Lessons Learned While Migrating From Swing to JavaFX

Martin P. Robillard and Kaylee Kutschera, McGill University

We describe a case study of the migration of an interactive diagramming tool written in Java from the Swing graphical user interface framework to JavaFX. The outcome is presented as five lessons about the search for information when migrating software between major frameworks.

EVEN THE MOST risk-adverse project leaders will eventually face the question of if they should migrate to a new framework or not. This question can fill people with dread because the number of technical details one must know to master a large framework is enormous, and the number of things that can go wrong in a major migration is basically infinite.

We recently faced these challenges during the evolution of JetUML, an open source diagram editor for the Unified Modeling Language (UML) that we develop and maintain for teaching and professional use. JetUML was built using the Swing graphical user interface (GUI) framework. Because our resources are minimal, we attempt to curtail the risk of any event that would require heavy development effort. For this reason, we had continually put off using the more modern JavaFX GUI framework. Ultimately, the deciding factor for moving to JavaFX was simply the inevitability of the migration. Adapting to new hardware environments (high-resolution displays, multiple monitors) was becoming necessary, and it was unavoidable that a future development would render the Swing-based application obsolete.

We undertook the complete migration of JetUML from one GUI toolkit to another with the goals of modernizing the software and learning about the major migration challenges. As is typically the case, we had a high level of expertise in the previous framework and only a modicum of expertise in the new one. To a large extent, the challenge was one of discovery. We, thus, construed the project as a case study with two main questions: What information was necessary to complete the migration, and how did we discover it? The experience led to a number of realizations about the learning process involved in moving between major frameworks.

A Brief History of the Project

JetUML is a medium-size, pure Java desktop application used to create and edit diagrams in UML. The project started in January 2015 as an offshoot of the Violet diagram editor. Although Violet was itself spun off as an open source project, Martin P. Robillard launched JetUML to focus exclusively on a minimalistic set of features intended to make diagramming as seamless as possible.
The main usage scenario for JetUML is the interactive creation and modification of diagrams as part of lectures, design reviews, and similar types of presentations. The application relies critically on its GUI framework, which was originally Abstract Window Toolkit/Swing.

Before the migration, JetUML consisted of 9,100 noncomment lines of Java source code (LOCs) and 1,700 lines of comments distributed through 83 source files organized in five packages (the data are for release 1.2). The project was also supported by a suite of 255 JUnit tests comprising 6,300 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 represents 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 they are not shown in the figure, the types Graph, Node, and Edge are extensively subclassed in the application with concrete elements (for example, ClassDiagramGraph, DependencyEdge, and so forth).

**Migration Process**

We organized our project in three phases—preparation, migration, and consolidation—with preparation and consolidation to be completed by the main project developer (Martin P. Robillard) and migration to be completed by Kaylee Kutschera. This developer alternation between phases created a requirement that the design be understandable at the end of each phase.

During the preparation period, Robillard 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, and so forth) from Swing classes to framework-independent equivalent classes
3. replacing the framework-dependent JavaBeans-based persistence

---

**FIGURE 1.** The architecture of JetUML prior to the migration. The diagram also illustrates the output of the tool.
with a framework-independent solution that used the JavaScript Object Notation (JSON).

Because the preparation stage was structured as a refactoring of the existing code, we deferred research about JavaFX to the following phase. As part of the second step, we decided to convert from floating-point geometry to integer geometry in an attempt to simplify the code. The result of the preparation phase was released as version 2.0 Alpha. It brought the code base size to 12,200 LOC in 126 files, with the support of 7,300 LOC 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 third-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 of 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. Attempting the migration was what really brought the question of information needs to a head. JavaFX is a huge framework: In Java Development Kit 8, there are close to 700 classes in javafx\(^8\) packages. With a framework that size, it is impossible to know everything in advance.

