Les Framework Java

Spring

Claude Duvallet

Université du Havre
UFR Sciences et Techniques
25 rue Philippe Lebon - BP 540
76058 LE HAVRE CEDEX
Claude.Duvallet@gmail.com
http://litis.univ-lehavre.fr/~duvallet/

Spring

Présentation de Spring (1/2)

- SPRING est effectivement un conteneur dit « léger », c'est-à-dire une infrastructure similaire à un serveur d'application J2EE.
- Il prend en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets.
- Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, vos classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications J2EE et des EJBs).
- C'est en ce sens que SPRING est qualifié de conteneur « léger ».

Présentation de Spring (2/2)

- Outre cette espèce de super fabrique d'objets, SPRING propose tout un ensemble d'abstractions permettant de gérer entre autres :
  - Le mode transactionnel.
  - L'appel d'EJBs.
  - La création d'EJBs.
  - La persistance d'objets
  - La création d'une interface Web.
  - L'appel et la création de WebServices.
- Pour réaliser tout ceci, SPRING s'appuie sur les principes du design pattern IoC et sur la programmation par aspects (AOP).
- Spring est disponible sous licence Apache 2.0.

Les services fournis par Spring (1/2)

Spring propose les services suivants (liste non-exhaustive) :

- Découplage des composants. Moins d'interdépendances entre les différents modules.
- Rendre plus aisés les tests des applications complexes c'est-à-dire des applications multicouches.
- Diminuer la quantité de code par l'intégration de frameworks tiers directement dans Spring.
- Permettre de mettre en oeuvre facilement la programmation orientée aspect.
- Un système de transactions au niveau métier qui permet par exemple de faire du "two-phases-commit".

Les services fournis par Spring (2/2)

- Un mécanisme de sécurité.
- Pas de dépendances dans le code à l'api Spring lors l'utilisation de l'injection. Ce qui permet de remplacer une couche sans impacter les autres.
- Une implémentation du design pattern MVC
- Un support du protocole RMI. Tant au niveau serveur qu'au niveau du client.
- Déployer et consommer des web-services très facilement.
- Echanger des objets par le protocole http.

Les modules de Spring (1/2)

Le framework est organisé en modules, reposant tous sur le module Spring Core :

- **Spring Core :** implémente notamment le concept d'inversion de contrôle (injection de dépendance). Il est également responsable de la gestion et de la configuration du conteneur.
- **Spring Context :** Ce module étend Spring Core. Il fournit une sorte de base de données d'objets, permet de charger des ressources (telles que des fichiers de configuration) ou encore la propagation d'évènements et la création de contexte comme par exemple le support de Spring dans un conteneur de Servlet.
- **Spring AOP :** Permet d'intégrer de la programmation orientée aspect.
- **Spring DAO :** Ce module permet d'abstraire les accès à la base de données, d'éliminer le code redondant et également d'abstraire les messages d'erreur spécifiques à chaque vendeur. Il fournit en outre une gestion des transactions.

Les modules de Spring (2/2)

- **Spring ORM :** Cette partie permet d'intégrer des frameworks de mapping Object/Relationnel tel que Hibernate, JDO ou iBatis avec Spring. La quantité de code économisé par ce package peut être très impressionnante (ouverture, fermeture de session, gestion des erreurs)
- **Spring Web :** Ensemble d'utilitaires pour les applications web. Par exemple une servlet qui démarre le contexte (le conteneur) au démarrage d'une application web. Permet également d'utiliser des requêtes http de type multipart. C'est aussi ici que se fait l'intégration avec le framework Struts.
- **Spring Web MVC :** Implémentation du modèle MVC. Personnellement j'utilise plutôt Struts mais c'est surtout une question d'habitude, c'est là la grande force de Spring, rien ne vous oblige à tout utiliser et vous pouvez tout mélanger. Ce qui n'est pas forcément une bonne idée mais nous en reparlerons.

Les patrons de conception

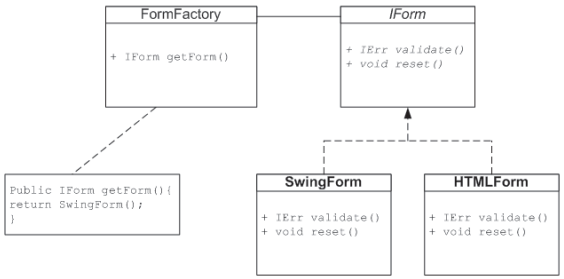
- En anglais, on parle de Design Pattern.
- Spring repose sur des concepts éprouvés, patrons de conception et paradigmes, dont les plus connus sont
  - IoC (Inversion of Control),
  - le singleton,
  - la programmation orientée Aspect,
  - ou encore le modèle de programmation dit "par template".
- Ces concepts n'étant pas propre à Spring, ils s'appliquent également à d'autres frameworks.

