Lessons Learned in Migrating from Swing to JavaFX

Martin P. Robillard and Kaylee Kutschera

Technical Report CS-TR-2018.2
School of Computer Science
McGill University
Montréal, QC, Canada

5 November 2018

Abstract

We describe our experience with the migration of a diagramming tool written in Java from the Swing Graphical User Interface framework to the more recent JavaFX framework, in the general context of research of software evolution. The experience led to a number of unexpected realizations about the impact of subtle differences in the design of framework features and their documentation.

Article

Even the most risk-adverse project leaders will eventually face the question of whether to migrate to a new framework. This question can be filled with dread because the number of things that can go wrong when migrating to a new framework is basically infinite. We recently faced this question in the evolution of JetUML, an open-source diagram editor we develop and maintain for teaching and professional use.[1] We undertook the complete migration of the tool from one GUI toolkit to another with the perspective of both modernizing the software and learning about the major migration challenges. In the end we successfully completed the migration at the cost of approximately 3 person-months. The experience led to a number of unexpected realizations about the impact of subtle differences in the design of framework features and their documentation.

A Brief History of the Project

JetUML is a medium-sized, pure-Java desktop application to create and edit diagrams in the Unified Modeling Language (UML). The project started in January 2015 as an offshoot of the original version of the Violet diagram editor.[2] Although Violet was itself spun-off as an open-source project, the first author launched JetUML to focus exclusively on a minimalistic set of features intended to make diagramming as quick and seamless as possible. The main usage

scenario for JetUML is live diagramming, that is, the creation and modification of diagrams as part of lectures, design reviews, and other similar types of presentations. Thus, the application relies heavily and critically on its graphical user interface framework, which was originally AWT/Swing.

Before the migration, JetUML consisted of 9.1k non-commented lines of Java source code (LOCs) and 1.7k lines of comments distributed over 83 sources files organized in five packages (the data is for Release 1.2). The project was also supported by a suite of 255 JUnit tests comprising 6.3k LOCs. Figure 1 illustrates some of the salient points of JetUML's architecture related to the migration effort. From the diagram, we can distinguish three layers of architectural elements. The necessary windowing elements of the GUI framework are grouped in the layer named "Swing". These are subclassed by the application, as in most cases of framework usage, resulting in a group of elements that represent what we refer to as the windowing, or "high-level" design elements (`EditorFrame`, `GraphFrame`, and `GraphPanel`). The bottom ("low-level") layer consists of the application classes necessary to construct and draw various diagrams. Although not shown on the figure, the types `Graph`, `Node`, and `Edge` are extensively subtyped in the application with concrete elements (e.g., `ClassDiagramGraph`, `DependencyEdge`, etc.).

