

Secteur Tertiaire Informatique  
Filière « Etude et développement »

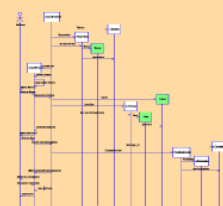
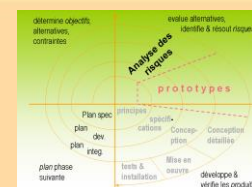
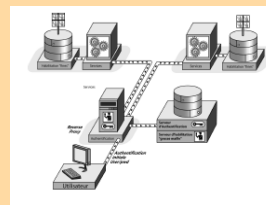
Séquence « Concevoir une application »

Concevoir une application N Tiers sécurisée

Apprentissage

Mise en situation

Evaluation



Version	Date	Auteur(s)	Action(s)
1.0	26/10/16	Lécu Régis	Création du document

# TABLE DES MATIERES

Table des matières .....	2
1. Introduction .....	5
2. Les principes de la conception sécurisée en N Tiers.....	6
2.1 Minimiser le périmètre de l'application et privilégier la simplicité .....	6
2.1.1 Spécialiser les couches .....	6
2.1.2 Limiter le couplage.....	7
2.1.3 Utiliser l'encapsulation dans chaque couche.....	8
2.2 Adopter une posture de méfiance .....	9
2.2.1 Considérer toute donnée externe comme potentiellement toxique.....	9
2.2.2 Sécuriser systématiquement les entrées/sorties .....	9
2.3 Appliquer le principe de défense en profondeur ( <i>defense in depth</i> ).....	9
2.4 Séparer et minimiser les droits .....	10
2.5 Journaliser .....	10
2.6 Utiliser des mécanismes de sécurité existants .....	10
3. Les patterns de conception et la sécurité .....	11
3.1 Quid de la mise en pratique ?.....	11
3.1.1 Application directe dans une technologie donnée .....	11
3.1.2 Utilisation de patterns de sécurité .....	11
3.1.3 Analyse de sécurité des Design Patterns existants.....	11
3.2 Analyse de sécurité des Design Patterns JEE.....	12
3.2.1 Couche présentation : Front Controller .....	12
3.2.2 Couche présentation : Application Controller .....	13
3.2.3 Couche métier : Session Façade.....	15
3.2.4 Couche métier : Application Service .....	16
3.2.5 Couche métier : Business Object.....	16
3.2.6 Couche métier : Transfer Object .....	17
3.2.7 Couche métier : Transfer Object Assembler .....	18
3.2.8 Couche intégration : Data Access Object.....	18

## Objectifs

A l'issue de cette séance, le stagiaire sera capable de :

- Mettre en œuvre les principes de sécurité généraux dans sa conception :
  - minimiser le périmètre de l'application et privilégier la simplicité,
  - séparer et minimiser les permissions et privilèges,
  - utiliser des mécanismes de sécurité existants et robustes,
  - adopter une posture de méfiance,
  - journaliser,
  - appliquer le principe de défense en profondeur.
- Appliquer les patrons de sécurité pour définir une architecture N Tiers sécurisée :
  - bien séparer les couches et définir leurs relations de confiance,
  - le client ne doit pas contenir d'éléments critiques,
  - la sécurité doit être vérifiée côté serveur.

## Pré requis

Cette séance suppose connus les principes généraux de conception sécurisée, la notion d'architecture N Tiers et les patterns objet courants.

## Méthodologie

Ce document peut être utilisé en présentiel ou à distance.

Il précise la situation professionnelle visée par la séance, la situe dans la formation, et guide le stagiaire dans son apprentissage et ses recherches complémentaires.

## Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

## Ressources

- Sur l'analyse de sécurité des patterns de conception :  
*Security\_Analysis\_of\_Core\_JEE\_Design\_Patterns.pdf* (OWASP, en anglais)  
*Security\_Analysis\_of\_Core\_JEE\_Design\_PatternsFR.doc* (traduction partielle).
- Sur différents patterns de sécurité : dossier [doc patterns](#)
- Sur les principes généraux de la conception sécurisée :  
*Secure-Programs-HOWTO* », chap. 7 *Design Your Programs for Security*  
*Secure-Software-4.Design.ppt*

# 1. INTRODUCTION

Cette séance fait partie du projet **CyberEdu**, qui prend en compte la sécurité dans tout le cycle de vie du logiciel, et dans toutes les couches des applications.

La situation professionnelle visée concerne la phase de conception logicielle.

En s'appuyant sur les exigences de sécurité identifiées dans l'analyse, le concepteur adoptera un point de vue sécurité sur la définition des couches de l'application et sur leur communication, en appliquant les principes généraux de sécurité informatique et les patterns de conception N tiers.

Dans cette situation de travail, le concepteur peut être seul ou dans une équipe réduite de conception.

Cette séance reprend les principes de la conception sécurisée dans le cadre d'une architecture N Tiers.

