JUICE: AN SVG RENDERING PEER FOR JAVA SWING

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science

by

Ignatius Yuwono

May 2006

ABSTRACT

SVG—a W3C XML standard—is a relatively new language for describing low-level vector drawings. Due to its cross-platform capabilities and support for events, SVG may potentially be used in interactive GUIs/graphical front-ends. However, a complete and full-featured widget set for SVG does not exist at the time of this writing. I have researched and implemented a framework which retargets a complete and mature raster-based widget library—the JFC Swing GUI library—into a vector-based display substrate: SVG. My framework provides SVG with a full-featured widget set, as well as augmenting Swing's platform coverage. Furthermore, by using bytecode instrumentation techniques, my Swing to SVG bridging framework is transparent to the developers—allowing them to implement their user interfaces in pure Swing.

TABLE OF CONTENTS

TABLE OF FIGURES

I INTRODUCTION

I.1 The Problem

SVG[1] is a relatively new standard XML "picture format" and may potentially be used in interactive graphical front-ends. SVG's markup allows the user to describe vector drawings using primitives such as paths, rectangles, and groups. In addition to that, SVG provides event handling mechanisms that conform to the DOM level 2[10] and level 3[11] Event standards.

Due to its cross-platform capabilities and support for events, SVG may potentially be used in interactive graphical front-ends. However, the problem is that there exists no complete and full-featured widget set for SVG at the time of this writing.

I.2 The Proposed Solution

The solution is to retarget rendering and event information from an existing full-featured and complete widget library into SVG. I also wanted the solution to be in Java because of its extensibility and ubiquity. In order to perform retargeting, the widget set must be written in Java, open-source, extensible, and have a clear separation between the widget set and the underlying rendering and event system. If possible, the widget set code should be left unadultered.

Upon researching several available Java widget sets, I chose the JFC Swing[3] GUI Framework.

Why the Swing framework? Sun's Swing framework offers a complete desktop GUI solution mainly geared towards the desktop developer. Here are some Swing features:

- Cross-platform for desktop operating systems
- Written in Java, making it accessible and extensible by developers
- Can be run with just the minimum requirement for Java: using the Sun JVM (Java Virtual Machine) without any special extensions
- Bundled with Sun's standard JDK for development purposes
- The most prevalent Java GUI toolkit

There are other GUI frameworks such as SWT[17], that is also partially written in Java, but fail to have clear separation between rendering code and native operating system code. Retargeting SWT into SVG means that we would have to re-implement the whole widget set, which is what I am trying to avoid.

I therefore designed and implemented the JUICE (Java User Interface Component Extension) framework as a solution to the non-existent SVG widget set problem. JUICE allows Java Swing widgets such as buttons, textfields, etc. to be used in SVG, thus offering a mature existing complete widget set to be used for GUI development in SVG.

I.2.1 Background

SVG (Scalable Vector Graphics) is a W3C standard XML markup language that is used to describe vector illustrations. Vector-based formats differ from raster-based formats (e.g., GIF, PNG, TIFF) in that vector representation is resolution independent. SVG can be rendered equally well on the web, desktop, and mobile devices.

JFC (Java foundation Classes) / Swing is the most prevalent Java GUI Toolkit today. Swing is extensible, which allows third party developers to develop Swing components and extensions. There exists a plethora of third party components, extensions, and applications for Swing.

I.2.2 Prior Work

Prior work with SVG has attempted to address the lack-of-widget technology gap by creating SVG-native widgets from scratch. Although these efforts may eventually prove fruitful, none to date has been able to provide a complete widget library that has localization, font, and look-and-feel support.

SVG developers customarily take the ad-hoc approach to widget construction: designing and implementing specialized widgets that are restricted to one particular application; Lewis, et al.[9] constructed widgets for a GUI specifically designed for genome data visualization.

Lindsey[8] created a sample buttons-only SVG widget, however this set requires low-level SVG customization and low-level SVG event handling callbacks.

Chatty, et al.[4] have combined a vector-based widget toolkit with a user interface designer to create a desktop user interface that is vector based. However, the designer still has to define the primitive interactions. For example, to create a pushbutton, the developers must define all the possible button state images such as the button idle SVG image, the button pressed SVG image, button over SVG image, etc. The developer then also has to explicitly link all of the states. Chatty's framework is feature-incomplete compared to Java Swing; for example, there is no layout management, there is no simple way to change the look-and-feel for all the widgets, no support for accessibility, and no easy extensibility.

Fettes, et al.[6] succeeded in building a Java-based SVG User Interface framework created from scratch, however, this is a recent effort and the features are also incomplete

compared to the Swing framework—which has been improved since its public introduction in 1998. Not only that, but the users now must learn a new framework in order to develop SVG applications.

I.2.3 Motivation

Publishing to SVG has several benefits:

- SVG is an open standard
- SVG is XML: declarative markup
- SVG is vector-based, making it infinitely zoomable
- There exists SVG viewers for desktop and mobile platforms.

Thus, if we have a widget set for SVG we can create user interfaces that are zoomable, and transformable through the use of XSLT. Transformability of user interfaces is important for delivery to multiple platforms, especially mobile platforms which usually have limited screen resolution. The transformations can be executed on the client-side as demonstrated by Marriot, et al.[5].

Swing is an excellent candidate for bridging because of its rich features, user-friendliness, wide acceptance, and extensibility. My framework, JUICE, allows Swing developers to code in Swing but display in SVG, thus obtaining the benefits of both Swing and SVG.

