ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THAPL02

BEST PRACTICES FOR EFFICIENT DEVELOPMENT OF JAVAFX APPLICATIONS

G.Kruk†, O.Alves, L.Molinari, E.Roux, CERN, Geneva, Switzerland

Abstract

JavaFX, the GUI toolkit included in the standard JDK, has reached a level of maturity enabling its usage for Control Systems applications. Property bindings, built-in separation between logic (Controller) and visual part (FXML) that can be designed with Scene Builder, combined with the leverage of Java 8 features such as lambda expressions or method references, make this toolkit a very compelling choice for the creation of clean and testable GUI applications.

This article describes best practices and tools that improve developer's efficiency even further. Structuring applications for productivity, simplified FXML loading, the application of Dependency Injection and Presentation Model patterns, testability are discussed among other topics, along with support of IDE tooling.

JAVAFX OVERVIEW

JavaFX, the successor of Swing, has been around already for a few years. Since the version 1.0 released in 2008, it has been progressively maturing, gaining in functionality and robustness, to be included in the JDK 8.

FXML, Controller and Scene Builder

Swing interfaces have been traditionally created using procedural code. Initialization and configuration of all components and containers had to be coded in Java and visual verification of every change required restarting the application. This was the main driver for WYSIWYG (What You See Is What You Get) editors that aimed to speed up the development and ease the maintenance. However, these editors were mostly generating Java code from the visual representation, a code that was hard to modify and maintain. For this reason many developers preferred to write it manually, resigning from the graphical design.

As an alternative, JavaFX comes with FXML - an XML-based markup language used to describe user interface, and with Scene Builder - a WYSIWYG editor that persists the visual representation in the FXML format.

An integral part of the FXML format is a possibility of declaring an associated controller class and exposing to it UI elements, and event handler hooks. The controller is then responsible for reacting on the events and updating the view accordingly.

This is an example of the *Inversion of Control* [1] principle. The controller does not need to lookup the UI elements it needs to interact with and the invocation of its event listener methods is handled by the FXML logic.

This is a major improvement compared to Swing. The developer can design the interface much faster, without writing and maintaining a lot of boilerplate code, and focusing on the application logic.

Properties and Bindings

Property is a value that represents the state of an object that can be retrieved and set (if it is not read-only). In addition, a property can be observable i.e. registered listeners will be notified every time the property value has changed. This pattern has been used for years by the Java Beans component architecture.

JavaFX provides a set of built-in classes representing properties that extend and enhance this idea with some useful and extremely powerful features.

The properties are often used in conjunction with binding, a mechanism of expressing direct relationships between variables. The binding observes a list of source variables (dependencies) for changes, and updates itself automatically once the change has been detected, applying an optional conversion function.

Since all JavaFX components keep their state in properties, it makes it particularly simple to bind state of different widgets, considerably reducing the amount of necessary code. In the following example the button will be disabled as long as the check box is not selected:

```
button.disableProperty().bind(checkBox.selectedProperty().not());
```

In a similar way UI widgets properties can be bound to observable values of the corresponding view model.

APPLICATION STRUCTURE

Developing GUI applications is not a trivial task. Developers have to address various general software engineering issues as well as GUI-specific ones. Even a single-page application might contain multiple sub-views that need to interact with each other. This brings questions on how the graphical components and their logic should be organized.

There is a quite known statement about clean code by Ward Cunningam, inventor of Wiki and co-inventor of Extreme Programming:

"You know you are working with clean code when each routine you read turns out to be pretty much what you expected".

This statement is true not only with respect to the code and routines it contains, but also to the overall application structure. Without a good structure, the complexity might quickly grow, making the maintenance and further extensions unnecessarily difficult. In addition, the usage of the same structure by all developers in a given organisation greatly facilitates collaborative work, shared support and possible take over of the application by peer developers.

† grzegorz.kruk@cern.ch

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THAPL02

Keeping it in mind, we wanted to propose a solution based on three ingredients: an agreed set of GUI design patterns separating the graphical design from the business logic, a convention for consistent code organization and naming, and finally some means that would facilitate and encourage applying such structure by all our developers. We wanted also to keep it as simple as possible. Many of our developers are not professional software engineers (e.g. operators or physicists) thus we did not want to impose on them usage of complicated frameworks or APIs.

GUI Patterns

There are several design patterns that have been proposed over the years by the software community to address common concerns in UI development. Concepts such as *Model-View-Controller* (MVC), *Model-View-Presenter* (MVP), *Presentation Model* (PM) or *Model-View-ViewModel* (MVVM) have been discussed in numerous articles, blog posts and forums.

