ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Manuel de l'architecte

Patrons d'architecture MVC, MVP et MVVM avec implémentation en JavaFX

1. Introduction

Trois patrons d'architecture font autorité en matière de développement d'interfaces graphiques (GUI). Ce sont les patrons Modèle-Vue-Contrôleur (MVC), Modèle-Vue-Présentation (MVP) et Modèle-Vue-Vue Modèle (MVVM).

La motivation première derrière ces trois patrons est la séparation des responsabilités afin d'augmenter la cohésion et réduire le couplage. Les responsabilités dans les interfaces graphiques font référence à deux logiques :

1/ **Logique du domaine ou logique métier** : c'est la partie de l'application qui crée, stocke et modifie les données et qui leur donne du sens. Elle est spécifique du domaine d'application et indépendante des problèmes d'interaction avec l'utilisateur. On y retrouve les classes du domaine et les règles métier qui régissent ces données.

2/ **Logique de présentation** : elle désigne tout type de logique qui spécifie comment l'interface graphique réagit aux interactions avec l'utilisateur ou aux changements induits dans le modèle de domaine.

Pour illustrer la différence entre ces deux logiques, prenons le cas d'une application qui gère l'évolution d'une température ambiante. Une règle indique que si la valeur de la température dépasse un certain seuil alors elle doit être affichée en couleur rouge sinon en noir. Nous divisons cette règle en trois parties :

1/ Si la valeur dépasse un certain seuil, elle est trop élevée.

2/ Si elle est trop élevée, elle devrait être affichée de manière spéciale.

3/ Pour marquer une valeur spéciale, elle est affichée en rouge sinon en noir.

La première partie fait partie de la logique métier. Une classe du domaine représente et gère la valeur de température. La deuxième partie correspond à la logique de présentation. La présentation sait du modèle quand la valeur est trop élevée et, par conséquent, la transmet à la représentation graphique avec une information indiquant qu'elle est spéciale. C'est le travail de la troisième partie que de traduire cette information par une couleur. Cela pourrait être autre chose qu'une couleur, comme un signe d'avertissement à côté de la valeur. Tout dépend du choix du concepteur et des possibilités de la bibliothèque graphique support (GUI framework). À ce niveau, il n'y a plus de logique particulière simplement des choix d'implémentation graphiques.

Dans ce qui suit, nous détaillons les trois patrons d'architecture avec leur façon de partitionner le projet et d'y répartir les responsabilités. Nous donnons ensuite une implémentation en Java avec la bibliothèque graphique JavaFX pour chacun d'eux.

2. Description des patrons

2.1 Modèle-Vue-Contrôleur

Le patron MVC est le premier à avoir été défini en 1979 par Trygve Reenskaug pour les applications SmallTalk. Dans ce patron la répartition des responsabilités est faite entre trois parties.

- Le **modèle** contient la logique métier et l'état courant de l'interface durant le cycle de dialogue avec l'utilisateur. Il peut être aussi simple qu'une valeur entière ou aussi complexe qu'un service nécessitant plusieurs niveaux de traitement avec persistance des données.
- La **vue** porte toute la logique de présentation. Son travail consiste à afficher les données du domaine et à recevoir des interactions de l'utilisateur. La partie vue peut correspondre à un écran avec plusieurs fenêtres, une simple fenêtre ou un élément d'une fenêtre. La vue ne gère pas les interactions avec les utilisateurs mais les délègue à un autre composant : le contrôleur.
- Le **contrôleur** reçoit les interactions de l'utilisateur de la vue et les traite en modifiant le modèle. Le code à l'intérieur du contrôleur correspond essentiellement à du code de liaison (*glue code*) entre la vue et le modèle.

Une fois le modèle modifié, il est nécessaire d'afficher les données mises à jour vers l'utilisateur. Pour cela, MVC utilise le patron de conception Observateur qui agit entre la vue et le modèle (Figure 1). La vue s'inscrit auprès du modèle, et reçoit une notification à chaque modification du modèle. Notons que même si le modèle interagit avec la vue, il n'a aucune idée de son existence grâce à l'interface qui joue le rôle de pare-feu.

Ce patron fonctionne donc en cycle. L'utilisateur interagit avec les composants graphiques des vues à sa disposition ce qui se déclenche la création d'événements qui sont envoyés au contrôleur qui leur est associé. Concrètement, ce ne sont pas directement les événements qui sont envoyés au contrôleur, mais cela passe par l'appel de méthodes spécifiques du contrôleur de manière à circonscrire le code graphique à la vue. En particulier, il ne doit pas y avoir d'importation de classes de la bibliothèque graphique dans le contrôleur (ie. pas de `import javafx.*`). Le contrôleur vérifie la conformité des interactions et déduit les modifications à apporter au modèle qui les intègre selon ses règles métier. Les modifications du modèle sont ensuite signalées à toutes les vues qui se mettent à jour en conséquence.

Mais certains événements de l'utilisateur ne demandent pas de modification des données. Par exemple, l'utilisateur veut afficher l'historique des températures de l'année. Cela nécessite simplement que la vue offre à l'utilisateur un affichage de cet historique. Dans ce cas, le contrôleur ne respecte plus le processus en boucle et active la vue par des méthodes spécifiques. C'est ce que l'on appelle un court-circuit.

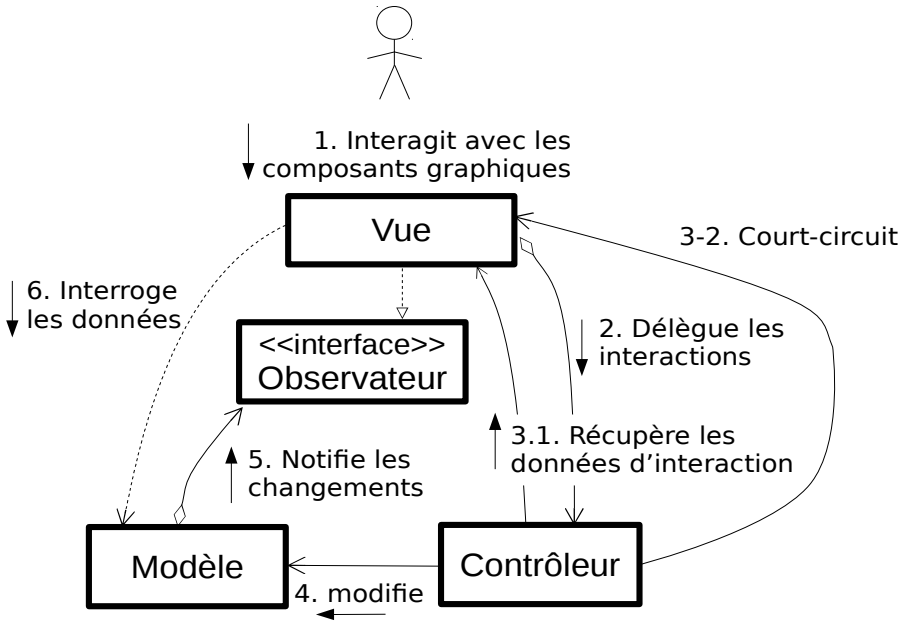

Figure 1. Diagramme de communication du patron MVC.