Before starting the migration, we searched the web for insights into the process and the likely problems we could run into. This investigation led to a collection of articles, forum posts, videos, and reference documentation. Unfortunately, at that stage, the technical advice proved to be either too specific (focusing on detailed uses cases) or too general (discussing broad issues, such as threading). The real questions we faced were questions of process: How do we find out what we need to know? How do we manage the unfamiliar? What are the traps and risks of venturing into the unknown?

Ultimately, we decided to research, in detail, the information that would allow us to devise an overall migration strategy, and defer more thorough investigation until we faced concrete technical issues. The information we needed concerned adapter components. They enable the use of elements of one framework in another architecture. 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), bottom-up migration (legacy Swing windows containing migrated JavaFX widgets), or any combination of the two that isolates changes and limits risk.\(^3\) In the end, we selected a top-down migration strategy so that we could move the more stable (“architectural”) part of the design first and defer migrating the drawing code, which required more uncertainty and experimentation, until the JavaFX-supported window structure was in place. The result of the migration phase was released as version 2.0, which was almost identical to 2.0 Alpha 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 stage was released as version 2.1, with 12,300 LOC in 142 files, with additional improvements committed to versions 2.2 and 2.3. The complete code base of all the releases of JetUML can be obtained from its GitHub repository.\(^1\) In the end, we successfully completed the migration at the cost of approximately three person-months, with additional reworking after the initial JavaFX release.

**Lessons Learned**
The migration process led to five major realizations and corresponding implications about looking for technical information to support a framework change. Although the insights are derived from our experience adapting to JavaFX, they are not tied specifically to details of that GUI framework, so we expect that they will generalize to situations where frameworks exhibit similar characteristics. For each lesson, we present a summary of technical details, a discussion of the information-seeking context and what we grew to understand, and how we would have leveraged each lesson had we known it in advance.

**Adapting to Detail**
Our first lesson concerns the information we expected to locate easily but that turned out to be largely insufficient. The cornerstone of our migration plan was to use adapter components to support an incremental process. Adapters are a classic means of moving from one framework to another,\(^4\)\(^5\) so we knew what to look for. Adapter components are available in both Swing and JavaFX.\(^6\) Swing includes a JFXPanel class to embed JavaFX components into a Swing application, and JavaFX includes a SwingNode class to embed Swing elements into
a JavaFX application. Both classes provide corresponding application programming interface (API) documentation. Additionally, the official Java documentation site includes an article on JavaFX-Swing interoperability. With this information in hand, it appeared that the use of adapter components was a straightforward task. Unfortunately, when we attempted to use them, numerous practical issues surfaced that required additional knowledge and identified the glaring limitations of the official documentation, which showed only how to use the adapters in basic scenarios and provided insights on just one additional concern, concurrency.

**Performance**

Top-down migration requires the use of the SwingNode adapter class, which can hold Swing content in JavaFX windows. This strategy leads to performance issues because SwingNode instances are not meant to hold heavyweight components. In contrast, bottom-up migration using the JFXPanel does not have these problems. Hybrid approaches, such as interposing a JavaFX component between Swing components, can result in major performance problems, such as large delays when first loading a window.

**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 elements from both architectures. To address sizing problems, one recommendation was to hardcode fixed preferred sizes until they could be properly computed by the framework, a process which would have added development overhead.

**Dependency Cycles**

During migration, cyclic dependencies between classes may be needed if it is necessary to access a parent component. These dependencies can be removed once the 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, a high-level windowing element (the tabbed pane) was migrated to JavaFX before the drawing area it contained, which remained a Swing component. The Swing drawing area needed a reference to a diagram’s tab to update the tab’s title. The JavaFX parent component was not accessible in the Swing child because there was no way to access the SwingNode instance that the child component was wrapped in.

In the end, the problem we faced was that highly relevant information was easy to find but inevitably shallow because the use of adapters is very context sensitive. The consequence was that we had to expend much effort hunting for scarce experience reports and experimenting to determine when to use the components. The lesson was that existing documentation on how to use a pivotal component in a small, synthetic scenario is bound to be incomplete.

In hindsight, it would have been better to make this realization earlier and invest time in prototyping and experimenting with the components in context. Ideally, though, documentation for adapters would benefit from a list of the main implications for using them.

