INFO-H-301
Programmation orientée objet

TP5 - ***Patrons de conception***

F. Servais & B. Verhaegen

Programmation Orientée-Objet

- "Controlling complexity is the essence of computer programming."
(Brian Kernighan)
- Programmation procédurale : Données, Opérations
- Programmation orientée-objet : (Données + Opérations)

Règles de bonne pratique

- Low Coupling
- High Cohesion
- http://en.wikipedia.org/wiki/GRASP_(Object_Oriented_Design)

Coupling / Dépendance

- Attribut
- Paramètre de méthode
- Variable locale
- Valeur de retour
- Héritage ou implémentation

Low Coupling

- Le couplage est une mesure de la dépendance d'un classe envers d'autres classes
- Limiter le couplage est essentiel pour un code maintenable
- On doit pouvoir modifier une classe sans toucher aux autres
- Valable pour les parties de l'application susceptible de changer

Low coupling: Loi de Demeter

- A qui une méthode peut-elle envoyer un message?
- Uniquement à:
  - l'objet courant: `this`
  - un paramètre de la méthode
  - un attribut ou un élément d'une collection en attribut
  - un objet crée à l'intérieur de la méthode

Low coupling: Loi de Demeter

- Le but est d’éviter que la classe ne dépende d’objets ‘éloignés’
- En d’autres mots: “Don’t talk to strangers”
- Eviter d’avoir des appels en chaîne (augmente le couplage):

```
sale.getPayment().getAmount()
```

High Cohesion

- Un objet forme un tout
- Une partie d’un objet ne constitue pas un objet
- Idéalement, toutes les méthodes d'une classe utilisent toutes ses données

Design Patterns

Qu'est-ce qu'un pattern ?

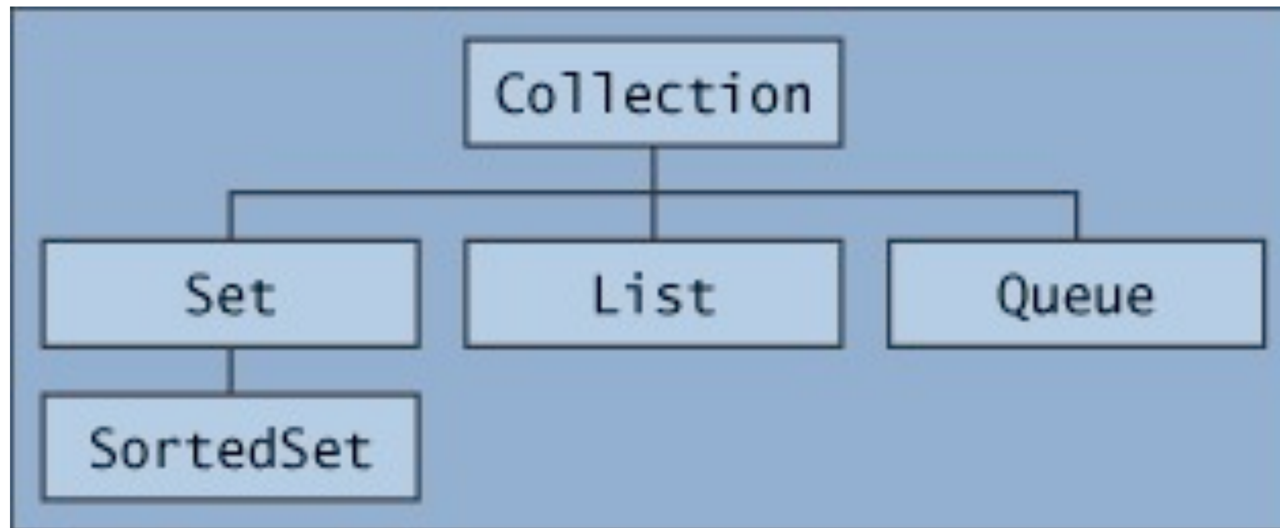
Solution à un problème connu

- Nom
- Description du problème
- Description de la solution : structure, participants, collaborations
- Intention
- Conséquences

Iterator

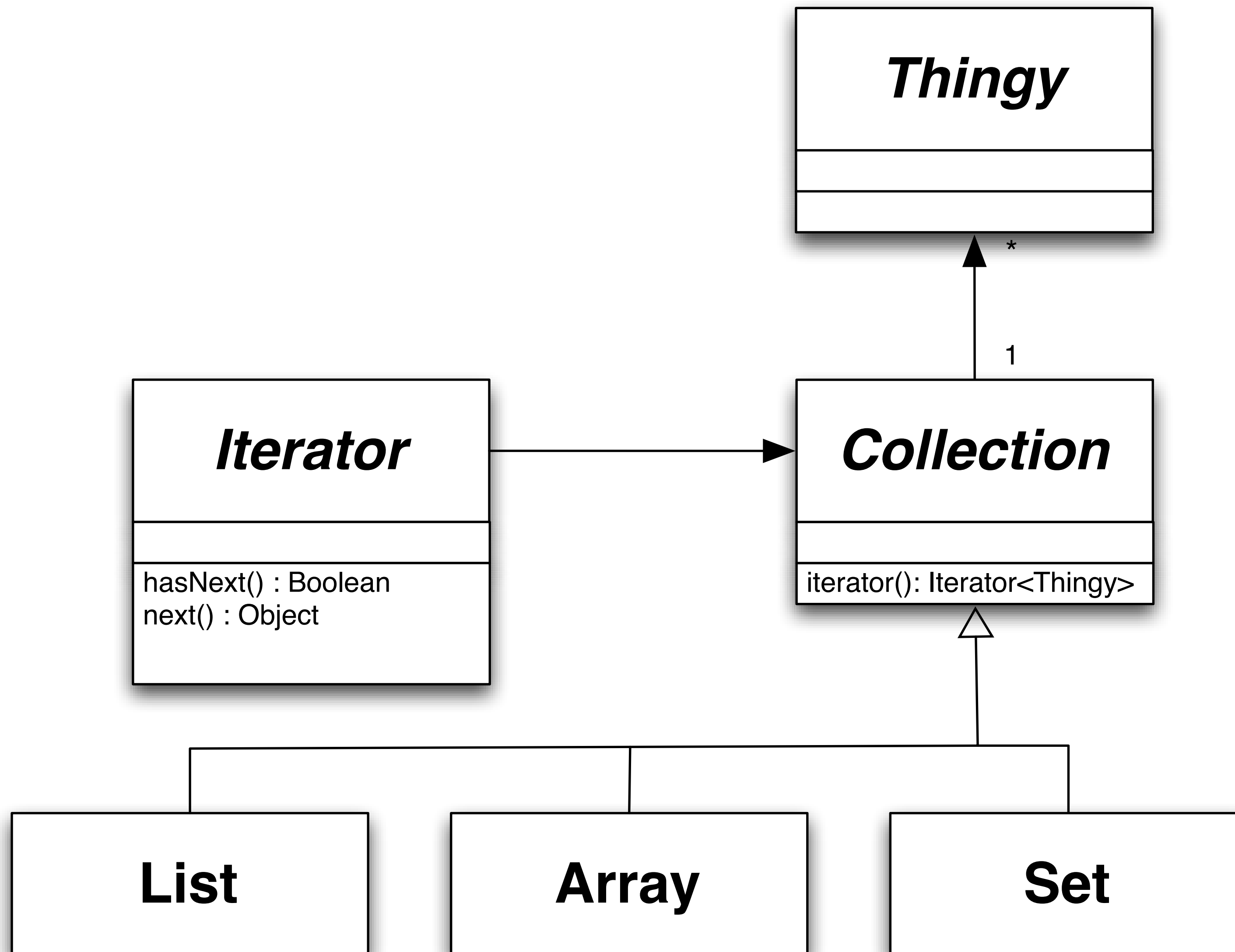
- Problème : on désire accéder aux éléments d'une collection indépendamment de la structure de celle-ci
- Solution : passer par un iterator qui définit les méthodes *hasNext*() et *next*()
- Conséquences : plus haut niveau d'abstraction mais perte de flexibilité
- Catégorie : comportemental

