



Architectes
de la sérénité

Design Logiciel

Plan

- Introduction
 - Les enjeux du Design
 - Les grands principes du Design
- Design Patterns
- Patterns d'architecture logicielle

Introduction

Introduction

- Enjeux économiques et de production
 - Le design donne des indications sur les bonnes méthodes de travail
 - R.E.M (Robuste, Extensible, Maintenable) par opposition à
 - F.R.I (Fragile, Rigide, Immobile)
- Pas de solution absolue
 - Des principes plus que des règles
 - Des pratiques méthodologiques
 - Des « recettes éprouvées » : les design patterns

Introduction

Les grands principes du design

- OCP (*Open/Close Principle*)
- Substitution (*Liskow Principle*)
- Séparation des rôles (*Interface Segregation Principle*)
- Inversion de contrôle (*IoC Principle*)

Introduction

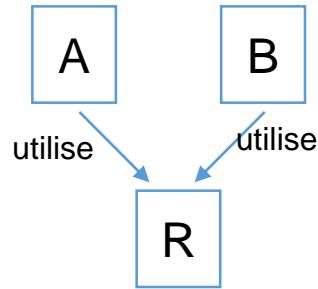
Open/Close Principle

- Bertrand Meyer, 1988
- Tout module (package, classe ou méthode) doit être :
 - ouvert aux extensions : le module peut ajouter de nouveaux comportements ou se comporter différemment lorsque les besoins de l'application changent
 - fermé aux modifications : le code du module ne peut pas être modifié. Seul l'ajout de code est permis.

Introduction

Open/Close Principle

- Etant donné une classe R utilisée par 2 classes A & B
 - A demande une modification du comportement de R
 - B doit accepter cette modification : que se passe-t'il si B ne veut pas de cette adaptation ?



- L'OCP indique que :
 - R ne doit pas modifier le comportement existant
 - R doit ajouter un nouveau comportement pour satisfaire la nouvelle fonctionnalité

Introduction

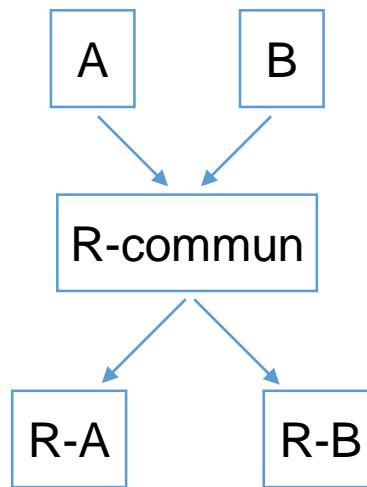
Open/Close Principle

- Séparer la spécification de la réalisation
 - Permet l'élévation du niveau d'abstraction
- Les traits remarquables de la classe sont contenus dans la spécification
- Les détails sont confinés dans les réalisations
 - Occultation des détails == **encapsulation**
- Avantages :
 - Garantir l'intégrité des données
 - Réduire le couplage entre classes

Introduction

Open/Close Principle

- Centralisation du comportement commun dans une classe R-Commun
- Séparation des éléments particuliers dans des parties spécifiques
 - en ajoutant/modifiant des fonctionnalités
- L'héritage est la concrétisation de ce mécanisme
 - application de l'OCP



Introduction

Principe de substitution (Liskow)

- « *Il doit être possible de substituer à n'importe quel objet instance d'une super-classe, n'importe quel objet instance d'une sous-classe sans que la sémantique du programme écrit dans les termes de la super-classe ne soit affectée* » Barbara Liskow, Proceedings of OOPSLA'87
- Un module utilisant une classe doit pouvoir utiliser les sous-classes sans le savoir
 - Toute classe dans la hiérarchie doit honorer tous les rôles de sa classe parente
 - Une classe fille ne doit pas être une restriction de la classe parente

Introduction

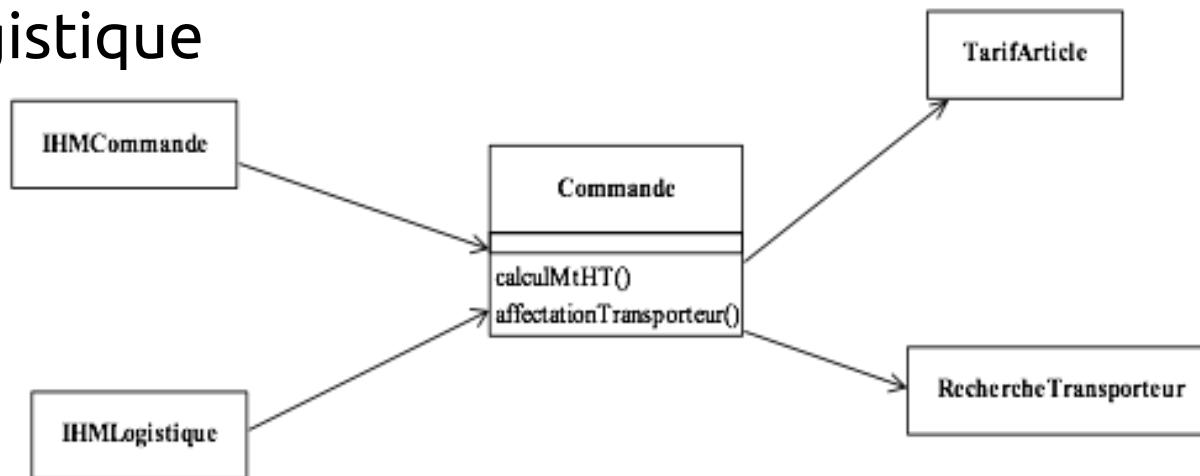
Principe de substitution (Liskow)

- Le polymorphisme
 - Une invocation de méthode déclenchera un traitement différent selon le type de l'objet
 - L'implémentation est choisie par l'objet invoqué
- Les objets doivent collaborer
 - sans connaître leur type réel
 - en traitant ceux du même genre de la même manière

Introduction

Séparation des rôles

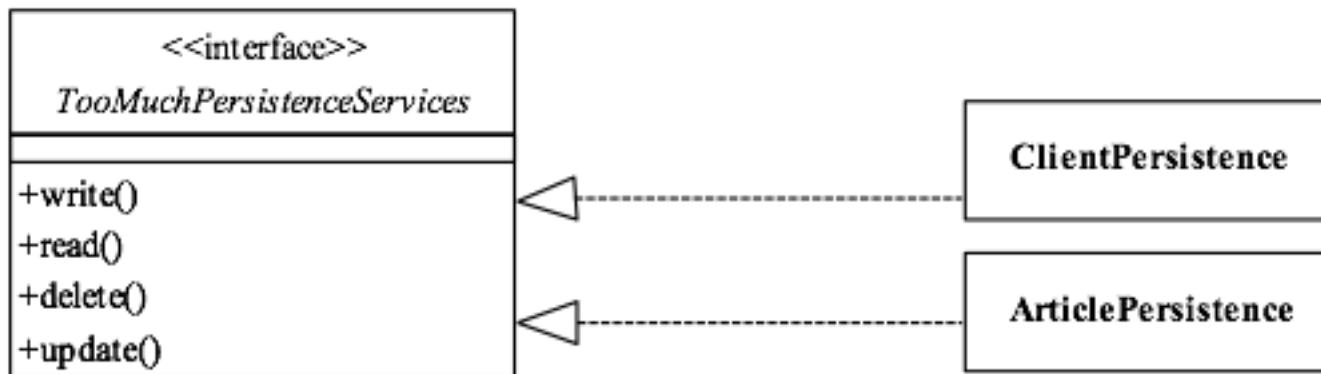
- La responsabilité d'une classe est souvent mal définie (perte de cohésion)
- Une modification de `affectationTransporteur()` entraîne une compilation et une livraison de la classe `IHMCommande`, qui n'est pas concernée
- Impossible de livrer la saisie des commandes sans les classes de la logistique



Introduction

Séparation des rôles

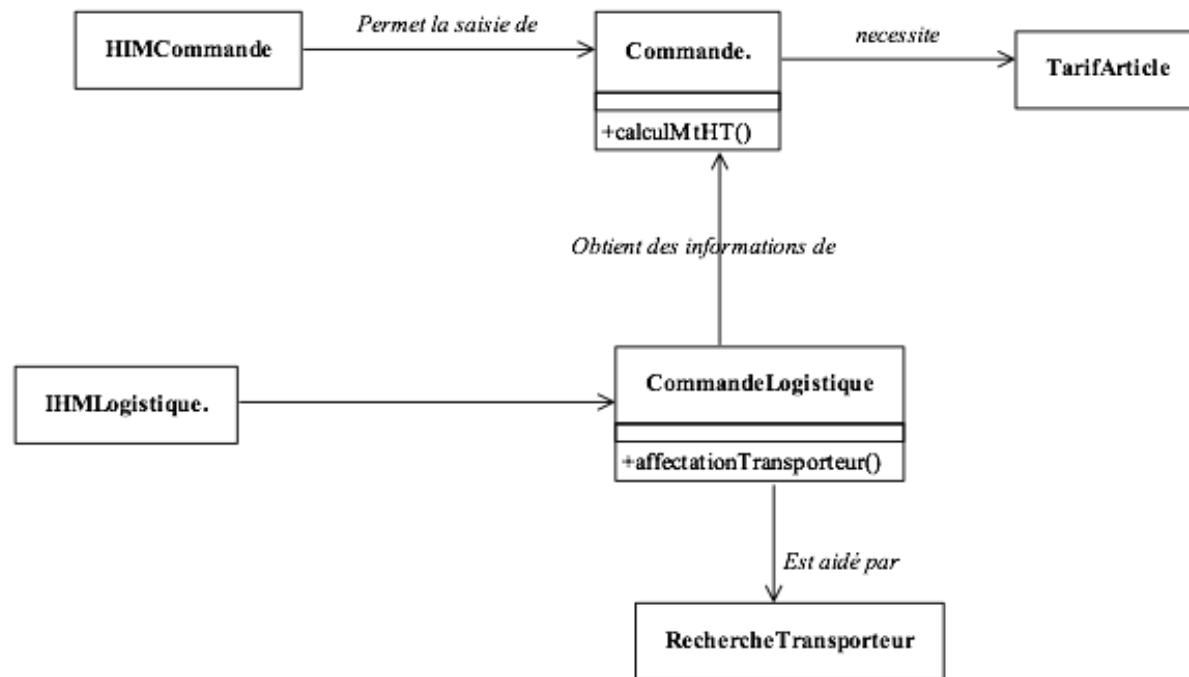
- Une modification de l'interface `delete()` entraîne une modification de `ArticlePersistence`, même s'il ne devrait pas être impacté par la modification
- Dans beaucoup d'architectures applicatives, quelques classes couvrent trop de services
 - Chaque client voit une interface trop riche dont une partie ne l'intéresse pas



Introduction

Séparation des rôles

- Solution : la séparation des services de l'interface
 - ISP : Interface Segregation Principle
 - Chaque classe doit avoir un seul rôle déterminé