I.3 Challenges

JUICE is an experiment in marrying two different rendering paradigms: raster(bitmap) based rendering and vector based rendering. The challenges listed below are mostly based on consolidating the differences between the two technologies:

- Swing is designed to be run on a raster-based display, while SVG is vector-based.
- Integrating two different repaint/painting methodologies:

Raster	**(SVG)Vector**
Concerned with screen-pixel management	Concerned with vector object management
One-to-one relation between pixels on the screen and the drawing data	A vector object does not directly correspond to pixels on the screen. These pixels are determined by the current transformation matrix which can include scaling, shearing, etc.
‘Dirty rectangle’ detection. A ‘dirty rectangle’ is an area of the screen that needs to be updated.	‘Dirty object’ detection. A ‘dirty object’ is an object which properties have changed. This also includes

	removal and addition of new objects
Screen areas that have been repainted over are lost.	Objects that are hidden by other objects are not lost. They are still in the vector tree
Non-declarative	Declarative

- Raster event model is one-to-one due to the direct relationship between screen pixels and drawing data. SVG (Vector) event model is not one-to-one because of the indirect relationship between pixels and vector objects. This is a problem because Swing assumes a pixel-based/raster display target.
- All SVG mouse and keyboard events must be ferried and translated into the Swing layer.
- Since keyboard events depend on focus events, focus events must be ferried as well.
- The internal workings of Swing are undocumented. The programming documentation that describes painting, repainting, and events does not indicate how the flow is actually implemented inside Swing.
- We need to understand how our chosen SVG toolkit and viewer, Apache's Batik[2] SVG toolkit works, since it is mostly designed for viewing static SVG Documents.
- Abstracting all this complexity from the user. JUICE must be transparent to the user. Programmers implements their GUI using Swing—complete with `ActionListeners` and other callback listeners—and have JUICE handle the SVG redirection.

II INTERACTING COMPONENTS

II.1 Swing

Swing is a multi-platform GUI (Graphical User Interface) toolkit for java desktop application developers. Swing is part of the JFC (Java Foundation Classes) which is a desktop-application oriented API. Besides Swing, JFC includes APIs for 2D graphics rendering, accessibility, and internationalization. Swing is an extension of the AWT (Abstract Window Toolkit) which was the first Java GUI Toolkit.

Swing is implemented entirely in Java on top of Java 2D and AWT. Java 2D provides Swing the ability to perform raster drawing operations in an object-oriented manner. AWT provides the hooks into the native operating system, thus having support for drag-and-drop, native windows, frames, dialogs, tooltips, and events.

Swing has a rich set of features such as pluggable look-and-feel, localization, and accessibility support. All of these features allows UI developers to make their User Interface code portable to other operating systems, without worrying about multiple chromes, multiple languages, and accessibility.

Since Swing components are implemented in Java, they are extensible and flexible. The developer can create a custom component based on an already existing base Swing component, customizing its looks and behavior.

Swing is currently supported in Windows, Linux/Unix, Solaris, and Macintosh operating systems.

II.1.1 Swing Design

II.1.1.1 General Design

Sun's Swing components are platform-independent Java components that are intricately tied to the native display and events substrate. This tie is necessary to have the drawing instructions be translated onto the screen. Most of the native coupling is provided by the underlying AWT toolkit upon which Swing is built. See **Figure 1** for a pictorial description.

The coupling of Swing with the native operating system is done by using 'peer' classes. These peer classes are platform-specific and know how to interact with the underlying platform. Swing communicates with relevant peer classes through the use of Java's JNI(Java Native Interface) mechanism.

Native peer implementations are provided by the AWT toolkit which has a peer for every component. Swing only extends the necessary legacy AWT containers such as `Window`, `Frame`, and `Dialog`. The rest are all peerless components.

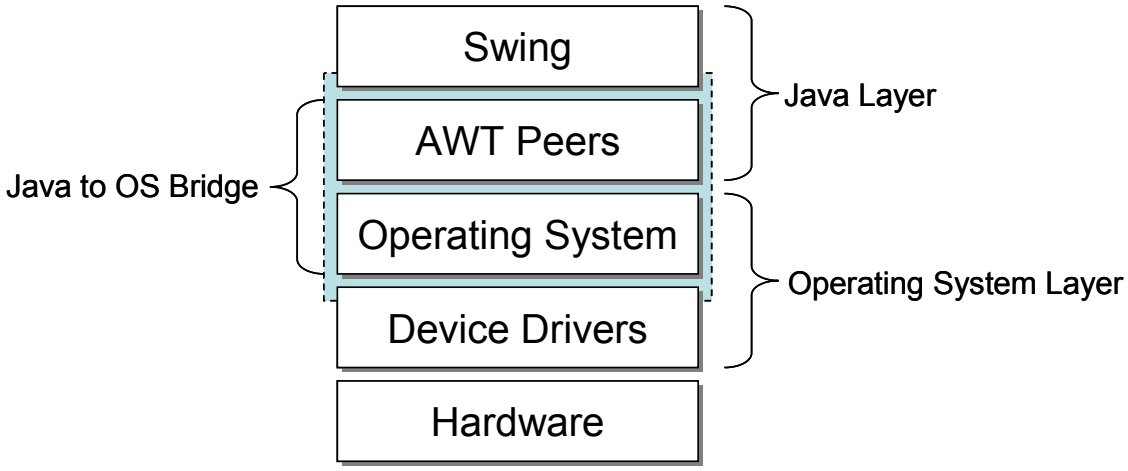

Figure 1: Swing and the OS

II.1.1.1.1 The Swing Rendering Model

II.1.1.1.1.1 Heavyweight Components

Heavyweight components can be defined as Swing components that have corresponding native peers. Heavyweight components usually are top-level containers that touches the user desktop directly. Examples of heavyweights include but are not limited to: `JFrame`, `JWindow`, and `JDialog` (see **Figure 2** for an example) . Heavyweights rely on the native operating system peer to provide them with positioning, z-ordering, clipping, and damage rectangle detection.