Le modèle de conception "fabrique" (factory) (1/3)

- C'est grâce à ce modèle que Spring peut produire des objets respectant un contrat mais indépendants de leur implémentation.
- En réalité, ce modèle est basé sur la notion d'interface et donc de contrat.
- L'idée est simplement d'avoir un point d'entrée unique qui permet de produire des instances d'objets.
- Tout comme une usine produit plusieurs types de voitures, cette usine a comme caractéristique principale de produire des voitures
- De la même façon une fabrique d'objets produira n'importe quel type d'objet pour peu qu'ils respectent le postulat de base.
- Ce postulat (le contrat) pouvant être très vague ou au contraire très précis.

Le modèle de conception "fabrique" (factory) (2/3)

- Voyons le diagramme de classe :



- On remarque facilement que le simple fait de changer la méthode `getForm()` permet de passer d'un formulaire de type swing à un formulaire de type html.
- Cela sans avoir le moindre impact sur tout le code qui en découle.

Le modèle de conception "fabrique" (factory) (3/3)

- Nous avons vu ci-dessus que Spring encourageait la programmation par contrat en voici l'une des nombreuses raisons.
- Bien entendu aussi longtemps que le programmeur chargé de réaliser la classe `SwingForm` n'aura pas terminé sa tâche, la méthode `getForm()` pourra renvoyer une instance d'une pseudo implémentation qui renvoie toujours null ou certaines erreurs remplies en dur pour pouvoir tester et développer les classes clientes.
- Naturellement à lui seul ce pattern ne suffit pas, il faudra lui adjoindre le singleton et des possibilités de configuration ainsi que le modèle "bean".
- Dès lors le pattern IoC (Inversion of control) sera réalisable.

Le singleton (1/2)

- Il s'agit certainement du patron de conception le plus connu.
- En effet, il est (très) utilisé dans beaucoup de domaines.
- Ce modèle revient à s'assurer qu'il n'y aura toujours qu'une instance d'une classe donnée qui sera instanciée dans la machine virtuelle.
- Les objectifs sont simples :
  1. Eviter le temps d'instanciation de la classe.
  2. Eviter la consommation de mémoire inutile.
- Ce modèle impose cependant une contrainte d'importance, la classe qui fournit le service ne doit pas avoir de notion de session.

Le singleton (2/2)

- C'est à dire qu'importe le module appelant, le service réagira toujours de la même façon à paramètre équivalent, il n'a pas de notion d'historique (ou de session).
- Le POJO qui implémentera le service ne doit pas stocker des informations au niveau de l'objet lui-même.
- Pour faire simple il ne faut pas modifier les variables membres au sein d'une opération (une méthode du service).
- Ce concept relativement simple à appréhender, est également simple à mettre en oeuvre. Du moins en théorie.
- L'objectif est de garantir l'unicité de l'instance, pour cela il faut interdire la création de toute instance en dehors du contrôle de la classe.
- Dans ce but, le constructeur est rendu privé et une méthode qui retourne une instance de la classe est mise à disposition.

Le diagramme de classe du Singleton (1/2)

- Ce diagramme de classe se traduit par la portion de code suivante :

```
public class Singleton {
  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }

  private Singleton() {}
  private static Singleton instance;
}
```


Le diagramme de classe du Singleton (2/2)

- En fait il manque quelque chose pour que ce soit une implémentation correcte du singleton.
- Il s'agit du problème du multithreading, en effet si plusieurs threads accèdent de façon simultanée à la méthode `getInstance()` il se peut que les deux threads détectent en même temps que le membre est null ce qui conduira à un problème.
- Pour cela, il faudrait déclarer la méthode synchronized ou tout du moins la portion de code qui manipule l'instance.
- C'est justement un des avantages de Spring, vous n'avez plus à vous soucier de cela, Spring le fait et il le fait très bien.

