

## Chapitre 1

# Bonnes pratiques pour améliorer l'architecture de son application Java



**Mohamed Romdhani**  
Avril 2017

## Sommaire

Amélioration de l'architecture des applications

- Les symptômes d'une mauvaise conception logicielle
- Les principes SOLID de R. Martin
- Les patterns de conception GOF et EAP de M. Fowler
- Couplage faible entre les modules injection de dépendances

## Les symptômes d'une mauvaise conception logicielle

## La rigidité du logiciel

**Rigidity is the tendency for software to be difficult to change, even in simple ways**

- **Symptom: Every change causes a cascade of subsequent changes in dependent modules.**
- **Effect: When software behaves this way, managers fear to allow developers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the developers will be finished.**
- **La rigidité : chaque évolution est susceptible d'impacter de nombreuses parties de l'application.**
  - Le développement est de plus en plus coûteux, ce qui introduit des risques au cours même du développement (ironiquement, c'est au moment où les échéances de livraison approchent et où la pression monte sur le projet que l'application devient la plus difficile à modifier).
  - Le coût des modifications étant élevé, le logiciel a peu de chances d'évoluer après sa mise en production.

## La fragilité du logiciel

Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.

- Symptom: Every fix makes it worse, introducing more problems than are solved.
- Effect: Every time managers/ team leaders authorize a fix, they fear that the software will break in some unexpected way.

- **La fragilité : la modification d'une partie de l'application peut provoquer des erreurs dans une autre partie de l'application.**

- Le logiciel est peu robuste, et le coût de maintenance reste élevé.
- Les modifications étant de plus en plus risquées, le logiciel a peu de chances d'évoluer après sa mise en production.

## L'immobilité du logiciel

Immobility is the inability to reuse software from other projects or from parts of the same project.

- Symptom: A developer discovers that he needs a module that is similar to one that another developer wrote. But the module in question has too much baggage that it depends upon. After much work, the developer discovers that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate.
- Effect: And so the software is simply rewritten instead of reused.

- **L'immobilité : il est difficile d'extraire une partie de l'application pour la réutiliser dans une autre application.**

- Le coût de développement de chaque application reste élevé puisqu'il faut repartir de zéro à chaque fois.
- Bien sûr, ces problèmes sont d'autant plus sensibles que l'application est volumineuse. Ils sont déjà perceptibles pour des applications de quelques milliers de lignes de code.

## Amélioration de l'architecture des applications

### Quelles sont les causes de mauvaises conceptions ?

- Obvious reasons: lack of design skills/ design practices, changing technologies, time/ resource constraints, domain complexity etc.
- Not so obvious:
  - Software rotting is a slow process .. Even originally clean and elegant design may degenerate over the months/ years ..
  - Unplanned and improper module dependencies creep in; Dependencies go unmanaged.
  - Requirements often change in the way the original design or designer did not anticipate ..

M.Romdhani, Avril 2017

7

## Les principes SOLID de R. Martin

Amélioration de l'architecture des applications

## Design Principles, classified

1. The Open/Closed Principle (OCP)	Gestion des évolutions et des dépendances entre classes
2. The Liskov Substitution Principle (LSP)	
3. The Dependency Inversion Principle (DIP)	
4. The Interface Segregation Principle (ISP)	
5. The Reuse/Release Equivalency Principle (REP)	Organisation de l'application en modules
6. The Common Closure Principle (CCP)	
7. The Common Reuse Principle (CRP)	
8. The Acyclic Dependencies Principle (ADP)	Gestion de la stabilité de l'application
9. The Stable Dependencies Principle (SDP)	
10. The Stable Abstractions Principle (SAP)	

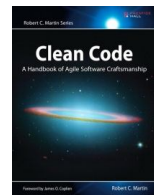
M.Romdhani, Avril 2017

9

Amélioration de l'architecture des applications

## SOLID de R. Martins

- Introduced by Robert C. Martins ("Uncle Bob")
  - Author of several books, e.g. "Clean Code"
- The S.O.L.I.D. design principles are a collection of best practices for object-oriented design.
  - All of the Gang of Four design patterns adhere to these principles in one form or another.
- **SOLID**
  - **S**ingle Responsibility Principle
  - **O**pen Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle
- Code becomes more *Testable* (TDD is not only about testing, more important it is about Design) and more *Maintainable*.



M.Romdhani, Avril 2017

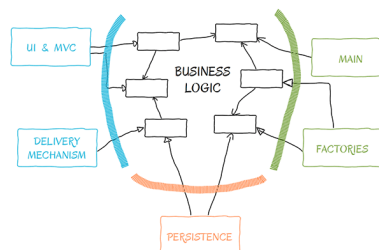
10

## The S.O.L.I.D. Design Principles

- The S.O.L.I.D. design principles are a collection of best practices for object-oriented design.
  - All of the Gang of Four design patterns adhere to these principles in one form or another.
  - The term S.O.L.I.D. comes from the initial letter of each of the five principles that were collected in the book *Agile Principles, Patterns, and Practices in C#* by Robert C. Martin, or Uncle Bob to his friends.
    - Single Responsibility Principle (SRP)
    - Open-Closed Principle (OCP)
    - Liskov Substitution Principle (LSP)
    - Interface Segregation Principle (ISP)
    - Dependency Inversion Principle (DIP)

## The Single Responsibility Principle (SRP)

- **Single Responsibility Principle (SRP)**
  - The principle of SRP is closely aligned with SoC. It states that every object should only have one reason to change and a single focus of responsibility.
  - By adhering to this principle, you avoid the problem of monolithic class design that is the software equivalent of a Swiss army knife.
  - By having concise objects, you again increase the readability and maintenance of a system.
- If we analyze this schema, you can see how the Single Responsibility Principle is respected.
  - Object creation is separated on the right in Factories and the main entry point of our application, one actor one responsibility.
  - Persistence is also taken care of at the bottom. A separate module for the separate responsibility.
  - Finally, on the left, we have presentation or the delivery mechanism if you wish, in the form of an MVC or any other type of UI.



