Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Tobias Ask

sb4e: an open source integration of the Scene Builder GUI editor into the Eclipse IDE

Facilitating JavaFX application development with an extension to the IDE

Master's thesis in Computer Science
Supervisor: Hallvard Trætteberg
June 2019

NTNU
Norwegian University of Science and Technology

Tobias Ask

sb4e: an open source integration of the Scene Builder GUI editor into the Eclipse IDE

Facilitating JavaFX application development with an extension to the IDE

Master's thesis in Computer Science
Supervisor: Hallvard Trætteberg
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

I detail the design, implementation and evaluation of an open source computer-aided software engineering (CASE) tool for developing Java applications with graphical user interfaces (GUIs) with the JavaFX framework. The tool is an integration of the Scene Builder editor for view documents in JavaFX — as pertaining to the *model-view-controller* architectural pattern — into the Eclipse integrated development environment (IDE). The tool extends the IDE with facilities for editing view documents written in FXML, the JavaFX framework's custom user interface markup language, in a drag and drop editor. In addition, it applies developer support mechanisms already present in the IDE to the JavaFX application development domain, to assist developers in maintaining the connection between views and their controllers that needs to be in place for a functional interaction between them in a running application. Further, I investigate the implications of the tool being open source on the project's sustainability, visibility and governance. Open source software (OSS) projects are unique with regard to all of these concerns.

Sammendrag

Jeg presenterer design, implementasjon og evaluering av et programvareutviklingsverktøy med åpen kildekode for å lage Java-applikasjoner med grafiske brukergrensesnitt med JavaFX-rammeverket. Verktøyet er en integrasjon av redigeringsverktøyet Scene Builder for *view*-dokumenter i JavaFX — som definert i henhold til arkitekturmønsteret *model-view-controller* — inn i utviklerverktøyet Eclipse. Verktøyet utvider Eclipse med muligheter for å redigere *view*-dokumenter skrevet i FXML, JavaFX-rammeverkets egendefinerte språk for å definere brukergrensesnitt, i et *dra-og-slipp*-verktøy. I tillegg anvender verktøyet mekanismer for utviklerstøtte som allerede finnes i Eclipse på området som angår utvikling av JavaFX-applikasjoner. Det gjøres for å assistere utviklere med å vedlikeholde koblingen som må være til stede mellom *view* og deres *controllere* for at de skal kunne kommunisere korrekt med hverandre i en JavaFX-applikasjon. Videre ser jeg på hvordan det at prosjektets kildekode er åpen, tilgjengelig for offentligheten, påvirker bærekraften, synligheten og styringen av prosjektet. Prosjekter med åpen kildekode er unike når det gjelder alle disse aspektene.

Preface

My master's degree in Computer Science at the Norwegian University of Science and Technology culminates in the project presented in this thesis. It accounts for thirty ECTS, and was written during the spring of 2019. The work was performed under the supervision of Associate Professor Hallvard Trætteberg.

The target audience is familiar with concepts in object-oriented programming and the Java programming language. Beyond that, some familiarity with basic UML diagram notation is preferred, as such diagrams are used frequently in favor of code to explain systems and processes.

Trondheim, June 2019
Tobias Ask

Acknowledgements

I foremost thank the supervisor of the project, Associate Professor Hallvard Trætteberg, for his support and guidance along the way. The integration was his idea, and he has been offering his expertise on a weekly basis since the beginning. His genuine interest in the project has been a motivational factor for me when working with the integration. While he has guided the project, he has also allowed me to work independently, exploring different directions at my own will.

Further, I am grateful to Gluon, the developers of Scene Builder, for making their product open-source. The development of the plug-in would naturally not have been technically feasible without them. I am also thankful to Tom Schindl, the founder of the *e(fx)clipse* project, for answering my technical inquiries. Lastly, I recognize and appreciate what an impressive technical achievement the Eclipse platform is.

Table of Contents

List of Figures

List of Tables

1 Introduction

In JavaFX[1] — a framework for developing applications with graphical user interfaces (GUIs) in the Java programming language — the interface of an application can be defined separately from the implementation of its behaviour. This is a technique known as the *model-view-controller* pattern, where controllers handle the interactions made with the interface, while the visual attributes of the interface such as the size, colour and placement of its graphical components are declared in views (Bass, Clements and Kazman, 2013, pp. 212-215). JavaFX controllers are written in Java, and views in the framework are declared in documents written in a custom markup language based on the Extensible Markup Language (XML),[2] called FXML (Brown, 2011).

As FXML is a textual data format, views declared therein can be edited in a text editor. However, an application in which they can be edited using a visual drag-and-drop editor has been developed, called Scene Builder.[3] It displays a preview of the view to the developer as they are working on it, enabling an editing process with instant feedback. Even though Scene Builder facilitates application development with JavaFX to a great extent, views and their controllers are still tightly linked, however, and seeing as controllers are coded in integrated development environments (IDEs), Scene Builder's usefulness as a stand-alone application is somewhat limited.

It was with this limitation in mind that I developed an integration of Scene Builder into the Eclipse IDE,[4] embedding the drag-and-drop editor within it for use with JavaFX views (Ask, 2018). This meant that the two development processes — i.e., controller coding and view editing — could be unified in a single application with a cohesive interface, and a consistent mental model. The integration process was successful because Scene Builder as a system is constructed in a decoupled manner, suitable for reuse (Ask, 2018), and because it is open source.

Amsden (2001) defines five distinct levels that classify the integration of external tools with the Eclipse platform, according to the extent and technical depth of their integration. The five levels, along with their descriptions, are listed in table 1.1. According to his classification, the initial integration integrated Scene Builder with Eclipse at the *UI* level. As it is the maximal attainable integration level, achieving it might indicate that there is little left to be done. After all, it goes a long way beyond the integration that existed before it, which was at the *Invocation* level. It is provided by *e(fx)clipse*[5], a collection of general JavaFX tools for Eclipse, and enables the launch of Scene Builder — as a stand-alone application — from within Eclipse to edit views. Classifying the initial integration's integration level as a maximal attainable level is misleading, however, as it did not interact much with the rest of the platform, beyond embedding Scene Builder in it. The integration reaching the *UI* level is also a natural consequence of Scene Builder being a graphical tool; any other type of integration would not have been meaningful.

Going from the *Invocation* to the *UI* level — while admittedly skipping the *Data* and *API* levels — is a good start in terms of facilitating JavaFX application development with Eclipse, but it is not an ideal integration. It quickly became apparent that there were many avenues of a deeper integration left to explore. If the integration of the applications is to be of a higher value to the end-user than the two of them side-by-side, it needs to provide

[1] `https://openjfx.io/` [2] `https://www.w3.org/TR/xml/` [3] `https://gluonhq.com/products/scene-builder/` [4] `https://www.eclipse.org/` [5] `https://www.eclipse.org/efxclipse/index.html`

Table 1.1: The five levels of integration with the Eclipse platform attainable by external tools, defined by Amsden (2001). The depth levels are inclusive, and listed from top to bottom in ascending order — i.e., the *UI* level is the maximal achievable level.

Integration level	**Description**
None	No integration, tools are separate and independent.
Invocation	Integration is through invocation of registered applications on resource types.
Data	Integration is achieved through data sharing.
API	Tools interact with other tools through Java APIs that abstract tool behavior and make it publicly available.
UI	Tools and their user interfaces are dynamically integrated at runtime including window panes, menus, toolbars, properties, preferences, etc.

additional user assistance. When considering the previously mentioned tight relationship between views and their controllers, it is clear that the IDE can assist developers in a number of ways by utilizing the many capabilities provided by the Eclipse platform. Consider, as an example that showcases some of these capabilities, the assistance Eclipse provides when a compilation error caused by a reference to an undefined method in a type is detected: First, the problem, along with its cause, is recognized. The problem is then reported to the developer in the form of a number of visual indicators, chief among which being the red squiggly line in the editor. When the developer hovers the mouse pointer over the squiggly line, the option of creating the unresolved method in the class is offered. If it is chosen, Eclipse adds the method to the class, and the red squiggly line disappears.

However trivial the example may seem, it serves the purpose of demonstrating how an IDE differs from a text editor. It also points towards the addition of a sixth integration level to the ones previously defined by Amsden (2001). I propose here the *Process facilitation* level, where the domain specific work process — JavaFX application development, in this case — is facilitated by the integration to an extent that exceeds the value achieved by a mere embedding of the tool within the platform. At this level, new functionality becomes available to the developer, and the functionality of the external tool itself is extended by utilizing capabilities provided in the platform.

1.1 Project objectives

In this project, I aim to build upon the previously developed integration to create a more complete developer tool that is both appealing to users as an end product, and maintainable by other contributors in the future as an open source software project. To accomplish this, I will proceed with my work according to the following objectives:

1. **Identify means by which the Eclipse plug-in can facilitate JavaFX application development**

   In pursuit of this analytic objective, I will analyze the connection between views and controllers in the JavaFX framework, and investigate how various mechanisms in the Eclipse platform can be applied to help maintain the connection. I will look to developer assistance mechanisms already present in the IDE for inspiration, and aspire to identify support mechanisms idiomatic to it that can be implemented. I will also explore how Scene Builder's facilities for interacting with the project the view document is located in can be extended by utilizing Eclipse mechanisms.

2. **Design and implement the identified assistance mechanisms**

   After I have performed my analysis, I will continue by designing and implementing the identified mechanisms. I will pursue conformity of implementation with existing techniques, best practices and patterns in the Eclipse platform, while attempting to make as few changes to reused Scene Builder components as possible.

3. **Ensure the project's impact and sustainability**

   I will strive to promote the project's use by as many users as possible. To this end, I will make the project's source code publicly accessible at a web-based hosting service such as GitHub.[6] No matter how good the end product turns out to be, its quality alone does not its imply success, however. I will therefore also take the governance of the project into account, looking at how to attract and manage end users and potential contributors alike, according to established practices. Another important aspect of the project's governance is its relation to its ecosystem — the upstream Scene Builder project in particular.

1.2 Thesis structure

This thesis is structured as follows: The immediately forthcoming chapter introduces the three technologies the plug-in concerns: *JavaFX*, *Scene Builder*, and *Eclipse*. Next, I analyze the connections between views and controllers in the JavaFX framework and investigate how Eclipse can assist in maintaining the correctness of them. The ensuing chapter details the implementation of the identified solutions. Then, I handle the topic of the plug-in project's governance, looking at how to promote its sustainability. I subsequently attempt to systematically evaluate the plug-in in terms of its usability and fulfillment of certain requirements. The thesis concludes with some closing remarks on the results of the evaluation, as well as thoughts on the future of the plug-in.

[6] `https://github.com/`

2 Background

This chapter describes the three technologies that the integration synergically joins: JavaFX, Scene Builder, and Eclipse. Section 2.1 covers the JavaFX framework briefly, focusing on the connection between a view and its controller therein. Section 2.2 introduces Scene Builder from the perspective of both its end-users — as a GUI editor — and that of developers extracting its components for use in other tools, as a component-based system. Section 2.3 presents Eclipse as an extensible platform and shows how the IDE can be extended with new functionality. It also introduces mechanisms in the platform that can be utilized by the integration to aid with JavaFX application development.

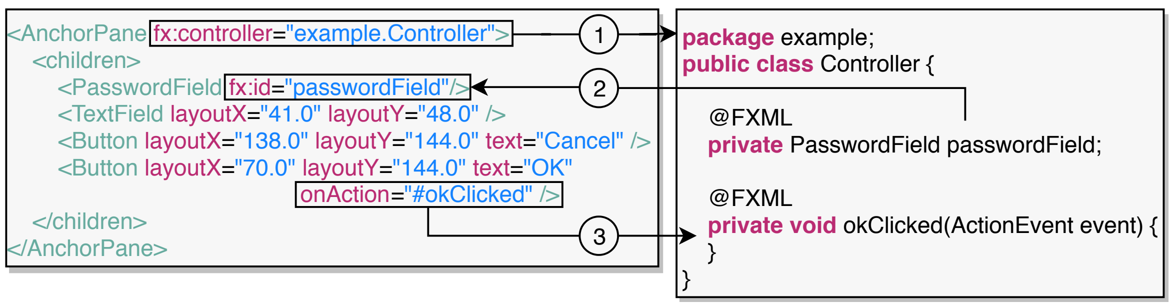
2.1 The JavaFX framework

JavaFX is a framework for the development of client applications with GUIs in the Java programming language. It came bundled with the main release of Oracle's Java Development Kit (JDK) up until its eleventh release; since then, it has been released and maintained separately (Smith, 2018). The framework is now open source, beneath the charter of the OpenJDK project.[1]

In JavaFX, the content of the GUI — the view — is represented as a collection of nodes arranged in a tree structure called a **scene graph**. It is a commonly used structure in graphical applications as it allows operations such as resizing to propagate downwards from a node to its children. Scene graphs can be defined in documents written in the framework's custom markup-language, FXML, that contain the declaration of the structure of the graph, along with the attributes of its nodes such as their positions, colors, and sizes. To make the GUI interactive, the view interacts with a Java class called its controller. There are three special attributes that are used to achieve the interaction between them: the *fx:controller* attribute, the *fx:id* attribute, and the various *event handler* attributes. Figure 2.1 highlights the use of these attributes in an example view along with its controller.

The **fx:controller** attribute is used to associate the controller with the view on a general level, by setting the root node's value of the attribute to the name of the controller class. Further, **fx:id** attributes link field declarations in the controller with their corresponding component instances in the view, to enable programmatic manipulation of them. The component instances are injected into the field declarations at runtime when the view document is loaded. Lastly, **event handler** attributes assign method declarations in the controller as the recipients of control flow when events are fired by view components. These events are normally triggered by user gestures such as drag-and-drop, key presses and mouse clicks. Event handlers are required to take a single argument of a type extending `javafx.event.Event`, and they must return `void` (Brown, 2011).

[1] `https://github.com/javafxports/openjdk-jfx`



```
<AnchorPane fx:controller="example.Controller">
  <children>
    <PasswordField fx:id="passwordField"/>
    <TextField layoutX="41.0" layoutY="48.0" />
    <Button layoutX="138.0" layoutY="144.0" text="Cancel" />
    <Button layoutX="70.0" layoutY="144.0" text="OK"
                                 onAction="#okClicked" />
  </children>
</AnchorPane>
```

```
package example;
public class Controller {

  @FXML
  private PasswordField passwordField;

  @FXML
  private void okClicked(ActionEvent event) {
  }
}
```

Figure 2.1: Examples of the three types of links between a view and its controller in JavaFX. Link 1 shows how the view points to the controller class name in the `fx:controller` attribute of the root component of the view's scene graph. The controller's `PasswordField` field points to the corresponding component in the view's scene graph, indicated by link 2. Last, the `Button` component's `onAction` attribute references the method declaration in the controller, as illustrated by link 3, assigning it as the recipient of program flow when the button's `Action` event is fired.

2.2 The Scene Builder GUI editor

Scene Builder is a *what you see is what you get* (WYSIWYG) editor for JavaFX view documents declared in FXML, the framework's custom markup-language. Originally developed by Oracle, it is now in the hands of Gluon, who have made the project open source.[2] Figure 2.2 depicts a screenshot of its main interface. It features many of the interface parts typically seen in resource editors, including a main workspace area; a palette from which items can be inserted into the workspace; and an editor for the properties of the selected element in the workspace. The following section describes and names these parts through the lens of an end user.

2.2.1 Scene Builder as an application

The **Library panel** displays a collection of the GUI components that are used to build the view. It is located at the top of the application interface's left-hand panel. By default, it contains the standard JavaFX components (buttons, input fields etc.), as well as some custom component types provided by Gluon. New items can be added to the panel's collection from external sources in the form of local Java Archives (JARs), FXML documents, and JARs fetched from online repositories, such as Maven Central. The **Document panel** is located below the Library panel, and is comprised of two sub-panels: the Hierarchy and the Controller panel. The **Hierarchy panel** displays the view components in a tree structure, providing an hierarchical overview of the content. The association from the view to its controller is managed from the **Controller panel**, which has an input field for the controller name, as well as an overview of all the `fx:id` declarations in the view. The input field for the controller class name shows suggestions of candidate controllers to the developer, which helps reduce the chance of a wrongly entered controller name — the consequences of which can be fatal.

In the middle of the application interface is the **Content panel**, which occupies most of its display area. It displays a static preview of what the view's content will look like when

[2] `https://github.com/gluonhq/scenebuilder`

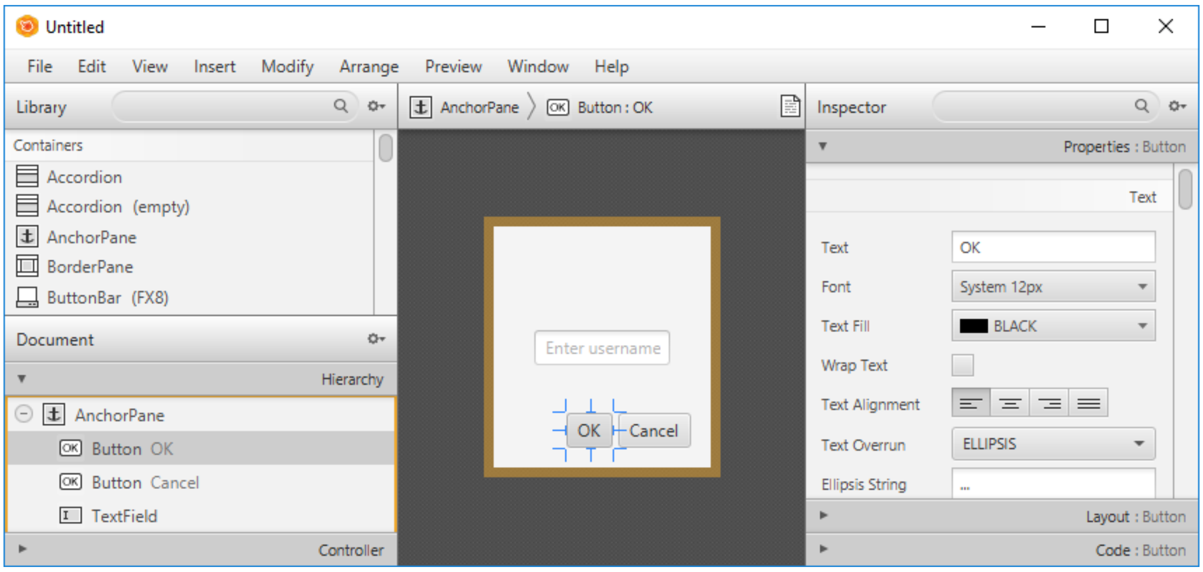


Figure 2.2: The main user interface of the Scene Builder application.

rendered in a running application. The developer modifies the layout structure of the view in the panel, including the placement, size and parent-child relations of the components. The **Inspector panel** is an editor for the attributes of the individual components of the view, such as their text, layout constraints and appearance. It is located in the right-hand panel of the interface, and is linked with the component selection in the content panel. Scene Builder is aware of the contents of the view's controller (if there is one), and shows suggestions of `fx:ids` and event handler names based on the controller's field and method declarations in their respective input fields in the panel.

