



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

www.heig-vd.ch

ERIC LEFRANÇOIS

Novembre 2017

POO

Part IV

MVC

& Modele « Observable-Observé »

Contenu

1	MVC & Modèle « Observable-Observé »	3
1.1	EN PRÉLIMINAIRE: LES MODÈLES DE CONCEPTION	3
1.2	LE MODÈLE MVC	4
1.2.1	LE « M » COMME « MODÈLE »	6
1.2.2	LE « V » COMME « VUE »	6
1.2.3	COUPLAGE VUE-MODÈLE: LE MODÈLE «OBSERVABLE-OBSERVÉ»	8
1.2.4	« C » COMME « CONTRÔLEUR »	12
1.3	JAVA ET LE MODÈLE OBSERVABLE-OBSERVÉ	13
1.3.1	API Observer	14
1.3.2	API Observable	14
1.3.3	LE MODÈLES DES LISTENERS (SWING)	15
1.4	MVC ET LE PATTERN «COUCHES»	15

1 MVC & Modèle « Observable-Observé »

MVC et Observable-Observé sont deux modèles de conception, - deux «Design patterns» -, qui comptent parmi les plus importants du Génie Logiciel. Avant de les étudier de manière spécifique, nous présenterons quelques généralités sur les modèles de conception.

1.1 EN PRÉLIMINAIRE : LES MODÈLES DE CONCEPTION

Les **modèles de conception**, ou encore «**schémas de conception**» sont des éléments essentiels de la fabrication des systèmes complexes, que l'on travaille dans le domaine de l'informatique ou ailleurs. Ils sont plus connus sous le vocable anglais «**design patterns**».

Dans le contexte de la programmation, il faut admettre en effet que la conception d'un logiciel orienté-objet est une tâche difficile. Très souvent, la conception élaborée par le développeur sera spécifique du problème à résoudre. Parfois, - *et c'est ce qui nous intéresse ici* -, cette conception sera suffisamment générale pour être réutilisée et adaptable à de nouveaux problèmes.

Les concepteurs d'expérience savent qu'il faut éviter de réinventer la roue à chaque expérience, qu'il ne faut pas chercher à résoudre un problème en partant de rien. Il vaut mieux réutiliser des solutions qui ont fait leurs preuves. C'est ainsi qu'une « bonne solution » sera réutilisée systématiquement.

Ces solutions sont appelées **modèles de conception**.

Cette annexe est consacrée à la présentation de deux modèles de conception fondamentaux que l'on rencontre dans la plupart des applications.

Documentation

- « Patterns in Java » de [Mark Grand] chez Wiley
- «UML & design patterns» de [Craig Larman] chez Wiley

Description d'un modèle de conception

La description d'un modèle de conception particulier s'articule autour de 3 particularités essentielles :

- **Le nom du modèle**
On parlera par exemple du modèle « MVC », du modèle du « Producteur-consommateur ». Le nom du modèle est bien entendu essentiel, est fait quasiment l'objet d'une standardisation dans le monde informatique. Cela facilite la communication entre développeurs.
- **L'énoncé du problème**
Où seront décrites la situation ou les situations dans lesquelles le modèle peut s'appliquer.
- **La solution**
Qui décrit les éléments à mettre en jeu : les différentes classes, avec leur relation et leur collaboration. Cette solution est toujours générique, le programmeur aura pour tâche de l'adapter à un contexte particulier.

1.2 LE MODÈLE MVC

MVC est sans aucun doute un des modèles les plus importants au sein de la grande famille des «[Design patterns](#)».

A l'origine, le modèle MVC a été élaboré pendant la conception du langage Smalltalk, langage orienté-objet, développé par Xerox (à Palo Alto en Californie), afin de résoudre les problèmes d'interfaçage avec l'utilisateur.

Fort de son succès, le même modèle est utilisé aujourd'hui pour concevoir l'architecture générale d'une petite application comportant une interface graphique.


Ainsi, la plupart des programmes réalisés dans le cadre de laboratoires de Programmation Orientée Objet pourraient être conçus en s'appuyant sur ce modèle.