Introduction

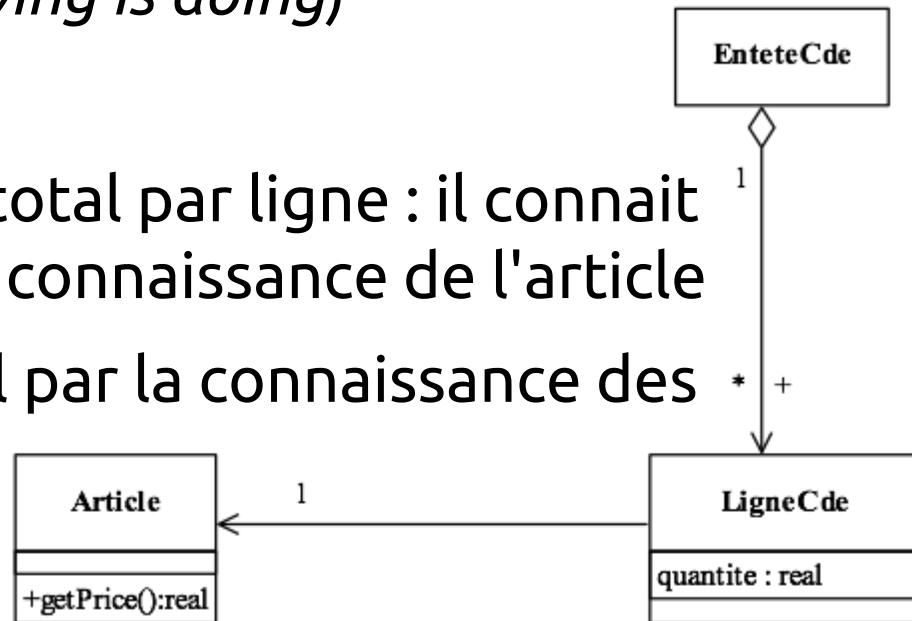
Séparation des rôles

- Comment assigner les rôles aux classes (qui fait quoi) ?
 - General Responsibility Assignment Software Patterns
 - [Graig Larman, "Applying UML and Patterns", 1998]
- "Savoir" (*Knowing*)
 - Connaissance des objets privés
 - Connaissance des objets liés
 - Connaissance de résultats de calculs ou dérivés
- "Savoir Faire" (*Doing*)
 - Faire quelque chose soit même
 - Initialiser les actions d'autres objets
 - Coordonner les activités d'autres objets

Introduction

Séparation des rôles

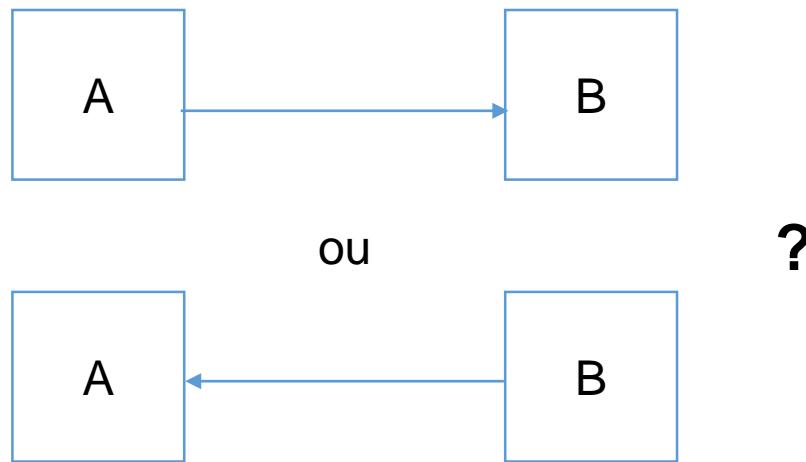
- Quel est le principe pour donner une responsabilité ?
 - Donner la responsabilité à la classe qui connaît l'information permettant de la réaliser
 - « *Celui qui sait, fait* » (*Knowing is doing*)
- LigneCde : calcule le sous-total par ligne : il connaît la quantité et le prix par la connaissance de l'article
- EnteteCde : calcule le total par la connaissance des lignes



Introduction

Gestion des dépendances / Inversion de contrôle

- Davantage de classes = davantage de dépendances
 - Nécessité de mettre de l'ordre
 - Toujours tendre vers le plus faible couplage possible



Introduction

Gestion des dépendances / Inversion de contrôle

- Les modules de haut niveaux ne doivent pas dépendre des modules de bas niveaux
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions
 - Robert C. Martin, « *Object Oriented Design Quality Metrics* », 1995
 - C'est le *principe d'inversion de contrôle*
- Une abstraction est une interface
- Un détail est un objet
- Un objet abstrait est de niveau intermédiaire

Introduction

Gestion des dépendances / Inversion de contrôle

- Plus un composant est proche du fonctionnel, plus il est de haut niveau
 - Un module fonctionnel ne doit pas dépendre des niveaux présentation ou physique
- La démarche de construction est la suivante :
 - définir ce que les composants de haut niveau demandent aux autres couches (par ex. écrire)
 - concrétiser ces besoins par des interfaces (par ex. Writer)
 - Implémenter cette interface dans les modules de bas-niveaux

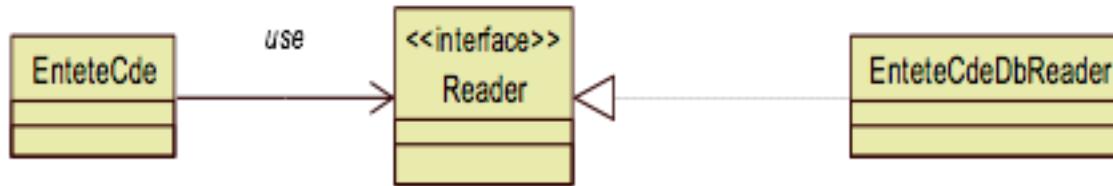
Introduction

Gestion des dépendances / Inversion de contrôle

- Principe de Hollywood
 - « do not call us, we'll call you »
- L'inversion de contrôle est un terme générique
 - plusieurs représentations existent
 - la plus connue : injection de dépendances
- Les objets ne vont pas « chercher » leurs dépendances
 - elles leur sont fournies par un tiers

Introduction

Gestion des dépendances / Inversion de contrôle



```
public class EnteteCde {  
    Reader reader;  
    public void lire() {  
        reader.lire();  
    }  
    public void setReader(Reader thereader) {  
        reader=thereader;  
    }  
}  
EnteteCde enteteCde = new EnteteCde("123");  
Reader reader = new EnteteCdeReader();  
enteteCde.setReader(reader);  
enteteCde.lire();
```

Introduction

Gestion des dépendances / Inversion de contrôle

- Couplage faible
 - Facile de remplacer des composants
 - Maintenance simplifiée
- Implémentations indépendantes des contextes d'utilisation
 - Composants réutilisables
- Développement incrémental et modulaire
- Tests simplifiés
 - Dépendances déjà isolées
 - Mock-objects (bouchons)

Introduction

Enjeux des grands principes du Design

- Construire un système capable d'évoluer
- En maximisant la réutilisation
- Pour des gains de qualité, de productivité et de maintenabilité
- Pas de recette miracle
- Appréhender ces principes pour se poser les bonnes questions
 - En conservant une approche KISS, DRY, YAGNI
- Adopter une démarche par le Design
- La POO permet intrinsèquement d'aborder cette démarche

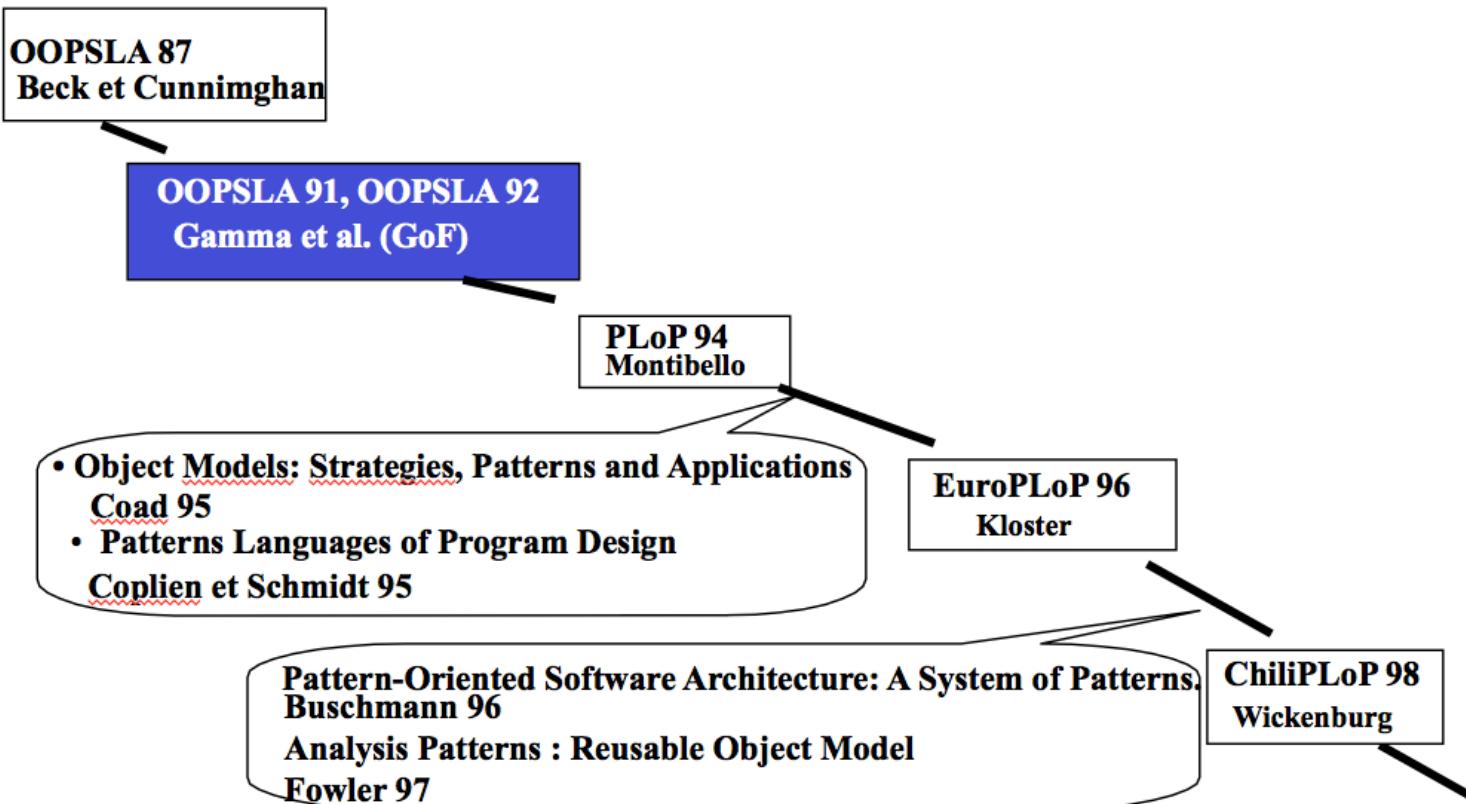
Design Patterns

Design Patterns

- Les design patterns (ou modèles de conception) décrivent des organisations pratiques de classes d'objets, reconnues pour apporter des solutions valables à certains problèmes de conception.
- A l'origine, ces organisations de classes résultent de recherches expérimentales, le concepteur objet tentant de faciliter la réutilisation et la maintenance du code.
- On peut donc considérer un modèle de conception comme une forme d'organisation transposable à plusieurs applications.