This concludes the presentation of the main elements of Scene Builder's interface, as well as most of its key functionality. While it glossed over much of the application's more sophisticated functionality and subtleties, it has introduced the terminology regarding the application's interface elements that is used in subsequent discussions. The next section introduces key parts of the system's architecture, a familiarity with which is also required for later discussions.

2.2.2 Scene Builder as a component-based system

Here, I introduce some parts of Scene Builder's architecture from the perspective of reusing its components in a different system than the stand-alone application. Consequently, the presentation will only focus on the parts that the integration interacts with directly, rather than the system as a whole. A more in-depth analysis of its architecture and how it makes its components suitable for reuse, in particular, can be found in my specialization report (Ask, 2018). It is worth mentioning at this point that it is in itself built with JavaFX, making it a meta editor for the framework.

At its root level, the system is split into two modules: the *app module* and the *kit module*. It is Gluon's intention that the latter is to be used to develop IDE integrations. Hence, much of the app module depends on components in the kit module, but not the other way around. The system can be viewed in a two-layered view wherein the app layer is the top layer, while the kit layer is the bottom layer. The interface's division into separate panels is reflected in the system architecture. Each panel has its own view and controller that

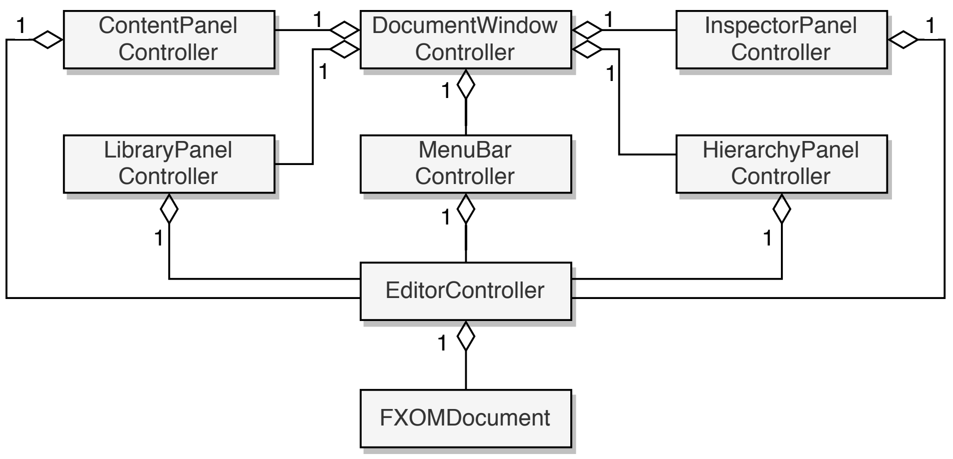


Figure 2.3: UML class diagram of Scene Builder's controller structure. The panel controllers are all independent of each other, but joined together in the main application window by the `DocumentWindowController`. They interact with the document via the `EditorController` through one-way associations, which means that it can be extracted and instantiated in isolation from the rest of the system. Note that some controllers are omitted for clarity.

concern only its functionality in isolation. The panels are independent of each other, and are only joined together to form the complete interface by the main window controller. The panels do not have direct associations to the document; rather, they connect to it via the *editor controller*, which is the glue that allows the panels to modify the document and communicate with the rest of the system. Figure 2.3 illustrates these points with a class diagram of Scene Builder's controller structure.

It is clear that the editor controller is the locus of the system's operation. There are, however, a few other, key components in the system that are worth mentioning. They can be seen in the class diagram in figure 2.4, which also illustrates how the system relates to the rest of the file system through these components. All document modifications go through the **Job manager**, which sits between the editor controller and the FXML document being edited. This structure implements the reversibility of the developer's operations that is needed in resource editors, and allows the developer to undo and redo their actions. All modifications are represented as abstract jobs that are pushed to the job manager, which handles their execution. When, for example, the controller for the Content panel modifies the document in response to the developer's actions, it instantiates a job that it pushes to the job manager, the reference to which it acquires through the editor controller.

The **Glossary** is responsible for inspecting the controller associated with the document (if any) to generate the suggestions that are shown in the input field for the controller name in the Document panel, and the fields for the `fx:ids` and the event handler names in the Inspector panel. It is an abstract class — with a default implementation included in the distribution — that can be subclassed to customize the way the system connects with the controller. The **Library** component manages the collection of components that is displayed in the Library panel. As mentioned, it supports the addition of items from external sources. Further, it also monitors a folder in the file system from which custom components are discovered and added automatically to its collection. Lastly, the Scene Builder system has

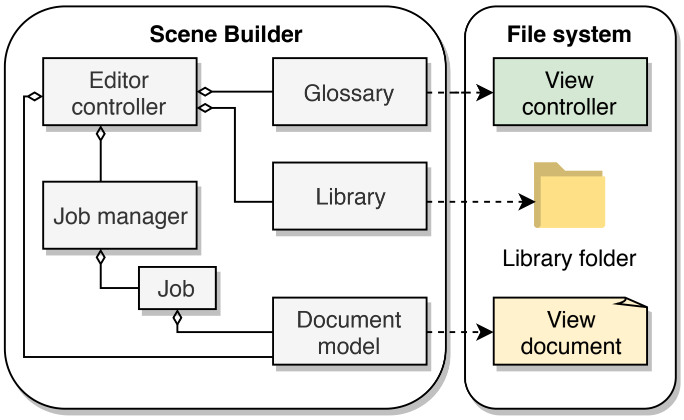


Figure 2.4: The key components of the Scene Builder system, and their relations to the rest of the file system, indicated by the dotted lines in the diagram. The *Glossary* inspects the view's controller to generate the suggestions shown in various input fields. The *Library* manages the collection of items available for insertion from the Library panel, and watches a folder from which components are automatically added to the item collection. The *Job manager* implements the reversibility of actions, while the *Document model* is a granular model of the view document being edited.

a rich **document model** of the FXML view document it edits. It models the document extensively, from a top-level, root object corresponding to the document in the file system, down to low-level objects that correspond to the attributes of the nodes in the document's scene graph. The *text* property of a Button is for example modelled as an object — clearly, this model supports granular modifications.

2.3 The Eclipse IDE and platform

Eclipse is commonly used as an IDE for Java application development. Originally developed by IBM, it is now maintained by the Eclipse foundation, who have made it open source.[3] The main application window of the IDE is called the **Workbench**. It consists of the main menu bar, a toolbar, and a number of tiled panes called views and editors, which are collectively referred to as *parts*. All the resources the developer is working on are contained within the **Workspace**, which is a collection of projects as viewed by Eclipse. It is implemented as a directory on the file system, but projects need not physically reside within it to be edited in Eclipse; rather, they can be linked to from the metadata in the folder.

Views are typically used to display information about resources, and to navigate resource hierarchies or global state in the Workbench. Some examples are the Package Explorer view that provides navigation of the resources in the Workspace, and the Problems View that collects and displays all the errors in the Workspace. **Editors** are used to modify resources. They follow an open-save-close modification model, and when the resource an editor is linked to contains unsaved changes, the editor is marked as *dirty*, which is indicated by an asterisk in the editor's tab.

While Eclipse's primary use is that of a Java IDE, it has an extensible architecture that supports the addition of new functionality to the IDE, which need not be directly related

[3] `https://git.eclipse.org/c/`

to pure Java application development. The ensuing section describes how this is done by plugging into the platform.

2.3.1 Plug-ins and extension points: adding functionality to the IDE

Eclipse is best viewed as an extensible platform — rather than an isolated application — to which new functionality can be added to create customized IDE instances that are suited for some domain specific development process. Functionality is contributed to the platform in the form of components called **plug-ins**, which are modular units from which the entire platform is assembled. As illustrated in figure 2.5 from Moir (2014), tools add functionality by plugging in to the platform. Plug-ins are essentially JARs with associated metadata regarding their interaction with the other plug-ins in the platform. A plug-in declares the set of plug-ins on which it depends in its *manifest*, along with the packages it exports for use by other plug-ins. This way, encapsulation is added at the package level.

Plug-ins interact with each other through **extension points**. They describe how one plug-in can extend the functionality of another, through a structured declaration in a schema file. The extension points extended by a plug-in are declared in its *plugin.xml* file. A declaration of an extension to an extension point typically contains the name of the extension point, and a reference to a class implementing the interface specified by the extension point in its declaration. Beyond that, the declaration often includes some metadata. When extending the *editors* extension point, for example, the declaration includes the file extension with which the contributed editor is to be associated. The platform specifies many extension points that plug-ins can extend to contribute to the platform, and participate in various processes that occur in response to developer actions. A plug-in can also define its own extension points to which other plug-ins can contribute their extensions.

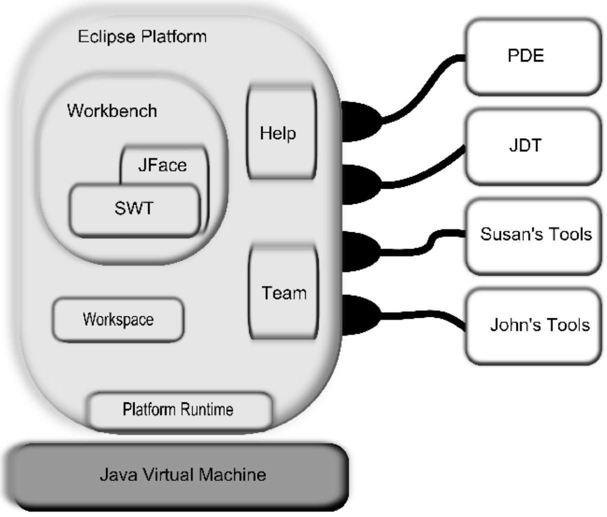


Figure 2.5: An overview of the Eclipse platform, illustrating how external tools *plug into* the platform to extend it with functionality (from Moir (2014)). The collection of plug-ins is structurally homogeneous, which means that external plug-ins are treated no different from the platform's internal plug-ins.

2.3.2 Problem mechanism

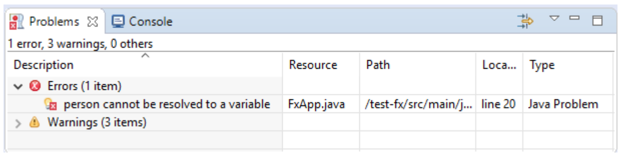
Eclipse has an extensive problem reporting and resolution mechanism that is built around two concepts: *markers* and *quick fixes*. Markers are used to indicate problems to the developer, and quick fixes are used to fix them. When the Eclipse compiler detects an unresolved type reference in a class file, for example, it attaches a marker to the affected class, indicating that the class file can not be compiled. The Java editor then recognizes the marker, and creates a visual representation of it in the form of a red, squiggly line beneath the unresolved type reference, as well as an indicator in the editor's marker bar. This highlights the important distinction between markers and their visual representation. Markers, as such, are pure metadata, and can be accessed and represented in any way imaginable. Consequently, they can also be manipulated without any consideration of their presentation.

Many problems are easy to fix, even when their effects are as severe as compilation failures. Consider again the example of an unresolved type reference. If the type is included in the standard library, such as a `java.util.List`, the fix is as simple as adding the import declaration for that type to the class. Seeing as such errors are both common and mundane, Eclipse has support for fixing them automatically with a single action that can be invoked by the developer. These actions are called quick fixes.

Problem reporting - markers

Markers are, in general, used to annotate specific locations within resources in the Workspace with metadata. Think of them as page markers in a book — attached to a certain page, they can also have text written on them, and their colors may have meanings attached to them given some kind of categorization scheme. Their three main uses in Eclipse are for *problems*, *tasks* and *bookmarks*, but this discussion focuses mainly on problem markers, as it is the type relevant for facilitating JavaFX application development. Eclipse gathers and displays all the problem markers in the Workspace in the Problems View, which is shown in figure 2.6. Problem markers have a severity level of either *information*, *warning* or *error* that indicates the consequences of the problem they represent. The severity levels are described in table 2.1, along with examples of them.

Severity is only one of the many attributes associated with a marker. Others include the path to the resource to which it is attached, the offending line number in that resource (if applicable), and a message intended for the developer. Plug-ins can contribute custom marker types to the platform via the `org.eclipse.core.resources.markers` extension point. Marker types are organized in a hierarchy, so custom markers can inherit from other types in the platform by declaring them as their super type. Custom markers can define



1 error, 3 warnings, 0 others

Description	Resource	Path	Loca...	Type
Errors (1 item)				
person cannot be resolved to a variable	FxApp.java	/test-fx/src/main/j...	line 20	Java Problem
Warnings (3 items)				

Figure 2.6: Eclipse's Problems view, showing an overview of all the problem markers in the workspace. They can be grouped by severity, as shown here, or various other attributes. Double clicking a problem entry opens the associated resource in an editor, with the offending line highlighted if possible.

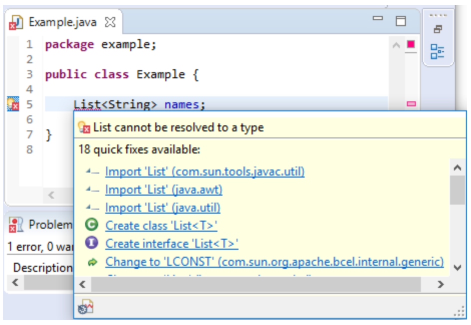
Table 2.1: The three severity levels of problem markers in Eclipse, used to indicate the potential consequences of the problems they represent.

Severity level	Description	Example
Information	Pure metadata	Very rarely used
Warning	A problem whose consequences are either mundane, circumstantial, or both	An unused import declaration
Error	A problem that prevents successful compilation	An unresolved type reference

arbitrary attributes whose names are `String`s, and values are either `String`s, `Integer`s or `Boolean`s. Plug-ins can manipulate markers by creating and removing them, setting their attributes, and querying resources for them. This is done through the `Resources` API of the platform (Eclipse Foundation, 2019, Platform Plug-in Developer Guide>Programmer's Guide>Resources overview).

Problem resolution - quick fixes

The true value of Eclipse's problem mechanism is realized by its mechanism for resolving the problems that have been reported. Their resolutions are offered in the form of one-shot actions that can be triggered by the developer. These are called quick fixes, and are accessible in a variety of ways, chief among which being from a pop-up window that is shown when hovering the mouse pointer over the red, squiggly line in the editor. An example of such a pop-up instance is shown in figure 2.7.



```
package example;

public class Example {

    List<String> names;

}
```

List cannot be resolved to a type

18 quick fixes available:

- Import 'List' (com.sun.tools.javac.util)
- Import 'List' (java.awt)
- Import 'List' (java.util)
- Create class 'List<T>'
- Create interface 'List<T>'
- Change to 'LCONST' (com.sun.org.apache.bcel.internal.generic)

Figure 2.7: Pop-up window showing the quick fixes suggested by Eclipse as remedies to a given problem that was detected during compilation. The window is shown by hovering the mouse pointer over the problem indicator in the Java editor. Clicking a quick fix hyperlink performs the selected fix on the affected resource.

Recall the example of a missing import declaration in a class. It is a typical example of a problem whose consequences are fatal, but resolution trivial. Eclipse has several similar, built-in quick fixes that resolve a variety of issues related to Java application development. While an extensive overview of them can be found in the documentation (Eclipse Foundation, 2019, Java development user guide>Reference>Available Quick Fixes), and is not relevant for this thesis, a few examples of them are given below to outline their variety in scope, as well as complexity:

- Creating methods for unresolved method references (in both the class that is being edited, as well as others)
- Removing unneeded catch blocks
- Changing project JRE compliance

Quick fixes are created by either *quick fix processors*, which are specific to markers on Java classes, or by the more general *marker resolution generators* that are applicable for markers on all kinds of resources. As shown in figure 2.8, the platform queries such processors when computing the quick fixes for a given problem. The processor creates a fix that solves the problem, and hands it back to the platform. Such a fix is abstract in nature, and can perform manipulations on both the resource tagged by the marker, as well as general workspace operations. When the developer invokes the quick fix at a later stage, the platform executes the code in the fix, and the marker of the problem targeted by the fix is removed.

Plug-ins can contribute custom quick fix processors to the platform by extending the `org.eclipse.jdt.ui.quickFixProcessors` extension point. The marker type for which the processor can provide fixes is specified in the extension declaration, along with a pointer to a class that implements the `IQuickFixProcessor` interface. When the platform looks for fixes for a given marker type, it queries only the processors registered for that type. In summary, plug-ins can define custom, domain specific problem types, and quick fix processors to fix these problems. While this is useful in theory, plug-ins also need a way to analyze and manipulate the resources and code of a project. The means Eclipse provides for this type of interaction is described next.