The decision of using one over another depends on many factors including type and size of the application, particular language and widget toolkit features, the level of desired testability, or personal preferences of developers.

JavaFX by itself does not impose any particular pattern, but it implies a natural split between the visual part (FXML) and the logic (controller). Nonetheless, depending on the chosen approach, the actual implementation of the FXML controller may take different forms.

In the simplest case it can play a role of a *Supervising Controller* [2] containing complete logic, initializing bindings between different components, handling input events, interacting with external services and updating the view (see Fig. 1). The application's state is mostly kept in the view but the controller may also keep part of it when necessary.

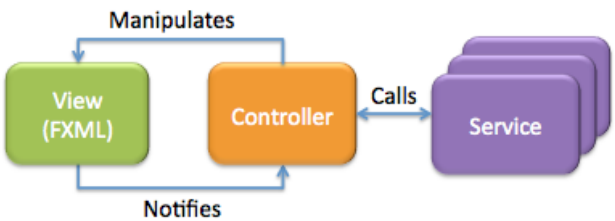

Figure 1: Simple variant of a Supervising Controller

For simple applications this might be completely sufficient and appropriate. By separating the behavioural complexity from the view it makes the application easier to understand and greatly improves its testability.

In applications that are more complex and contain multiple views, it is typically much more profitable to introduce some sort of model.

In one variant, the model can be completely passive, containing only state (properties) of the view to be shared with other views and their controllers. This eliminates the need of controllers exposing state of their views to other controllers. In many cases it may also eliminate the need of controllers knowing each other, removing coupling between them and therefore improving their testability. In such case the model is the only communication channel between different entities (see Fig. 2).

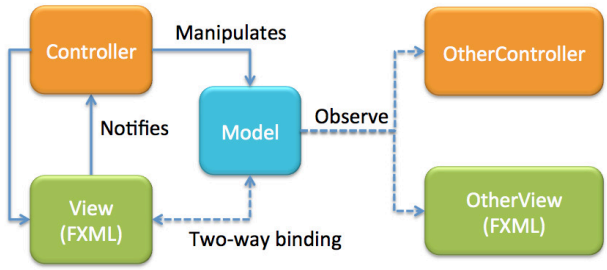

Figure 2: Example usage of a shared model

Finally, if testability is the main driver and writing unit tests involving GUI components is either not desired or difficult, the model can completely take over business logic, becoming a realization of a *Presentation Model* [3] like on the Figure 3. The responsibility of the FXML controller is reduced to a role of a "thin" bridge between the view and the model, which handles all events and updates the view via property bindings.

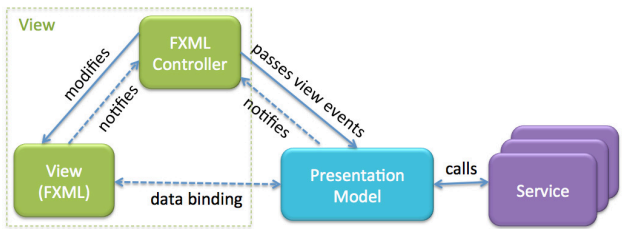

Figure 3: An example of a Presentation Model

This approach requires a bit more coding compared to previous scenarios, giving in exchange fully testable business logic, not bound to any graphical components.

Conventions

Both, the FXML file and its controller should have names allowing an easy identification that they belong to the same view, without looking at their content.

In fact, JavaFX introduced a naming convention for nested controllers [4]. For instance, if an included view ID is *dialog*, then the corresponding controller can be referenced as *dialogController*. This convention could be extended to other entities associated with a single view such as model, service, CSS or resource bundle properties file. In addition, all files related to a single view could be placed in a dedicated Java package, named after the view. In such case the content of every package would be similar:

- [view_name].*fxml*
- [view_name]*Controller.java*
- [view_name]*Model.java*
- [view_name]*Service.java*
- [view_name].*css*
- [view_name].*properties*

Note that only the FXML, controller and in most cases also model files are be present, while CSS, resource bundle and any additional files are optional.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THAPL02

Such convention is very easy to remember. With a glimpse of an eye one can recognize all the elements and have a good idea what is inside.

Afterburner.fx

Almost identical structure has been proposed by Adam Bien in his *Afterburner.fx* framework [5], with the only difference that instead of *Controller* he uses a name *Presenter*. The framework is minimalistic and very simple but at the same time brings a lot of added value by leveraging the *Convention Over Configuration* principle.

The central entity of the framework is *FXMLView*. It is an abstract class that must be extended for every view and given a conventional name e.g. *DeviceView.java* would have its *device.fxml, DevicePresenter.java, device.css* etc. Relying on the conventional name, the *FXMLView* loads all related files and binds them together, supporting JSR 330 [6] *@Inject* Dependency Injection (DI) for controllers, models and services.