L'inversion de contrôle (1/5)

- L'inversion de contrôle (IoC : Inversion of Control) ou l'injection de dépendances (Dependency Injection) est sans doute le concept central de Spring.
- Il s'agit d'un "design pattern" qui a pour objectif de faciliter l'intégration de composants entre eux.
- Le concept est basé sur le fait d'inverser la façon dont sont créés les objets.
- Dans la plupart des cas, si je souhaite créer un objet qui en utilise un autre je programmerai quelque chose du type :

```
DBConnexion c=new DBConnexion("jdbc:... ") ;
Pool p=new Pool(c);
//reste du code
SecurityDAO securityDao=new SecurityDao(p);
SecurityBusiness securityBusiness=new securityBusiness(securityDao);
```


L'inversion de contrôle (2/5)

- À présent votre chef de projet, vous dit : "le client a changé d'avis, il souhaite utiliser LDAP et non sa base de données pour l'identification".
- Donc, vous devez changer en :

```
Properties p=new Properties ("ldap.properties");
LDAPConnexion c=new LDAPConnexion(p) ;
//reste du code
SecurityDAO securityDao=new SecurityDao(c);
SecurityBusiness securityBusiness=new securityBusiness(securityDao);
```

- De là, votre chef de projet revient, et vous dit que la direction a décidé d'en faire un produit standard compatible avec toutes les bases de données et systèmes d'authentification comme Active Directory.
- Bon, à présent pour vous ça devient moins drôle.
- La solution consiste à bien séparer les couches et à utiliser des interfaces pour se faire.

L'inversion de contrôle (3/5)

- Naturellement, il ne s'agit pas de quelque chose de propre à Spring, donc dans un premier temps nous aurons :

```
IConnexion c=new LDAPConnexionImpl () ;
ISecurityDAO securityDao=new LDAPSecurityDaoImpl(c);
ISecurityBusiness securityBusiness=new SecurityBusinessImpl(securityDao);
```

- Maintenant le problème qui se pose est celui de la configuration, comment configurer de manière simple les drivers et les objets nécessaires.
- Et surtout comment faire pour que le tout s'intègre proprement dans l'application.

$\Rightarrow$ Entrée en jeu de l'inversion de contrôle.

- L'objectif n'est plus de fournir un objet à un autre objet mais au contraire de faire en sorte que, l'objet dont on a besoin, sache lui-même ce dont il a besoin.
- Et le cas échéant si un des objets nécessaires a lui-même des dépendances qu'il se débrouille pour les obtenir.

L'inversion de contrôle (4/5)

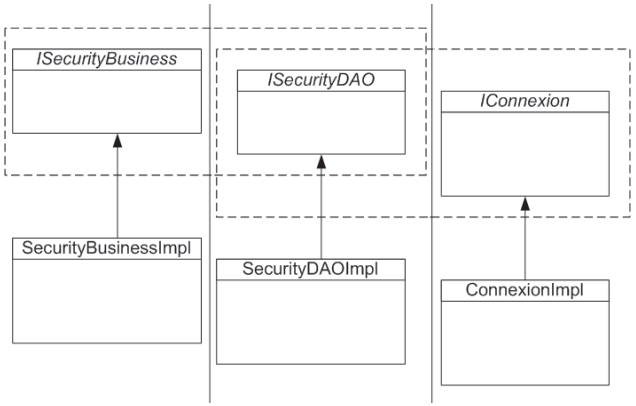
- Ce qui devrait résulter en :

```
ISecurityBusiness securityBusiness=IoCContainer.getBean("ISecurityBean);
```

- La méthode `getBean` est purement formelle pour l'exemple, l'objectif est de montrer que le code est réduit à sa forme la plus simple, les dépendances étant déclarées dans la configuration du conteneur.
- Le rôle de cette méthode :
  1. Elle résout le nom ISecurityBean dans son arbre de dépendances et trouve la classe qui l'implémente.
  2. Elle en crée une instance (cas d'une injection pas mutateurs).
  3. Elle crée les objets dépendants et les définit dans l'objet ISecurityBean grâce à ses accesseurs.
  4. Et enfin, récursivement elle fait de même pour toutes les dépendances des dépendances.

L'inversion de contrôle (5/5)

- Sur la figure suivante on remarque que les dépendances entre les couches sont réduites au minimum et que l'on peut facilement remplacer une implémentation par une autre sans mettre en danger la stabilité et l'intégrité de l'ensemble.