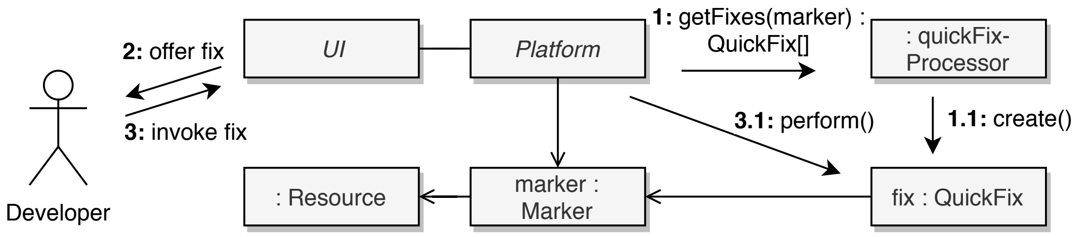
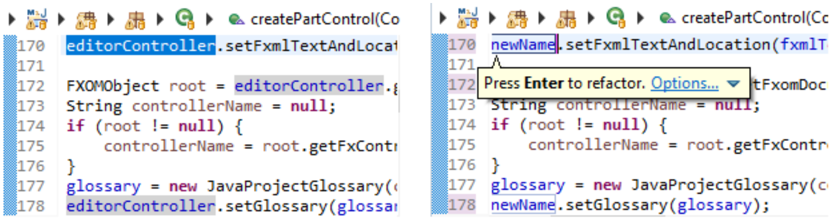


Figure 2.8: Quick fixes are created and offered to the developer via quick fix processors. Note that the QuickFix object used here is a simplified representation of a quick fix, whose implementation may vary greatly. Note also the abstract nature of the offer and invoke fix messages.

2.3.3 Refactoring support

Fowler (2002) defines a **refactoring** as *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.* Eclipse supports a variety of these kinds of changes implemented as one-shot, executable commands placed in a standardized framework that makes them eligible for preview in wizards and preemptive error-checking. An overview of the supported commands can be found in the documentation (Eclipse Foundation, 2019, Java development user guide>Reference>Refactoring), but an illustrative example of its use is to rename all variable references in a program simultaneously, as shown in the screenshots depicted in figure 2.9.

The refactoring mechanism is — as most of the mechanisms in the platform — extensible, and plug-ins can participate in refactoring processes by extending extension points for various processes such as *copy*, *rename*, and *move*. Contributed classes implement various methods to hook into refactorings. The most important ones are `RefactoringParticipant.initialize(Object)`, wherein the participant receives its handle to the element subject to refactoring (a method declaration, for example), and `RefactoringParticipant.createChange`, where the `Change` object representing the refactoring is instantiated.



```
editorController.setFxmlTextAndLocat

FXOMObject root = editorController.g
String controllerName = null;
if (root != null) {
    controllerName = root.getFxContr
}
glossary = new JavaProjectGlossary(
editorController.setGlossary(glossar
```

```
newName.setFxmlTextAndLocation(fxmlT

                                    tFxomDoc
String controllerName = null;
if (root != null) {
    controllerName = root.getFxContr
}
glossary = new JavaProjectGlossary(c
newName.setGlossary(glossary);
```

Figure 2.9: An exemplary usage of Eclipse's refactoring support to rename all references to a variable in a class simultaneously.

2.3.4 Java Development Tooling

In addition to its programming language-agnostic core, Eclipse has a rich model of the Java projects in its Workspace called the **Java model**. It is comprised of objects associated with creating, editing and building Java programs, arranged in a tree structure (Eclipse Foundation, 2019, JDT Plug-in Developer Guide>Programmer's Guide>JDT Core). Figure 2.10 shows how the objects in the model map to the various artifacts of a Java project, as displayed in the Package Explorer view. All `IJavaProject` elements are children of the root `IJavaModel` element that represents the entire workspace. Access to model elements is provided through the `JavaCore` class from the `org.eclipse.jdt.core` plug-in. The root `IJavaModel` element handle can for example be obtained by calling the `JavaCore.create(IWorkspaceRoot)` method.

It is by utilizing the capabilities provided by the Java model that plug-ins can perform static analysis on the Java source code in the workspace, and manipulate it to support application development. It enables a wide range of possible quick fix implementations. Fields, methods and imports can for example be easily added to a class programmatically. It is possible by virtue of Eclipse's fine-grained model of Java classes, represented by the `ICompilationUnit` class. It models the field and method declarations of a class as objects, as seen in the class diagram of the model (figure 2.11).

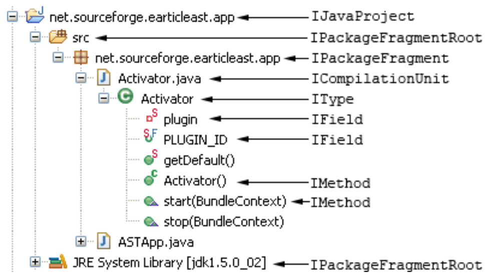


Figure 2.10: The mapping from objects in the Java model to artifacts of a Java project, as shown in the Package Explorer view (from Kuhn and Thomann (2006)).

Eclipse keeps track of the changes that are made to elements in the model throughout its life-cycle. Clients interested in these changes can register as listeners to the model, and receive change notifications through the `IElementChangedListener` interface. The notifications are given in the form of `ElementChangedEvents`. These objects contain rich information about the changes, the most important of which being the associated *Java element delta* that provides access to an hierarchical, fine-grained overview of a modification represented as a tree of deltas, and a flag indicating the type of change (add, change, or remove). When a change is made to a class file, for example, an `ElementChangedEvent` is broadcasted with a delta tree whose root delta is associated with the main `JavaModel` object. Traversal of the broadcasted delta tree eventually leads to the class' corresponding `CompilationUnit` object.

Even though the members of a class are modelled with `IFields` and `IMethods`, and these can be manipulated programmatically, it is sometimes necessary to go further in the analysis of classes. As the Java model is intended to be a lightweight representation of the workspace for performance reasons (Aeschlimann, Bäumer and Lanneluc, 2005; Kuhn and Thomann, 2006), Eclipse provides a detailed model of classes that is contained separately from the main Java model, described next.

Abstract Syntax Trees

The Java model models the workspace from the root model object all the way down to a class' fields and methods, but no further. Abstract Syntax Trees (ASTs) can be used when a more fine-grained model of a class is needed, as they model it down to its nuts and bolts. With ASTs, classes are represented as a tree of nodes, where each node corresponds to an element in the Java language. An example of an AST is shown in figure 2.12, where the relationship between its nodes and the elements of the class it represents is highlighted. ASTs can be analyzed by `ASTVisitors` that traverse their nodes. This is a technique known as the *visitor* pattern (Martin, 2002), wherein visitors implement various `visit(ASTNode)` methods that are called during a traversal of the tree.

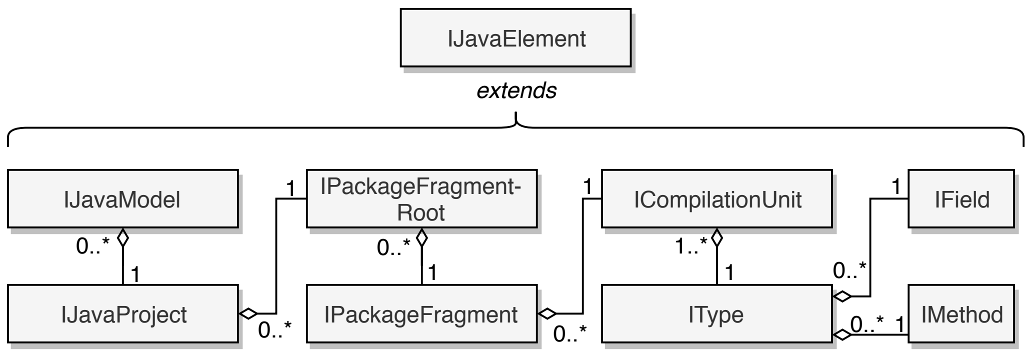
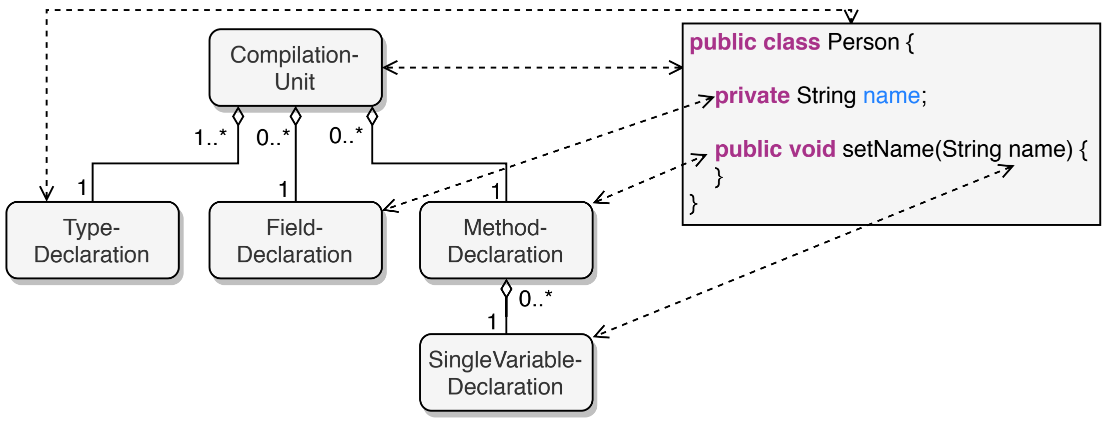


Figure 2.11: Class diagram of Eclipse's Java model. Each association is a parent-child association. All of the interfaces in the diagram extend the high-level, generic `IJavaElement` interface.



```
public class Person {
 private String name;
 public void setName(String name) {
 }
}
```

Figure 2.12: An example of an AST, alongside the class it models. The links between its nodes and their corresponding Java elements in the class declaration are highlighted by the dotted arrows.

3 Requirements identification

In this chapter, I identify various means by which the Eclipse IDE can facilitate JavaFX application development, given what has been established about the connection between views and their controllers in the JavaFX framework, and the mechanisms provided by the Eclipse platform for source code analysis and manipulation. I begin by giving a recapitulation of the view-controller links in section 3.1. Sections 3.2, 3.3, and 3.4 detail the consequences of discrepancies in each of the link types, and identify ways a developer tool can help prevent them, stated as implementation requirements. Section 3.5 summarizes the findings.

3.1 Overview

The high-level goal is to assist developers with maintaining the correctness of the links between a view and its controller. Recall that there are three types of such links:

1. The high-level association between the view and its controller, set in the *fx:controller* attribute of the scene graph's root node.
2. Component instance references (CIRs) from field declarations in the controller to component instances in the view, declared in *fx:id* attributes.
3. Assignments from various component events to their respective event handlers in the controller, set in various *event handler* attributes.

These links are modelled by the class diagram in figure 3.1, which shows the connection between Eclipse's Java model of classes in projects contained in the workspace, and the FXML document model of JavaFX views. The dotted lines connect the elements of the models whose names have to match for a functioning view-controller interaction in the application at runtime.

3.2 High-level association

To use a view as part of a JavaFX application's GUI, the view document must be loaded programmatically at some point in the program. This is done with the use of an FXML document loader, whose `load` method returns an object instance representing the root of the scene graph declared in the document. If the loader is unable to locate the specified controller for a view — when there is a typo in the entered name, for example — the application crashes during the loading process due to a `java.lang.ClassNotFoundException`. When considering that this is not encountered until runtime, we arrive at the first instance where the integration can assist developers. If the integration can search the project containing the view for its associated controller upon assignment, and conclude that it can not be found, a warning should be displayed to the developer, informing them that the view can not be loaded.

How should such a warning be presented? From what has been established regarding Eclipse's problem mechanism, it is clear that an unresolved controller reference warrants a problem marker attached to the view document. Indicating the marker in the Scene Builder editor can also be considered. Recall that in it, the association is set from an input field

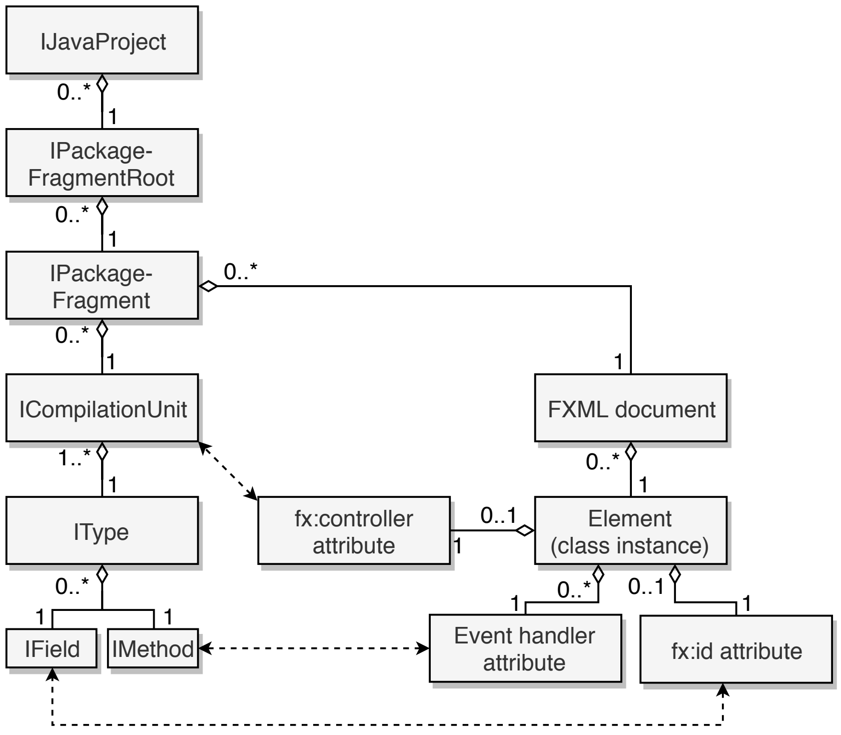


Figure 3.1: The connection between Eclipse's Java model of classes in the workspace and the FXML document model of views in JavaFX. The model elements whose contents must cohere if the interaction is to function correctly are connected by the dotted lines.

for the controller name in the Document panel. One option can for example be to show a warning in the field when an *Unresolved controller* marker on the view document is detected. Implementing such a warning entails the inclusion of a custom `InfoPanelController`, with a custom input field implementation that is aware of any markers on the editor's document.

It should also be considered if it is possible to detect the point in the program where the view is loaded, and put a problem marker there, as well. This is technically challenging in the general case, however; while discovering the calls to `FXMLLoader.load` is a straightforward process as such, inferring the exact location it is set to load from is not, as the Uniform Resource Locator (URL) could very well come from a variable, whose value is not resolved until runtime.

A quick fix that adds the missing controller to the project should be offered. It can also be imagined that the controller already exists, in which case a quick fix that changes the reference to the existing controller could be offered. Care must be taken to correctly identify the controller, however, in a manner similar to that used to suggest controller candidates. Support for maintaining the association if the controller is renamed should also be added, by hooking into the refactoring process and updating the reference in the view to the new name of the controller.

3.2.1 Requirements

From the preceding discussion, we arrive at the following ways the high-level association should be maintained:

- Unresolvable controller references in views should be detected and reported. A problem marker of the *Error* severity level should be added to the view document, with a message saying that the controller can not be found.
- A quick fix that adds the controller to the project should be offered.
- If the controller is renamed using Eclipse's refactoring mechanism, the reference in the view should be updated in accordance with the new controller name.

3.3 Component instance references

For the FXML document loader to be able to inject the component instances declared in the view document into the controller's field declarations, their names have to match. Note that, unlike the main, high-level association that points from the view to the controller, CIRs go in the opposite direction. This is reflected in the direction of the unresolved references that can cause problems: While the application crashes if the `fx:controller` attribute points to a missing controller, dangling `fx:id` attribute values — i.e., *id*s declared in the view that are not referenced by any controller field declarations — do not cause any harm. Controller field declarations that lack counterparts in the view, however, can cause subtle issues. The application is able to launch without problems in spite of such declarations, but the instances are not injected into the fields, which remain uninitialized. If the fields are accessed at some point in the program's execution, it runs into a null pointer exception, with potentially fatal consequences.

CIRs in the controller that lack corresponding `fx:id` declarations in the view should be reported with problem markers of the *warning* severity level attached to the controller, as it means that they will remain uninitialized at runtime. The markers should reference the lines of the offending field declarations to further guide the developer. It could be argued that such markers should have the *problem* severity level, instead, since any attempts to access the uninjected field will result in null pointer exceptions. It is, however, not possible — in the general case — to determine whether the field will be accessed at runtime using static analysis. Consider, as an example that emphasizes this difficulty, a field access from within an event handler. Whether the field will be accessed at runtime depends in this case on the user's interaction with the application, and can not be inferred using static analysis during compilation. Markers could arguably also be added to every point in the code that accesses uninjected fields, despite the uncertainty regarding the control flow.

It is not possible to provide a useful quick fix for unresolved CIRs in the general case. Consider the case where the view does not contain a corresponding component — then, a quick fix should not simply insert a component into the view at an arbitrary position in its scene graph. Such a quick fix would arguably do more harm than good. It could be argued that if the view contains one or more corresponding components that lack `fx:id` declarations, quick fixes should be offered that add a corresponding `fx:id` to one of the components. Such fixes are not easily represented to the developer in any meaningful way, however, as there is no obvious way to generally distinguish between for example two `Button` components at different places in the scene graph.

A quick fix that could be explored is one that renames an unresolved field if it closely resembles an `fx:id` in the view. This would require a computation of the *edit distance*[1] between the references, whose computational cost could outweigh its usefulness — after all, the cases where it would be applicable are arguably quite rare. Determining the distance threshold at which two references should be considered too far apart to be likely caused by a typo presents another problem, with no general case solution. Given that a CIR is correct, support for refactoring should be provided; if the field is renamed, it should be reflected in the component's `fx:id`.

3.3.1 Requirements

In conclusion, the following steps should be taken to prevent unresolved component references:

- Unresolved CIRs should be detected and reported with problem markers of the *Warning* severity level attached to the controller class, pointing to the line numbers of the offending field declarations.
- If a resolved CIR is renamed using refactoring, the change should be reflected in the component's `fx:id` in the view document.

3.4 Event handler associations

To direct the application's runtime control flow to method declarations in the controller when events are fired from components in the view, the declarations need to be referenced from event handler attributes of the components. Note that the direction of the association — from view to controller — is opposite to that of CIRs. As the program control flow is directed *unconditionally* to controller event handlers in a running application, the FXML document loader will raise an exception when attempting to load a view with unresolved event handler references. Hence, missing event handlers in the controller should be reported with a problem marker attached to it, of the *Error* severity level. A similar marker should also be attached to the view document.

