Chapitre 05

Les Designs Patterns

Des Designs Patterns et leurs utilisations dans la réalisation d'un programme informatique orienté objet

1. INTRODUCTION 2

2. FACTORY OU FABRIQUE DE CREATION 3

2.1. NOM ET ROLE 3
2.2. DESCRIPTION DE LA SOLUTION 4
2.2.1. VARIANTE DE BASE 4
2.2.2. UNE CLASSE ABSTRAITE DU PRODUIT : PLUSIEURS TYPES DE PRODUITS 4
2.2.3. UNE CLASSE ABSTRAITE DE FACTORY 5
2.3. EXEMPLE DE FACTORY 6
2.4. UTILISATION DU DP 6

3. LE SINGLETON 7

3.1. NOM ET ROLE 7
3.2. DESCRIPTION DE LA SOLUTION 7
3.3. EXEMPLE DE SINGLETON 8

4. L'OBSERVATEUR 8

4.1. NOM ET ROLE 8
4.2. SYNCHRONE MINIMAL 9

5. LE MVC 10

5.1. NOM ET ROLE 10
5.2. DESCRIPTION DU PROBLEME A RESOUDRE 10
5.3. DESCRIPTION DE LA SOLUTION 12
5.4. EXEMPLE DE MVC 12

1. Introduction

Les "patrons de conception" ou "Design Patterns" sont des modèles standard et réutilisables de conception de la solution à une problématique donnée dans la réalisation d'un programme informatique.

Dans une démarche de conception, quand on crée des programmes informatiques, on retrouve souvent des démarches identiques (même spécifiques à votre SI) qui se traduisent par des assemblages de composants informatiques semblables.
Certains de ces assemblages ou architecture semblables ont été modélisés et forment un ensemble de modèles qu'il faut apprendre à connaître car pourquoi réinventer la roue à chaque fois.

De la même manière que nous avons des algorithmes types (modèle de code ou méthode générique) qui reviennent souvent, pour modéliser les traitements informatiques, nous avons les design patterns pour les assemblages des classes d'objet.

Ainsi, là où un algorithme s'attache à décrire d'une manière précise comment résoudre un problème particulier, les patrons de conception décrivent des procédés de conception généraux et permettent en conséquence de mieux **capitaliser** l'expérience appliquée à la conception logicielle. Il a donc également une grande influence sur l'architecture logicielle d'un système.

L'objectif visé est la **réutilisation** des moyens de conception logicielle afin de réduire les coûts d'ingénierie et de garantir un bon niveau de qualité.

Propriétés d'un PATRON :

- Un patron est élaboré à partir de l'expérience acquise au cours de la résolution d'une classe de problèmes apparentés. Il capture des éléments de solution communs.
- Un patron définit des principes de conception, non des implémentations spécifiques de ces principes
- Un patron fournit une aide à la documentation, par ex. en définissant une terminologie, voire une description formelle (‘langage de patrons »)

Un design pattern est défini par :
- un nom
- une description du problème à résoudre
- une description de la solution : le patron de conception (schémas UML par ex) + texte explicatif.

Certains patrons sont un assemblage d'autres patrons de plus bas niveau. Cela signifie bien que ces patrons constituent bien une boite à outils de conception.
Par exemple : le design pattern MVC (Model Vue Controller) utilise les design patterns Observateur, Stratégie et Composite.

Les patrons de conception les plus connus sont plus d'une vingtaine.

On distingue 3 familles de patrons de conception selon leur utilisation :
- de **construction** (ou **création**) : ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- **structuraux** : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
- **comportementaux** : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

(En jaune les DP qui sont développés dans le cours NSY 102)
(En vert les DP qui sont développés dans le cours NFP 121 et NSY 102)

Création :

- Moniteur	Builder
- Fabrique	Factory
- Prototype	Prototype
- Singleton	Singleton

Structure :

- Interface (ou contrat)	Interface
- Delégateur	Delegate
- Inversion de contrôle	IoC
- Injection de dépendance	Injection
- Adaptateur	Adapter
- Pont	Bridge
- Objet composite	Composite
- Décorateur	Decorator
- Façade	Facade
- Poids-plume	Flyweight
- Proxy	Proxy

Comportement :

- Chaîne de responsabilité	Chain of responsibility
- Commande	Command
- Interpréteur	Interpreteur
- Itérateur	Iterator
- Médiateur	Mediator
- Mémento	Memento
- Observateur	Observer
- Etat	State
- Stratégie	Strategy
- Patron de méthode	Template Method
- Visiteur	Visitor
- Fonction de rappel	Callback == Observer == Listener
- MVC	MVC

2. Factory ou Fabrique de création

2.1. Nom et rôle

Factory ou Usine de fabrication

Une telle usine fabrique des objets, à la demande, au sens des LOO.

