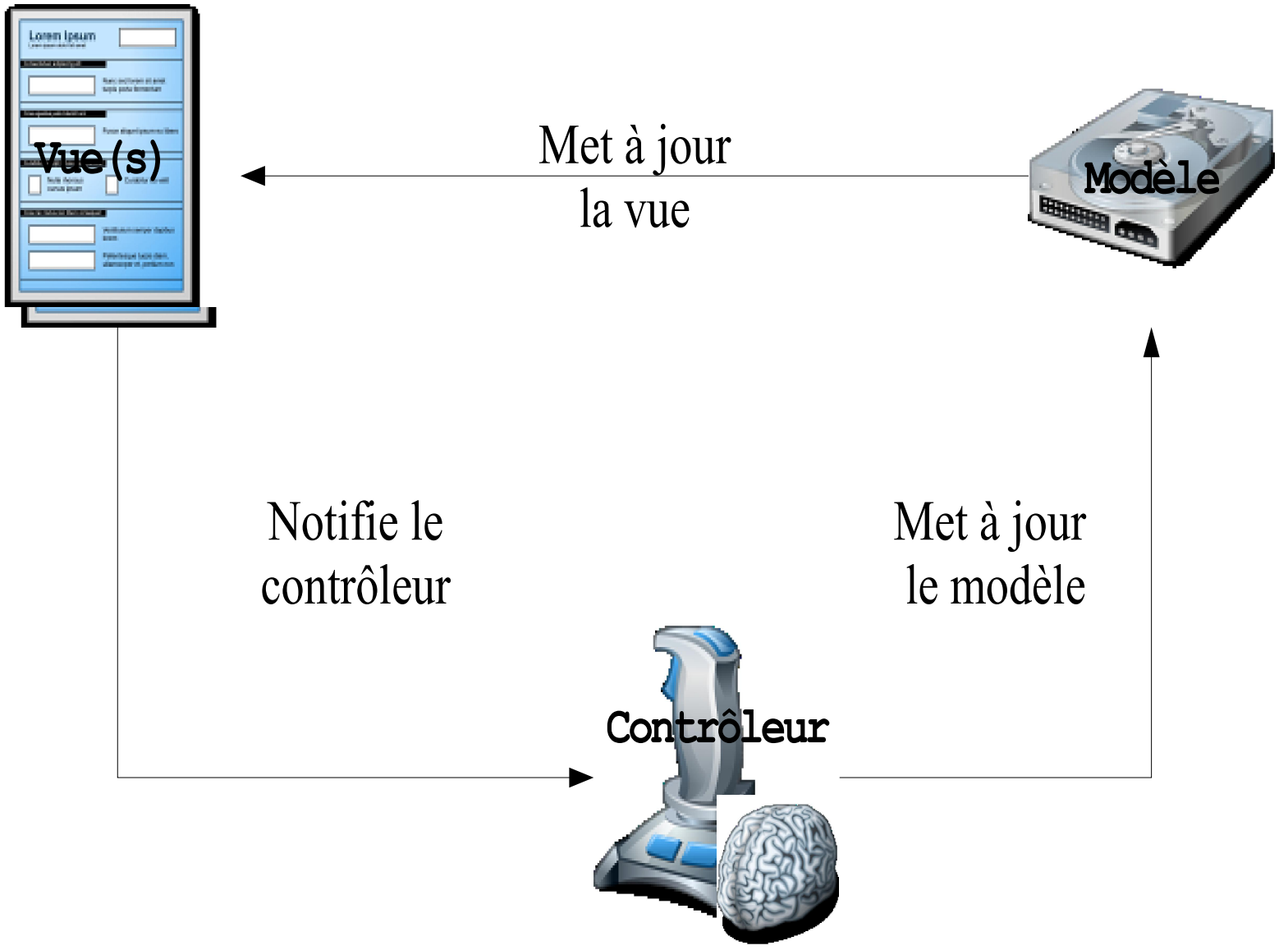
Architectures avancées

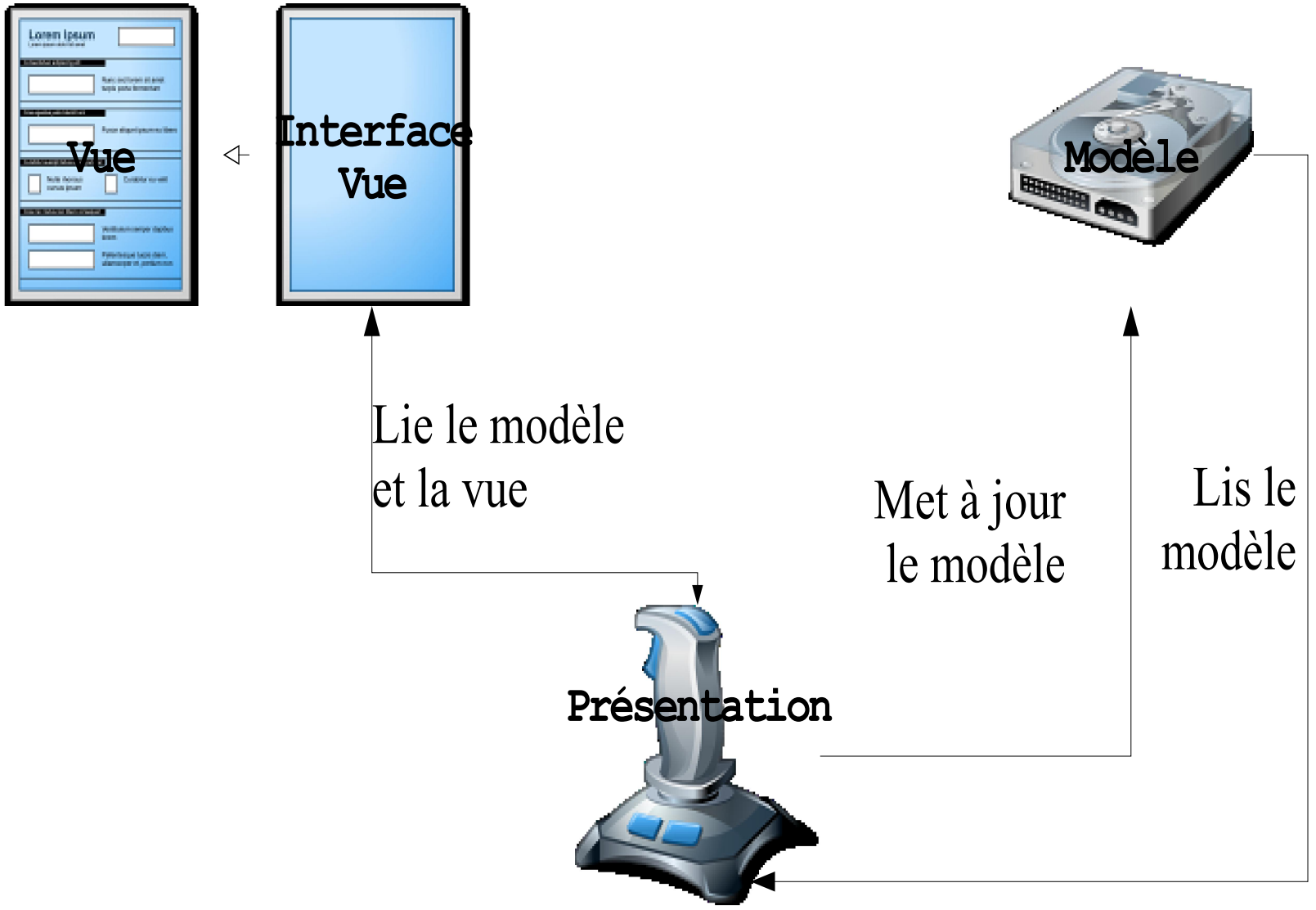
Architecture MVC

Découpage Model View Controller



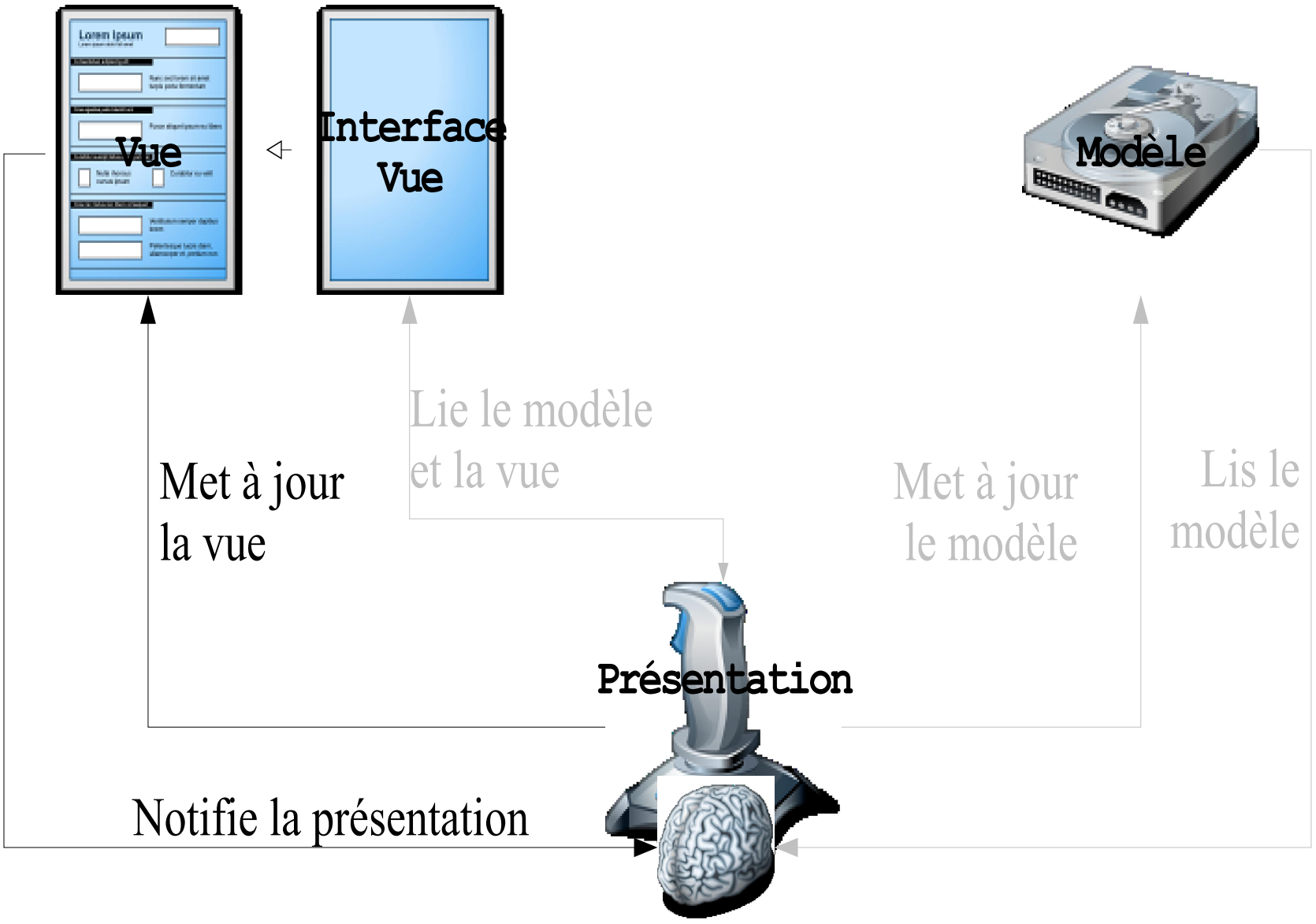
ARCHITECTURE MVP

Découpage Model View Presenter



ARCHITECTURE MVP

Découpage Model View Presenter



MVP vs MVC

Des différences nettes

- MVC
  - Le contrôleur décrit un comportement, peut être partagé par plusieurs vues
  - Le contrôleur choisit la vue à afficher
  - Le modèle utilisé peut être fortement couplé à la vue, notamment par databinding
- MVP
  - Souvent une Présentation par Vue (des vues complexes peuvent avoir plusieurs Présentations), une implémentation classique en « Presentation Model » impose 1 View = 1 Presenter
  - Le Presenter se charge du binding au modèle, la Vue en est plus fortement découplée (elle n'a pas de connaissance du modèle)
  - Le Presenter est fortement couplé à l'interface de la Vue : plus facilement testable

FXML + Controller

Fonctionnement classique

- La notion de Controller FXML possède quelques limitations
  - Déclaration explicite dans le fichier FXML
  - L'utilisation d'interface est non gérée (le FXML impose que le Controller soit instanciable via un constructeur par défaut)
- Couplage fort entre le FXML (la Vue) et son Controller
  - Le FXML connait la classe de son Controller
  - Le Controller expose des champs au FXML via annotations @FXML, voir contient des références à des composants
- Couplage fort possible entre le FXML et son modèle
  - Il suffit d'utiliser des POJO avec des champs de type Property
  - Binding utile mais couplage du modèle à JavaFX