A quick fix for such markers that adds an empty method declaration with the appropriate name to the controller should be offered. If the event handler is already present in the controller, but with a mismatch between its name and that declared in the view, a rename quick fix could also be considered, similar to that discussed previously. However, the issue of the edit distance threshold determination is also relevant here. Refactoring support should be implemented for resolved event handlers, in the form of renaming the reference in the view.

3.4.1 Requirements

To conclude, the following steps can be taken to maintain the coherence of event handler references:

- Declared event handlers in the view that lack corresponding method declarations in the controller should be detected and reported as problem markers attached to both the controller and the view, with the *Error* severity level.

[1] The edit distance between two strings is a measure of their similarity represented as the number of steps required to edit one of them to match the other.

- When resolved event handlers are renamed using Eclipse's refactoring mechanism, the change should propagate to the view.

3.5 Summary

Table 3.1 contains an overview of the three link types, along with the requirements identified for each of them in the preceding discussions. These will serve as guidelines for the implementation detailed in the following chapter.

Table 3.1: Various attributes of the links between controllers and views in the JavaFX framework, including descriptions of how Eclipse can assist with maintaining the correctness of the links.

	Link type		
	High-level association	Component instance reference	Event handler reference
Direction	View to controller	Controller to view	View to controller
FXML attribute	*fx:controller*	*fx:id*	*onAction*, *onMousePressed* etc.
Discrepancy consequences	Crash during loading	Field is not injected	Crash during loading
Problem marker severity	Error	Warning	Error
Quick fix	Add controller to project	Rename to *fx:id* in view, if possible	Add method to controller
Refactoring support	Update controller reference in view	Update *fx:id* in view	Update event handler reference in view

4 Design and implementation

This chapter details the steps taken to implement the functionality outlined in the previous chapters in a technical manner. An overview of some of the implementation of the initial integration is first given in 4.1 for completeness. The implementation of the initial integration is described in greater detail in my specialization project report (Ask, 2018). The extension of Scene Builder's connection to the view document's project — along with its corresponding ability to provide suggestions to the developer — is subsequently described in sections 4.2 and 4.3. Last, the implementation of the view-controller coherence assistance functionality is presented in section 4.4.

4.1 Integration of the Scene Builder panels

The integration extracts Scene Builder's Content, Library and Document panels and embeds them in an Eclipse editor for FXML documents. Scene Builder's Inspector panel is extracted to an Eclipse view. While most of the work behind the extraction was done as part of my specialization project (Ask, 2018), the key implementation details of these two components are described next to make the thesis independent of that report.

4.1.1 The FXML Editor

Scene Builder's Content, Library, Hierarchy, and Controller Panels have been embedded in an Eclipse editor, hereinafter referred to as the *FXML Editor*. It is contributed by extending the `org.eclipse.ui.editors` extension point defined by the platform. Plug-ins declaring extensions to the extension point are required to provide implementations of the `org.eclipse.ui.IEditorPart` interface, of which there are some abstract base implementations — with generic behavior deemed useful for most editors — available in the platform. The FXML Editor subclasses the `EditorPart` class, which takes care of a lot of the required wiring with the workbench. This choice comes with the requirement that its constructor be parameter-less. While this does not impose direct restrictions on the editor's behavior, it does limit its testability.[1] The following methods are left for subclasses to implement:

`void init(IEditorSite, IEditorInput):` called by the platform shortly after the editor instance is constructed. The platform will not display the editor's GUI contents at this point; it simply hands the editor its input. In this method, the FXML Editor performs some basic initialization of members, and extracts the URL from the `IEditorInput` object representing the FXML document to an instance field.

`void createPartControl(Composite):` called by the platform when it is ready to display the graphical contents of the editor. Here, the FXML Editor proceeds by initializing the various Scene Builder components required for a functioning editor. An FXML document model is constructed from the previously extracted URL of the input document, and gets associated with an `EditorController` (EC) instance. This instance gets connected to the panel controllers that constitute the editor's GUI.

[1] For the purpose of unit testing, it is generally preferable that objects receive their dependencies in their constructors, as it allows for an easy replacement of the dependencies with *mock* objects in the tests. This approach requires the use of a dependency injection container in the running system, however, which introduces an extra layer of complexity to the program.

`boolean isDirty()`: called by the platform to query the *dirty status* of the editor, and with that determine whether or not to display an asterisk in the editor's title bar. It is up to the editor to notify the platform that this method should be called, however, by means of updating its *dirty property*, to which the platform is registered as a listener. The FXML Editor maintains its dirty status by keeping a copy of the most recently saved FXML document as a `String`, and comparing it to the `String` representation of Scene Builder's document model whenever it changes. Notifications about changes to the model are given to the editor since it registers as a listener to the revision property of Scene Builder's `JobManager` in the `createPartControl` method.

`void doSave(IProgressMonitor)`: called by the platform when the user initiates a *Save* command. The FXML Editor's implementation overwrites the contents of its input document with that of Scene Builder's document model. It also makes sure to update the editor's local `String` copy of the document so as to be able to correctly maintain the dirty status.

4.1.2 The Inspector View

Scene Builder's Inspector Panel is embedded in an Eclipse view. In order for it to correctly display its content — the attributes of the view component selected in the active FXML Editor — it needs to link with the editor's EC. As the panel did not support this kind of updatable reference to the EC in its original form, it needed an alteration to fit the needs of the integration (Ask, 2018). The Inspector View utilizes the *part service* in the workbench to maintain the connection to the FXML Editor that is active in the workbench (if there is one). Clients of the service can register as listeners to receive notifications about part life-cycle events that occur as results of the developer opening, switching between, and closing parts. Notifications about the following events are broadcasted: *partActivated*, *partBroughtToTop*, *partClosed*, *partDeactivated*, and *partOpened*.

When registered as a listener to the part service, the Inspector View receives notifications about the life-cycle events of all the parts in the workbench; not just FXML Editors. This means that care must be taken to ensure that it updates its EC reference correctly in response to the various events. In particular, the order of the events can be misleading. Consider, as an example, a scenario where two parts (Part A and B) are open. When the developer closes one of them (Part A), and the other one (Part B) is consequently activated, the following sequence of events is broadcasted to part listeners (the part the event notification concerns is in parentheses):

1. Part deactivated (Part A)
2. Part activated (Part B)
3. Part closed (Part A)

Updating the reference when the FXML Editor plays the role of Part B in such a scenario is a straightforward endeavour: If the part that the event notification sent to the view concerns is an FXML Editor instance, the view should grab a reference to that editor's EC, and override its own EC reference with the grabbed one. It is when handling such scenarios where the FXML Editor acts as Part A, however, that an implementation choice must be made. When should the view clear its content — i.e., its component selection? It may seem appropriate to clear it in response to the deactivation event, but as the view needs to

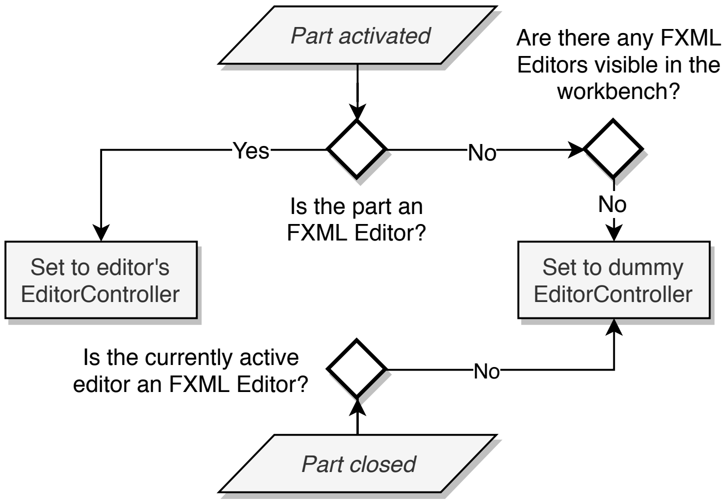

Part activated
Are there any FXML Editors visible in the workbench?
Yes
No
Is the part an FXML Editor?
No
Set to editor's EditorController
Set to dummy EditorController
Is the currently active editor an FXML Editor?
No
Part closed

Figure 4.1: Flowchart illustrating how the Inspector view updates its `EditorController` reference — and with that its content — in response to part life-cycle event notifications received from the workbench's part service. The dummy `EditorController` is an empty instance that is used to clear the view content. Note that the events (*part activated* and *part closed*) occur at separate times. Whether or not there are visible (or active) FXML Editors in the workbench is determined by using the API provided by `org.eclipse.ui.IWorkbenchPage`.

update its EC reference to a dummy instance to clear its content, such an approach would entail two subsequent updates of the panel's EC reference, which is a reasonably complex operation in itself.[2]

The final consideration to consider arises in the case where FXML Editor instances play the roles of both Part A and B in the scenario. Then, the view's response to the *part closed* event should *not* be to set its EC reference to the dummy one, as that would override the reference just set in response to the activation event. Rather, it should only clear its content in such cases when the part that is currently active is not an FXML Editor. A flowchart illustrating the view's responses to the *part closed* and *activated* event notifications is shown in figure 4.1. Note that it is not the only way to achieve the correct behavior, but rather the one deemed the simplest, yet correct, implementation. As is often the case when dealing with object state coherence, there are many choices of implementation, and just as many pitfalls.

4.2 Extending Scene Builder's Glossary

Scene Builder's Glossary component implements the system's connection with the controller. It is responsible for computing the suggestions shown in the controller class input field in the Document panel, as well as the ones shown in the fields for a component's `fx:id` and event handler specifications, found in the Inspector panel. The details of the computation of these suggestions are given next. How the integration improves on the default implementations by leveraging the Java model is also presented.

[2] The process entails addition and removal of a fair amount of property listeners.

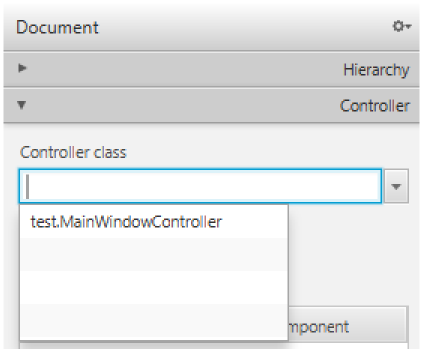


Figure 4.2: The input field in the Controller Panel for the name of the controller assigned to the view, showing suggestions of candidate controllers. The candidates are computed by scanning the directory in which the view is located for Java files, and filtering them based on various selection criteria such as file names. Both the scanning and the selection algorithm can be modified by providing a custom `Glossary` implementation.

4.2.1 Controller candidate suggestions

Scene Builder's Document panel has an input field for the name of the controller associated with the document (the main, high-level association indicated by link 1 in figure 2.1). As seen in the screenshot of the field, depicted in figure 4.2, it offers suggestions of candidate controllers to the user, based on the results of a scan of the directory in which the view document is located. A candidate controller is a class that is flagged by the scan as a potential controller for the view, according to some criteria. The scan is performed by the `Glossary` at the request of the Controller panel during its initialization, as shown in figure 4.3. `Glossary` is an abstract class, and the implementation of the scanning algorithm is left for subclasses to provide.

Scene Builder comes with a default implementation that performs the scan in the following way: First, the source folder to scan is determined. The algorithm is able to detect if the view document is part of a Maven project, and scans the `src/main/java` folder instead of the `resources` folder — where the document is located — should that be the case. Next, the algorithm identifies Java files as candidate controllers based on whether they meet one of the following criteria:

1. Its file name matches that of the document
2. Its file name matches that of the document, followed by the *Controller* postfix (e.g. `MainWindowController`)
3. It either a) has one or more `@FXML` annotated members, or b) implements the `Initializable` interface

While useful, the default `Glossary` implementation suffers from a couple of weaknesses. For one, it can only suggest candidates in projects that either use the Maven project structure of `src/main/[java/resources]`, or the default single source folder structure. The extent to which this is actually a problem might be limited, though, as the Maven project structure is very common. Its second weakness is that when the algorithm is considering the third of the aforementioned criteria, it relies heavily on the use of regular expressions to parse the Java file content. Catching all use cases with such expressions is not an easy task. These expressions are also quite intricate, and thus highly error-prone should there be any changes

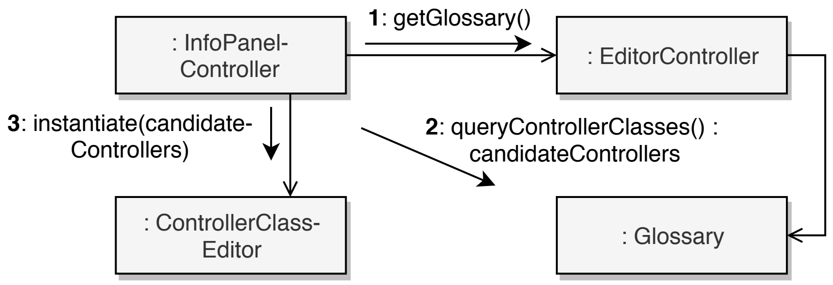


Figure 4.3: The process in which the candidate controller suggestions shown in the Controller panel are acquired. During its initialization, the `InfoPanelController` queries the `Glossary` about potential controllers in the same project as the document being edited. The objects do not have a direct relation to each other; rather, they are connected through the `EditorController`. This means that `InfoPanelController`'s behavior can be altered from the outside by handing the `EditorController` a custom `Glossary` implementation. The suggestions computed by the `Glossary`'s scanning algorithm are handed to the `ControllerClassEditor`, which contains the implementation of the input field depicted in figure 4.2. Note that this process only occurs during the initialization of the system, at the `InfoPanelController`'s request.

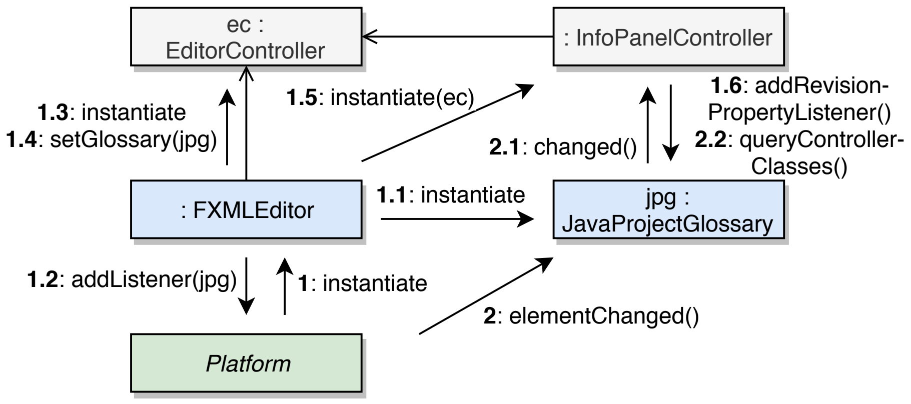


Figure 4.4: The initialization of the custom `Glossary` implementation provided by the integration. After the object is constructed, it is associated with the `EditorController` used by the `InfoPanelController`, which adds itself as a listener to the revision property. When a change occurs in the workspace at a later stage (initiating the second call flow in the diagram), the `JavaProjectGlossary` receives a change notification it examines to determine whether it should trigger an update of the candidate controller list. If that is the case, it updates its revision property, and the `InfoPanelController` is notified. This mechanism ensures that the controller candidate suggestions stay relevant throughout the lifetime of the system, unlike the default mechanism.

to the specification. While the Java Language Specification is not likely to be subject to change any time soon, the JavaFX language specification might be. The last, and by far most problematic weakness, however, is that the scan is only performed once, during the initialization of the Controller panel. This means that any modification, addition or removal of files occurring after-the-fact go completely unnoticed by the `Glossary`. All of these weaknesses can be remedied by providing a custom implementation that leverages Eclipse's Java model, as shown next.

The integration provides a custom `Glossary` implementation — `JavaProjectGlossary` — that tracks changes to the workspace by listening to the Java model. The process of its initialization, as well as its interaction with the rest of the system, is illustrated by the communication diagram in figure 4.4. When changes are made to the model as the result of the developer making a change to a class, for example, the `JavaProjectGlossary` is notified about the change by the platform. Further, the abstract `Glossary` superclass has a *revision property* that the `InfoPanelController` can listen to. Thus, when the `JavaProjectGlossary` receives Java model change notifications, it can analyze them to determine whether the candidate controller suggestions should be revised, and update its revision property if that is the case. The `InfoPanelController` is subsequently notified about the update, and queries the `Glossary` for the revised candidate list, which in turn is displayed in the input field. All of this means that the suggestions shown in the Controller panel reflect the state of the project at all times throughout the system's lifetime, instead of only a snapshot of its state at system start-up — as is the case with the default `Glossary`.

4.2.2 Fx:id suggestions

In a similar fashion to the controller class input field, Scene Builder also shows suggestions of `fx:id`s and event handler names in the Inspector panel. They are computed by scanning the view document's controller for any `@FXML` annotated members. The process is similar to that where controller name suggestions are computed, shown in figure 4.3 — the main difference is that `fx:id` and event handler name suggestions are computed each time the selection in the editor changes, which means that they are kept in sync with the controller throughout the lifetime of the system. The `fx:id` extraction mechanism in the default `Glossary` implementation is not completely flawless, however, as the suggested ids are not filtered according to the type of the selected component.

Consider as a minimal example the following controller:

```
public class ExampleController {
    @FXML private Button button;
    @FXML private Label label;
}
```

In the default implementation, *button* and *label* are both suggested as `fx:id`s in the input field of the Inspector Panel when any `Button` and `Label` component without a specified id is selected, which is not what the developer expects. The type of the selected component is in fact included as an argument in the call to `Glossary.queryFxIds`, but since the default `BuiltinGlossary` does not have a model of the controller, it is not able to consider the component type parameter. The custom `JavaProjectGlossary` leverages Eclipse's JDT to analyze the controller's AST when computing the suggestions. When it is queried about the ids in the view's controller by the `InspectorPanelController`, it instantiates an `ASTVisitor`, and initiates its traversal of the controller's AST, in search for id and event handler declarations therein. The process is illustrated by a communication diagram in figure 4.5. In its traversal, the visitor can inspect the type of the field declarations it visits, and is with that able to keep a mapping of the discovered `fx:id` declarations keyed by their type. Consequently, the custom `Glossary` is able to take the component type into consideration when computing the id suggestions, and filter them accordingly.

