

Principes de conception et Design Patterns

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Université
Lille1
Sciences et Technologies

UFR IEEA
Formations en
Informatique de
Lille 1

Vie d'un source...

- 1 joli, pur, “beau”
- 2 une première “hérésie”
- 3 de plus en plus d'horreurs
- 4 toujours plus d'horreurs
- 5 des horreurs partout

Conséquences :

- 1 de moins en moins maintenable et évolutif
- 2 design submergé par les “horreurs”
- 3 effet “spaghetti”

Symptômes

- les dégradations du design sont liées aux modifications des spécifications,
- ces modifications sont “à faire vite” et par d’autres que les designers originaux
 - ⇒ “ça marche” mais sans respect du design initial
 - ⇒ corruption du design (avec effet amplifié au fur et à mesure)
- mais les modifs sont **inévitables**
 - ↪ la conception/design doit permettre ces modifications (les amortir : “firewalls”)
- les dégradations sont dues aux dépendances et à l’architecture des dépendances

Principes et Patterns

- organisation des dépendances inter-classes
- abstraction des problèmes traités

Principe ouvert-fermé

Le code doit être *ouvert* à l'extension et *fermé* aux modifications.

- On doit pouvoir étendre une application sans toucher à ce qui existe (et fonctionne)

The Liskov Substitution Principle (LSP)

Principe de substitution de Liskov

Les sous-classes doivent pouvoir remplacer leur classe de base.
Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser "inconsciemment" des objets dérivés de cette classe.

On doit pouvoir *upcaster* sémantiquement

Le dilemme cercle/ellipse

un *Cercle* est une *Ellipse*

Ellipse : 2 foyers

Circle : 1 seul centre

problème avec `setFoyers(Point p1, Point p2)`

ne pas utiliser l'héritage juste pour factoriser du code
héritage \implies extension/spécialisation

(ce n'est pas le cas *Cercle* pour *Ellipse*, dans ce cas utiliser la
composition)

Dependency Inversion Principle (DIP)

ou **Inversion of Control**

Dépendre des abstractions

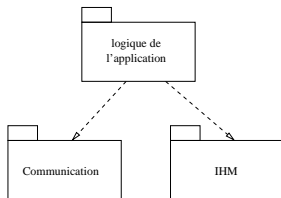
Dépendez des abstractions, ne dépendez pas des concrétisations.

- 1 Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions
- 2 Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.

Constat

- Généralement : le réutilisation des modules métiers (de haut niveau) n'est pas possible : ils sont construits sur les modules de bas niveau directement
↳ IHM, communication , etc.

Problèmes

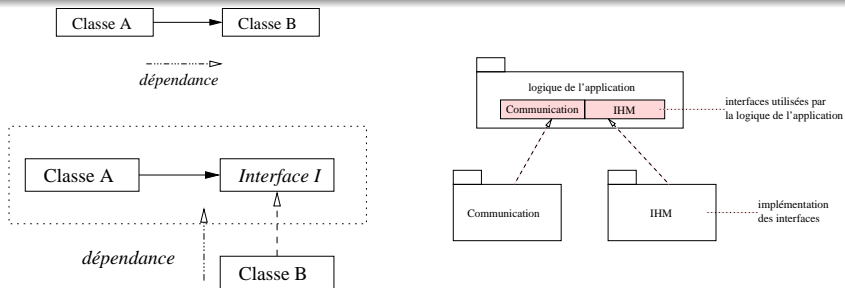


- modification nécessaire des modules de haut niveau lorsque les bas niveaux sont modifiés
- réutilisation des modules de haut niveau indépendamment des bas niveaux impossible
⇒ pas de réutilisation de la logique de l'application en dehors du contexte technique!