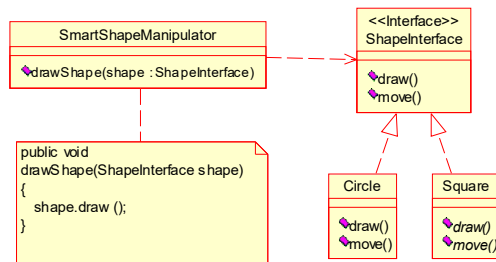
## The Open-Closed Principle (OCP)

### Open-Closed Principle (OCP)

- The OCP states that classes should be open for extension and closed for modification, in that **you should be able to add new features and extend a class without changing its internal behavior**.
- The principle strives to avoid breaking the existing class and other classes that depend on it, which would create a ripple effect of bugs and errors throughout your application.



- **If I need to create a new shape, such as a Triangle, I must modify the 'drawShape()' function.**
- **Then, new features can be added to the application by adding new code rather than by changing working code. Thus, the working code is not exposed to breakage.**



M.Romdhani, Avril 2017

13

## The Liskov Substitution Principle (LSP)

### Liskov Substitution Principle (LSP)

- The LSP dictates that you should be able to use any derived class in place of a parent class and it have behave in the same manner without modification.
- This principle is in line with OCP in that it ensures that a derived class does not affect the behavior of a parent class, or, put another way, derived classes must be substitutable for their base classes.



- **A client of a base class should continue to function properly if a derivative of that base class is passed to it.**

- In other words, if some function takes an argument of type Policy, then it should be legal to pass in an instance of PersonalAutoPolicy (provided PersonalAutoPolicy is directly/indirectly derived from Policy)
- The Liskov substitution principle is closely related to **the design by contract methodology**, leading to some restrictions on how contracts can interact with inheritance
- *Violations of LSP are latent violations of OCP*

M.Romdhani, Avril 2017

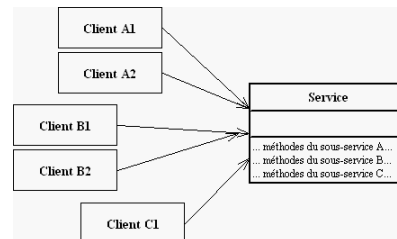
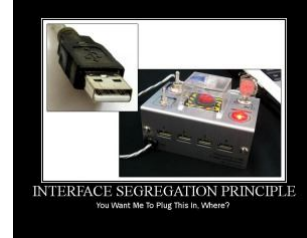
14

## The Interface Segregation Principle (ISP)

Amélioration de l'architecture des applications

### Interface Segregation Principle (ISP)

- The ISP is all about splitting the methods of a contract into groups of responsibility and assigning interfaces to these groups to prevent a client from needing to implement one large interface and a host of methods that they do not use.
- The purpose behind this is so that classes wanting to use the same interfaces only need to implement a specific set of methods as opposed to a monolithic interface of methods.
- If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each type of client and multiply inherit them into the class.



M.Romdhani, Avril 2017

15

## The Dependency Inversion Principle (DIP)

Amélioration de l'architecture des applications

### Dependency Inversion Principle (DIP)

- The DIP is all about isolating your classes from concrete implementations and having them depend on abstract classes or interfaces.
- It promotes the mantra of coding to an interface rather than an implementation, which increases flexibility within a system by ensuring you are not tightly coupled to one implementation.



### Dependency Injection (DI) and Inversion Of Control (IOC)

- *DI is the act of supplying a low level or dependent class via a constructor, method, or property.*
- *These dependent classes can be inverted to interfaces or abstract classes that will lead to loosely coupled systems that are highly testable and easy to change.*
- *In IoC, a system's flow of control is inverted compared to procedural programming. An example of this is an IoC container, whose purpose is to inject services into client code without having the client code specifying the concrete implementation. The control in this instance that is being inverted is the act of the client obtaining the service.*

M.Romdhani, Avril 2017

16



## The Dependency Inversion Principle (DIP)

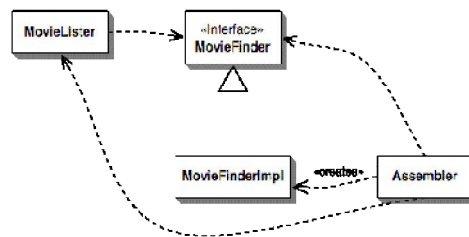
Amélioration de l'architecture des applications

### ■ L'exemple de Martin Fowler [<http://martinfowler.com/articles/injection.html>]

- Dont la lecture semble indispensable

### ■ Thème :

- La liste des films d'un certain réalisateur de cinéma ...
  - À la recherche de films dans une liste sous plusieurs formats
    - Texte, CSV, XML, BDD ...



M.Romdhani, Avril 2017

17

## MovieLister, MovieFinder, Assembler

Amélioration de l'architecture des applications

```
public class Movies {
    private MovieFinder finder;

    public Movies() {
    }

    public void setFinder(MovieFinder finder) {
        this.finder = finder;
    }

    public List<Movie> moviesDirectedBy(director: String) {
        List<Movie> allMovies = finder.findAll();
        for (m in allMovies) {
            if (!movie.getDirector().equals(director))
                allMovies.remove(m);
        }
        return allMovies;
    }
}

public class SemiColonDelimitedMovieFinder implements MovieFinder {
    private String filename;

    public void setFilename(String filename) {
        this.filename = filename;
    }
}
```

Déjà vu !

M.Romdhani, Avril 2017

18

## Les patterns de conception GOF et EAP de M. Fowler

### Les Design Patterns

Amélioration de l'architecture des applications

- Décrivent un modèle de solution éprouvé à un problème récurrent dans un contexte donné
- Présentés sous forme standard :
  - Exposé du problème
  - Contexte du problème et exemple concret
  - Description du modèle de la solution
  - Variantes et conditions d'utilisation
  - Avantages et inconvénients
- Issus du domaine de l'architecture des bâtiments et de l'urbanisme
- Appliqués à l'informatique par *Gamma & all* (Gof) sous la forme des design patterns

## Les patterns, pourquoi ?

- **Profiter de l'expérience et du savoir-faire de la communauté en terme de conception**
  - Améliorer la qualité et la maintenabilité du code
  - Éviter de tomber dans les erreurs et les pièges connus
- **Établir un "vocabulaire commun"**
  - Plus simple de dire, "Nous mettons une Façade ici".
- **Offrir une vision plus abstraite de l'architecture logicielle**
  - Libérer les concepteurs/développeurs de détails d'implémentation, surtout dans les phases initiales du cycle de développement
- **En résumé, Servir le rôle de "référentiel"**