L'utilisation du modèle MVC n'est pas limitée toutefois aux applications de petite taille ! 😊

Il a été intégré depuis dans le pattern architectural «[Couches](#)» («[Layers](#)» en Anglais). Un pattern qui sert de base à l'élaboration de la structure à grande échelle d'un système [Voir le paragraphe [MVC ET LE PATTERN «COUCHES»](#) en page 15].

Principe du modèle MVC

L'objectif de l'architecture est de séparer et de découpler les 3 composantes d'un « [module](#) », voir « **Note 1** » ci-dessous :

- La composante **Modèle**
Cette composante comprend une de plusieurs classes qui regroupent les structures d'informations manipulées par le module.
- La composante **Vue**
 Cette composante comprend également une ou plusieurs classes !
Ces dernières réalisent l'interface avec l'utilisateur. La vue n'est pas limitée à l'affichage du modèle ; elle peut comprendre des composants graphiques tels que boutons, menus ou encore champs de saisie, qui permettent à l'utilisateur d'interagir avec l'application.
- La composante **Contrôleur**
Cette dernière composante est responsable du contrôle général du module, comme notamment de l'ordonnancement des opérations.



Dans la plupart des cas, la composante contrôleur ne comprend **qu'une seule classe** ! Toutefois, dans le cas problèmes complexes, le contrôleur peut déléguer une partie de son travail à une série de sous-contrôleurs (les « adjoints du directeur »..)



Note 1 : Par « [module](#) », nous entendons un bout de programme responsable de la mise en œuvre d'une certaine fonctionnalité (p.e. dans GAPS, le module qui serait responsable de la gestion des professeurs).



Note 2 : Par « [découpler](#) », nous entendons faire en sorte que les liens entre les différentes composantes soient le plus faible possible, que ces composantes présentent un maximum d'indépendance les unes par rapport aux autres.

Les avantages obtenus

La mise en œuvre du modèle MVC présente essentiellement deux avantages :

- **Maintenance et mise au point**

Conçu de cette manière, un module sera plus facile à mettre au point et à maintenir : la séparation claire des 3 composantes M, V et C permet de localiser facilement la partie à corriger ou à modifier. Cette dernière sera en général, - *et avec un peu de chance* -, limitée à l'une ou à l'autre des 3 composantes, et pourra être modifiée indépendamment des deux autres.

- **Réutilisation**

Si les trois composantes sont fortement découplées les unes des autres, elles seront facilement réutilisables dans le cadre de nouvelles applications.

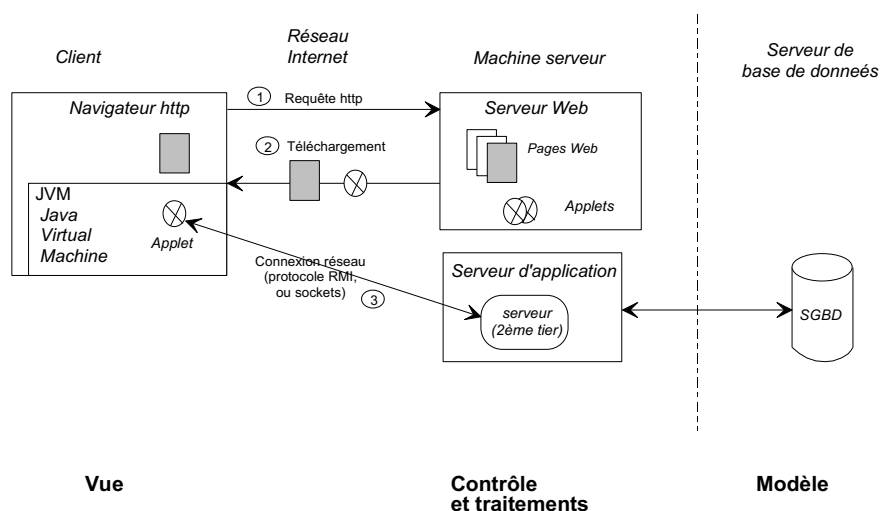
Quelques analogies avec le modèle MVC

Dans le monde informatique, on peut retrouver certains principes sous-jacents au modèle MVC dans maintes architectures. Notamment, l'idée de séparer très clairement les composantes **Présentation**, **Informations** et **Traitements** n'est pas étrangère au concept.