Puis elle présente les patterns de sécurité applicables à la conception N Tiers, en montrant :

- qu'ils respectent les principes généraux de sécurité ;
- comment ils s'appliquent aux différentes couches de l'architecture N Tiers.

## 2. LES PRINCIPES DE LA CONCEPTION SECURISEE EN N TIERS

Nous allons voir comment les principes de la conception sécurisée s'appliquent à l'architecture N Tiers.

### 2.1 MINIMISER LE PERIMETRE DE L'APPLICATION ET PRIVILEGIER LA SIMPLICITE

« *Keep Code small and simple* »

Plus le code est petit et simple, plus il est facile de vérifier la cohérence fonctionnelle et la sécurité de l'application.

Ce principe s'applique à toutes les couches : interface utilisateur, couches de liaison entre client et serveur (*middleware*), objets métier, couches de persistance et procédures stockées.

Chaque couche est définie par une interface, et toute fonctionnalité superflue dans une interface augmente la surface d'attaque.

C'est d'autant grave s'il s'agit d'une fonctionnalité oubliée dans une couche interne, qui n'est pas accessible de l'interface utilisateur : par exemple une procédure stockée, un EJB session, un Web Service utilisés pendant la phase de test ou dans une itération précédente, qui ont été oubliés dans le code final.

Un utilisateur malveillant trouvera peut-être cette fonctionnalité et y accédera par une interface utilisateur d'attaque.

Il est donc essentiel dans une approche sécurité et particulièrement dans une architecture N Tiers, de :

- faire simple dès la conception
- remanier le code à chaque itération et supprimer les fonctionnalités périmées ou inutiles
- lever une exception « *non implémentée* » sur les fonctionnalités non implémentées, définies dans une interface en prévision des itérations suivantes
- faire des vérifications croisées entre les couches pour détecter les incohérences : éléments de l'interface utilisateur qui appellent des fonctionnalités non implémentées ou supprimées ; fonctionnalité muette oubliée dans une couche interne, et inaccessible depuis l'interface utilisateur fournie.

#### 2.1.1 Spécialiser les couches

Chaque interface, classe ou méthode n'a qu'un objectif limité, qu'elle réalise complètement.

Cette exigence de **forte cohérence** contribue à la qualité de chaque couche et augmente la résistance aux attaques, en limitant les risques d'escalade : un composant sera peut-être compromis mais l'attaque sera circonscrite à une fonctionnalité précise.

Appliquée à une architecture en couches, l'exigence de forte cohérence implique la spécialisation des couches :

##### a) L'interface utilisateur (client lourd, Web ou nomade)

- elle ne contient pas de logique métier ;
- elle gère uniquement la logique de la couche présentation : dialogue avec l'utilisateur, validation des entrées, affichage des erreurs ;
- elle ne donne pas d'indications sur les couches internes.

Concevoir une application N Tiers sécurisée

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Elle utilise un langage fonctionnel et non technique : message d'erreur « *utilisateur inexistant* » et non

« *error jdbc 2.1, sql-server2014 : unknown user ; database=publi ; server=serveur-sql3* ».

#### b) Les couches de liaison (*middleware*) et couches techniques

- elles doivent être spécialisées et donner le moins de visibilité possible sur les autres couches ;
- par exemple, la couche de persistance du serveur applicatif communiquera avec la base de données par des entités, avec des annotations techniques Java JEE ;
- mais la couche de liaison entre le client et le serveur applicatif n'utilisera pas directement des entités, mais des objets POJO<sup>1</sup> extraits des entités, sans annotations techniques (pattern **DTO** : *Data Transfer Object*).

#### c) La couche métier

- elle ne réalise que les traitements métier, et doit être indépendante de l'interface utilisateur et des couches techniques ;
- elle ne renvoie que des erreurs fonctionnelles, pour ne rien divulguer sur son implémentation ;
- les différents traitements peuvent être réservés à certains utilisateurs ou groupes d'utilisateurs, mais la couche métier ne gère pas elle-même le mécanisme d'authentification, qui doit être pris en charge par une couche technique en amont ;
- à l'intérieur de la couche métier, on distinguera les processus métier qui sont présents dans l'interface publique, et les objets métiers qui sont utilisés par ces processus ;
- par exemple, pour Météo France, les processus métier décrits dans l'interface pourraient être : le temps du lendemain, la tendance à 12 jours, les alertes etc.
- les objets métier contiennent les différents modèles de prévision, qui sont appelés par les processus métier, mais ne doivent surtout pas être vus de l'utilisateur final ou d'un attaquant ;
- seuls les traitements métier qui ont un sens pour l'utilisateur final seront donc exposés dans l'interface de la couche métier.

### 2.1.2 Limiter le couplage

Le **faible couplage** facilite l'évolutivité de l'application, en limitant les conséquences de la modification d'un composant sur les autres composants.

Mais il réduit aussi les risques d'escalade, en limitant les points d'accès et les droits du composant compromis vers les autres composants.