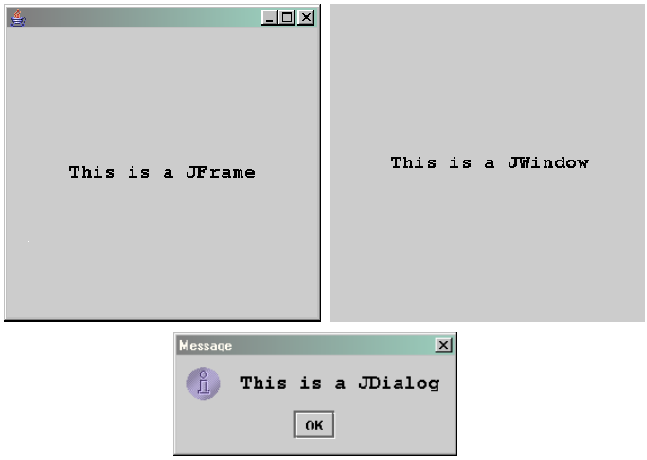

Figure 2: JFrame, JWindow, and JDialog

They also share a common trait in that they extend directly from the corresponding AWT class, instead of extending JComponent or extending a descendant of `JComponent`. For example, `JFrame` extends AWT's `Frame` class. Remember that AWT elements each have a corresponding peer. **Figure 3: Swing/AWT Inheritance** shows the general hierarchy of Swing and AWT components.

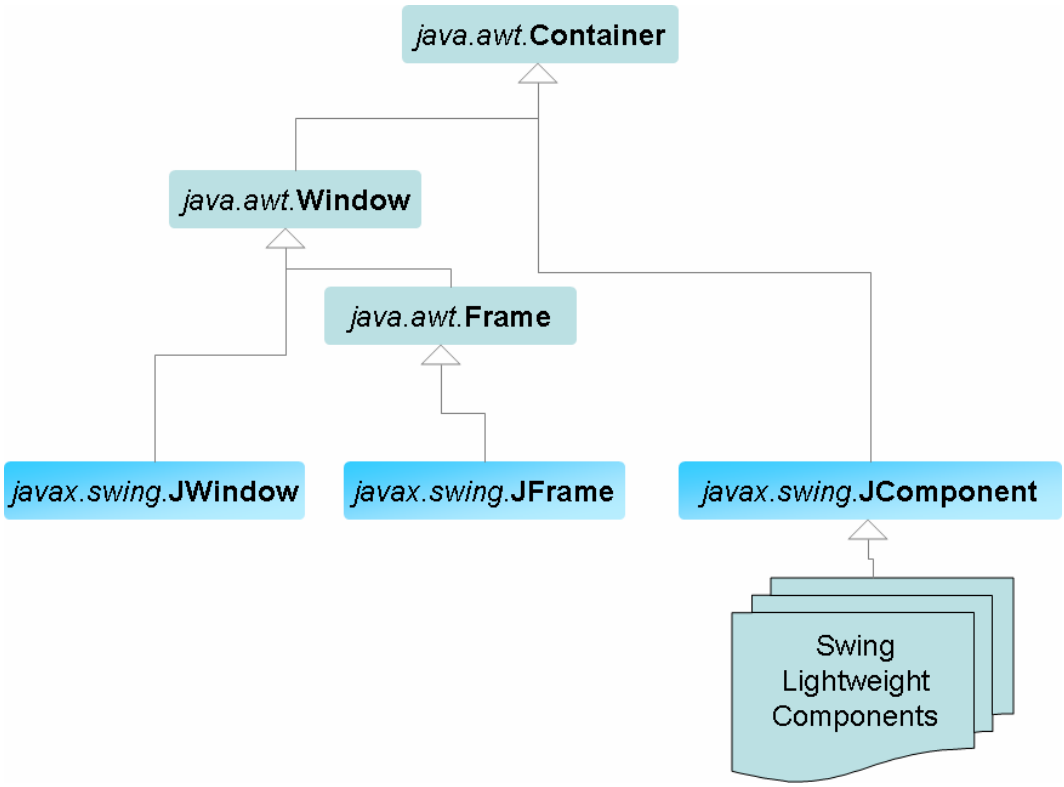

Figure 3: Swing/AWT Inheritance

Heavyweight components obtain their graphics context directly from the operating system through their native peers. This allows them to exist directly within the operating system's graphical user environment. In AWT, each user interface element, be it a button or a text field, has a corresponding peer. This allows buttons direct interaction with the operating system. This is not the case in Swing.

II.1.1.1.1.2 Lightweight Components

Lightweight components can be defined as Swing components that do not have corresponding native peers. These peerless components usually are user interface elements that can be put inside a top level container. Examples of Lightweights include but are not limited to: `JButton`, `JTextField`, and `JTree`. Please look at **Figure 4: JTextField and JButton** for an example.

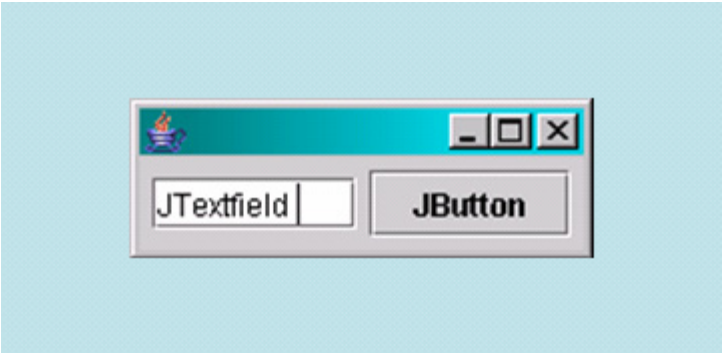

Figure 4: JTextField and JButton

Unlike heavyweight components, lightweight components extend `JComponent` directly. `JComponent` is Swing's base component for non-top-level components. `JComponent` provides its subclasses with pluggable look and feel, keyboard event handling, tool tip handling, accessibility method stubs, bean properties, bordering, double buffering, and graphics handling--all in Java.

Swing is based on an MVC-like design (**Figure 5: Swing component model and UI delegates**). This design has one model, and a corresponding UI delegate that does the actual drawing. Every Swing component has an associated UI Delegate. Which UI delegate to use is determined at runtime by the UI manager based on either the system-default or user-specified look-and-feel. The UI Delegate is written entirely in Java.