Par exemple, le modèle client-serveur «3-tiers» décompose l'application distribuée en trois couches: Client - Application serveur - Base de données. L'objectif du modèle tient principalement à la séparation des 3 composantes:

- Le «**Client**», on l'on trouve tout ce qui a trait à la présentation, et que nous pourrions assimiler à la «**vue**» du modèle MVC.
- Le «**middle-tier**»: l'application serveur, qui regroupe tout ce qui concerne les traitements, mais aussi le contrôle de l'application, et qui se rapproche très fortement du «**contrôleur**» du modèle MVC.
- La **base de données**, dans la quelle sont enregistrées les structures de données persistantes de l'application, et que l'on pourrait assimiler à la partie «**modèle**» de MVC.

Figure 1: Application Client-Serveur 3-tiers, analogie MVC



1.2.1 LE «M» COMME «MODÈLE»

Un modèle est un objet qui participe à la définition des données manipulées par l'application.

Définition 1: La couche Modèle

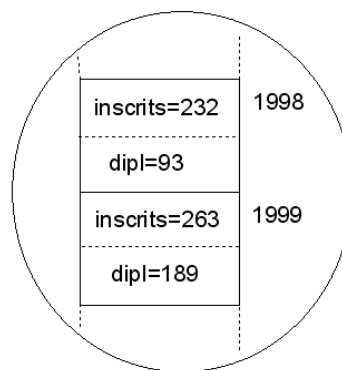
La couche Modèle est typiquement un **paquetage** qui comprend l'ensemble des objets modèles de l'application.

Un exemple

Supposons une application destinée à établir et visualiser le bilan d'une école telle que la nôtre. Cette application manipulera pour chaque année deux types d'informations : le nombre d'inscriptions (`nbInscrits`), et le nombre de diplômés (`nbDiplômés`).

`nbInscrits` et `nbDiplômés` peuvent constituer un modèle de notre application (typiquement, un tableau dynamique dont chaque entrée le nombre d'inscrits et le nombre de diplômés pour une année donnée).

Figure 2: Le modèle, un tableau dynamique



1.2.2 LE «V» COMME «VUE»

Une vue est un objet participant à la réalisation de l'interface avec l'utilisateur. Elle permet :

- D'une part de visualiser la valeur d'un objet modèle.
- D'autre part, et le cas échéant, de modifier la valeur d'un tel objet.

Une vue est en général un objet complexe, composite, qui peut contenir des «sous-vues».

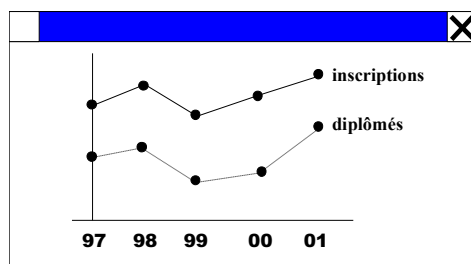
Typiquement, une vue sera composée de graphiques, de champs d'affichage, de champs de saisie, de boutons, etc.. Souvent, on désigne les vues par le terme anglais «**GUI**» («**G**raphical **U**ser **I**nterface»).

Définition 2: La couche Vue

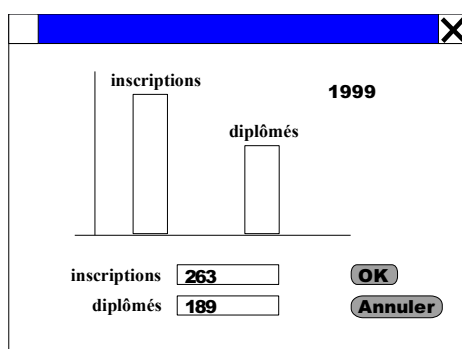
La couche Vue est typiquement un **paquetage** qui comprend l'ensemble des vues de l'application.

En reprenant l'exemple présenté plus haut, nous pouvons imaginer que notre modèle est associé à 3 « vues », qui peuvent être représentées simultanément :

- Un graphique, utilisé uniquement pour visualiser le modèle pour les années 1997 à 2001 ;



- Un graphique, utilisé pour visualiser le modèle pour une année spécifique, offrant la possibilité d'en modifier la valeur au moyen de deux champs de saisie et d'un bouton «OK» ;



- Un tableur, utilisé pour visualiser et pour modifier les données du modèle.