**False Friends**

In linguistics, the expression false friend refers to a term in one language that is deceptively similar to a term in another language but has a different meaning. For example, the term bil- lion commonly denotes $10^9$ in English but $10^{12}$ in French and German. As we realized, the same phenomenon occurs in the context of translation between frameworks.

A major concern for a diagram editor is drawing shapes. Swing supports this functionality through a Shape class hierarchy, with subclasses such as Arc2D, Ellipse2D, and so forth. 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 noticed that it defined a near-equivalent API with a class Shape that had comparable subclasses with the same name (except the 2D suffix). This realization initially gave us the impression that migrating the drawing
code would be a trivial exercise in mechanical translation and that the exact correspondence of names even made the need for advanced API migration mining tools superfluous. 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, 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 former 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 drawing primitives).

The problem was that, by noticing an API type hierarchy with nearly identical names, we made the false assumption that we did not need to invest additional effort in an information search for the corresponding part of the migration. The consequence was that we needed to change the original design after the migration was partially completed. The lesson we learned was simple: Beware of false friends. A basic procedure that would have avoided the deception would have been to inspect references to (Swing) Shape classes to ensure that the same services were available in JavaFX. As it turned out, they were not.

Feature Gap

A large influence on the design of JetUML’s Swing version was what originally seemed like a modest feature: the ability to draw shapes directly on user interface components. In Swing, most user interface components are subclasses of JComponent, which defines a paintComponent(Graphics) method. Subclasses can then override this method to draw shapes directly on the component. We leveraged this simple feature to orchestrate the visual rendition of diagrams. Figure 2 summarizes the key aspects of this design. The main point of the figure is the simplicity of the design: When it determines that a refresh operation is necessary, the framework simply calls paintComponent(Graphics) which delegates the drawing of the graph. The code in graph.draw(...) then calls drawing primitives directly on the Graphics object.

The biggest hurdle we faced in our migration project was the realization that it was not possible to draw directly on user interface components in JavaFX and that a major redesign of the drawing mechanism was, thus, necessary. Unfortunately, we found no information on how to handle this situation. The ability to draw shapes obviously exists in JavaFX. However, because the mechanism was redesigned, there was no straightforward migration path to be followed (for instance, replacing a class with another). We conjecture that because the number of ways to adapt to the feature gap was open-ended and context-sensitive it was not feasible to document specific migration paths. In any case, the circumstances created two major sources of information necessity: 1) discovering viable migration paths and choosing one, and 2) locating all the technical knowledge required to implement the selected approach.

In JavaFX, the creation of drawings can be done in two ways that operate at two levels of abstraction. The high-level mechanism involves creating instances of class Shape and adding them to a Pane to constitute a collection of drawable objects. The low-level mechanism involves creating a drawing directly on a Canvas instance by using drawing primitives on the canvas’s graphics context and

---

**FIGURE 2.** The design of the diagram-drawing concern.
embedding the canvas in a pane. Each of these mechanisms forms a somewhat polarized version of Swing’s original drawing approach, which combined elements of both. After extensive experimentation to fulfill the first information need, we determined that using a scene to represent a drawing was too inefficient, and we opted for the embedded-canvas solution. Unfortunately, trying to implement the canvas-based migration of the drawing feature turned out to be frustrating because technical issues caused by the difference in drawing mechanisms kept turning up and had to be researched extensively.

As an illustration, the worst issue was that in the Swing-based version of JetUML the drawing area was resizable. To match the original feature, we tried to make the canvas resizable. Technically, this only required overriding a few methods. Despite a number of resources documenting straightforward solutions, none were applicable. In experimenting with a resizable canvas, we ran into numerous layouting and sizing issues when trying to integrate it with a scroll pane (a component that allows scrolling areas larger than the window size). Ultimately, one of the nicest features of JetUML, a diagram space that seamlessly adapts to the window size, could no longer be reasonably supported. We stopped trying and ended up investing a considerable amount of redesign effort in rethinking how the tool would work with a fixed diagram size.

