Enhancement Proposal

IntelliJ IDEA

EECS 4314
Dr. Jack Jiang
December 2, 2019

Report Prepared by Lamport's Legion:

Paul Sison
Connor A
Aya Abu Allan
Jonas Laya
Jeremy Winkler
Damanveer Bharaj
Connor D

Table of Contents

Table of Contents **2**

1.0 Abstract **3**

2.0 Introduction & Overview **3**

3.0 Proposed Change & Background **4**

4.0 Motivation **5**

5.0 Approach **6**

5.1 Approach One 6

5.2 Approach Two 6

5.3 Decision 7

6.0 Required Changes **7**

6.1 High-level 7

6.2 Component Level 9

7.0 Testing Plan **11**

8.0 Impacts **12**

8.1 Intro **12**

9.0 Risks **15**

10.0 Lessons Learned **16**

11.0 Conclusion **17**

11.0 References **18**

1.0 Abstract

The proposed change of this report is to switch from using Swing to JavaFX for the UI framework. The main benefit that comes from this switch is the ability to adhere to the MVC design pattern. Many benefits come with the introduction of the MVC pattern which includes separation of concerns, maintainability, collaboration, and design of the UI. There are three main approaches we could take to implement this switch. Two of which are to take an incremental approach. The first incremental approach is to do a top-down migration. The second is to do a bottom-up migration. The final approach is to do a full rewrite of the system. The full rewrite approach was decided to be too heavy of a task to do at once, so we decided to go with the incremental approach. Of these, we decided to go with the top-down approach since it was thought to be easier since some Swing components might be difficult to migrate to JavaFX, so those can be left to later to be redesigned. The high-level changes required would be relatively small, but since the goal is to adhere to the MVC design pattern, as we are doing the top-down approach, every UI component will be separated into its own folder which contains the separation of model, view, and controller.

During the migration phase, there will be a considerable amount of Swing and JavaFX components running in parallel. In order to allow these components to work together, a middle-man called the Runnable interface will be introduced. Thus, causing an increase in Runnable file dependencies in the Platform architectural components. However, these dependencies are expected to fade away by the end of the migration. As this is a large migration, a separate build may be necessary to ensure no significant issues arise. As Swing is outdated and JavaFX is open source, JavaFX's lifetime will most likely be much longer. The adherence to the MVC design pattern and the improved look and feel of JavaFX apps will also lead to a better product. Many of the files that depend on Swing are related to the component-based architecture, which is an integral part of the IntelliJ system, so it is risky. The plugin dependency on components and the UI is also a potential risk of the system. The main stakeholders include IntelliJ's stakeholders, users, IntelliJ developers, plugin developers, competitors.

2.0 Introduction & Overview

The purpose of this report is to look at a potential change that could be implemented into the IntelliJ platform and look into some details related to the change. The requirement for the proposed change was to not be already part of IntelliJ's plan of changes.

We will start by going over the proposed change and some background knowledge that will help to understand the different sides of the change. Then we will go deeper into what the benefits are that come from this change. The benefits received from this change are general programming practices that are useful, especially in large software systems such as IntelliJ.

Then we will go over the three main approaches we found to make this change, discuss some benefits and detriments of each approach, then make a decision about which approach to take. The decision will give some explanation of why the approach was given and compare some of the benefits of each approach. From here, the high level and component level changes will be investigated and we will look at the overall system to see what components from the concrete architecture that will have to change while implementing the approach. We will start by looking at the negative impacts that may occur to the system while actually implementing the system

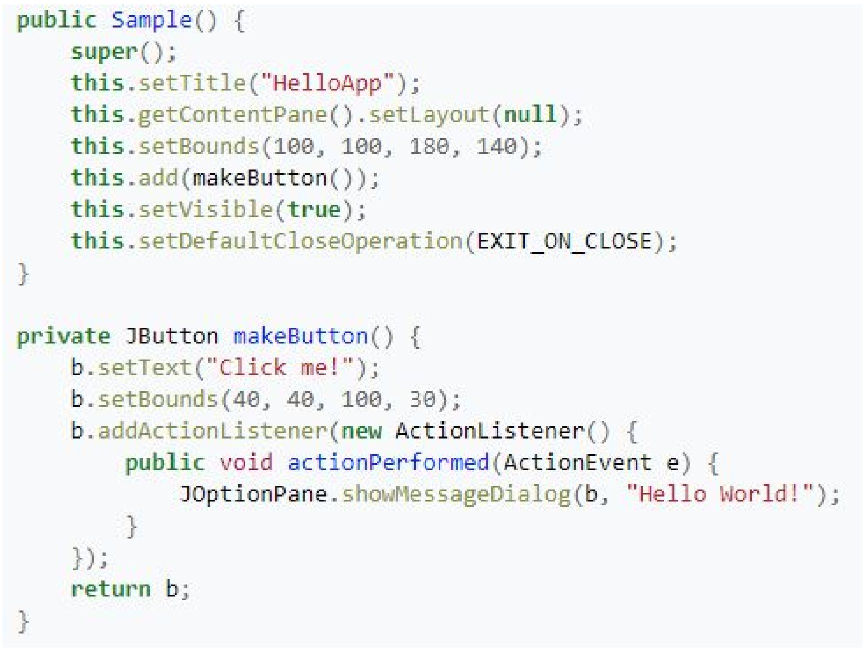
From here, we will move onto looking into the impacts this change will make to the system. Then, using some of the tools we created for the last assignment, we will look more into the dependencies of the system to see some of the changes needed and look more into the benefits that will come from the impact. After, we will go deeper into some risks and identifying stakeholders.