L'objectif : abstraire la façon de créer (en mémoire) les objets

Dans un système d'information (SI), il existe des objets particuliers qui sont des données récurrentes devant être créées de nombreuses fois durant le fonctionnement du programme, et ceci à de multiples endroits du programme.

Ces données sont des objets métiers (dans une architecture Client-Serveur, on parle de Business Object). Ces objets sont souvent stockés ou persistants dans une base de données). Le factory assure alors les propriétés liées à la base de données.

Il y a donc un besoin de centraliser cette création d'objet mais aussi d'être le moins impacté possible si la classe de ces objets évolue.

2.2. Description de la solution

La solution est de créer une classe qui va servir de point central de création de l'objet.

2.2.1. Variante de base

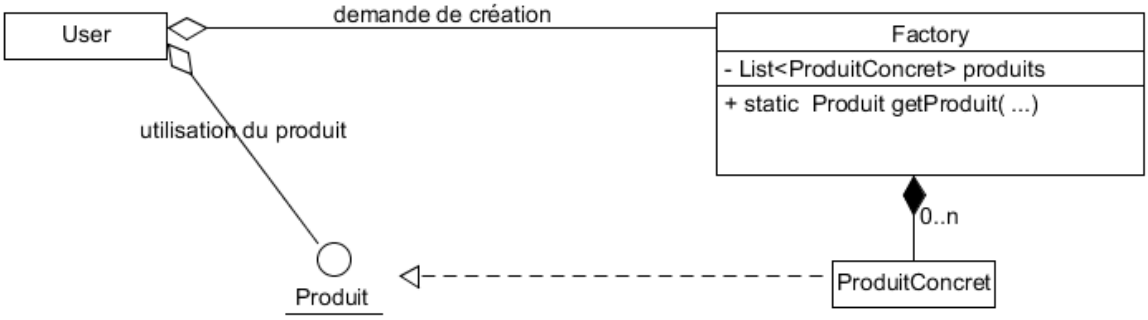
Une usine (ou factory) est une classe qui contient des méthodes de création d'objet qui reposent sur les principes suivants :

1/ Chaque méthode de création retourne une interface donnée (vision abstraite)

2/ Chaque méthode de création exploite un constructeur d'une classe (vision concrète)

3/ Cette classe concrète implémente les méthodes de l'interface

L'adhérence de l'évolution d'un factory est donc restreint aux interfaces : de Produite et de l' "interface" d'appel de la méthode de création..



Une classe utilisatrice **User** fait une demande de création d'un objet à un **Factory**.

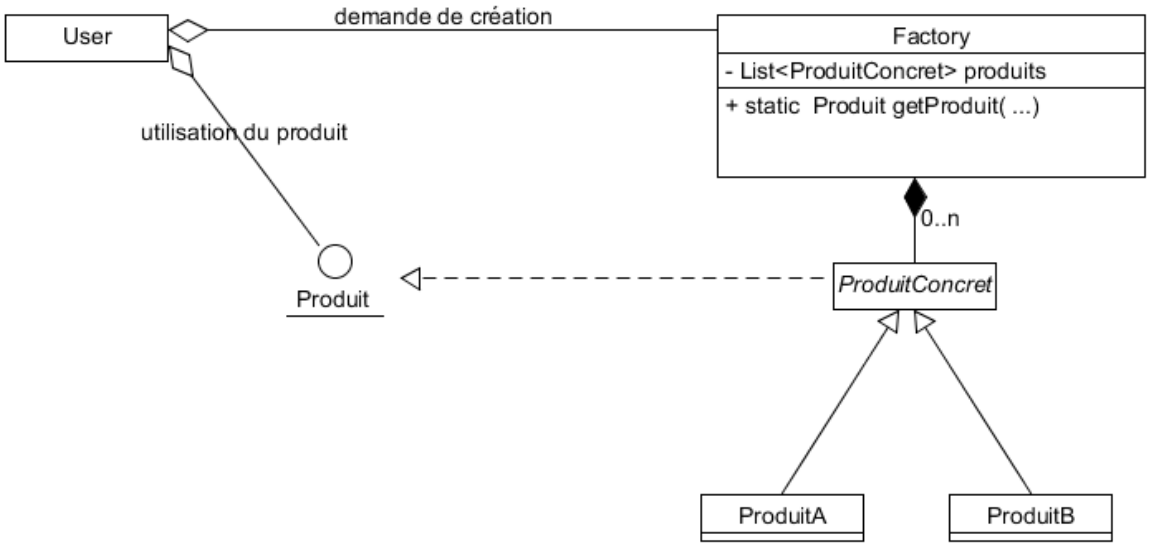
Le factory retourne un objet qui est vu par le User sous l'aspect d'une interface **Produit.**

```
Factory     f = new Factory(….) ;

Produit p = f.getProduit(…);
```