Dans une architecture N Tiers, la limitation du couplage et la non divulgation d'informations sur les autres couches conduit à compartimenter les couches par des intermédiaires : patterns **Contrôleur** ou **Façade**.

---

<sup>1</sup> POJO : *Plain Old Java Object* : « bon vieil objet Java »



Si l'interface utilisateur appelle directement tous les processus métiers (par exemple, des EJB Session) :

- cela crée un couplage fort entre l'interface utilisateur et chaque processus métier ;
- l'interface utilisateur est contrainte d'implémenter un peu de logique métier, pour gérer les enchaînements ;
- le couplage fort avec les objets métiers et l'implémentation locale d'une logique fonctionnelle vont nuire à l'évolution de l'interface utilisateur et à son portage dans une autre technologie : par exemple du client lourd au client léger Web. Le développeur d'interface utilisateur n'étant pas forcément familiarisé avec tous les concepts du domaine applicatif, il aura du mal à manipuler les objets métier, pour l'évolution ou le portage ;
- en décompilant le code de l'interface utilisateur, un attaquant récupérera de nombreuses informations métier, qu'il pourra utiliser pour construire une interface parallèle d'attaque, vers les processus métier distants.

Les GRASP<sup>2</sup> patterns **Contrôleur** et **Façade** permettent de limiter ce couplage et de faciliter la réutilisation :

- le **Contrôleur** rassemble tous les processus métier qui doivent être publiés pour l'utilisateur final, dans une interface unique ; l'interface utilisateur n'a donc plus à connaître le détail des classes de la couche métier, et utilisera le Contrôleur comme une bibliothèque de fonctions, bien documentée ;
- la **Façade** joue un rôle semblable de regroupement, mais dans la couche métier : elle rassemble dans une même interface des traitements corrélés, par exemple qui appartiennent à un même Use Case.

Ces patterns fondamentaux peuvent être utilisés à plusieurs niveaux, sous différentes appellations, selon les couches et les technologies. En Java EE, ils sont implémentés dans les patterns que nous détaillerons dans le chapitre suivant : *Session Façade*, *Application Service*, *Front Controller*, *Application Controller*.

### 2.1.3 Utiliser l'encapsulation dans chaque couche

Ce mécanisme contribue à la qualité du logiciel et à sa sécurité :

- le mécanisme d'abstraction facilite le travail du développeur, en lui permettant de manipuler des modèles simples ;
- il contribue à la défense du composant logiciel, en cachant ses mécanismes internes et leurs vulnérabilités.

Dès la conception, il faut distinguer dans chaque couche les fonctionnalités à publier dans l'interface, de l'implémentation qui doit être cachée aux autres couches.

---

<sup>2</sup> *General Responsibility Assignment Software Patterns* : patterns généraux d'affectation des responsabilités, qu'il faut appliquer dans une bonne conception.

## 2.2 ADOPTER UNE POSTURE DE MEFIANCE

### 2.2.1 Considérer toute donnée externe comme potentiellement toxique.

Toutes les couches sont exposées à des attaques potentielles, d'un attaquant externe ou d'un utilisateur interne malveillant :

- l'interface utilisateur est particulièrement exposée, puisqu'elle est en contact direct avec l'utilisateur, normal ou malveillant. Comme les remparts extérieurs dans un château-fort, c'est le premier niveau de retranchement, qui doit filtrer ses entrées autant que possible ;
- mais les autres couches côté serveur (web, applicatif, SGBD), sont également exposées aux attaques car :
  - elles peuvent recevoir des entrées erronées ou des attaques qui auraient échappé aux filtres de l'interface utilisateur ;
  - mais aussi des données falsifiées, envoyées directement par un utilisateur malveillant qui aurait réussi à court-circuiter l'interface utilisateur normale de l'application, par une interface d'attaque.

### 2.2.2 Sécuriser systématiquement les entrées/sorties

Il faut donc sécuriser toutes les interfaces en validant les types et la sémantique de toutes les entrées. En retour, il faut informer l'appelant du résultat du traitement, par un code retour ou une exception.

Dans une architecture en couche, la validation des entrées est elle-même spécialisée et progressive, en allant de l'interface utilisateur aux objets métier :

- l'interface utilisateur peut faire des vérifications de base, sur le typage et la plage de valeurs : par exemple, un entier positif inférieur à 100 ;
- l'objet métier a une base de connaissances qui lui permet des vérifications plus poussées sur la sémantique de la donnée : par exemple, cet entier est la pression maximale d'un réservoir, et elle ne doit pas dépasser 20 Bar.

## 2.3 APPLIQUER LE PRINCIPE DE DEFENSE EN PROFONDEUR (DEFENSE IN DEPTH)

Les objets métier constituent le cœur de l'application et son dernier retranchement (le « donjon »). Une défense périmétrique (interface utilisateur) pouvant être mise en défaut, il est impératif que les objets métier ne lui fassent pas confiance et vérifient à nouveau les données en entrée.