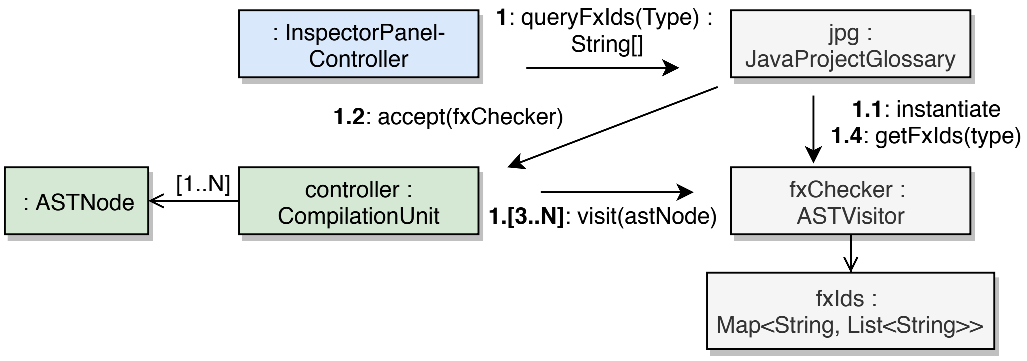

: InspectorPanel-Controller
1: queryFxIds(Type) : String[]
jpg : JavaProjectGlossary
1.2: accept(fxChecker)
1.1: instantiate
1.4: getFxIds(type)
: ASTNode
[1..N]
controller : CompilationUnit
1.[3..N]: visit(astNode)
fxChecker : ASTVisitor
fxIds : Map<String, List<String>>

Figure 4.5: The custom `Glossary`'s process of computing the `fx:id` suggestions that are shown to the developer in the `fx:id` input field, found in Scene Builder's Inspector Panel. When queried about the ids, the glossary inspects the controller by way of AST analysis. The visitor traverses the controller's AST in search of id declarations, which it stores as a mapping keyed by component type.

4.3 Extending Scene Builder's facilities for custom components

As the behaviour of JavaFX components is defined in regular Java code, they can be subclassed to provide components with custom behaviour. However, Scene Builder is unable to open documents that reference such custom components when they are located outside of the source folder in which the view document is located, for reasons explained next. Consider an example project with the following directory structure, where the component implementation and the view document are located in separate source folders:

```
example-app
|-- src
    |-- main
        |-- java
            |-- example
                |-- CustomComponent.java
        \-- resources
            |-- example
                |-- View.fxml
```

The custom component is used in the view, like so:

```
<?import example.CustomComponent>

<CustomComponent>
    // attributes and children...
</CustomComponent>
```

In such cases, the document can not be opened in Scene Builder, because the class loader employed by the FXML document loader in the construction of the system's document model does not have the class that defines the custom component on its class path.[3] By default, the `FXMLLoader` uses the system class loader, whose default class path is set to the current working directory (Oracle Corporation, 2019, ClassLoader). Hence, for the document with the custom component to be correctly loaded, the path to the class implementing it would need to be supplied using either the `-classpath` command-line option when launching the application, or by setting the `CLASSPATH` environment variable — neither of which being particularly convenient alternatives.

The system's class loading mechanism can, however, be easily extended. As shown in figure 4.6, the class loader used to construct the document model can be replaced by a custom subclass through the `Library`. The integration provides such a custom class loader that can load any class on the project's class path. As illustrated by the communication diagram in figure 4.7, it does this by utilizing a method provided by the `JavaRuntime` API in Eclipse's JDT that computes the runtime class path of a Java project, and returns it in the form of URLs pointing to its class path entries. During its construction, the custom loader uses the method to construct an `URLClassLoader` with the URLs of the project's class path entries, and delegates to that in subsequent calls to its `findClass`. With that, the Scene Builder system is extended with the ability to load view documents with custom components defined in classes located outside of the view document's source folder.

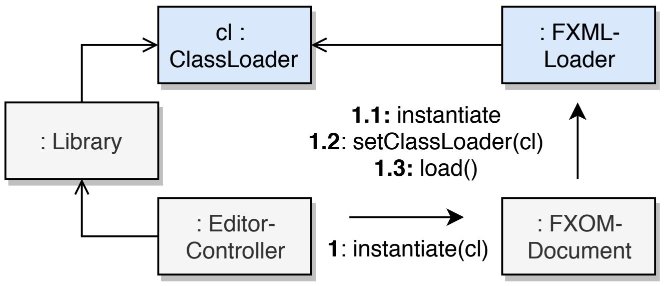


Figure 4.6: The construction of Scene Builder's document model, which entails calls to a class loader. The class loader employed by the `FXMLLoader` to load the document is acquired from the `EditorController`'s `Library`. This means that a custom implementation capable of loading classes outside of the document's source folder can be used, instead of the default one, which lacks such a capability.

[3] The FXML document loader employs a class loader to resolve both the components referred to in the view, and the controller assigned to it.

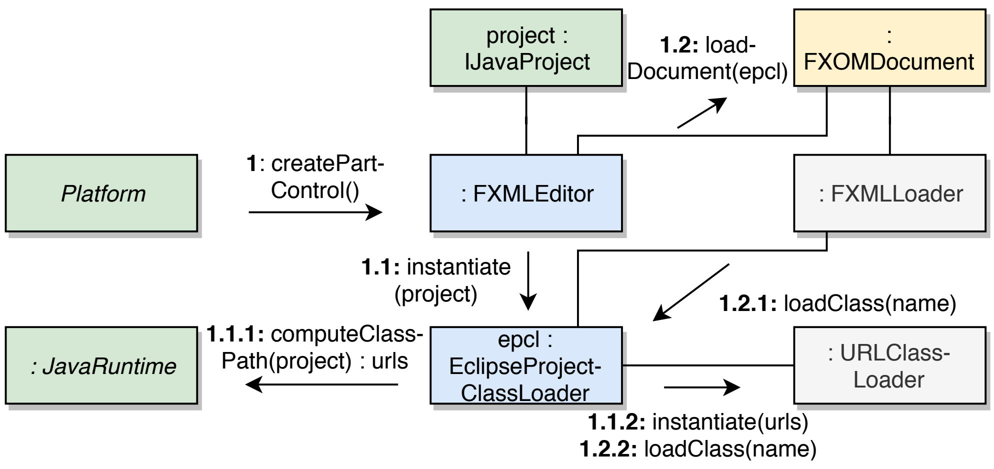


Figure 4.7: The construction and class loading mechanism of the custom class loader that is contributed by the integration (`EclipseProjectClassLoader`), which can load any class found on the class path of the view document's project. During its initialization, the loader constructs an `URLClassLoader` from the URLs of the project — the computation of which is handled by the `JavaRuntime` API from the platform. The loader delegates to the helper `URLClassLoader` in its loading process.

4.4 Detecting, reporting and fixing view-controller discrepancies

Chapter 3 concluded that problem markers should be added to views and controllers if there are discrepancies in the links between them, but a central question remains — how and when should the markers be added? Eclipse's build process is open for participation, and plug-ins can hook into it to receive notification when projects in the workspace are being built. Rich information about the project is broadcasted and can be utilized to analyze views and controllers, in ways described next.

4.4.1 Analysis and marker creation from builder

The plug-in participates in Eclipse's build process by contributing an incremental project builder to the platform, with the responsibility of detecting discrepancies between views and controllers and reporting them as problem markers. An important distinction in Eclipse's build process is that between a full and an incremental build: A **full build** is performed from scratch, and treats all resources in a project as if they have never been seen by the builder. An **incremental build** uses a "last build state," maintained internally by the builder, to do an optimized build based on the changes in the project since the last build (Eclipse Foundation, 2019, Platform Plug-in Developer Guide>Programmer's Guide>Advanced resource concepts). Most of the time, once a project has been opened, incremental builds are triggered. How the builder handles these is described next.

The builder analyzes the project it is associated with by visiting it according to the visitor pattern. When a full build is triggered, the visiting process of a *resource visitor* that analyzes all the project's FXML documents according to the requirements defined in chapter 3 is initiated from the resource corresponding to the project. In its inspection of a document, the visitor starts by checking if the document contains an *fx:controller*

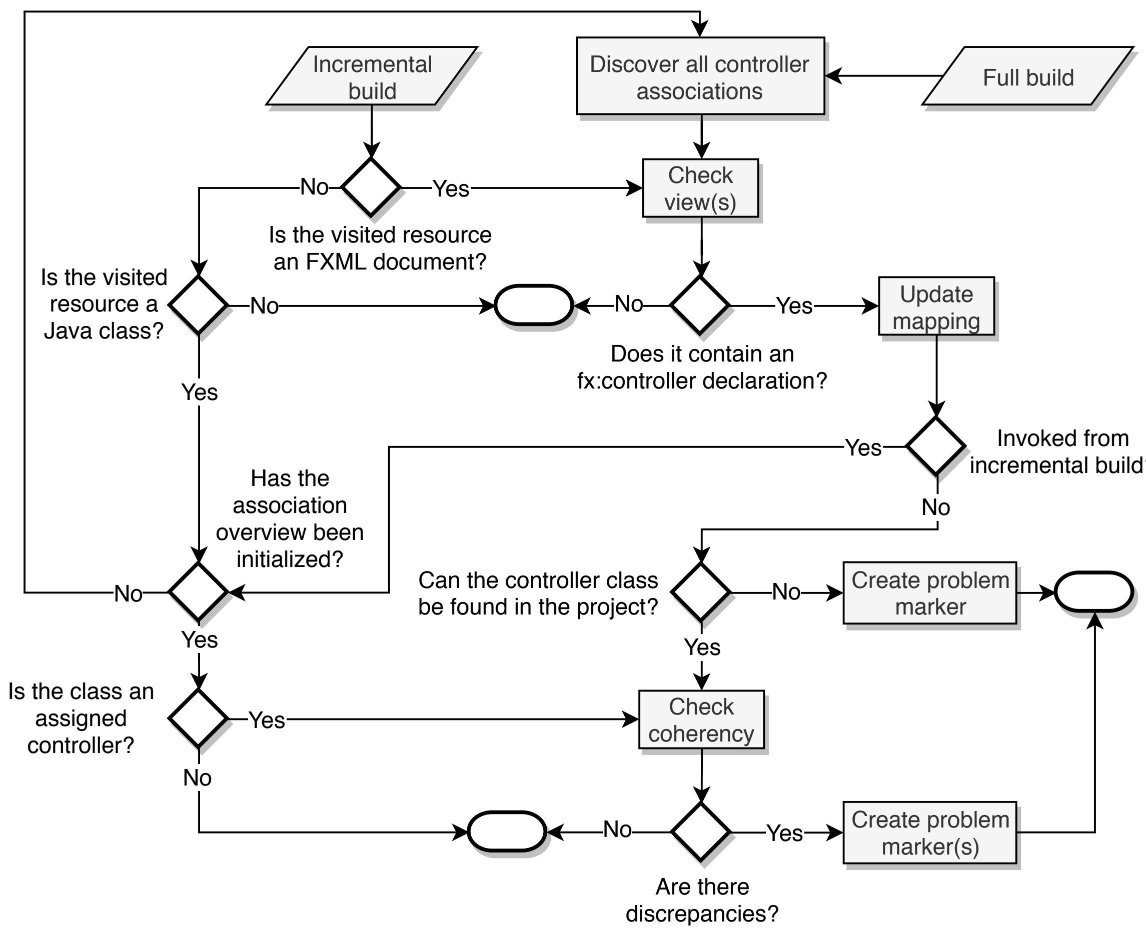

Incremental build
Discover all controller associations
Full build
No
Yes
Is the visited resource an FXML document?
Check view(s)
Is the visited resource a Java class?
No
No
Does it contain an fx:controller declaration?
Yes
Update mapping
Yes
Yes
Invoked from incremental build?
Has the association overview been initialized?
No
No
Can the controller class be found in the project?
No
Create problem marker
Yes
Yes
Is the class an assigned controller?
Yes
Check coherency
No
No
Yes
Create problem marker(s)
Are there discrepancies?

Figure 4.8: Flowchart showing how the builder runs to discover and report discrepancies between views and controllers. Incremental builds are handled by *resource delta visitors*, while full builds are handled by *resource visitors*. Both visitor types delegate to the common validation logic implemented in the builder class. Note that the *Discover all controller associations* action triggers multiple *Check view(s)* actions.

declaration. If that is the case, it proceeds by checking whether the specified controller class can be found in the project by calling a method from the JDT Application Programming Interface (API). If it can not be found, the visitor attaches a marker to the document and ends its analysis there. If the controller is found, however, the visitor instead checks the view and controller's coherency in the *fx:id*s and event handler declarations. A flowchart of this process can be seen in figure 4.8, starting from the *full build* invocation.

Handling incremental builds correctly involves more subtleties. Information about these builds is given in the form of resource deltas. They are visited by a *resource delta visitor*, whose `visit(IResourceDelta)` is called for the changes in the affected resource, which can correspond to either a view or a controller. When it is called for views, an analysis similar to that described earlier can be applied, but when it is called for controllers — in response to changes in a class, for example — there needs to be some information in place for a correct discrepancy check. In particular, the visitor needs to know whether the class it is inspecting is an assigned controller to a view in the project. To determine this, the builder needs to maintain an overview of all the controller associations in the project. A complicating subtlety in that regard, however, is that Eclipse performs full builds very sparingly. Hence, it is possible that the very first call to the builder during its lifetime stems from an incremental build of a controller. If that is the case, a full traversal of the project is needed to construct the association overview, as shown by the flow from the *incremental build* branch of the flowchart to the *Discover all controller associations* action. A communication diagram of the process in shown in figure 4.9.

Eclipse uses the concept of *project natures* to associate builders with projects. Maven projects opened in Eclipse will for example have a *Maven nature* associated with them. Since the platform only runs the declared builders for projects' registered natures when building them, builders can restrict their scope using natures. The plug-in defines a custom *JavaFX project nature*, with which the custom builder is associated. The nature can be added to or removed from projects dynamically from their context menus in the Package Explorer view.

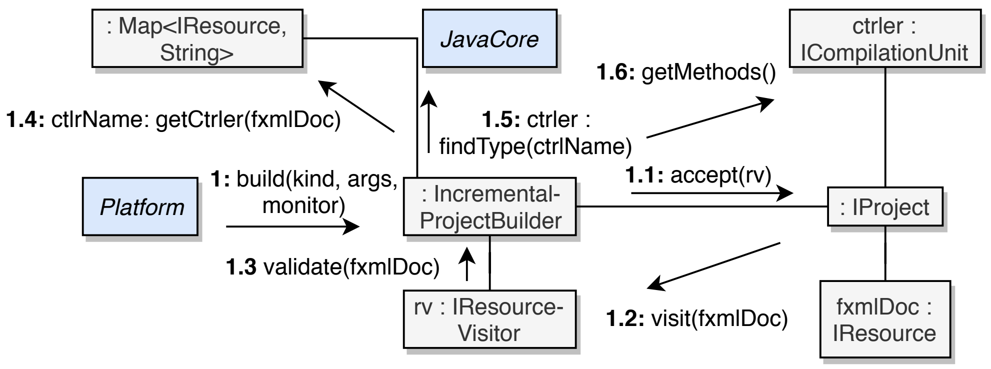


Figure 4.9: The builder's process of validating a view and its controller during full builds. The *kind* argument supplied in the call to the builder's `build` method can be used to distinguish between incremental and full builds. A resource visitor delegates to validation logic in the builder, which uses its association overview (the map from `IResource`s to controller names) to get the view's specified controller name. The builder utilizes `JavaCore` from the JDT to acquire the `CompilationUnit` corresponding to the controller class that is used to perform the validation. If discrepancies are found, they are reported on the controller — a process omitted here for clarity.

4.4.2 Quick fixes

The plug-in contributes a custom marker type to the platform for view-controller discrepancies, and a *marker resolution generator* for fixing such markers. It generates resolutions based on an attribute that signifies the problem type, which is either an unresolved controller, a missing *fx:id*, or a missing event handler implementation. As it is not (arguably) possible to generate meaningful resolutions for missing *fx:id*s (see section 3.3), the generator ignores them, and generates resolutions for unresolved controllers and missing event handlers. The resolution for unresolved controllers opens a dialog that prompts the user to create the controller class in the project. The generation and execution of it is best illustrated with the communication diagram of figure 4.10. When it is executed by the platform, it instantiates a `Wizard` and a `NewClassWizardPage` to show inside it. It calls various JDT APIs to provide pre-defined values for the name of the class, its package name, and its source folder location so that the developer only needs to confirm the dialog to create the class. If the class is created, it is opened in an editor, and the marker is deleted.

The resolution for missing event handler problem markers adds an empty method declaration to the controller. The way this is done is illustrated by the communication diagram in figure 4.11. First, it derives the controller's `IType` from the supplied marker. Next, it extracts the name of the method to be added from an attribute of the marker, and calls `IType.createMethod(String)` that takes a textual representation of the method to be added as its parameter. It also adds the FXML annotation, and adds it to the class' import declarations if needed. Last, the marker is deleted.