Design Patterns

Historique de la notion de « pattern »



Design Patterns

- Les patterns sont des figures types de relations entre des classes pour répondre à un problème donné.
- Les patterns sont au design ce que l'algorithme est à la programmation
- Il existe 23 patterns historiques écrits par la « bande des quatres » en 1995 : il s'agit des patterns du *Gof*
 - [Erich Gamma, Richard Helm, Ralph Johnson, John Wlissides, "*Design Pattern. Elements of Reusable Object-oriented software*", 1995]

Design Patterns

Définition

« Un Design pattern systématiquement nomme, motive et explique un design général qui répond à un problème fréquent dans système orienté objet.

Il décrit le problème, la solution, quand appliquer la solution et ses conséquences.

Il donne aussi des conseils d'implémentation et des exemples. La solution est un dessin général des relations entre objets et classes qui résolvent le problème.

La solution est personnalisée et implémentée pour résoudre le problème dans un contexte particulier »

[GOF95]

Design Patterns

En clair

- Solutions simples et élégantes pour résoudre des problèmes donnés à l'aide d'un langage objet
- Développés avec un objectif d'une meilleure réutilisation et d'une meilleur flexibilité
- Indépendants des langages
- Les patterns sont des unités de raisonnements
 - Ces "unités" sont et doivent être adaptées aux contextes particuliers ...

Les patterns sont construits par problème à résoudre (intention) et non par structure !

Design Patterns

Les différents patterns

- Patterns d'architecture
 - schémas d'organisation structurelle de logiciels (pipes, filters, brokers, blackboard, MVC, ...)
- Design Patterns
 - caractéristiques clés d'une structure de conception commune à plusieurs applications, de portée plus limitée que les architectural patterns
- Idioms ou coding patterns
 - solution liée à un langage particulier
- Anti-patterns
 - mauvaise solution ou comment sortir d'une mauvaise solution
- Organizational patterns
 - organisation de tout ce qui entoure le développement d'un logiciel (humains)

Design Patterns

Schéma de présentation d'un pattern

- Introduction
 - Un nom: Permet la communication entre développeur
 - Une catégorie: Création, structure, comportement
- Le problème
 - Une intention: Description rapide
 - Une Motivation: Le problème à résoudre (un scénario concret)
 - Champs couvert (applicability): Explique quand appliquer le pattern
- La Solution
 - La Structure: Décrit les éléments participants au pattern (Diagramme de classe)
 - Les rôle : Rôle de chacun des participant
 - Les collaborations: (Diagramme de séquences)
- Critique et exemple
 - Les conséquences: Décrit l'intérêt de la solution, ses avantages et ses coûts, ses variantes, ...
 - Les implémentations: Dangers, Variantes entre les différents langages, ..
- Exemple de code:
 - Les implémentations connues: exemples dans le JDK, ...
 - Les patterns liés: Patterns proches ou utilisés

Design Patterns

Comment choisir un Design Pattern ?

- La bonne pratique : Ignorer les Design Patterns avant d'avoir des vrais problèmes
- Il faut partir de ces problèmes et ne choisir que parmi des familles de solutions adaptées à NOS problèmes

Design Patterns

Des problèmes ?

- Comment faire un système d'export de données XML/texte/binaire vers n formats ?
- Comment minimiser la duplication de code lorsque l'on gère une hiérarchie d'éléments qui se contiennent l'un l'autre (analogie système de fichier)
- Comment extirper de son code une dépendance vers une classe (un package) concrète dont on utilise un élément ?
- Comment appliquer une opération à une collection d'élément de types différents ?
- ...

Design Patterns

Comment choisir un Design Pattern (suite) ?

- Prendre en considération la manière dont les Design Patterns résolvent des problèmes de conception
- Explorer systématiquement les sections intention
- Etudier les relations entre Patterns peut aider à l'utilisation d'un groupe de Patterns
- Etudier les Patterns selon leur classe (Créateur, Structure, Comportement)
- Regarder ce qui peut être modifié sans remettre en cause toute l'architecture

Design Patterns

Design Patterns du Gof

- Classés par domaine fonctionnel :
 - Patterns de création
 - Patterns structurels
 - Patterns comportementaux
- Distinction également en fonction de la notion de portée:
 - Portée classe
 - Portée Objet

Design Patterns

Design Patterns JavaEE

- Plus récents, mise en situation des patterns du GOF en partie
- Classés par couche applicative
 - Couche Présentation
 - Couche Business
 - Couche Intégration
- Il existe de nombreux autres patterns

Quelques Design Patterns Gof

Design Patterns

Les Patterns de création

- Formes de création :
 - Abstraire le processus d'instanciation.
 - Rendre indépendant de la façon dont les objets sont créés, composés, assemblés, représentés.
 - Encapsuler la connaissance de la classe concrète qui est instanciée.
 - Cacher ce qui est créé, qui crée, comment et quand.

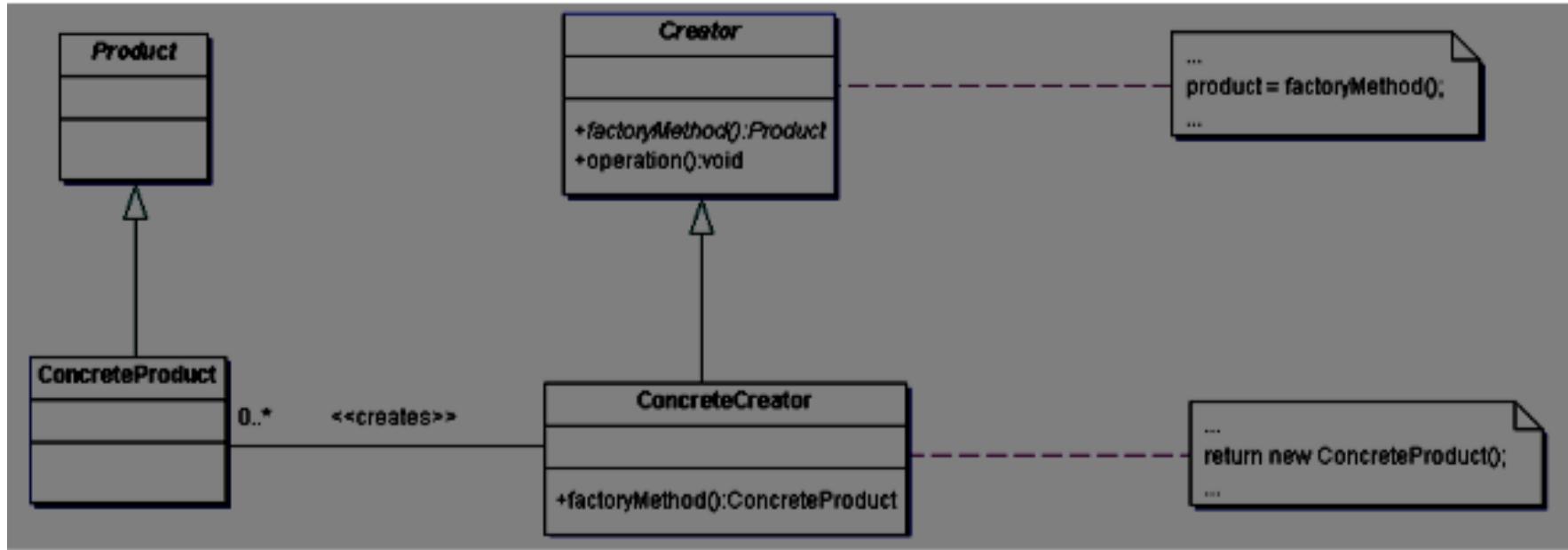
Design Patterns

Les Patterns de création

- Factory Method
 - Fournit une interface pour la création d'objets. Découpe l'utilisation d'un objet de sa création.
- Abstract Factory
 - Similaire pour la création de familles d'objets.
- Singleton
 - Garantit l'unicité de l'instance d'un classe.
- Builder
 - Création d'une variété d'objets complexes issue d'une source.
- Prototype
 - Création d'objet par copie d'objets existants.

Design Patterns

Factory Method



Product: l'interface des objets créés par la «factory method»

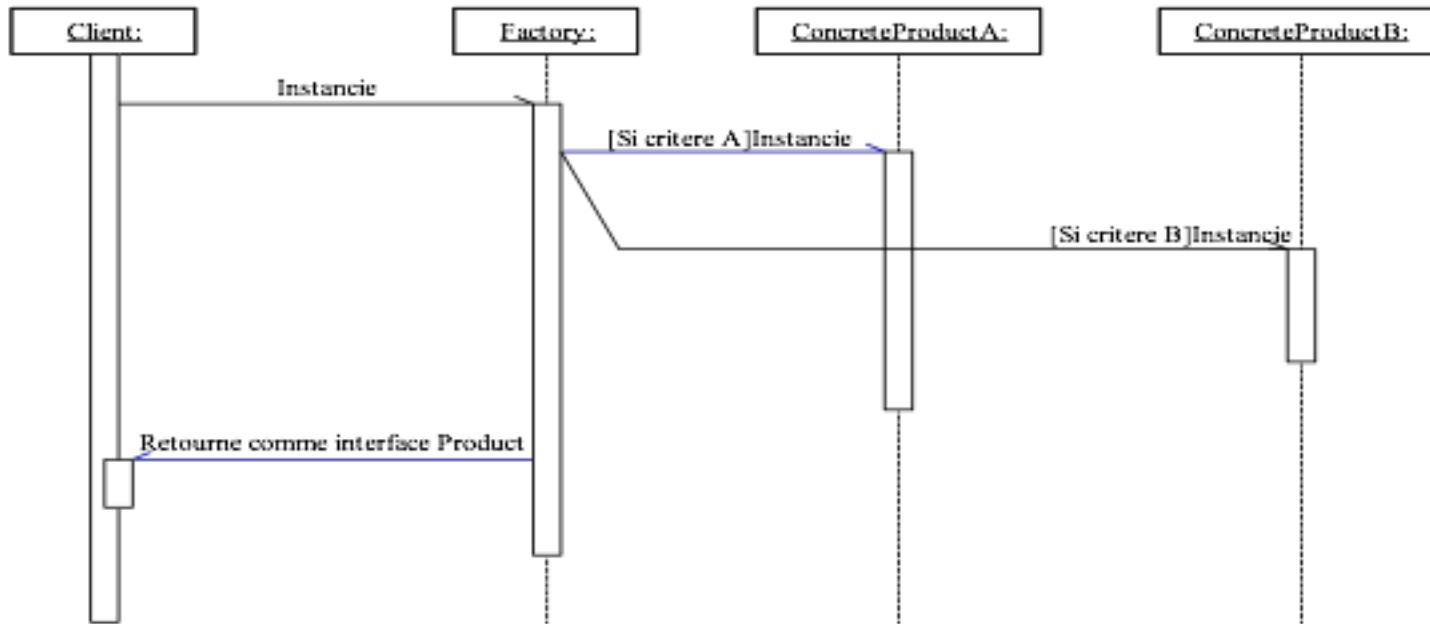
ConcreteProduct: réalise l'interface

Creator: Factory déclarant la «factory method» et l'utilise pour manipuler des produits

Concrete Creator: surcharge ou réalise la «factory method». **factoryMethod()** est responsable de la fabrication d'un objet

Design Patterns

Factory Method



Collaboration

Le client cherche à utiliser une interface "Product". Il en fait la demande à une classe appelée "Factory"

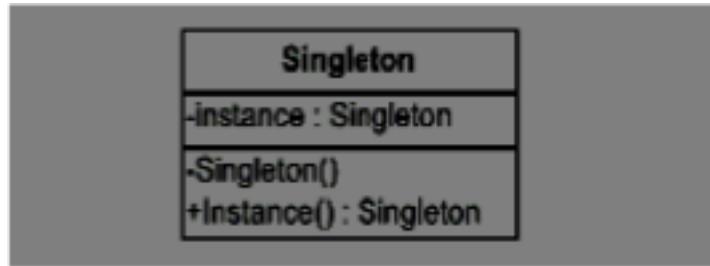
La classe "Factory" crée (en fonction des paramètres en entrée) un nouvel objet "ConcreteProduct" implémentant l'interface attendue "Product". Elle renvoie cet objet au client

Design Patterns

Singleton

Intention

Permet de créer une seule instance de classe. Permet de garantir qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe.