Iterator



http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html

Iterator



Iterator & Java *For* Loops

```
for(int i=0 ; i<basket.size(); i++) {
   Egg egg = basket.get(i);
   egg.break();
}


for(Iterator<Egg> iterEggs = basket.iterator(); iterEggs.hasNext(); ) {
   Egg egg = iterEggs.next();
   egg.break();
}
```

Iterator & Java *For* Loops

```
for(int i=0 ; i<basket.size(); i++) {
   Egg egg = basket.get(i);
   egg.break();
}
```

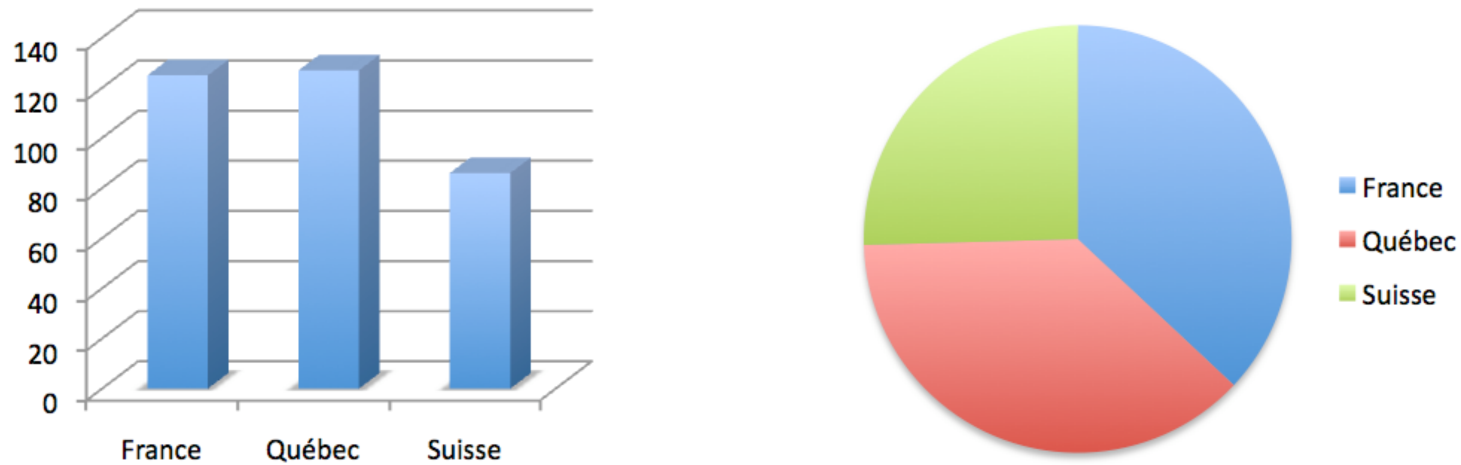
```
for(Iterator<Egg> iterEggs = basket.iterator(); iterEggs.hasNext(); ) {
   Egg egg = iterEggs.next();
   egg.break();
}
```

```
for(Egg egg: basket) {
   egg.break();
}
```

Observer

- Problème : plusieurs objets, les observateurs, sont notifiés lorsqu'un objet, le sujet, change d'état
- Solution : Remplacer la dépendance de B par une dépendance sur une interface minimaliste
- Conséquences : couplage abstrait, broadcast, mises à jour en cascade
- Catégorie : Structurel