La notion d'injection de dépendances

- Il existe trois types d'injection :
  - **L'injection par constructeurs.**
  - **L'injection par mutateurs (setters).**
  - **L'injection d'interface.**

L'injection par constructeurs

- Ce type d'injection se fait sur le constructeur, c'est-à-dire que le constructeur dispose de paramètres pour directement initialiser tous les membres de la classe.

```
Object construitComposant(String pNom){
    Class c=rechercheLaClassQuiImplemente(pNom) ;
    String[] dep= rechercheLesDependance(pNom) ;
    Params[] parametresDeConstructeur;
    Pour chaque element (composant) de dep
    Faire
        Object o= construitComposant(composant) ;
        Rajouter o à la liste de parametresDeConstructeur ;
    Fin Faire
    construireClasse( c, parametresDeConstructeur)
}
```


L'injection par mutateurs (setters)

- Ce type d'injection se fait après une initialisation à l'aide d'un constructeur sans paramètre puis les différents champs sont initialisés grâce à des mutateurs.
- `composant.setNomMembre(o)` : le nom `setNomMembre` est trouvé grâce à la configuration du composant où il est déclaré : le membre à initialiser est `nomMembre`.

```
Object construitComposant(String pNom){
    Class c=rechercheLaClassQuiImplemente(pNom) ;
    Object composanr=new c() ;
    String[] dep= rechercheLesDependance(pNom) ;
    Params[] parametresDeConstructeur;
    Pour chaque element (composant) de dep
    Faire
        Object o= construitComposant(composant) ;
        composant.setNomMembre(o) ;
    Fin Faire
}
```

L'injection d'interface (1/2)

- Cette injection se fait sur la base d'une méthode.
- Elle est proche de l'injection par mutateurs, la différence se résume à pouvoir utiliser un autre nom de méthode que ceux du "design pattern bean".
- Pour cela, il faut utiliser une interface afin de définir le nom de la méthode qui injectera la dépendance.

```
Object construitComposant(String pNom){
    Class c=rechercheLaClassQuiImplemente(pNom) ;
    Object composanr=new c() ;
    String[] dep= rechercheLesDependance(pNom) ;
    Params[] parametresDeConstructeur;
    Pour chaque element (composant) de dep
    Faire
        Object o= construitComposant(composant) ;
        composant.méthodeInjection(o) ;
    Fin Faire
}
```


L'injection d'interface (2/2)

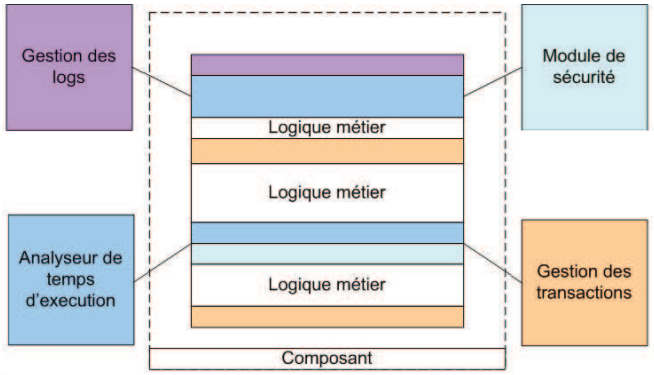
- `composant.méthodeInjection(o)` : le nom `méthodeInjection` est trouvé grâce à la configuration du composant où il est déclaré.
- La méthode qui injecte l'objet est définie par une interface.
- Dans notre exemple, l'interface serait :

```
public interface IInjectMethode{
    public void méthodeInjection(Object o);
}
```

- L'implémentation du composant se devra alors d'implémenter cette interface également.

Programmation orienté Aspect (1/4)

- Comme nous pouvons le voir dans la figure suivante, un module ou composant métier est régulièrement pollué par de multiples appels à des composants utilitaires externes.



Programmation orienté Aspect (2/4)

- De fait, ces appels rendent le code plus complexe et donc moins lisible.
- Comme chacun sait, un code plus court et donc plus clair améliore la qualité et la réutilisabilité.
- L'utilisation de composants externes implique :
  1. Enchevêtrement du code.
  2. Faible réutilisabilité.
  3. Qualité plus basse due à la complexité du code.
  4. Difficulté à faire évoluer.