The overall change proposed is to move from Swing to JavaFX to adhere to the MVC design pattern. Many benefits come with this change including separation of concerns. The approach we decided on was an incremental top-down approach since it is the most viable for this system. The changes required include moving to the MVC pattern and migrating the old Swing components into JavaFX components. JavaFX was seen to be better since Swing is outdated, JavaFX is open source, it adheres to MVC, and it has a better look and feel. Some risks seen are budget due to the size of the change and the unstable builds during the migration process.

3.0 Proposed Change & Background

The change we are proposing is to switch from Swing to JavaFX. Both Swing and JavaFX are front end frameworks used to develop Java-based UIs. Swing was a framework developed over 20 years ago and due to its age, it does not fit with modern programming practices. The main design pattern we are looking at for this report is MVC, and it is the reason we are proposing the switch from Swing to JavaFX.

The main concept of MVC is that the program is separated into three main components; the model which contains the data, the view which displays the data, and the controller which updates the model. The reason why we are proposing this change is due to the fact that Swing cannot properly follow the MVC pattern. Looking at the code below, we can see that things like layout, size, and other properties of the components are described by the same code where the controller is being defined. So, when using Swing, the controller and view are tightly coupled to each other. This makes the code not follow the MVC pattern that we are proposing.

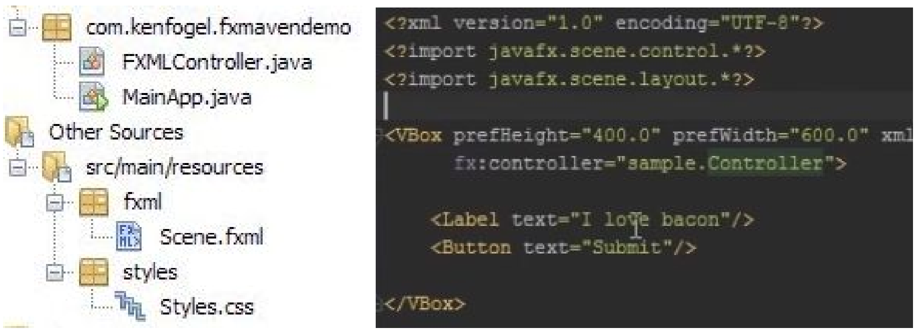


```java
public Sample() {
    super();
    this.setTitle("HelloApp");
    this.getContentPane().setLayout(null);
    this.setBounds(100, 100, 180, 140);
    this.add(makeButton());
    this.setVisible(true);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

private JButton makeButton() {
    b.setText("Click me!");
    b.setBounds(40, 40, 100, 30);
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(b, "Hello World!");
        }
    });
    return b;
}
```

Figure 1: Code snippet of basic Swing UI

When looking at the JavaFX, we can completely separate the view, model, and controller into separate files to help with the MVC design pattern. As we can see in the folder structure below of a JavaFX app, the controller is easily separated from the view, which is the fXML and CSS files. This approach is very similar to the most common UI development, web development. Similar to web development, JavaFX has fXML instead of HTML to represent structure and css to represent the look, which together makes up the view.



```
com.kenfogel.fxmavendemo
    FXMLController.java
    MainApp.java
Other Sources
src/main/resources
    fxml
        Scene.fxml
    styles
        Styles.css
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox prefHeight="400.0" prefWidth="600.0" xml
      fx:controller="sample.Controller">

    <Label text="I love bacon"/>
    <Button text="Submit"/>

</VBox>
```

Figure 2: File structure of JavaFX and example fxml file

4.0 Motivation

As discussed before, with the change from Swing to JavaFX, comes the benefit of adhering to the MVC design pattern. One main benefit of MVC is the separation of concerns. In the MVC design, we have three main components which are completely separate concerns, so it makes sense to keep them separate. With Swing, as shown before, the controller and view are very tightly coupled when they shouldn't care about each other. With the change to JavaFX, it is easy to separate each concern into its own file.

Something that comes from the separation of concerns is the increased maintainability of the system. Since each component is clearly separated into smaller components, the system becomes easier to maintain since the use of each file is clearly defined. This maintainability is something especially important when dealing with such a large scale application such as IntelliJ.

Collaboration is another benefit that comes with this switch. With the increased separation, it becomes easier for developers to work on the same section, but still on different files so things like merge conflicts don't occur as often.

Also, with most IDE's looking relatively similar, it becomes difficult for IntelliJ to differentiate themselves. By default, many JavaFX applications look better and are easier to customise due to the similarity to web development. Having a UI that can differentiate itself from the competitors, IntelliJ will have a big advantage.