2.1.1 Bilan

Le grand intérêt de ce patron est sa capacité d'extension. Le patron Observateur isole les vues du modèle, il est donc facile d'ajouter une vue sans modifier l'existant. Mais, ce patron est en réalité inutilisé tel que parce qu'il présente des défauts rédhibitoires :

- Il ne remplit pas complètement l'objectif de séparation claire des responsabilités. En particulier, la logique de présentation est répartie dans la vue et dans le modèle.
  - Par exemple, c'est a priori la vue qui décide que la température doit être affichée d'une façon spéciale. C'est particulièrement néfaste. L'interface utilisateur ne devrait pas dépendre de la manière dont les données sont organisées dans le modèle du domaine.
  - Le fait que l'état courant du dialogue avec l'utilisateur soit stocké dans le modèle rend ce modèle dépendant des problèmes d'interfaçage ce qui est contraire à l'ambition initiale du découplage.
- Le défaut le plus critique de MVC est sa résistante aux tests. Puisque la vue contient des composants graphiques, elle ne peut être testée que visuellement. Donc, le code de la logique de présentation contenue dans la vue ne peut pas être testé automatiquement, cela devient du code non testable qui contient potentiellement des bugs.
- Enfin, un dernier inconvénient est la lourdeur de la boucle d'action. Même s'il est possible de mettre des courts-circuits entre la vue et le contrôle, dès que l'on a besoin des données du domaine pour mettre à jour la vue, il faut effectuer toute la boucle alors que l'on sait très bien que seule une vue est concernée.

2.2 Modèle-Vue-Présentation

Une manière d'améliorer MVC et de suppléer à ses défauts est de réduire le couplage entre la vue et le modèle. Si nous établissons une règle selon laquelle toutes les interactions entre la vue et le modèle doivent passer par le contrôleur, le contrôleur devient le lieu unique pour la mise en œuvre de la logique de présentation. La vue est débarrassée de toute logique et le modèle est débarrassée de la gestion de l'état courant du cycle de dialogue avec l'utilisateur. Le contrôleur devient alors une présentation et l'architecture reprend celle d'une architecture 3-étages (*three-tiers*).

La nouvelle structuration, données en Figure 2, présente une bonne intelligibilité des parties et une séparation claire de leur responsabilité.

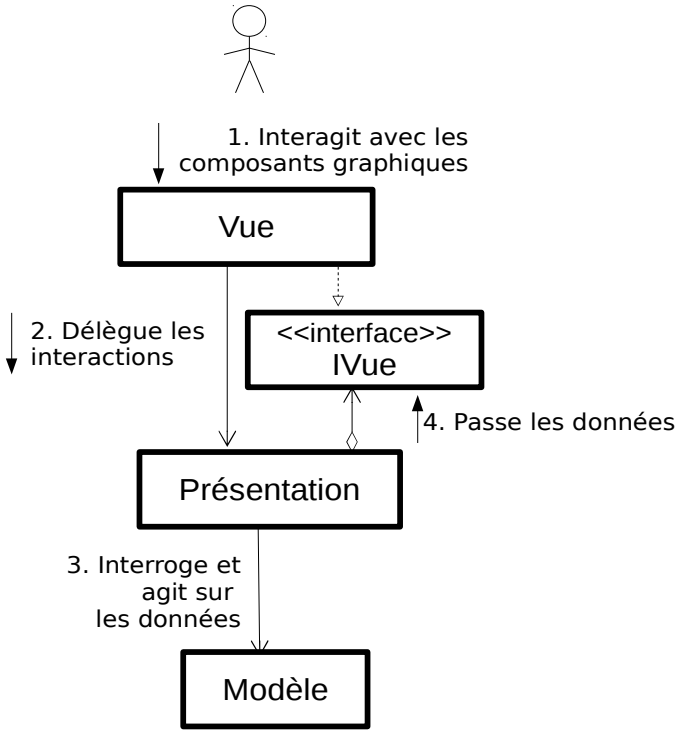

Figure 2. Diagramme de communication du patron MVP.

- Le **modèle** est uniquement centré sur la logique métier. Il est totalement déconnecté du dialogue avec l'utilisateur. Cette fois, le modèle est totalement indépendant des autres parties. Concrètement, il n'y a aucune importation de classes des autres parties dans le modèle.
- La **vue** ne concerne que la gestion graphique. Elle est totalement dépendante de la bibliothèque graphique utilisée. A contrario, elle est totalement indépendante du reste de l'application et débarrassée de toute logique.
- La **présentation** contient toute la logique de présentation ainsi que l'état courant du dialogue avec l'utilisateur. Elle sait comment réagir aux événements de l'utilisateur en modifiant les données du domaine si nécessaire puis en répercutant les effets vers la vue. Toutefois elle est indépendante de la bibliothèque graphique utilisée. Concrètement, le code n'importe aucune classe de la bibliothèque graphique. Les classes de la présentation font donc le lien

entre des classes du modèle et celles de la vue.

Contrairement à MVC, il n'y a pas de patron Observateur entre le modèle et la vue. La raison est que la vue ne contient plus de logique de présentation. C'est maintenant le travail de la présentation de savoir quand mettre à jour la vue. Les méthodes de mise à jour appartiennent à l'interface `IVue` entièrement définie par la présentation, ce qui est un excellent exemple du principe d'inversion de dépendance.

Le cycle de dialogue est donc centré sur la présentation. L'utilisateur interagit avec la vue qui fait appel à des méthodes de la présentation. En fonction de la logique de présentation, la présentation met à jour le modèle du domaine et répercute les effets sur les vues.

2.2.1 Bilan

Le patron MVP est un raffinement du patron MVC sur trois points :

- La vue ne connaît pas le modèle, il n'y a plus de couplage entre les vues et le modèle.
- La vue est débarrassée de toute logique.
- La vue est la seule à contenir du code de la bibliothèque graphique.