## Historique des Design Patterns

Christopher Alexander <i>The Timeless Way of Building</i> <i>A Pattern Language: Towns, Buildings, Construction</i>	Architecture	1970'
Gang of Four (GoF) <i>Design Patterns: Elements of Reusable Object-Oriented Software</i>	Object Oriented Software Design	1995'
Sun Microsystems Microsoft	J2EE Design Patterns Microsoft Patterns and Practices	2000'
Many Authors	Other Areas: SOA, BPM, Web 2.0, HCI Education, ...	2008'

## Enterprise Software Patterns

- Alur, Crupi, and Malks – Core J2EE Patterns
- Marinescu – EJB Patterns
- Fowler – Patterns of Enterprise Application Architecture – [martinfowler.com/eaCatalog](http://martinfowler.com/eaCatalog)
- Evans – Domain Driven Development – [domainLanguage.com](http://domainLanguage.com)
- Hohpe and Woolf – Enterprise Integration Patterns – [enterpriseIntegrationPatterns.com](http://enterpriseIntegrationPatterns.com)
- Hohmann – Beyond Software Architecture - [lukehohmann.com](http://lukehohmann.com)

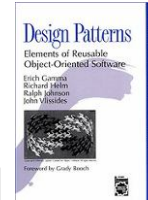
**Implémenter efficacement les  
patterns GOF avec Java**

## Organisation du catalogue des Design patterns GOF

Amélioration de l'architecture des applications

- **Créationnel** : processus de création des objets
- **Structurel** : composition des classes ou des objets
- **Comportemental** : comment les classes et les objets interagissent et distribuent les responsabilités

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	<a href="#">Factory Method (107)</a>	<a href="#">Adapter (139)</a>	<a href="#">Interpreter (243)</a> <a href="#">Template Method (325)</a>
	Object	<a href="#">Abstract Factory (87)</a> <a href="#">Builder (97)</a> <a href="#">Prototype (117)</a> <a href="#">Singleton (127)</a>	<a href="#">Adapter (139)</a> <a href="#">Bridge (151)</a> <a href="#">Composite (163)</a> <a href="#">Decorator (175)</a> <a href="#">Facade (185)</a> <a href="#">Proxy (207)</a>	<a href="#">Chain of Responsibility (223)</a> <a href="#">Command (233)</a> <a href="#">Iterator (257)</a> <a href="#">Mediator (273)</a> <a href="#">Memento (283)</a> <a href="#">Flyweight (195)</a> <a href="#">Observer (293)</a> <a href="#">State (305)</a> <a href="#">Strategy (315)</a> <a href="#">Visitor (331)</a>



M.Romdhani, Avril 2017

25

## Les patterns du catalogue GOF

Amélioration de l'architecture des applications

### ■ 23 patterns dans leur ouvrage de référence

Gamma E, Helm R, Johnson R and Vlissides J (1995) **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA: Addison-Wesley.

### ■ Trois catégories:

- Patterns Créationnels
- Patterns Structurels
- Patterns Comportementaux

### ■ Souvent critiqués par les puristes pour l'aspect "basique" du catalogue

## Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

M.Romdhani, Avril 2017

26

## Les patterns créationnels

- **Les patterns créationnels concernent le processus de création d'objets. Ils créent les objets pour les clients, au lieu que les clients instancient les objets directement. Cela donne plus de flexibilité au logiciel pour décider quels objets doivent être créés dans un cas donné.**
  - **Fabrique Abstraite (Abstract Factory) :**
    - Fournit une interface pour créer des familles d'objet liés ou dépendants sans avoir à spécifier leurs classes concrètes.
  - **Constructeur (Builder) :**
    - Sépare la construction d'un objet complexe de sa représentation, afin que le même processus de construction puisse créer différentes représentations.
  - **Fabrication (Factory Method) :**
    - Définit une interface pour créer un objet, mais laisse aux sous-classes le soin de décider quelle classe instancier. Le pattern Fabrication laisse une classe déléguer son instanciation à des sous-classes.
  - **Prototype (Prototype) :**
    - Spécifie quelles sortes d'objets créer en utilisant une instance prototype, et crée de nouveaux objets en copiant ce prototype.
  - **Singleton (Singleton) :**
    - Fait en sorte qu'une classe n'aie qu'une seule instance, et fournit un point d'accès global à celle-ci.

M.Romdhani, Avril 2017

27

## Les patterns structurels

- **Les patterns structurels concernent la composition de classes et d'objet. Ils aident à composer des groupes d'objets en structures plus larges, comme par exemple des interfaces utilisateur complexes ou des données comptables.**
  - **Adaptateur:**
    - Convertit l'interface d'une classe en une autre interface attendue par les clients. L'Adaptateur permet de travailler ensemble à des classes pour lesquelles ce serait normalement impossible à cause d'interfaces incompatibles.
  - **Pont:**
    - Sépare une abstraction de son implémentation de sorte que les deux puissent varier indépendamment.
  - **Composite:**
    - Compose des objets en structures d'arbres pour représenter des hiérarchies. Composite permet à des client de traiter des objets ou des composition d'objets de la même manière.
  - **Décorateur:**
    - Attache des responsabilités additionnelles à un objet de manière dynamique. Les Décorateurs fournissent une alternative flexible au sous-classage pour étendre les fonctionnalités.
  - **Façade:**
    - Fournit une interface unifiée à un ensemble d'interfaces dans un sous-système. La Façade définit une interface de plus haut niveau qui rend le sous-système plus facile à utiliser.
  - **Poids mouche:**
    - Utilise un procédé de partage pour supporter de petits objets efficacement.
  - **Proxy:**
    - Fournit un substitut à un objet pour contrôler l'accès à celui-ci.