FXML + Controller

Autre alternative de déclaration

- Utilisation personnalisée de l'API FXMLLoader
  - Possibilité de définir un Controller Factory
  - Possibilité de définir un Builder Factory (pour les composants)
- Création de FXML sans notion de Controller
  - Celui-ci sera défini au moment du chargement avec FXMLLoader
  - Plusieurs contrôleurs possibles pour un même FXML
  - Possibilité de changer l'ordre d'instanciation des composants et contrôleurs
  - Possibilité d'utiliser un framework d'IOC

Inversion Of Control

Injecter des dépendances

- Principe d'Hollywood « Don't call us, we'll call you »
- Principe de l'injection
  - Déclarer des objets managés par un framework IOC comme Spring
  - Déclarer les dépendances des objets entre eux « de l'extérieur », chaque objet ne connaît pas explicitement l'autre

```
<beans>
  <bean id="personne"
     class="com.zenika.domain.PersonneImpl">
    <property name="adresse" ref="adresse" />
    <property name="profession" ref="profession"/>
  </bean>
  <bean id="adresse"
    class="com.zenika.domain.AdresseImpl">
  </bean>
  <bean id="profession"
    class="com.zenika.domain.ProfessionImpl">
  </bean>
</beans>
```

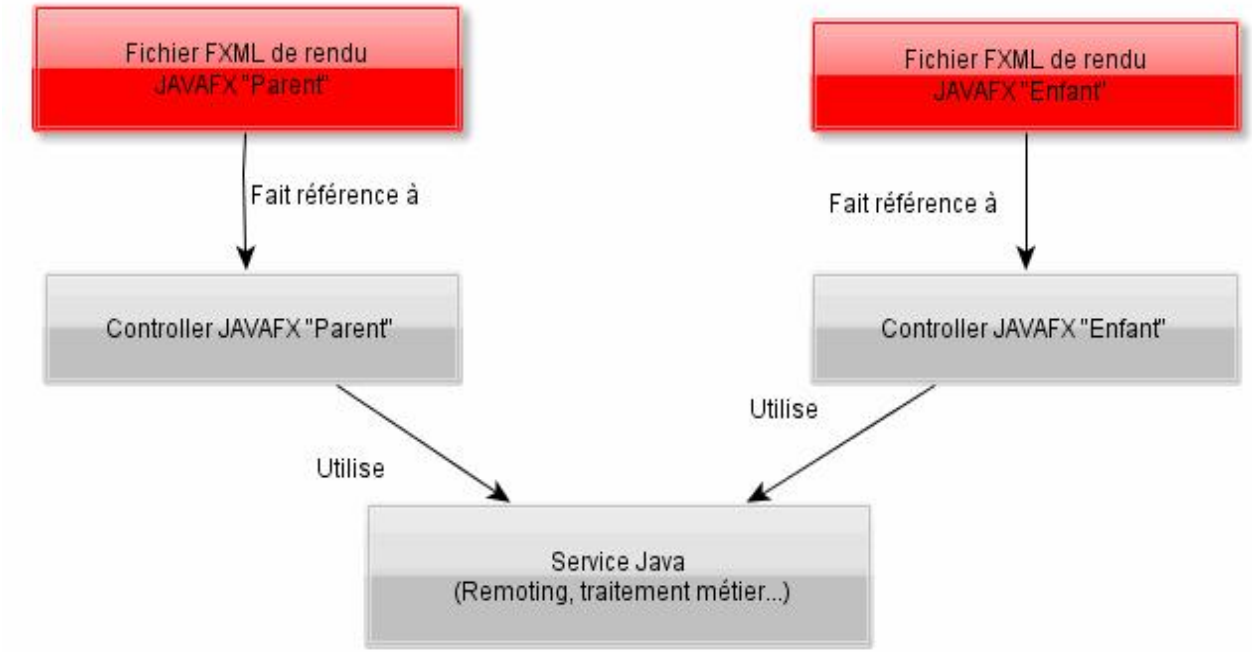
Injection et JavaFX

Fonctionnement et extension

- JavaFX utilise déjà le principe d'IOC (Inversion Of Control) dans ses contrôleurs
  - Utilisation de noms identiques entre FXML et Controller
  - Utilisation d'annotations @FXML
- Néanmoins, ce système est limité, il peut être avantageux d'étendre ce principe d'injection à d'autres classes
  - Mutualisation de code
  - Partage de valeurs entre plusieurs contrôleurs
  - Intégration plus aisée à d'autres technologies Java