Cet objet est une instance de la classe **ProduitConcret** qui implémente l'interface **Produit**.

2.2.2. Une classe abstraite du Produit : plusieurs types de produits



Il existe des Factory qui créent des objets concrets de différentes natures en fonction des paramètres de la méthode getProduit différents du user.

Dans ce cas le factory crée des objets concrets de différentes natures dont les classes d'appartenance héritent d'une classe abstraite ProduitConcret qui implémente l'interface des produits Produit. Ceci pour que le User perçoit chacun des produits d'une manière unique.

Le factory abstrait ainsi les choix de conception et d'implémentation des objets créés.

La classe abstraite est utilisée pour créer une collection polymorphe dans le factory.

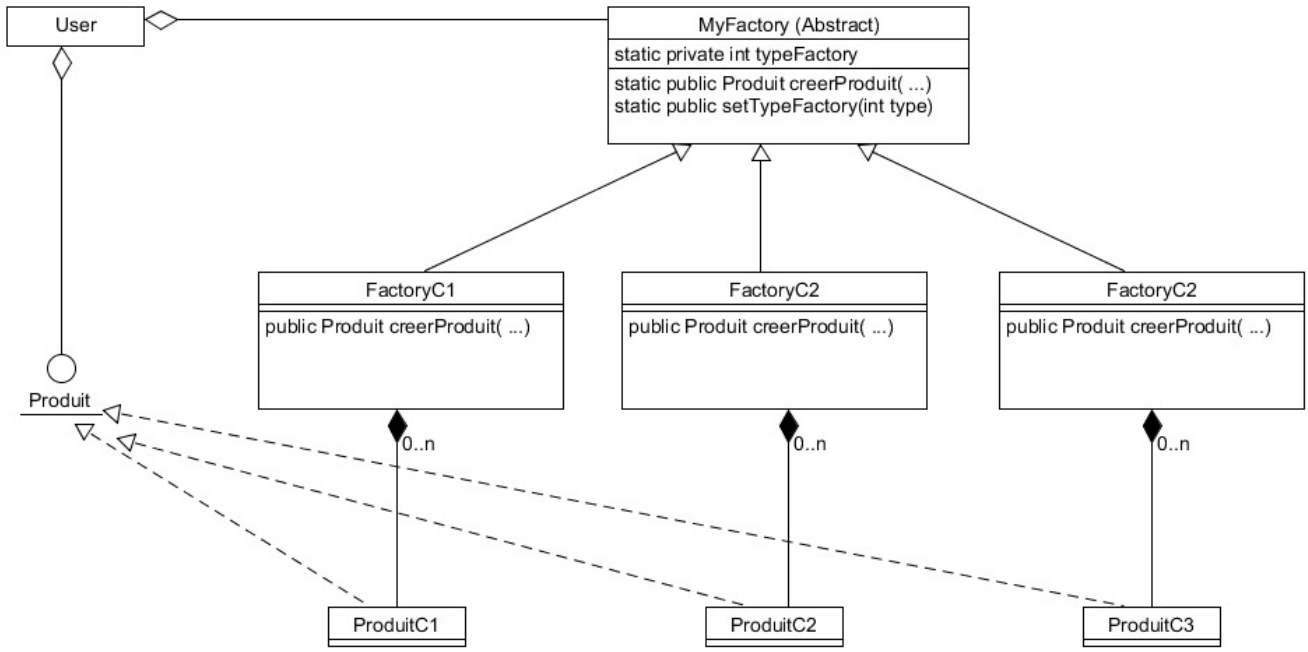
Au-delà de la création, le factory définit des méthodes de gestion des produits : recherche, accès, indexation, comparaison, tri, modifications, …

On peut aussi avoir des méthodes du factory comme :
- Produit search(….)
- void removeAll()
- void add(Produit p) // A réserver.
- … etc..