De façon générale, aucune couche ne doit prêter une confiance totale à la couche précédente. Même si une couche n'est visible que sur le serveur lui-même ou sur le réseau local de l'entreprise, elle n'est pas à l'abri d'une attaque par un utilisateur interne malveillant.

Rappelons qu'avec l'utilisation souvent permise en entreprise des ordinateurs portables et *smartphone* personnels (**BYOD**, « *bring your own device* »), une attaque peut être conduite dans le périmètre de l'entreprise, par un logiciel qui a infecté l'appareil à l'extérieur de l'entreprise.

Cette porosité entre l'entreprise et le monde extérieur rend une défense périmétrique insuffisante et exige de défendre chaque couche de l'application.

## 2.4 SEPARER ET MINIMISER LES DROITS

Pour prévenir autant que possible les attaques et éviter les escalades en cas d'attaque, il faut authentifier ses utilisateurs et leur attribuer le minimum de droits sur l'application.

Un utilisateur non authentifié n'a aucun droit : il faut interdire par défaut plutôt qu'autoriser.

Un utilisateur authentifié n'a accès qu'à ce qu'on lui autorise explicitement : il faut utiliser des listes blanches plutôt que noires.

Toutes les activités des différentes couches doivent être associées à des permissions et réservées aux utilisateurs authentifiés et autorisés :

- fenêtre dans un client lourd ;
- page web ;
- traitements d'un objet métier, ou seulement certains traitements.

L'authentification doit se faire dans la couche externe (interface utilisateur) et l'identité de l'utilisateur doit transiter de bout en bout, à travers les différentes couches.

Dans des applications à haut niveau de sécurité, les couches intermédiaires peuvent à nouveau valider l'identité de l'utilisateur, en se basant sur d'autres critères.

## 2.5 JOURNALISER

Toutes les couches serveur (web, objets métier, base de données) doivent journaliser leurs actions, afin de pouvoir mesurer l'étendue et les conséquences d'une attaque. Mais pas n'importe comment ni en stockant n'importe quoi.

La journalisation ne devra pas comporter :

- d'informations techniques qui permettraient à un attaquant de rassembler des connaissances sur le contexte et l'implémentation de l'application, pour préparer son attaque ;
- d'informations personnelles sur les utilisateurs, contraires à la réglementation sur la vie privée (**CNIL**) ;
- des informations non filtrées qui pourraient abriter des attaques, prenant effet au réaffichage du journal : en particulier **XSS** (*Cross Site Scripting*).

## 2.6 UTILISER DES MECANISMES DE SECURITE EXISTANTS

Une bonne conception d'architecture N Tiers n'est déjà pas simple.

Elle doit être pensée d'emblée en intégrant les objectifs de sécurité, ou il n'y aura JAMAIS de sécurité. Il est pratiquement impossible de bricoler des mécanismes de sécurité après coup, à partir d'une mauvaise conception initiale.

Vues les difficultés, il ne faut donc pas improviser et s'appuyer sur des mécanismes de sécurité existants (par exemple ceux de Java EE).

## 3. LES PATTERNS DE CONCEPTION ET LA SECURITE

### 3.1 QUID DE LA MISE EN PRATIQUE ?

Comment mettre en pratique ces principes dans la conception d'une application ?

#### 3.1.1 Application directe dans une technologie donnée

Ces principes sont assez clairs : il suffit de repérer les moyens techniques existants dans chaque technologie, pour les implémenter.

Par exemple, dans les séances suivantes sur **CyberEdu**, nous verrons comment gérer les autorisations dans les EJB, comment authentifier et gérer les autorisations avec l'API Java **JAAS**, comment sécuriser le *middleware* avec **SSL/TSL**.

Cette méthode convient pour une application donnée, ou pour une entreprise spécialisée dans une technologie donnée, mais elle pose un problème de transfert de connaissance : est-on sûr de trouver en Microsoft C#, l'équivalent de JAAS etc. ?

Les patterns fournissent un niveau d'abstraction intermédiaire entre les principes généraux et les détails d'implémentation. Ils fournissent un langage commun aux développeurs soucieux d'intégrer la sécurité dans leur conception.

#### 3.1.2 Utilisation de patterns de sécurité

Ces patterns ont été développés spécifiquement pour la sécurité, parallèlement aux **Design Patterns** usuels, qui sont ou devraient être familiers pour les développeurs objets.

Il en existe plusieurs bibliothèques.

Ce sont de bons outils de conception, qu'il faut obligatoirement maîtriser en tant que concepteur spécialisé dans la sécurité informatique.

**Inconvénient** : pour un concepteur-développeur soucieux d'intégrer la sécurité dans son projet, sans se spécialiser dans le domaine, cela fait double emploi avec les patterns de conception. Il y a même parfois des doublons, avec des noms différents.



