DORANCO

Design Patterns & Architecture Applicative

Présenté par **Ryadh HADJ MOKHNECHE**
hm_ryadh@yahoo.fr

Introduction à la Conception objet et aux Design Patterns

Ryadh HADJ MOKHNECHE

Design Patterns

- Les design patterns ("patrons de conception") : Ce sont des méthodes utilisées en développement logiciel.
- Permettent d'**optimiser**, de **clarifier** du code informatique et de le rendre **plus robuste**, en répondant principalement à une problématique posée par les choix de conceptions.

Design Patterns

- Les design patterns remplissent souvent plusieurs conditions qui respectent les principes **SOLID** (1).

(**1**) introduits par Michael Feathers et Robert C. Martin au début des années 2000

Design Patterns

- Les principes **SOLID** sont des principes qu'utilisent les développeurs lors de la conception d'application dite "agile".
- De cette manière, on peut dire d'une conception qu'elle répond aux attentes de l'évolutivité du code, i.e. qu'elle est plus facile à manipuler, maintenir etc...

- Les 5 principes de SOLID sont considérés comme des principes de **bon sens**.
- Aucun de ces principes ne nécessite une connaissance approfondie d'un langage donné.
- Ces bons sens correspondent à 3 métriques : Cohésion, couplage et Encapsulation.

1) Cohésion

- elle traduit à quel point les pièces d'un seul composant sont en relation les unes avec les autres.
- Un module est dit cohésif lorsque, au haut niveau d'abstraction, il ne fait qu'une seule et précise tâche.
- Plus un module est centré sur un seul but, plus il est cohésif.

2) Couplage :

- mesure l'interconnexion des modules.
- Deux modules sont dits couplés si une modification d'un de ces modules demande une modification dans l'autre.

3) Encapsulation :

– L'idée est d'intégrer à un objet tous les éléments nécessaires à son fonctionnement, que ce soit des fonctions ou des données.

3) Encapsulation : (suite)

- Le corolaire est qu'un objet devrait (et non pas doit, comme c'est souvent expliqué) masquer la cuisine interne de la classe, pour exposer une interface propre, de façon à ce que ses clients puissent manipuler l'objet et ses données sans avoir à connaitre le fonctionnement interne de l'objet.

Design Patterns

- **SOLID** : acronyme représentant cinq principes de bases pour la POO.
- Ces cinq principes sont destinés à apporter une ligne directrice permettant le développement de logiciel plus fiable et plus robuste.
- Ils sont considérés comme des principes de **bon sens**.

1. Single responsibility Principle (SRP)

(Principe de Responsabilité unique)

⇒ Une classe, une fonction

ou une méthode doit avoir

une et une seule responsabilité.

Ce principe se base sur les travaux de Tom DeMarco, en 1979, qui le qualifie de principe de cohésion :

Design Patterns

- Si une classe a plus d'une responsabilité, alors ces responsabilités deviennent couplées.
- Des modifications apportées à l'une des responsabilités peuvent porter atteinte ou inhiber la capacité de la classe de remplir les autres.
- Ce genre de couplage amène à des architectures fragiles qui dysfonctionnent de façon inattendue lorsqu'elles sont modifiées.

Donc, le principe, réduit à sa plus simple expression, est :

qu'une classe donnée ne doit avoir qu'une seule responsabilité, et, par conséquent, qu'elle ne doit avoir qu'une seule raison de changer.

Les avantages de cette approche sont les suivants :

- diminution de la complexité du code
- augmentation de la lisibilité de la classe
- meilleure encapsulation, et meilleure cohésion, les responsabilités étant regroupées.

2. Open/closed Principle (OCP)

(Principe Ouvert/fermé)

⇒ une entité applicative (class, fonction, module ...) doit être ouverte à l'extension, mais fermée à la modification.

Ce principe est issu d'un article de Bertrand Meyer datant de 1988.

L'idée originale de Meyer était qu'en fait :

- Une classe logicielle était un package de données, qui devait, une fois implémentée, ne plus être modifiée que pour corriger une erreur.
- Toute nouvelle fonctionnalité ne devrait, selon lui, pouvoir être rajoutée qu'en ajoutant une nouvelle classe, laquelle pouvait éventuellement hériter de la classe d'origine.

Design Patterns

De nos jours, le sens qu'on lui donne en général est celui repris dans un article de 1996 par Robert Martin.