M.Romdhani, Avril 2017

28

## Les patterns comportementaux

- Les patterns comportementaux caractérisent les façons dont les classes et les objets interagissent et se partagent les responsabilités. Ils aident à définir la communication entre les objets du système et comment le flot d'information est contrôlé dans un programme complexe.
  - **Chaîne de responsabilité:**
    - Évite de coupler l'émetteur d'une requête à son receveur en donnant à plus d'un objet la chance de traiter la requête. Les objets récepteurs sont chaînés et passent la requête le long de la chaîne jusqu'à ce qu'un objet la traite.
  - **Commande:**
    - Encapsule une requête en tant qu'objet, ceci permettant de paramétrer les clients avec des requêtes différentes, mettre les requêtes en file d'attente ou les inscrire dans des fichiers journaux, et de supporter l'annulation des opérations.
  - **Interpréteur:**
    - Pour un langage donné, définit une représentation de sa grammaire avec un interpréteur qui utilise la représentation pour interpréter les phrases du langage.
  - **Itérateur:**
    - Fournit un moyen d'accéder séquentiellement aux éléments d'un objet agrégé sans exposer sa représentation sous-jacente.
  - **Médiateur:**
    - Définit un objet qui encapsule la façon d'interagir d'un groupe d'objets. Le Médiateur favorise un couplage lâche en empêchant les objets de faire référence aux autres explicitement, et permet de faire varier leurs interactions indépendamment.

## Les patterns comportementaux (suite)

- **Memento:**
  - Sans violer le principe d'encapsulation, capture et externalise l'état interne d'un objet, de sorte que cet objet puisse être restauré dans cet état plus tard.
- **Observateur:**
  - Définit une relation un-à-plusieurs entre des objets de sorte que lorsqu'un objet change d'état, tous les objets dépendants sont notifiés et mis à jour automatiquement.
- **État:**
  - Permet à un objet de modifier son comportement lorsque son état change. On aura l'impression que l'objet change de classe.
- **Stratégie:**
  - Définit une famille d'algorithmes, encapsule chacun, et les rend interchangeables. La Stratégie permet à un algorithme de varier indépendamment des clients qui l'utilisent.
- **Patron de méthode:**
  - Définit le squelette d'un algorithme dans une opération, en reportant certaines étapes à des sous-classes. Le Patron de méthode permet à des sous-classes de redéfinir certaines étapes d'un algorithme sans changer la structure de l'algorithme.
- **Visiteur:**
  - Représente une opération à accomplir sur les éléments de la structure d'un objet. Le Visiteur permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels il opère.

## Implémentation du pattern Singleton en Java

Amélioration de l'architecture des applications

- Le pattern singleton est un des patterns les plus connus dans le génie logiciel. Fondamentalement, un singleton est une classe qui permet une seule instance d'elle-même, et habituellement donne un accès simple à cette instance

- **Première version - Non thread-safe**

```
public class Singleton {  
    private Singleton() {} /** Constructeur privé */  
    /** Instance unique non préinitialisée */  
    private static Singleton INSTANCE = null;  
    /** Point d'accès pour l'instance unique du singleton */  
    public static Singleton getInstance()  
    {  
        if (INSTANCE == null)  
        {  
            INSTANCE = new Singleton();  
        }  
        return INSTANCE;  
    }  
}
```

Deux threads différents pourraient tous les deux avoir évalué le **test if (instance == null)** et obtenir vrai comme résultat, les deux créeront une instance, ce qui viole le pattern singleton

M.Romdhani, Avril 2017

31

## Implémentation du pattern Singleton en Java

Amélioration de l'architecture des applications

- **Seconde version - Simple thread-safety**

```
public class Singleton  
{  
    private Singleton() {} /** Constructeur privé */  
  
    /** Instance unique non préinitialisée */  
    private static Singleton INSTANCE = null;  
  
    /** Point d'accès pour l'instance unique du singleton */  
    public static synchronized Singleton getInstance(){  
        if (INSTANCE == null)  
        {  
            INSTANCE = new Singleton();  
        }  
        return INSTANCE;  
    }  
}
```

Cette implémentation est thread-safe. Le thread met un verrou sur la méthode `getInstance()`, et ensuite vérifie si l'instance a été créée ou non avant de créer l'instance.

- Malheureusement, les performances en souffrent car un verrou est obtenu à chaque fois que l'instance est requise.

M.Romdhani, Avril 2017

32



## Les patterns EAP (Entreprise Architecture Patterns) de M. Fowler

### Overview of Fowler's Enterprise Patterns

Amélioration de l'architecture des applications

- **Martin Fowler's Patterns of Enterprise Application Architecture** book is a best practice and patterns reference for building enterprise-level applications.

- **Layering**

- Three principal layers

- **Domain Logic Patterns**

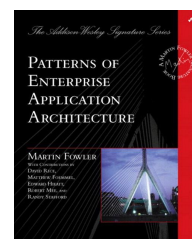
- Transaction Script, Active Record, and Domain Model