2.2.3. Une classe abstraite de Factory

La problématique : quand il existe des algorithmes indépendants pour créer les produits. Par exemple créer des objets de DAO (Data Acces Object) en fonction de différents types de bases de données.

Le principe : le factory est une classe abstraite qui implémente une interface contenant la méthode de création des objets. Des classes de factory concrètes héritent de la classe abstraite. Le choix de l'algorithme se fait par l'utilisation d'une méthode statique de la classe abstraite.



2.3. Exemple de Factory

Voir sur le site du cours NFP121 l'exemple
http://coursjava.fr/NFP121_Exemples.php?repertoire=Exemple02_FactoryBibliotheque

Commentaires en cours :

- Factory basé sur le §2.2.2
- Les services du factory
- Id
- Abstraction du produit
- Les différences avec l'exemple 01

Compléter l'exemple pour ajouter tous les champs de saisie pour AbstractMultiMedia, Livre, Film et Jeu.

Ajouter une IHM pour faire l'emprunt d'un media.

2.4. Utilisation du DP

Le DP du Factory est incontournable dans la conception d'une application informatique car dès que l'on doit gérer une "collection" d'informations que l'on veut gérer en mémoire de l'application, on peut créer un Factory.

Le Factory offre une sécurité des données, il gère le cycle de vie de ces données.

Il est souvent utilisé en interface d'une base de données pour laquelle on veut optimiser l'accès à certaines données. Il sert alors de système de cache.

Il permet un accès direct à certaines données à travers un réseau informatique en utilisant le DP de Proxy de communication.

Il permet la centralisation unique des données et est donc souvent associé au DP Singleton. Il facilite les moyens de recherche, d'interrogation d'information.

Par exemple, il gère les EJB Session dans les conteneurs de l'architecture JEE, les files de messages dans les architectures MOM, les canaux de communication, les annuaires, … etc …
Il sert de structure d'accueil à l'implémentation du système de mapping entre des données et la base de données (ex hibernate).

La liste serait longue d'énumérer tous les exemples d'utilisation d'un factory.

En résumé : Gestion de données => factory
et comme les données sont incontournables dans un SI ….

3. Le singleton

3.1. Nom et rôle

Singleton

Le rôle de ce design pattern est la création d'une instance d'objet unique dans l'Application (la JVM).

L'objectif est de limiter le nombre d'instance d'une classe qui dans le cas d'un singleton est toujours égal à 1.
Si le singleton est vu à travers une interface alors un singleton est un factory à instanciation unique

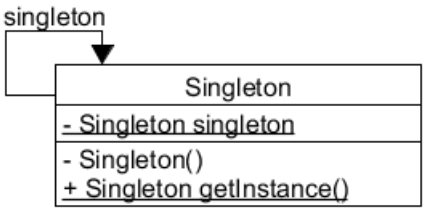
L'usage d'un singleton est principalement de pouvoir accéder à un objet principal et unique de n'importe où dans le code (sans avoir besoin de le passer en paramètre ou en attribut d'un objet).

L'accès à un tel objet se fait donc par l'appel d'une méthode statique.

Un singleton est l'équivalent strict de ce que l'on appelle dans d'autres langages : une variable globale.

3.2. Description de la solution

On crée une méthode **getInstance** qui va retourner la création du singleton mais on fera le nécessaire pour que si on appelle une seconde fois la méthode *getInstance,* au lieu de créer un nouvel objet, la méthode retournera l'objet précédemment créé.



3.3. Exemple de Singleton

```
  class CarteGriseFactory {
    // Le singleton : l'usine unique de carte grise
    //
    static private CarteGriseFactory usine = null;

    // stockage des objets de l'usine
    // Chaque carte-grise est stocké dans une table de hashing dont la clef
est
    // une référence déterminée par la classe CarteGriseImpl.
    //
    private Hashtable<String,CarteGrise> cartegrises;

    // Constructeur de création de l'usine
    //
    private CarteGriseFactory()
    {
      cartegrises = new Hashtable<String,CarteGrise>();
    }

    // Methode qui retourne le singleton
    //
    static public CarteGriseFactory getInstance()
    {
      if (usine==null) usine = new CarteGriseFactory();
      return usine;
    }
```

Voir sur le site du cours NFP121 l'exemple
http://coursjava.fr/NFP121_Exemples.php?repertoire=Exemple03_SingletonCarteGrise

Commentaires en cours :

- Implémentation du Singleton
- Démonstration
- Au passage : itération du factory de carte grise

Voir sur le site du cours NFP121
http://coursjava.fr/NFP121_Exercices.php?repertoire=Exercice05_JeuDuPenduSingleton

Re-écrire l'exercice 03 du Jeu de pendu en transformant la classe JeuPenduIHM et JeuPendu tous les deux en singletons.

