Observateur et bus d'événements

DUT M3105 : Conception et programmation objet avancées

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

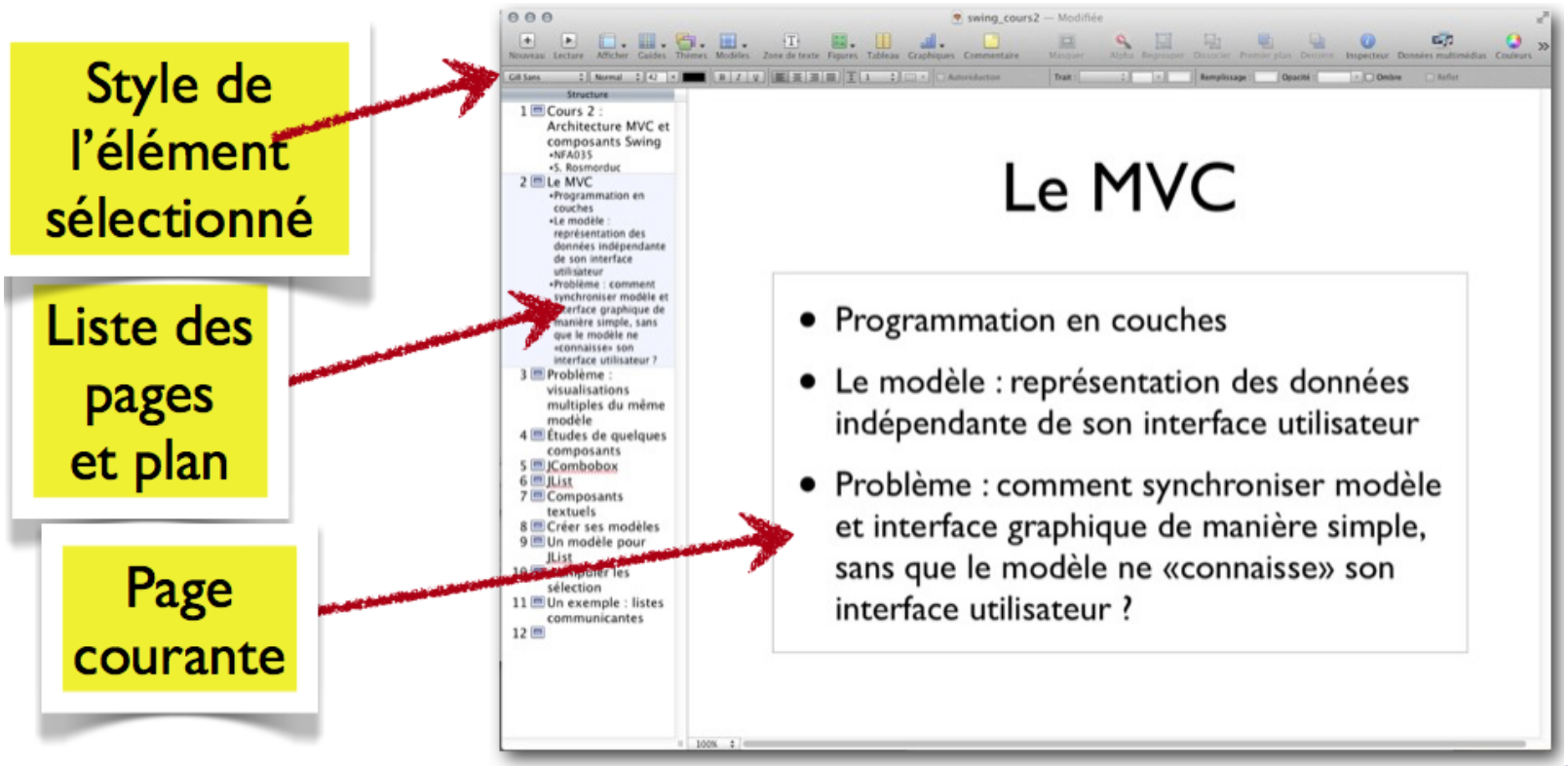
2017-2018

Observer (Observateur)

Motivation

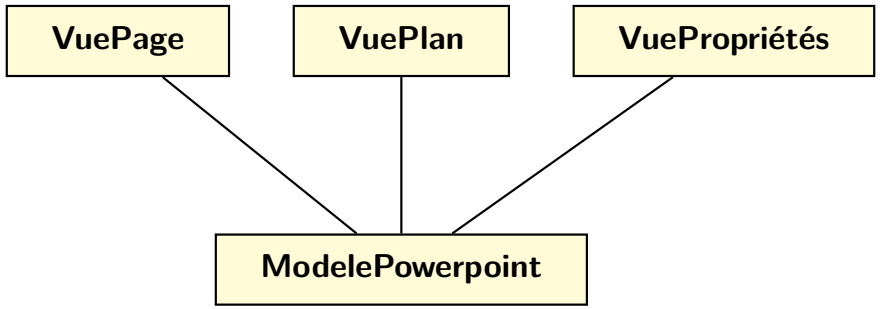
Permettre à *des* objets d'être prévenus quand un composant est modifié sans introduire de dépendances entre le composant et les objets en question.