Design Patterns

Singleton par l'exemple (approche simpliste)

```
public class LigneCommandeFactory {  
    private static LigneCommandeFactory instance;  
  
    private LigneCommandeFactory {  
        // Quelque chose si nécessaire  
    }  
  
    public static final LigneCommandeFactory getInstance() {  
        if (instance==null)  
            instance = new LigneCommandeFactory ();  
  
        return instance;  
    }  
    ...  
}
```

Design Patterns

Singleton par l'exemple : dangers

- Non prise en charge de la concurrence d'accès
 - Quid en cas de multi-thread ?
- Solutions :
 - Synchronisation : coût
 - Synchronisation fine : risque
 - Pas de contrôle sur le moment de l'instanciation (tout est-il prêt ?)

Design Patterns

Singleton par l'exemple : singleton protégé

```
public class LigneCommandeFactory {  
  
    private static LigneCommandeFactory instance;  
  
    private LigneCommandeFactory {  
        // Quelque chose si nécessaire  
    }  
  
    public static final LigneCommandeFactory getInstance() {  
        if (instance==null)  
        {  
            synchronized(LigneCommandeFactory.class)  
            { instance = new LigneCommandeFactory(); }  
        }  
  
        return instance;  
    }  
  
    ...  
}
```

Design Patterns

Singleton par l'exemple : singleton doublement protégé

```
public class LigneCommandeFactory {  
  
    private static LigneCommandeFactory instance;  
  
    private LigneCommandeFactory {  
        // Quelque chose si nécessaire  
    }  
  
    public static final LigneCommandeFactory getInstance() {  
        if (instance==null)  
        {  
            synchronized  
            {  
                if (instance == null) {  
                    instance = new LigneCommandeFactory();  
                }  
            }  
        }  
        return instance;  
    }  
    ...  
}
```

Design Patterns

Singleton par l'exemple : singleton finalement protégé

```
public class LigneCommandeFactory {  
    private static LigneCommandeFactory instance = null;  
    private static ThreadLocal initHolder = new ThreadLocal();  
  
    private LigneCommandeFactory {  
        // Quelque chose si nécessaire  
    }  
  
    public static final LigneCommandeFactory getInstance() {  
        if (initHolder.get() == null) {  
            synchronized {  
                if (instance==null) {  
                    instance = new LigneCommandeFactory();  
                    initHolder.set(Boolean.TRUE);  
                }  
            }  
        }  
        return instance;  
    }  
    ...  
}
```

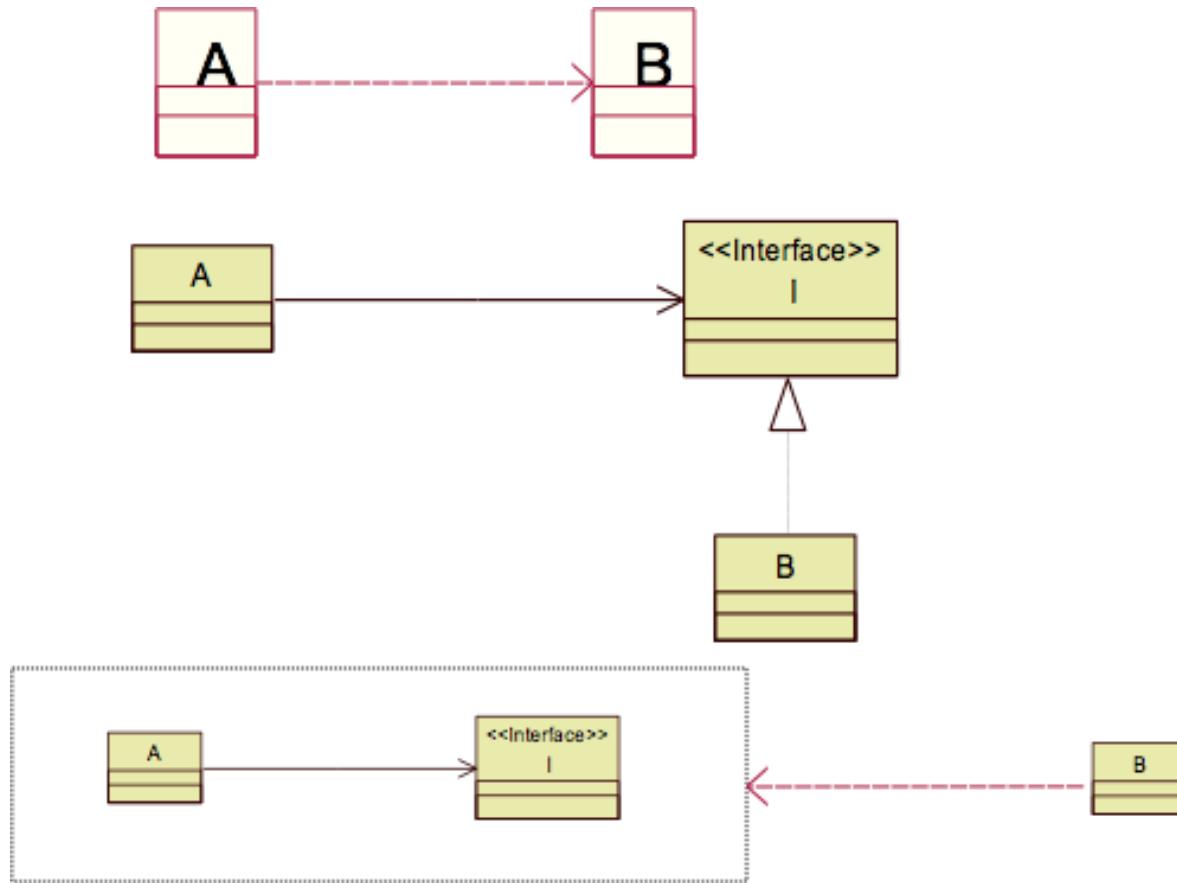
Design Patterns

Singleton par l'exemple : conclusion

- Un pattern peut être la cause de dégâts immenses
- L'implémentation d'un pattern requiert sa compréhension et la maîtrise de la cible technique
- L'utilisation systématique est un non sens

Introduction

IoC pattern



Introduction

Design Pattern : IoC

- Mise en oeuvre
 - Les dépendances du bean seront satisfaites après sa création
 - par les paramètres des constructeurs
 - ou par appel de setters après instantiation
 - Un conteneur de bean peut injecter ces dépendances
 - C'est le rôle des conteneurs dits « légers »
 - Spring, Guice, Weld



Introduction

IoC par l'exemple

- Exemple
 - Un chat veut jouer...
- Trois approches :
 - de base, sans injection
 - avec injection manuelle
 - avec injection par conteneur

Introduction

IoC par l'exemple : sans IoC

Utilisation :

```
Chat monChat = new Chat();
monChat.jouer();
```

```
package com.injection.none;
import com.injection.none.jeu.Souris;

public class Chat {
    private Souris souris = new Souris();
    public void jouer() {
        souris.jouer();
    }
}
```

Introduction

IoC par l'exemple : sans IoC

- Bilan
 - Sans injection de dépendances :
 - Modélisation fermée
 - Le chat ne peut jouer qu'avec une souris
 - Fort couplage (relation, création, utilisation)
 - Dépendance vis-à-vis de la souris
 - Impossibilité de changer de jouet sans recompiler le chat

Introduction

IoC par l'exemple : injection manuelle

Utilisation :

```
Chat monChat = new Chat();

//Souris implémente IJouet
IJouet jouet = new Souris();
```

```
package com.injection.with;
import com.injection.with.jeu.IJouet;

public class Chat {
    private IJouet jouet;
    public void jouer() {
        jouet.jouer();
    }
    public void setJouet(IJouet jouet) {
        this.jouet = jouet;
    }
}
```

```
monChat.setJouet(jouet);
monChat.jouer();
```

Introduction

IoC par l'exemple : injection manuelle

- Bilan
 - Avantages de l'injection manuelle :
 - Le chat expose sa dépendance avec setJouet()
 - Aucune dépendance vers des implémentations
 - Il peut donc jouer avec n'importe quel jouet
 - Inconvénients
 - L'utilisation est figée (recompilation nécessaire pour changer de jouet)

Introduction

IoC par l'exemple : injection avec conteneur

- La modélisation est la même
- Seule l'utilisation change :
 - Configuration en XML ou par annotations
 - Chargement du contexte
 - Récupération du chat

Introduction

IoC par l'exemple : injection avec conteneur

Configuration XML (exemple : Spring) :

```
<beans>
    <bean id="leJouet" class="com.injection.with.jeu.Souris" />
    <bean id="chat" class="com.injection.with.Chat">
        <property name="jouet" ref="leJouet" />
    </bean>
</beans>
```

Introduction

IoC par l'exemple : injection avec conteneur

Configuration par annotations (normalisées avec JavaEE6)

```
public class Chat {  
    @Inject  
    private IJouet jouet;  
}
```

Introduction

IoC par l'exemple : injection avec conteneur

- Bilan
 - Mêmes avantages que l'injection manuelle
 - Tout est paramétré
 - Il faut tout de même configurer l'injection!
 - Elle est centralisée en XML
 - Elle est type-safe en annotations
 - Il n'est pas même pas nécessaire de re-compiler en configuration XML

Design Patterns

Les Patterns structurels (wrappers)

- Adapter
 - Permet de convertir une interface d'une classe en une autre interface, utile à une classe cliente spécifique.
- Decorator
 - Permet de modifier dynamiquement les fonctionnalités d'un objet.
- Proxy
 - Crée un substitut simple et léger d'un objet complexe. Utile quand la création d'un objet et son utilisation effective ne coïncide pas.