A parcourir pour se faire une idée :

Dans le dossier [docs patterns](#), vous trouverez plusieurs ouvrages en anglais, sur les patterns de sécurité. Le plus connu : [Core-Security-Patterns.pdf](#) (*abstract* et *poster* du livre).

#### 3.1.3 Analyse de sécurité des Design Patterns existants

C'est également une démarche courante, qui a été choisie par l'OWASP.

C'est celle que nous allons suivre, car c'est la plus adaptée pour des développeurs objet qui s'initient à la conception sécurisée.

La méthode consiste à relire les Design Patterns (ou ici leurs variantes Java JEE) et faire un commentaire de sécurité, en respectant les principes généraux de la conception sécurisée.

**Pour chaque pattern, cela conduit à préciser ce qu'il faut :**

⇒ **Eviter :**

Mauvaises pratiques que l'on doit exclure dans la version sécurisée du Pattern ;

⇒ **Implémenter :**

La fonction du Pattern et sa situation dans les couches en font l'emplacement idéal pour implémenter certaines fonctions de sécurité : authentification, validation des entrées, journalisation, interface etc.

### 3.2 ANALYSE DE SECURITE DES DESIGN PATTERNS JEE



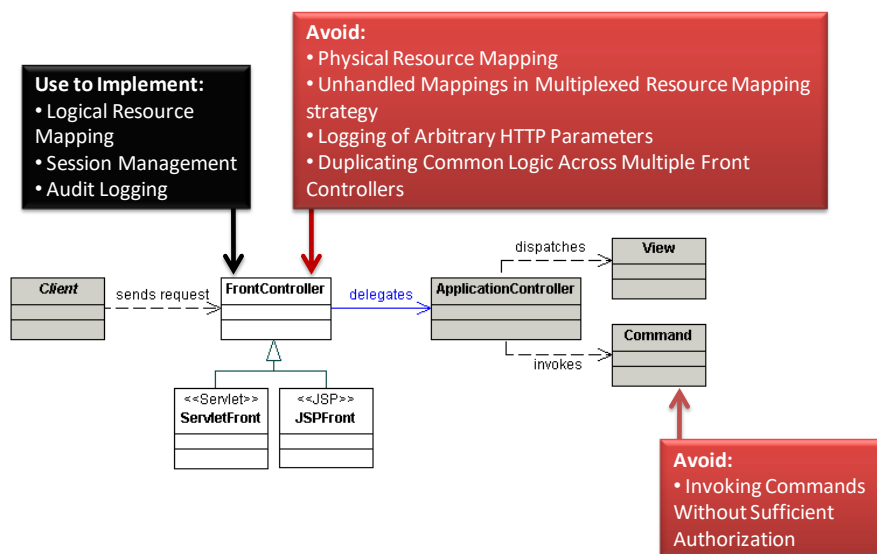
Document : [Security\\_Analysis\\_of\\_Core\\_JEE\\_Design\\_PatternsFR](#)

Dans ce document OWASP (libre de droit), nous avons traduit les patterns les plus courants (en rouge dans la table des matières).

Vous allez les lire en recherchant pour chaque contrainte de sécurité (**Eviter** ou **Implémenter**), les principes généraux de sécurité qui les justifient. Lire également l'introduction et la présentation de la couche métier (p. 3 et p. 25).

Nous allons donner quelques repères sur chacun d'entre eux, en les classant par couches.

#### 3.2.1 Couche présentation : Front Controller



C'est un contrôleur web qui est une variante du GRASP Contrôleur : il centralise toutes les requêtes d'un site, et délègue l'action demandée à un *Application Controller*.

Rattachons chaque contrainte de sécurité à son principe général.

**A éviter :**

- Mapper des ressources physiques

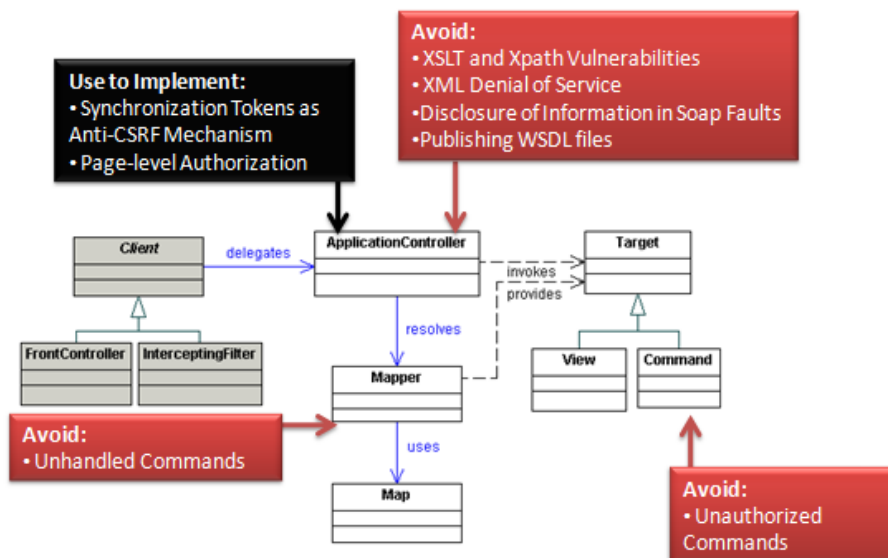
⇒ **encapsulation** : on ne doit voir que des ressources logiques qui sont une abstraction du stockage et non le stockage lui-même.