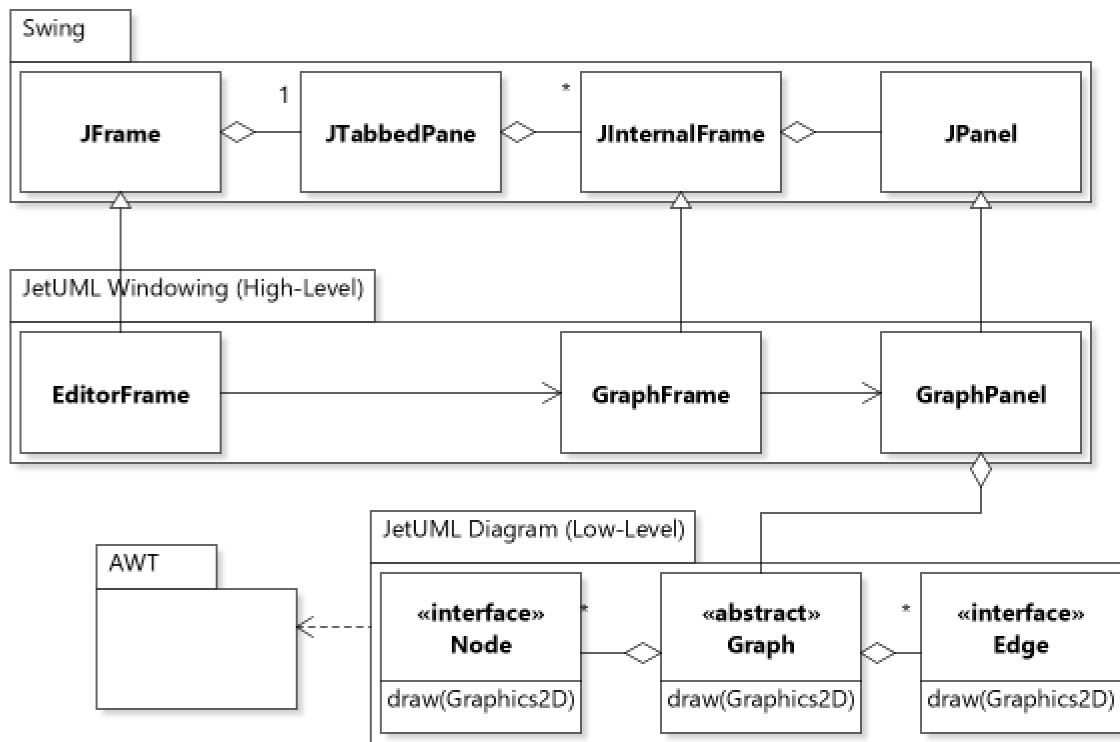

Figure 1 Architecture of JetUML Prior to the Migration. The diagram also illustrates the output of the tool.

Because JetUML was derived from Violet, almost all of Violet's architectural decisions were initially retained for JetUML and proved to be valuable in the long term. However, two particular decisions turned out to be problematic: the tangling of the diagram structure

definition with its presentation, and the heavy use of reflective features. Both of these decisions made sense in the context of Violet, which was one instance of a generic framework for building different graph editor applications. In the case of a special-purpose UML editor, however, these decisions quickly transformed into a serious piece of technical debt. The first issue is exemplified by the definition of a method `draw` in the elements of the diagram layer. The issue with reflection is not illustrated, but caused the encoding of diagrams in XML to contain references to framework-dependent types.

Why Migrate to JavaFX

Because the amount of resources available to support the development of JetUML is minimal, the principle guiding its evolution is to minimize the risk of any event that would require heavy development effort. For this reason, the reliance on any major framework is avoided and the migration to JavaFX was continually put off. Ultimately, the main deciding factor for moving to JavaFX was simply the inevitability of the migration, for two reasons:

1. Adapting to hardware environments (high-DPI displays, multiple monitors) was becoming necessary, and with Swing no longer supported, it was inevitable that a future development would eventually render Swing-based applications terminally obsolete;
2. We were putting off a major cleanup of the code to eliminate technical debt because this cleanup was going to be necessary to adapt to a new framework anyways. But, as long as this inertia was not vanquished, every further change contributed to degrading the structure of the application.

Migration Process

We organized the migration in three phases: *Preparation*, *Migration*, and *Consolidation,* with the Preparation and Consolidation phases to be completed by the main project developer (the first author) and the Migration phase to be completed by the second author. This alternation in developers between phases thus created a strong requirement for the design to be understandable at the end of each phase.

In the Preparation phase, the first author refactored the design to isolate as much as possible of the code that relied on the Swing framework. This refactoring involved three major efforts:

1. Separating the view from the model for diagram elements (nodes and edges);
2. Converting all references to framework-dependent geometric objects (points, lines, etc.) from Swing classes to framework-independent specific classes.
3. Replacing the framework-dependent JavaBeans-based persistence with a framework-independent solution that used the JSON notation.

Interestingly, as part of the second step, we decided to convert from floating-point geometry to integer geometry, in an attempt to simplify the code, which turns out to have had unintended consequences. The result of the Preparation phase was released as version 2.0-alpha. It brought

the code base size to 12.2kLOC in 126 files with the support of 7.3kLOC of testing code in 310 tests. The 34% increase in the size of the code base was due primarily to the integration of a subset of a 3rd-party JSON processing library and the creation of numerous new classes to isolate geometry operations and separate the view from the model for diagram elements.