Typiquement, lien entre un modèle et ses visualisations :



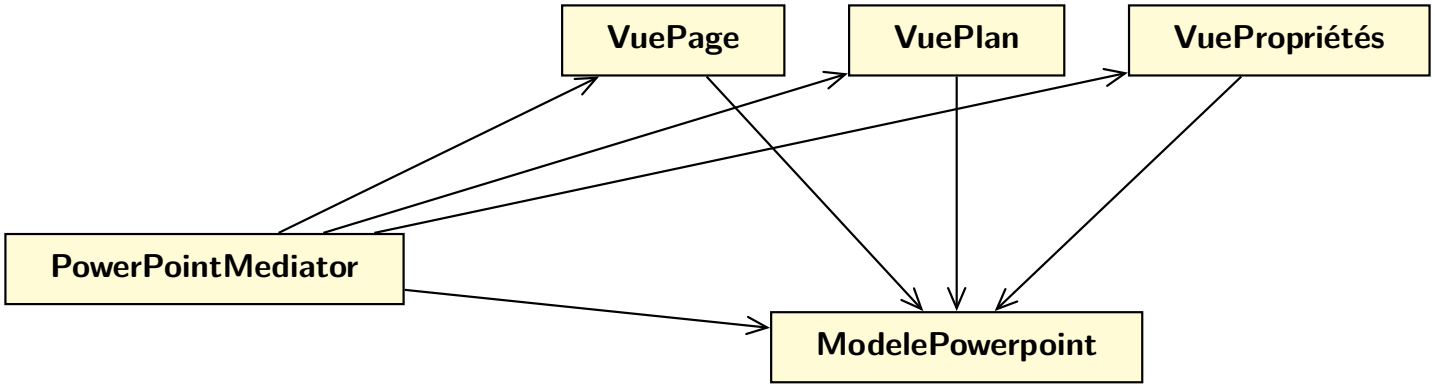
Observateur

Typiquement, on veut résoudre des cas comme :



- les vues doivent connaître le modèle pour se mettre à jour ;
- le modèle doit pouvoir contacter les vues pour leur dire de se dessiner...
- on ne souhaite pas que le modèle connaisse ses vues !

Solutions 1 ?



- Le Médiateur (oui, encore un pattern) connaît tout le monde ;
- toutes les modifications passent par lui ;
- simple, mais peu adapté quand il y a beaucoup de composants indépendants ;
- c'est architecture Modèle/Vue/Présentateur, assez indiquée pour les formulaires.

Solutions 2 ? ? ?

```
1 // NON !
2 class ModelePowerpoint {
3     VuePage vuePage;
4     VueProprietes vueProprietes;
5     VuePlan vuePlan;
6
7 }
```

- Introduit une dépendance entre la couche modèle et la couche de visualisation...
- Du coup, dépendance circulaire, couplage fort, tout ça...
- Pas gérable.