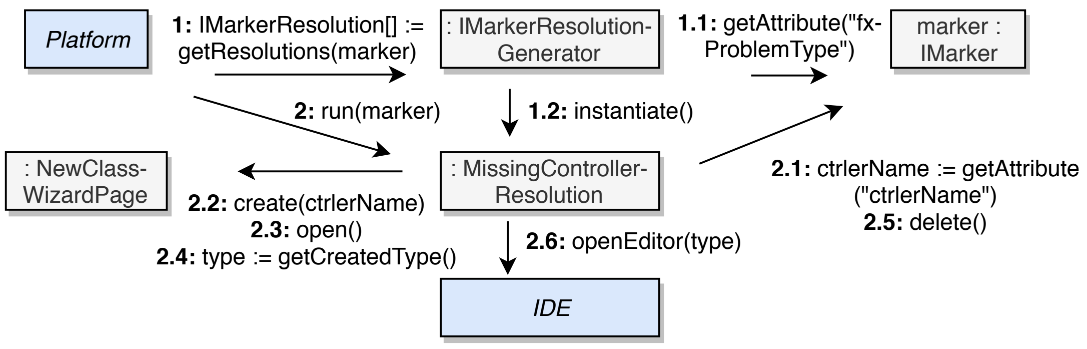

Platform
1: IMarkerResolution[] := getResolutions(marker)
: IMarkerResolution-Generator
1.1: getAttribute("fx-ProblemType")
marker : IMarker
2: run(marker)
1.2: instantiate()
: NewClass-WizardPage
: MissingController-Resolution
2.1: ctrlerName := getAttribute ("ctrlerName")
2.5: delete()
2.2: create(ctrlerName)
2.3: open()
2.4: type := getCreatedType()
2.6: openEditor(type)
IDE

Figure 4.10: The generation and execution of a resolution for an *unresolved controller* problem marker, which opens a wizard dialog prompting the developer to add the controller to the project. If the class is created, it is opened in an editor. Note that it is assumed here that the marker's value of the *fxProblemType* attribute indicates an unresolved controller, which is not always the case, and that some calls that setup the wizard are omitted for clarity.

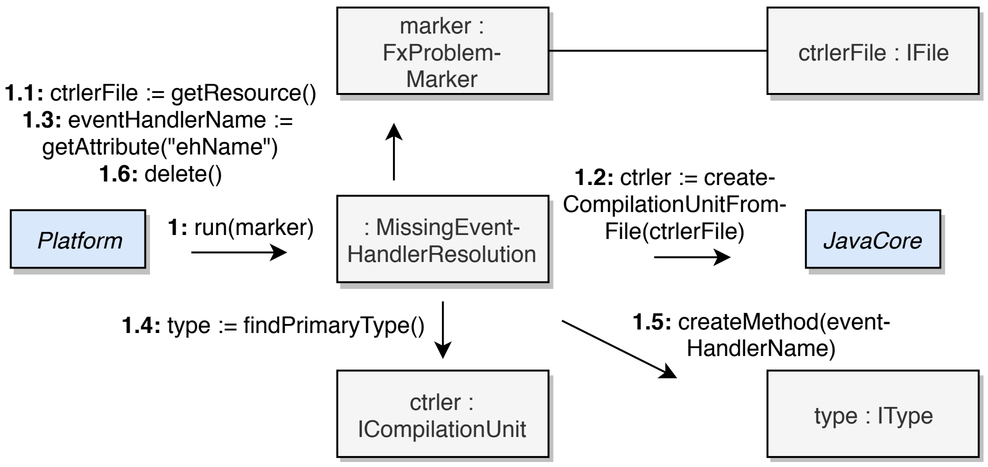

marker : FxProblem-Marker
ctrlerFile : IFile
1.1: ctrlerFile := getResource()
1.3: eventHandlerName := getAttribute("ehName")
1.6: delete()
1.2: ctrler := create-CompilationUnitFrom-File(ctrlerFile)
Platform
1: run(marker)
: MissingEvent-HandlerResolution
JavaCore
1.4: type := findPrimaryType()
1.5: createMethod(event-HandlerName)
ctrler : ICompilationUnit
type : IType

Figure 4.11: The execution of the *missing event handler* problem marker resolution. It extracts the `IType` of the controller from the supplied marker, and calls its `createMethod` method, which takes a `String` representation of the method content as a parameter. Not shown here is the addition of the FXML import to the class' import declarations if it is needed.

5 Project promotion and governance

In this chapter, I cover the topic of making the project publicly accessible as an open source software (OSS) project, with the goal of reaching out to as many users as possible to establish an initial userbase, and to ensure its sustainability as a community maintained project by managing contributors. This involves the initial preparation of a few key documents to go along with the project, as recommended by community standards; the contents of which are covered in section 5.1. Section 5.2 looks at recommended approaches for announcing the project to its interested audience and increasing its visibility in the community. Section 5.3 discusses the management of contributors that may show an interest in the project — the topic of project governance — and how it can adapt to changes in the upstream Scene Builder project in the future.

5.1 Public availability

To make the project along with its source code available to the public, I have put it on the GitHub OSS project hosting platform.[1] They recommend, in their guide for starting OSS projects,[2] to include the following pieces of documentation along with the project: a README; a license; contributing guidelines, and a code of conduct. The contents of these documentation artifacts are elaborated next.

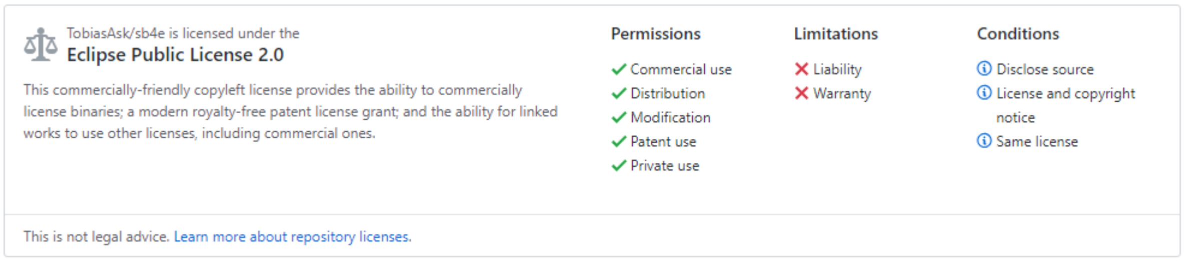
Project readme

Several authors highlight the importance of a project's *readme* in their recommendations for starting OSS projects (Cooper and R. Nyman, 2013; Petrov, 2016; Klishin, 2013; Fogel, 2017). The readme should strive to answer a few key questions about the project, including what it does, why it is useful, and how to get started using it. It should clearly communicate the problem the project solves, and how it solves it. Cooper and R. Nyman (2013) argue that this is best conveyed with a thoughtful *"[project name] is a..."* sentence early on in the text. Further, the readme should state any dependencies the project might have, and include installation instructions for new users. It should also include demonstrations of how it can be used in the form of screenshots — or even videos — depicting its features for projects with GUIs, or in the form of code snippets showcasing its APIs if the project is a library or framework. Fogel (2017) recommends that the readme includes an overview of the project's development status and maturity.

License, contributing guidelines and code of conduct

Including a **license** that defines terms and conditions for the use, modification and distribution of an OSS project's source code is important, as access to it on GitHub does not imply any restrictions — nor permissions — in this regard.[3] The Open Source Initiative defines the distribution terms with which a software license must comply to be regarded as open source on their website.[4] There are some licenses that have been adopted widely enough to be considered standards; examples include the BSD, MIT, and Apache licenses. Eclipse has its own variant, under which the *sb4e* project is licensed. GitHub can display

[1] `https://github.com/TobiasAsk/sb4e` [2] `https://opensource.guide/starting-a-project/`
[3] `https://opensource.guide/legal/` [4] `https://opensource.org/osd`


TobiasAsk/sb4e is licensed under the
Eclipse Public License 2.0

This commercially-friendly copyleft license provides the ability to commercially license binaries; a modern royalty-free patent license grant; and the ability for linked works to use other licenses, including commercial ones.

Permissions	Limitations	Conditions
✓ Commercial use	✗ Liability	ⓘ Disclose source
✓ Distribution	✗ Warranty	ⓘ License and copyright notice
✓ Modification		ⓘ Same license
✓ Patent use		
✓ Private use		

This is not legal advice. Learn more about repository licenses.

Figure 5.1: GitHub's license information panel, here showing the details of *sb4e*'s license, the Eclipse Public License.

a summary of a license's terms and conditions automatically given a license file in the repository, as seen in the screenshot of the license information panel in figure 5.1.

A **code of conduct** (COC) is defined by GitHub as *"a document that establishes expectations for behavior for your project's participants."*[5] It pertains to the social side of the project rather than the technical, and is meant to make community behavior explicit. A common COC is the *Contributor Covenant*,[6] which seeks to promote diversity among contributors. *sb4e* adopts it as a means of welcoming contributors.

Whereas a COC communicates social expectations, **contributing guidelines** communicate technical expectations and conventions. It conveys information about how potential contributors can contribute, and will typically contain instructions on how to proceed to contribute; links to specific areas where help is wanted; and conventions employed in the project such as automated testing and coding style. The JavaFX project's contributing guidelines,[7] for example, contain a step-by-step description of how they accept and handle pull requests.

5.2 Acquiring users and contributors

Once the project has been made publicly available, it is time to reach out to its audience and announce its launch to them — one can not expect users to simply come running, given the sheer number of projects already hosted on GitHub and its limited reach. McDonald and Goggins (2013) conducted interviews with lead and core developers on various OSS projects on GitHub regarding their views on project success and found that a commonly mentioned measure of success was the number of contributors. Hence, community outreach should be considered an important aspect of the project's promotion. To that end, Pazdera (2015) suggests publications in the form of blog posts and announcements in community forums. He also recommends promoting the project as answers to questions in general coding question forums such as StackOverflow and Reddit. Petrov (2016) argues that partnering up with other OSS projects in the project's domain is a good way to extending project reach.

In the case of *sb4e*, there are a couple of promising avenues for announcement. The *Eclipse community forums*[8] is a natural place to start; it has a sub-forum dedicated to the *e(fx)clipse* project that is a good candidate for an announcement post. The founder of

[5] `https://opensource.guide/code-of-conduct/` [6] `https://www.contributor-covenant.org/`
[7] `https://github.com/javafxports/openjdk-jfx/blob/develop/.github/CONTRIBUTING.md`
[8] `https://www.eclipse.org/forums/`

that project also has a blog to which a guest post can be submitted. StackOverflow has a *scenebuilder* question tag that sees quite a lot of activity,[9] and Reddit has a *JavaFX* sub-forum with 1,711 Members. The Scene Builder and e(fx)clipse projects are both also OSS projects; thus, according to Petrov (2016), it is beneficial for the promotion of *sb4e* to contribute to and partner up with them, if only to establish a preliminary contact. The Eclipse Marketplace is a hosting site for Eclipse extensions from which they can be discovered and installed in an IDE instance. Anyone can submit their extensions to it for a review and potential hosting. It does, however, require an existing *Eclipse update site* where the project has been built.

Dabbish et al. (2012) interviewed GitHub users to examine the effects of the site's social translucence, and found that users make a number of social inferences from various visible cues on a project's site during browsing. In particular, they found that users take the project's recent activity history into consideration when deciding whether or not to contribute, and when judging which projects are most likely to succeed. As the source code, and by extension the implementation choices and architectural decisions that have been made, is transparent when residing on GitHub, is is likely that users also take its quality into consideration. Thus, working in accordance with established practices and patterns to ensure a high code quality can serve as a means of assuring others of the state of the project.

One of the key traits of OSS development is that the distinction between users and contributors is blurred. Raymond (1999) emphasizes this point and encourages OSS developers to treat their users as co-developers. The Mozilla Science Lab proposes the use of *personas* and *pathways* as a means of attracting users and growing them into contributors over time.[10] A persona is a detailed description of an imaginary user that embodies their needs and motivation, and can be used by developers to see things from a different perspective than their own. Mozilla defines a pathway is a step-wise description of the persona's interaction with the project, from an initial introduction all the way to a potential leadership role.

GitHub's issue list can be used as a way of welcoming contributions by making not only bugs, but also missing features explicit and transparent. Issues have labels that convey information about their type (bug, enhancement etc.), as well as their status. Two labels that target contributors specifically are the *good first issue* and *help wanted* labels, which can be used to guide newcomers towards these issues, create an entry point for them, and lower the contribution barrier of the project, which is often quite high (Krogh, Spaeth and Lakhani, 2003; Ducheneaut, 2005). It is also important that the issue list be open for addition by anyone. This way, *sb4e* can be tested by its users, elevating their status to potential bug reporters. Aberdour (2007) claims that this transparency in the issues list is an important advantage of OSS projects compared to closed-source development. Raymond (1999) points this out with the phrase *"given enough eyeballs, all bugs are shallow"*.

Following Mozilla's template for pathways, an imagined pathway for a contributor interacting with *sb4e* might look like this:

1. **Discovery**: Sees announcement post on the *Eclipse community forum.*
2. **First Contact:** Visits the project's GitHub page and tries it out following instruc-

[9] There are 1,579 questions asked with the tag on the site at the time of writing, with 12 questions asked in the last week.

[10] `https://mozillascience.github.io/working-open-workshop/personas_pathways/`

tions in the readme.

3. **Participation:** Finds an issue in the issue list with the *good first issue* label, implements the bug fix or feature request, and opens a pull request.
4. **Sustained Participation:** Starts following the project on GitHub, and creates and/or fixes issues as they occur.
5. **Networked Participation:** Talks to fellow programmers also involved with JavaFX application development and recommends the project.
6. **Leadership**: Acquires push access to the repository.

5.3 Managing project growth

If a program is to remain useful and relevant over time, it must continue to evolve and adapt itself (L. Nyman and Lindman, 2013). This section considers two aspects of project evolution. First, the management of contributors is discussed. Next, the project's adaptation to and incorporation of changes in the upstream Scene Builder project is handled.

5.3.1 Project governance

If the project gains momentum in terms of community interest and participation, the management of contributors should be considered. Gardler and Hanganu (2013) define a governance model as a model that *"describes the roles that project participants can take on and the process for decision making within the project. In addition, it describes the ground rules for participation in the project and the processes for communicating and sharing within the project team and community"*. They distinguish between two opposing types of models according to the degree to which project control is distributed: In a **benevolent dictator** model, project control is the responsibility of a single person, typically the founder. The **meritocracy** model contrasts it, by giving contributors increasing influence over time based on their merits. While the distinction is binary, it labels two ends of a continuum on which most projects land somewhere in the middle. A project's model can also change over time, and Fogel (2017) notes that projects tend to move from dictatorships towards more openly democratic systems as they mature.

Raymond (1999) defines two opposite development styles in terms of openness towards external contributions, called contribution models: The **cathedral** model takes a very rigid standpoint, wherein a centralized, *a priori* approach is taken to build software. Development is done in isolation from contributors, with infrequent public releases. Opposed to that is the **bazaar** model, inspired by the way the Linux project is run. Key here is an openness towards users and a frequent release schedule. When projects are compared along the openness dimensions of these two models, they land on a location in two-dimensional space. Figure 5.2 shows such a space, with some exemplary OSS projects placed therein. While the *sb4e* project's need for a governance model might not seem immediately pressing — its participants are, after all, only me — Gardler and Hanganu (2013) argue that even small, recently started projects need such models. In particular, they claim that an explicit statement of how interested third-parties can contribute is key to attracting their interest.

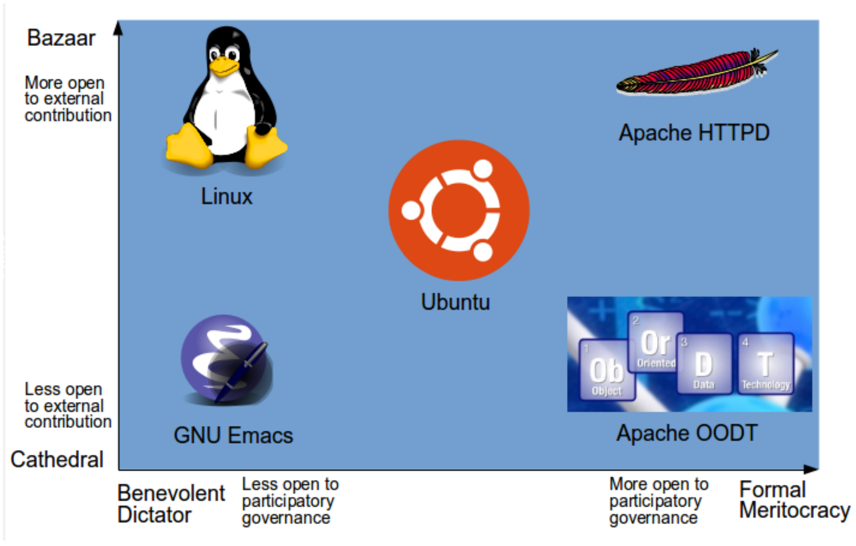

Bazaar
More open to external contribution
Linux
Apache HTTPD
Ubuntu
Less open to external contribution
Cathedral
GNU Emacs
Apache OODT
Benevolent Dictator
Less open to participatory governance
More open to participatory governance
Formal Meritocracy

Figure 5.2: Comparison of the openness of selected OS projects' governance and contribution models, from Gardler and Hanganu (2013). The combination of the *sb4e* project's current *cathedral* model with a benevolent dictatorship places it besides GNU Emacs in this two-dimensional space.

5.3.2 Adapting to changes in the upstream Scene Builder project

When looking at the project's long-term sustainability, it is important to take its eco-system relations into consideration. The most important actor in *sb4e*'s eco-system is the upstream Scene Builder project. The consumption of Scene Builder's components in the integration requires minor changes to some of them. How should these modifications be applied, when keeping updates to the upstream project in mind? And how should the integration consume the components in the first place? One option is to consume them as an **external dependency** in the form of a JAR that Gluon releases as part of the main Scene Builder releases, called the *Scene Builder kit.* It is comprised of the components of the kit module in isolation (recall Scene Builder's two-layered architecture discussed in section 2.2.2). This consumption model is illustrated in figure 5.3a. As the components would be consumed in their binary form, this consumption model requires the inclusion of the changes in the upstream project. This ties the *sb4e* project's version of the components to Gluon's release schedule — hence, the approach is not ideal.