A spreadsheet with a blue title bar and a close button (X). The table has three columns: 'inscriptions', 'diplômés', and an unlabeled column for years. The data is as follows:

	inscriptions	diplômés
1997	210	165
1998	249	172
1999	204	89
2000	232	93
2001	263	189

Below the table are two buttons: 'OK' and 'Annuler'.



Ces 3 vues pourraient être présentées simultanément à l'écran de l'ordinateur. Dans ce cas, si une saisie est effectuée dans l'une ou l'autre des deux dernières vues, il serait intéressant que la modification soit reportée de manière quasi simultanée dans les deux autres vues.. On verra cela plus loin.

1.2.3 COUPLAGE VUE-MODÈLE: LE MODÈLE «OBSERVABLE-OBSERVÉ»

Entre le modèle et ses différentes vues, MVC peut s'appuyer parfois sur un modèle sous-jacent : le **modèle observable-observé**, un modèle de conception à part entière.

La non-réutilisation des couches Vue & Contrôleur

On se doit de constater que la partie interface utilisateur d'une application est fortement dépendante de l'application elle-même et a peu de chance, à ce titre, d'être réutilisable d'une application à l'autre.

Il en est de même de la couche Contrôleur, qui est propre à l'application.



En règle générale, c'est donc la couche Modèle (p.e. une « Pile d'informations ») qui présente le plus de chance de pouvoir être réutilisée.

De manière toute aussi générale, le modèle est naturellement découplé du contrôleur : il attend en effet d'être utilisé par ce dernier mais il ne le connaît pas, à la manière d'un serveur qui attendrait les requêtes d'un client sans connaître ces derniers.

Par contre, il n'est pas rare que le programmeur du modèle ait introduit dans le code du modèle des parties concernant son affichage, ce qui n'est pas bien du tout... Il faut donc dans ce dernier cas découpler le modèle de sa vue.

Séparer le Modèle de la Vue

Tenant compte de ce qui précède, il est donc souhaitable qu'il n'y ait aucun couplage, - *ou à la limite un couplage très faible* -, entre la Vue et le Modèle. C'est ce que l'on appelle le principe de **séparation Modèle-Vue**.



Objectifs :

- Permettre de développer séparément les couches Modèle et Vue
- Réduire l'impact de l'évolution des besoins de l'interface sur la couche Modèle
- Permettre d'ajouter facilement de nouvelles vues sans avoir à modifier la couche Modèle
- Permettre d'avoir plusieurs vues simultanées sur le même objet modèle
- Permettre de déployer la couche Modèle et la couche Vue de manière distribuée (couches communiquant à distance sur le réseau, avec le modèle du côté Serveur et la Vue du côté Client).
- Améliorer la portabilité, la réutilisation de la couche Modèle

Pour appliquer ce principe, on peut remarquer :

- Que les objets du Modèle **ne doivent pas «connaître» directement les objets de la Vue**, savoir comment les manipuler en leur demandant par exemple d'afficher quelque chose, de changer de couleur, etc.
En revanche, **les objets de la couche Vue peuvent connaître directement les objets de la couche Modèle et savoir comment on les utilise**. La couche Vue n'a pas la prétention d'être une couche réutilisable et peut donc être dépendante des autres composantes.
- Que les objets de la Vue sont dotés de peu d'intelligence et doivent se contenter de gérer les entrées/sorties, de capturer les événements de l'interface, mais ne doivent surtout pas offrir de fonctionnalités propres à l'application (le propre des composants Modèle et/ou Contrôleur).

Comment appliquer ce principe ?

- **Modèle de scrutation**
Une première solution consiste à appliquer le principe de séparation au pied de la lettre: les objets Modèle n'envoient aucun message à la couche Vue.
Dans cette solution, on va mettre en œuvre un modèle de scrutation: les vues qui le désirent vont interroger à intervalles de temps réguliers (par exemple toutes les secondes) tous les objets modèles qui la concernent de manière à rafraîchir leur interface.
- **Modèle Observable-Observé**
Le modèle de scrutation – à savoir *interroger à tour de rôle tous les objets Modèle* - peut convenir dans certains cas. Par contre, s'il s'agit d'une application de surveillance, - comme par exemple la surveillance du trafic d'un réseau de télécommunications ou d'une application de simulation s'appuyant sur des milliers d'objets modèle-, le mode de scrutation peut s'avérer **très inefficace**.