The main focus for the Migration phase was to migrate the code while changing as little as possible in the look and functionality of the tool, and to accomplish this goal in as incremental a way as possible given the features of the source and destination frameworks. Before starting the Migration phase, we searched the web for insights into the migration process and the likely problems we could run into. This investigation lead to a collection of articles, forum posts, videos, and reference documentation. Unfortunately, at that stage the technical advice proved either too specific (focusing on detailed uses cases) or to general (discussing broad issues such as threading). Ultimately we deferred more detailed background research until we faced concrete technical issues.

The result of the migration phase was released as version 2.0, which was almost identical to 2.0a in terms of size metrics.

Finally, the idea of the Consolidation phase was to solidify the migrated version with various cosmetic improvements, design simplifications, and adaptations made directly possible by JavaFX. The result of this phase was released as version 2.1, with 12.3kLOC in 142 files. The complete code base of all releases of JetUML can be obtained from its GitHub repository.[1]

Lessons Learned

The migration process led to a number of lessons learned, each derived from the discovery of insights we wished we could have had before starting.

Exact Correspondence in Class and Method Names Leads to a False Sense of Security

A major concern for a diagram editor is to draw shapes. Swing supports this functionality partly through a `Shape` class hierarchy, with subclasses such as `Arc2D`, `Ellipse2D`, `QuadCurve2D`, etc. In Swing, a `Shape` instance can be drawn on a graphics context simply by calling `context.draw(Shape)`. In our preliminary investigation of JavaFX, we quickly noticed that it defined a near-equivalent API, with also a class `Shape` with equivalent subclasses with the same name (except the `2D` suffix). This heartwarming realization lured us into thinking that migrating the drawing code would be a trivial exercise in mechanical translation, and the exact correspondence of names even made the need for advanced API migration mining tools superfluous.[3] Unfortunately, the feeling of elation was shattered when we realized that in JavaFX the graphics context object does not have a method to draw `Shape` instances, and that in fact in JavaFX, `Shape` instances are not used to draw shapes directly, but rather for a new purpose that did not exist in Swing (to place shapes in a scene). Consequently, the code had to be extensively refactored to adapt our old strategy (to create a `Shape` instance in various diagram element classes and draw it once), to one that was supported by JavaFX (namely, to

draw shapes in each diagram element class using available primitives such as `strokeRect` to draw a rectangle).

The lesson for API migration in general is that superficially equivalent framework features can hide big changes in API usage scenarios that can require redesign. For migration to JavaFX specifically, the similarities in shape type hierarchies actually hide a different approach to creating drawings.

Feature Redesign Leads to Cascading Impact

The adaptations we had to implement to account for the new way to draw in JavaFX were in fact a consequence of a different organization of the draw feature supported by the toolkit. In JavaFX, the creation of drawings from `Shape` instances can be done though two different mechanisms that operate at two different level of abstraction. The high-level mechanism involves creating instances of class `Shape` and adding them to a `Pane`, so as to constitute a collection of drawable objects. The low-level mechanism involves creating a drawing directly on a `Canvas` instance by using primitive drawing methods on the canvas's graphics context. Each of these mechanisms forms a somewhat polarized version of Swing's original drawing approach, which combined elements of both. For example, with the high-level mechanism, event handlers can be directly added to `Shape` instances and each instance also has its own Z-coordinate, which would allow us to easily move shapes on top of each other. On the other hand, unlike a `Pane`, `Canvas` is not resizable by default, but it has better performance because drawings are rendered directly as opposed to having `Shape` objects added to a scene graph, managed, and then rendered. Considering that performance is an important requirement of JetUML, we used a `Canvas` as the drawing method. Conceptually, this mechanism was also closer to the original way of drawing. Unfortunately, by splitting the original drawing feature into two variants, some of the desirable functionality of Swing had to be forfeited.