Another option is to clone the repository, copy the project's kit module into the plug-in project and consume the components from there (and commit the module as part of the *sb4e* project to its repository). The issue with this approach is that any custom changes are overwritten when new versions of the files they are made in become available in the upstream project. There are two remedies to this problem: patching, and the inclusion of the project as a version-controlled sub-project called a *Git submodule*[11] instead of a mere copy of it. **Patching** can be used to apply the changes automatically. In this approach, the required changes are first stored as *diff files*, which are textual representations of the difference between two versions of the same file. These files can then be applied to the source files in the patching process, which modifies files according to their diffs, leaving them in an updated state. The process can be applied automatically as part of the build

[11] `https://git-scm.com/book/en/v2/Git-Tools-Submodules`

process. With this approach, updated files are pulled from the upstream project and copied into the Scene Builder project copy when updates are made. Then, the patches are applied to make the changes required by the integration. The consumption model is illustrated in figure 5.3b.

A better approach is to include a fork of the Scene Builder project as a **Git submodule** within the plug-in's Git repository, as illustrated in figure 5.3c. Initially, this was not an option, since the project used the Mercurial Version Control System (VCS) instead of Git, but it was made possible by the project's recent migration to GitHub (and thus also to the Git VCS) on May 10, 2018.[12] This is preferable to the patching approach since upstream changes are merged in with the changes required by the plug-in automatically by Git — i.e., when new files are fetched from the main project, they do not overwrite the ones existing in the project copy. The submodule can refer to both the fork and the main repository as its remotes so that updates are easily fetched, while modifications can reside on the fork.

Table 5.1 shows a comparison of the three discussed consumption models based on their various prerequisites and implications. Based on these attributes, the use of a Git submodule is to be preferred. Changes to the upstream project are easily incorporated and merged with the changes *sb4e* requires, without a need for patching, nor a need for the changes to be pushed upstream.

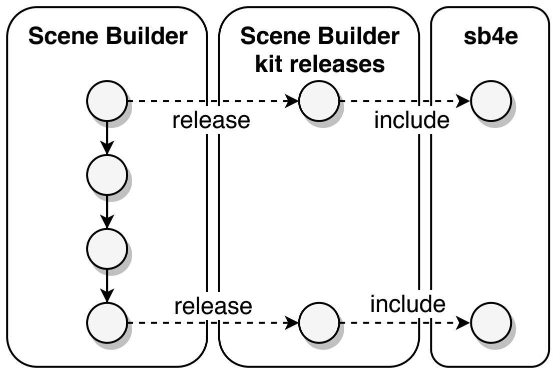
Table 5.1: Comparison of the three main consumption models according to their attributes impacting the plug-in's governance.

	Consumption model		
	Kit releases	Git submodule	Sub-project
Presupposes matching VCSs	No	Yes	No
Presupposes merging the changes upstream	Yes	No	No
Presupposes removal of Gluon Mobile dependency	Yes	Yes	Yes
Latest version	No	Yes	Yes
Requires patching	N/A	No	Yes
Modules	Kit	App & kit	Kit

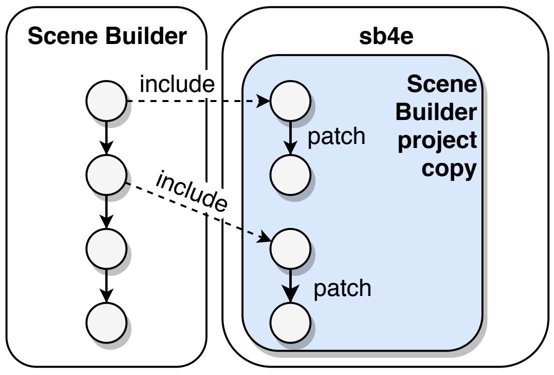
Gluon Mobile

Even though most of the Scene Builder project is open source, a small part of it is proprietary to its developers, Gluon. Specifically, the application has since its 8.3.0 release included some custom components and Cascading Style Sheet (CSS) themes from an external dependency called Charm Glisten by default (Gluon HQ, 2016), which is part of a larger, proprietary client-side library called Gluon Mobile. Glisten is available for download at a dependency repository in the form of a JAR that can be fetched automatically by build systems such as Gradle and Maven, but it is not open source. For general use, developers must pay for a license to use it (Gluon HQ, 2018). Gluon offers short-term, free licenses for

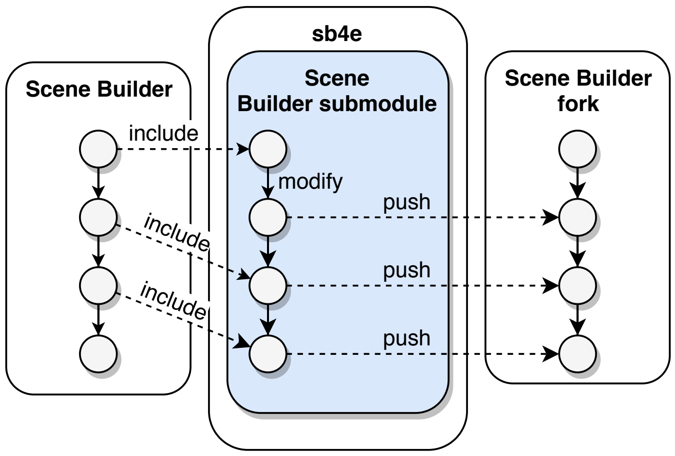
[12] `https://tinyurl.com/y3xfwffp`



(a) Consuming the Scene Builder kit as an external dependency.



(b) Consuming Scene Builder as a project copy within the main plug-in project.



(c) Consuming Scene Builder as a *Git Submodule* within the main plug-in project.

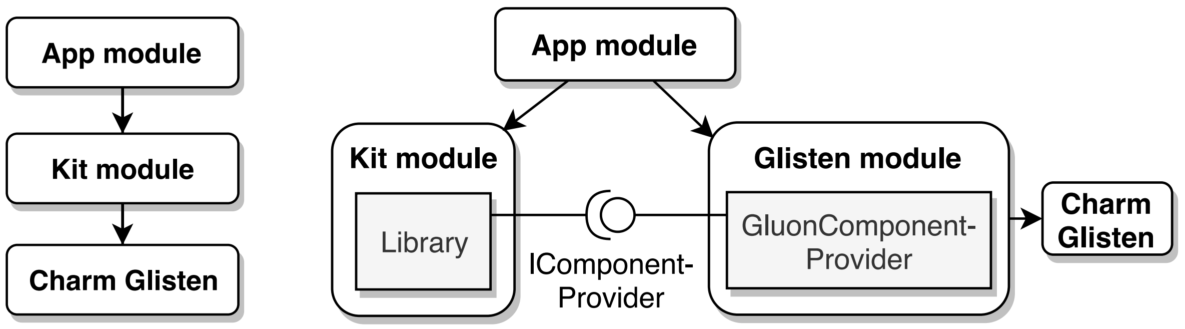
Figure 5.3: The three consumption models by which the *sb4e* can consume the components of Scene Builder, and respond to changes in the project. If sb4e consumes Scene Builder's components as an external dependency — the approach illustrated in **(a)** — it fetches the latest release of *kit* when it is released and references the components in the JAR. In the approach illustrated in **(b)**, sb4e consumes the components from an internal copy of the project, and the files must be patched following each pull. With the use of a Git sub-module, shown in **(c)**, the modifications required by *sb4e* are only applied once after the main project is pulled. When following updates occur, they can be merged on top of the changes by Git.

use by open source projects that developers can apply for,[13] but this is still problematic for sb4e, which is licensed under the open source Eclipse license.

Currently, all references to the custom components must be removed manually from the source code. While this allows *sb4e* to build and run without the Glisten dependency, thus avoiding any licensing issues, it is not a particularly extensible nor long-term solution. In the long term, it is desirable to have the *kit* module — as a general-purpose, open source library — independent of any direct dependencies on proprietary components in its default state. It is, however, also essential that the main product release remains the same, with the components included. The dependency graphs in figure 5.4 show how the *kit* module can be decoupled from the external *Charm Glisten* library by introducing a new module on which the *app* module depends. While the *kit* module references the library in other components besides the `Library`, whose dependency is dealt with in the figure, the approach is applicable to other components, as well. Direct references to the Charm dependency from the *kit* module can be replaced by abstract references to which the new module provides service implementations.

The mechanism with which the implementations are discovered can also be left abstract, but a standard implementation that uses Java's built-in Service Provider Interface (SPI) mechanism has been included, to include the components in the application by default. The *Glisten* module declares its implementations in a text file in its `META-INF/services` folder, from which the SPI's `ServiceLoader` discovers and loads the implementations at runtime (Oracle Corporation, 2017, Creating Extensible Applications).

Making the look-up mechanism abstract means that the plug-in can load custom component providers using Eclipse's extension point mechanism, instead of SPI. This means that the components can be added dynamically at runtime if Gluon releases their provider as a plug-in. Since the plug-in itself then has no direct reference to the Charm Glisten dependency, it is still completely open source. This mechanism also makes the plug-in extensible with all kinds of custom components in the form of plug-ins, and not just Gluon's.



(a) Direct dependency.

(b) Indirect dependency via the novel *Glisten* module.

Figure 5.4: Dependency graphs showing how the *kit* module can be decoupled from its current direct dependency on the proprietary *Charm Glisten* library (as in graph **(a)**) by introducing a new module between them (graph **(b)**). The *app* module depends on both of them to keep the custom components in the stand-alone application by default.

[13] `https://gluonhq.com/programs/free-gluon-licenses/`

6 Evaluation

This chapter contains an evaluation of the work that has been done. Section 6.1 evaluates three of the plug-in's contributions to the Eclipse platform — the FXML Editor, the Inspector View, and the builder used to detect view-controller discrepancies — in accordance with certain industry standards regarding their implementation. Section 6.2 details the planning, execution and results of a usability test of the plug-in that was conducted. Last, section 6.3 analyses and critiques various implementation choices and approaches.

6.1 Evaluation in accordance with RFRS Considerations

The plug-in's FXML Editor, Inspector View, and builder can be evaluated in accordance with the various Ready for Rational Software (RFRS) Requirements listed by Clayberg and Rubel (2009). Originally put forward by IBM Corporation (Clayberg and Rubel, 2009, Appendix B), and based largely on Eclipse's User Interface Guidelines (Edgar et al., 2007), these requirements are meant to ensure an overall consistency in functionality and look and feel across plug-ins. While their fulfillment, or lack thereof, does not necessarily determine the quality of a plug-in, they have been treated as useful guidelines. The results of the evaluation are listed in table 6.1.

6.1.1 Inspector View

#1 **Views for navigation** is a requirement that states:

> *Use a view to navigate a hierarchy of information, open an editor, or display the properties of an object.*

It is **fulfilled**, as the Inspector View displays the properties of the component selected in the editor.

#2 **Views save immediately** is a requirement that states:

> *Modifications made within a view must be saved immediately. [...]*

It is **fulfilled** since document modifications made from the Inspector View are applied immediately to the underlying model, and the editor is marked as dirty.

#3 **View initialization** is a requirement that states:

> *When a view first opens, derive the view input from the state of the perspective. The view may consult the perspective input or selection, or the state of another view. [...]*

It is **fulfilled**; if an editor with an active selection is open when the Inspector View is opened, the properties of the selected component are displayed in the view.

#4 **View global actions** is a requirement that states:

> *If a view has support for cut, copy, paste, or any of the global actions, the same actions must be executable from the same actions in the window menu and toolbar. [...] The following are the supported global actions: undo, redo, cut, copy, paste, print, delete, find, select all, and bookmark.*

This requirement is only **partially fulfilled**. The applicable actions (cut, copy and paste) have not been integrated with the Eclipse platform. They are, however, still accessible using keyboard shortcuts, since these are forwarded from SWT to the underlying OS by the `FXCanvas`.

#5 **Persist view state** is a requirement that states:

> *Persist the state of each view between sessions. If a view is self-starting in the sense that its input is not derived from selection in other parts, the state of the view should be peristed between sessions. [...]*

It is not applicable to the Inspector View, since its input is derived exclusively from the selection in the FXML Editor.

#6 **Register context menus** is a best practice that states:

> *Register all context menus in the view with the platform. In the platform, the menu and toolbar for a view are automatically extended by the platform. By contrast, the context menu extension is supported in collaboration between the view and the platform. To achieve this collaboration, a view must register each context menu it contains with the platform.*

It is **not applicable** to the Inspector View. As the view does not contain any resources on which context menus can be opened, it can not be said to have context menus. The panel opens JavaFX menus for cut, copy and paste on right-clicks within the input fields, and, while it can be argued that these are context menus, there is no meaningful way the platform can contribute to it.

#7 **Action filters for views** is a best practice that states:

> *Implement an action filter for each object type in the view. [...]*

Again, this is not applicable, as the view does not contain any objects/resources on which filters can be applied.

6.1.2 FXML Editor

#1 **Using an editor to edit or browse** is a requirement that states:

> *Use an editor to edit or browse a file, document, or other input object. [...]*

It is **fulfilled** by the FXML Editor, as it is used to edit FXML documents.

#2 **Editor lifecycle** is a requirement that states:

> *Modifications made in an editor must follow an open-save-close lifecycle model. When an editor first opens, the editor's contents should be unmodified (clean). If the contents are modified, the editor should communicate this change to the platform. In response, an asterisk should appear in the editor's tab. The modifications should be buffered within the edit model until such time as the user explicitly saves them. At that point, the modifications should be committed to the model storage.*

It is **fulfilled** by the FXML Editor, as it keeps a copy of the most recently saved document text, and compares this to that of Scene Builder's model whenever it changes to maintain the dirty status. When the developer performs the *Save* command, the contents of the input document, along with the document text used for comparison, are overwritten with that of the document model.

#3 **Accessing global actions** is a requirement that states:

> *If an editor has support for cut, copy, paste, or any of the global actions, the same actions must be executable from the same actions in the window menu and toolbar. [...]*

It is **fulfilled** by the FXML Editor, as it integrates with Eclipse's *operation history* to support undo/redo, and maps Eclipse's global actions, such as cut, to Scene Builder's corresponding document actions.

#4 **Closing when the object is deleted** is a requirement that states:

> *If the input to an editor is deleted and the editor contains no changes, the editor should be closed. [...]*

It is **fulfilled** by the FXML Editor. By using a resource change listener that tracks changes made to the editor's input document from outside of Eclipse, the editor is able to close itself when the underlying file is deleted. Note that removal of the document is only detected immediately when the "Refresh using native hooks or pulling" option is enabled; otherwise, a refresh of the workspace is required for immediate detection.

#5 **Synchronize external changes** is a requirement that states:

> *If modifications to a resource are made outside the workbench, users should be prompted to either override the changes made outside the workbench or back out of the Save operation when the save action is invoked in the editor.*

It is **fulfilled** by the FXML Editor, using the same detection mechanism that also fulfills the previous requirement.

#6 **Registering editor menus** is a best practice that states:

> *Register with the platform all context menus in the editor. [...]*

The FXML Editor does **not comply** with the best practice — rather, Scene Builder's native context menu is used. Even though this breaks with the principle of keeping the GUI consistent across plug-ins, implementing it was deemed too complex an effort, with little functionality to be gained. As Scene Builder's context menu functionality is closely linked to JavaFX, duplicating all of it in SWT's menu system would require much work, as well as a lot modifications to the core of Scene Builder — the avoidance of which was a high priority during development. Using Scene Builder's native context menu also means that the platform cannot contribute to the menu, which is a recognized weakness. The extent of the weakness can be argued, however, as the contents of the menu is highly domain specific to JavaFX components.

#7 **Editor action filters** is a best practice that states:

> *Implement an action filter for each object type in the editor. [...]*

The FXML Editor does **not comply** with the best practice, for much of the same reasons given in the previous discussion.

#8 **Unsaved editor modifications** is a best practice that states:

If the input to an editor is deleted and the editor contains changes, the editor should give the user a chance to save the changes to another location, and then close.

The FXML Editor **complies** with the best practice; if it is dirty and the input document is removed, a *Save as* dialog is shown. Note that, at the time of writing, the Eclipse's Java editor does not behave in quite such a manner. It displays a warning dialog, as part of the refactor process, but a *Save as* dialog is not shown. Hence, it is unclear what should be considered the best practice when dealing with such events.

#9 **Prefix dirty resources** is a best practice that states:

If a resource is dirty, prefix the resource name presented in the editor tab with an asterisk.

While the editor **complies** with the best practice, as it correctly maintains its dirty status and notifies the platform of status changes, it is arguably the responsibility of the platform, and not of the editor.

#10 **Editor outline view** is a best practice that states:

If the data within an editor is too extensive to see on a single screen, and will yield a structured outline, the editor should provide an outline model to the Outline view. [...]

It is purposefully **not fulfilled** by the editor, as Scene Builder's Hierarchy Panel — included in the editor — was deemed a replacement of the Outline view. While it can be argued that this breaks with the norm of the Eclipse platform, the Hierarchy Panel was kept as is because of the functionality built into it that the Outline view can not provide.

#11 **Synchronize with outline view** is a best practice that states:

Notification about location between an editor and the Outline view should be two-way. [...]

It is **not fulfilled**, as the editor does not link with the Outline view. It should be noted, however, that there is a strong built-in link between the Hierarchy Panel and the Content Panel, which mirrors the functionality described in the consideration.

6.1.3 Builder

The *RFRS requirements* contain six items that deal with builders, but not all of them are relevant for *sb4e*'s builder. Hence, some requirements have been excluded.

#1 **Do not replace existing builders** is a requirement that states:

Extensions cannot replace the builders associated with project natures provided by the workbench, or by other vendors.

It is **fulfilled** as *sb4e* does not override existing builder upon assignment.

#2 **Do not misuse the term "build"** is a best practice that states:

[...] do not use the term "build" in your product implementation or documentation [...]

The builder **complies** with the best practice, as the term is not misused.

#3 **Respond to clean-build requests** is a best practice that states:

> *Builders should respond to CLEAN_BUILD requests. [...]*

The builder **complies** with it, as it removes all JavaFX problem markers created during previous runs.

#4 **Use IResourceProxy when possible** is a best practice that states:

> *Builders often need to process all the reosources in a project when an `IncrementalProjectBuilder.FULL_BUILD` has been requested. There is an improved technique available starting with Eclipse 2.1. `IResourceProxyVisitor` should be used in place of an `IResourceVisitor`. The proxy visitor provides access to lightweight `IResourceProxy` objects. These can return a real `IResource` object when required, but when not required, they result in improved overall builder performance for full builds.*

The builder does **not comply** with the best practice, as it needs full `IResource` objects in its analysis.