La définition étendue qui en est donnée est la suivante. « Les modules qui se conforment au principe ouvert/ferme ont deux attributs principaux qui sont : »

1) Ils sont « ouverts pour l'extension »

⇒ Cela signifie que le comportement du module peut être étendu, que l'on peut faire se comporter ce module de façons nouvelles et différentes si les exigences de l'application sont modifiées, ou pour remplir les besoins d'une autre application.

2) Ils sont « fermés à la modification »

⇒ Le code source d'un tel module ne peut pas être modifié. Personne n'est autorisé à y apporter des modifications.

Donc, le principe, réduit à sa plus simple expression, est :

de tendre, non plus vers des objets immuables, mais vers des objets auxquels les clients pourront ajouter de nouveaux comportements sans en modifier la mécanique interne.

Objet immuable : objet dont l'état ne peut pas être modifié après sa création, à la différence d'un **objet variable**

La différence fondamentale entre les deux approches est que :

- dans un cas, on va utiliser un héritage d'implémentation, alors que

- dans l'autre cas, on va se baser sur des contrats.

Les avantages du principe OCP sont les suivants :

- Plus de flexibilité par rapport aux évolutions.
- Diminution du couplage.

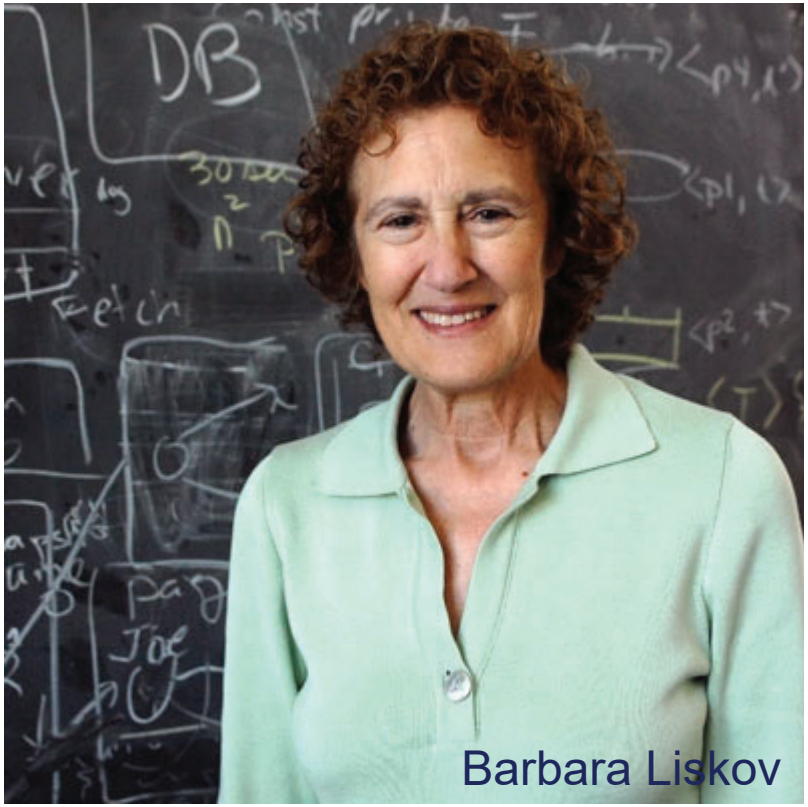
3. Liskov substitution Principle (LSP)

(Principe de Substitution de Liskov)

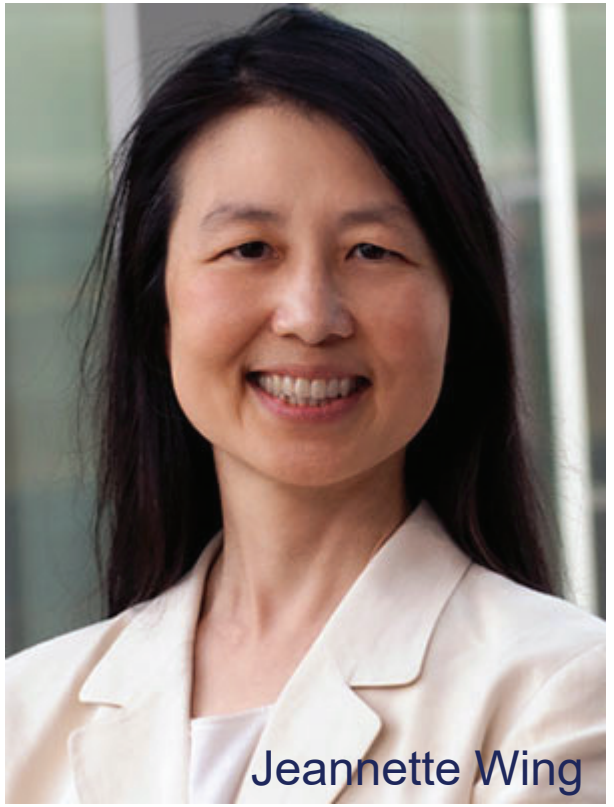
⇒ une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme.