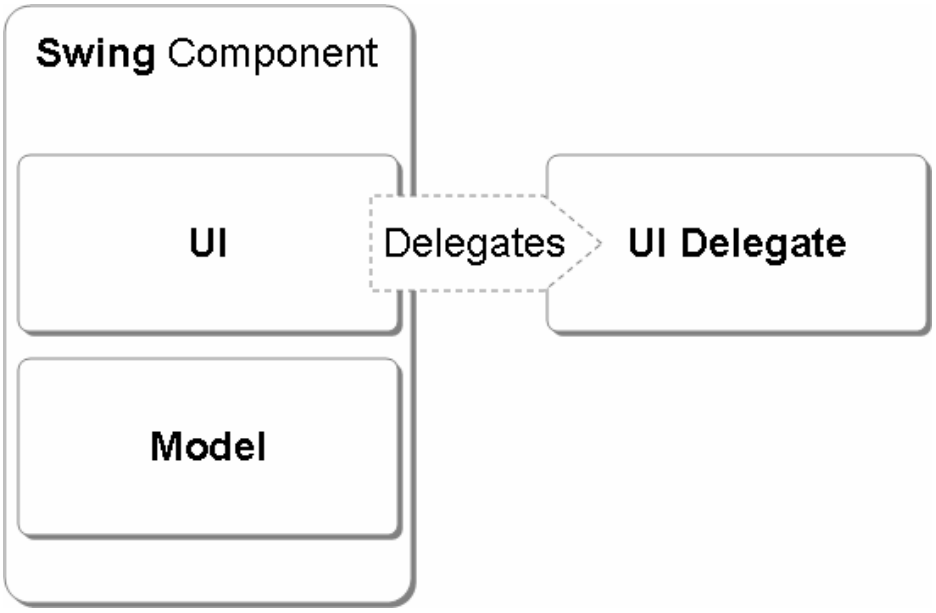

Figure 5: Swing component model and UI delegates

Once a Swing component is rendered onto the screen, all subsequent UI updates for that component goes through one update thread per application. To make the thread accessible to users, Swing provides a way to queue update instructions to the single Swing thread.

Drawing information in the lightweight components is displayed through the screen by borrowing the graphics context of a heavyweight ancestor (**Figure 6: Graphics context passing**).

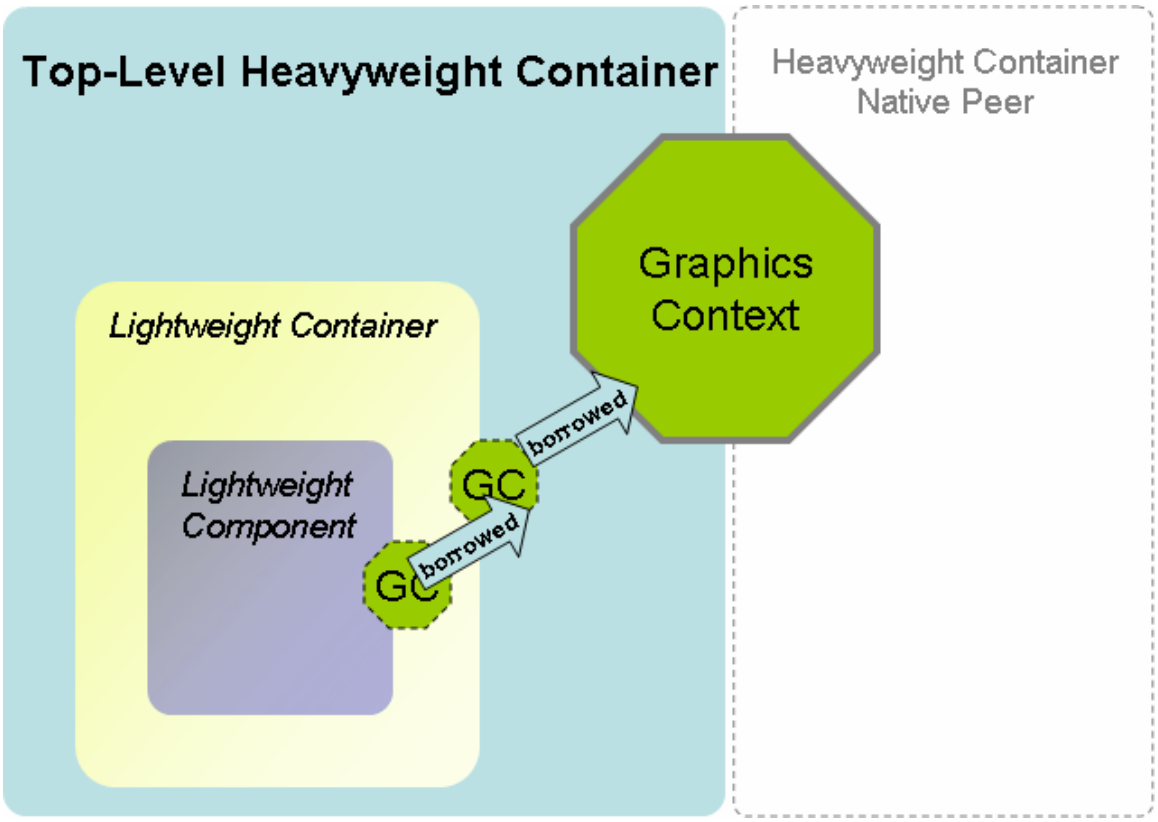

Figure 6: Graphics context passing

II.1.1.1.1.3 Graphics Context

Graphics contexts allows applications to draw component representations straight onto a rendering device. The rendering device can be a screen device or a printing device. Conceptually, any device will work as long as there is an implementation of the graphics context for that device. Most graphics context are implemented natively by the

underlying operating system. To make such a graphics context accessible by Java objects, the graphics context is wrapped into a Java object through the use of JNI (Java Native Interface).