5.0 Approach

5.1 Approach One

There are two intuitive methods on how to migrate IntelliJ IDEA's old Swing UI to the new JavaFX. The first is via an incremental approach, i.e. converting the Swing components in small chunks until the full migration is achieved. This approach can be further classified into top-down or bottom-up migration approaches. With top-down, the topmost component is migrated first to JavaFX and the smaller Swing component follows suit. The SwingNode adapter class helps in this process allowing developers to embed Swing components into the migrated JavaFX window. Bottom-up migration, on the other hand, is the complete opposite. For bottom-up, migration begins with the low-level components followed by the higher-level ones that depend on them, and so forth. The JFXPanel adapter class is used so the converted low-level JavaFX components can be embedded to the original Swing application. However, even with the provided adapter classes, interoperability between the two graphics libraries is far from perfect. For every migrated component, all its higher (if bottom-up) or lower-level (if top-down) dependencies also have to be considered. This creates development overhead for work that will eventually be discarded when the whole application has been fully migrated. The SwingNode class also has known crashing issues when the Swing component being embedded is not lightweight. Performance and aesthetics also take a hit when JavaFX and Swing are mixed in a single application. On the bright side, rollbacks are more manageable when required because the changes made to the source code are relatively small each time.

5.2 Approach Two

The second approach is full rewrite, i.e. all the Swing UI components are converted to JavaFX in one go. This can be achieved by identifying and isolating every Swing-dependent component in the IDEA app and converting those into JavaFX if a 1-1 equivalence exists, e.g. JButton to Button, JLabel to Text, JPanel to Layout, etc. If not, then the component has to be redesigned using similar mechanisms made available in JavaFX that would deliver the same functionality. Whenever a component has been converted, all dependencies that previously relied on it now have to point to the migrated version. The obvious benefit of this approach is that the interoperability issues no longer become a concern, and it becomes more probable that the end product will be more stable. However, being a complete overhaul, the redesign phase alone would take more time and effort.

5.3 Decision

If we were to put ourselves in the shoes of the leadership behind IntelliJ, it is a given that the company should take an approach that will keep us competitive in the IDE market. Full rewrite is labour-intensive and will take away manpower that instead can be assigned to make improvements in other areas of the IDE. While both approaches are technically challenging, an incremental approach is relatively easier to implement and debug since we are only changing a small portion of the code at a time. And between the two incremental approaches, top-down migration would be more preferred because some Swing components will not be easily migratable to JavaFX, in which cases, we can embed them in using SwingNode while a redesign or a workaround is being implemented.

6.0 Required Changes

6.1 High-level

There are not many high-level architectural changes anticipated with our proposal. The migration process would need to preserve the existing functionality of the system so change is reserved for the UI component. Some layers of the architectural structure of IntelliJ will be affected but only in terms of replacing Swing dependencies to JavaFX.

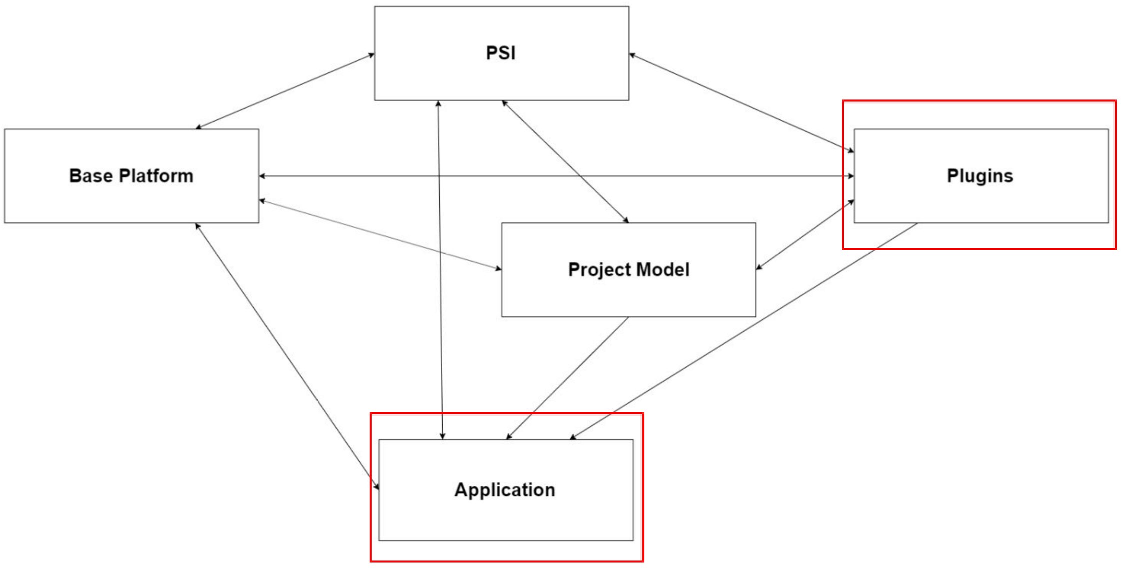


Figure 3: High-level concrete architecture with affected components

As discussed earlier, much of the motivation behind this upgrade is to adhere more closely to the MVC design pattern. With Swing, IntelliJ's user interface complies with an alternate version of it called the model-delegate pattern where the viewer and controller are combined into a single element:

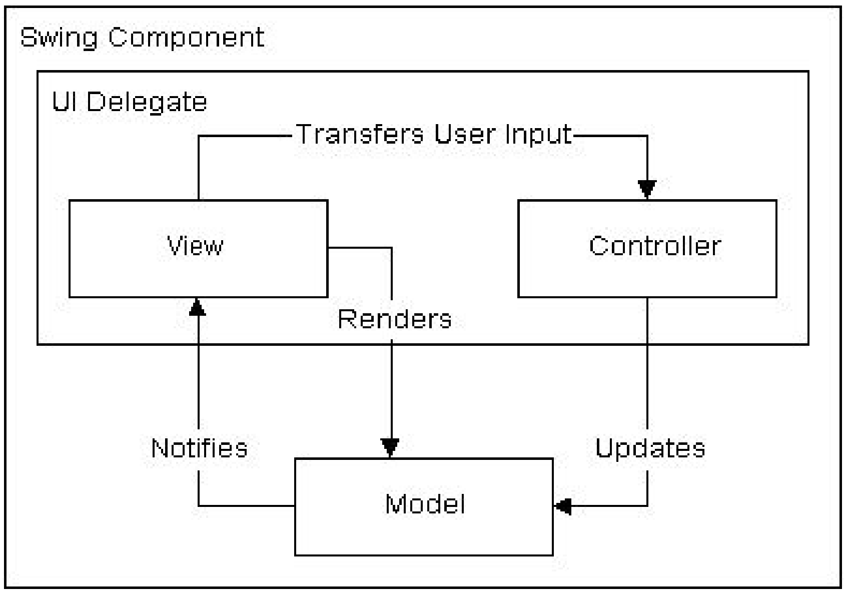


Figure 4: Swing's model-delegate design pattern

Upon switching to JavaFX, IntelliJ will be capable of more consistent MVC support across its components.

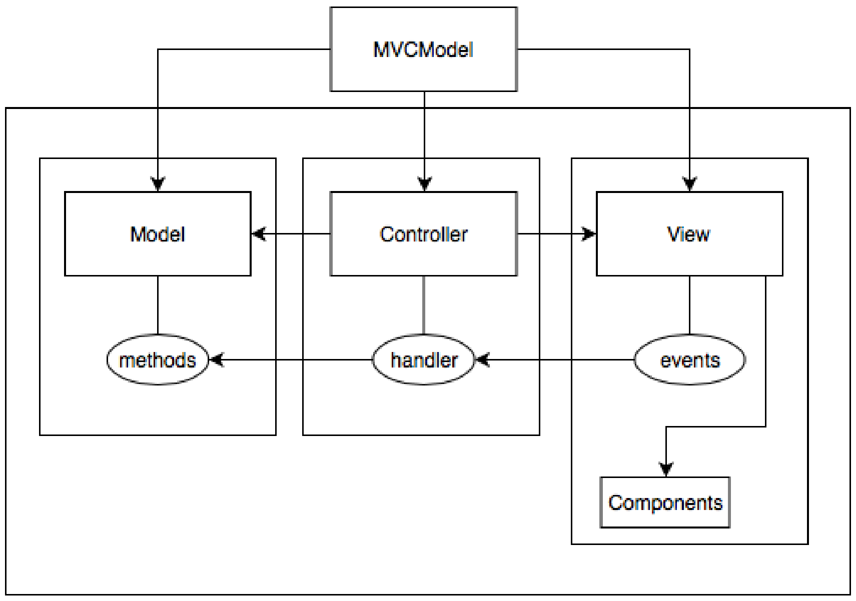


Figure 5: MVC design pattern model in JavaFX

6.2 Component Level

Ideally, the goal of this project is to replace all of the existing swing dependencies found in IntelliJ IDEA with its JavaFX counterparts. Given the scale of software (a fully developed software that's been pushed in the market for years), such a task seems troublesome at the least. It will most probably be the case that the software will not undergo a 100% conversion to JavaFX, due to compatibility and function preservation issues. However, a successful migration will get us reasonably close to 100%. Our approach for completing this task means that during migration, the IntelliJ software will be running both Swing and JavaFX components simultaneously. Even though migration will be done on groups of code, cross-communication between the Swing and FX components is expected.

Using an architectural extraction technique, scrML, it was discovered that IntelliJ IDEA IDE has over 3600 files being dependant on some branch of javax.swing. Further analysis indicated that most of these dependencies originate from 2 components of the IntelliJ architecture, the Platform and Plugin components. This is reasonable since the Platform component was found to contain the message passing functionality of the software. UI support is heavily dependant on such transfer of data IO.