- **Object Relational Mapping**

- Unit of Work, Repository, Data Mapper, Identity Map

- **Web Presentation Patterns**

- MVC, MVP, MVVM

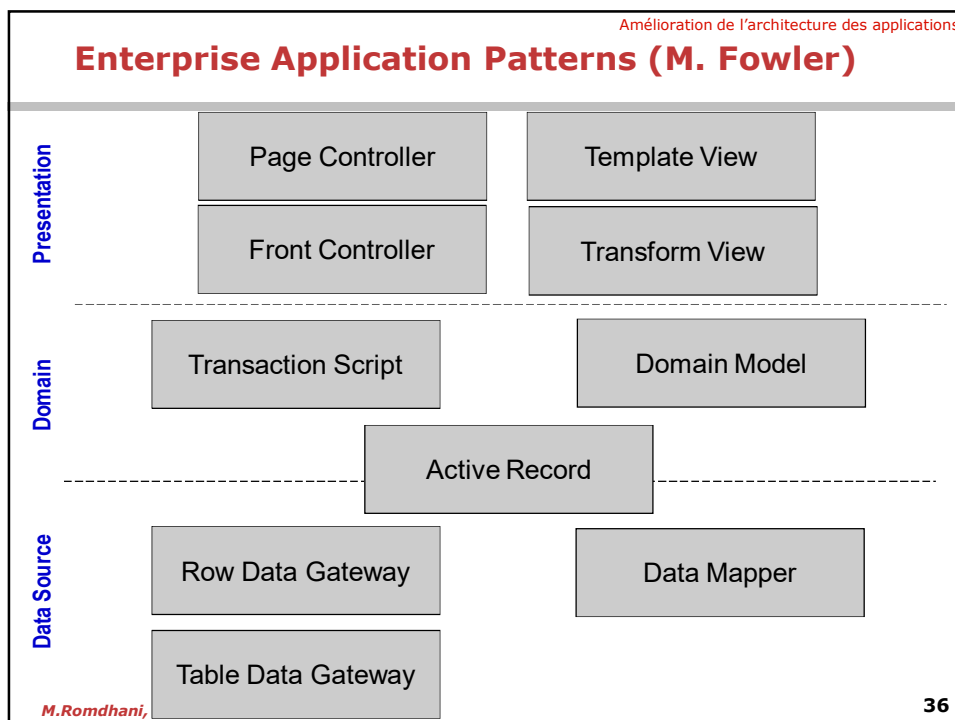


Amélioration de l'architecture des applications

## Three Primary Layers

- **Presentation**
  - Interacts with the “user” of the application
  - eg: rich client, HTML browser, web service
- **Domain**
  - Business rules, validations, calculations
- **Data Source**
  - Connects to the rest of the enterprise environment
  - Persistence: RDBMs
  - Messaging, TP monitors, legacy apps....

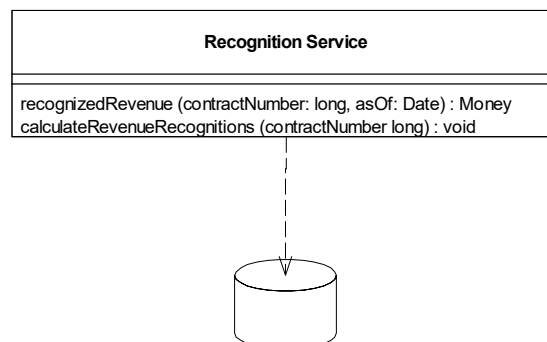
M.Romdhani, Avril 201735

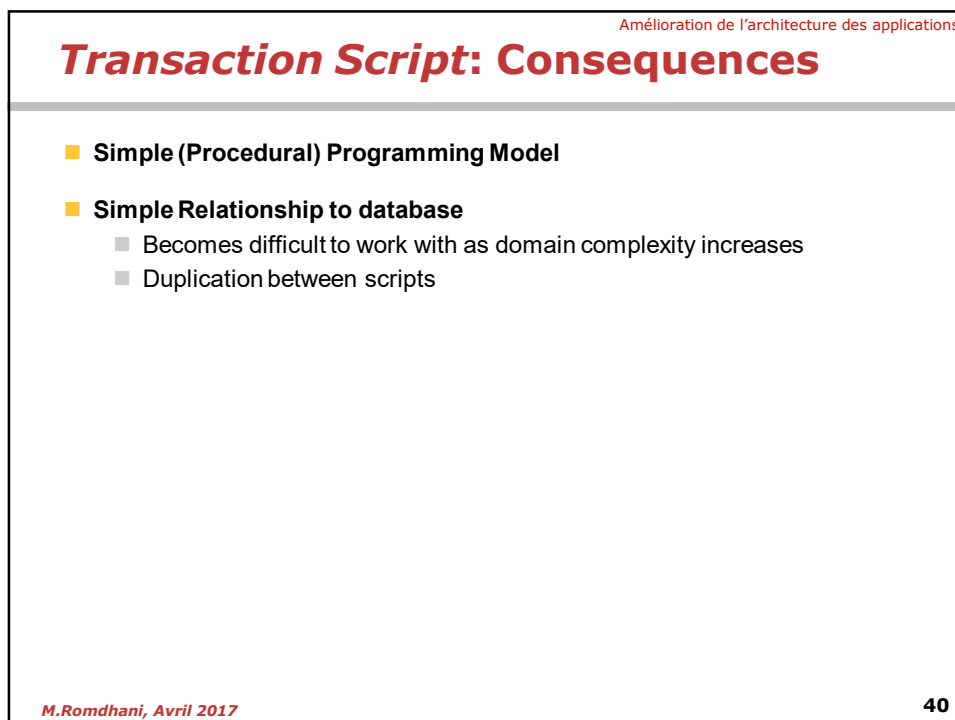
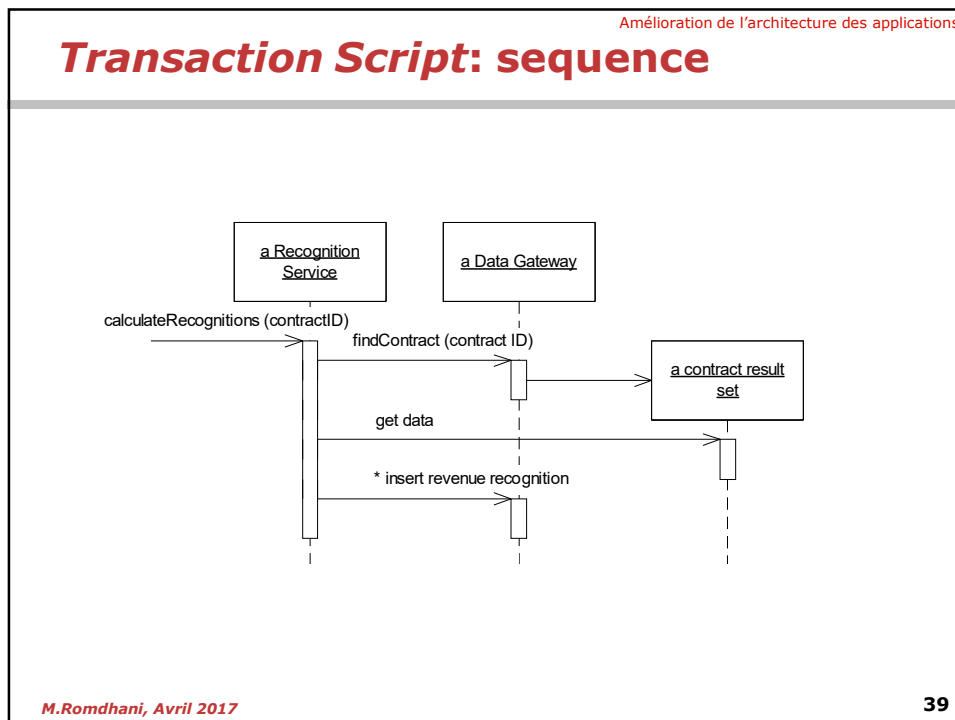


