EPFC

Architecture MVC - DP Observateur

La première architecture complète proposée pour remédier aux défauts de notre application est l'architecture MVC (vue également aux cours de **TGPR** et **PRWB**)

En général, cette architecture utilise quelques **Design Patterns** (***DP***)

Les DP effectivement utilisés dépendent de la complexité des différentes couches architecturales

Dans notre petite application, nous utiliserons uniquement le DP **Observateur** (***Observer***)

MVC en quelques mots (rappels)

Architecture visant à séparer les préoccupations de présentation de celles de contrôle de celles concernant l'état de l'application (le modèle)

- **M**odèle - Concerne l'état de l'application : données liées à l'application, règles métiers, etc … (*Grosso modo*, le modèle de la V1 mais que nous adapterons avec le DP Observer / Observable)

- **V**ue - Classes qui s'occupent de l'IHM (Interface Homme Machine). Généralement, elles prennent en compte les entrées / sorties.

- **C**ontrôleur - Classes qui s'occupent de "jouer" les scénarios des UC. Elles reçoivent les requêtes des utilisateurs (via les vues) et demandent au modèle d'effectuer le travail nécessaire.

Design Pattern **Observateur** : motivations

- Dans l'architecture MVC telle qu'expliquée ci-dessus, le **Contrôleur** est l'intermédiaire entre la **Vue** et le **Modèle**. Il doit explicitement appeler les méthodes de la vue lorsque le modèle a été mis à jour.

- Par ailleurs, il est fréquent que plusieurs éléments de la vue doivent être redessinés lorsqu'un élément du modèle change (surtout dans les interfaces graphiques).

- Autre cas : nous pourrions imaginer de "logguer" les différentes opérations effectuées par l'utilisateur. Le contrôleur devrait alors, en sus, envoyer des commandes à une classe qui s'occuperait de cette responsabilité.

La solution : Design Pattern ***Observateur***

- Un élément du modèle est le ***Sujet*** (observable).

- D'autres éléments de l'application (en général des éléments de la Vue), les ***Observateurs***, peuvent s'enregistrer auprès du ***Sujet***.

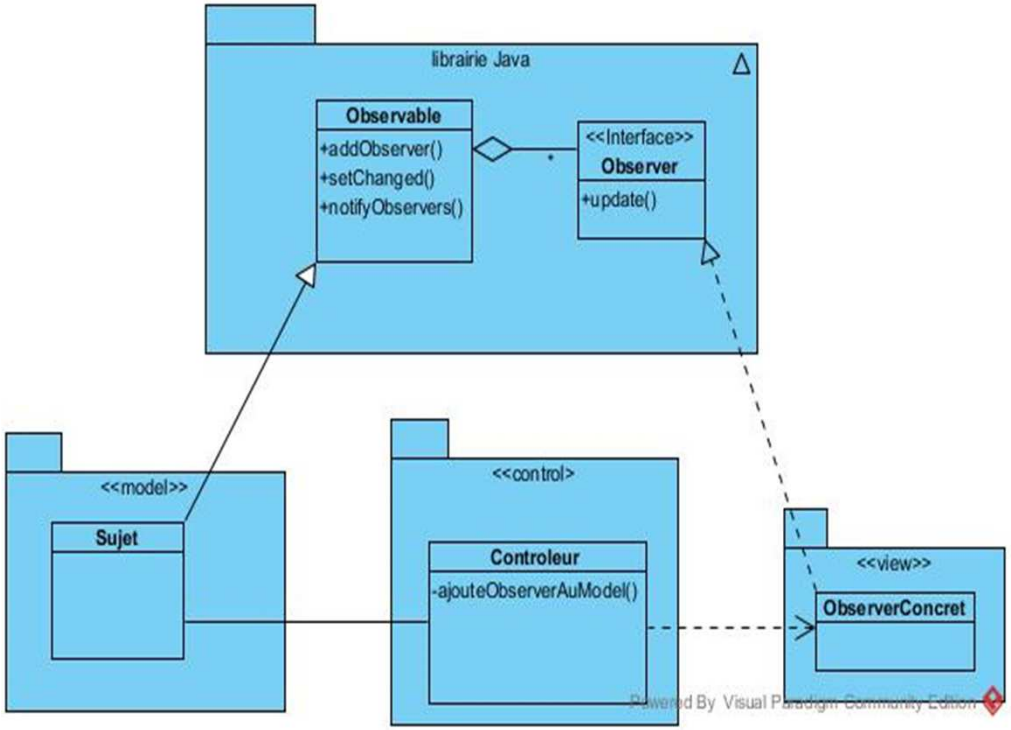
- Quand le sujet change (mise à jour de son état), il informe les observateurs qui peuvent alors se rafraîchir.