To make all graphics context accessible in the same fashion, all graphics context must extend the abstract class `Graphics`. The abstract `Graphics` class serves as the visible interface to implementors. This abstraction is necessary for Swing to be portable to different devices. The `Graphics` class contains methods that allows its users to give primitive drawing instructions such as draw a line, a rectangle, image, to the underlying device.

Heavyweight components have their own graphics context provided by their native peer. Lightweight components obtain the graphics context from their top-level heavyweight ancestor. The Swing component model uses a variant of the Composite design pattern, in that every `JComponent` can be put inside of a `Container`, but not every `Container` is a `JComponent`. The containers that are not a `JComponent` instances are usually top-level heavyweight containers. The lightweight descendants then request the graphics context from the heavyweights. If the immediate parent is not a heavyweight, then that parent will ask its containing parent until a heavyweight ancestor is found. But sharing the context means that multiple lightweight components will be competing to update the context; this will be a performance bottleneck when multiple repaints have to be done.

II.1.1.1.1.4 Painting

There are two types of painting calls categorized by the initiator: system-initiated painting and application-initiated painting.

System-initiated painting is done by the AWT container and can be triggered by either the underlying operating system, or by a lightweight. These events usually happens the first time a component is being shown, resized, hidden, or destroyed.

Application-initiated painting is triggered by the user by explicitly calling the `repaint` method.

Swing also has extra painting features such as double buffering and transparency support. Double buffering support is provided at the component level; however, the double buffering properties of a container is usually propagated down. Swing uses the standard double-buffering approach by drawing to a single offline buffer before sending any information to the screen.

Swing introduces three new painting methods on top of the AWT `paint` and `update` methods. These methods are: `paintComponent`, `paintBorder`, and `paintChildren`.

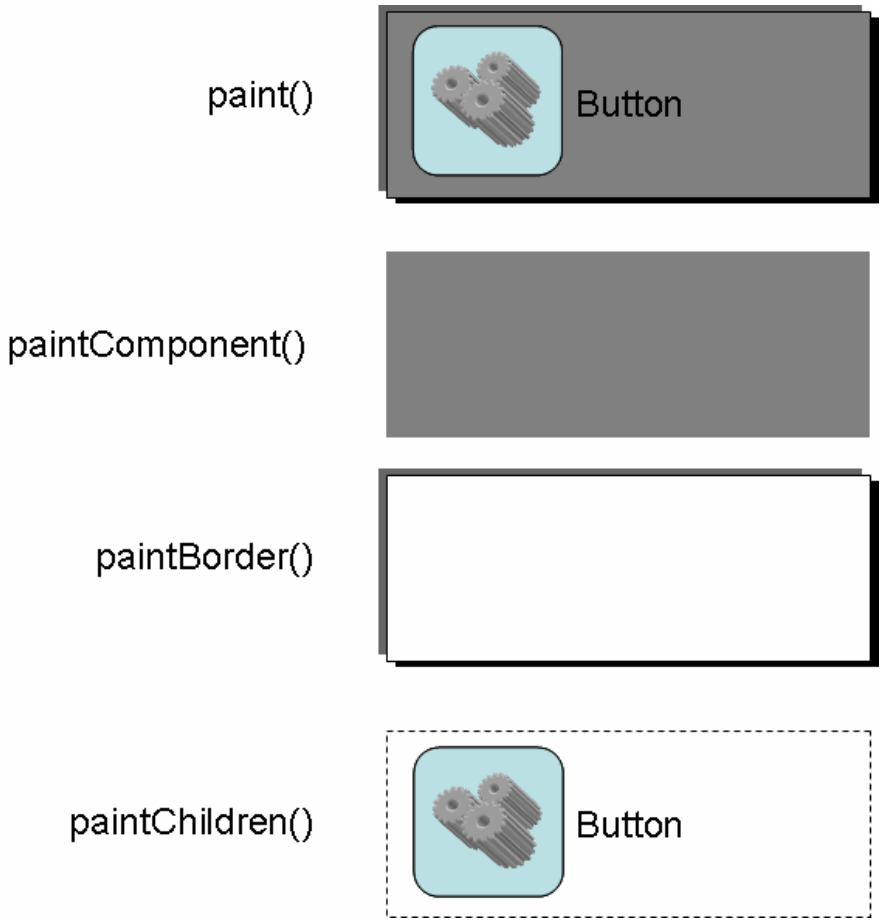

Figure 7: Different paint methods

The first method, `paintComponent`, is the main method that invokes the component's UI delegate. It can also contain custom drawing information provided by user extension classes. The second method, `paintBorder`, is responsible for calling the appropriate `BorderUI` class that does the actual rendering. The method `paintChildren` enumerates the visible and partially visible children, and calls their `paint` method. Please look at **Figure 7: Different paint methods** for a visual overview.

II.1.1.1.1.5 UI Delegates

For Swing components, `paint` calls `paintComponent`, which in turns calls the UI delegate's update method. Remember that the UI delegate is the one that is responsible for the actual drawing of a component. The UI delegate then determines if the component needs to be filled, and continues to invoke the UI delegate's `paint` method.

The actual rendering instructions are given inside the UI delegate's `paint` method. This is because Swing was designed to have a pluggable look and feel without changing the way the developer interacts with Swing.

II.1.1.1.2 The Swing Event Model

Swing's event model is based on AWT's event model (see **Figure 8: The AWT event queue**). The AWT low level event model is simple. There is one central queue for events, and events are being dispatched through a central dispatcher as well. Applications then can register listeners to listen to a particular event.