- Lancer des commandes sans autorisation suffisante
  - ⇒ **séparer et minimiser les droits**
- *Unhandled Mappings in Multiplexed Resource Mapping strategy*
  - ⇒ **adopter une posture de méfiance** : validation complète des entrées pour détecter un mapping inexistant
  - ⇒ **encapsulation** : on ne renvoie que des messages génériques pour ne pas révéler des détails d'implémentation
- Journaliser des paramètres HTTP arbitraires
  - ⇒ **Journaliser** : pas d'informations d'implémentation ni d'informations personnelles.

#### A utiliser pour implémenter :

- Mapper des ressources logiques
  - ⇒ **encapsulation** : le mapping logique est une abstraction qui interdit de voir les répertoires physiques et les protège contre les attaques
- Gestion de session
  - ⇒ **séparer et minimiser les droits** : permet d'authentifier les utilisateurs, et de gérer leurs autorisations.
- Journalisation de l'audit
  - ⇒ **journalisation** : on ne mémorise que le nom de l'utilisateur authentifié etc.

### 3.2.2 Couche présentation : Application Controller



C'est la partie Contrôleur du Pattern MVC.

Concevoir une application N Tiers sécurisée

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

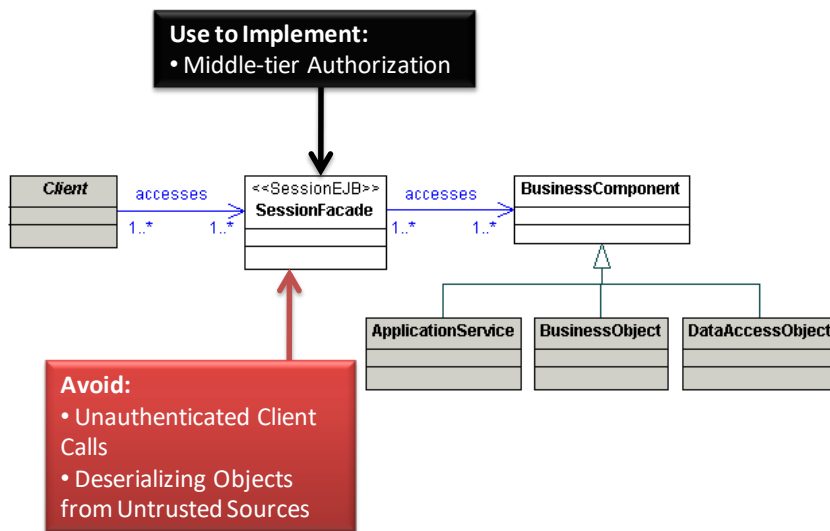
### A éviter :

- Commandes inexistantes :
  - ⇒ **adopter une posture de méfiance** : validation des entrées pour détecter les commandes inexistantes
  - ⇒ **encapsulation** : on ne renvoie que des messages génériques pour ne pas révéler des détails d'implémentation et éviter les attaques XSS
- Vulnérabilités XSLT et XPath
  - ⇒ **adopter une posture de méfiance** : validation des entrées par une liste blanche pour détecter une attaque XSLT et XPath ; encodage des données utilisateur dans la génération des vues.
- Dénier de service XML
  - ⇒ **adopter une posture de méfiance** : validation des entrées (contenu et taille des messages XML)
- Divulgaration d'information dans les erreurs SOAP
  - ⇒ **encapsulation** : on ne renvoie que des messages d'erreur génériques pour ne pas révéler des détails d'implémentation.
- Publier des fichiers WSDL
  - ⇒ **encapsulation** : on ne publie que les interfaces et des informations fonctionnelles ; le fichier WSDL est un détail d'implémentation.

### A utiliser pour implémenter :

- Synchronisation du jeton comme mécanisme Anti-CSRF
  - ⇒ **séparer et minimiser les droits** : le jeton Anti-CSRF protège la session qui contient l'identité et les autorisations de l'utilisateur authentifié
- Autoriser au niveau de la page
  - ⇒ **séparer et minimiser les droits** : permet de contrôler l'accès à la page, selon l'utilisateur authentifié.
  - ⇒ **défense en profondeur** : on ne suppose pas que l'accès a été autorisé en amont ; on vérifie les autorisations d'accès à la page.

### 3.2.3 Couche métier : Session Façade



Situé entre le client (lourd ou web) et les composants métier, il implémente strictement le GRASP Contrôleur.

