ferme la porte à la modification, on ouvre la porte à l'extension
- **L** - Le Principe de Substitution de Liskov
  On n'utilise pas différemment les classes mères et enfant.
- **I** - Le Principe de Ségrégation des Interfaces
  Les interfaces doivent être le plus petites possibles
- **D** - Le Principe d'Inversion de Dépendance
  Les modules haut-niveau ne doivent pas dépendre des modules bas niveau

Les principes SOLID en détail

COMMENT LES APPLIQUER ?

S - PRINCIPE DE RESPONSABILITÉ UNIQUE (SRP)

Qu’est-ce qu’une responsabilité ?

C’est une raison pour laquelle une classe peut changer.

On va chercher à limiter leur nombre dans une classe et dans chaque fonction, au plus proche de 1.

Exemple de non respect du principe

La cuirchette (spork)

Comment détecter si une classe viole le principe ?

- On se pose la question de ce qu'elle fait :
  Si on utilise les mots « et », « ou », « sauf si »…
- Elle a plusieurs niveaux d'abstraction :
  Mélange de règles métier et de concepts techniques
- Elle peut avoir besoin de changer pour plusieurs raisons

O – OPEN-CLOSED PRINCIPLE (OCP)

Pourquoi ouvert et fermé ?

Ouverts à l'extension, mais fermés à la modification.

Autrement dit, il devrait être possible d'ajouter de nouvelles fonctionnalités à un système sans modifier le code existant.

Ca permet d’ajouter du code à application sans risquer d'introduire des erreurs dans le code déjà testé et validé.

Exemple de non respect du principe

Dans un Puzzle, chaque pièce a une forme unique, elle peut seulement être remplacée par une pièce de forme identique.

Si on la perd ou si on veut la modifier, il faut changer toutes les pièces autour.

Exemple de respect du principe

Dans le TP sur le calcul de formes, c'est facile de calculer la somme des aires de toutes les formes instanciées, si elles héritent d'une classe abstraite « Forme ».

On peut ajouter facilement de nouvelles classes qui héritent de Forme pour les ajouter au calcul.

Comment détecter si du code viole le principe ?

- Elle contient des gros if/elseif, ou des switch
- Aucune classe abstraite ou interface dans le projet
- Absence de polymorphisme

L – LISKOV SUBSTITUTION PRINCIPLE (LSP)

De quoi s’agit-il ?

Le Principe de substitution de Liskov stipule qu'un objet d'une classe dérivée doit pouvoir être substitué par un objet de sa classe de base sans altérer les propriétés désirables du programme.

Les sous-classes doivent être capables de remplacer leurs superclasses sans affecter le fonctionnement du programme.

Exemple de non respect du principe

Si on considère que les membres de la classe Oiseau doivent implémenter la méthode « voler », alors faire hériter la classe Pingouin de la classe Oiseau viole le principe. (Le pingouin ne vole pas, il nage)

De même, faire hériter Carré de Rectangle viole le principe.

Exemple de non respect du principe

```
class Square : public Rectangle
{
public:
    Square(int side): Rectangle(side, side){}
    void setLength (int newLength)
    {
        this->length = newLength;
        this->width = newLength;
    }

    void setWidth (int newWidth)
    {
        this->length = newWidth;
        this->width = newWidth;
    }
};
```

```
int main()
{
    Rectangle square(2,3);

    square.setLength(3);
    square.setWidth(4);

    cout << square.computeSurface() << endl;
}
```

Comment détecter si du code viole le principe ?

- les sous-classes remplacent des méthodes et modifient le comportement prévu ou introduisent des effets de bord.
- Il faut absolument vérifier le type concret d’une variable pour garantir le bon fonctionnement du programme.

I – INTERFACE SEGREGATION PRINCIPLE (ISP)

De quoi s’agit-il ?

Le Principe de ségrégation des interfaces stipule que les interfaces doivent être aussi petites que possible. C’est-à-dire qu’elles ne doivent contenir que les méthodes qui doivent absolument être appelées ensemble pour remplir une fonction.