Design Patterns

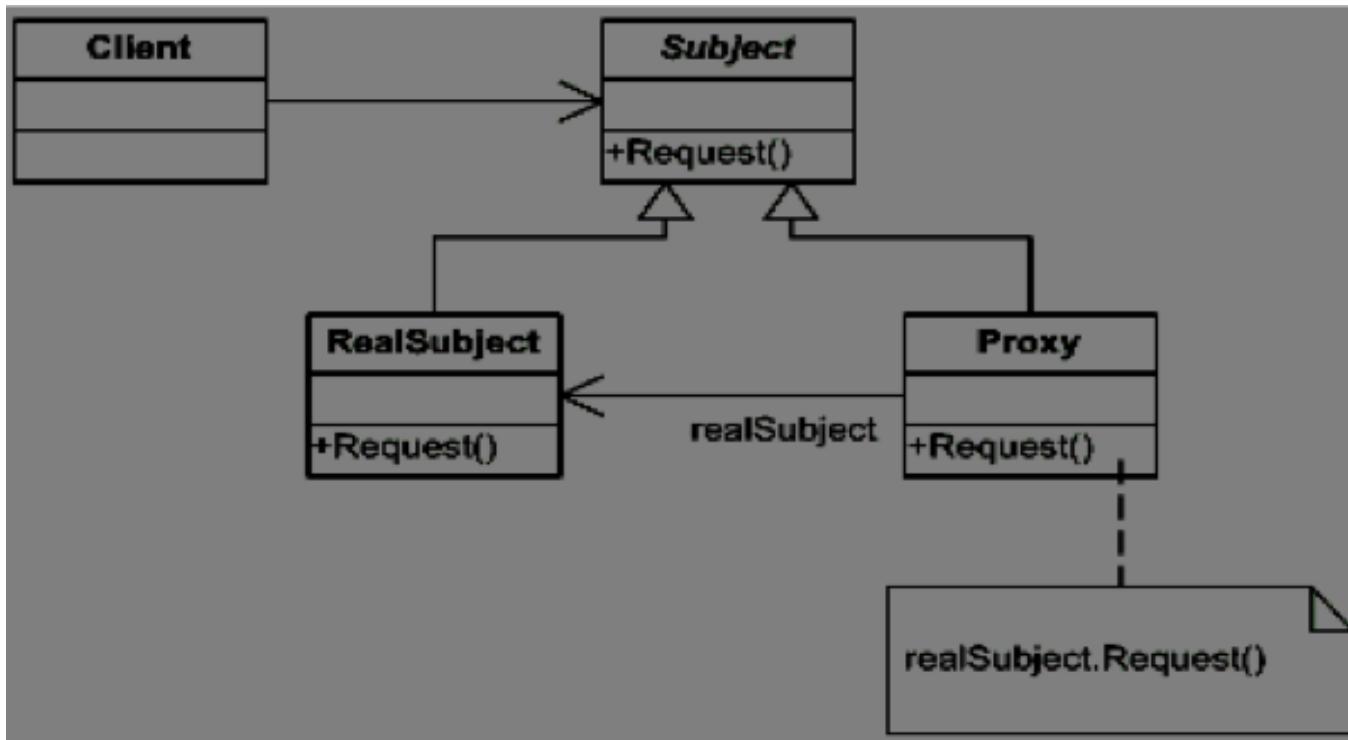
Les Patterns structurels (wrappers)

- Flyweight
 - Partage de petits objets en grand nombre
- Bridge
 - découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre.
- Composite
 - Construit un arbre d'objet simples et composites, utile dans les IHM avec notion de conteneurs.
- Facade
 - Construction d'une interface pour un sous-système qui masque sa complexité.

Design Patterns

Proxy

- Ce pattern fournit un remplaçant d'un objet pour en contrôler l'accès



Design Patterns

Proxy Pattern

- Subject
 - Définit l'interface commune pour le Sujet réel et le proxy.
Ainsi un proxy peut être utilisé n'importe où un Sujet réel est attendu
- RealSubject
 - Définit le Sujet Réel représenté par le proxy

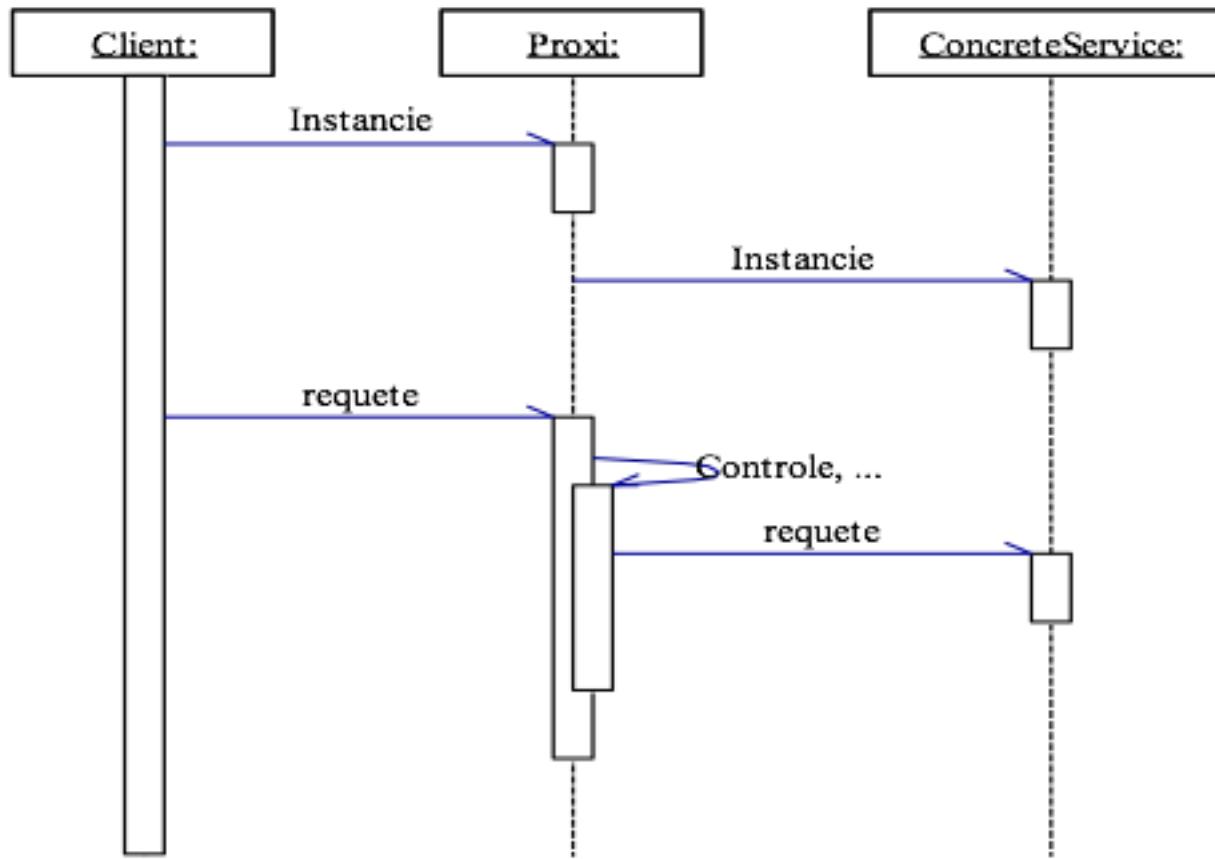
Design Patterns

Proxy Pattern

- Le proxy maintient une référence qui lui permet d'accéder au sujet réel. Le proxy peut faire référence au Sujet si l'interface Sujet Réel et l'interface Sujet sont les mêmes.
- Le proxy fournit une interface identique à celle du sujet ainsi le proxy peut se substituer aux sujets réels.
- Le proxy contrôle l'accès au Sujet réel et peut être responsable de sa création et de sa suppression
- D'autres responsabilités dépendent du type de proxy:
 - Les *Remote Proxies* sont responsable de l'encodage de requête et du fait de l'envoyer au Sujet réel dans un autre espace d'adressage
 - Les *Virtual Proxies* peuvent cacher plus d'informations sur le Sujet réel ainsi ils peuvent différer le moment d'y accéder.
 - Les *Protection Proxies* vérifient que l'appelant a les permissions d'accès requises pour effectuer la requête.