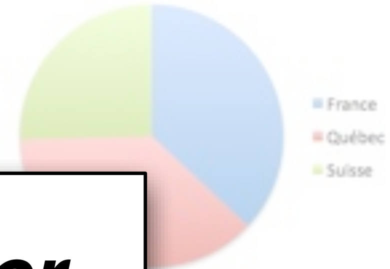
Observer Pattern

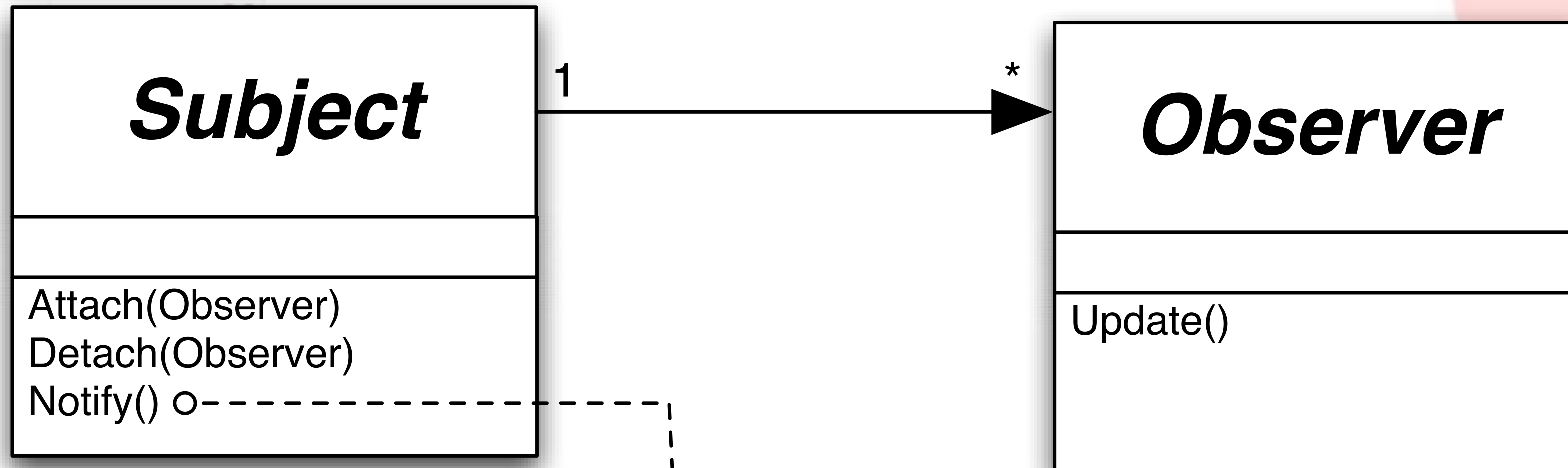


France	125
Québec	127
Suisse	86

Observer Pattern

France	125
Québec	127
Suisse	[illegible]



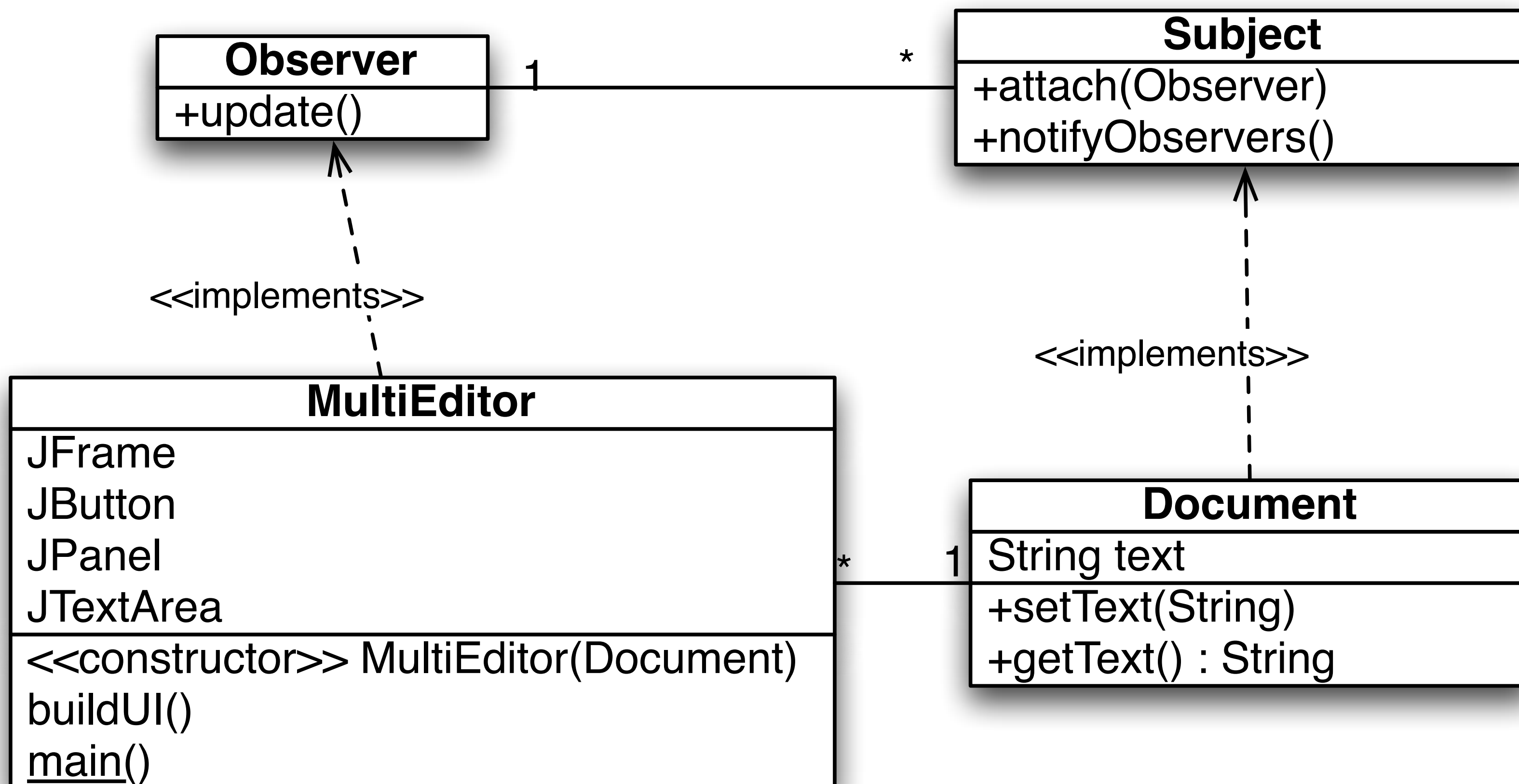



```
for all o in observers {
   o→Update()
}
```

Observer

- Un *observateur* observe des *observables*.
- En cas de *notification*, les observateurs effectuent une action en fonction des informations qui viennent des observables.
- La notion d'*observateur/observable* permet de coupler des modules de façon à réduire les dépendances aux seuls phénomènes observés.
- Notion *d'événements*.