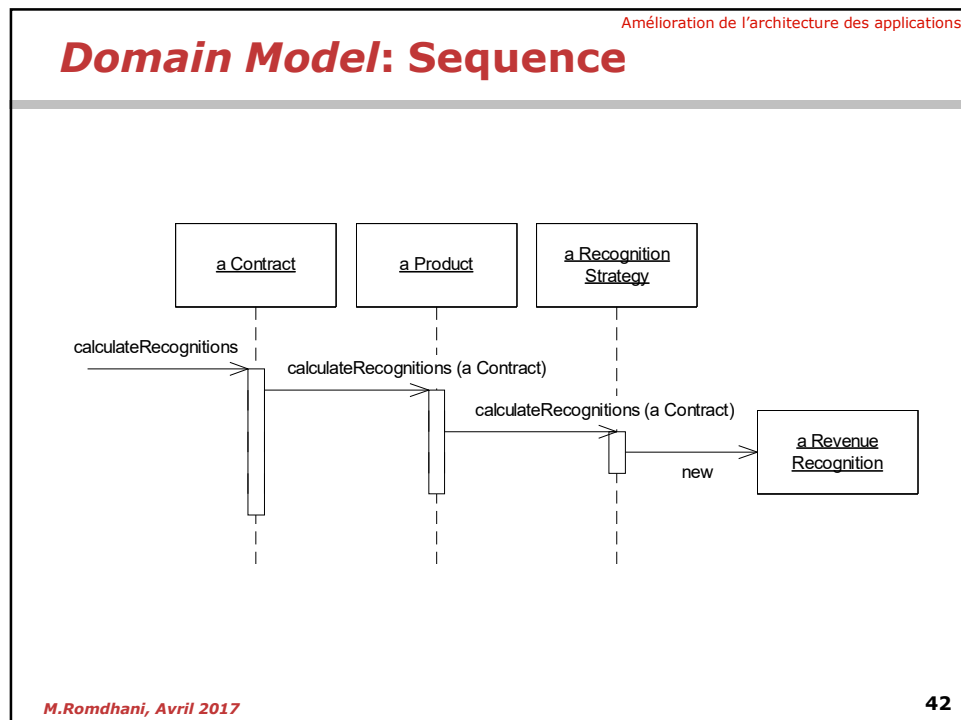
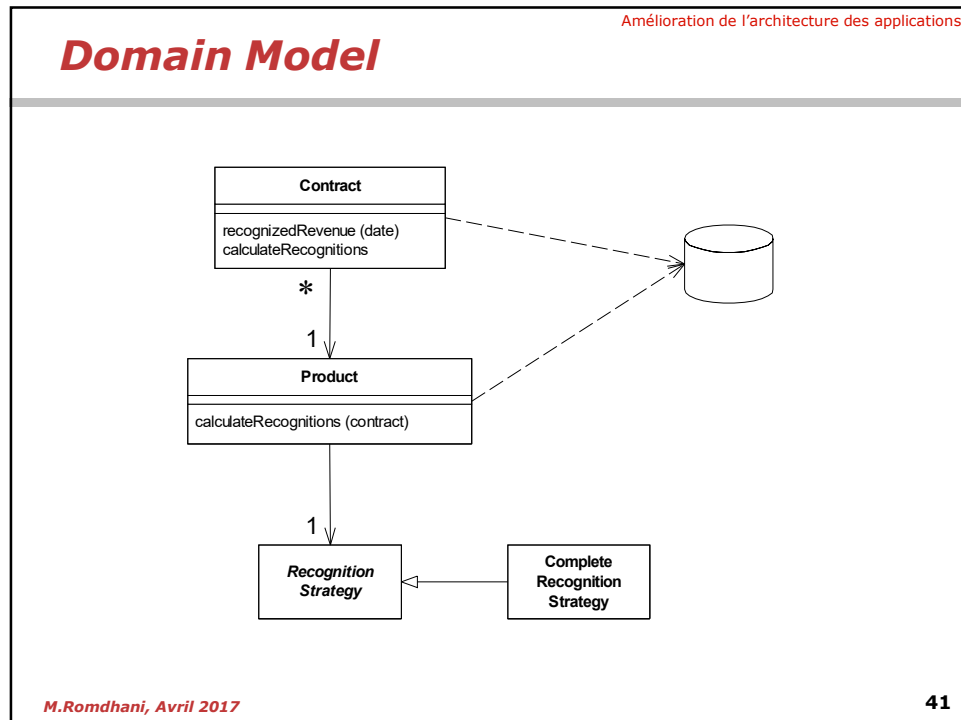
## Organizing Domain Logic

