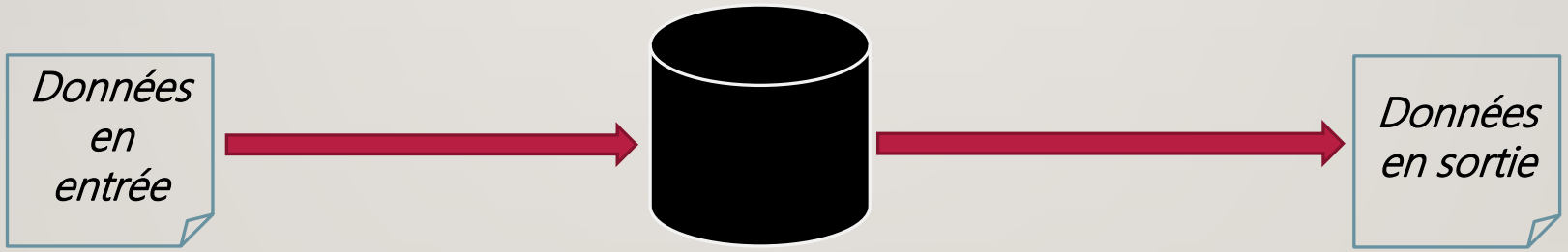
PLATEFORMES & OUTILS DE DÉVELOPPEMENT

CHAPITRE 1 : LES PRINCIPES DE CONCEPTION SOLID

Licence 3 Semestre 5, IAI – Togo (2020 – 2021)

1. LES PRINCIPES SOLID : QUE SONT-ILS ?

- Un logiciel peut être confectionné de plusieurs façons différentes
- Une même fonctionnalité peut être codée sous des designs complètement différents.
- Existence d'une architecture prédéfinie.
- Problème : le développeur est laissé à lui-même quand il faut coder.



Code Legacy

1. LES PRINCIPES SOLID : QUE SONT-ILS ?

- Inventé par Michael Feathers, blogueur et auteur du livre Working Effectively with Legacy Code
- Popularisé par Robert C. Martin (Uncle Bob), co-auteur du Manifeste Agile
  - Site Web : Clean coders
  - Auteur de livres comme Clean Code, The Clean Coder, Clean Architecture

2. SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- **Une entité logicielle (classe, méthode) ne devrait avoir qu'une seule raison de changer**
- Principe le plus connu et le plus simple à comprendre
- Que faire pour savoir si une classe respecte le SRP ?
  - Il faut dire : « La classe X fait… »
  - Si cette phrase contient des « et » et/ou des « ou », votre classe a plus d'une responsabilité
  - Si elle contient des mots génériques comme « gérer » ou « objet », alors il y a un souci.
- Par contre, il ne faut pas pousser le concept trop loin. Par exemple, une classe qui formate le numéro de téléphone d'un utilisateur peut être un peu trop spécifique.

3. OPEN/CLOSED PRINCIPLE (OCP)

- Une application doit prendre en charge les changements fréquents qui sont effectués au cours du développement et au cours de la maintenance

Exemple : De nouvelles fonctionnalités peuvent être ajoutées de temps en temps.

- En tant qu'ingénieur logiciel, nous aimerions donc réécrire les classes et les méthodes existantes aussi rarement que possible.
- Pourquoi ? Parce qu'il est un peu frustrant de réécrire une classe dont les méthodes ont déjà été testées : il va falloir refaire tous les tests.

3. OPEN/CLOSED PRINCIPLE (OCP)

- Que dit le principe ? Les entités d'un logiciel devraient être fermées aux modifications mais ouvertes à l'extension.
- Plus concrètement, la conception et l'écriture du code doivent être effectuées de manière à ce que les nouvelles fonctionnalités soient ajoutées avec le minimum de changements au niveau du code existant.
- Lorsqu'on dit qu'une classe est fermée aux modifications et ouverte à l'extension, il faut comprendre qu'il est préférable de créer une sous-classe ou d'ajouter des membres pour lui apporter une modification d'état ou de comportement que de modifier les membres existants.

3. OPEN/CLOSED PRINCIPLE (OCP)

- Donc, si vous devez modifier le corps d'une méthode, sa signature ou encore le type d'une propriété d'une classe, vous avez de très fortes chances d'être en train de briser l'OCP.
- Les principes SOLID ne sont que des principes comme les structures de données, les types de données abstraits, etc. Il faudra donc les implémenter à l'aide des design pattern
- Par exemple, l'OCP peut être mis en œuvre avec le **Strategy Pattern** ou le Template Pattern

4. LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Le LSP est un principe très simple même si son nom fait un peu peur
- Nous créons souvent des hiérarchies de classes dans nos applications (héritages)
- Le LSP a été formulé par Barbara Liskov et Jeannette Wing de la manière suivante :

> *Si q(x) est une propriété démontrable pour tout objet x de type T, alors q(y) est vraie pour tout objet y de type S tel que S est un sous-type de T.*

4. LISKOV SUBSTITUTION PRINCIPLE (LSP)

- On peut le reformuler ainsi :

  Si S est un sous-type de T, alors tout objet de type T peut être remplacé par un objet de type S sans altérer les propriétés désirables du programme concerné.

- Le LSP peut être mis en œuvre avec le Template Pattern

4. LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Problème :

Soit une classe Rectangle représentant les propriétés d'un rectangle : hauteur, largeur. On lui associe donc des accesseurs pour accéder et modifier la hauteur et la largeur librement. En postcondition, on définit la règle : la hauteur et la largeur sont librement modifiables.

Soit une classe Carré que l'on fait dériver de la classe Rectangle. En effet, en mathématiques, un carré est un rectangle. Donc, on définit naturellement la classe Carré comme sous-type de la classe Rectangle. On définit comme postcondition la règle : les « quatre côtés du carré doivent être égaux ».

5. INTERFACE SEGREGATION PRINCIPLE (ISP)

- Que dit le principe ?

  Aucun client ne devrait dépendre de méthodes qu'il n'utilise pas.

- En effet, on remarque que lorsqu'une classe implémente une interface, elle est obligée d'implémenter toutes les méthodes de l'interface même celles qu'elle n'utilise pas. On dit qu'il y a du gras dans l'interface.
- L'ISP stipule donc qu'il faut diviser les interfaces volumineuses en plus petites, plus spécifiques de sorte que les clients n'aient accès qu'aux méthodes qui les intéressent : ***interface de rôle***.

6. DEPENDENCY INVERSION PRINCIPLE (DIP)

- Quand nous développons, nous implémentons dans un premier temps les modules de bas niveau.
  Exemple : envoi de mails, connexion à la BD, parser XML/JSON, etc.
- Puis, nous implémentons les modules de haut niveau qui dépendent des modules de bas niveau.
- Ceci relève de l'intuition. Mais ce n'est pas bien.

6. DEPENDENCY INVERSION PRINCIPLE (DIP)

- Que faire lorsque nous voulons changer les modules de bas niveau ?

Par exemple si l'on veut modifier une connexion à une BD (MySQL par Oracle) ? Ou stocker dans un fichier JSON au lieu d'un fichier CSV ?

- Il faut donc une couche abstraite (interface) permettant aux modules de communiquer entre eux
- Ainsi, les modules de haut niveau ne s'appuient pas directement sur les modules de bas niveau. Mais ils dépendront tous des interfaces et devraient ignorer leurs existences mutuelles.
- Principe le plus important mais difficile à appliquer.