In most cases the view class is empty and its only purpose is to define the conventional name and give access to the instantiated root node defined in the FXML.

Being inspired by the framework we realized that its main idea could be simplified even further.

FxmlView

Rather than relying on the view class defining the conventional name, we decided to use the controller class that needs to be implemented in any case.

As a result we developed a generic *FxmlView* that for the given controller class loads the associated FXML, resource bundle and applies CSS file, eliminating the need of dedicated view classes. A basic usage is illustrated below:

```
public class App extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        FxmlView mainView = new FxmlView(MainController.class);
        Scene scene = new Scene(mainView.getRootNode());
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args){
        launch(args);
    }
}
```

To instantiate controllers, the *FxmlView* relies on the associated controller factory. By default it is initialized to *DefaultControllerFactory* that also supports *@Inject* annotation. For every call, the factory creates a new instance of the controller but all the dependencies (models, services, etc.) are treated as singletons and kept in an internal cache. Thanks to that models can be easily shared between different controllers as in the following example:

```
public class PersonModel {

    @Inject
    @Named("person.age.visible")  // Inject value of a property
    boolean defaultShowAge;

    // Inject value of a property with name equal to the field name
    @Inject
    Side personDetailsPaneSide;
     …
}

public class PersonService {
    Person findByName(String name) {
       //...
    }
}

class PersonController {
    @Inject
    PersonModel model;

    @Inject
    PersonService service;
     …
}

class AddressController {
    @Inject
    PersonModel model;
     …
}
```

In this example, fields of the *PersonModel* are be initialized from an optional properties provider (*Function<String, Object>),* with a fallback to JVM properties (with applied conversion from String to the corresponding primitive or enum value).

The *DefaultControllerFactory* instantiates all dependencies using their default constructor. If this behaviour is not desired or not sufficient, the dependency instance can be also registered manually as in the following example:

```
PersonService service =
     ServiceLocator.getService(PersonService.class);
DefaultControllerFactory factory = …;
factory.setDependency(PersonService.class, service);
```

The *DefaultControllerFactory* can be easily replaced by another implementation based e.g. on a more powerful DI engine such as Spring or Google Guice:

```
SpringControllerFactory springFactory = …;
FxmlView. setControllerFactory(springFactory);
```

The *SpringControllerFactory* would simply return controller bean defined in the Spring application context.

GUI TESTING

User Interface test automation is a tricky practice, posing a unique set of challenges compared with testing of non-graphical components. For instance, certain functionality of components may work only if they are visible on the screen. Also in some cases events are not executed immediately in the current thread but scheduled in an event queue for later execution by the GUI thread.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THAPL02

For this and other reasons it is often believed that automation of GUI testing is complex, requires a lot of additional work and overall it is not worth the effort. While this might be true in some cases, it definitely does not apply to majority of applications. In fact, automated testing of critical paths could be implemented with an effort equivalent to the implementation of server-side unit tests. But even more than on the server-side, the testability of developed applications must be taken into account from the very beginning.

In general there are three approaches to automated GUI testing:

- Testing only business logic that has been separated from GUI components
- Testing that involves interactions with and verification of graphical components
- Robot-based testing, where a library or tool mimics user actions (mouse and keyboard) and allows verifying the resulting state of the interface.

These three techniques do not exclude each other. On the contrary, they can be used together in a complementary way.

Separating Logic From GUI

Separating business logic from visual components is probably the most popular way of improving testability of an application. The goal is to make the view as "thin" as possible by putting all the associated logic in controllers and models. This typically should include logic related to the visual aspects of the interface such as colors, visibility or layout properties.

All the techniques discussed in *GUI Patterns* paragraph can be applied in a JavaFX application to make a separation between the view and controller. However, some of the JavaFX built-in features favour usage of some patterns over others. In particular, the presenter of the *Passive View* [7] pattern holds a complete responsibility of updating the view. This enables a high level of testability but severely limits usage of property bindings, a mechanism that does not only save a lot of code but also makes the application cleaner and easier to maintain.

Therefore for JavaFX it is typically much more advantageous to employ the *Presentation Model* [3] or *Model-View-ViewModel* [8] patterns. For the price of slightly lower test coverage, the developer can fully profit from the property bindings.