#### A éviter :

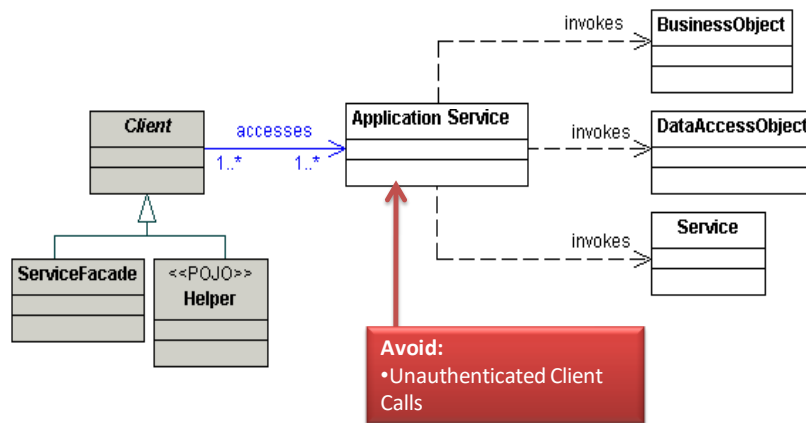
- Les appels d'un client non authentifié
  - ⇒ **séparer et minimiser les droits** : ne rien autoriser à des utilisateurs non authentifiés
  - ⇒ **défense en profondeur** : même si le client a passé la couche Web, sa requête est refusée par le *Session Façade*
- Désérialiser des objets depuis des sources non sûres
  - ⇒ **adopter une posture de méfiance** : traiter les objets sérialisés comme des entrées non sûres, qu'il faut valider
  - ⇒ **séparer et minimiser les droits** : authentifier les requêtes AVANT de désérialiser les objets transmis

#### A utiliser pour implémenter :

- *Middle Tier Authorization at Session Facade* :
  - ⇒ **séparer et minimiser les droits** : le Session Façade est le point d'entrée du serveur applicatif ; il gère l'authentification et sert de « portier » pour la couche métier.



### 3.2.4 Couche métier : Application Service



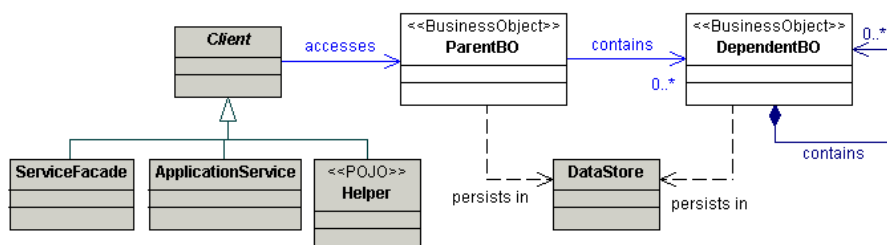
Situé dans la couche métier, il implémente strictement le GRASP Façade.

C'est le point d'entrée obligé d'un sous-système, Use Case, dont il gère les objets métier (*BusinessObject*), les accès aux données (*DataAccessObject*) et les appels à d'autres services (*Service*).

**A éviter :**

- Les appels d'un client non authentifié
- ⇒ **séparer et minimiser les droits** : ne rien autoriser à des utilisateurs non authentifiés

### 3.2.5 Couche métier : Business Object



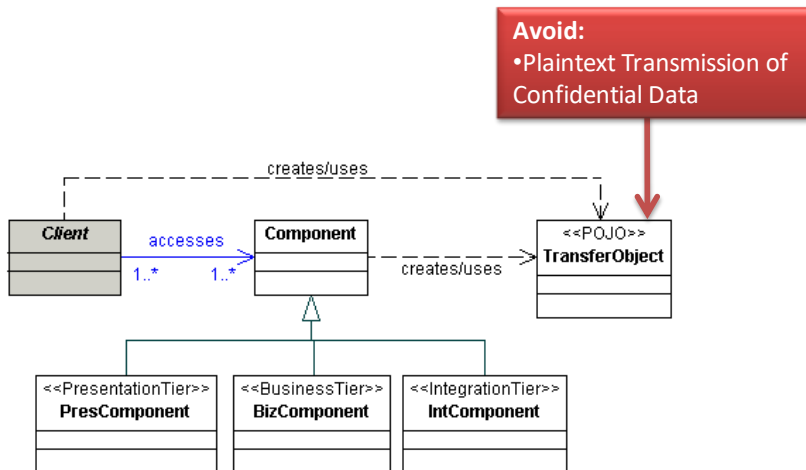
Place dans l'architecture JEE :

- le *Session Façade* est un EJB Session, avec une interface *@Remote*
  - c'est le seul point d'entrée de toute l'application, publié dans l'annuaire JNDI
  - il est responsable de l'authentification de l'utilisateur
  - ses méthodes sont annotées avec des rôles de sécurité, ce qui les réserve à certains utilisateurs ou groupes.
- chaque *Application Service* est un EJB Session avec une interface *@Local*
  - il n'est pas accessible directement depuis le réseau. Seulement par l'intermédiaire du *Session Façade*
  - il gère un Use Case, un sous-ensemble fonctionnel cohérent de l'application (un service) constitué de plusieurs objets métier, et il expose des processus métiers.