Ce principe a été défini par Barbara Liskov et Jeannette Wing.

Design Patterns



Barbara Liskov


Jeannette Wing

Sa formulation consiste à vérifier la propriété de substitution suivante :

Si pour chaque objet o1 de type S il existe un objet o2 de type T tel que pour tout programme P défini en termes de T, le comportement de P est inchangé quand on substitue o1 à o2, alors S est un sous-type de T.

Design Patterns

À cette définition fonctionnelle légèrement alambiquée, on préfère, une fois de plus, la définition simplifiée donnée par Robert Martin :

«Les sous-types doivent être substituables à leur type de base. »

Plus simple, non ;) ?

Design Patterns

Oui mais.. On peut dire : « à partir du moment où ma classe S hérite de ma classe T », je dois pouvoir caster S en T et là ça va marcher…

Justement, non…

Le but de ce principe est exactement de pouvoir utiliser une méthode sans que cette méthode ait à connaitre la hiérarchie des classes utilisées dans l'application,

ce qui veut dire :

- pas de cast
- surtout pas d'introspection/réflexion (car si on vérifie le type d'objet, on va violer non seulement LSP, mais aussi OCP).

Les avantages du principe LSP sont les suivants :

- Augmentation de l'encapsulation.
- Diminution du couplage. En effet, LSP permet de contrôler le couplage entre les descendants d'une classe et les clients de cette classe.

4. Interface segregation Principle (ISP)

(Principe de Séparation des interfaces)

⇒ préférer plusieurs interfaces spécifiques pour chaque client (classe qui l'implémente) plutôt qu'une seule interface générale.

Autrement dit, les clients d'une entité logicielle ne doivent pas avoir à dépendre d'une interface qu'ils n'utilisent pas.

Donc, de façon générale, plus une interface est compliquée, plus l'abstraction qu'elle présente est vaste, plus elle est susceptible d'évoluer avec le temps.

Chaque évolution de l'interface va entrainer donc une livraison ou une modification de l'ensemble des clients.

Le premier effet d'avoir une interface trop compliquée va se ressentir assez vite.

En effet, toute classe implémentant une interface doit implémenter chacune de ses fonctions ⇒ On va donc très vite se retrouver avec une confusion sur le rôle des sous-classes.

Exemple : si l'interface doit changer pour un client A qui utilise la fonction A, cela va aussi impacter le client B, qui n'utilise pas la fonction A, mais qui dépend de la même interface.

Les avantages de l'application du principe ISP sont les suivants :

- Une diminution du couplage entre les classes (les classes ne dépendant plus les unes des autres).
- Les clients augmentent en robustesse.

5. Dependency inversion Principle (DIP)

(Principe de l'Inversion des dépendances)

Ce principe est secondaire, car il résulte d'une application stricte de deux autres principes à savoir les principes OCP et LSP.

Sa définition ⇒ il faut dépendre des abstractions, pas des implémentations.

Et sa définition de façon plus concise :

- Les modules de haut niveau (contenant les fonctionnalités métier) ne doivent pas dépendre des modules de bas niveau (gérant la communication entre machines, les logs, la persistance). Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Explication :

Si on change le mode de fonctionnement de

- la BDD (passage de Oracle à SQL Server),
- du réseau (changement de protocole),
- de système d'exploitation,

Alors, les classes métier ne doivent pas être impactées.

Inversement, le fait de changer les règles de validation au niveau de la partie métier du framework ne doit pas demander une modification de la base de données (à la limite, modifier une fonction, mais ne pas changer les briques de base).

Le principe DIP est équivalent au principe d'Hollywood (« Ne nous appelez pas, nous vous appellerons »), qui est une forme plus générale d'inversion de contrôle (IoC).