Idée du modèle Observable-Observé

→ Plutôt que d'être surveillé régulièrement par les différentes vues, chaque objet modèle aura la responsabilité à chaque changement d'état d'envoyer une notification de changement à toutes les vues concernées.

Concrètement, **un objet Modèle ne connaîtra les objets Vue qu'en termes d'objets génériques répondant à une interface rudimentaire (« interface » au sens Java) qui sera limitée à seule et unique méthode : une méthode de notification de changement d'état.**

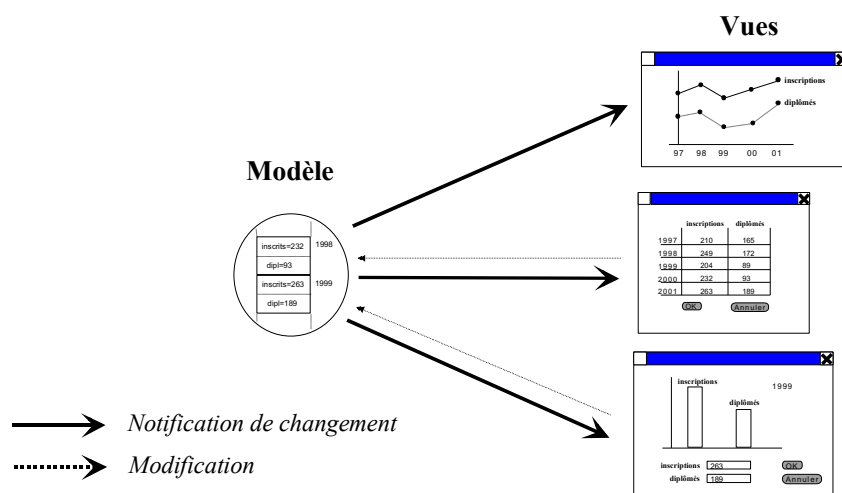
L'interface étant limitée à une seule méthode, - la *méthode de notification* -, ce modèle se caractérise par un faible couplage entre la couche Modèle et la couche Vue.

Principe du modèle Observable-Observé

Ce modèle décrit l'interdépendance entre un objet (l'observable), qui change d'état, et tous les objets qui dépendent de ce changement d'état (les observateurs).

Note : Dans cette première approche, on fera abstraction du contrôleur. Ce dernier sera introduit un peu plus loin.

Figure 3: Le modèle Observable-Observé



Enoncé du problème

- **Découplage..**
L'objectif du modèle est de découpler autant que faire ce peut le sujet observable des observateurs. Ainsi, les classes qui représentent l'observable et les classes qui représentent les observateurs pourront être modifiées indépendamment les uns des autres et pourront être réutilisées séparément.
- **Attente passive: notification par «callback»**
L'objet observateur sera notifié par «callback» de l'occurrence d'un changement d'état: il sera rappelé par le biais d'un envoi de message lui signalant l'événement.
Etant déchargé d'avoir à interroger continuellement le sujet observable, l'observateur peut faire de « l'attente passive » (*terminologie de la programmation concurrente*), et libérer ainsi des ressources système.

Solution

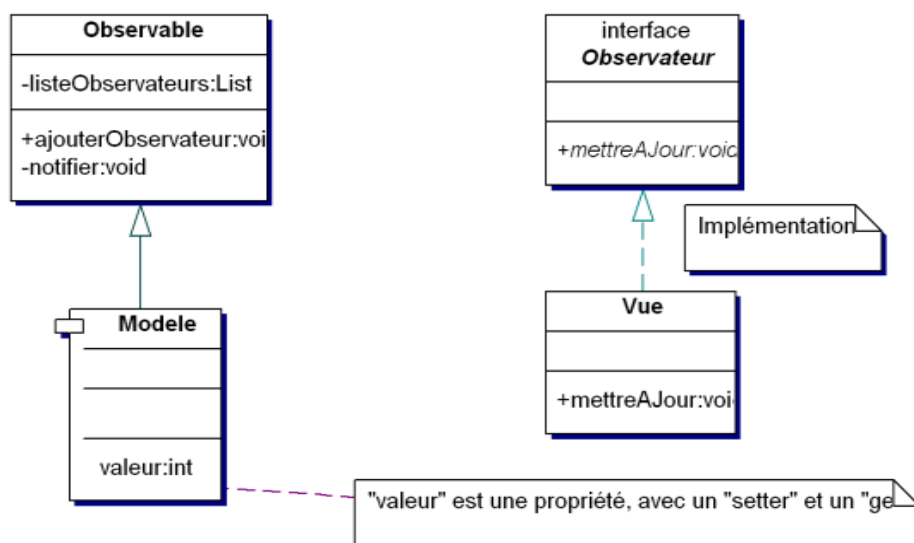
Lorsque le sujet « observable » est modifié, il diffuse des notifications à tous ses « observateurs »: «Coucou, je suis modifié!». Il le fait de manière uniforme, sans connaître la nature des observateurs.