Programmation orienté Aspect (3/4)

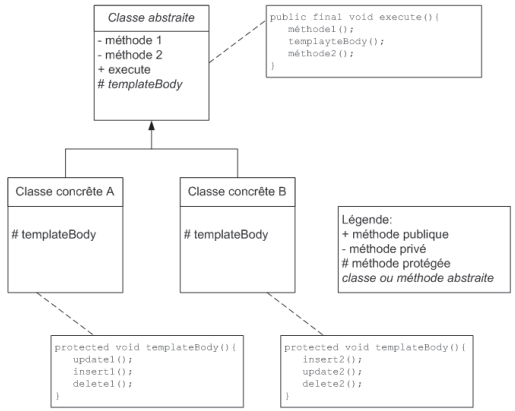
- La solution constituerait donc à externaliser tous les traitements non relatifs à la logique métier en dehors du composant.
- Pour ce faire il faut pouvoir définir des traitements de façon déclarative ou programmative sur les points clés de l'algorithme. Typiquement avant ou après une méthode.
- Dans la plupart des cas, ce genre de traitements utilitaires se fait en début ou en fin de méthode, comme par exemple journaliser les appels ou encore effectuer un commit ou un rollback sur une transaction.
- La démarche est alors la suivante :
  1. Décomposition en aspect : Séparer la partie métier de la partie utilitaire.
  2. Programmation de la partie métier : Se concentrer sur la partie variante.
  3. Recomposition des aspects : Définition des aspects.

Programmation orienté Aspect (4/4)

- Il existe deux types de programmation orientée aspect, ces deux techniques ont chacune des avantages et des inconvénients :
  - l'approche statique, c'est-à-dire que la connexion entre l'aspect et la partie métier se fait au moment de la compilation ou après dans une phase de post-production. Par exemple par manipulation du bytecode.
  - ⇒ Comme toujours cette méthode intrusive n'est pas forcément la plus transparente.
  - L'approche dynamique, dans ce cas, la connexion s'effectue par la réflexion donc au moment de l'exécution.
  - ⇒ Cette méthode bien que plus transparente est naturellement plus lente, mais présente l'avantage de pouvoir être reconfigurée sans recompilation.

Programmation par template (1/3)

- L'objet de ce design pattern est de séparer l'invariant d'un procédé de sa partie variante.
- Dans Spring les templates sont très utilisés dans le cadre de l'accès aux données et de la gestion des transactions.



Programmation par template (2/3)

- La partie invariante du code est placée dans la partie abstraite de la classe, ainsi toute classe qui héritera de cette classe abstraite n'aura qu'à implémenter la partie variante.
- Voici un exemple d'utilisation du pattern template tiré de la documentation de Spring :

```
tt.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
    updateOperation1();
    updateOperation2();
    }
});
```

Programmation par template (3/3)

- Voici comment est déclarée cette classe dans Spring :

```
public abstract class TransactionCallbackWithoutResult
        implements TransactionCallback {
    public final Object doInTransaction(TransactionStatus status) {
        doInTransactionWithoutResult(status);
        return null;
    }

    protected abstract void doInTransactionWithoutResult(
                                            TransactionStatus status);
}
```

- Ce qui satisfait précisément au schéma de classe déclaré ci-dessus.
- La partie variante est implémentée dans la méthode abstraite, en l'occurrence ce que vous voulez effectuer dans une transaction.

Le modèle MVC 1 et 2 (Model View Controller) (1/9)

- Ce n'est peut-être pas le concept le plus important dans Spring, dans la mesure où il a été largement démocratisé par Struts.
- Cependant encore trop de programmeurs ont tendance à mélanger toutes les couches, à mettre des traitements métiers dans la jsp, ou encore des servlets dans lesquelles ils mélangent allégrement html, javascript, métier et bien d'autres choses.
- Étant donné que l'un des objectifs les plus importants de Spring est la séparation des couches, la partie MVC et le concept d'un point de vue général semblent indispensables.
- Comme cité précédemment l'objectif est de séparer les couches et les technologies.

Le modèle MVC 1 et 2 (Model View Controller) (2/9)

- Pour réaliser la séparation entre les couches et les technologies, le paradigme se divise en trois parties :
  - Le modèle (Model).
  - La vue (View).
  - Le contrôleur (Controller).
- La couche d'accès aux données est ignorée ici parce que sous jacente à la couche métier.

Le modèle MVC 1 et 2 (Model View Controller) (3/9)

Le modèle (Model) :

- C'est la représentation des informations liées spécifiquement au domaine de l'application.
- C'est un autre nom pour désigner la couche métier.
- La couche métier ou plutôt la partie qui représente l'information en respectant une structure liée au domaine d'activité, celle qui effectue des calculs ou des traitements amenant une plus value sur l'information brute.
- Par exemple le calcul du total des taxes sur un prix hors taxe ou encore des vérifications telles que : Y a t-il encore des articles en stock avant d'autoriser une sortie de stock.