Les avantages du principe DIP sont les suivants :

- une nette diminution du couplage
- une meilleure encapsulation, l'implémentation concrète pouvant éventuellement être choisie dynamiquement.

Conclusion :

- Utiliser ces 5 principes SOLID demande de les conserver à l'esprit durant chaque session de développement, et le coût initial d'introduction peut être décourageant pour certains projets.

Conclusion : (suite)

- Pour cette raison, ils sont souvent utilisés conjointement avec une méthodologie Agile, qu'elle soit XP, Scrum ou autre, supportée en tout cas par une panoplie de tests unitaires ou utiliser la TDD.

Conclusion : (suite)

- Le choix donc pour se conformer ou non à ces principes doit se faire en fonction du contexte du projet, et de l'importance que l'on donne à la qualité et à l'évolutivité du programme développé.

Un design pattern est caractérisé par :

1. **Nom :** qui permet de l'identifier clairement
2. **Problématique :** description du problème auquel il répond
3. **Solution :** description de la solution souvent accompagnée d'un schéma UML
4. **Conséquences :** les avantages et les inconvénients de cette solution.

Comment procéder ?

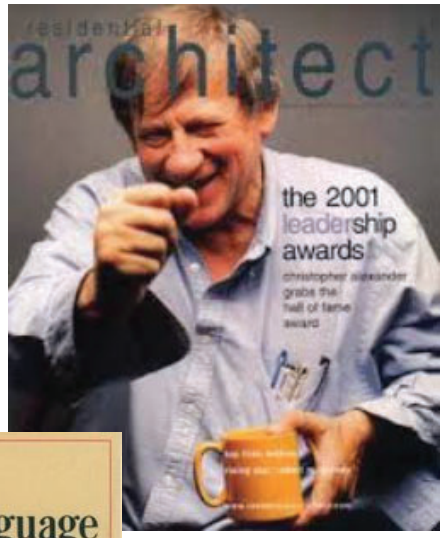
- Il est important de bien identifier le problème de conception avant de choisir un pattern, sinon le pattern choisi s'adaptera mal avec le problème et pourrait créer encore plus de problèmes qu'au départ.

Comment s'en servir ?

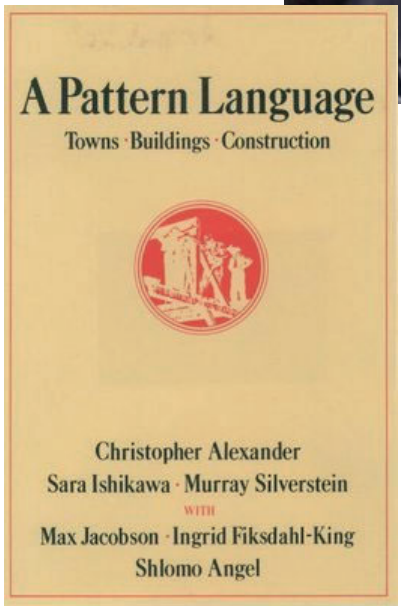
- Trouver le problème
- Identifier le Design Pattern qui pourrait résoudre le problème
- Créer sa propre solution en utilisant la solution générique du Design Pattern

Design Patterns

Design Patterns :



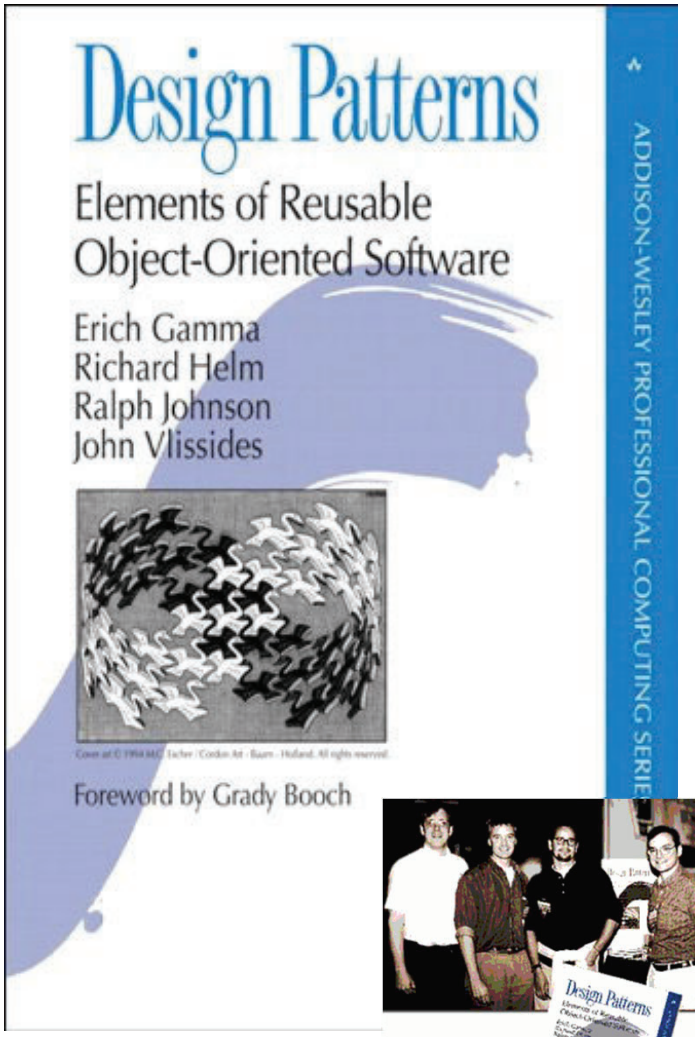
- proviennent à l'origine de l'architecte *Christoper Alexander*.
- Il rédigea un livre nommé « *A Pattern Language* », définissant un ensemble de patrons d'architecture.