4. L'Observateur

4.1. Nom et rôle

Observer ou observateur = listener

Le rôle de ce DP est de notifier des objets, appelés « observateur » d'information qui s'abonnent au préalable à un objet, l'observable, qui gèrent l'ensemble des observateurs.

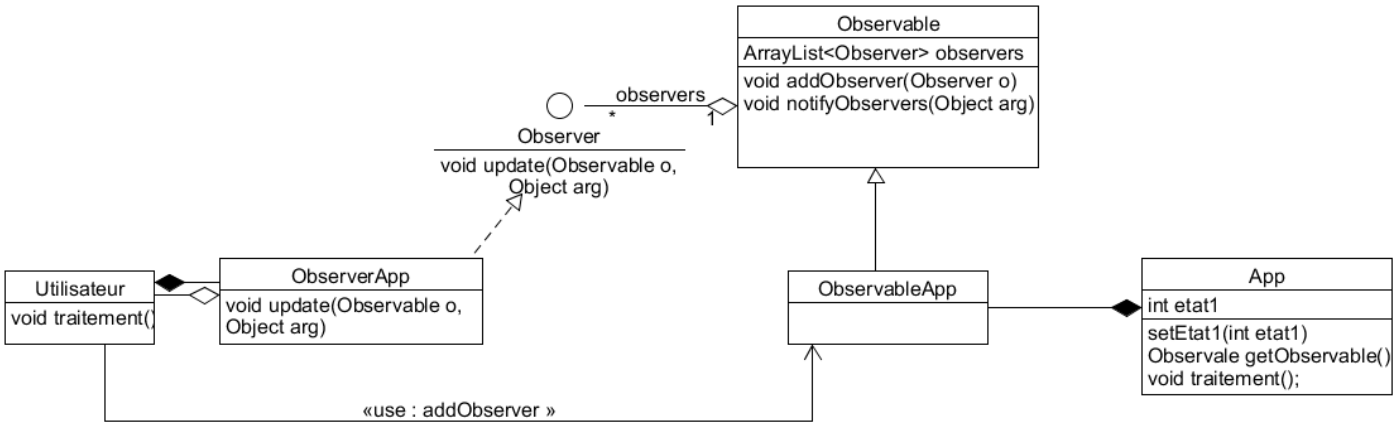
Il existe différentes formes d'observateur en fonction de l'impact sur l'objet cible et/ou du mode synchrone ou asynchrone de l'observation :

- synchrone minimal (classique)
- synchrone sur changement d'état (voir NSY 102)
- synchrone proxy (voir NSY 102)
- asynchrone push (voir NSY 102)
- asynchrone pull (voir NSY 102)

4.2. Synchrone minimal

Ce DP sert de base à tous les DP Observateur.
Les principes sont les suivants :

- Chaque utilisateur s'abonne à l'objet cible (à travers un "Observable")
- Chaque utilisateur crée un "Observer" qui est vu par l'Observable sous la forme d'une Interface
- L'Observable notifie chaque Observer du changement de son état
- Il y a au moins autant d'Observer qu'il existe d'utilisateur
- Il y a au moins autant d'Observable qu'il existe d'objet cible



Voir sur le site de NSY 102 l'exemple :
http://coursjava.fr/NSY102_Exemples.php?repertoire=ExempleCh04_10a_DPObserverSimple

La notification aux observateurs est synchrone.
Cela implique que, tant que les observateurs n'ont pas finis leurs méthodes de mise à jour (méthode update) alors l'objet cible est en attente et les autres observers aussi.

Voir sur le site du cours NFP121
http://coursjava.fr/NFP121_Exercices.php?repertoire=Exercice06_JeuDuPenduObservateur

5. Le MVC

5.1. Nom et rôle

Modèle-Vue-Contrôleur

Le design pattern Modèle-Vue-Contrôleur (MVC) est un pattern architectural qui sépare les données (le modèle), l'utilisateur du modèle (la vue) et la logique de contrôle du modèle (le contrôleur).

5.2. Description du problème à résoudre

Le problème est d'imaginer une architecture logicielle dans la réalisation d'application client/serveur (ex: Application Internet) qui assure une bonne maintenabilité de ses composants.