The next step is to study the exact implementation changes that will be performed on the components containing swing dependencies. Research into Swing and JavaFX indicated that given a running application, all swing components of the application are placed onto an Event Dispatch Thread (EDT), and all JavaFX components are placed on a JavaFX User Thread [1]. On a software application running both Swing and JavaFX components, these two threads will run in parallel with each other. Any operations performed from/to a particular UI component must occur on the respective thread [1]. For example, if a user makes a change on a swing component that triggers data manipulation on a JavaFX component, the change on the JavaFX component must not come directly from the EDT (Swing thread).

Instead, a middle-man between the EDT and JavaFX thread is used to cross communicate. This middle-man is known as the Runnable Interface.

The runnable interface is an important Java functional interface that allows packets of work to be completed as a new thread that runs concurrently with the application [1]. The use of the runnable interface to allow communication between the two UI threads means that during the migration, we will expect to see a quadratic change in the number of Runnable dependencies being presented in the Platform architectural components.

This is better seen in a step by step description of how the runnable interface works during migration. During migration, a component of the software with Swing dependencies is replaced with its JavaFX counterparts. This then presents a decrease in the number of Swing file dependencies and an increase in the number of JavaFX file dependencies. The new software chunk is then placed back into the overall software. Now, this software will run both the EDT and JavaFX User thread. To allow communication between the new JavaFX code and the remaining Swing application, the two parts need to be stitched back together using Runnable threads. This in return, will increase the number of Runnable File dependencies between the connecting software components. At some point in the project, the number of JavaFX components will begin to outweigh the number of Swing components. Causing a peak in Runnable dependencies. At the end of the project migration, the number of Swing dependencies will be close to none and JavaFX dependencies at its highest. This means that eventually, the number of Runnable dependencies will eventually decrease and disappear. Thus causing the quadratic change in runnable file dependencies. As a result, we will see a minimal change in the overall software architecture as described in the high-level architectural changes section earlier in this report.

The following figure is a sequence diagram showing the process of how the EDT and JavaFX thread works together in parallel during runtime.

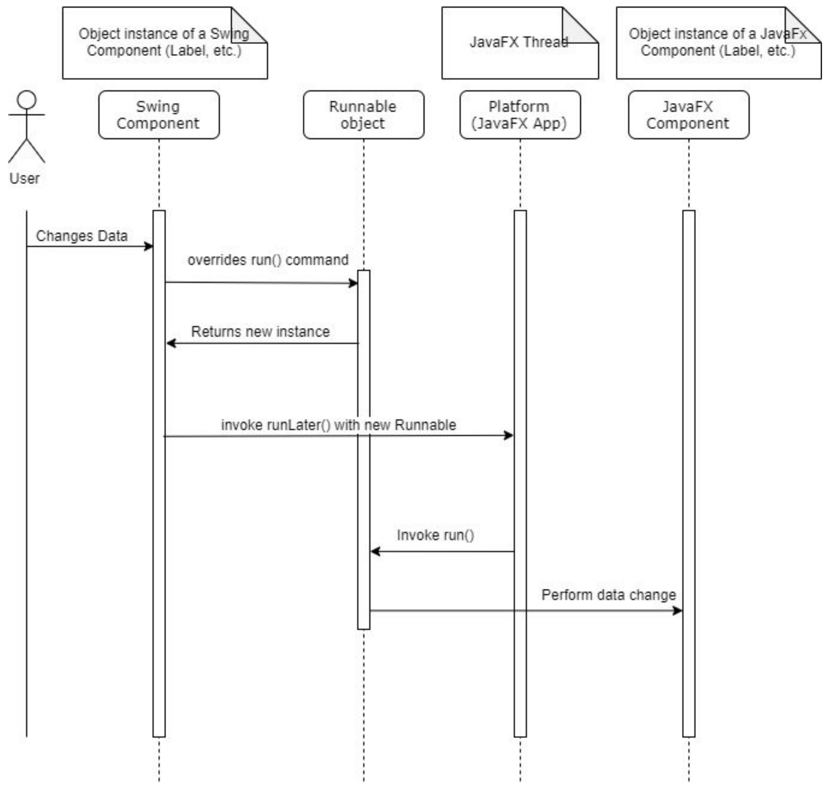


Figure 6: A user changes data on a swing component triggering a change in a JavaFX component

In this figure, we can see that a user makes a data change on a Swing component (I.e. JButton, etc), which triggers a change on a JavaFX component (I.e. JavaFX Label, etc.). What occurs here is that the Swing component will create an instance of a new Runnable object. This runnable object will have its run() command overridden with the desired task to be performed on the JavaFX component. The Swing component calls the runLater() command of the JavaFX thread with the runnable as its parameter. When the JavaFX thread is free to run the runnable (Concurrency now being a factor), it will perform the task onto the JavaFX component. This, in turn, allows cross-communication between Swing and JavaFX.

7.0 Testing Plan

Migrating a system presents a large challenge, but at least for testing there is an advantage in having an “oracle” - we know what the expected behaviour and outcome of the system should be based on how the existing system responds to a given set of inputs.