Design Patterns

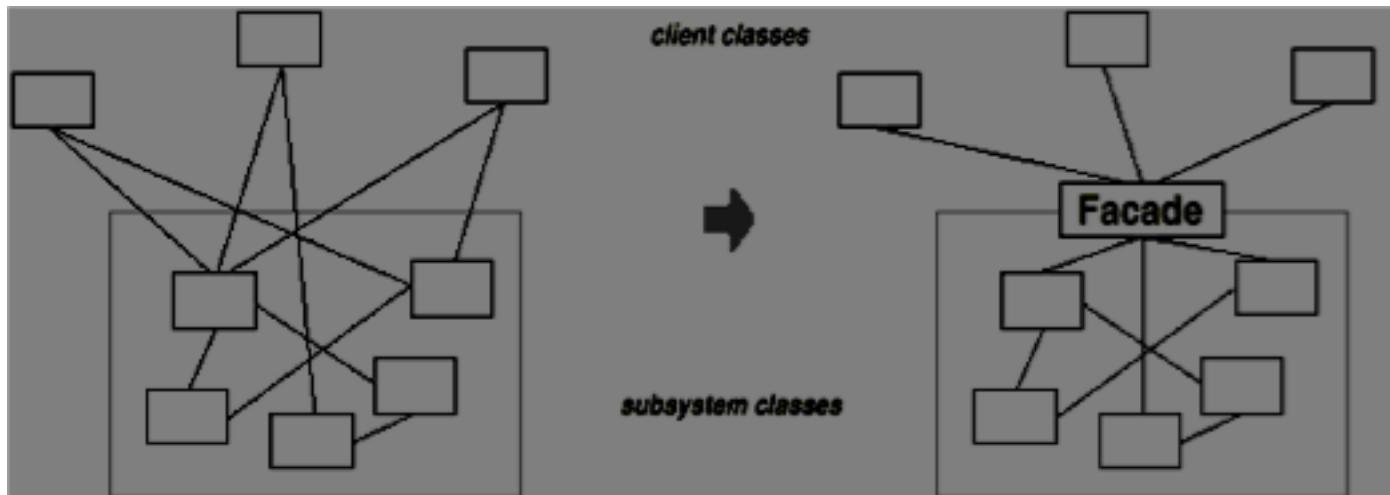
Proxy Pattern



Design Patterns

Façade

- Ce pattern fournit une interface unique pour l'ensemble d'un sous-système



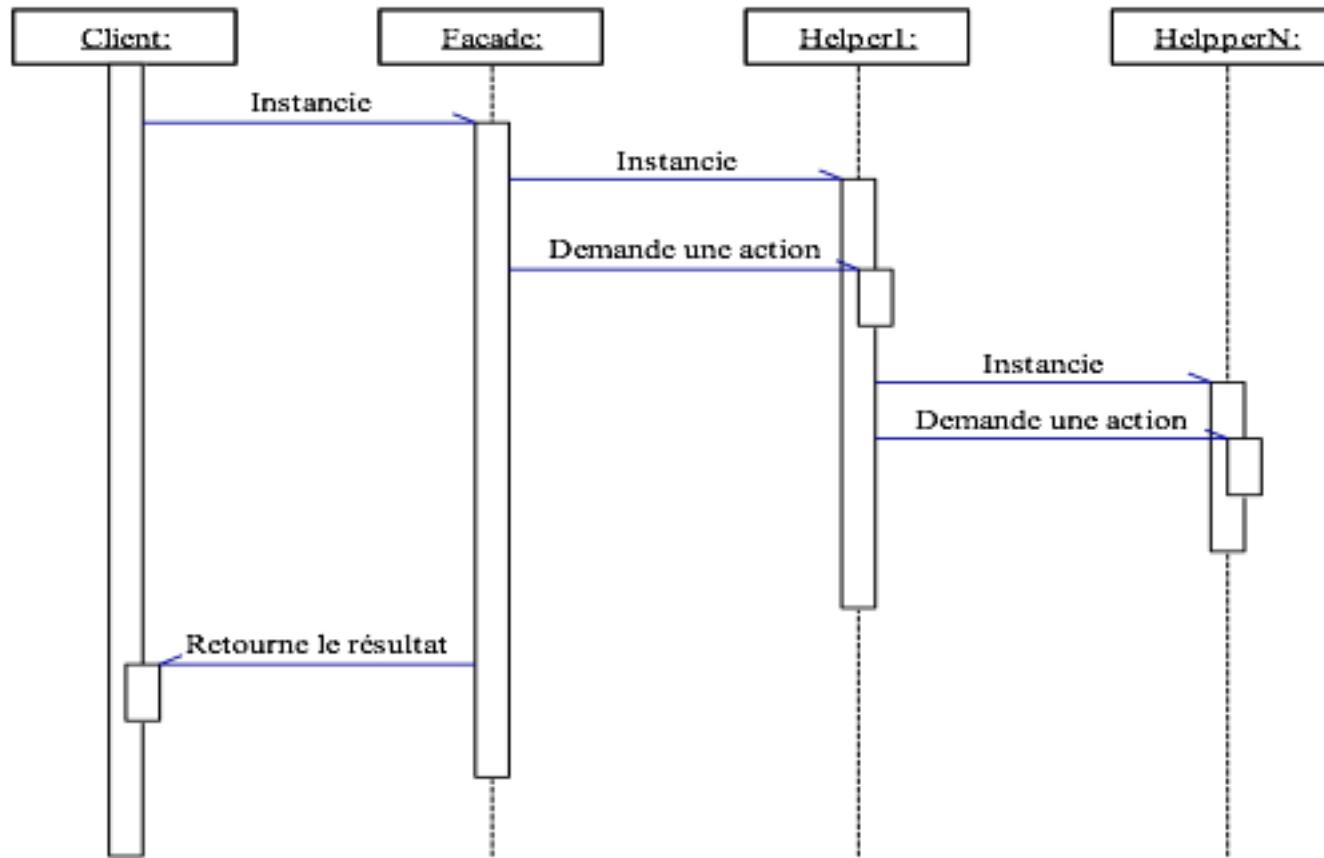
Design Patterns

Façade

- On utilise Façade lorsque on veut :
 - fournir une interface simple à un système complexe
 - introduire une interface pour découpler les relations entre deux systèmes complexes
 - construire le système en couche

Design Patterns

Façade



Design Patterns

Les Patterns comportementaux

- Chain of responsibility
 - Permet de passer un objet à travers une chaîne d'objet jusqu'à ce qu'un objet approprié puisse le traiter.
- Command
 - Permet typiquement d'implémenter le mécanisme «Undo» pour une liste de commande.
- Interpreter
 - Utilisé par les parsers, analyse syntaxique
- Iterator
 - Parcours une collection d'objet
- Mediator
 - Encapsule une logique d'interaction entre objets en évitant que ceux-ci n'ait à se référer les uns les autres.

Design Patterns

Les Patterns comportementaux

- Memento
 - Stockage temporaire d'un état partiel d'un objet. Permet de le reconstruire par la suite.
- Observer
 - Permet à une classe d'être notifiée du changement d'état d'une autre classe.
- State
 - Permet à un objet de modifier son comportement selon son état.
- Strategy
 - Permet d'encapsuler une famille d'algorithmes interchangeable

Design Patterns

Les Patterns comportementaux

- Template
 - Permet de découper un algorithme complexe en petits morceaux. Une classe principale définit la structure et d'autres classes implémentent les opérations.
- Visitor
 - Permet d'ajouter fictivement une méthode à une classe sans pour autant modifier sa structure.

Design Patterns

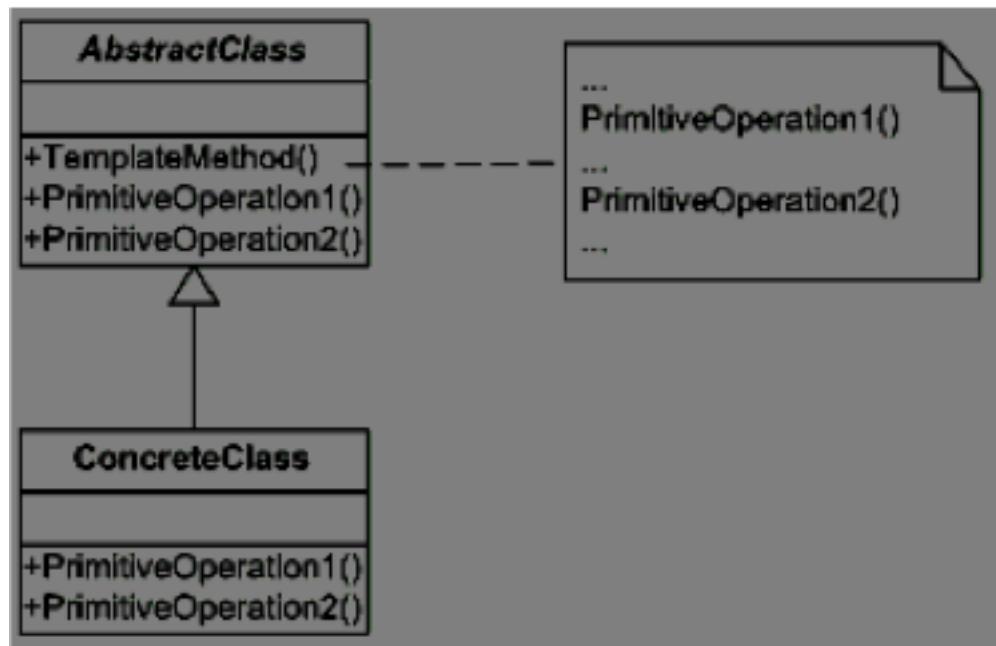
Template

- Définition d'un squelette d'algorithme dont certaines étapes sont fournies par une classe dérivée.
- Utilisé lorsqu'il est nécessaire d'implémenter une fois pour toute les parties invariantes d'un algorithme et de laisser aux sous-classes de soin d'implémenter les parties dont le comportement est conçu comme modifiable.

Design Patterns

Template

- **AbstractClass:**
 - définit des opérations primitives abstraites que les sous-classes concrètes surchargent pour implémenter les étapes de l'algorithme
 - Implémente une `TemplateMethod` qui définit le squelette de l'algorithme
- **ConcreteClasse**
 - implémente les opérations primitives



Quelques Design Patterns JavaEE

Design Patterns

Introduction aux Design Patterns JavaEE

- Les Design Patterns JavaEE s'appuient sur les Best Practices.
- Sun a émis des recommandations pour éviter de commettre des erreurs de conception et de réinventer systématiquement la roue.
- En utilisant les DP JEE, on s'assure d'utiliser une architecture multi-tiers.
- Les Design Patterns JEE donnent des solutions pour utiliser les composants métiers (par ex: les EJBs) de manière optimale.
- Ils s'appuient sur les principes de GoF.
- Les Frameworks de présentation (comme Apache Struts) , de mapping O/R (comme Hibernate) implémentent les Design Patterns JavaEE.

Design Patterns

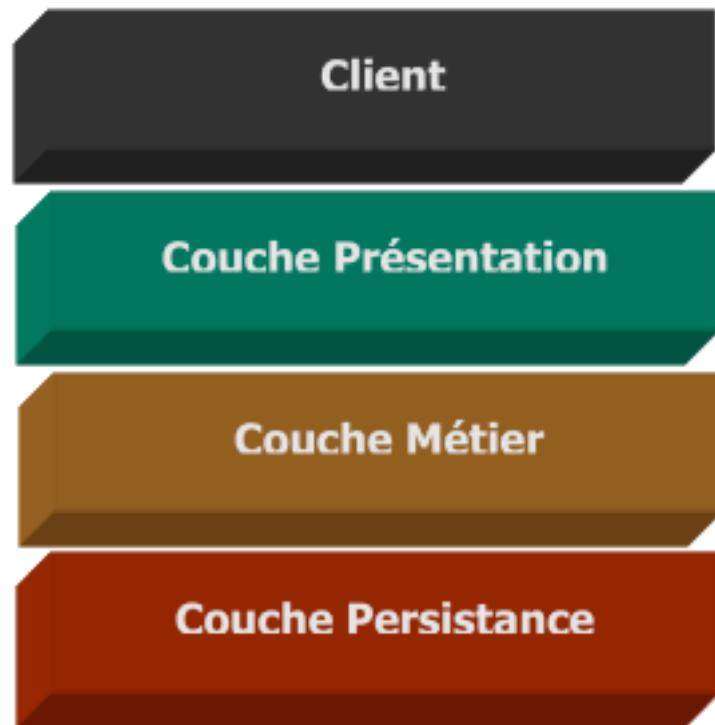
Introduction aux Design Patterns JavaEE

- Concernant le monde JavaEE, Sun a entrepris une démarche de formalisation assez conséquente
 - Répartition des Design Patterns selon plusieurs couches dans l'architecture applicative.
- Cette classification reprend bien évidemment des patterns existants, mais intègre surtout des patterns issus des travaux de la communauté Java sur le développement d'applications Web.
- Les API fournies par Sun (et maintenant Oracle) sont bien évidemment exploitées au maximum dans l'élaboration de stratégies d'implémentation de ces patterns, et parfois même sont à l'origine de la naissance de certains patterns.

