# TD N°6

**Ajouter une interface utilisateur, *Design Pattern* `Model View Controller`**

Nous allons construire une petite application avec une interface utilisateur pour faire un essai. Cet essai pourra ensuite être utilisé pour des projets plus ambitieux.

Pour utiliser `JavaFX`, il faut disposer de cette bibliothèque et l'exploiter.

1. Première méthode : à la main

   On peut télécharger (pour les portables personnels, sinon au Cremi il est déjà installé) `Javafx` ici :

   `https://gluonhq.com/products/javafx/`

   Utiliser la version 21.0.5 de Javafx

   Puis compiler un projet JavaFX avec la commande

   ```
   javac -d bin --module-path "<répertoire JavaFX>/lib"
   --add-modules=javafx.controls,javafx.fxml -sourcepath src \$(find src -name "*.java")
   ```

   Et l'exécuter avec la commande

   ```
   java -cp bin --module-path "<répertoire JavaFX>/lib
   --add-modules=javafx.controls,javafx.fxml <package>.<MainClass>
   ```

   Bien-sûr, il est plus efficace et sûr d'utiliser des outils de compilation comme Maven, Gradle, Build, Makefile, etc. Mais la ligne de commande a l'intérêt d'être toujours la source de ces applications.

2. Seconde méthode : utiliser Maven

   Merci à Tom Lacombe qui nous donne cette configuration :

   Utiliser le fichier de configuration `pom.xml` pour une installation automatiquement

   pour compiler : `mvn clean package`

   pour exécuter : `mvn exec:java`

## Projet

Il s'agit de faire une simple application "todolist" qui ressemble à ceci :

Les fonctionnalités sont les suivantes :

— Ajouter un nouveau "ticket" dans une liste *first-in first-out*. On peut taper un texte qui correspond à ce ticket.

— Traiter le premier ticket de la liste et le supprimer de la liste.

— item afficher tous les tickets dans l'ordre de leur arrivée.