Additionally, maintaining strict interface requirements between classes means that developers should be able to reuse parts of old test cases to test the JavaFX components as they had the Swing components. In some cases, there may be some issues however and they may need to write some new test cases as well. One main issue, however, is that IntelliJ provides no guidelines on testing UI components; in fact, they practice manual testing of Swing components as stated on the SDK website [6].

If we were working at IntelliJ, I think we would see this as a good opportunity to introduce guidelines for testing UI components, since creating regression tests for UI components will pay off heavily as the number of manual hours of testing goes down. Our testing plan includes four sections: unit testing, performance testing, automated UI testing, and test coverage analysis.

Unit Testing

Unit testing will remain relatively similar to before. Many JUnit tests that would have been created for Swing components can be reused for JavaFX so long as the interface has stayed the same. Most of the rewriting here will include updating the event listeners, and what parts of the new events the tests are looking for since Swing and JavaFX produce different events.

Performance Testing

Since we are also making a shift in the underlying concurrency (JavaFX and Swing run on different threads), we also need to create some benchmark test cases in order to ensure that the new JavaFX version is actually integrating well without causing performance issues (and ideally, improving the results).

Automated UI Testing

This area has a lot of work that could be done here. Adopting a framework such as Abbot or something more advanced like Squish could pay off heavily in the long run. Tests would still have to be written for testing use cases (e.g. a user going through and clicking things, resizing, etc.), and compared against the expected results. But once these tests are built, they remove the need to manually test every component after any change to a section of the UI.

Test Coverage

Finally, an important part of any testing system is analyzing test coverage. Using a tool like EclEmma is essential to ensure that we have (as much as possible) exhaustively tested all of the code that is being pushed to production.

Since we are adopting a bottom-up approach, it makes sense to also introduce new testing practices incrementally as well, and slowly build up the repository of test cases. This integrates well with the bottom-up approach since if I have already re-written a nested component in JavaFX, and am now working on a parent component, I can run all of the automated tests for the child to ensure the parent hasn't messed anything up.

8.0 Impacts

8.1 Intro

So when we talk about this Swing to JavaFX migration with words like adapters and cross-compatibility coming up, it can sound like a simple project. The reality though is that regardless of design pattern and framework features, migrating a framework so entrenched in the IntelliJ platform is almost never that simple. While FX is able to extend and use Swing components, that doesn't mean *it is* Swing. It's still an entirely different entity with its own usages and paradigms. This means that these adapters and cross-compatibility messages we've been doubling down on between the two frameworks won't always be the IntelliJ's saving grace. Not all the Swing components and structures will always translate nicely to FX.

If IntelliJ were to invest in a major architectural change like this, the repository where developers work on this change will most likely be far from stable. This means that the development team will most likely break off into different feature paths, some working on the migration branch, and some of the original build. This is because the original build will retain its functionality, while the migration build's functionality will be questionable.

8.2 Getting Deeper

In order to properly assess the impact of a framework change like this, it's a good idea to look at how much of the system depends on said dependency. One of the iterations of our dependency extraction tool from the previous assignment was known for picking up too many * imports (full packages), and this was originally noticed by picking up Java Swing as a large dependency of the system.

Since we knew this build picked up Swing, we used its dependency .TA file that it output to look at what components of the system depended on Swing. When we looked into this list of dependencies, we started to see the word "Component" in the title of a lot of the Java classes depending on Java Swing. You can see this in **Figure 7**.

```
1477   PathMappingsComponent.java
1478   PathMappingsComponent.java
1479   PathMappingsComponent.java
```

Figure 7: An example of a file depending on Java Swing with the word "Component" in the name

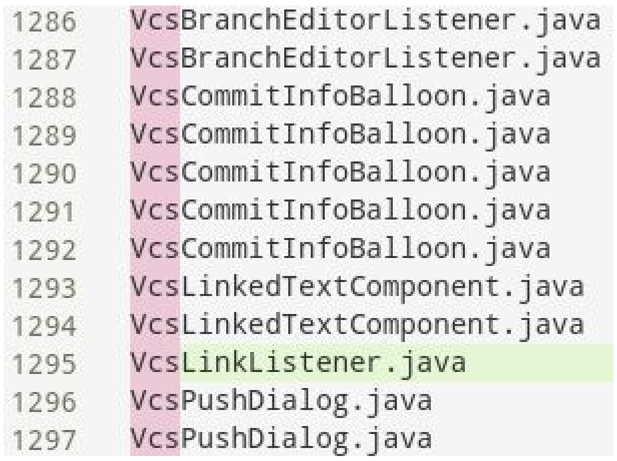
The thing about finding that a lot of "Component" based classes import Swing is a bit intimidating, seeing that IntelliJ's overall architecture is described by IntelliJ as a "Component-based Architecture".

8.3 Components

Finding these dependencies means that in order to carry out this framework migration, the "cornerstone" architecture of IntelliJ needs to be rebuilt from the ground up. This makes this project risky because the Component-based model is what makes the IntelliJ UI modular, maintainable and extensible.