Design Patterns

Design Patterns :

- Plus tard, les design patterns sont formalisés dans le livre « *Design Patterns – Elements of Reusable Object-Oriented Software du Gang Of Four* »



Design Patterns :

- Les auteurs du livre : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.

Design Patterns :

- Chaque design pattern doit répondre à un problème pouvant se manifester lors des choix de conception, et ce de telle sorte à ce que le design pattern puisse être utilisé plusieurs fois dans diverses situations.

Les 3 modèles de design pattern :

Il existe différents types de design patterns, que nous pouvons regrouper sous 3 familles ou modèles.

Design Patterns

- Les patterns de **Création** : permettent d'instancier et de configurer des classes et des objets.
- Les patterns de **Structuration :** permettent d'organiser les classes d'une application.
- Les patterns de **Comportement :** permettent d'organiser les objets pour qu'ils collaborent entre eux.

Design Patterns

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

Design Patterns

Classification		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Design Patterns

Design pattern de création

- Factory Method
- Abstract Factory
- Singleton
- Builder
- prototype

Design pattern de structure

- Composite
- Bridge
- Decorator
- Adapter
- Proxy
- Façade

Flyweight
(Poids-mouche)

Design pattern de comportement

- Iterator
- Command
- Mediator
- State
- Observer
- Templatre
- Chain of Responsability
- Memento
- Flyweight
- Strategy
- Visitor

Design pattern de BluePrint

- Struts
- WebServices
- JSF

Les 23 Gang du Design Pattern

Ryadh HADJ MOKHNECHE

Les 23 Gang du Design Pattern

Singleton

- Garantie qu'il existe au plus une seule instance d'une classe.
- Exemple :
  - Gestion de pool de connexion.

Les 23 Gang du Design Pattern

Singleton

- Constructeur privé
- Méthode d'accès ***static.***
- Code
  - Monothread : SingeltonMonoThread
  - Multithread : SingeltonMultiThread

Les 23 Gang du Design Pattern

Factory Method

- Création indirecte d'instance déléguée au niveau d'une sous classe.
- Elle gère les détails de la création des objets.
- Exemple pas bon : **VoitureHybride.**

Factory Method

- Solution : **FactoryMultiMarque**
- Une interface abstraite **(Usine)** pour crée un produit **(Voiture).**
- La spécialisation se fera au niveau des classe factory concrete **(FactoryMultiMarque)** et des produits concret **(407, Serie3).**

Les 23 Gang du Design Pattern

Abstract Factory

- Instanciation indirecte de familles de produits. (ex: Look & Feel "Windows" ou "X11/Motif").
- En changeant de fabrique on génère des objets différents(look1,look2, ...) qui ont néanmoins les mêmes fonctionnalités (même interfaces).

Les 23 Gang du Design Pattern

Abstract Factory

- Fournir une interface abstraite pour crée une famille de produit.
- FabriqueComposant :
  - FabriqueComposantSolide
    - Crée une roue : plate, plastique
    - Crée une planche : en chêne , en plastique
    - Crée une visse : en acier, en bois
  - FabriqueComposantLeger

Les 23 Gang du Design Pattern