Les observateurs doivent au préalable avoir souscrit aux notifications, en «s'abonnant» auprès du sujet observable.

Une fois notifiés, les observateurs interrogent le sujet pour connaître son état. En règle générale, les observateurs ne s'intéressent qu'à une partie de l'état du modèle.

Diagramme de classes du modèle Observable-Observé

Voici un diagramme de classes représentant le modèle MVC, exprimé en UML.



- Les différentes sont abonnées au modèle par le biais du message `ajouterObservateur`

Par exemple, une vue peut s'abonner elle-même au près du modèle :

```
leModèle.ajouterObservateur(this) ;
```

- Le modèle inscrit chaque nouvel observateur dans une liste dynamique interne.
- En réponse à toute modification de valeur (comme par exemple via un message `setValeur`), le modèle invoque la méthode privée `notifier()`, qui a pour effet de diffuser le message `mettreAJour()` à tous les abonnés.
- Une fois notifiée, une vue peut connaître la nouvelle valeur du modèle en l'interrogeant (comme par exemple via le message `getValeur`).

L'interface Observateur

Cette interface doit être implémentée par tous les observateurs qui désirent s'inscrire auprès d'un observable.

Cette interface **assure un découplage très fort entre les observateurs et les « observables »** : le paramètre de la méthode `ajouterObservateur` est de type `Observateur`. Ainsi, **la classe même des observateurs peut être absolument quelconque** pour autant qu'on y trouve la méthode `mettreAJour`.

L'observable n'a pas à connaître ni le nombre ni la nature des classes qui se mettront à son écoute. Notamment, l'observable n'a pas à connaître à quelle partie de son état s'intéresse l'observateur qui vient de s'inscrire: il lui suffira de lui envoyer la notification `mettreAJour`.

La classe Observable

Cette classe met en oeuvre la mécanique d'inscription et de notification. Elle peut être dérivée (par héritage) par toutes les classes qui implémentent un modèle.



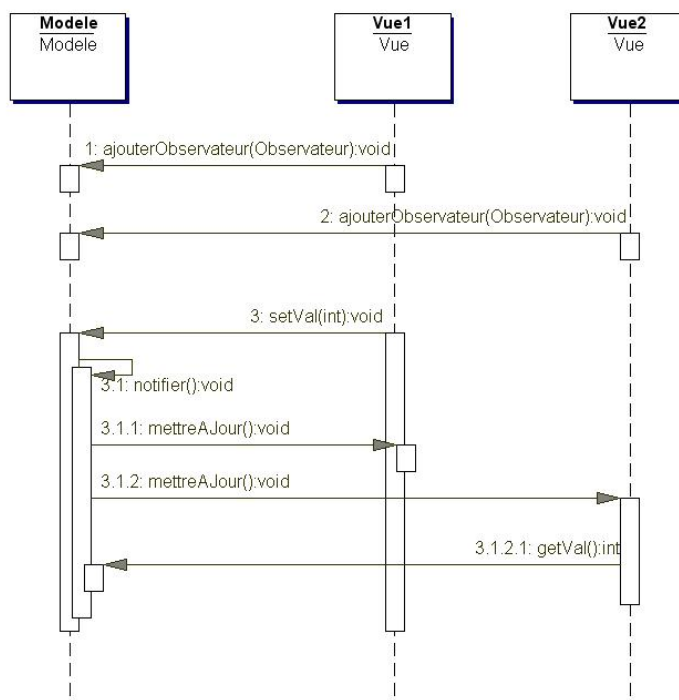
Remarques sur le mode d'abonnement des vues auprès d'un modèle

Dans l'exemple ci-dessus, « `leModèle.ajouterObservateur(this)` » ; », c'est la vue elle-même qui s'abonne au modèle.