Cela n'est pas nouveau, il faut séparer les systèmes d'information en au moins 2 couches:
- la partie applicative qui réalise les traitements "métier" et gère les données
- la partie IHM qui réalise l'interface avec les utilisateurs

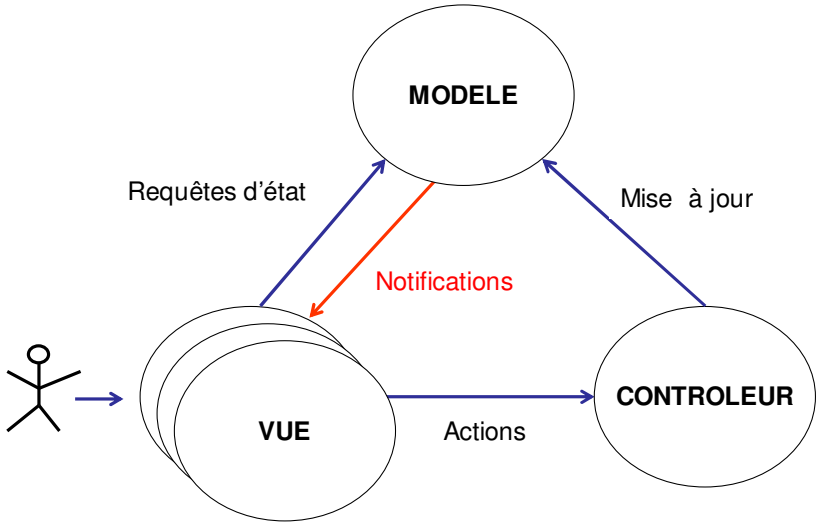
Ce qui est nouveau est l'apparition d'un nouveau composant : le contrôleur.
Ce composant est issu de la réflexion de la conception des applications internet dont le besoin est de contrôler :
- la logique d'enchaînement des vues (pages)
- le contrôle d'accès aux services (traitements)
- les appels aux services applicatifs
- …

Ce modèle de conception impose donc une séparation en 3 couches :

Le modèle : Il représente les données de l'application. Il définit aussi l'interaction avec la base de données et le traitement de ces données.
La vue : Elle représente l'interface utilisateur, ce avec quoi il interagit. Elle n'effectue aucun traitement (métier), elle se contente simplement d'afficher les données que lui fournit le modèle. Il peut tout à fait y avoir plusieurs vues qui présentent les données d'un même modèle.
Le contrôleur : Il gère l'interface entre le modèle et la vue. Il va interpréter la requête pour mettre à jour le modèle. Il effectue la synchronisation entre le modèle et les vues.

La synchronisation entre la vue et le modèle se fait avec le pattern *Observateur* (Voir l'exemple Ch04_11 plus bas) Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour.

Voici un schéma des interactions entre les différentes couches :



Les **Actions** sont contrôlées, filtrées, analysées par le CONTROLEUR. Les traitements du CONTROLEUR sont plus ou moins sophistiqués.

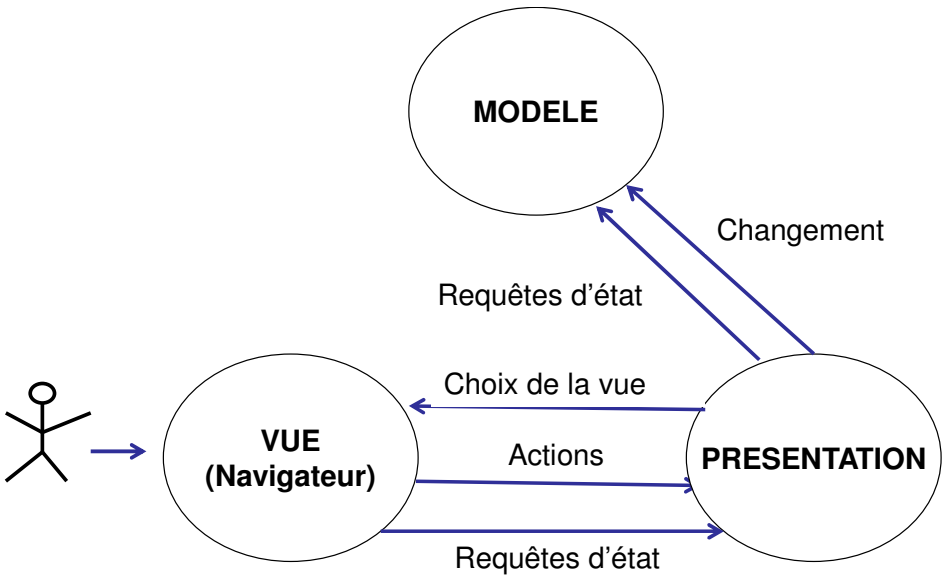
Ces traitements réalisent des **Mises à jour** du modèle. Ainsi, le MODELE réalise des traitements métiers consistant à mettre à jour l'état de ces données. Ces traitements métiers sont souvent très sophistiqués dont la mise à jour de la base de données.