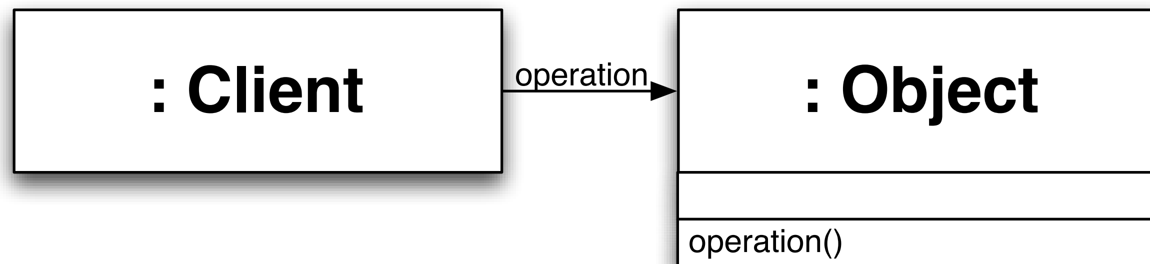
Exemple



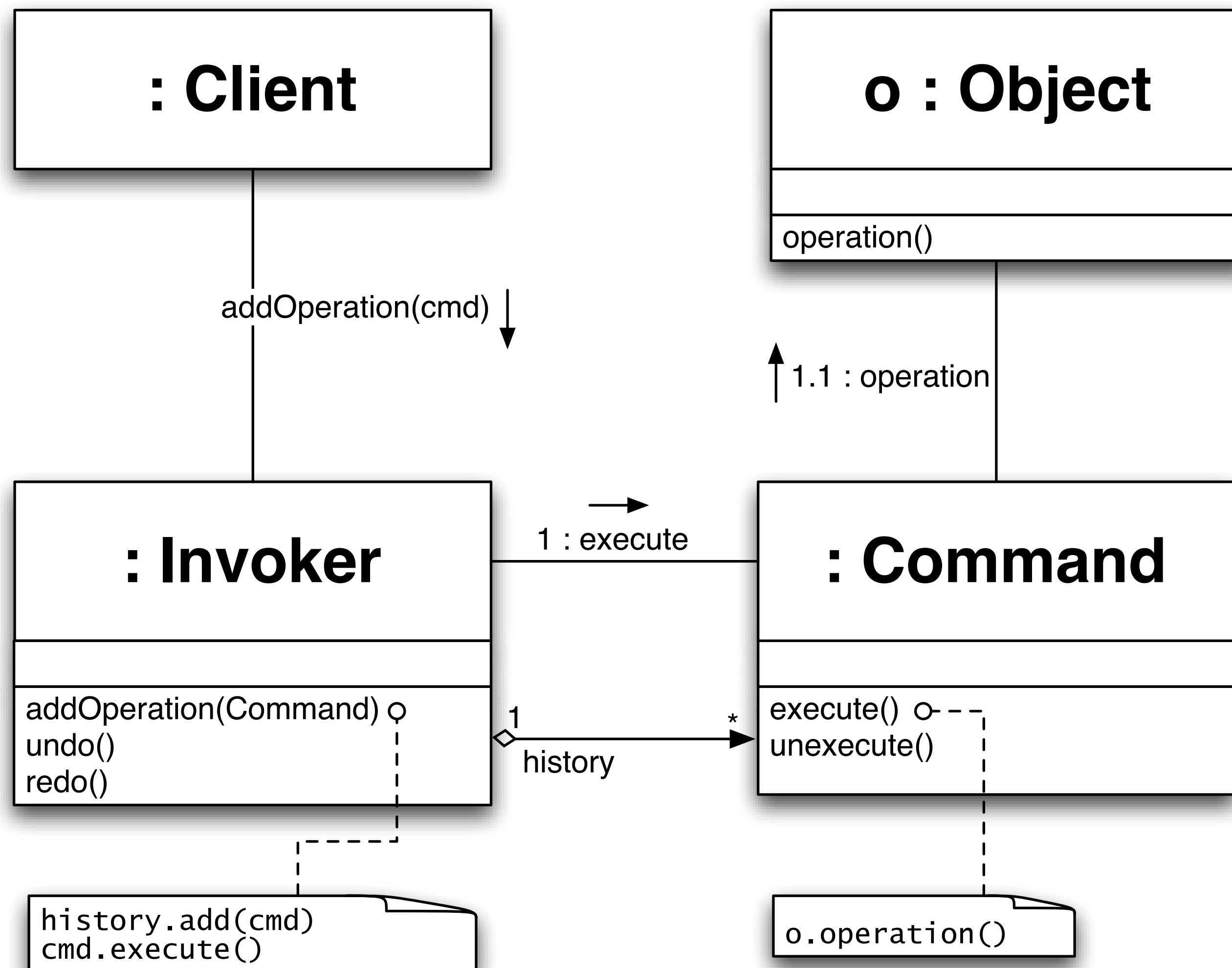
Command

- Problème : annuler et/ou sauvegarder des actions
- Solution : représenter chaque action par une instance d'une classe
- Conséquences : supporte logging, undo, redo
- Catégorie : comportemental