Design Patterns

Présentation du modèle en couches

- Les DP Java EE sont répartis en couches



Client
Navigateur Web ou interface client Riche

Couche Présentation
Patterns J2EE responsables de l'affichage des pages

Couche Métier
Patterns J2EE représentant le modèle objet

Couche Persistance
Patterns J2EE responsables de l'accès aux données

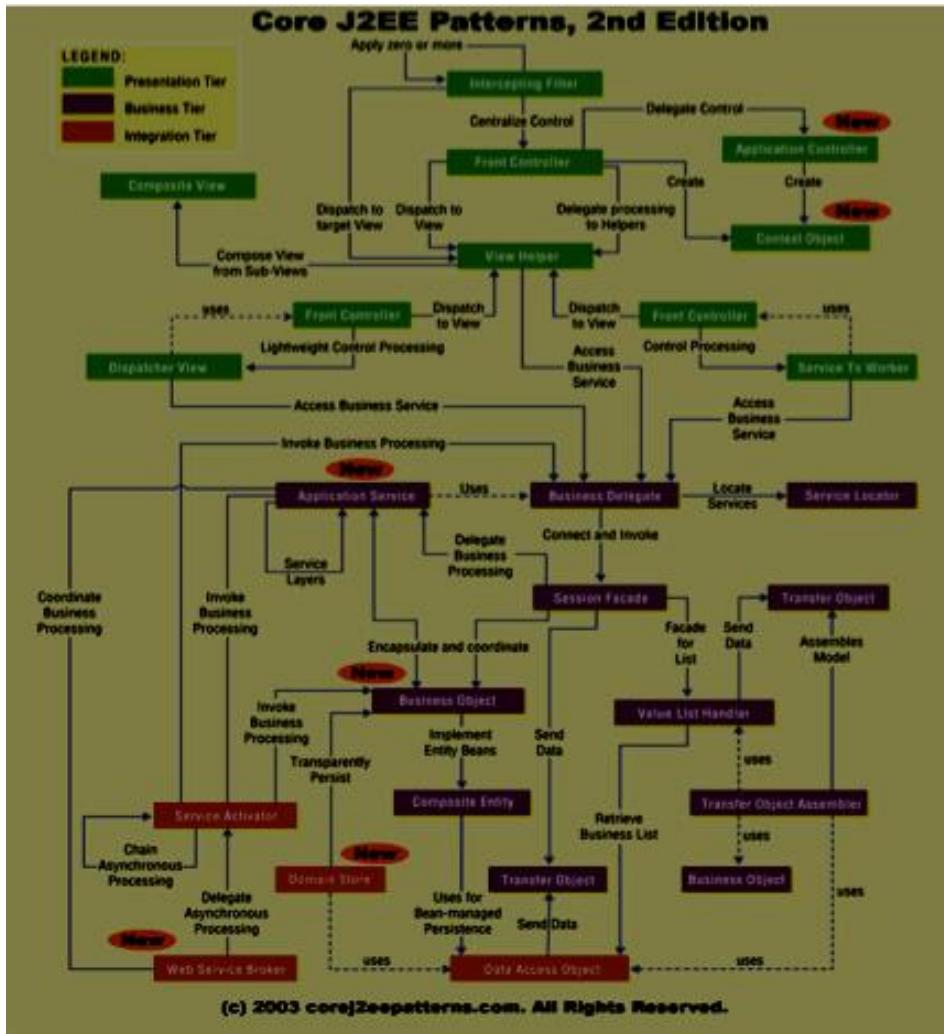
Design Patterns

Synthèse des Design Patterns JavaEE

Couche	Nom du Pattern
Couche Présentation	Intercepting Filter Front Controller Composite View View Helper Service to Worker Dispatcher View Context Object
Couche Métier	Application Controller Business Delegate Session Facade Service Locator Transfer Object Composite entity Transfer Object Assembler Value List Handler Business Object
Couche Intégration	Application Service Data Access Object Service Activator Domain Store Web Service Broker

Design Patterns

Synthèse des Design Patterns JavaEE



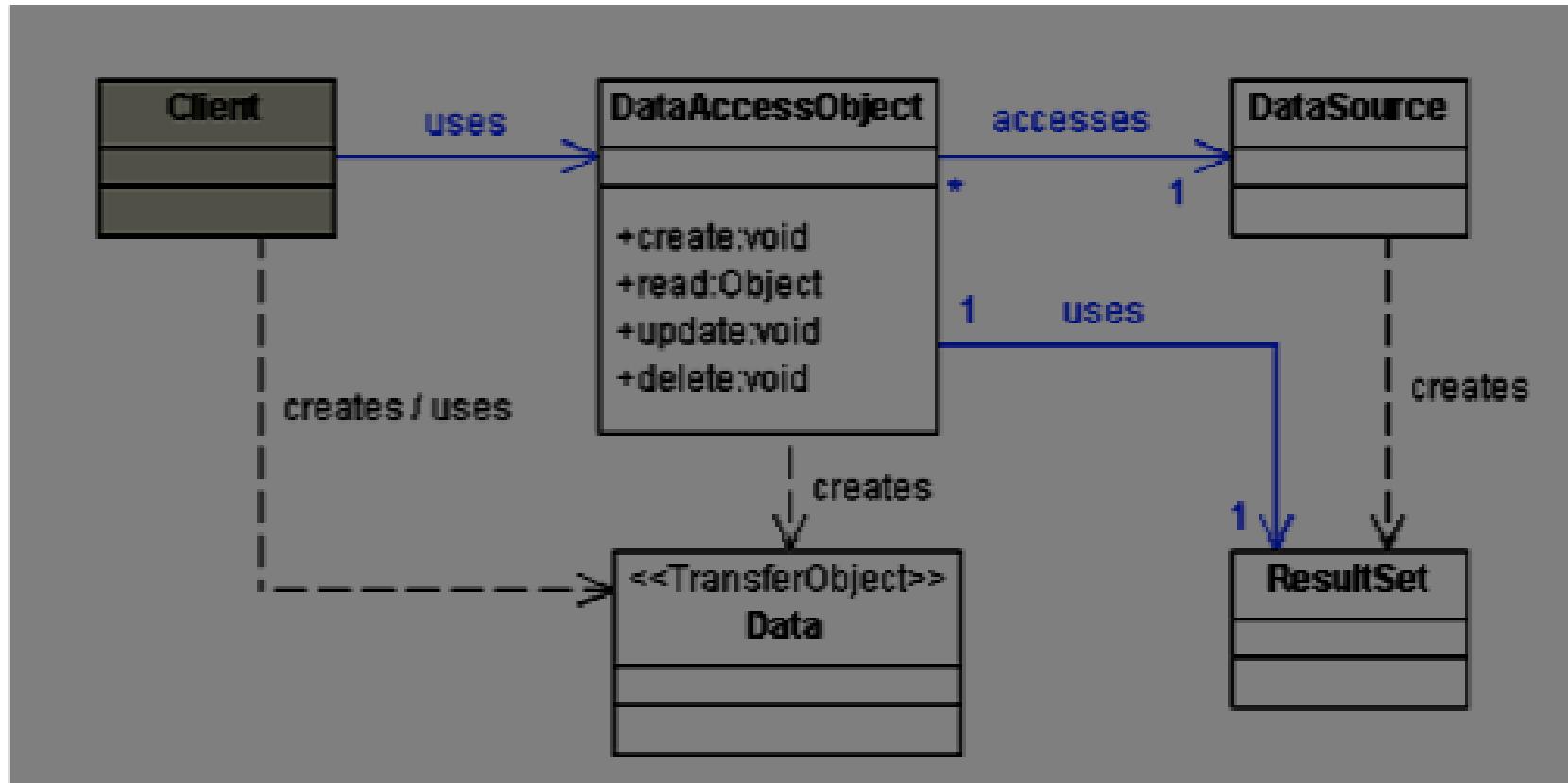
Design Patterns

Pattern DAO (Data Access Object)

- Extraction du code d'accès aux données
- Dans la logique de l'architecture multi-tiers, la couche permettant l'accès aux données doit être découplée des autres couches. La conséquence est un faible couplage, améliore la réutilisabilité et la maintenabilité.
- La solution consiste à définir un objet pour l'accès aux données (Data Access Object).
- Le pattern *Data Access Object* permet d'abstraire et d'encapsuler tous les accès du système de stockage. Il permet de gérer les connections avec toutes sortes de sources de données afin d'obtenir les données.

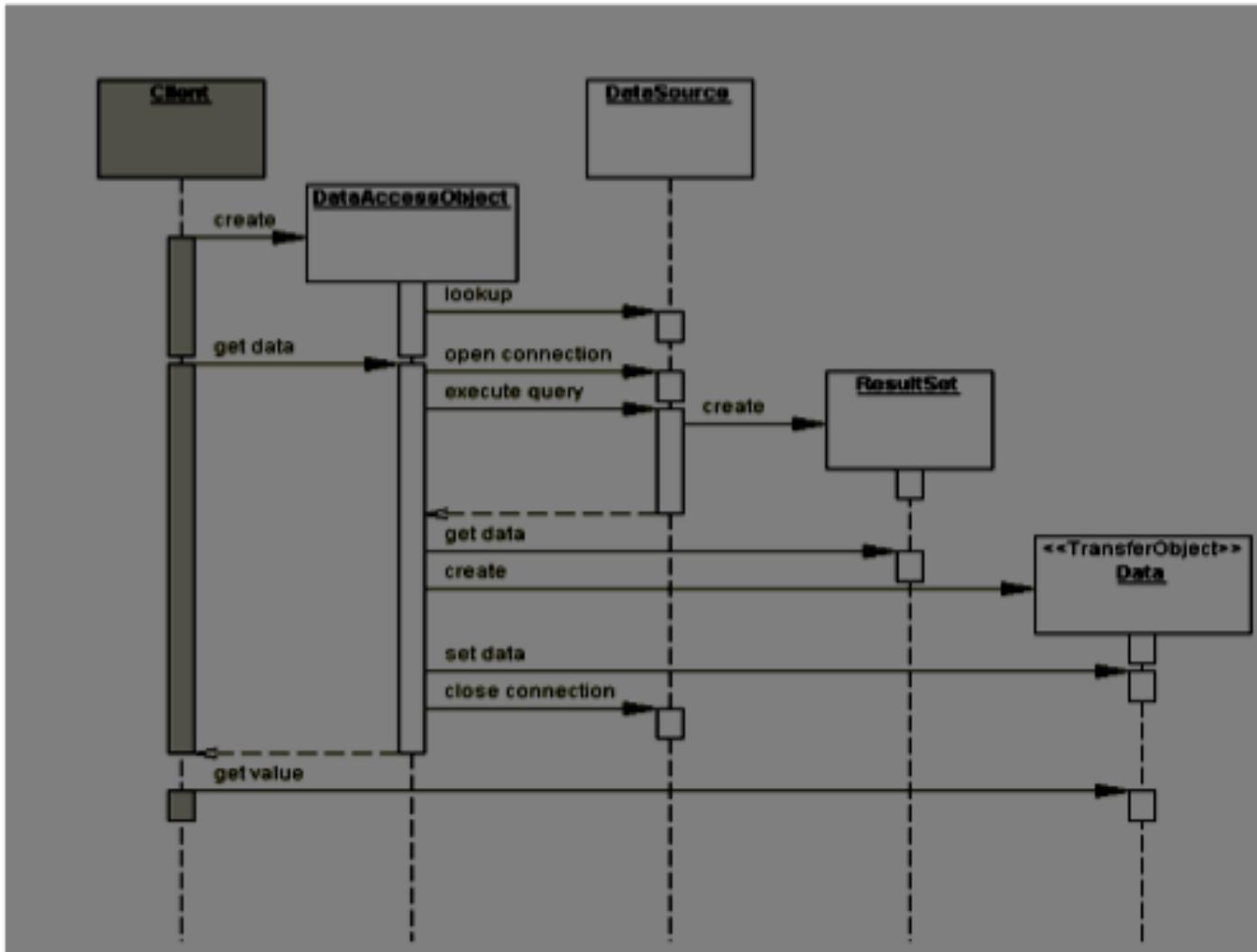
Design Patterns

Pattern DAO (Data Access Object)



Design Patterns

Pattern DAO (Data Access Object)