Le MODELE prévient alors TOUTES les VUES en leurs faisant des **Notifications**. Ce qui permet à la VUE de se rafraichir.

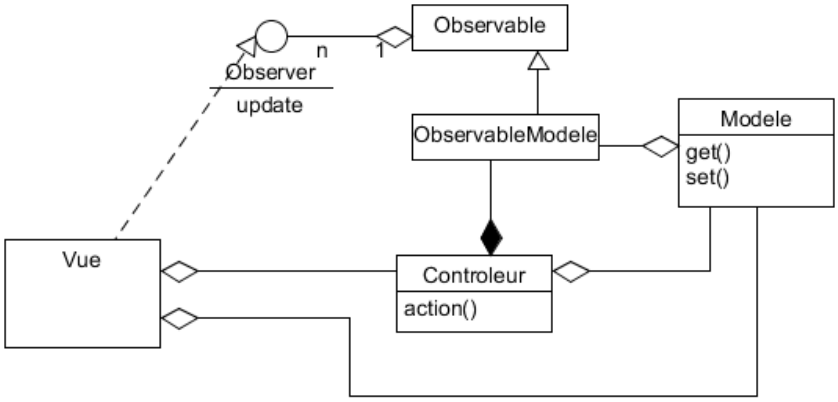
La VUE peut directement demander au MODELE les états de ses données (phase d'initialisation, requêtes cycliques de rafraichissement, …) à travers des **Requêtes d'état**.

L'utilisation de ces requêtes d'état rend la VUE dépendante du MODELE et peut nuire à la vision d'un couplage lâche entre la VUE et la couche métier de l'application.

Une variante du modèle MVC adapté au monde de l'internet le modèle MVP (P pour Présentation) :



5.3. Description de la solution



5.4. Exemple de MVC

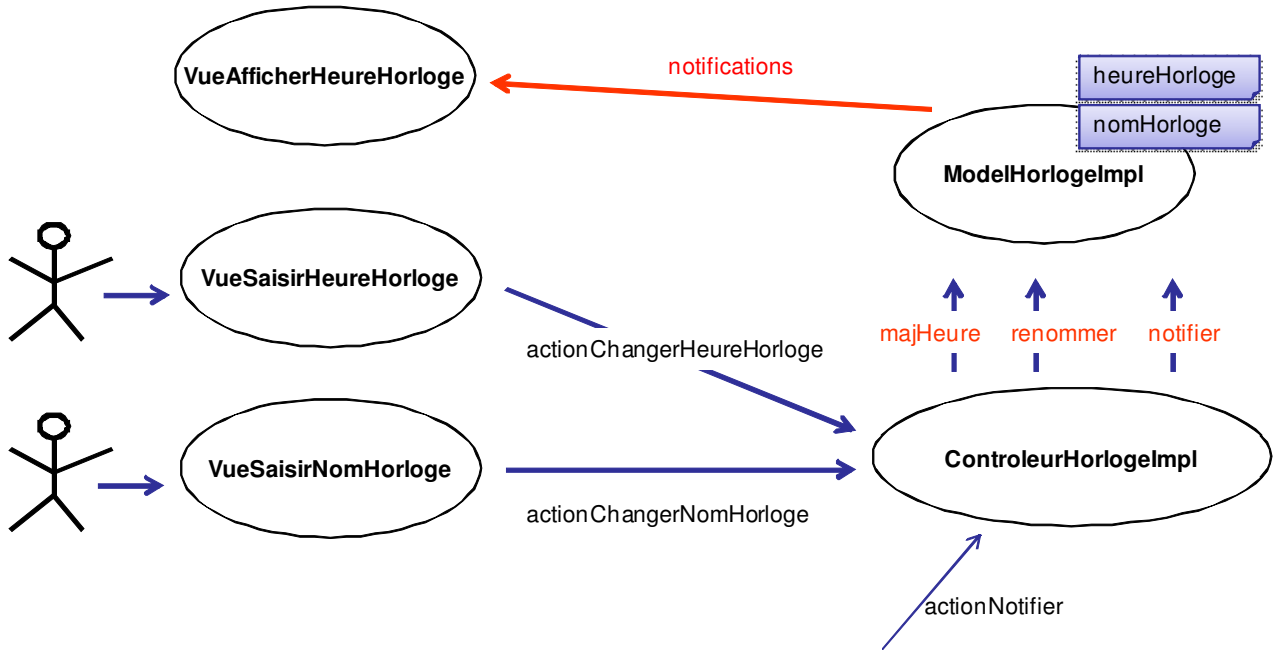
Dans cet exemple ont se propose de créer un modèle qui gère un temps d'horloge basé sur l'horloge interne de l'ordinateur.