It can be expected that not all features of a framework will be ported, at least initially. This is not an issue when there is a straightforward migration path or alternative. For example, JavaFX does not provide any functionality to save an image to a file. However, there is a well-documented workaround, namely, to convert a JavaFX image to a Swing image using a provided utility method and then using the legacy API call to save the image. In the case of the drawing feature, the problem was not only the lack of a formerly supported function in JavaFX but the absence of a canonical migration path for reimplementing the utility. The consequence was not only a little as possible about the look and operation of the application. For this reason, our search for information focused on discovering migration paths that were literally at the code statement level. However, through later code inspection, we realized that this strategy had been overly aggressive. We had reimplemented various features that had been custom-built into the Swing version because no other option was available.

Feature Blindness

Another issue we faced during our search was that we explicitly avoided looking for information when we should have. One of our strict risk-management strategies during the migration phase was to change as much as possible about the look and operation of the application. For this reason, our search for information focused on discovering migration paths that were literally at the code statement level. However, through later code inspection, we realized that this strategy had been overly aggressive. We had reimplemented various features that had been custom-built into the Swing version because no other option was available.

As an example, one of the custom GUI components for the Swing version was a toolbar. A toolbar is a container for a button group that allows activating or selecting “tools,” such as to create a node in a diagram. In modern GUI applications, toolbar components can reorganize or hide some tools when the default layout does not fit in the display. Swing did not offer this feature, which was why we have created a custom implementation. As part of our migration process, we recreated the custom version in JavaFX, only to realize that all the desired functionality was available from JavaFX’s `ToolBar` component. Another example was the implementation of a “drop shadow.” Swing did not have explicit support for adding drop shadows to image elements, and a lot of code was necessary to compute additional shapes and bounding boxes that included a

Existing documentation on how to use a pivotal component in a small, synthetic scenario is bound to be incomplete.
FEATURE: MIGRATING BETWEEN GUI FRAMEWORKS

In summary, the problem we faced was that, by insisting on a literal migration, we overlooked many opportunities for simplifying the code. The consequence was twofold: First, we spent a great deal of effort migrating complex code that could be simplified, and second, we eventually spent additional effort implementing the improvement as part of later releases. To be sure, identifying features that we can leverage as part of the migration is not free. One problem is that, although knowledge of what we need a posteriori is obvious, before the migration it is much harder to know what new features can reasonably be used in the migrated version. Ultimately, the lesson we drew was not to be too strict about the migration. Although this is easy to say after the fact, we hypothesize that, with a minimum of a priori requirements analysis, it would have been possible to identify the two most obvious features, namely, the tool bar component and drop shadow effect.

Hidden Information

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 overaggressive aliasing and set the issue aside for the JavaFX-based version 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. Our examination 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.”9 This means that to have a point exactly map to a pixel and render sharply it needs to have the coordinates translated by (+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 provided solution 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 no mention is made of blurriness. A Stack Overflow post10 turned out to be instrumental in helping us assemble the solution to this puzzle from disparate pieces.

In this case, the problem was that information about an important technical concern with pervasive implications was not located where we would have expected it (namely, an overview page, tutorial, or migration guide). Rather, it was placed in API documentation, which we normally access as a reference when trying to answer detailed technical questions. The consequence was that instead of integrating this key piece of information into the migration process, it had to be used post hoc during recovery efforts. Specifically, we solved the problem in a later release by directing all shape drawing requests through methods that simply shifted the original integer coordinates by 0.5 in each dimension. The lesson was to deeply investigate, at least, the specific implications of design decisions that are related to

ABOUT THE AUTHORS

MARTIN P. ROBILLARD is a professor of computer science at McGill University, Montréal, Quebec, Canada. Robillard received a Ph.D. and an M.Sc. in computer science from the University of British Columbia, and a B.Eng. from École Polytechnique de Montréal. His research interests include problems related to software evolution, architecture and design, and software reuse. Contact him at martin@cs.mcgill.ca.