FXML Controller Testing

Even the simplest form of a *Supervising Controller,* containing references to graphical components (from the FXML), can be tested quite well using classical unit tests as in the following example:

```
@RunWith(FxJUnit4Runner.class)
public class MainControllerTest {

    @Test
    @RunInFxThread
    public void testCopyMessage() {
        FxmlView mainView = new FxmlView(MainController.class);
        MainController controller = mainView.getController();
        controller.inputTextField.setText("test");
        controller.copyButton.fireEvent(new ActionEvent());
        assertEquals("test", controller.outputLabel.getText());
    }
}
```

The *MainControllerTest* class, placed in the same package as the *MainController*, has access to its package-visible fields, including graphical components injected from FXML. Therefore it can change their properties, fire events and verify state.

There are however two constraints on such tests to run. First, creation of any FX control requires prior initialization of the FX toolkit, which normally is done by the *Application* class at the start-up. Secondly, certain operations are permitted only from the *FX Application Thread*, therefore executing them from an arbitrary JUnit thread would be rejected.

We addressed those two issues by implementing *FxJUnit4Runner*. It is an extension of standard JUnit test runner that initializes the FX toolkit and, in presence of the *@RunInFxThread* annotation, runs the corresponding tests in the FX thread.

We use our own implementation of the runner, but its recipe can be found on different forums as well as in ready to use implementations [9].

TestFX

Robot-based Java GUI testing frameworks were around since the beginnings of AWT/Swing, but most of them faced two major issues: the tests were usually quite verbose and the graphical components were typically looked up by their location on the screen, making the tests very fragile to even minor changes in the layout.

In contrary TestFX, the most popular testing framework for JavaFX, does not suffer these problems. Like other similar tools, it gives a programmatic control of a "robot" that one can use to click on buttons, type into text components and generally mock user interactions. However the fluent API, supported by powerful matchers, allows writing tests that are concise, clean and easy to understand. Also, rather than relying on component's location (although this is also possible), it leverages the usage of CSS IDs and class names, that are natural part of JavaFX interfaces. Here is one of TestFX examples:

```
// given:
rightClickOn("#desktop").moveTo("New").clickOn("Text Document");
write("myTextfile.txt").push(ENTER);

// when:
drag(".file").dropTo("#trash-can");

// then:
verifyThat("#desktop", hasChildren(0, ".file"));
```

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THAPL02

The remaining, trickiest part to address is mocking behavior of external services and access to resources (devices, databases, etc.) that most of controls applications rely on. But extracting such external calls into dedicated service/access points, applying Dependency Injection and usage of mocking libraries like Mokito [10], may come here with a rescue. The initial architectural and structural choices may either facilitate or heavily hinder such practices. That is why it is extremely important to think about testability before the application development even starts.

TOOLS

Tools may not only greatly speed up the development of JavaFX applications but also help keeping a proper structure.

Scene Builder

Scene Builder (see Fig. 4) is the first and most important tool that every JavaFX developer should install and use. It allows dragging and dropping UI components in the working area, modifying their properties or applying styles, in a quite efficient and user-friendly manner.

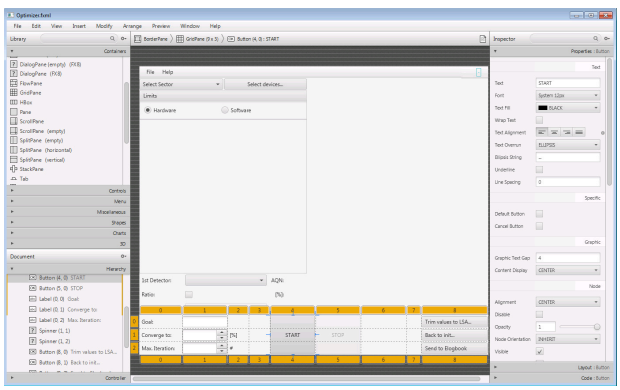

Figure 4: Scene Builder

Although not all properties and event handlers can be configured via the tool, it covers vast majority of needs of a typical application, leaving more custom cases to be coded in the FXML controller.

E(fx)clipse

JavaFX developers that use Eclipse can also profit from *e(fx)clipse* plugin [11]. It provides several handy tools making the development more efficient. We mostly use generation of JavaFX getters and setters, a specialised CSS editor (see Fig. 5) that knows and prompts JavaFX-specific attributes and occasionally also FXML editor to adjust configuration generated by the *Scene Builder*.

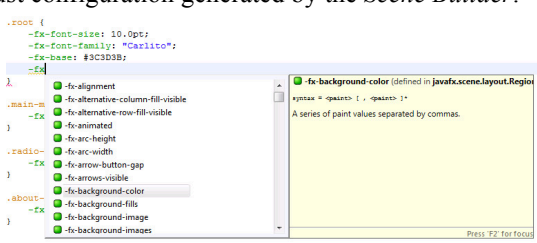

Figure 5: CSS editor

Plugins providing similar functionality exist for other IDEs, so NetBeans and IntelliJ users can also profit from smart JavaFX editors and automation of several tasks.