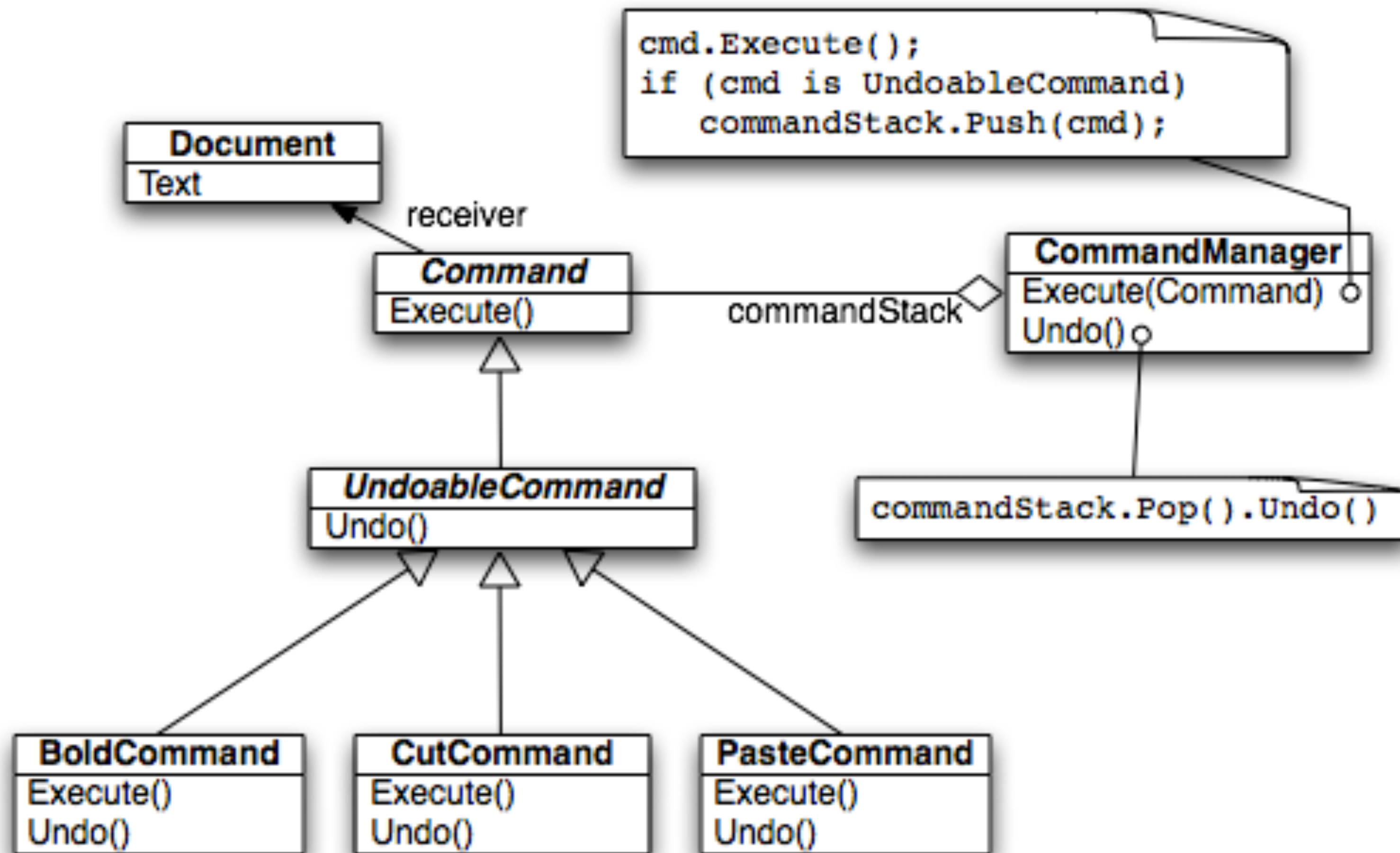
Sans Command Pattern



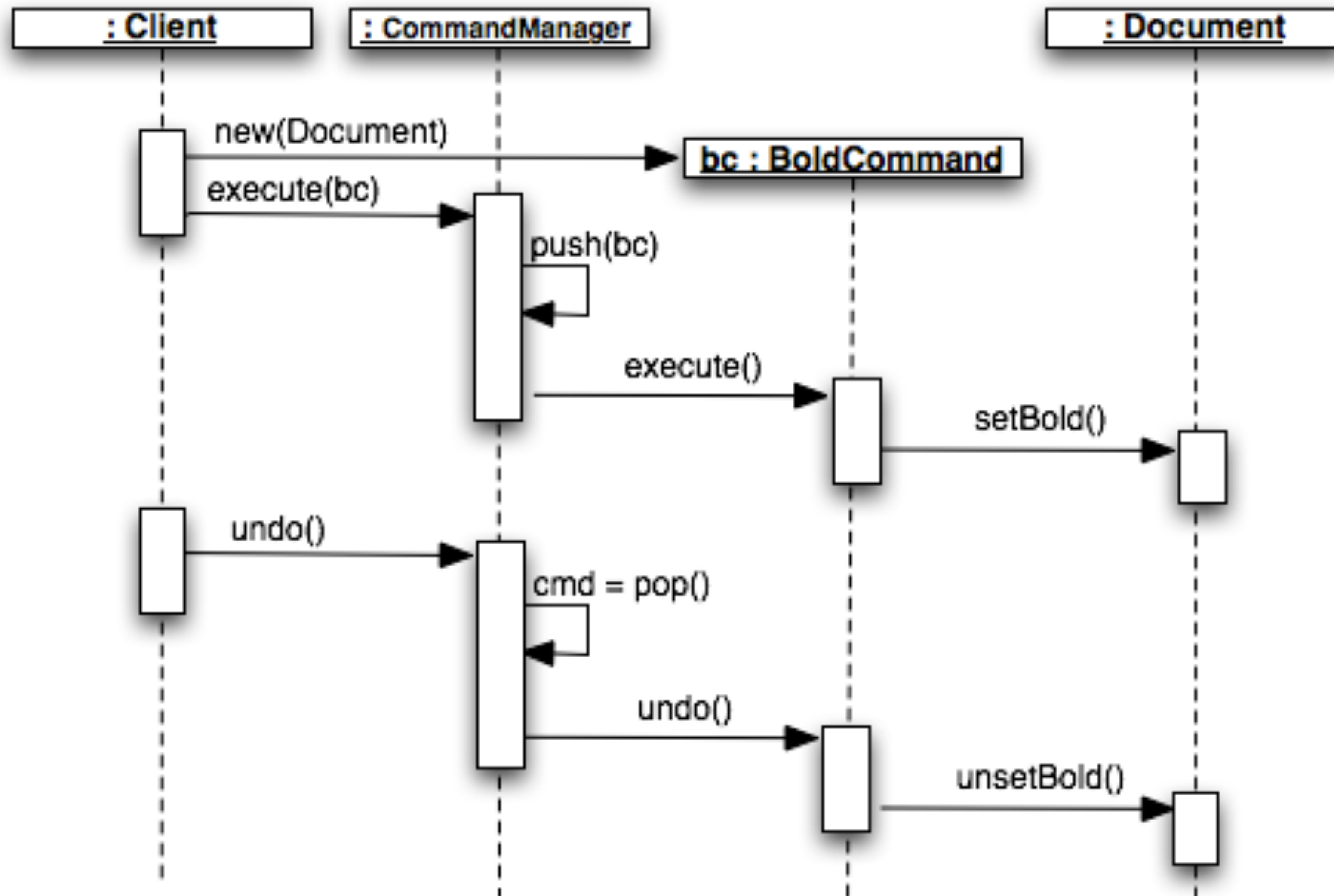
Avec Command Pattern



Exemple



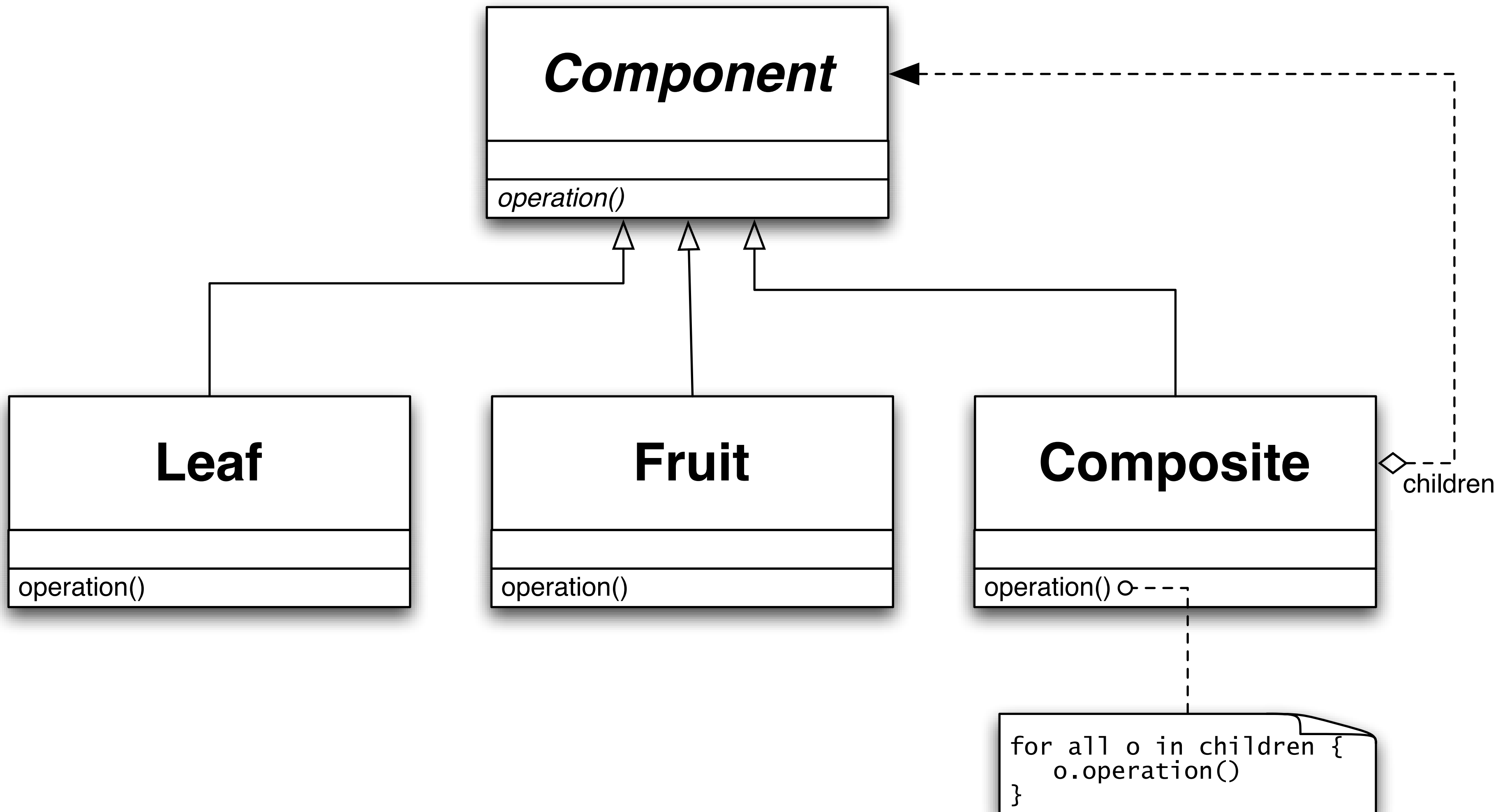
Exemple



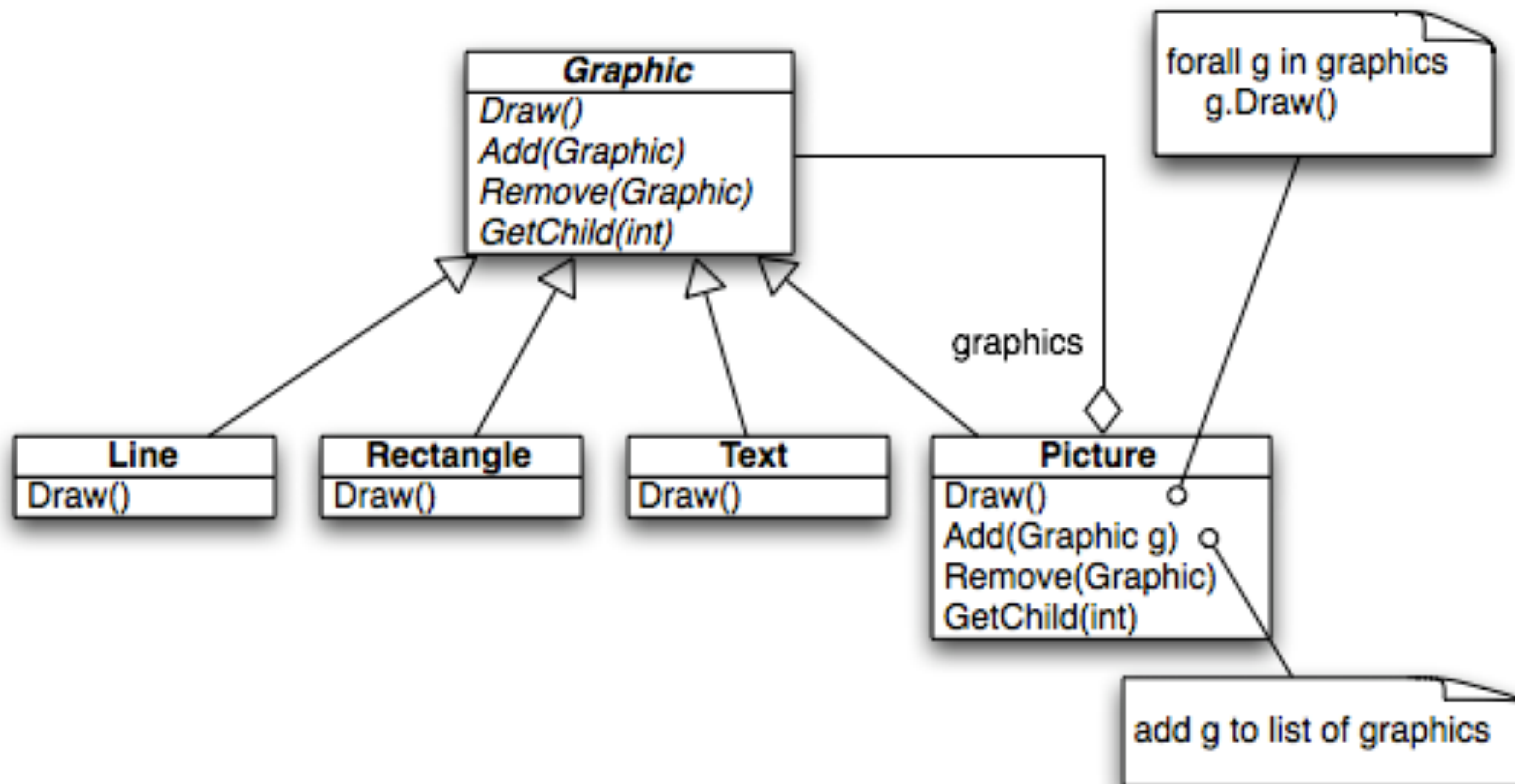
Composite

- Problème : représenter des données possédant des caractéristiques récursives