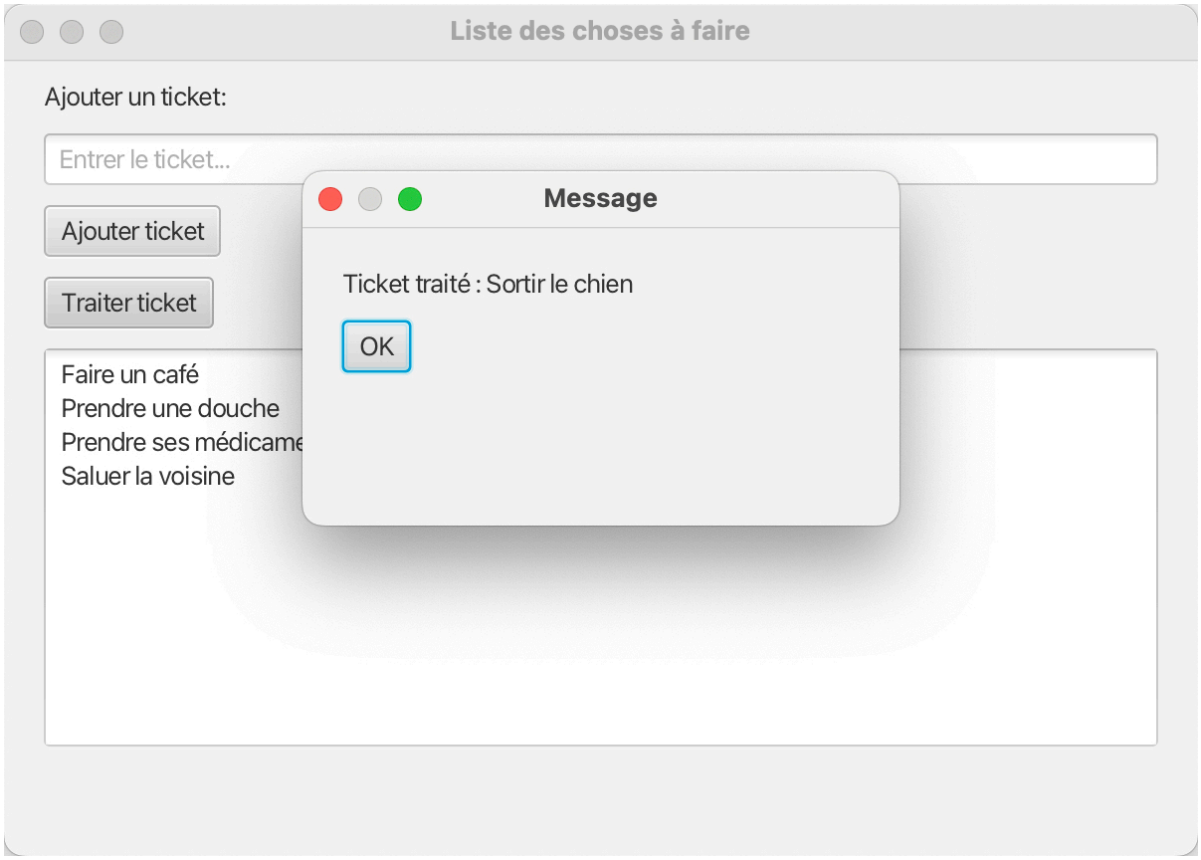
### Mise en œuvre

Le projet est construit autour d'un modèle MVC comme ceci :

Sources fournis :

Pour vous aider, nous fournissons un squellette du code avec les éléments graphiques qui tourne, mais qu'il faut compléter :

```
src/mvcExample
src/mvcExample/MainApp.java
src/mvcExample/controller
src/mvcExample/controller/Controller.java
src/mvcExample/model
src/mvcExample/model/Model.java
src/mvcExample/view
src/mvcExample/view/View.java
```




## Définition du `Controller`

Le `Controller` se charge de collecter les commandes qui correspondent aux événements des boutons.

Pour cela, nous allons écrire une interface fonctionnelle `Command` :

```
@FunctionalInterface
public interface Command {
    void execute();
}
```

et une classe `CommandManager` qui gère une liste *fifo* de commandes (attention, il ne s'agit pas encore le la liste *fifo* des tickets).

La classe `CommandManager` n'aura que deux méthodes :