Les avantages de ce patron sur MVP sont indéniables :

- D'abord, la testabilité de l'application est largement améliorée. Seules les classes de la partie vue ne peuvent pas être testées unitairement. Mais cette fois, ces classes ne portent plus de logique. La vue est découplée de la présentation par l'utilisation de l'interface `IVue`. L'intérêt est aussi de pouvoir doubler la vue par des doublures « *mocks* » pour les tests unitaires de la présentation.
- Puisque toute logique de présentation est en un seul endroit, porter l'application vers une autre bibliothèque graphique est facilité. Il faut juste réimplémenter l'interface de la vue. Ceci est vrai au moins en théorie. En pratique, il serait très difficile d'implémenter exactement la même application avec Swing ou JavaFX. Il peut se poser des problèmes spécifiques de *threading*, par exemple. Qui est responsable de ce que les méthodes de la vue ne soient appelées que dans les threads appropriés ? Probablement la vue parce que la présentation n'a aucun moyen de déterminer quel thread est concerné s'il n'a aucune idée de la bibliothèque graphique utilisée. Mais cela impose une charge supplémentaire à la vue. Néanmoins, MVP est probablement aussi proche que possible de l'indépendance à la bibliothèque graphique utilisée.

Le côté négatif est que la présentation contient maintenant beaucoup de code redondant (*boilerplate code*)[1]. À chaque fois qu'un retour d'information sur la vue est nécessaire, c'est la présentation qui prend des mesures. Elle le fait en appelant des méthodes de la vue telles que `displayText1(text)` ou `setButton1Enabled(true)`

1 Le *boilerplate code* correspond à du code dupliqué à plusieurs endroits avec peu de différence. On le retrouve notamment dans les langages dit verbeux.

pour tous les textes ou boutons de l'interface. C'est du code redondant qui rend le code très verbeux et alourdie la maintenance. C'était précédemment une partie de la vue dans MVC, donc ce n'est pas pire. Mais c'est toujours une bonne idée de se débarrasser le plus possible de code redondant. C'est là que le patron MVVM prend son intérêt.

2.3 Modèle-Vue-Vue Modèle

Le patron MVVM, décrit en Figure 3, est fondamentalement le même que MVP, à l'exception d'une différence majeure. Le retour d'information n'est plus pris en charge par la présentation mais par un mécanisme de liaison de données (*data binding*). La présentation devient la vue-modèle, c'est-à-dire un modèle qui permet d'accéder aux données prêtes à l'affichage dans la vue à travers le patron Observateur, tout comme dans le MVC. Sauf que maintenant, ce patron n'est plus installé sur la vue entière mais sur chaque composant graphique de la vue. À chaque fois que la vue-modèle modifie un attribut alors le composant graphique lié dans la vue est informé et peut se mettre à jour automatiquement. Tandis que dans MVP, la présentation pousse ces données vers la vue, dans MVVM la vue tire ces données de la vue-modèle. Ce pourrait être une mauvaise chose parce que l'on pourrait croire que l'on ajoute du code avec une logique et donc qu'il faut tester alors que l'on ne peut pas le faire automatiquement, mais en réalité cette responsabilité est entièrement mise en œuvre par la bibliothèque graphique et donc n'a pas besoin d'être testée.

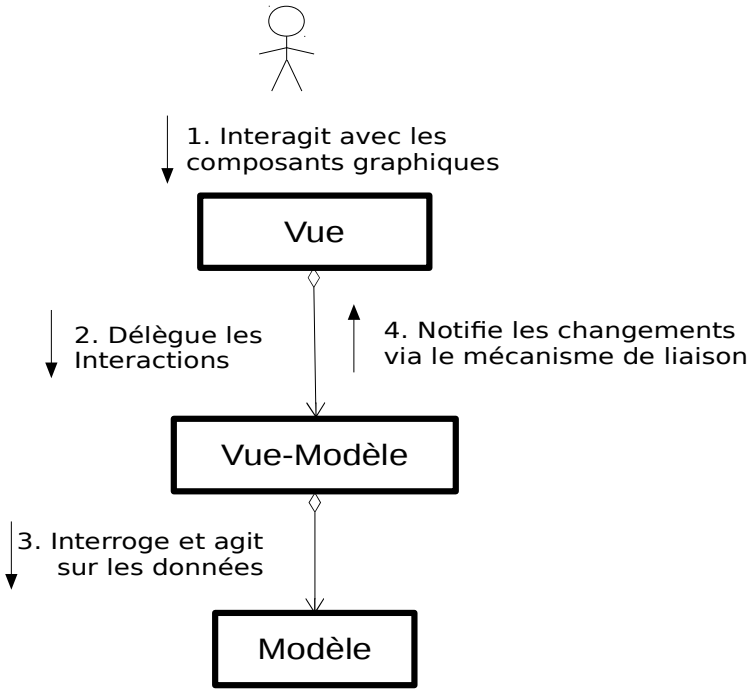

Figure 3. Diagramme de communication du patron MVVC.

2.3.1 Bilan

- Le **modèle** reste le même que pour le patron MVP. Il ne contient donc que la logique métier.

- La **vue** ne contient aucune logique. La différence avec MVP réside dans l'utilisation de liaisons entre des composants de la vue et des attributs de la vue-modèle.
- La **vue-modèle** contient la logique de présentation et l'état de l'interface comme la présentation de MVP.

Ce patron ne requière pas d'interface entre la vue et la vue-modèle puisque que la vue-modèle est déjà complètement découplée de la vue.

On retrouve bien évidemment les mêmes avantages que le patron MVP notamment pour la testabilité. La vue-modèle peut être testée facilement. L'avantage indéniable par rapport à MVP est la réduction du code redondant dans la vue-modèle.

Toutefois, la mise en œuvre des liaisons peut être complexe. Il n'est pas raisonnable d'implémenter le patron Observateur sur tous les types d'attributs de la vue. C'est donc généralement une mauvaise idée d'utiliser MVVM si la bibliothèque graphique n'intègre pas nativement ce mécanisme comme dans .NET/WPF ou JavaFX.

Même avec un mécanisme de liaison de données natif, ce n'est pas toujours possible de l'utiliser. C'est le cas par exemple si l'ordre des mises à jour est important ou si elles sont soumises à des conditions. Ces conditions ne peuvent être injectées dans le mécanisme automatique. Néanmoins, MVVM ne pose des problèmes que pour des situations non standards. Pour la plupart des cas, il diminue vraiment la quantité de code redondant.