Ainsi, on ne devrait jamais avoir une méthode non implémentée dans une classe qui implémente l’interface.

Exemple de non respect du principe

```
class IMachine {
public:
    virtual void print(Document& doc) = 0;
    virtual void scan(Document& doc) = 0;
};
```

Exemple de non respect du principe

```
class IMachine {
public:
    virtual void print(Document& doc) = 0;
    virtual void scan(Document& doc) = 0;
};
```

Exemple de respect du principe

```
class IPrinter {
public:
    virtual void print(Document& doc) = 0;
};

class IScanner {
public:
    virtual void scan(Document& doc) = 0;
};
```

Comment détecter si du code viole le principe ?

- Une sous-classe n’a pas besoin d’implémenter une des fonctions de l’interface pour fonctionner. Ou bien une méthode héritée mais jamais appelée.
- Une classe qui travaille avec une interface, mais qui n’appelle que certaines méthodes de l’interface.

D – DEPENDENCY INVERSION PRINCIPLE (DIP)

De quoi s’agit-il ?

Le principe d’inversion de dépendances stipule 2 choses :

1. Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux devraient dépendre des abstractions.

2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

De quoi s’agit-il ?

Les modules de haut niveau sont soit des classes qui contiennent les règles métier, soit des classes qui représentent des concepts abstraits. Ce sont celles qui représentent le « quoi ».

Les classes de bas niveau sont celles qui travaillent à un niveau d’abstraction plus faible. Elles sont chargées du « comment ». (Connexion filesystem, DB, etc.)

De quoi s’agit-il ?

Cela veut donc dire qu’une classe qui contient des règles métier ne doivent jamais dépendre de la manière technique dont les données sont récupérées ou sauvegardées.

Exemple de non respect du principe

- La partie du code qui applique les règles métier a besoin d'un objet qui représente la connexion à la base de données (ou l'affichage) pour être instanciée.

- Une classe qui orchestre des calculs a besoin de classes qui sont chargées de stocker la donnée (API, interactions utilisateur, filesystem, BDD.)

Exemple de respect du principe

- La partie du code qui applique les règles métier a besoin d'une interface qui est héritée par un objet qui représente une manière de récupérer des données.
- Ce n'est pas le module qui contient les règles métier qui a une référence vers le projet d'infrastructure, mais l'inverse.

Comment détecter si du code viole le principe ?

- On peut regarder le sens des dépendances entre les modules d'une solution. (Encore faut-il avoir séparé en modules les différents aspects).
- Pas d'injection de dépendances
- Couplage fort

Autres principes fondamentaux

Loi de Déméter

ON NE PARLE PAS AUX INCONNUS

De quoi s’agit-il ?

La loi de Déméter est souvent appelée le principe de « moindre connaissance ». Elle vise à encourager des couplages faibles en limitant les interactions entre les objets.

La loi dit qu'un objet ne doit communiquer qu'avec ses amis immédiats et ne doit pas "connaître" les détails internes des objets avec lesquels il interagit.

Exemple de non respect du principe

```
class Shop {
public:
    void purchase(Person* person) {
        // Accès direct à l'objet Wallet à partir de l'objet Person,
        // et ensuite modification de l'état de l'objet Wallet.
        if(person->getWallet()->getMoney() > 50) {
            person->getWallet()->setMoney(person->getWallet()->getMoney() - 50);
        }
    }
};
```

Exemple de respect du principe

```
class Shop {
public:
    void purchase(const Person& person) {
        float price = 50;
        if(person.canAfford(price)) {
            person.makePayment(price);
        }
    }
};
```

Comment détecter si du code viole le principe ?

- On a de longues chaînes d’appel.
- Plus d’un point par ligne.
- On est obligés de connaître « les voisins des voisins » pour faire un traitement ou obtenir une donnée.

Module C++

FIP 1A CNAM

Séance 4