Application Creator

On top of that we developed our custom Eclipse plugin to generate seed applications and views based on a predefined set of templates.

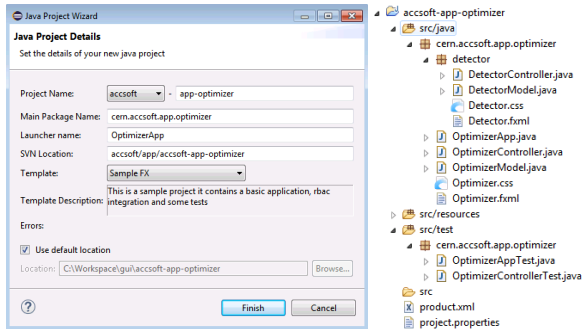

Figure 6: Configuration dialog and generated project

It is tailored to the CERN environment, suggesting project and java package name following our convention as well as corresponding location in the SVN repository. The developer just types the project name and selects one of available project templates (see Fig. 6). The plugin then generates a ready to run JavaFX project with proper structure and configured set of core dependencies.

The different templates contain sample applications that vary in level of complexity and applied pattern. The simplest is a one-page *Hello World* application with just a single FXML, corresponding controller and CSS file. Other samples contain two or more views and demonstrate different ways of interactions between them and with external services. All of the templates follow the project structure and naming conventions as described in the *Conventions* paragraph. They also all contain corresponding test classes for both: the classical unit test of the controller and TestFX tests cases.

A part of the plugin is also a view creator, visible on Figure 7.

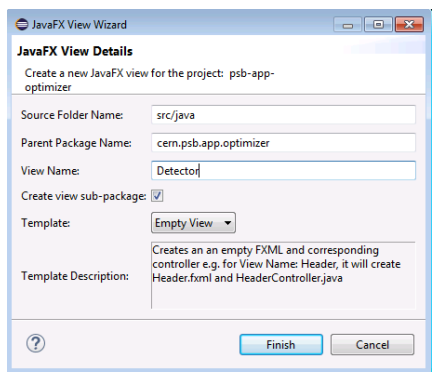

Figure 7: View configuration dialog

The developer types name of the view, selects one of the view templates and the wizard creates the corresponding java package containing all related files.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THAPL02

Finally, we added a small, but extremely useful feature that generates controller's fields and event handles based on the FXML. It compares the content of existing controller, that might be half-developed, with the FXML and generates only these fields and event handlers that are not yet present, inserting them in the right place in the class i.e. fields are added just after the existing FXML fields (or at the beginning of the class) and event handler methods are added after the last existing handler (or at the end of the class).

CONCLUSION

The JavaFX WYSIWYG approach is very efficient. Combined with property bindings and Java 8 lambda expressions, allows rapid creation of code that is concise and at the same time easy to understand.

With a simple and consistent convention, developers do not have to spend their time rethinking code organization and naming. The convention is enforced by the *FxmlView*, which also simplifies instantiation of views and saves developers from writing repeatable code.

The built-in features of JavaFX, usage of appropriate design pattern and framework like TestFX enable easy implementation of unit and integration tests.

Finally, the power of appropriate tooling should not be forgotten. Even small features that automate repeatable tasks may boost the development and make it much more pleasant. The *Application Creator* allows our developers to set up a new application or view within a couple of seconds. But more importantly, by providing meaningful templates, it promotes best practices. The idea is more essential than the tool. One can create a similar tool within a day or two, or use an existing one like *Lazybones* [12]. Such small investment brings a huge return in productivity and maintainability.

REFERENCES

[1] Inversion of Control, https://en.wikipedia.org/wiki/Inversion_of_control
[2] Supervising Controller, https://martinfowler.com/eaaDev/SupervisingPresenter.html
[3] Presentation Model, https://martinfowler.com/eaaDev/PresentationModel.html
[4] Nested Controllers, https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html#nested_controllers
[5] Afterburner.fx, https://github.com/AdamBien/afterburner.fx
[6] JSR 330, https://jcp.org/en/jsr/detail?id=330
[7] Passive View, https://martinfowler.com/eaaDev/PassiveScreen.html
[8] ViewModel, https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel
[9] jfx-testrunner, https://github.com/sialcasa/jfx-testrunner
[10] Mockito, https://github.com/mockito/mockito
[11] e(fx)clipse, http://www.eclipse.org/efxclipse/index.html
[12] Lazybones, https://github.com/pledbrook/lazybones