EXEMPLE D'UTILISATION

FXML + Controller via IoC



- Permet de gérer l'accès à des ressources communes
- Permet de modifier par la suite l'implémentation d'un service sans impacter les classes qui l'utilisent

Intégration à Spring

ControllerFactory spécifique

- Utilisation d'un Controller Factory qui utilise Spring comme instanciateur de contrôleurs
- Possibilité d'injecter des beans Spring dans les contrôleurs

```
public class SpringControllerFactory<T>
   implements Callback<Class<T>, Object> {

   private ApplicationContext applicationContext;

   public SpringControllerFactory(final ApplicationContext appContext) {
      this.applicationContext = appContext;
   }

   /**
    * Instantiate JavaFX controller from Spring context.
    */
   @Override
   public Object call(final Class<T> aClass) {
      return this.applicationContext.getBean(aClass);
   }
}
```

Intégration à Spring

BuilderFactory spécifique

```
public class SpringBuilderFactory implements BuilderFactory {
    private class SpringComponentBuilder<T> implements Builder<T> {
        private final Class klass;
        public SpringComponentBuilder(final Class klass) {
            this.klass = klass;
        }

        @Override
        public T build() {
            final T bean = (T) applicationContext.getBean(this.klass);
            return bean;
        }
    }

    private final ApplicationContext applicationContext;
    public SpringBuilderFactory(final ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }

    @Override
    public Builder<?> getBuilder(final Class<?> klass) {
        final Component annotation = klass.getAnnotation(Component.class);
        if (annotation != null) return new SpringComponentBuilder(klass);
        else return new JavaFXBuilderFactory();
    }
}
```

Intégration à Spring

Exemple

- Affecter les objets Factory au FXMLLoader

```
FXMLLoader loader = new FXMLLoader();
SpringControllerFactory springControllerFactory =
    new SpringControllerFactory(applicationContext);
loader.setControllerFactory(springControllerFactory);

SpringBuilderFactory springBuilderFactory =
    new SpringBuilderFactory(applicationContext);
loader.setBuilderFactory(springBuilderFactory);
```

  - Il est donc possible d'injecter des beans Spring dans les composants et contrôleurs
  - Pour que les composants JavaFX soient instanciés par Spring ils doivent être déclarés dans le contexte Spring avec l'annotation @Component
  - Ces composants doivent être déclarés en scope « prototype » car ce ne sont pas forcément des singletons

JavaFX + Spring

Une association utile

- Plusieurs avantages à utiliser Spring
  - Technologie éprouvée, support très étendu
  - Mutualisation de code facilitée sur le code métier non IHM
  - Intégration facilitée avec une grande partie des technologies Java
  - Possibilité d'utiliser Spring Events comme bus d'événement (publish / subscribe)
  - Meilleure testabilité avec la possibilité d'utiliser des classes de type mock
  - Il existe toutefois quelques limitations au niveau de Spring AOP car les implémentations des composants JavaFX ne permettent pas toujours d'injecter des proxys

Le design pattern MVVM

Model-View View-Model

- On peut utiliser un design pattern nommé MVVM (Model-View View-Model) qui est une variante de Presentation Model
  - Le contrôleur est une classe qui étend un composant JavaFX
  - Chaque FXML ne connaît pas son contrôleur et ne le référence donc pas
  - Chaque FXML a pour racine un composant JavaFX qui est le même que celui qui est étendu par le contrôleur
  - Le contrôleur charge le FXML dans son constructeur
- Le composant est chargé de déclarer son rendu FXML
  - Suppression de la balise <fxml:include>
  - Permet l'extension / surcharge des méthodes JavaFX
  - Permet l'ajout d'événements personnalisés par databinding