3. Considérations supplémentaires

3.1 Modèle passif et modèle actif

Les trois patrons tels qu'ils ont été présentés jusqu'ici considèrent que le modèle ne peut être changé que par l'intermédiaire de l'interface graphique. C'est le cas le plus courant et correspond à un modèle dit *passif*. Mais, si le modèle peut être changé par d'autres moyens, les conceptions proposées ne sont plus suffisantes. Par exemple, si une application visualise la température atmosphérique en temps réel qu'elle va chercher sur un site distant et actualise régulièrement dans le modèle, dans ce cas, le modèle est dit *actif*.

Pour prendre en compte un modèle actif, il est nécessaire d'introduire à nouveau le patron de conception Observateur pour que la modification du modèle provoque la mise à jour de la vue. Pour le patron MVC, il n'y a rien à faire puisqu'il implémente déjà le patron Observateur entre le modèle et la vue. Pour les patrons MVP et MVVM, il est nécessaire de réintroduire ce patron entre le modèle et la présentation ou la vue-modèle comme décrit dans la Figure 4.

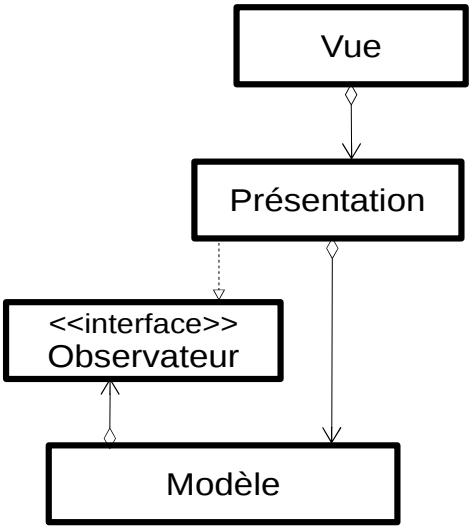

Figure 4. Diagramme de classes de la version active du modèle dans le patron d'architecture Modèle-Vue-Présentation.

3.2 Client-serveur

La description faite jusqu'à présent considère que l'application est résidente sur un seul nœud. Mais, les patrons peuvent facilement être étendus aux cas d'applications réparties sur le réseau qui partagent au moins un même modèle. Les figures 5 et 6 donnent une répartition pour le patron MVP. Le modèle est stocké du côté serveur. Les vues sont sur les clients de ce serveur. Le code des parties VC, VP ou VVM sont ensuite répartis entre le serveur et le client. Si la majorité de ce code réside sur le client, on parle de client lourd (voir la Figure 5) sinon de client léger (voir la Figure 6). On utilisera ici un patron Procuration pour rendre cette répartition transparente du côté serveur comme du côté client et donner l'impression que tout est résidant sur un seul nœud dans chaque nœud. Les problèmes de communication sont localisés entre la procuration et l'objet réel.

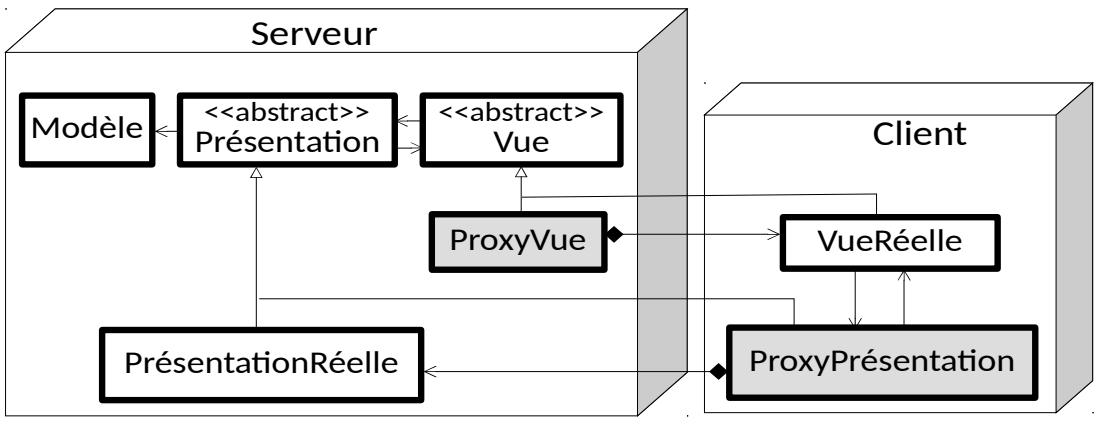

Figure 5. Diagramme de déploiement d'une architecture MVP répartie en client-serveur. C'est une répartition de type client léger puisque le client ne porte que la vue.

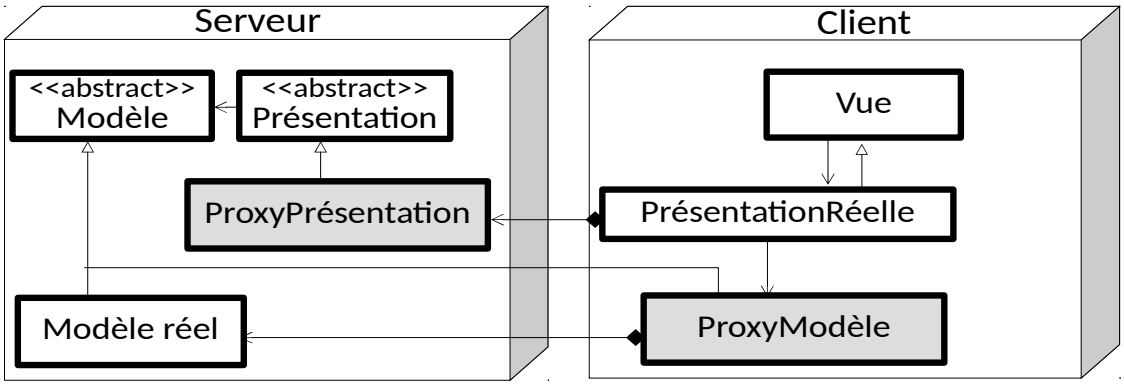

Figure 6. Diagramme de déploiement d'une architecture MVP répartie en client-serveur. C'est une répartition de type client lourd puisque la présentation est locale au client.

4. Implémentation des patrons en Java avec JavaFX

Dans cette section, nous présentons une implémentation en Java de chacun des patrons d'architecture. La bibliothèque graphique utilisée est JavaFX.