On peut envisager d'autres manières de faire. Par exemple que ce soit le contrôleur qui abonne la vue au modèle: « `leModèle.ajouterObservateur(laVue)` » ; »

Diagramme de séquence

Voici maintenant un diagramme de séquence illustrant le modèle Observateur, exprimé encore une fois en UML.



Dans le cadre de ce scénario, on peut remarquer que la vue 1 a provoqué un changement d'état du modèle. Toutes les vues sont alors notifiées - y compris la vue 1 -. En réaction, la vue 2 interroge le modèle pour connaître la nouvelle valeur.



Remarques :

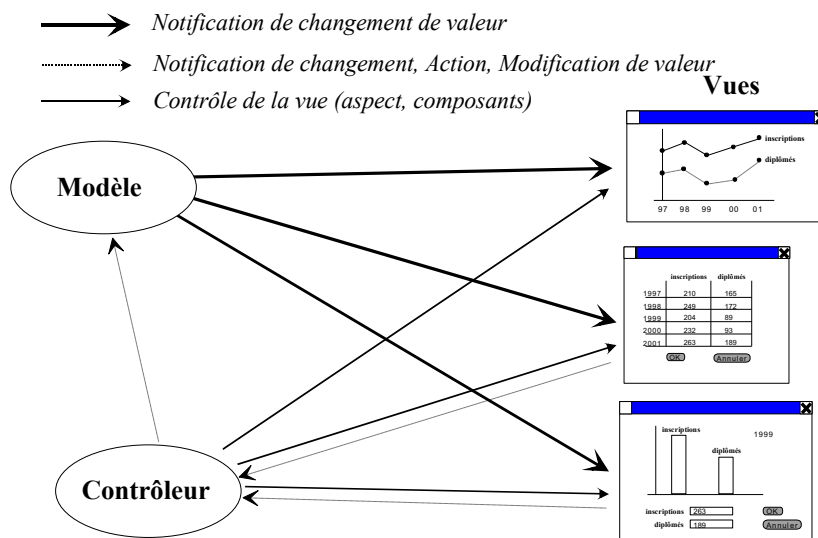
- 😊 Parmi toutes les méthodes qui apparaissent dans ce diagramme, seules les méthodes `setVal` et `getVal` doivent être programmées ! Les autres sont en effet héritées.
- Les méthodes `setVal` et `getVal` sont invoquées par les vues, ce qui prouve que ces dernières «connaissent» le modèle et savent quels messages on peut lui envoyer.

1.2.4 «C» COMME «CONTRÔLEUR»

Il arrive fréquemment que le contrôleur soit absent du modèle MVC.

Son rôle est de définir les réactions face aux actions de l'utilisateur (modification de données, pression de bouton, sélection dans une liste, etc..).

Typiquement, un contrôleur est maître d'œuvre, un « big boss », qui définit l'ordre des opérations à effectuer («workflow» en anglais) . Notamment, c'est le contrôleur qui autorise ou non les actions commandées par l'utilisateur, c'est le contrôleur qui agit sur l'aspect des différentes vues (en activant ou non tel ou tel bouton, etc..).



Elimination du contrôleur

Si le contrôleur se contente de jouer les intermédiaires entre le modèle et les différentes vues, sans filtrer ou sans ajouter « d'intelligence » spécifique, il est possible de l'éliminer : on peut alors gagner en flexibilité et en efficacité.

Si on élimine le contrôleur, il est vivement conseillé de placer son intelligence dans le Modèle (on choisira par exemple un des objets modèle en lui confiant le rôle du contrôleur) plutôt que dans la Vue. La vue devrait rester le plus « bête » possible.



De manière générale, il est déconseillé d'éliminer le contrôleur !

Pour quelle raison ? Pour conserver une structure de code la plus claire qui soit.