Les 3 vues ci-dessous sont créées. Ensuite un programme principal utilise ces vues pour afficher l'heure de l'horloge, mettre à jour l'heure de l'horloge et changer le nom de l'horloge.

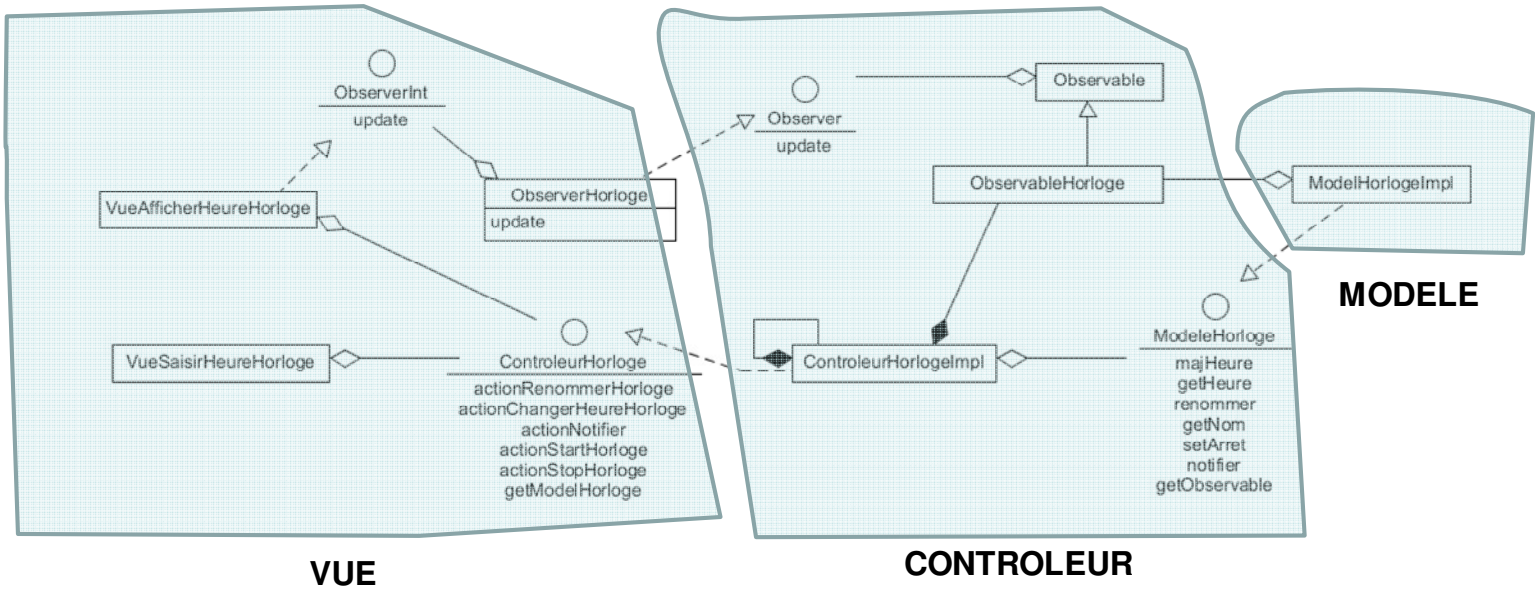
La vue d'affichage de l'heure de l'horloge est prévenue des modifications du modèle (heure et nom) par notification.

Les vues de saisi de l'heure et le nom de l'horloge passe par le contrôleur pour réaliser les actions.

Aucune des vues n'ont de lien avec le modèle.



On obtient le schéma UML suivant :



Voir sur le site de **NSY 102** l'exemple
http://coursjava.fr/NSY102_Exemples.php?repertoire=ExempleCh04_11_ModeleMVC