4.1 Exemple d'une application fil rouge

En guise d'exemple d'implémentation, prenons le cas d'une fenêtre de connexion telle que celle présentée en Figure 7. L'utilisateur entre son nom dans le champ texte puis presse sur le bouton « entrer ». L'application affiche en retour un message de bienvenue. Si le bouton est pressé sans qu'aucun nom n'ait été donné, le message de bienvenue s'adresse à un inconnu. Le nom de l'utilisateur est stocké indépendamment de l'interface. C'est un exemple très simple qui n'utilise qu'une seule vue et qu'une donnée qui est le nom de l'utilisateur.

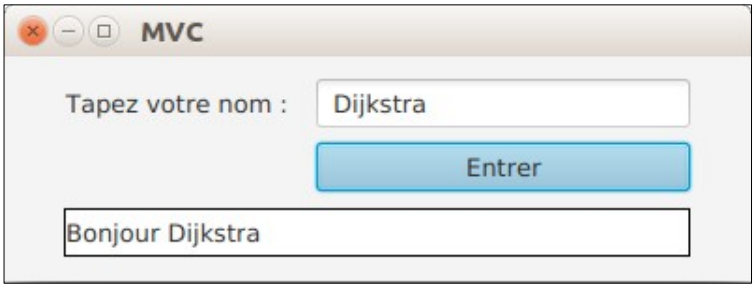

Figure 7. La fenêtre de connexion de l'application utilisée comme exemple.

4.2 Particularités de l'implémentation avec JavaFX

La particularité de JavaFX est d'utiliser un fichier XML pour décrire la composition de l'interface graphique et une classe de contrôle associée qui reçoit les événements d'interaction de l'utilisateur. Dans ce qui suit, une vue est toujours la combinaison du fichier FXML, d'un fichier CSS et de la classe de contrôle associée (Figure 8).

Ces trois fichiers sont regroupés dans le même paquet selon le principe de

fermeture commune. Le fichier FXML contient l'organisation des composants graphiques. Le fichier CSS précise le rendu visuel de ces composants graphiques. La classe de contrôle dont il est question en JavaFX n'a rien à voir avec la partie contrôleur du patron MVC. Elle est une partie de la vue et c'est pourquoi nous l'avons nommée `LoginView` dans les implémentations (bien que JavFX le nomme controller).

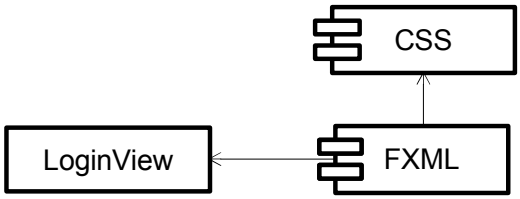

Figure 8. Organisation d'une vue avec JavaFX.

4.2.1 Le fichier FXML

Le fichier FXML suivant donne le rendu de la Figure 7. Il est partagé par toutes les implémentations des architectures MVC, MVP et MVVM.

fr.ensicaen.ecole.guipatterns.mvc.loginscreen.fxml

```
1  <AnchorPane prefHeight="117.0" prefWidth="392.0"
2          stylesheets="@loginscreen.css"
3          fx:controller="fr.ensicaen.ecole.guipatterns.mvc.LoginView">
4    <children>
5    <Label layoutX="33.0" layoutY="17.0" text="%login.message.text" />
6    <TextField fx:id="_input" layoutX="167" layoutY="12" prefWidth="200"/>
7    <Button defaultButton="true" layoutX="167.0" layoutY="46.0"
8         onAction="#handleSayHello"                                    (1)
9         prefWidth="200.0" text="%login.button.text" />
10   <Label id="console" fx:id="_message" layoutX="32.0" layoutY="81.0"
11        prefHeight="26.0" prefWidth="335.0" />
12   </children>
13 </AnchorPane>
```

1 Le nom de la méthode de la classe de contrôle qui est appelée quand l'utilisateur active le bouton.

4.2.2 Cas des vues composites

Quand une vue est composée de plusieurs parties, il faut l'organiser en plusieurs fichiers FXML. Ceci se fait directement dans le fichier FXML maître avec un lien explicite vers les fichiers FXML composants.

```
<fx:include source="fichier.fxml" />
```

À chacun de ces fichiers XML est associée une nouvelle classe de contrôle. Dans ce cas, la présentation est organisée selon le patron Médiateur, entre un contrôleur principal et des contrôleurs particuliers.

4.3 Code commun à toutes les implémentations

Les données d'entrée sont transformées par une classe de service. Cette classe appartient au modèle.

La fonctionnalité de la classe de service consiste simplement à construire le

message de bienvenue en fonction de la présence ou non d'un nom dans le champ texte.

fr.ensicaen.ecole.guipatterns.mvc.LoginService.java

```
1  public class LoginService {
2     public String sayHello( String name ) {
3        if (name.isEmpty()) {
4           return "Bonjour inconnu";
5        } else {
6           return "Bonjour " + name;
7        }
8     }
9  }
```

4.4 Implémentation du patron Modèle-Vue-Contrôleur

L'implémentation suit le diagramme de classes donné en Figure 9.

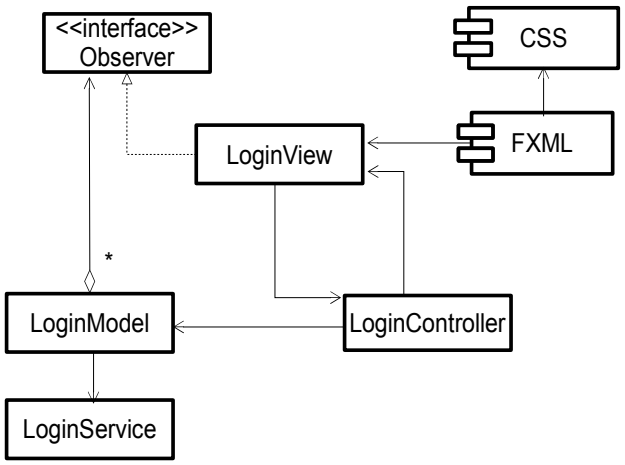

Figure 9. Le patron d'architecture MVC implémenté avec JavaFX.