Le modèle MVC 1 et 2 (Model View Controller) (4/9)

La vue (View) :

- Cette couche ou ce module effectue le rendu de la couche métier dans une forme qui est compréhensible par l'homme, par exemple une page html.
- Attention souvent le concept MVC est associé à une application web mais ce n'est pas obligatoire : en effet le framework Swing qui permet de produire des interfaces utilisateurs "sous forme de clients lourds" permet également d'implémenter MVC.

Le contrôleur (Controller) :

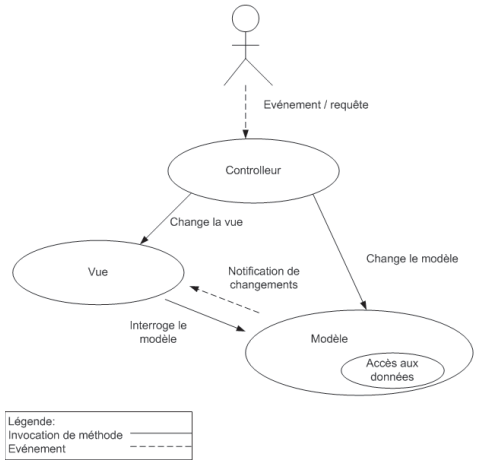
- Ce module organise la communication entre les deux premiers.
- Il invoquera une action dans la couche métier (par exemple récupérer les données d'un client) suite à une action de l'utilisateur et passera cette information à la vue (jsp qui affiche les détails).

Le modèle MVC 1 et 2 (Model View Controller) (5/9)

- La couche d'accès aux données est ignorée ici parce que sous jacente à la couche métier.
- Tout d'abord cela vous permettra de pouvoir debugger et tester plus facilement le code en l'isolant de la partie affichage.
- En effet, la plupart des bugs sont des problèmes avec l'interface donc ce n'est pas la peine de compliquer encore la donne avec des bugs métiers ou d'accès aux données.
- De plus, le jour où il vous faudra rendre l'application compatible avec les téléphones portables vous n'aurez qu'à remplacer la partie Vue par une vue qui supporte wml par exemple.

Le modèle MVC 1 et 2 (Model View Controller) (6/9)

- Le modèle MVC version 1 :

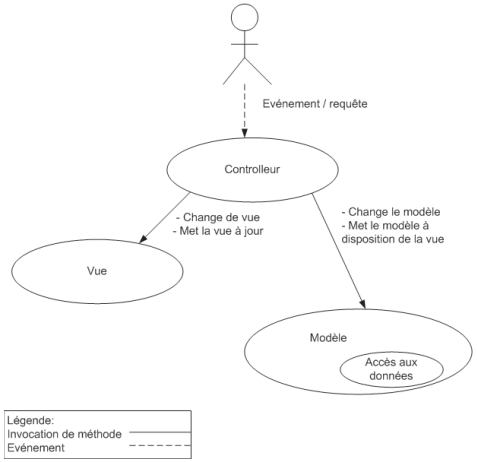


Le modèle MVC 1 et 2 (Model View Controller) (7/9)

- Le problème de ce design pattern est la partie concernant les notifications de changement dans le modèle.
- En effet, si ce point ne pose pas de problème dans le cadre de swing par exemple, où les composants de la vue sont connectés et capables d'intelligence, il n'en est pas de même pour les applications web.
- Dans le cadre d'une application web, c'est le design pattern MVC2 qui est utilisé car il ne nécessite pas l'emploi du design pattern observer.
- Ce dernier permet la notification sur les composants : il observe le modèle et permet à la vue de réagir pour se mettre à jour.

Le modèle MVC 1 et 2 (Model View Controller) (8/9)

- Le modèle MVC version 2 :



Le modèle MVC 1 et 2 (Model View Controller) (9/9)

- On remarque que c'est le contrôleur qui devient le module central.
- De fait, la vue peut à présent se contenter d'afficher sans aucune intelligence.
- Cependant, même si le design MVC permet de mieux séparer les couches, il ne faut pas oublier qu'il ne s'agit pas de la façon de procéder la plus intuitive.
- Elle induit donc d'investir du temps dans la réflexion sur la façon de séparer les différentes couches et technologies et surtout de bien réfléchir dans quelle couche s'effectue quel traitement.
- De plus les frameworks génèrent souvent plus de fichiers et naturellement plus de configuration.
- Ce surplus de complexité est cependant bien contrebalancé par la flexibilité supérieure, la plus grande fiabilité, une plus grande facilité pour tester et débugger (puisque que l'on peut tester les bouts un à un).