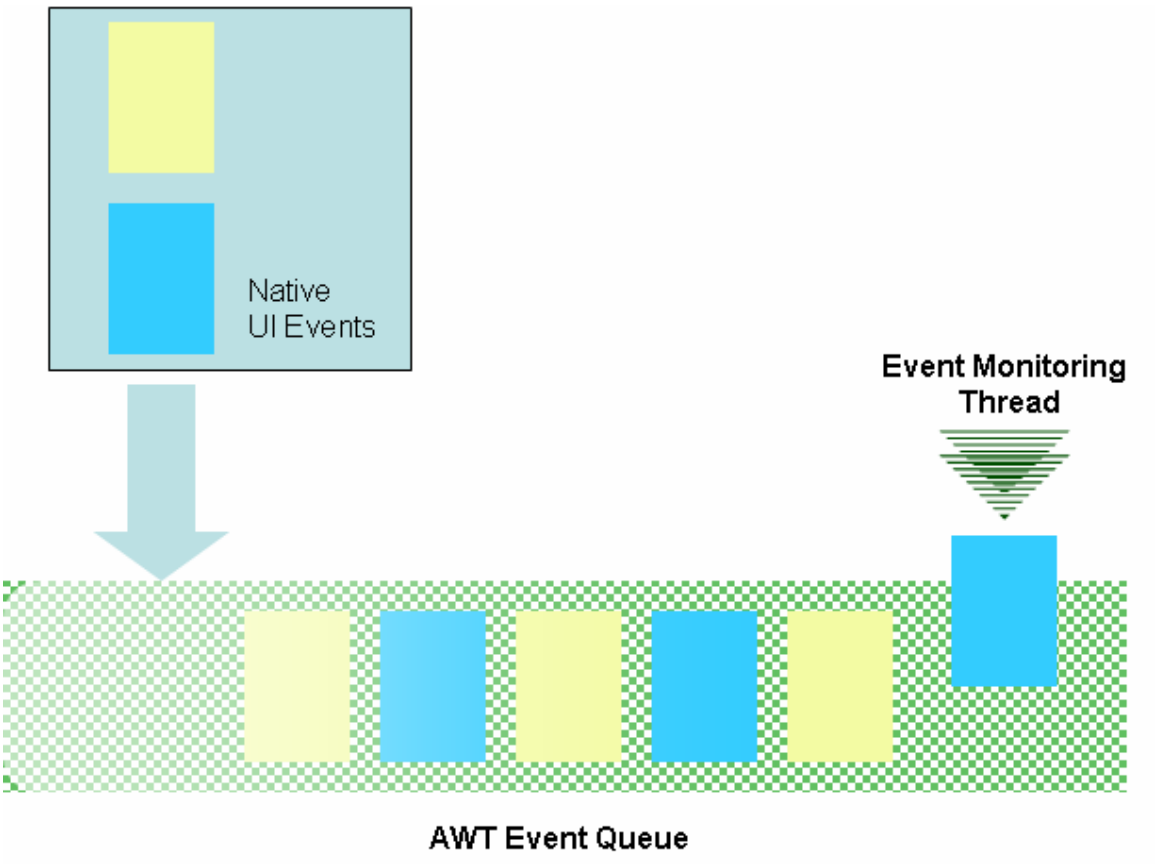

Figure 8: The AWT event queue

There are two general types of events: low-level events, and high-level events. Low-level events are atomic events such as mouse-down, mouse-up, key-up, and key down. Low-level events are very basic in nature, and are taken from the operating system as-is. High-level events are a composition of low-level events. A button's `ActionEvent`, for

example involves a mouse-down followed by a mouse-up within the button bounds; if the mouse-up is registered outside the boundary, the `ActionEvent` is not fired.

II.2 Apache Batik SVG Framework

The Batik SVG framework is the first SVG framework that offers a complete set of features to support static SVG generation, processing, and rendering. It is developed under the Apache Software Foundation license, and conforms tightly to the SVG standard.

II.2.1 Batik Design

The main Batik design consists of two parts, the Batik core-area and the Batik low-level area. The Batik core area contains classes that can be used by the developer to process, generate, and display SVG. The Batik low-level area contains classes that provide the necessary foundation for the core area classes. They perform the actual grunt work for the core classes.

II.2.1.1 Core Area Classes

There are five main Batik core component subsystems. They are: The SVG Generator, The Batik transcoder, The Batik SVG DOM API, The Batik `JSVGCanvas`, and the Batik bridge.

The main component of the SVG Generator is `SVGGraphics2D`. SVG Graphics 2D allows Java applications to easily convert raster drawing instructions to vector drawing instructions. `SVGGraphics2D` is a direct replacement for any Java class that uses the Java 2D API for rendering. All of this is possible because of the abstract nature of the Graphics class that allows for direct substitution. SVGGraphics2D takes in Java 2D drawing instructions and converts them into the appropriate SVG DOM representation.

The Batik transcoder subsystem provides translation from an SVG Document or DOM, to any supported raster image format. This provides a way for saving an SVG document as an image, and also for printing.

The SVG DOM API is an implementation of the SVG W3C standard. As any DOM standard, it allows developers to interact directly with the markup document in an object oriented fashion. Batik provides a complete static SVG DOM implementation, which allows developers to parse, process, and output SVG documents.

`JSVGCanvas` is a Swing UI component made solely for the purpose of displaying SVG documents. `JSVGCanvas` comes with `Interactor` classes that allows the user to zoom, rotate, and pan the document. There are also `Interactor` implementations that follows hyperlinks, and provide text selection.

The Batik bridge subsystem is the bridge that links the SVG DOM with an internal representation that is used to display SVG Graphics. This subsystem creates a mapping between an SVG DOM element object with its corresponding vector representation. For each SVG DOM element, there is a corresponding vector tree which are implemented as Batik GVT(Graphics Vector Toolkit) objects. The GVT is part of the Batik low-level classes.

II.2.1.2 Low Level Classes

The Batik low-level area classes provides all the groundwork for the functioning of the core area classes. They are not meant to be used by developers directly. There are three main subsystems: the Graphics Vector Toolkit (GVT), the SVG Renderer, and the SVG Parser.