MVVM avec Spring

Exemple

```
<?xml version="1.0" encoding="UTF-8"?>
<fx:root type="javafx.scene.layout.GridPane"
         xmlns:fx="http://javafx.com/fxml"
         styleClass="launchPadPane"
         alignment="BASELINE_CENTER"
         maxHeight="Infinity"
         maxWidth="Infinity"
         hgap="25"
         vgap="25">

    <Label text="Un composant en MVVM !"
             maxWidth="Infinity"
             maxHeight="Infinity"
             GridPane.hgrow="ALWAYS"
             GridPane.vgrow="ALWAYS"
             GridPane.rowIndex="1"
             GridPane.columnIndex="0"/>

</fx:root>
```

MVVM avec Spring

Exemple

```
@Component
@Scope(value = "prototype")
public class LaunchPadMainPane extends GridPane implements Initializable {

   public LaunchPadMainPane() {
                ApplicationContext applicationContext =
                        new ClassPathXmlApplicationContext(new String[]{"META-INF/spring/beans.xml"});

                FXMLLoader loader = new FXMLLoader();
                SpringControllerFactory springControllerFactory = new SpringControllerFactory(applicationContext);
                loader.setControllerFactory(springControllerFactory);

                SpringBuilderFactory springBuilderFactory = new SpringBuilderFactory(applicationContext);
                loader.setBuilderFactory(springBuilderFactory);

                URL url = getClass().getResource("/fxml/LaunchPadMainPane.fxml");
                loader.setRoot(this);
                loader.setLocation(url);
                loader.load();
   }

   @Override
   public void initialize(final URL url, final ResourceBundle resourceBundle) {}

   @PostConstruct
   public void afterPropertiesSet() {}
}
```

MVVM avec Spring

Contexte

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ...>

    <context:annotation-config>true</context:annotation-config>

    <context:component-scan annotation-config="true"
            scoped-proxy="no"
            base-package="com.zenika.javafx.spring,
                com.zenika.javafx.components,
                com.zenika.javafx.dao,
                com.zenika.javafx.dao.impl,
                com.zenika.javafx.services,
                com.zenika.javafx.services.impl"/>

</beans>
```

Null Object Pattern

Principe

- Utiliser une valeur pour symboliser la notion de "null"
- Exemple : une méthode retournant une List, qui retourne Collections.emptyList()
  - Pas besoin de vérifier les valeurs nulles par des if / else
  - On manipule toujours le résultat de la méthode de la même manière (dans notre exemple on va itérer dessus)
  - La manipulation du null object n'aura pas d'incidence sur notre système (l'itération va se terminer immédiatement)

Null Object Pattern

Databinding

- Lors de l'initialisation d'un écran, les données à afficher ne sont pas encore disponibles, que faire ?
  - Utilisation d'un Glass Pane et d'un Splash Screen
  - Indiquer dans les champs de saisie un texte spécifique
- Si un écran est entièrement bindé à des objets Property, alors le null object pattern peut servir
  - Etendre la classe métier bindable
  - Définir dans cette classe les champs à afficher correspondants à la valeur nulle
  - Initialiser la vue avec un objet de type Null Object

Null Object Pattern

Exemple

```
public class Adresse {

    private StringProperty codePostal;
    private StringProperty codePays;
    private StringProperty cedex;
    private StringProperty localite;

    public Adresse() {
        codePostal = new SimpleStringProperty();
        codePays = new SimpleStringProperty();
        cedex = new SimpleStringProperty();
        localite = new SimpleStringProperty();
    }
    ...
}
```

```
public class NullAdresse extends Adresse {

    public NullAdresse() {
        codePostal = new ReadOnlyStringWrapper("XXXXX");
        codePays = new ReadOnlyStringWrapper("--");
        cedex = new ReadOnlyStringWrapper("");
        localite = new ReadOnlyStringWrapper("Ville inconnue");
    }
}
```

DATABINDING SANS PROPERTY

BeanPathAdapter

- Permet la génération d'objets Property à la volée à partir d'un bean Java « standard »

```
BeanPathAdapter<AdresseSansProperty> adresseBeanPathAdapter =
    new BeanPathAdapter<AdresseSansProperty>();
adresseBeanPathAdapter.setBean(new AdresseSansProperty());

Label codePostal = new Label();
adresseBeanPathAdapter.bindBidirectional("codePostal",
    codePostal.textProperty());
```

  - Disponible dans le toolkit jfxtras
  - http://jfxtras.org/

- Technique intéressante, surtout combinée avec un framework type MVP