Concevoir une application N Tiers sécurisée

- Le *Business Object* est un POJO qui ne réalise pas de fonctionnalité de sécurité par lui-même :
  - car l'authentification a déjà été réalisée par le *Session Façade* et passée à l'*Application Service*, qui est le point d'entrée vers un groupe de *Business Object* (regroupés en package).
  - mais il obéit aux principes de **posture de méfiance** et de **défense en profondeur** : l'objet respecte soigneusement les règles du développement sécurisé et teste à nouveau toutes ses entrées.

### 3.2.6 Couche métier : Transfer Object



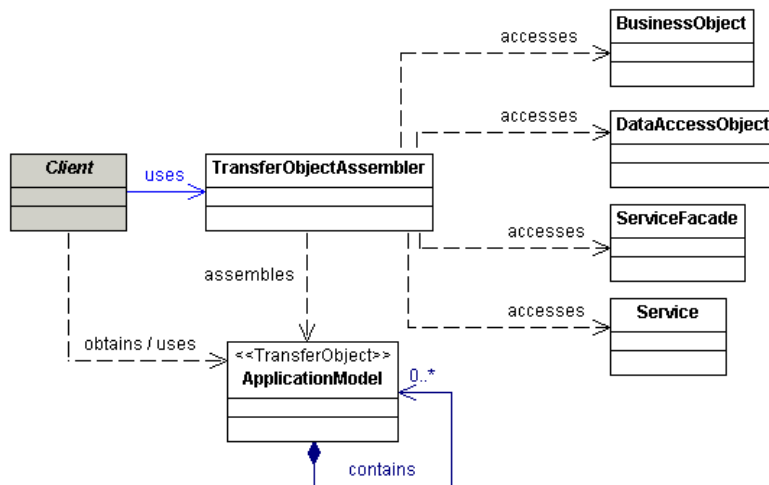
Encore appelé **DTO** (*Data Transfer Object*)

⇒ **encapsulation** : il permet une mise en forme des données serveur (par exemple, des entités) et évite de divulguer des informations confidentielles ou d'implémentation.

#### A éviter :

- La transmission en clair de données confidentielles
  - ⇒ **adopter une posture de méfiance** : sécuriser les entrées/sorties

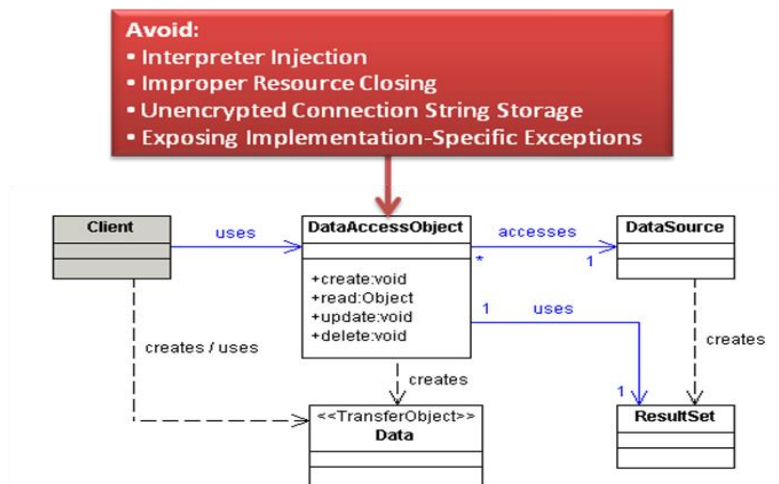
### 3.2.7 Couche métier : Transfer Object Assembler



C'est une fabrique (*factory*) qui permet d'uniformiser la construction des Transfer Object.

⇒ **simplicité du code** : évite la duplication du code.

### 3.2.8 Couche intégration : Data Access Object



**A éviter :**

- L'injection
  - ⇒ **adopter une posture de méfiance** : vis-à-vis des entrées non sûres qui ne doivent pas être passées directement aux interpréteurs, par peur des injections
- Stockage d'une chaîne de connexion non chiffrée
  - ⇒ **adopter une posture de méfiance** : ne stocker aucune information de sécurité en clair, qui puisse être détournée par un attaquant
- Exposer des exceptions dépendant de l'implémentation
  - ⇒ **encapsulation** : on ne renvoie que des messages d'erreur génériques pour ne pas révéler des détails d'implémentation.

Concevoir une application N Tiers sécurisée

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

## **CRÉDITS**

### **OEUVRE COLLECTIVE DE L'AFPA**

Sous le pilotage de la DIIP  
et du centre sectoriel Tertiaire

### **EQUIPE DE CONCEPTION**

Chantal PERRACHON – IF Neuilly-sur-Marne  
Régis Lécu – Formateur AFPA Pont de Claix