In the Swing-based version of JetUML, the drawing area is a resizable panel contained in a scroll pane, upon which shapes are drawn directly. In JavaFX we had to embed a `Canvas` in the pane because it is not possible to draw directly on container elements. This seemingly innocuous constraint led to a cascade of impacts on the design that reached the user-visible features. The trigger for the cascade was the requirement to make the canvas resizable. Technically, this only requires overriding a few methods. However, in experimenting with a resizable canvas, we ran into numerous layouting and sizing issues when trying to integrate it with a `ScrollPane` (a component that allows scrolling areas larger than the window size). Another notable difference when drawing on the canvas is that there are no methods such as AWT's `Component`'s `repaint()` to deal with drawing components and redrawing when the component becomes invalidated. Due to these subtle mismatches between requirements (a resizable and scrollable drawing area) and the effective support for these requirements through class `Canvas` and `ScrollPane`, one of the nicest features of JetUML, a diagram space that seamlessly adapts to

the window size, could no longer be reasonably supported: we gave up trying to support a seamlessly resizable diagram, which was natural in Swing, and ended up investing a considerable amount of redesign effort in rethinking how the tool would work with a fixed diagram size.

These important design changes between Swing and JavaFX are likely to have a major impact for most applications that involve 2D drawing, yet at preparation time this information remained invisible to us. In hindsight, one of the main lesson we draw from this issue regards our lack of awareness of the importance of our reliance on a resizable drawable area. Because this feature was so seamlessly supported by the framework, it had not been conspicuous as a functionality to experiment with during the Preparation phase. The general implication is to try to identify important features in the abstract, independently of their implementation.

Adapter Components are No Silver Bullets

Adapters are a classic strategy for incrementally migrating from one framework to another.[4, 5] The presence of adapter components in both Swing and JavaFX[10] could make one think that there is unlimited flexibility for defining an incremental migration that minimizes the impact of changes at every step. Specifically, with both Swing-to-JavaFX and JavaFX-to-Swing adapters, it can in principle be possible to follow either a strategy of top-down migration (migrated JavaFX windows containing legacy Swing widgets) or bottom-up migration (legacy Swing windows containing migrated JavaFX widgets), or any combination of the two that isolate changes and limits risk.[6] One popular answer on an on-line forum even claims that adapters make migrations to JavaFX “easy”.[7]

Unfortunately, many practical issues with adapters put a limit on this flexibility:

Performance: Top-down migration requires the use of the `SwingNode` adapter class, which can hold Swing content in JavaFX windows. This strategy unfortunately leads to performance issues because `SwingNode` is not meant to hold heavyweight components. In contrast, bottom-up migration using the `JFXPanel` (which can hold JavaFX components in Swing windows), does not have these problems. Hybrid migration strategies, such as interposing a JavaFX component between Swing components, can result in major performance problems such as large delays when first loading a window. These problems occur since JavaFX and Swing separately determine their layout which makes it difficult for the application to compute the appropriate sizing for all components.

Computing Dimensions: Because components in one framework are embedded in the other, we found that it was not possible to properly compute the preferred sizes of components from both frameworks. To address sizing problems, it is recommended to hard code fixed preferred sizes until they can be properly computed by the framework[6], which adds development overhead.

Concurrency: Threading complications are a by-product of Swing running on the AWT event dispatch thread (EDT) and JavaFX running on the JavaFX application thread. To modify a JavaFX component from the AWT EDT, we need to package the code as a `Runnable` functional interface and provide it to the JavaFX method `Platform.runLater`. Likewise, to modify a Swing component from the JavaFX application thread, we need to use `SwingUtilities.invokeLater(Runnable)` or similar. Both approaches involve additional overhead and clutter.

Look and Feel: During migration, JavaFX and Swing components will have different looks. These differences can be minimized by customizing JavaFX using CSS, thereby providing a more cohesive look throughout the migration process, however, this again causes additional overhead and clutter.