4.4.1 L'interface Observer

L'interface `Observer` correspond strictement au patron de conception Observateur avec notamment la méthode `update()`. Elle permet de découpler la vue du modèle. Il s'agit ici de la version tirée (*pull*) du patron. La méthode `update()` prend en paramètre une référence sur le modèle.

fr.ensicaen.ecole.guipatterns.mvc.Observer.java

```
1  public interface Observer {
2       public void update( LoginModel model );
3  }
```

4.4.2 La partie Vue

La vue implémente l'interface et utilise les accesseurs de la classe modèle (i.e la méthode `getMessage()`) pour récupérer les données.

fr.ensicaen.ecole.guipatterns.mvc.LoginView.java

```
1  public class LoginView implements Observer {
2     private LoginController _controller;
3     @FXML private TextField _input;
```

```
4     @FXML private Label _output;
5
6     public void setController( LoginController controller ) {          (1)
7        _controller = controller;
8     }
9
10    public String getInput() {                                         (2)
11       return _input.getText();
12    }
13
14    @Override
15    public void update( LoginModel model ) {                           (3)
16       _output.setText(model.getMessage());
17    }
18
19    @FXML private void handleSayHello( ActionEvent event ) {           (4)
20       _controller.sayHello();
21    }
22 }
```

1 Connexion à la partie contrôleur.

2 Publication de la valeur d'entrée.

3 Méthode de mise à jour de la vue faisant partie du patron Observateur.

4 Méthode *handler* invoquée à chaque activation du bouton « Entrer ».

Remarque : on ne passe pas directement la méthode *handler* (ie. `handleSayHello()`) associée à l'événement d'activation du bouton « Entrer » directement au contrôleur comme préconisé par JavaFX, mais on appelle une méthode du contrôleur (ie. `sayHello()`) n'utilisant pas de code graphique dans le corps de la méthode *handler*. Le but est de ne pas avoir d'importation de classe de la bibliothèque graphique dans le contrôleur. Pour la même raison, on ne passe pas l'événement comme paramètre de la méthode du contrôleur.

4.4.3 La partie Modèle

Le modèle est composé de la classe de service `LoginService` présentée plus tôt et de la classe `LoginModel`, qui est ici réduite au stockage du nom de connexion renseigné par l'utilisateur.

fr.ensicaen.ecole.guipatterns.mvc.LoginModel.java

```
1  public class LoginModel {
2     private String _name;
3     private boolean _changed;
4     private List<Observer> _observers = new ArrayList<>();
5     private final LoginService _loginService;
6
7     public LoginModel() {
8        this(new LoginService());
9     }
10
11    public LoginModel( LoginService loginService ) {
12       _loginService = loginService;
13    }
14
15    public void setMessage( String name ) {                            (1)
```

```
16      _name = name;
17      setChanged();
18      notifyObservers();
19    }
20
21    public String getMessage() {
22      return _loginService.sayHello(_name);
23    }
24
25    public void addObserver( Observer observer ) {                    (2)
26      _observers.add(observer);
27    }
28
29    private void setChanged() {
30      _changed = true;
31    }
32
33    private void clearChanged() {
34      _changed = false;
35    }
36
37    private void notifyObservers() {
38      if (!_changed) {
39        return;
40      }
41      clearChanged();
42      observers.forEach(update(this));
43    }
44  }
```

1 Notification d'un changement aux vues. C'est l'implémentation du patron Observateur.

2 Ajout d'une vue au modèle.

4.4.4 La partie Contrôleur

Le contrôleur réagit aux événements de l'utilisateur. La méthode `sayHello()` appelée à l'activation du bouton lit le nom de connexion renseigné par l'utilisateur dans la vue et l'envoie au modèle.

fr.ensicaen.ecole.guipatterns.mvc.LoginController.java

```
1   public final class LoginController {
2     private LoginModel _model;
3     private LoginView _view;
4
5     public void setModel( LoginModel model ) {
6       _model = model;
7     }
8
9     public void setView( LoginView view ) {
10      _view = view;
11    }
12
13    public void sayHello() {                                          (1)
14      String name = _view.getInput();                                 (2)
15      _model.setName(name);                                           (3)
16    }
17  }
```

1 Méthode associée à l'événement d'activation du bouton de l'interface.

2 Lecture de la donnée d'entrée de l'utilisateur.

3 Mise à jour du modèle avec le nom de l'utilisateur.

4.4.5 La classe d'application

Cette classe fait l'assemblage dynamique des parties de l'architecture et lance l'application.

fr.ensicaen.ecole.guipatterns.mvc.LoginMain.java

```
1  public final class LoginMain extends Application {
2    public static void main( String[] args ) {
3      Locale.setDefault(new Locale("fr", "FR"));
4      launch(args);
5    }
6
7    @Override
8    public void start( final Stage primaryStage ) throws Exception {
9      primaryStage.setTitle("MVC");
10     FXMLLoader loader =  new FXMLLoader(getClass().
                                getResource("loginscreen.fxml"));        (1)
11     Parent root = loader.load();
12     LoginView view = loader.getController();
13     LoginModel model = new LoginModel();
14     model.addView(view);                                              (2)
15     LoginController controller = new LoginController();
16     controller.setView(view);                                         (2)
17     controller.setModel(model);                                       (2)
18     view.setController(controller);                                   (2)
19     Scene scene = new Scene(root, 400, 120);
20     primaryStage.setScene(scene);
21     primaryStage.show();
22   }
23 }
```

1 Lecture du fichier FXML et construction de la vue.

2 Installation des liaisons entre les composants.

4.4.6 Implémentation des tests

Ce patron n'est pas complètement testable. La vue étant essentiellement graphique, la logique contenue dans la classe `LoginView` ne peut pas être testée. Par contre, la classe `LoginView` est doublée (*mock*) pour tester les autres composants. Elle ne peut donc pas être déclarée final.

La classe `LoginModel` est doublée dans les tests unitaires du contrôleur et est espionnée dans les tests d'intégration. Elle ne peut pas être déclarée `final`. Elle présente deux constructeurs. Un premier constructeur est utilisé pour les tests. Il prend en paramètre une classe de service, ce qui permet d'utiliser une doublure lors des tests. Le constructeur par défaut crée une instance de la classe de service réelle pour un fonctionnement en mode application.

La classe `LoginControleur` est testée mais n'a pas besoin d'être doublée pour tester les autres composants. Elle peut donc être déclarée `final` si elle n'est pas espionnée dans les tests unitaires ou d'intégration.

4.5 Implémentation du patron Modèle-Vue-Présentation

L'implémentation est donnée dans le diagramme de classes Figure 10.