Inversion des dépendances

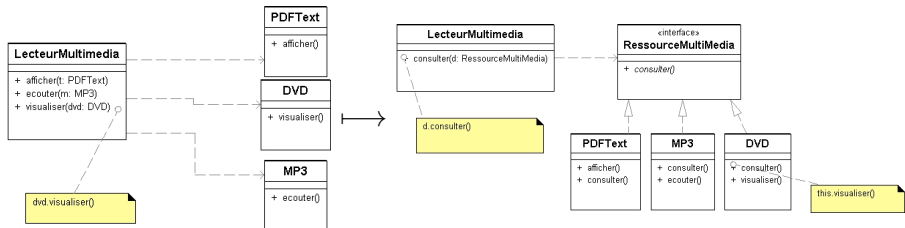
Principe d'inversion des dépendances

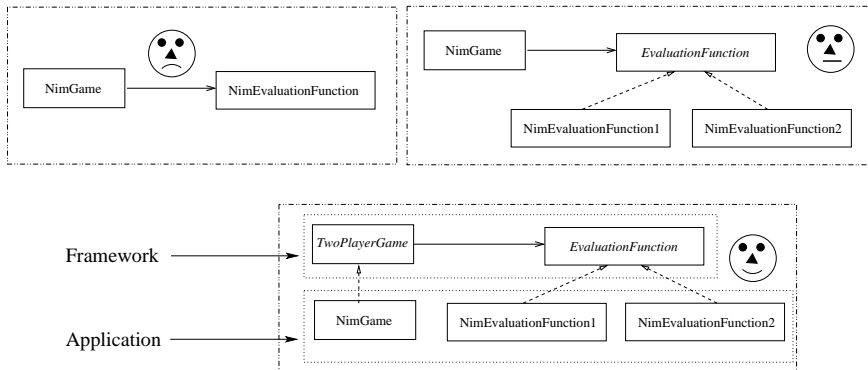
Les modules de bas niveau doivent se conformer à des interfaces définies (car utilisées) par les modules de haut niveau.



- Dépendre d'interfaces et de classes abstraites plutôt que de classes.
- utiliser l'abstraction (encore et toujours...)

⇒ Frameworks





Interface Segregation Principle (ISP)

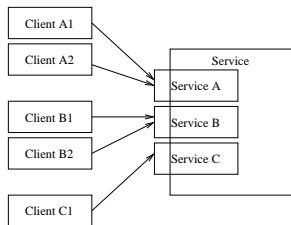
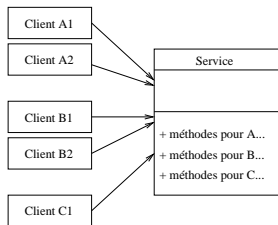
Principe de séparation des interfaces

Plusieurs interfaces client spécifiques valent mieux qu'une seule interface générale. Les classes clientes ne doivent pas être forcées de dépendre d'interfaces qu'elles n'utilisent pas.

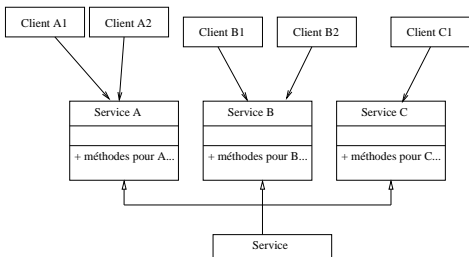
- Éviter qu'un client voit une interface qui ne le concerne pas
- Éviter que les évolutions dues à une partie du service aient un impact sur un client alors qu'il n'est pas concerné

Solution = Séparation

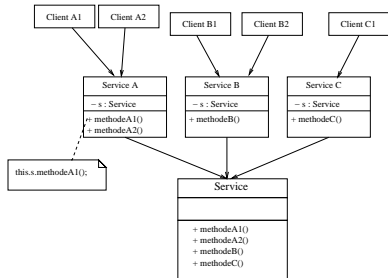
N'offrir aux classes clientes qu'un accès limité...



Mise en œuvre



- polymorphisme (+ upcast)
(JAVA via interfaces)



- utilisation du design pattern
Adapter

Patrons de conception

Design Patterns

Elements of reusable objected-oriented softwares

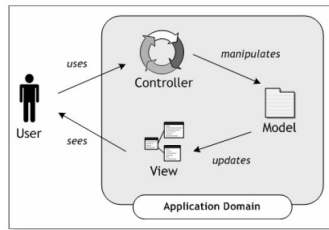
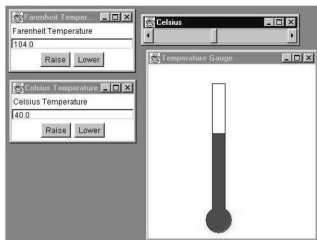
Addison Wesley

Gang of Four : E. Gamma, R. Helm, R. Johson, J. Vlissides

“Design Patterns. Tête la première.” Eric et Elisabeth Freeman

MVC, l'ancêtre

- Un pattern pour les IHM : découplage IHM/applicatif. s'appuie sur les patterns Observer et Strategy.
- Plusieurs vues possibles pour une “même donnée” \implies gérer la cohérence.