- Les ***Observateurs*** disposent d'une interface d'accès commune, ce qui fait que le Sujet ne doit pas connaître ses observateurs en détail. Ce dernier point est crucial car il permet un découplage entre le Modèle et la Vue.

Design Pattern ***Observateur*** en Java

- Le ***Sujet*** hérite de la classe `Observable`.
- Les ***Observateurs*** implémentent l'interface `Observer`.
- `Observable` contient une liste d'instances d'`Observer`(s). On ajoute un `Observer` via la méthode `addObserver` de `Observable`.
- A chaque changement (important) du sujet, on notifie les observateurs (méthode `notifyObservers` de `Observable`).
- `notifyObservers` a pour effet d'appeler la méthode `update` de l'interface `Observer`.
- Les classes qui implémentent `Observer` doivent donc définir la méthode `update` en fonction de leurs besoins.

Design Pattern **Observateur avec Java** : schéma



Voici un schéma MVC avec le pattern Observer/Observable

Controleur contient une référence au ***Sujet*** car il doit faire appel à lui pour les mises à jours et autres services, mais par contre, il n'a pas besoin de garder une référence vers la vue car il ne "parle" plus directement avec elle.

EPFC

Application démo - Implémentation (voir code)

Découpe du programme en 4 *packages* qui ont des responsabilités différentes

- package `main` : s'occupe de lancer le programme et d'instancier les objets nécessaires
- package `ctrl` : s'occupe de traiter les inputs provenant des utilisateurs
- package `model` : s'occupe de conserver les données et de faire respecter les règles métiers
- package `view` : s'occupe des affichages et de traduire et de transférer au contrôleur les inputs des utilisateurs

Démo - Package `main` - classe `Main`

- Lance le programme et créée les instances :
- du modèle : classe `Text` qui étend `Observable`
- du contrôleur : classe `Ctrl`
- de la vue : classe `View` qui implémente `Observer`

Elle ajoute ensuite l'objet de la classe `View` (comme `Observer`) à l'objet Observable de la classe `Text`.

Démo - Package `model`- classe `Text`

```
public class Text extends Observable {

          public enum TypeNotif {
        INIT, LINE_SELECTED, LINE_UNSELECTED, LINE_UPDATED, LINE_ADDED
          }

          public boolean addLine(String line) {
        ...
        notif(TypeNotif.LINE_ADDED);
        ...
          }

    public boolean updateSelectedLine(String txt) {
        ...
        notif(TypeNotif.LINE_UPDATED);
        ...
          }

    public void notif(TypeNotif typeNotif) {
        setChanged();
        notifyObservers(typeNotif);
    }

          ...

}
```

La classe `Text` étend la classe `Observable`

`enum` servant à la notification d'un changement. *L'Observer* utilisera cette information

Notifications qu'une ligne a été ajoutée ou modifiée

`notif` est la méthode qui sert à notifier. En Java, pour que la notification soit effective, il faut d'abord appeler `setChanged` La méthode `notifyObservers` fournit un paramètre (`typeNotif`) qui sera utilisé dans la méthode `update` de *l'Observer*

Démo - Package `view` - classe `View`

```
public class View extends VBox implements Observer {

    private static final int TEXTSIZE = 400, SPACING = 10;

    private final HBox editionZone = new HBox(), textZone = new HBox();
    private final TextField editLine = new TextField();
    private final ListView<String> lvLines = new ListView<>();
    private final Label lbNbLines = new Label();
    private final Button btAddLine = new Button();

    private final Ctrl ctrl;

    public View(Stage primaryStage, Ctrl ctrl) {
        this.ctrl = ctrl;
        configComponents();
        configListeners();
        Scene scene = new Scene(this, 600, 400);
        primaryStage.setTitle("Gestion de texte - MVC");
        primaryStage.setScene(scene);
    }

    ...

}
```

Constantes liées à la taille des composants et à l'espacement

La vue garde une référence vers le contrôleur.