KAYLEE KUTSCHERA is a software engineer leading iOS development for crisis response on Google Maps. Her research interests include software design, maintenance, and evolution. Kutschera received a B.Sc. in mathematics and computer science at McGill University, Montréal, where she worked under the supervision of Martin P. Robillard. Contact her at kaylee.kutschera@mail.mcgill.ca.
the migration. 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 probe the ramifications of this design decision.

Most of the challenges we faced were issues 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 that we did not have in advance. Most of these insights could have been obtained through additional investigation and experimentation. However, before undertaking the migration, the task of discovering all relevant information seemed unsurmountable: There was too much material to peruse, and our needs were too vague.

Fulfilling developers’ information needs is an active research area in software engineering. The paper “Patterns of Knowledge in API Reference Documentation” provides a structured review of the foundational work in this area. In certain cases, technology may help to surface useful information, for example, by discovering insightful sentences in forums like Stack Overflow or repackaging technical information into a more convenient format. However, solutions that rely on existing documentation are dependent on the quality of the documentation and do not address situations where information needs must be fulfilled through experimentation.

The lessons described in this article enable a more general solution: a structured approach to identifying information needs related to a migration. Each of the lessons provides a general context where migration information was challenging to obtain, examples of the potential consequences of insufficient research in that context, and insights on how to better structure the information search in such a context.

Acknowledgments
We are grateful to Sebastian Baltes, Mathieu Nassif, Christoph Treude, and the anonymous reviewers for their insightful feedback.

References
1. M. P. Robillard et al., “JetUML,” GitHub, 2015. [Online]. Available: https://github.com/prmr/JetUML
2. C. Horstmann, “Violet,” in The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks, A. Brown and G. Wilson, Eds. Morrisville, NC: Lulu Press, 2012.
3. R. Cuprak, “Moving from Swing to JavaFX,” YouTube, June 8, 2015. [Online]. Available: https://www.youtube.com/watch?v=ggB2B7YLOjE
4. T. T. Bartolomei, K. Czarnecki, R. Låmmel, and T. van der Storm, “Study of an API migration for two XML APIs,” in Proc. 2nd Int. Conf. Software Language Engineering, Oct. 2009, pp. 42–61. doi: 10.1007/978-3-642-12107-4_5.
5. T. T. Bartolomei, K. Czarnecki, and R. Låmmel, “Swing to SWT and back: Patterns for API migration by wrapping,” in Proc. 2010 IEEE Int. Conf. Software Maintenance. doi: 10.1109/ICSM.2010.5610429.
6. Oracle, “JavaFX: Interoperability,” 2012. [Online]. Available: https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/swing-fx-interoperability.htm
7. M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, “Automated API property inference techniques,” IEEE Trans. Softw. Eng., vol. 39, no. 5, pp. 613–637, May 2013. doi: 10.1109/TSE.2012.63.
8. M. Nutt, “JavaFX: Resize canvas when screen is resized,” Stack Overflow, Mar. 3, 2014. [Online]. Available: https://stackoverflow.com/q/23449932
9. Oracle, “Class javafx.scene.Node,” 2008. [Online]. Available: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html
10. R. Leitão, “JavaFX graphics ‘blurred’ or anti-aliasied? (No effects used),” Stack Overflow, Mar. 19, 2012. [Online]. Available: https://stackoverflow.com/q/9779693
11. W. Maalej and M. P. Robillard, “Patterns of knowledge in API reference documentation,” IEEE Trans. Softw. Eng., vol. 39, no. 9, pp. 1264–1282, Sept. 2013.
12. C. Treude and M. P. Robillard, “Augmenting API documentation with insights from Stack Overflow,” in Proc 38th Int. Conf. Software Engineering, 2016, pp. 392–403.
13. S. Henß, M. Monperrus, and M. Mezini, “Semi-automatically extracting FAQs to improve accessibility of software development knowledge,” in Proc. 34th Int. Conf. Software Engineering, 2012, pp. 793–803.