- Solution : définir une classe abstraite dont hérite les contenus et les conteneurs
- Conséquences : facile d'ajouter des contenus et conteneurs, simplifie le code client
- Catégorie : structurel

Composite



Exemple



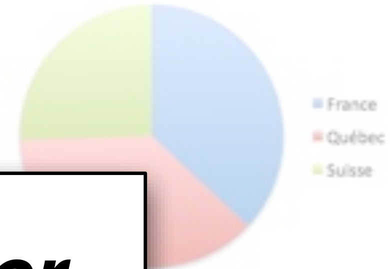
INFO-H-301
Programmation orientée objet

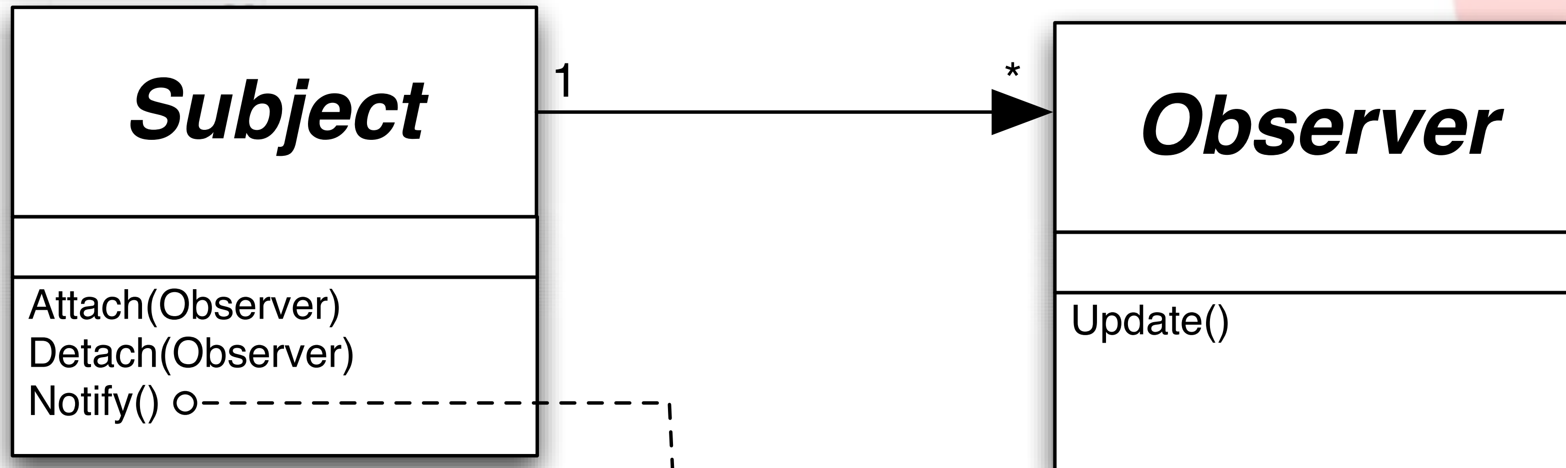
TP5 - ***Model-View-Controller***

F. Servais & B. Verhaegen

Observer Pattern

France	125
Québec	127
Suisse	

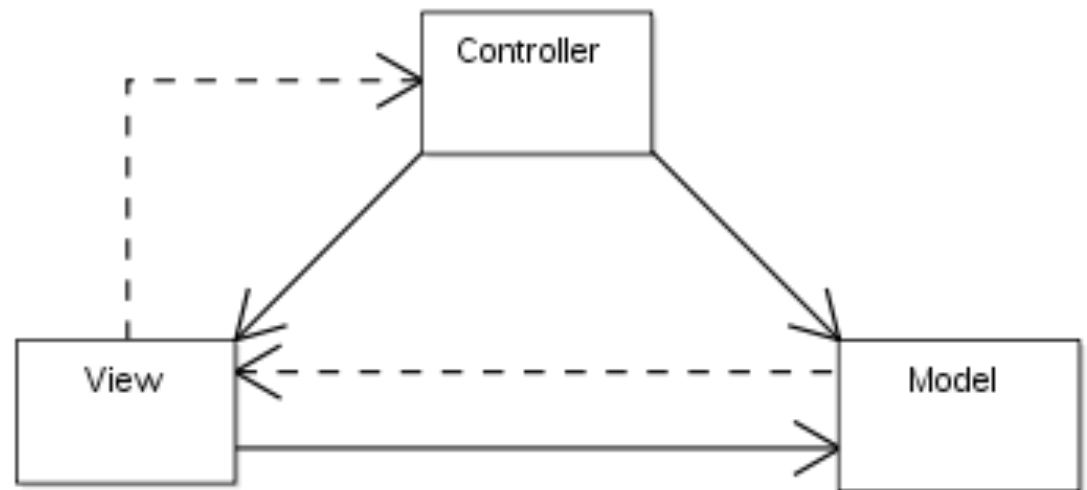





```
for all o in observers {
   o→Update()
}
```