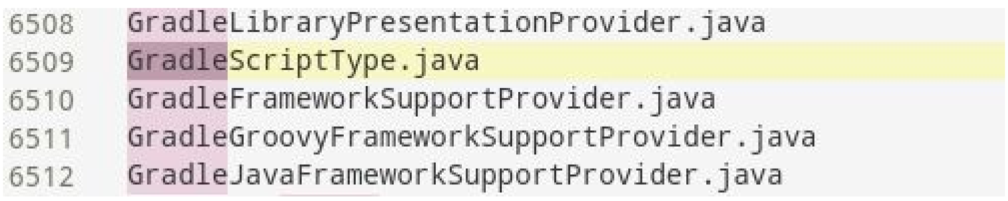
8.4 Plugins

The IntelliJ documentation states that Components are the fundamental concept of plugin integration. Pretty much any plugin embedded in the IntelliJ platform leverages the Component model, and by extension, Swing. This includes any tooling used in the IDE (Build tools, VCS, Testing frameworks, Debuggers) as well as the individual frameworks that fall under those categories.



```
VcsBranchEditorListener.java
VcsBranchEditorListener.java
VcsCommitInfoBalloon.java
VcsCommitInfoBalloon.java
VcsCommitInfoBalloon.java
VcsCommitInfoBalloon.java
VcsCommitInfoBalloon.java
VcsLinkedTextComponent.java
VcsLinkedTextComponent.java
VcsLinkListener.java
VcsPushDialog.java
VcsPushDialog.java
```

Figure 8: An example of some VCS system files that depend on Java Swing



```
GradleLibraryPresentationProvider.java
GradleScriptType.java
GradleFrameworkSupportProvider.java
GradleGroovyFrameworkSupportProvider.java
GradleJavaFrameworkSupportProvider.java
```

Figure 9: An example of some Gradle files that depend on Swing, Gradle is an example of a Java build tool that's built into IntelliJ.

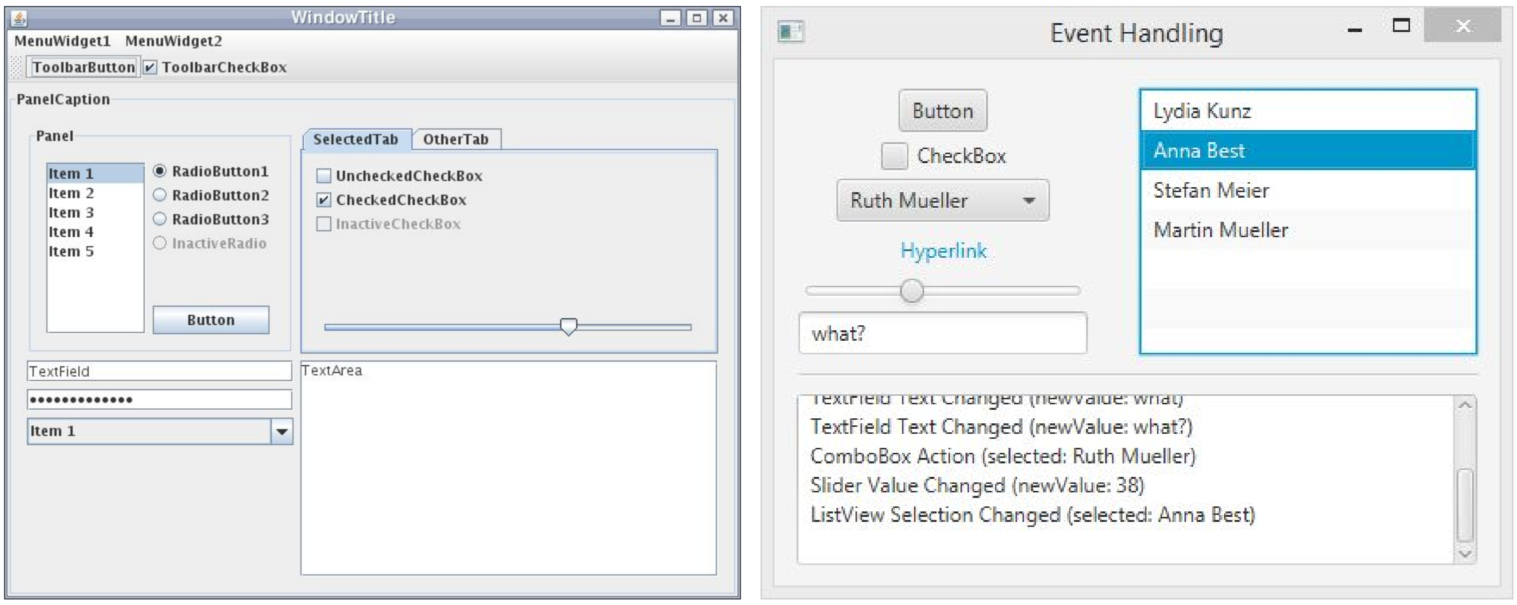
8.5 Positives

The largest positive of this framework migration is the support and ecosystem surrounding each individual framework. Java Swing was first developed in 1996 and has been mostly static for over a decade. Improvements aren't being made to the framework, just small stability enhancements and porting to newer operating systems.