Dependency Cycles: During migration, it may be necessary to have cyclic dependencies between classes if it is necessary to access a parent component. These dependencies can be removed once child and parent are contained in the same framework and can access each other through the scene graph. For example, in JetUML, because we did a top-down migration, the tabbed pane was migrated to JavaFX before the drawing area it contains, which remained a Swing component. A reference to a diagram's tab was needed by the Swing drawing area to update the tab's title properly as the modification of a diagram can change the title on its tab. The JavaFX parent component was not accessible in the Swing child because there is no way to access the `SwingNode` instance that the child Swing component is wrapped in. Normally, to access a Swing component's parent, one would use `getParent()` which will return the parent Swing component. If a Swing component is wrapped in a `SwingNode` the `getParent()` call of the Swing component will not return the `SwingNode` it is wrapped in but a `null` value.

So, although adapter components make incremental migration possible, their use requires additional adaptive code that typically degrades the design and performance. A major consideration when using adapters is whether adapted code will be released or not. In our case, the use of adapters was strictly for between-release use. For this reason, most of the downsides, such as performance or different look-and-feel, did not have a visible impact on the production version. In the end we selected a top-down migration strategy so that we could migrate the more stable ("architectural") part of the design first, and defer the migration of the drawing code, which required more uncertainty and experimentation, to later, when the JavaFX-supported window structure was in place.

Counter-Intuitive Decision for Pixel Alignment Induces Post-Migration Rework

After migrating the drawing feature we noticed that the formerly razor-sharp rendering of diagrams we had experienced with Swing had been replaced by somewhat blurry diagrams. We initially blamed the problem on over-aggressive aliasing and set the issue aside for the initial JavaFX-based release 2.0. After the release and extensive experimentation on different displays, we concluded that the blurriness of diagrams was a major step back, and further

investigated the issue. This investigation revealed that in the JavaFX framework "At the device pixel level, integer coordinates map onto the corners and cracks between the pixels and the centers of the pixels appear at the midpoints between integer pixel locations."[8] This means that to have a point exactly map to a pixel and render sharply, this point needs to have coordinates (0.5, 0.5). This crucial piece of information is unfortunately buried in one paragraph of the class-level documentation for class `Node`. Even more confusing, a different paragraph in the class-level documentation for `Shape` describes the blurriness problem exactly, but the solution described is inapplicable for applications that use the low-level (canvas-based) drawing mechanism. In the latter case, it is the insight in class `Node` that applies, even though in that case no mention is made of blurriness. A Stack Overflow post[11] turned out to have been instrumental in helping us assemble the solution to this puzzle from disparate pieces.

In the end, we solved the problem by directing all shape drawing requests through static methods that simply shifted the original integer coordinates by 0.5 in each dimension. This solution was a bit dispiriting after a wholesale conversion to integer geometry, but ultimately it works well and has only minimal impact on the design and performance. Once we had the proper insights, implementing the solution was a straightforward task that we completed as part of the Consolidation phase. The general lesson is that although it's impossible to reason about information we don't have, the design decisions we make are indeed known, and it's possible to organize systematic investigation around them. So, although we were unaware of the floating-point pixel geometry issue, we had decided to move to integer geometry. In hindsight, it would have been a good idea to further investigate the ramifications of this design decision. The lesson for migrating to JavaFX is to specifically account for the new approach to pixel geometry.

Not All Obvious Features Are Supported

One of our goals for the migration, from which we were expecting to derive much satisfaction, was to complete a full migration, and completely shed any reference to the Swing framework. In the end we came within a hair's breadth of reaching the goal, but our hopes were dashed by the shocking discovery that JavaFX does not provide any functionality to save an image to a file. One of the key features of JetUML is to be able to export an image of the current diagram to a file in a standard image format (e.g., PNG). In Swing this can be done trivially with a call to `ImageIO.write(…)`, which takes as one of its argument an AWT rendered image object. After the usual research and experimentation, we concluded that there was no reasonable equivalent in JavaFX, and one must use `SwingFXUtils.fromFXImage` to convert from `javafx.scene.Image` to `java.awt.Image` which can then be used by `ImageIO` to write to a file. Thus, we will have to tolerate references to a legacy framework until this feature is added, and so will any JavaFX application that needs to save a JavaFX image to a file.