Design Patterns

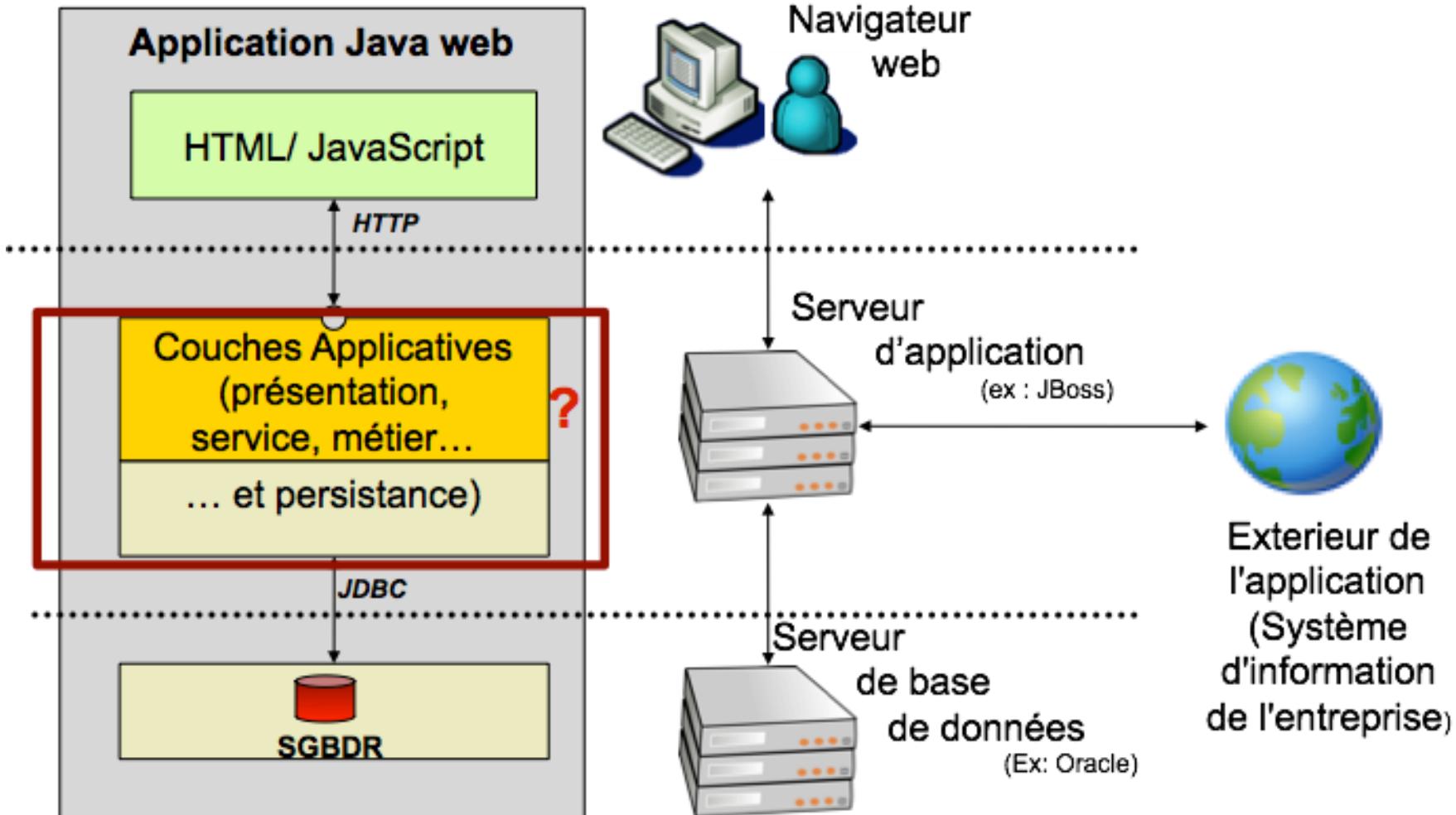
Pattern DAO (Data Access Object)

- Conséquences
 - Donne plus de transparence pour accéder aux données
 - Donne une vision orientée objet pour l'accès aux données
 - Favorise la migration des données
 - Réduit la complexité du code côté client
 - Met en évidence une couche intégration
 - Utilisation de fabriques et fabriques abstraites
- Patterns relatifs
 - Transfer Object
 - Factory et Abstract Factory [GoF]
 - Transfer Object Assembler
 - Value List Handler

Patterns d'architecture

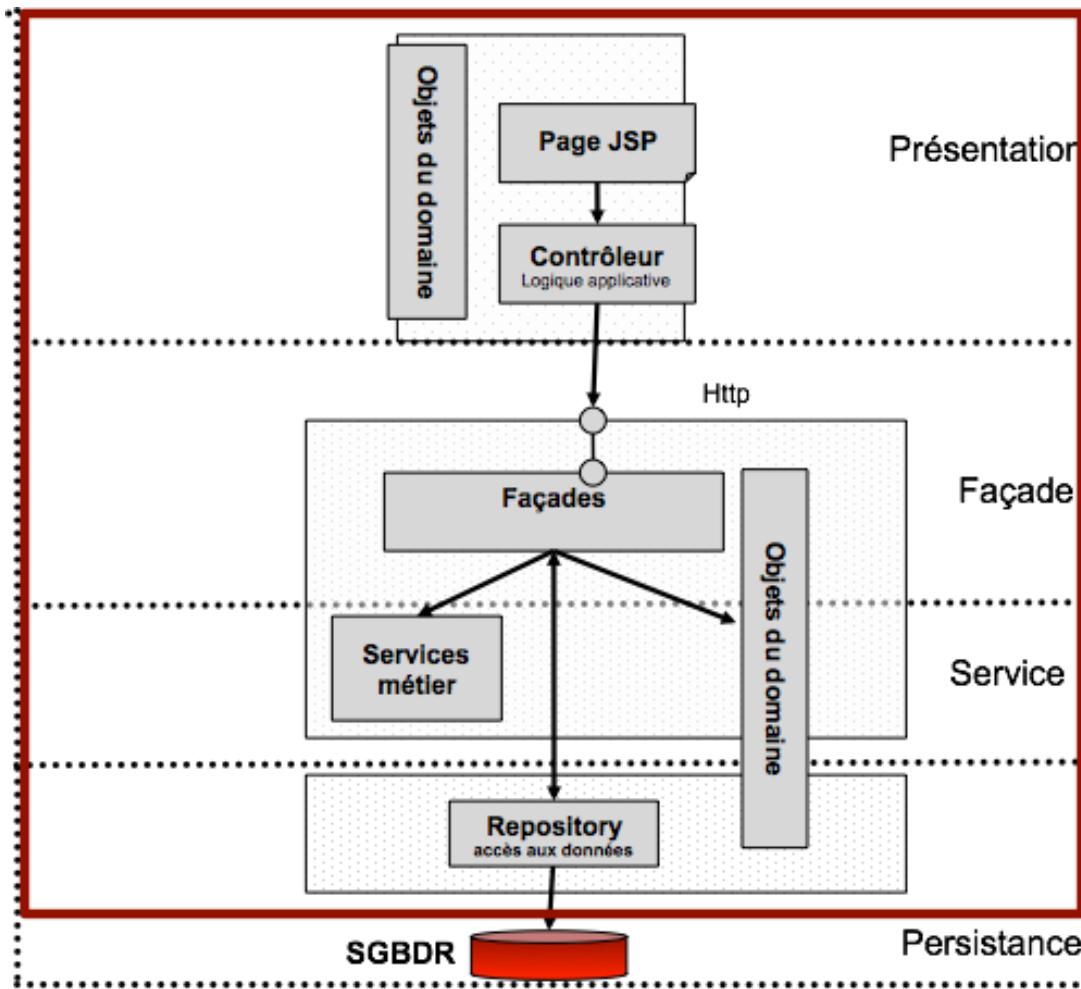
Patterns d'architecture

Exemples d'architecture applicative



Patterns d'architecture

Des couches logicielles



Introduction

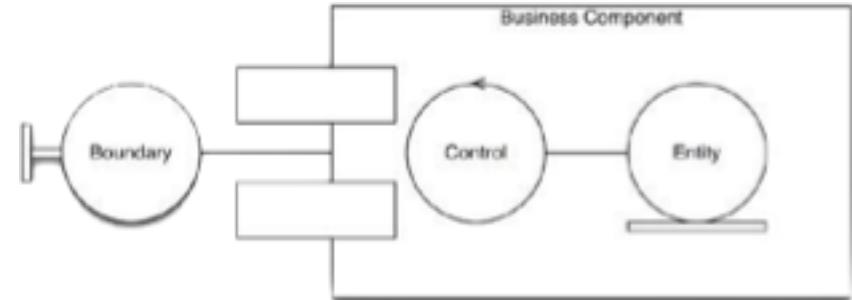
Patterns d'architecture

- 2 grandes familles
 - Piloté par le Service (Service Oriented Architecture)
 - Piloté par le Domaine (Domain Driven Design)
- Les 2 s'appuient sur le pattern ECB
 - Entity-Control-Boundary
 - similaire au pattern MVC
 - ne se limite pas qu'à la couche présentation

Introduction

Patterns d'architecture

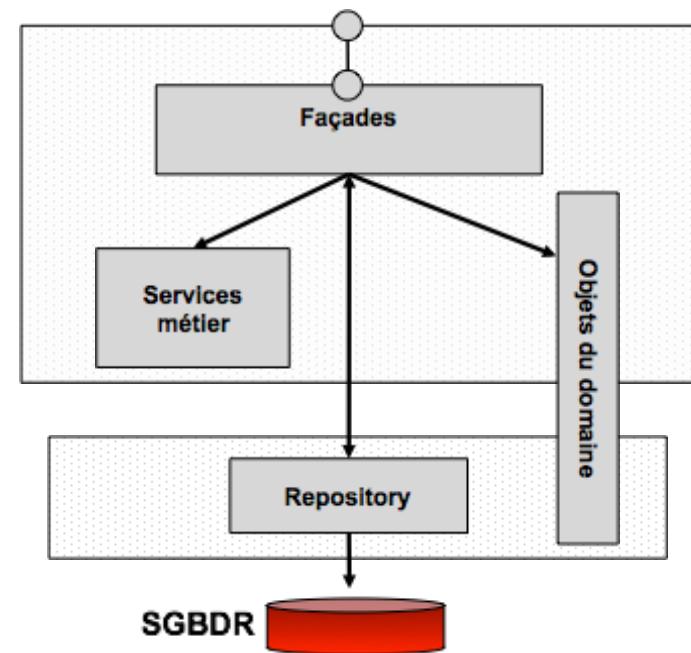
- **Entity :**
 - Appelé également «objet du domaine »
 - Élément persistant
 - Eventuellement logique métier
- **Boundary :**
 - Élément en périphérie du système
 - S'occupe de la communication avec l'extérieur
- **Control :**
 - Élément orchestrateur qui implémente un scénario (fonctionnel) donné
 - Correspond notamment aux services métier



Introduction

Patterns d'architecture : Lean Service Oriented Architecture

- Le Control (Services Métiers / Repository) est le composant le plus important
 - Les objets du domaine sont anémiques (Anemic Object Model)
 - Aucune logique métier
 - Le reflet de la base de données (POJO)



Introduction

Patterns d'architecture : Domain Driven Architecture

- Les entités du domaine sont le socle de l'application
 - Gèrent leur état
 - Et la persistance de leur état
 - Et implémentent la logique métier
 - → PDO (Persistent Domain Objet)
- Les services (Control) perdent la logique de l'application