- *Transaction Script*
- *Domain Model*

## Transaction Script



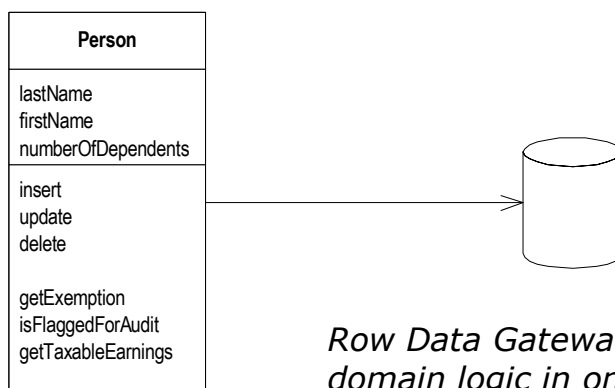


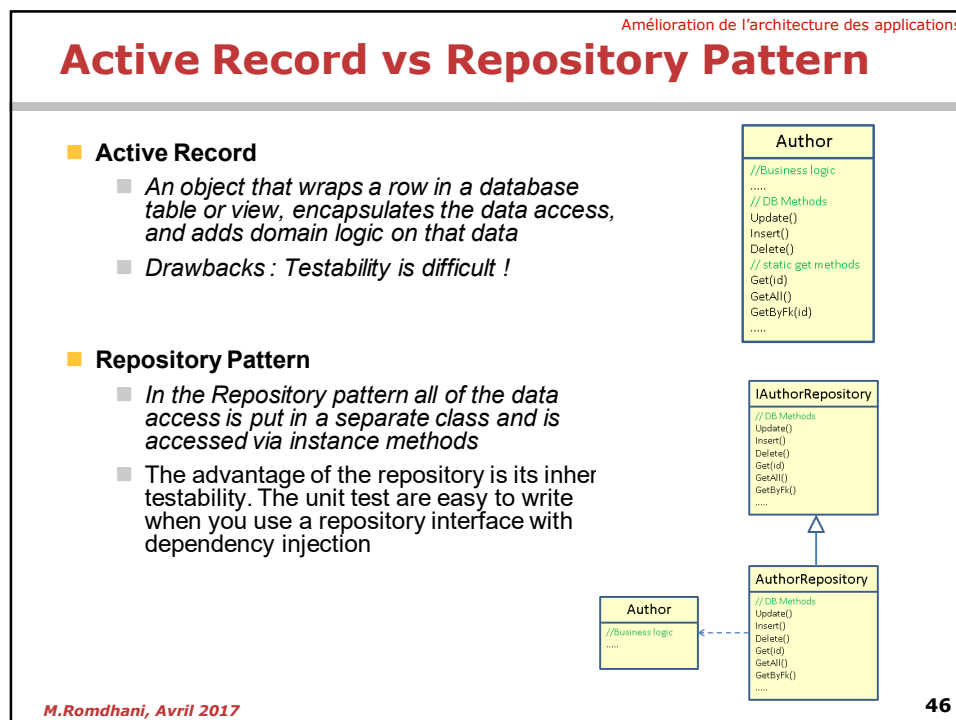
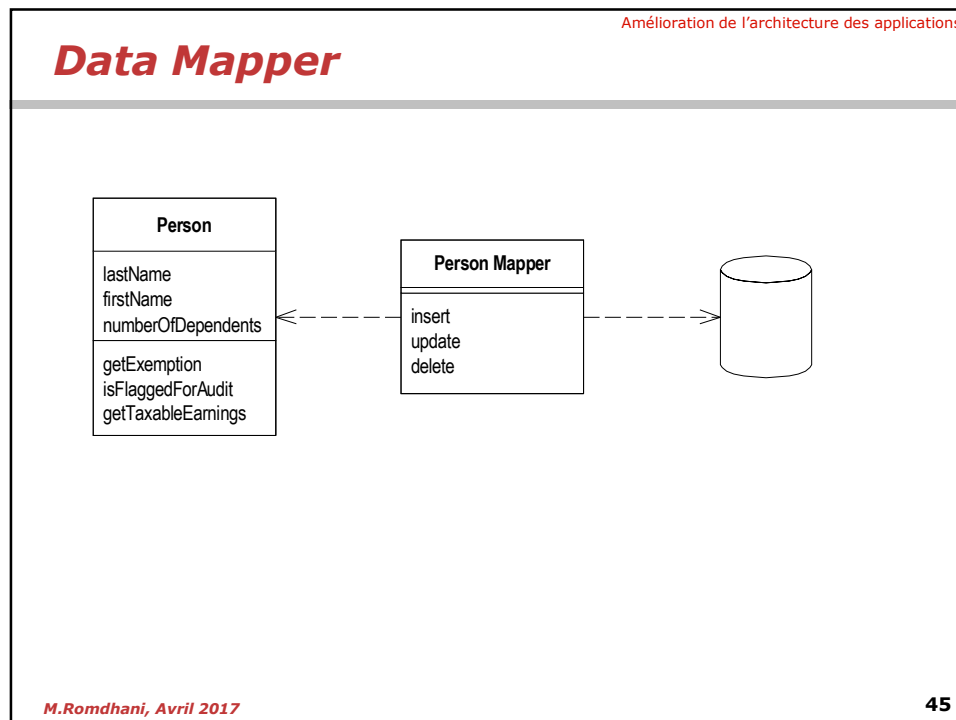


## Domain Model: Consequences

- Can deal with very complex domain logic
- Paradigm Shift
- Can have complex mapping to database

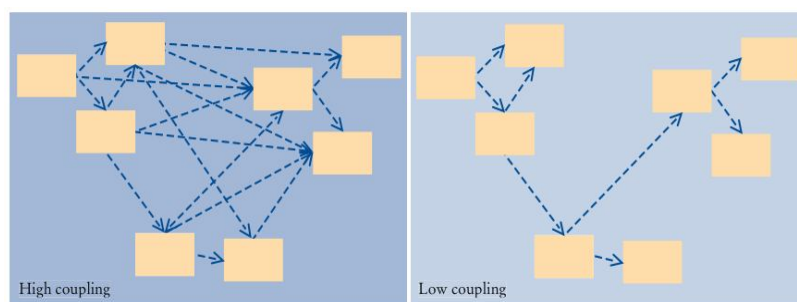
## Active Record





## Couplage faible entre les modules injection de dépendances

## Couplage faible



■ Source : <http://horstmann.com/sjsu/cs46a/ch08/ch08.html>



## Couplage faible, pourquoi faire ?

### ■ Le bon sens

- A dépend de B qui dépend de C ...
- A dépend de B qui dépend de A ...
  - Cohésion à revoir ...

### ■ Avantages escomptés

- Maintenance
- Substitution d'une implémentation par une autre
- Tests unitaires facilités

### ■ Usage de patrons

- Une solution

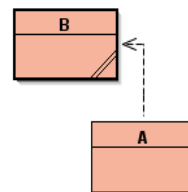
### ■ Un outil de mesure du couplage (parmi d'autres) DependencyFinder

- <http://depfind.sourceforge.net/>

### ■ Un exemple les classes A et B sont en couple

## Exemple : les classes A et B

```
public class A{  
    private B b;  
  
    public A() {  
        this.b = new B();  
    }  
    public void m() {  
        this.b.q();  
    }  
}  
public class B{}
```



A dépend de B → couplage fort de ces classes ...