1.3 JAVA ET LE MODÈLE OBSERVABLE-OBSERVÉ

La librairie Java met à disposition l'interface **Observer** et la classe **Observable**. Cette interface et cette classe sont définies dans le package « **java.util** ».

La classe **Observable** peut être utilisée de deux manières :

- Par héritage : en dérivant la classe de l'objet observable de la classe **Observable**;

- Par association: si notre observable hérite déjà d'une autre classe, - *et comme Java est limité à l'héritage simple* -, il nous suffira de lui adjoindre un sous-objet de type `Observable` à qui les messages seront relayés, au moyen par exemple d'une classe interne.

1.3.1 API Observer

```
interface Observer {
    public void update(Observable o, Object arg);
}
```

La méthode `update` est invoquée à chaque fois que l'objet observé invoque sa méthode `notifyObservers` (voir ci-dessous la classe `Observable`). Tous les observateurs sont alors automatiquement notifiés.

Paramètres:

- L'objet observé (utile si le même observateur observe plusieurs objets différents)

arg Un argument passé par la méthode `notifyObservers`

1.3.2 API Observable

`Observable` est une classe qui contient notamment les méthodes suivantes :

```
public void addObserver(Observer o)
```

→ Pour ajouter un nouvel observateur à la liste des observateurs de l'objet

```
public void deleteObserver(Observer o)
```

→ Pour supprimer un observateur de la liste

```
protected void setChanged()
```

→ Pour enregistrer le fait que l'état de cet objet a été modifié: passage du flag interne «modified» à l'état `true`.

```
public void notifyObservers()
```

→ Pour notifier tous les observateurs que cet objet a été modifié. L'argument `arg` reçu par la méthode `update` de l'observateur vaudra `null`.

Cette notification a lieu si et seulement si la méthode `setChanged()` a été invoquée au préalable. L'exécution de `notifyObservers` a pour effet de remettre le flag interne «modified» à `false`.

```
public void notifyObservers(Object arg)
```

→ Pour notifier tous les observateurs l'objet passé en paramètre a été modifié. L'argument `arg` sera reçu par la méthode `update` de l'observateur (deuxième paramètre de la méthode `update`).

Typiquement, cette méthode sera invoquée par le modèle en mettant la valeur «`this`» en paramètre.

Comme pour `notifyObservers()`, cette notification a lieu si et seulement si la méthode `setChanged()` a été invoquée au préalable. L'exécution de `notifyObservers` a pour effet de remettre le flag interne «modified» à `false`.

1.3.3 LE MODÈLES DES LISTENERS (SWING)

Le modèle Observable-Observé est efficace dans la mesure où l'observable ne génère qu'un seul type d'événements. Dans le cas contraire, les observateurs risquent d'être notifiés pour des événements qui ne les intéressent pas forcément, générant une perte d'efficacité de l'application.

A titre d'exemple, considérons l'exemple du composant Swing `JButton` en Java. Ce dernier est susceptible de générer différents types d'événements :

- Des événements de type `Action` (bouton pressé);
- Des événements de type `Focus` (le bouton a obtenu le focus);
- Des événements de type `Key` (une touche a été pressée alors que le bouton a le focus);
- Des événements de type `MouseEvent` (la souris est déplacée sur le bouton);
- etc..

L'application directe du modèle Observateur impliquerait un abonnement à tout l'ensemble des événements, même si seule la pression du bouton nous intéresse. Pour éviter cette lourdeur, les concepteurs du langage ont mis en place un modèle inspiré du modèle Observateur.

Intitulé «**modèle des listeners**», ce modèle permet à tout observateur potentiel de s'abonner uniquement à un sous-ensemble particulier d'événements. A cette fin, des méthodes spécifiques d'inscription sont mises à disposition: `addActionListener`, `addFocusListener`, `addKeyListener`, etc...

1.4 MVC ET LE PATTERN «COUCHES»

Un simple modèle de conception à son origine, le modèle MVC a fait longue route!

Il a été intégré depuis dans le pattern **architecturalCouches** («**Layers**»). Un pattern qui sert de base à l'élaboration de la structure à grande échelle d'un système.

On y retrouve la Vue (couche Présentation), le Modèle (couche Domaine) et le Contrôleur (couche Application)

Figure 5: Le pattern Couches (Layers)