Souvent au niveau IHM, vue et contrôleur se confondent.

Applicable à d'autres domaines que IHM

Description d'un pattern

- *Pattern Name and Classification* Le nom du pattern, évoque succinctement l'esprit du pattern.
- *Intent* Une brève description qui répond aux questions suivantes :
 - Qu'est ce que le design pattern fait ?
 - Quels sont son objectif et sa justification ?
 - Quel problème de conception particulier vise-t-il ?
- *Also Known As* Autres noms connus pour le pattern, si il y en a.
- *Motivation* Un scénario qui illustre un problème de conception et comment la structuration des classes et des objets dans ce pattern le résolvent. Le scénario devra aider à comprendre la description plus abstraite du pattern qui suit.

- *Applicability*
 - Quelles sont les situations dans lesquelles le pattern peut être appliqué ?
 - Quels sont les exemples de mauvaise conception que le pattern peut attaquer ?
 - Comment reconnaître ces situations ?
- *Structure* Une représentation graphique des classes dans le pattern utilisant une notation basée sur l'Object Modeling Technique (OMT).
- *Participants* Les classes et/ou les objets qui participent au design pattern et leurs responsabilités.
- *Collaborations* Comment les participants collaborent pour tenir leurs responsabilités.
- *Consequences*
 - Comment le pattern satisfait-il ses objectifs ?
 - Quelles sont les conséquences et résultats de l'usage du pattern ?
 - Quels points de la structure permet il de faire varier indépendamment ?

- *Implementation*
 - Quels pièges, astuces ou techniques devriez vous connaître pour implémenter ce pattern ?
 - Y a-t-il des problèmes spécifiques à un langage ?
- *Sample Code* Portions de code qui illustrent comment vous pourriez implémenter ce pattern.
- *Known Uses* Exemples d'utilisation du pattern trouvés dans des systèmes réels.
- *Related Patterns*
 - Quels design patterns sont fortement liés à celui-ci et quelles sont les différences importantes ?
 - Avec quels autres patterns celui-ci devrait il être utilisé ?

Trois catégories

- **Creational Patterns** Concernent l'abstraction du processus d'instanciation.
- **Structural Patterns** Concernent la composition des classes et objets pour obtenir des structures plus complexes.
- **Behavioural Patterns** Concernent les algorithmes et la répartition des responsabilités entre les objets.

Creational Patterns

Abstraction du processus d'instanciation.

- Permettent de rendre le système indépendant du mode de création des objets qui le compose.
- Un creational pattern de classe utilise l'héritage pour faire varier la classe instanciée.
- Un creational pattern d'objet délègue la création à un autre objet.

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Prototype** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Singleton** Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Composition de classes et objets pour obtenir des structures plus complexes.

- Un structural pattern de classe utilise l'héritage pour composer des interfaces ou des implémentations.
- Un structural pattern objet décrit comment composer des objets pour obtenir de nouvelles fonctionnalités, avec possibilité de faire évoluer la composition à l'exécution.

Structural Patterns

- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

- **Facade** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** Provide a surrogate or placeholder for another object to control access to it.

Behavioural Patterns

Algorithmes et répartition des responsabilités entre les objets.

- Décrivent également les patterns de communication entre classes et objets : permettent donc de se dégager du problème du flux de contrôle et de se concentrer sur les relations entre objets.
- Un behavioural pattern de classe utilise l'héritage pour distribuer les comportements entre les classes.
- Un behavioural pattern objet utilise l'héritage plutôt que la composition. Certains décrivent les coopérations entre objets, d'autres utilisent l'encapsulation du comportement dans un objet auquel sont déléguées les requêtes.

- **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.