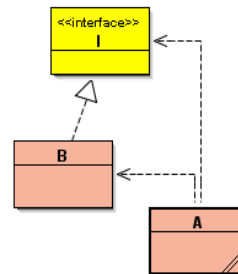
(A extends B idem ...)

## Une interface I, les classes A et B

```
public class A{
    private I i;

    public A(){
        this.i = new B();
    }
}

public class B implements I{}
```



A dépend toujours de B,  
-> la notion d'interface ... ne suffit pas,

à moins que

nous puissions ajouter un paramètre de type I au constructeur de A  
(cela implique une **délocalisation** de B...)

M.Romdhani, Avril 2017

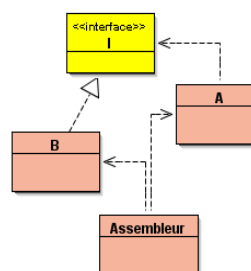
51

## Exemple suite : Une interface I, les classes A, B et +

```
public class A{
    private I i;

    public A(I i){
        this.i = i;
    }
}

public class B implements I{}
```



A ne dépend plus de B,

Nécessité d'un assembleur,

```
I référence = new B();
a = new A( référence);
```

I comme injecteur de référence,

M.Romdhani, Avril 2017

52

## Exemple suite : Une interface I, les classes A, B et +

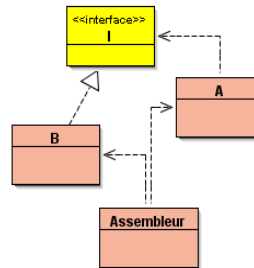
Amélioration de l'architecture des applications

```
public class A{
    private I i;

    public A(I i){
        this.i = i;
    }
}

public class B implements I{}

public class Assembleur{
    Assembleur(){
        A a = new A( new B());
    }
}
```



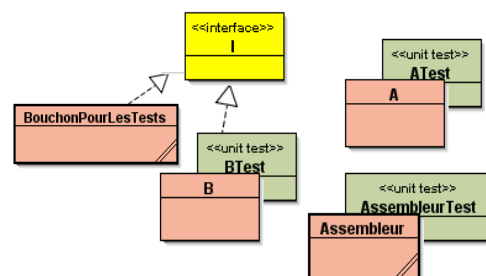
-> Nous injectons une référence lors de l'appel du constructeur de A

M.Romdhani, Avril 2017

53

## Premier Gain : les tests unitaires plus simples

Amélioration de l'architecture des applications



### ■ De vrais tests à l'unité ...

Des *bouchons* pour les tests peuvent être utilisés

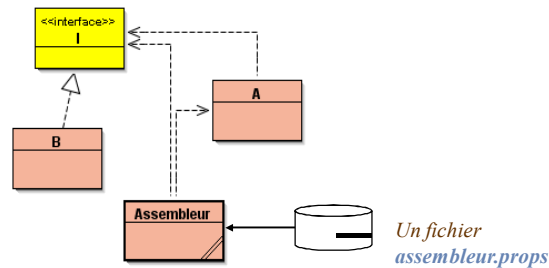
<http://easymock.org/>

M.Romdhani, Avril 2017

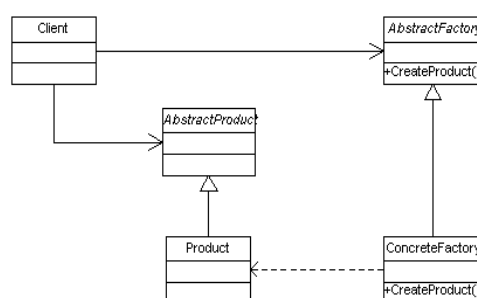
54

## Une variante de l'assembleur

- Depuis un fichier, l'assembleur extrait le nom de la classe
  - Un fichier de configuration



## Et le patron Factory Method ?



- Définit une interface pour la création d'un objet, *mais* laisse aux sous classes le soin de décider quelle classe est à instancier,
  - comme si nous disposions d'un constructeur « dynamique »

## Retour sur le Container ou Assembleur

Amélioration de l'architecture des applications

- **Configuration/Conteneur-Assembleur**
- **Une interrogation légitime**
  - Il devrait bien exister des « conteneurs » tout prêts ...
- **Container as framework**
  - Effectue l'injection de dépendance selon une configuration
    - Séparation effective de la configuration de l'utilisation
  - Contient les instances créées
    - Accès aux instances via le conteneur

M.Romdhani, Avril 2017

57

## Des conteneurs tout prêts

Amélioration de l'architecture des applications

- **Spring**
  - Injection par mutateur
- **picoContainer**
  - Injection par constructeur
- **XWork**
- **HiveMind**
- ...
- **Par annotations ...**
  - Google Guice, EJB3
  - @Inject



M.Romdhani, Avril 2017

58

## L'exemple de Martin Fowler

Amélioration de l'architecture des applications

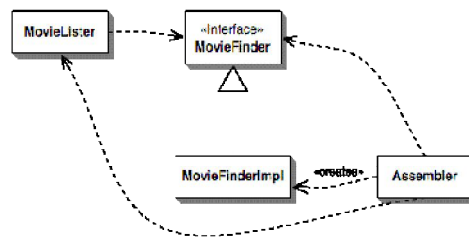
[<http://martinfowler.com/articles/injection.html>] ]

### ■ Dont la lecture semble indispensable

#### ■ Thème :

- La liste des films d'un certain réalisateur de cinéma ...

- À la recherche de films dans une liste sous plusieurs formats
  - Texte, CSV, XML, BDD ...



M.Romdhani, Avril 2017

59

## MovieLister, MovieFinder, Assembler

Amélioration de l'architecture des applications

```

public class Movies {
    private MovieFinder finder;

    public Movies() {
    }

    public void setFinder(MovieFinder finder) {
        this.finder = finder;
    }

    public List<Movie> moviesDirectedBy(director: String) {
        List<Movie> allMovies = finder.findAll();
        for (m in allMovies) {
            if (!movie.getDirector().equals(director))
                allMovies.remove(m);
        }
        return allMovies;
    }
}

public class SemiColonDelimitedMovieFinder implements MovieFinder {
    private String filename;

    public void setFilename(String filename) {
        this.filename = filename;
    }
}
    
```

Déjà vu !

### ■ Assembler as Container

M.Romdhani, Avril 2017

60