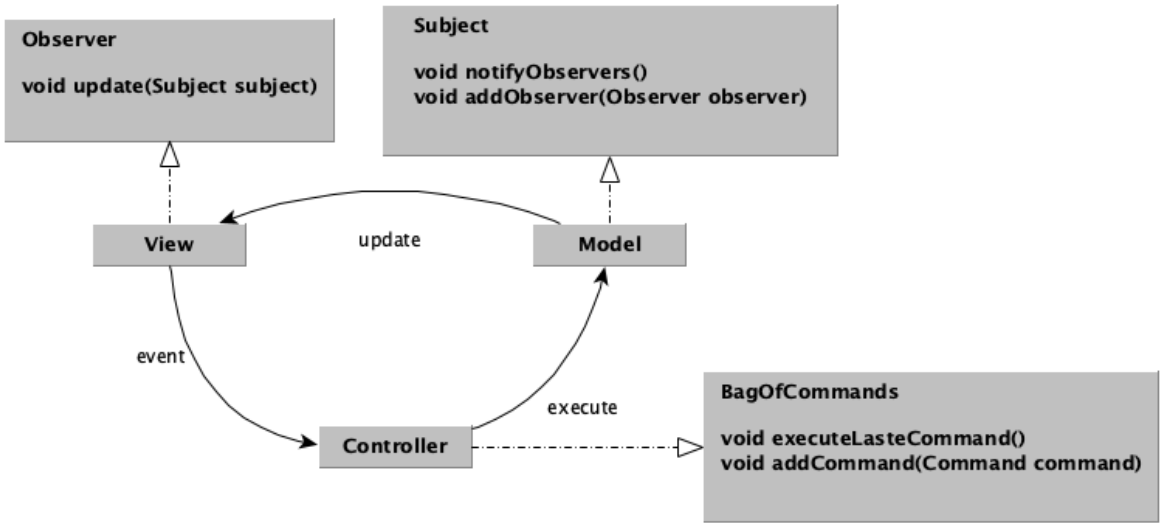
— `void addNewCommand(Command command)`

— `void executeNextCommand()`

Dans le `Controller`, assurez-vous que l'évènement `EventHandler<ActionEvent>` du bouton `addTicketButton` implémente `handle`

```
// Ajout d'une nouvelle commande
commandManager.addNewCommand(() -> {
        // Code où le modèle ajoute un nouveau ticket
        model.addTicket(ticketText);
    });
// Execution de la première commande
commandManager.executeFirstCommand();
```

Faire de même avec le bouton `processTicketButton`




## Définition du `Model`

Le modèle contient une liste *fifo* de *tickets*. Il s'agit d'implémenter cette liste.

### Définition de l'`update` de la vue

La vue est un reflet du modèle. Pour créer cela, nous allons mettre en place un mécanisme où la vue sera mise-à-jour, non pas par le `Controller`, mais par le `Model`

### Méthode

1. Créer les interfaces `Observer` et `Subject` du *Design Pattern* `Observer`
2. La vue implémente `Observer` et modèle implémente `Subject`
3. Le controller s'assure que la vue est bien un `observer` du modèle.
4. La méthode `notifyObservers()` du modèle est lancée automatiquement quand celui-ci change (on y ajoute un ticket, on traite un ticket)
5. La méthode `update()` de la vue met à jour automatiquement
   — La liste des tickets en attente
   — Le texte du ticket en cours de frappe
   — Le fait que la fenêtre *popup* soit ouverte ou non avec, le cas échéant, le texte du ticket traité

## Pour aller plus loin et traiter de façon asynchrone la liste des commandes :

Créer un `thread` dans `CommandManager` pour que les commandes soient traitées automatiquement sans demander explicitement `commandManager.executeFirstCommand()`

La principale difficulté est de traiter un nouveau *thread* qui est sans conflit avec le *thread* de `JavaFX`. Pour cela, on utilise `javafx.application.Platform.runLater()` qui permet de s'assurer qu'une tâche sera exécutée sur le `JavaFX Application Thread`, même si elle est appelée à partir d'un autre *thread*. Elle prend comme argument un objet `Runnable`, qui représente le code à exécuter. Dans notre cas, il s'agit de la $\lambda$-abstraction de la commande.

Un autre point délicat :

La $\lambda$-abstraction de la commande que l'on peut écrire $() \rightarrow command.execute()$ doit utiliser un objet command qui est immuable en Java. Cette précaution permet de s'assurer de la cohérence lors de l'exécution.

voici un exemple de code :

```
package controller;

import javafx.application.Platform;
import java.util.LinkedList;
import java.util.Queue;

public class CommandManager {
    private Queue<Command> bagOfCommands;
    private boolean isRunning;

    public CommandManager() {
        bagOfCommands = new LinkedList<>();
        isRunning = false;
        startProcessingCommands();
    }

    public synchronized void addCommand(Command command) {
        bagOfCommands.add(command);
    }

    public void startProcessingCommands() {
        if (isRunning) return;
        isRunning = true;

        // Lancer un thread pour le traitement des commandes
        new Thread(() -> {

            // Boucle infinie du thread
            while (true) {
                Command command = null;

                // Synchronisation pour accéder à la file en toute sécurité
                synchronized (this) {
                    if (!bagOfCommands.isEmpty()) {
                        command = bagOfCommands.poll();  // Récupère la
commande suivante
                    }
                }

                if (command != null) {
                    // Cette curieuse affectation car
                    // une variable utilisée dans une lambda-abstraction
                    // doit être "effectivement finale".
                    final Command finalCommand = command;

                    // Exécute la commande sur le thread JavaFX
                    Platform.runLater(() -> finalCommand.execute());
```

```
                }
                try {
                    // Pause entre deux traitements
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    break;
                }

            }
        }).start();  // Démarrer le thread
    }
}
```