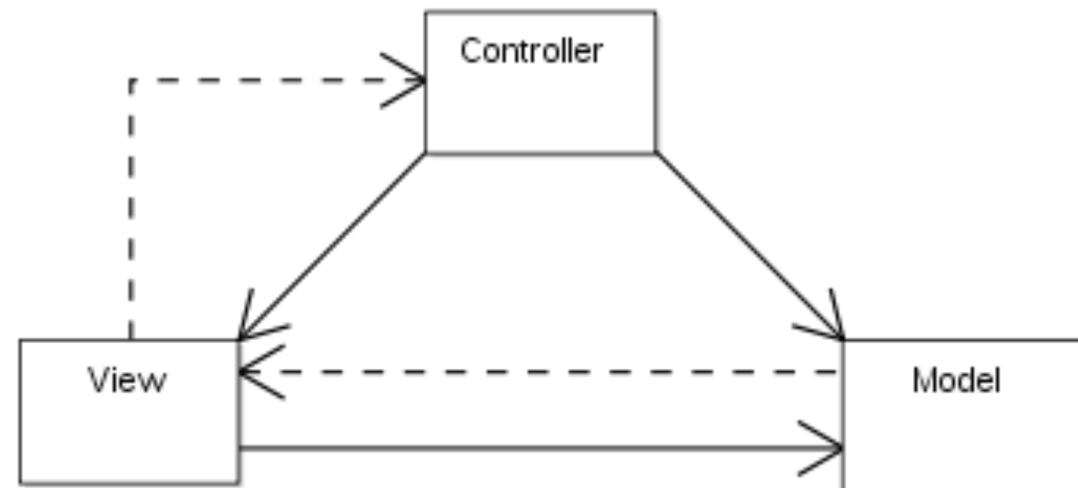
Le patron Model-View-Controller

- Architecture de la partie interface utilisateur d'une application
- Séparation en 3 parties:
  - Modèle: données
  - Vue: interface visuelle
  - Contrôleur: interactions



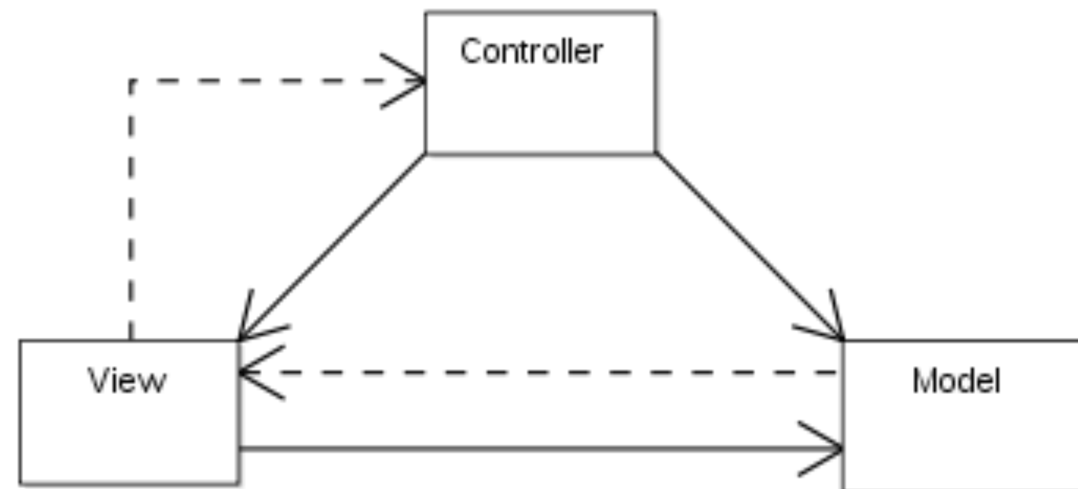
Modèle

- Garant de l'intégrité des données
- Le seul à modifier les données
- Ne connaît ni la vue, ni le contrôleur.
- Interagit avec la vue via un Observer (découplage)



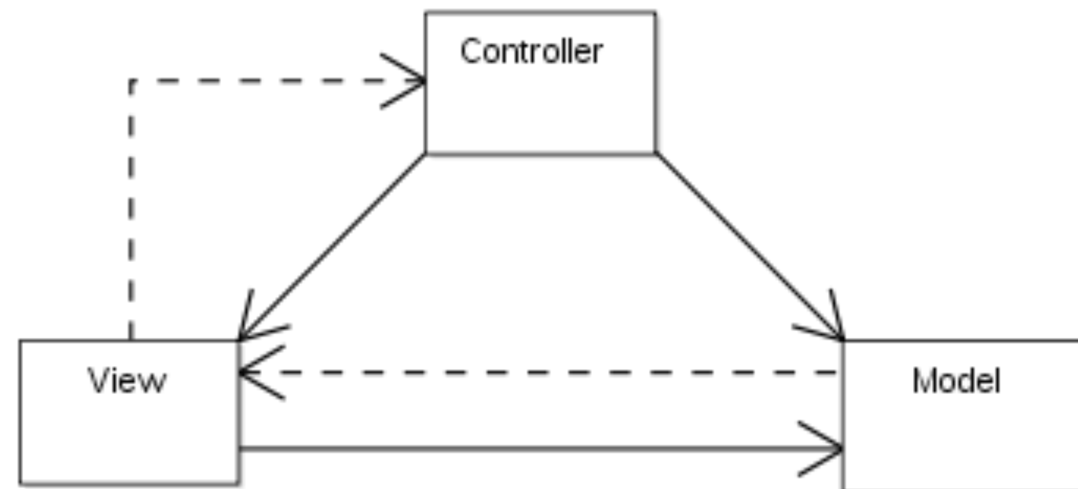
Vue

- Présente les données
- Reçoit les actions des utilisateurs et les transmettre à un contrôleur
- La vue connaît le modèle, et est notifié par Observer de tout changement
- La vue ne connaît pas les contrôleurs et les notifie via un Observer



Controller

- Reçoit les événements de la Vue
- Transforme ces événements en mises à jour à effectuer sur le modèle
- Le contrôleur connaît la vue car doit comprendre les événements
- Le contrôleur connaît le modèle et lui transmet les mises à jour



Résumé

- L'utilisateur clique sur quelque chose
- La vue reçoit le click et notifie tous les contrôleurs intéressés
- Le contrôleur analyse l'événement et demande au modèle de faire les mises à jour correspondantes
- Le modèle met à jour les données et notifie les Observer (la vue)
- La vue se met en jour en demandant au modèle les données nécessaires