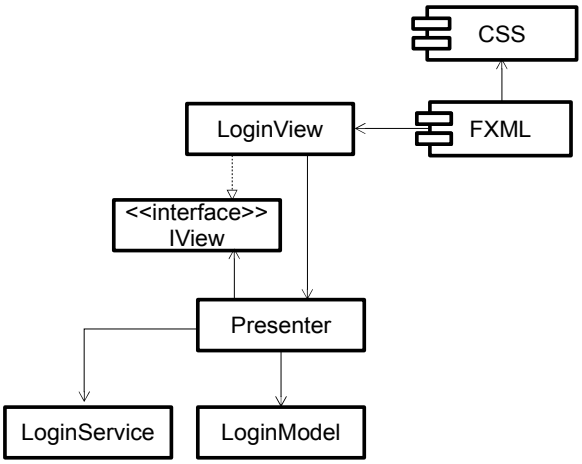

Figure 10. Le patron d'architecture MVP implémenté avec JavaFX.

4.5.1 La partie Vue

Dans cette implémentation, l'interface `IView` permet de découpler la partie vue de la partie présentation.

fr.ensicaen.ecole.guipatterns.mvp.IView.java

```
1  public interface IView {
2     public String getInput();
3     public void setPresenter( LoginPresenter presenter );
4     public void setMessage( String text );
5  }
```

La vue dans ce patron est passive. Son implémentation est exactement la même que pour le patron MVC.

fr.ensicaen.ecole.guipatterns.mvp.LoginView.java

```
1  public final class LoginView implements IView {
2     private LoginPresenter _presenter;
3     @FXML private TextField _input;
4     @FXML private Label _output;
5
6     @Override
7     public void setPresenter( LoginPresenter presenter ) {                (1)
8        _presenter = presenter;
9     }
10
11    @Override
12    public String getInput() {                                            (2)
13       return _input.getText();
14    }
15
16    @Override
17    public void setMessage( String text ) {                               (3)
18       _output.setText(text);
19    }
20
21    @FXML private void handleSayHello( ActionEvent event ) {              (4)
22       _presenter.sayHello();
```

```
23      }
24    }
```

1 Connexion à la présentation.

2 Publication de la valeur d'entrée.

3 Méthode faisant la mise à jour du message de bienvenue.

4 Méthode *handler* invoquée à chaque activation du bouton « Entrer ».

On notera que comme pour le patron MVC et pour éviter d'avoir des liens avec la bibliothèque graphique utilisée, on ne passe pas directement la méthode *handler* à la présentation. Elle reste dans la vue qui appelle une méthode appropriée de la présentation.

4.5.2 La partie Modèle

Pour s'assurer de l'indépendance de cette partie, il ne doit y avoir aucun import de classes des deux autres parties dans les classes de cette partie. L'implémentation du modèle ne présente aucune particularité par rapport au patron de base. Dans notre exemple, le modèle est restreint à une donnée de stockage.

fr.ensicaen.ecole.guipatterns.mvp.LoginModel.java

```
1   public class LoginModel {
2     private String _name;
3
4     public void setName( String name ) {
5       _name = name;
6     }
7
8     public String getName() {
9       return _name;
10    }
11  }
```

4.5.3 La partie Présentation

Il faut garantir que cette partie soit totalement indépendante de la bibliothèque graphique utilisée. En particulier, il ne devrait pas y avoir d'import de classes de la bibliothèque graphique dans les classes de cette partie (ie, pas de `import javafx.*`).

Comme la présentation ne se contente pas de répondre à des sollicitations de la vue mais prend également en charge son pilotage, la dépendance est bidirectionnelle entre les deux parties. Afin de minimiser cette dépendance, la présentation communique avec la vue au travers de l'interface `IView`.

fr.ensicaen.ecole.guipatterns.mvp.LoginPresenter.java

```
1   public final class LoginPresenter {
2     private LoginModel _model;
3     private IView _view;
4     private final LoginService _loginService;
5
6     public LoginPresenter() {
7       this(new LoginService());
8     }
9
```

```
10    public LoginPresenter( LoginService loginService ) {
11       _loginService = loginService;
12    }
13
14    public void setModel( LoginModel model ) {
15       _model = model;
16    }
17
18    public void setView( IView view ) {
19       _view = view;
20    }
21
22    public void sayHello() {
23       String input = _view.getInput();                          (1)
24       _model.setMessage(input);                                 (2)
25       String message = _loginService.sayHello(input);           (3)
26       _view.setMessage(message);                                (4)
27    }
28 }
```

1 Lecture de la donnée d'entrée de l'utilisateur.

2 Changement du modèle avec le nom de l'utilisateur.

3 Construction du message de bienvenue en utilisant la classe de service.

4 Changement du message sur l'interface graphique.

4.5.4 La classe d'application

La liaison entre les trois parties est faite dans la classe de lancement de l'application.

fr.ensicaen.ecole.guipatterns.mvp.LoginMain.java

```
1  public final class LoginMain extends Application {
2     public static void main( String[] args ) {
3        launch(args);
4     }
5
6     @Override
7     public void start( final Stage primaryStage ) throws Exception{
8        primaryStage.setTitle("MVP");
9        FXMLLoader loader = new FXMLLoader(getClass().
                                  getResource("loginscreen.fxml"));   (1)
10       Parent root = loader.load();
11       IView view = loader.getController();
12       LoginPresenter presenter = new LoginPresenter();
13       view.setPresenter(presenter);                               (2)
14       presenter.setView(view);                                    (2)
15       LoginModel model = new LoginModel();
16       presenter.setModel(model);                                  (2)
17       Scene scene = new Scene(root, 400, 120);
18       primaryStage.setScene(scene);
19       primaryStage.show();
20    }
21 }
```

1 Lecture du fichier FXML et construction de la vue.

2 Installation des liaisons entre les composants.

4.5.5 Implémentation des tests

Le patron MVP permet de maximiser l'utilisation des tests unitaires. Seule la partie vue n'est pas testable unitairement, mais dans ce patron, elle n'a plus aucune logique. Toute la logique de présentation est maintenant localisée dans la présentation.

Seule la classe `LoginView` peut être déclarée `final`. La classe `LoginModel` est doublée dans les tests unitaires de la classe `LoginPresenter` et espionnée dans les tests d'intégration. La classe `LoginPresenter` est elle-même espionnée dans les tests d'intégration.

4.6 Implémentation du patron Modèle-Vue-Vue Modèle

Ce patron est très proche du patron MVP comme le montre le diagramme de classes de la Figure 11. La différence est non visible. Elle consiste en l'utilisation du mécanisme de liaison de JavaFX (basé sur les *JavaBeans*) pour lier des composants de la vue à des propriétés de la vue-modèle. La vue-modèle remplace alors la présentation. Il n'a plus de lien explicite entre la vue et la vue-modèle et donc plus besoin d'interface entre eux. La vue-modèle ignore totalement la vue.