The Graphic Vector Toolkit (GVT) provides a vector-object view of the SVG DOM tree. These objects are geared more towards graphics rendering and DOM event handling purposes. The GVT objects retain the same structure as their SVG DOM counterpart, and are generated by the Batik Bridge Subsystem.

The Batik SVG renderer subsystem is accountable for rendering and repainting of the GVT tree. Although the default renderer that comes with Batik is a raster-based renderer, the Batik specification states that the renderer is pluggable and is not limited to a raster implementation only. The default renderer computes the changed graphics vector tree and repaints the necessary are on the screen. Since SVG is vector-based, the renderer must also take into account the zoom factor currently being used, since zooming changes what pixels will be rendered onto the screen

The Batik SVG parser subsystem is made for reading and parsing SVG documents, translating them into the appropriate DOM representation. The parser subsystem is a conglomeration of several sub-parsers, which is necessary considering the existence of several complex SVG elements with complex attributes such as transform, path, and color.

II.2.1.3 JSVGCanvas

`JSVGCanvas` is a Swing UI element that allows for rendering SVG documents. `JSVGCanvas` interacts with several different underlying components in the Batik framework: the SVG Document Loader, the GVT builder, the `GVTRenderer`, and the `UpdateManager`. Although Batik supports only static SVG documents, it also allows for some dynamic changes to the contained SVG document. This is why all the above components are necessary for `JSVGCanvas` to be able to present an SVG document.

JSVGCanvas has one associated SVGDocument object, which comes from loading an SVG document and parsing it into an SVG DOM representation. This SVGDocument is then translated into vector objects by the GVT builder, and rendered onto the screen by the GVT renderer. Subsequent updates are handled by the UpdateManager that manages any addition, removal, and changes to the SVG Document associated with the JSVGCanvas.

II.2.1.4 Update Manager

The update manager is responsible for handling updates to the SVGDocument associated with the JSVGCanvas. The document is only rendered onto the screen when the UpdateManager finishes its scanning cycle. Batik also enforces that any updates to the SVG document be done in the UpdateManager thread.

Batik provides a way for developers to access this thread's update queue by either calling invokeLater or invokeAndWait, similar Swing.

The UpdateManager monitors its invoke queue for any updates, and performs those updates within a certain receiving window—effectively modifying the monitored SVGDocument. Once it is done incorporating the changes, the UpdateManager signals the renderer to render the updates onto the screen.

The UpdateManager starts asynchronously, which means that the UpdateManager may start any time the JSVGCanvas is loaded. This is a problem because all updates outside of the update manager's thread will be ignored. But since the UpdateManager follows the observer-observable design pattern, we can attach observers to the update manager. The UpdateManager will notify its observers when the UpdateManager is started, when an update is currently taking place, when an update is completed, and when the manager is being shut down.

As we do not have complete control of the update manager, these listeners provide sufficient information to track the progress of Batik in receiving dynamic updates.

II.2.2 Batik Event Handling

Batik conforms closely to the DOM event specification. DOM events are platform-independent, and works on DOM nodes and elements. At the time of the implementation, Batik supports both DOM Level 2 and DOM Level 3 events. This means that Batik supports mouse events, focus events, and keyboard events on the SVG nodes.

III JUICE (JAVA UI COMPONENT EXTENSION) FRAMEWORK

III.1 High Level Design And Architecture

There are three main items that the JUICE framework accomplished:

- Take Swing drawing instructions and route it into SVG instead of into the native platform.
- Translate Swing events from the particular SVG viewer rather than directly from its peer.
- Hide the complexity of the framework from the application developer.

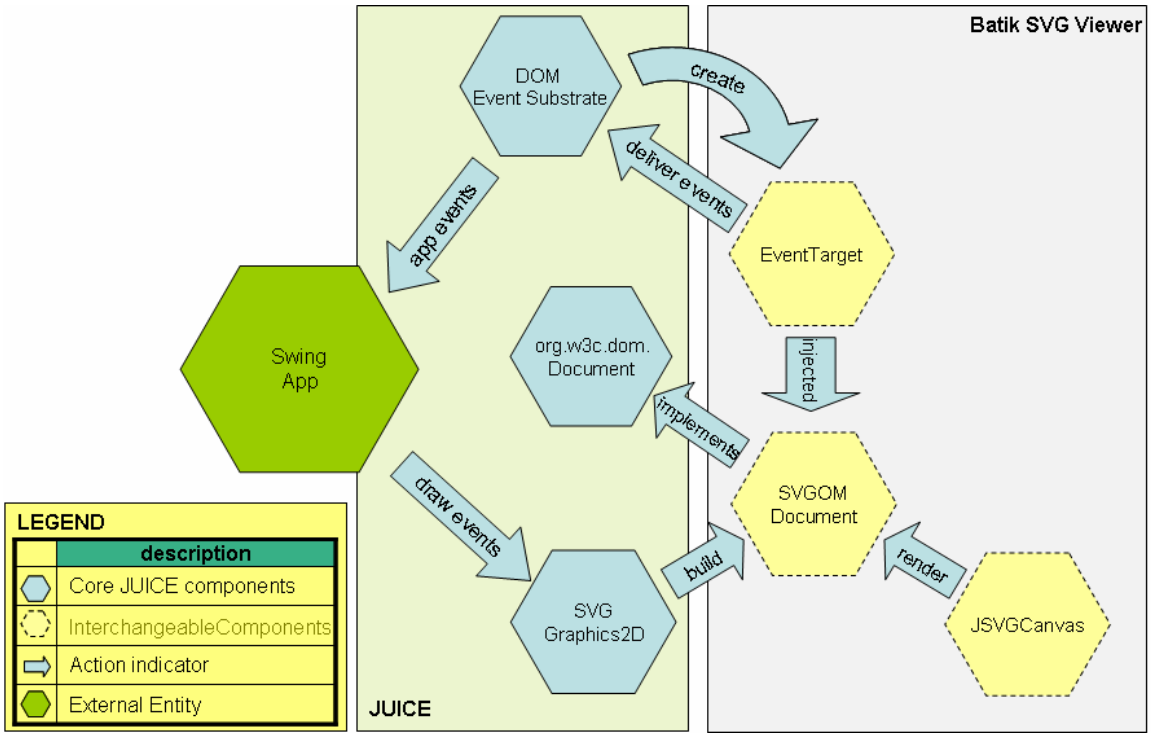

Figure 9: JUICE high-level design

The 'external entity' shown in **Figure 9: JUICE high-level design** above is the user code. The figure above contains the core of JUICE, namely the Graphics and Event Bridge subsystems. The user code will be instrumented at the bytecode level by JUICE;

JUICE replaces the standard classes with custom JUICE classes transparently in order to perform all the necessary routing and bridging.