Démo - Package `view` - classe `View`

```
public class View extends VBox implements Observer {

    ...
    private void configListenerEditLine() {
        editLine.setOnAction(e -> {
            if (editLine.isEditable()) {
                ctrl.updateSelectedLine(editLine.getText());
            }
        });
    }

    private void configListenerBtAddLine() {
        btAddLine.setOnAction((ActionEvent event) -> {
            ctrl.addLine();
        });
    }

    private void configListenerSelectionLine() {
        getListViewModel().selectedIndexProperty()
                .addListener(o -> {
                    ctrl.lineSelection(getListViewModel().getSelectedIndex());
                });
    }

   ...
}
```

Remarque : en JavaFX, pour configurer les réponses à des événements liés aux actions de l'utilisateur (click, appui sur des touches du clavier, …), il faut fournir des *lambdas* à des propriétés des composants graphiques.

La vue se contente de "traduire" les actions de l'utilisateur en appels d'opérations sur le contrôleur. On parle *d'adaptateur* (il s'agit également d'un Design Pattern)

Démo - Package `view` - classe `View`

```
private void configFocusListener() {
    lvLines.focusedProperty().addListener((observable, oldValue, newValue) -> {
        if (newValue)
            editLine.requestFocus();
    });
```

Quand le ListView reçoit le focus, il le donne à la zone de texte

```
    editLine.focusedProperty().addListener((observable, oldValue, newValue) -> {
        if (newValue)
            editLine.selectAll();
    });
}
```

Et quand la zone de texte reçoit le focus, elle sélectionne tout son contenu

Pas de changement ici par rapport à la version précédente : c'est toujours la vue qui a des préoccupations de logique applicative

Démo - Package `view` - classe `View`

```
public void update(Observable o, Object arg) {
    Text text = (Text) o;
    Text.TypeNotif typeNotif = (Text.TypeNotif) arg;
    switch (typeNotif) {
        case INIT:
            lvLines.getItems().setAll(text.getLines());
            lbNbLines.setText("Nombre de lignes : " + text.nbLines());
            setTextZoneEditable(false);
            break;
        case LINE_SELECTED:
            setTextZoneEditable(true);
            editLine.setText(text.selectedLine());
            break;
        case LINE_UNSELECTED:
            setTextZoneEditable(false);
            getListViewModel().select(-1);
            break;
        case LINE_UPDATED:
            lvLines.getItems().set(text.SelectedIndex(), text.SelectedLine());
            editLine.setText("");
            setTextZoneEditable(false);
            getListViewModel().select(-1);
            break;
        case LINE_ADDED:
            lvLines.getItems().add(text.SelectedLine());
            lbNbLines.setText("Nombre de lignes : " + text.nbLines());
            getListViewModel().select(text.SelectedIndex());
            setTextZoneEditable(true);
            break;
    }
}
```

La méthode `update` reçoit 2 paramètres : un qui correspond à l'`Observable`, l'autre à des informations fournies par celle-ci. Dans notre cas, il s'agit du type de notification.

Logique applicative : le *switch* exprime ce qu'il faut faire pour chaque type de notification

Démo - Package `ctrl` - classe `Ctrl`

```
public class Ctrl {

    private final Text text;

    public Ctrl(Text text) {
        this.text = text;
    }

    public void lineSelection(int numLine) {
        if (numLine >=0 && numLine < text.nbLines())
            text.select(numLine);
        else
            text.unselect();
    }

    public void addLine() {
        text.addLine("");
    }

    public void updateSelectedLine(String txt) {
        text.updateSelectedLine(txt);
    }
}
```

Le contrôleur transfère les appels de la vue vers au modèle.

Le contrôleur décide de quel(s) appel(s) il fait au modèle en fonction de la valeur des paramètres : préoccupation de logique applicative

EPFC

MVC - Observateur - Discussion

- Grand apport par rapport à V1 : découpage des responsabilités. Chaque (ensemble de) classe(s) a une plus grande **cohésion** : leur responsabilité est cohérente, c'est à dire liée aux mêmes genres de préoccupations

- Le **couplage** est relativement faible (les classes ne dépendent pas trop les unes des autres). Entre autre, le modèle n'a aucune connaissance des classes de vue (il parle uniquement à des *observateurs*).

- Le principal inconvénient réside dans le fait que beaucoup de la logique se retrouve toujours dans la vue (principalement dans la méthode *update*). Si on venait à modifier le comportement de notre application, on devrait donc modifier le code de la vue. Il y a donc encore deux raisons de modifier la vue