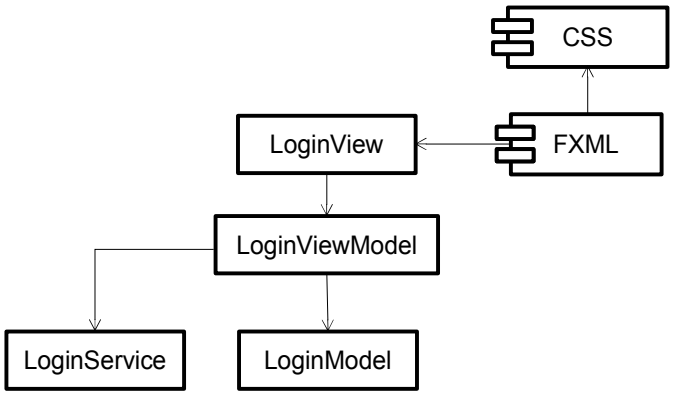

Figure 11. Le patron d'architecture MVVM implémenté avec JavaFX.

4.6.1 La partie Vue

La vue est encore passive. Le mécanisme de liaison est représenté en JavaFX par les propriétés. Les propriétés JavaFX font la synchronisation entre des composants graphiques de la vue et les données de la vue-modèle.

fr.ensicaen.ecole.guipatterns.mvvm.LoginView.java

```
1   public final class LoginView {
2      private LoginViewModel _viewModel;
3      @FXML private TextField _input;
4      @FXML private Label _output;
5
6      public void setViewModel( LoginViewModel viewModel ) {
7         _viewModel = viewModel;
8         _input.textProperty().bindBidirectional(_viewModel.inputProperty());   (1)
9         _output.textProperty().bindBidirectional(_viewModel.messageProperty()); (1)
10     }
11
12     @FXML private void handleSayHello( ActionEvent event ) {
13        _viewModel.sayHello();                                                  (2)
```

```
14      }
15    }
```

1 Les propriétés de la vue-modèle sont synchronisées avec les composants de la vue.

2 Méthode *handler* invoquée à chaque activation du bouton « Entrer ».

Comme précédemment, on ne passe pas directement la méthode *handler* à la vue-modèle de manière à ne pas intégrer de code JavaFX dans la vue-modèle.

4.6.2 La partie Modèle

Le modèle ne fait que stocker le nom de l'utilisateur.

fr.ensicaen.ecole.guipatterns.mvvm.LoginModel.java

```
1   public class LoginModel {
2     private String _name;
3
4     public void setName( String name ) {
5       _name = name;
6     }
7
8     public String getName() {
9       return _name;
10    }
11  }
```

4.6.3 La partie Vue-Modèle

La vue-modèle contient la logique métier. Elle ne met pas à jour les données de la vue puisque le lien est fait automatiquement par la liaison des propriétés.

fr.ensicaen.ecole.guipatterns.mvvm.LoginViewModel.java

```
1   public final class LoginViewModel {
2     private StringProperty _input;                                    (1)
3     private StringProperty _message;                                  (1)
4     private final LoginService _loginService;
5     private LoginModel _model;
6
7     public LoginViewModel() {
8       this(new LoginService());
9     }
10
11    public LoginViewModel( LoginService loginService ) {
12      _loginService = loginService;
13    }
14
15    public void setModel( LoginModel model ) {
16      _model = model;
17    }
18
19    public StringProperty inputProperty() {
20      if (_input == null) {
21        _input = new SimpleStringProperty(this, "input: ", "");
22      }
23      return _input;
24    }
25
26    private String getInput() {
```

```
27      return _input == null ? null : inputProperty().get();
28    }
29
30    public StringProperty messageProperty() {
31      if (_message == null) {
32        _message = new SimpleStringProperty(this, "message");
33      }
34      return _message;
35    }
36
37    private void setMessage( String output ) {
38      messageProperty().set(output);
39    }
40
41    public void sayHello() {
42      String input = getInput();                                    (2)
43      String message = _loginService.sayHello(input);               (3)
44      _model.setMessage(input);                                     (4)
45      setMessage(message);                                          (5)
46    }
47  }
```

1 Les attributs *input* et *message* sont déclarés en tant que propriétés JavaFX.

2 Lecture de la donnée d'entrée de l'utilisateur.

3 Construction du message de bienvenue en utilisant la classe de service.

4 Changement du modèle avec le nom de l'utilisateur.

5 Changement de la valeur de la propriété avec le nouveau message.

4.6.4 La classe d'application

La classe d'application fait le lien entre les parties du patron.

fr.ensicaen.ecole.guipatterns.mvvm.LoginMain.java

```
1   public final class LoginMain extends Application {
2     public static void main( String[] args ) {
3       launch(args);
4     }
5
6     @Override
7     public void start( final Stage primaryStage ) throws Exception {
8       primaryStage.setTitle("MVVC");
9       FXMLLoader loader = new FXMLLoader(getClass().
                                    getResource("loginscreen.fxml"));   (1)
10      Parent root = loader.load();
11      LoginView view = loader.getController();
12      LoginViewModel viewModel = new LoginViewModel();
13      view.setViewModel(viewModel);                                   (2)
14      LoginModel model = new LoginModel();
15      viewModel.setModel(model);                                      (2)
16      Scene scene = new Scene(root, 400, 120);
17      primaryStage.setScene(scene);
18      primaryStage.show();
19    }
20  }
```

1 Lecture du fichier FXML et construction de la vue.

2 Installation des liaisons entre les composants.

4.6.5 Implémentation des tests

Les tests sont ici facilités par le fait que les dépendances sont unidirectionnelles. La partie vue n'a pas besoin d'être testée puisqu'elle ne contient pas de logique. Elle peut donc être déclarée `final`.

La partie vue-modèle ne dépend que du modèle qui doit être doublé pour les tests unitaires. Le modèle ne peut donc pas être `final`. La vue-modèle peut être déclarée `final` si elle n'est pas espionnée dans les tests d'intégration.

Comme pour MVP, la partie modèle ne dépend d'aucune autre partie et peut être testée en isolation.

5. Références

Sergei Tachenov, « MVC, MVP and MVVM, pt. 1: The Ideas », *http://www.tachenov.name/2016/09/30/208/*. Consulté le 26 juillet 2017.

Andres Almiray, « MVC Patterns », *http://aalmiray.github.io/griffon-patterns/*. Consulté le 26 juillet 2017.