New JavaFX Features Naturally Replace Cumbersome Approximations

To end on a positive note, one pleasant surprise came in the Consolidation phase, where we refactored the code to replace our custom-built toolbar component with JavaFX's `Toolbar`, and our custom code for drawing drop shadows with JavaFX's shadow feature. The pleasant surprise was not so much that the new framework provided these obvious features, but rather how much legacy code was necessary to support them. For example, our version of the toolbar included some inelegant code required to reorganize the tools in the toolbar when the default layout would not fit in the display. This feature comes out of the box in JavaFX. Even more impactful was our decision to replace the legacy drop shadow feature. The original feature involved a staggering amount of complexity, related to computing additional shapes and bounding boxes that included a drop shadow for all different types of diagram elements. With JavaFX this became a few lines of code to add a visual effect. This migration-related improvement effort had a large impact on the quality of the overall design, at the small cost of investigating the drawing features of the new framework when necessary.

Conclusions

In hindsight, most of the challenges were challenges of information discovery. To migrate the code from Swing to JavaFX while maintaining high code and design quality standards required a number of key insights which we did not have in advance. Most of these insights could have been obtained through advanced investigation and experimentation. There are, however, two inter-related issues with this line of thinking. One issue is simply that this investigation has a significant cost. The second issue is for many of the situations described in this article, we were not aware of the information need before planning or attempting specific migration tasks. For instance, we never expected that pixels indexed though integers would misalign with device boundary and be blurry. Although technically documented, this information was, at least in our experience, not very easily discoverable. In a perfect world, a lot of the information we found surprising should not have been. The challenge is that there is too much design information to know, and documenters of the software frameworks cannot anticipate what will be relevant or not, or all the ramifications of their design decisions. In certain instances, technology can help surface useful information, for example by discovering insightful sentences in forums like Stack Overflow and injecting them in reference documentation.[9] Given the detailed and context-specific nature of the information needs related to framework usage, it may ultimately be impossible to catalog or anticipate all the design decisions that may impact a migration from one framework to another so as to properly raise awareness about them. The general lesson here is that given the amount of details involved in a major framework migration, there will inevitably be surprises, so a risk management strategy needs to not only seek to minimize the likelihood of surprises, but also to ensure that the project is not fragile to them.

Acknowledgements

The authors are grateful to Sebastian Baltes, Mathieu Nassif, and Christoph Treude for insightful feedback.

References

1. JetUML GitHub Repository. https://github.com/prmr/JetUML.
2. Cay Horstmann. Violet. In Amy Brown and Greg Wilson, eds., The Architecture of Open Source Applications, Chapter 22, Lulu.com, 2012.
3. Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. IEEE Transactions on Software Engineering, 39(5):613-637, May, 2013.
4. Bartolomei T.T., Czarnecki K., Lämmel R., van der Storm T. (2010) Study of an API Migration for Two XML APIs. In: van den Brand M., Gašević D., Gray J. (eds) Software Language Engineering. SLE 2009. Lecture Notes in Computer Science, vol 5969. Springer, Berlin, Heidelberg
5. Thiago Tonelli, Krzysztof and Ralf, "Swing to SWT and back: Patterns for API migration by wrapping," 2010 IEEE International Conference on Software Maintenance, Timisoara, 2010, pp. 1-10.
6. Cuprak, R. (2013, September). Moving from Swing to JavaFX. Lecture presented at JavaOne, San Francisco. Retrieved from https://www.youtube.com/watch?v=ggB2B7YLOjE
7. Steve Zara, How can I migrate Swing applications to JavaFX easily?, https://www.quora.com/How-can-I-migrate-Swing-applications-to-JavaFX-easily?share=1
8. JavaFX API documentation for class Node. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html
9. Christoph Treude and Martin P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In Proceedings of the 38th ACM/IEEE International Conference on Software Engineering, pages 392-403, May, 2016.
10. https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/swing-fx-interoperability.htm
11. Stack Overflow. "JavaFX graphics "blurred" or anti-aliased? (No effects used)", https://stackoverflow.com/questions/9779693/javafx-graphics-blurred-or-anti-aliased-no-effects-used