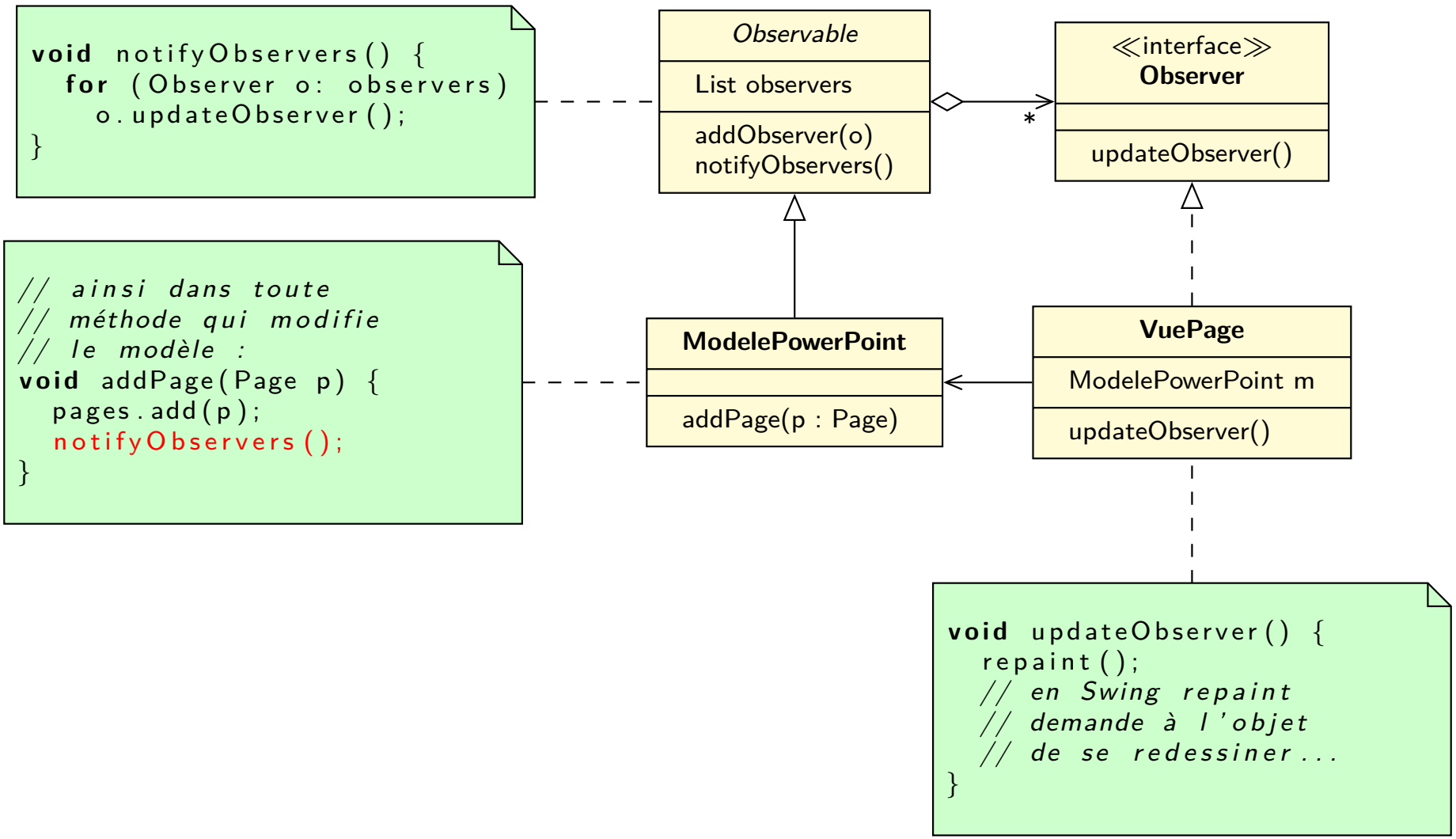
Solutions 3 ?

```
class ModelePowerpoint {
    List<TrucsAPrevenir> trucsAPrevenirs;

}
```

- On s'arrange pour que les vues implémentent la même interface...
- Du coup, on peut les mettre dans une liste ;
- Tout ce que connaît le modèle, c'est l'interface en question ;
- La couche Modele est correctement isolée de la couche Presentation ;
- l'interface contient une méthode `updateObserver` ;
- quand celle-ci est appelée, le composant graphique sait qu'il doit se redessiner.

Solutions 3


void notifyObservers() {
for (Observer o: observers)
o.updateObserver();
}
Observable
List observers
addObserver(o)
notifyObservers()
≪interface≫
Observer
updateObserver()
*
// ainsi dans toute
// méthode qui modifie
// le modèle :
void addPage(Page p) {
pages.add(p);
notifyObservers();
}
ModelePowerPoint
addPage(p : Page)
VuePage
ModelePowerPoint m
updateObserver()
void updateObserver() {
repaint();
// en Swing repaint
// demande à l'objet
// de se redessiner...
}

Utilisations du pattern Observateur

- dans les interfaces graphiques pour le M/V/C ;
- chaque fois qu'on veut « écouter » les modifications apportées à un objet (exemple : log) ;
- variante : bus d'événements (augmente le découplage) ;

Conséquences

- en swing, chaque composant graphique a son type de modèle ;
- du coup, notre classe `ModelePowerpoint` ne convient pas directement ;
- soit on l'adapte pour qu'elle exporte des modèles utilisables par swing ;
- soit on définit des `Adaptateurs` : des objets qui implémentent l'interface attendue par Swing, mais qui manipulent les données de notre modèle.
- JavaFX est plus propre à ce niveau : les observateurs y sont très génériques (interface `Property`)