JavaFX was initially developed in 2006, and it went Open Source in 2011. Since JavaFX isn't tied to a corporate entity, its development cycle will (theoretically) never end, and

IntelliJ can even get involved in the framework's development. JavaFX also comes with a much more maintainable UI framework since it supports the full Model-View-Controller design pattern.

The last really important note about JavaFX is the look and feel of a **default** FX application. Since an FX application looks good/modern right out of the box, a lot less work is done on the cosmetics of the application, and more work is done on the functionality of the application itself.



```
TextField Text Changed (newValue: what)
TextField Text Changed (newValue: what?)
ComboBox Action (selected: Ruth Mueller)
Slider Value Changed (newValue: 38)
ListView Selection Changed (selected: Anna Best)
```

Figure 10: An example of a Swing GUI beside a JavaFX GUI, both with minimal styling improvements.

9.0 Risks

9.1 Stakeholders

The primary stakeholders in the IntelliJ platform are the following groups. When we assess the risks of the system, it's important to keep them in mind as we look into them. The important stakeholders that we've identified are the following:

- Jetbrains stakeholders
- Users
- Jetbrains development team
- IntelliJ plugin developers
- Competition for IDE platforms

9.2 Budget

So when we talk about a company like Jetbrains, it's important to keep in mind that Jetbrains is a company with a finite amount of resources. This type of migration is massive, and it could end up being the most expensive project the company has had to date. For a company like Jetbrains, if they invest in a project like this, the future of the company may depend on it. If the migration doesn't pan out to be as good as the original product, Jetbrains as a company could make a similar plummet that Netscape did when they rewrote their browser.

9.3 User Base

The primary goal of the IntelliJ platform is to retain one of the largest market shares in development IDEs. To continue to hold their existing market share, users have to be satisfied with the product they're being provided. If an IntelliJ user tries out the newest version of JavaFX and doesn't like it, they're going to move to another IDE or code editor. The migration to JavaFX needs to have little to no negative impact in order to grow or maintain its user base.

IntelliJ isn't a code editor like its competitors VSCode, Atom or Sublime. It is an **IDE**. Users who use an IDE expect it to be much more capable than a typical code editor. Some users who are newer to programming may not know how to use their development tooling without IntelliJ. This means that if the component model is rebuilt, the plugins will be affected. This effect needs to be accounted for to retain all functionality, or there is a large risk of losing users.

10.0 Lessons Learned

"Cross Compatibility"

Despite the claims that JavaFX and Swing are fully cross-compatible, it seems that for larger applications like IDEA there will be some harder issues to address to be fully cross-compatible. The main issue identified was that they run on different threads, causing concurrency/communication issues. Additionally, this means that to be part of the overhaul team, you would require background knowledge of threading to understand how they actually are cross-compatible. The main lesson learned here is that cross-compatibility is never that easy.

MVC Reaffirmed

After looking at the differences in how JavaFX files are developed over Swing, we saw some great benefits in over architecture that would help IDEA. The main benefit comes in because Swing has the Model and View intertwined, whereas JavaFX is full MVC. This reaffirms the good software design patterns learned in the first couple of years of programming, and how these design patterns can have huge impacts when looking at programs with millions of lines of code.

Long Term and Large Scale Impacts

Another lesson we’ve learned is that it’s impossible to account for all the possible issues that will come up because in the end, you can never be fully sure what will be able to be converted and what cannot. Additionally, something like this overhaul may at first seem *possible* (if still a titanic task), but after looking into the details, it’s pretty clear why IntelliJ hasn’t attempted this. Although it seemed like a great idea to start with (modernizing the front-end), it’s clear it’s far too large of a task to tackle now.

11.0 Conclusion

In conclusion, our team has decided that this extension of IntelliJ IDEA would actually not be a good direction to go. Our suggested approach of bottom-up implementation would allow for the progressive adoption of JavaFX; however, it would also fracture the development teams into two parts - those responsible for maintaining old Swing components, and those responsible for updating and maintaining new elements. The testing of these new components would also be fractured, and in the end, this would be a massive undertaking for the company that would be extremely expensive. If they fell behind or slipped up at any point by running into any kind of cross-compatibility issue with popular plugins, they could leave themselves very vulnerable to their competition. Although originally we all thought this would be a cool way to modernize the UI, we realize now it’s far too late for IntelliJ to adopt JavaFX, and they will continue to be married to Swing for the foreseeable future.

11.0 References

[1] “Integrating JavaFX into Swing Applications”, “Oracle”,
https://docs.oracle.com/javafx/2/swing/swing-fx-interoperability.htm

Images:
https://en.wikipedia.org/wiki/Swing_(Java)
https://code.makery.ch/blog/javafx-8-event-handling-examples/
https://dzone.com/articles/nb-class-maven-2-kf
https://www.youtube.com/watch?v=K7BOH-Ll8_g
https://www.educba.com/java-swing-vs-java-fx/
https://www.jetbrains.org/intellij/sdk/docs/basics/testing_plugins.html