#5 **Builders must be added by natures** is a best practice that states:

> *A builder must be added to a project by a nature. [...]*

sb4e **complies** with the best practice, as it contributes a custom *JavaFX project* nature to which the builder is associated.

6.2 Usability test

A usability test was conducted to evaluate the integration's ease of use. Here, I detail the steps involved in its preparation and performance, and present the observations that were made during the test, along with conclusions inferred from them. Note that the test was performed at a stage in the development where the discrepancy assistance mechanisms had not yet been implemented.

Goals

The main goal of the test was to examine usage patterns in JavaFX application development with the plug-in to identify both possible usability problems, and potential user assistance mechanisms that could be added. Note that this should not entail testing the usability of Scene Builder itself; rather, only usage patterns pertaining to the integration as a whole should be examined. Another goal was to compare the usability of using the two applications side-by-side, and using the integrated version.

Recruitment of participants

An announcement asking for voluntary test participants was sent to students enrolled in various university courses via their learning management system. The announcement contained a description of the product, as well as an outline of the test plan. Eight students responded to the announcement within a few days — this was considered a successful recruitment, as it is considered acceptable to have between five and twelve test subjects

Table 6.1: The results of evaluating *sb4e*'s FXML Editor, Inspector View and builder according to the RFRS Requirements.

RFRS #	Consideration name	Fulfillment status
1	Views for navigation	✓
2	Views save immediately	✓
3	View initialization	✓
4	View global actions	Partially fulfilled
5	Persist view state	*Not applicable*
6	Register context menus	×
7	Action filters for views	*Not applicable*
1	Using an editor to edit or browse	✓
2	Editor lifecycle	✓
3	Accessing global actions	✓
4	Closing when the object is deleted	✓
5	Synchronize external changes	✓
6	Registering editor menus	×
7	Editor action filters	×
8	Unsaved editor modifications	✓
9	Prefix dirty resources	✓
10	Editor outline view	×
11	Synchronize with outline view	×
1	Do not replace existing builders	✓
2	Do not misuse the term "build"	✓
3	Respond to clean-build requests	✓
4	IResourceProxy when possible	×
5	Builders must be added by natures	✓

(Dumas and Redish, cited in Sharp, Rogers and Preece (2007)). Out of the initial eight respondents, seven of them participated in the test.

The task

The test subjects were tasked with developing a minimal JavaFX application. Since the subjects were all students in their second or fourth semesters, the complexity of the application — i.e, the capabilities of the resulting interface and the amount of work needed to produce it — was intended to be low. Another factor that also contributed to a lower complexity level was that I would be observing the subjects on my own without external aids. Despite being of low complexity, the tasks were not trivial; they would still require working with both the view and the controller.

The task was to produce an application with an interface similar to that shown in figure 6.1. The application had one requirement: when the user clicks the button, the letter case of the text should switch to the opposite of what it was prior to the click. The following steps, carried out in no particular order, are required to achieve the desired result:

- Create a view with a `Pane` as the root, and a `Button` and a `Label` as its children.
- Create a controller with a `Label` field subject to injection, and a method that implements the switching behavior.
- Assign the controller to the view, the `fx:id` of the label to the field in the controller, and the `#onAction` event handler to the method in the controller.

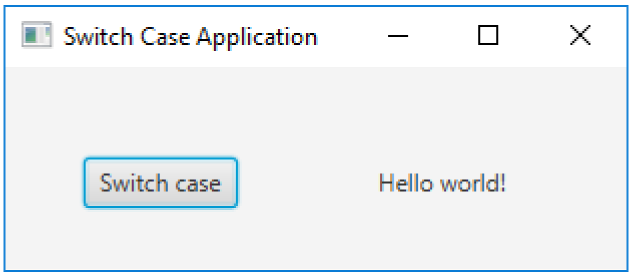


Figure 6.1: The minimal Switch Case application that the test subjects were tasked with creating.

Test procedure

Sharp, Rogers and Preece (2007) suggest a number of quantitative performance measures that can be used in an evaluation — the time spent to complete a task, and the number of errors made, for example — as well as user satisfaction questionnaires and interviews. As the tasks given to the test subjects were intentionally limited in terms of both their scope and expected time to completion, quantitative measures related to time were deemed unfruitful. Hence, focus was rather placed on observing the subjects as they performed their tasks, taking note of any issues that might occur, and observing possible emerging issue patterns across the participants. Informal interviews concerning their experiences were performed with the test subjects after they had completed the tasks.

All the test subjects performed their tasks separately in allocated time slots, so that they could be manually observed. Upon arrival, the subjects were greeted in a friendly manner, with the goal of establishing a relaxed atmosphere. The goal of the test was stated, emphasizing that the system under test was the software, and not the subjects themselves. The task along with basic instructions were handed out as print-outs. The subjects were also encouraged to think out aloud as they performed the task. Six of the seven subjects did not use their own machines,[1] for one of two reasons: Some of them had only JDK 9 installed, and since the plug-in did not support it at the time, they were unable to install the plug-in. Others had not had the opportunity to install the plug-in before showing up for the test. The computer screens of the subjects were observed while they performed their tasks. Any mistakes that were made, as well as any comments stated out loud, were noted on paper. Assistance was provided only if the subjects were completely halted during execution. One of the test participants showed a particular interest in the project. As he was the last participant of the day, with no time limit for the session, he performed the task with the use of the stand-alone application separate from Eclipse as well, for comparison.

Observations

As no formal data collection was performed, I present the most notable observations regarding usage patterns:

- **Observation 1**: All seven test subjects entered the `fx:ids` and *event handler* method names in the document (using the FXML Editor) *before* adding them to the controller class.
- **Observation 2**: All seven test subjects entered the controller name manually, and three of them used the incorrect form of *ClassName.java*, instead of the fully qualified name (*package.ClassName*) required by the FXML document loader.
- **Observation 3:** The test subject that performed the task with the use of first the integrated version, and then the applications side-by-side, encountered issues when switching between the application windows when using the stand-alone version.
- **Observation 4:** The test subject that performed the task in stand-alone spent some time moving the document to the correct location in the project.

Inferences

Based on observations 1 and 2, I inferred the following: The suggestions shown in the input fields for the controller name, `fx:ids` and event handler names are of limited value, as all participants finished their work with the view *before* moving on to the controller. User assistance that can detect discrepancies and prompt the developer to act accordingly is preferred to the suggestions in the input fields. Further, from observations 3 and 4, I concluded that the overall usability of the integration was superior to that of the combination of the stand-alone applications.

[1] This is acknowledged as a detrimental factor for the usefulness of the test. It is desirable to try out the plug-in in as many different environments as possible during testing, in search of miscellaneous bugs that can arise from subtle variations in the host environment.

6.3 Discussion

Here, I point out various aspects that could have been done differently.

6.3.1 On view-controller coherence

In Eclipse, there are two ways of participating in the build process: by extending either the `org.eclipse.core.resources.builders` extension point and contributing an `IncrementalProjectBuilder`, or the `org.eclipse.jdt.core.compilationParticipant` point, contributing a `CompilationParticipant`. The key difference between the two is that while the former receives notification about changes made to any type of resource in the workspace, the latter only receives notification about changes made to Java classes. *sb4e* uses the former for this exact reason, but there is a drawback to the approach. Since builders are only ran after a file has been saved, they can not detect discrepancies prior to a save invocation. This is subtly different from the way problem detection is done for Java classes in the IDE, where problems are detected and reported during *reconcile* operations, which are performed more frequently than builds, incremental or not. For example, if an unresolved type reference resulting from a misspelling is detected, it is reported immediately, prior to a save. Implementations of `CompilationParticipant` can override the `reconcile` method to hook into reconcile operations. While the difference is subtle, it can influence developers using *sb4e* for the first time. In particular, they may expect discrepancies to be detected and reported in the same fashion as Java problems.

6.3.2 Regarding community outreach

Not all of the various means of spreading the word about the project discussed in section 5.2 have been carried out. Some measures have been taken: We have made an announcement post on the *Eclipse* community forum,[2] which received some attention. I have also contributed two changes to the Scene Builder project, and the developers are aware of the work that has been done with the integration. Further, I have had some contact with the main developer of the *e(fx)clipse* project, in the form of both e-mails and a report of a crucial bug. Beyond that, little promotion efforts have been made, and it is acknowledged that the project has not been promoted widely enough, and that it could benefit from a more substantial promotion.

6.3.3 Comments on the usability test

In hindsight, the usability test could have been performed differently, to yield more informative results. For one, the time allocated to each participant — thirty minutes — proved to be on the short side. While all participants were able to complete the main task in time, only one of them had the time to perform it using first the integration, followed by the combination of Eclipse and Scene Builder as a stand-alone application. If all the participants had had the time to do that, more data could have been collected regarding the differences between the integration and the stand-alone application.

Another issue is that of deciding on which data collect in the first place. A prepared questionnaire might have been preferable to the semi-structured interviews that were

[2] `https://www.eclipse.org/forums/index.php/t/1091755/`

conducted. It is possible, however, that the answers to both of these two types of feedback gathering processes would have been biased towards positive responses regardless of the product's usability, seeing as the participants were aware that their interviewer was also the sole developer.

6.3.4 Remarks on the automated test suite

There is a general lack of automated *unit* tests for the project. This is because it is difficult to test Eclipse classes in isolation, as most of them are tied closely to the part of the platform to which they contribute. Unit testing an editor, for example, involves calls to its `init(IEditorSite, IEditorInput)` and `createPartControl(Composite)` methods, which both require either extensive mock objects (with their own mocked dependencies), or actual object implementations. Using actual implementations defeats the purpose of unit testing, however, which is to examine the behavior of an object in isolation from the actual behavior of its dependencies.

There is support for writing and executing automated *system* tests in an environment with a fully instantiated platform. Some tests have been written, but as they require a lot of code to set up an environment (such as a project with a view and a controller) and emulate user behavior, and take a long time to run, manual testing in a prepared example project is arguably a better and more useful approach.

6.3.5 On refactorings

Not all the refactorings identified in chapter 3 have been implemented yet. I implemented support for event handler name refactorings as a way of becoming familiar with the technicalities of refactoring process participation. The approach to add support for `fx:id` and controller name refactorings is very similar, and their implementations were not prioritized.

7 Conclusion

What started out as a mostly graphical embedding of the Scene Builder panels in the IDE has been developed into a more complete developer tool in terms of its utility and the developer assistance mechanisms it provides. I have analyzed the connection between views and their controllers in the JavaFX framework — paying particular attention to the consequences of discrepancies in the various links between them on a running application — and have identified how Eclipse mechanisms can be applied to help maintain the correctness of it. I have designed and implemented the developer assistance mechanisms, utilizing Eclipse's facilities for static code analysis by means of its Java model; problem reporting by means of problem markers; and quick fixes that perform code manipulation by means of marker resolutions. This was done in a manner that is idiomatic to Eclipse's general approach to developer assistance. I have also applied Eclipse mechanisms to extend the context awareness of various Scene Builder components.

To increase the project's impact and visibility to potential users, I have made the project publicly accessible in an open source software repository. I have begun to promote the project through various online channels, and have established contact with prominent actors in the project's community. I have attempted to make the project welcoming to users and contributors alike by crafting a project *readme*, adopting a code of conduct, and attaching the Eclipse license. Further, I hope that by having followed various Eclipse guidelines and best practices during development, I have promoted the confidence others gain in the state of the project during inspection — and, by extension, their willingness to contribute.

Suggestions for further work

From the discussions in section 6.3, it is clear that there is room for improvement. Beyond that, smart quick fixes that can catch instances where an event handler name has been misspelled, for example, could yield valuable utility. These types of quick fixes were briefly discussed in sections 3.3 and 3.4. Further, some of Scene Builder's functionality has not yet been integrated — the most prominent of which being the zooming functionality, and the CSS analyzer.

References

Aberdour, M. (2007). «Achieving Quality in Open-Source Software». In: *IEEE Software* 24.1, pp. 58–64. ISSN: 0740-7459. DOI: 10.1109/MS.2007.2.

Aeschlimann, Martin, Dirk Bäumer and Jerome Lanneluc (2005). «Java Tool Smithing: Extending the Eclipse Java Development Tools». In: *eclipseCON 2005*. URL: https://eclipsecon.org/2005/presentations/EclipseCON2005_Tutorial29.pdf.

Amsden, Jim (2001). *Levels of Integration*. URL: https://www.eclipse.org/articles/Article-Levels-Of-Integration/levels-of-integration.html (visited on 17/04/2019).

Ask, Tobias (2018). *Integrating the Scene Builder GUI editor into the Eclipse IDE*. URL: http://folk.ntnu.no/tobiaas/fordypningsprosjekt_tobias_ask.pdf (visited on 09/05/2019).

Bass, Len, Paul Clements and Rick Kazman (2013). *Software Architecture in Practice*. 3rd. Addison-Wesley.

Brown, Greg (2011). *Introducing FXML*. URL: http://fxexperience.com/wp-content/uploads/2011/08/Introducing-FXML.pdf (visited on 24/04/2019).

Clayberg, Eric and Dan Rubel (2009). *eclipse Plug-ins*. Third Edition. Addison-Wesley.

Cooper, Peter and Robert Nyman (2013). *How to Spread The Word About Your Code*. URL: https://hacks.mozilla.org/2013/05/how-to-spread-the-word-about-your-code/ (visited on 05/06/2019).

Dabbish, Laura et al. (2012). «Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository». In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW '12. Seattle, Washington, USA: ACM, pp. 1277–1286. ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145396. URL: http://doi.acm.org/10.1145/2145204.2145396.

Ducheneaut, Nicolas (2005). «Socialization in an Open Source Software Community: A Socio-Technical Analysis». In: *Computer Supported Cooperative Work (CSCW)* 14.4, pp. 323–368. ISSN: 1573-7551. DOI: 10.1007/s10606-005-9000-1. URL: https://doi.org/10.1007/s10606-005-9000-1.

Eclipse Foundation (2019). *Eclipse documentation*. URL: https://help.eclipse.org/2019-03/index.jsp (visited on 09/04/2019).

Edgar, Nick et al. (2007). *Eclipse User Interface Guidelines*. URL: https://wiki.eclipse.org/User_Interface_Guidelines (visited on 04/04/2019).

Fogel, Karl (2017). *Producing Open Source Software: How to Run a Successful Free Software Project*. Second Edition. O'Reilly Media. URL: http://www.producingoss.com/.

Fowler, Martin (2002). *Refactoring: Improving the Design of Existing Code*. Second Edition. Addison Wesley.

Gardler, Ross and Gabriel Hanganu (2013). *Governance models*. URL: http://oss-watch.ac.uk/resources/governancemodels (visited on 03/06/2019).

Gluon HQ (2016). *Scene Builder Documentation*. URL: https://docs.gluonhq.com/scenebuilder/ (visited on 21/05/2019).

– (2018). *Gluon Mobile Documentation*. URL: https://docs.gluonhq.com/charm/5.0.1/ (visited on 21/05/2019).

Klishin, Michael (2013). *How to Make Your Open Source Project Really Awesome*. URL: http://blog.clojurewerkz.org/blog/2013/04/20/how-to-make-your-open-source-project-really-awesome/ (visited on 05/06/2019).

Krogh, Georg von, Sebastian Spaeth and Karim R Lakhani (2003). «Community, joining, and specialization in open source software innovation: a case study». In: *Research Policy*. DOI: https://doi.org/10.1016/S0048-7333(03)00050-7. URL: http://www.sciencedirect.com/science/article/pii/S0048733303000507.

Kuhn, Thomas and Olivier Thomann (2006). *Abstract Syntax Tree*. URL: https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html (visited on 09/04/2019).

Martin, Robert C. (2002). «Visitor». In: *Agile Software Development, Principles, Patterns, and Practices*. Pearson. Chap. 29, pp. 525–558.

McDonald, Nora and Sean Goggins (2013). «Performance and Participation in Open Source Software on GitHub». In: *CHI '13 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '13. Paris, France: ACM, pp. 139–144. ISBN: 978-1-4503-1952-2. DOI: 10.1145/2468356.2468382. URL: http://doi.acm.org/10.1145/2468356.2468382.

Moir, Kim (2014). «Eclipse». In: *The Architecture of Open Source Applications*. Chap. Eclipse.

Nyman, Linus and Juho Lindman (2013). «Code Forking, Governance, and Sustainability in Open Source Software». In: *Technology Innovation Management Review* 3.1, pp. 7–12. URL: https://search.proquest.com/docview/1614473085?accountid=12870.

Oracle Corporation (2017). *The Java Tutorials*. URL: https://docs.oracle.com/javase/tutorial/ (visited on 21/05/2019).

– (2019). *Java documentation*. URL: https://docs.oracle.com/en/java/javase/12/docs/api/index.html (visited on 11/05/2019).

Pazdera, Radek (2015). *Spreading the word about your open-source project*. URL: https://radek.io/2015/09/28/marketing-for-open-source-projects-3/ (visited on 05/06/2019).

Petrov, Andrey (2016). *How to make your open-source project thrive*. URL: https://about.sourcegraph.com/blog/how-to-make-your-open-source-project-thrive-with-andrey-petrov (visited on 05/06/2019).

Raymond, Eric (1999). «The cathedral and the bazaar». In: *Knowledge, Technology & Policy* 12.3, pp. 23–49. ISSN: 1874-6314. DOI: 10.1007/s12130-999-1026-0. URL: https://doi.org/10.1007/s12130-999-1026-0.

Sharp, Helen, Yvonne Rogers and Jenny Preece (2007). *Interaction Design: Beyond Human-Computer Interaction*. 2nd Edition. John Wiley & Sons.

Smith, Donald (2018). *The Future of JavaFX and Other Java Client Roadmap Updates*. URL: https://blogs.oracle.com/java-platform-group/the-future-of-javafx-and-other-java-client-roadmap-updates (visited on 23/05/2019).