III.1.1 Bridging Graphics

There are several design decisions to consider based on the problem domain. The first problem is that Swing drawing instructions are Java 2D instructions, which are raster based; the second problem is that Swing is designed based on the assumption that the target platform is a raster system. However, SVG is a vector-based system, where some of the raster concepts can not be applied. Therefore, there must be a part of the framework that translates Swing-generated raster drawing information into SVG DOM nodes. Fortunately, the Batik `SVGGraphics2D` class accomplishes just that. As mentioned before in chapter 2, `SVGGraphics2D` turns Java 2D drawing instructions into SVG DOM elements. We then display the contents in the Batik SVG viewer using `JSVGCanvas`.

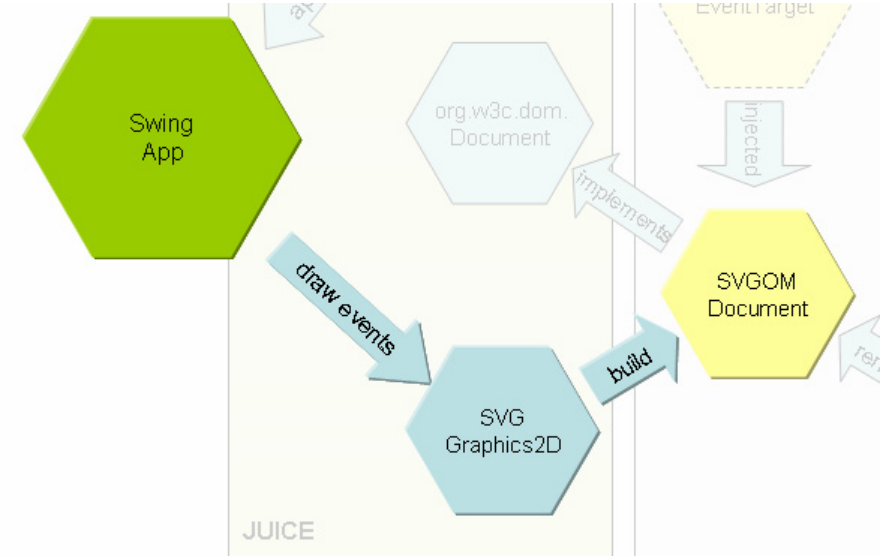

Figure 10: JUICE rendering subsystem

III.1.2 Bridging Events

JUICE takes events originating from the SVG Viewer and delivers them to the underlying Swing application, where the event will be processed. Therefore the interactive SVG viewer must have support for the DOM event standard, which is the main event type in SVG. The SVG DOM events will then be translated and sent to the Swing layer by JUICE.

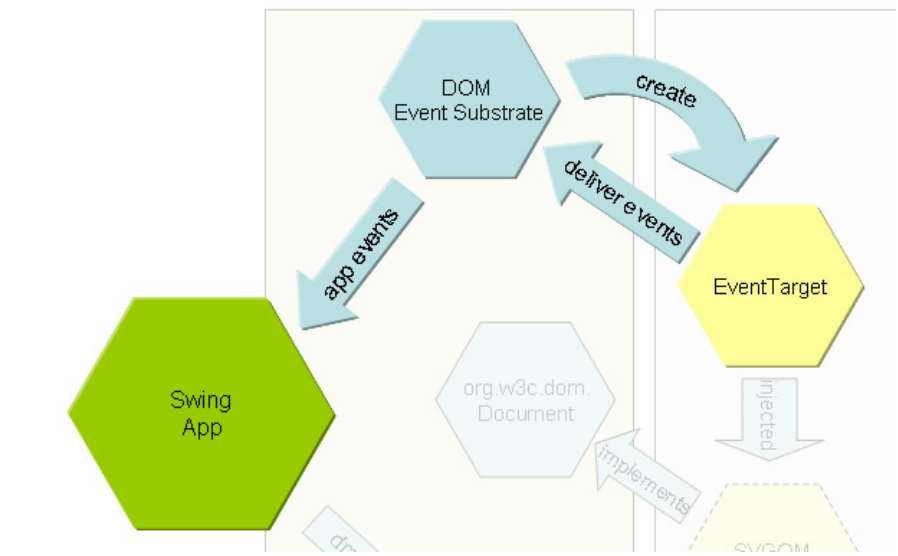

Figure 11: JUICE event subsystem

III.1.3 User Transparency

To fully leverage Swing, JUICE must be transparent to the user. The ideal goal is that the user implements an application in Swing, and then plugs it into JUICE for SVG platform rendering. This is accomplished through bytecode instrumentation/transformation.

III.2 Low-Level Design And Implementation

III.2.1 Bridging Graphics

III.2.1.1 Preliminary Approach

The initial rendering approach of JUICE (see **Figure 12: Initial rendering approach**) is simple and is divided into three main steps:
- JUICE intercepts Swing drawing content from the topmost lightweight container
- JUICE translates the drawing content from raster to vector using `SVGGraphics2D`
- JUICE replaces the SVG DOM of the viewer with the resulting vector tree

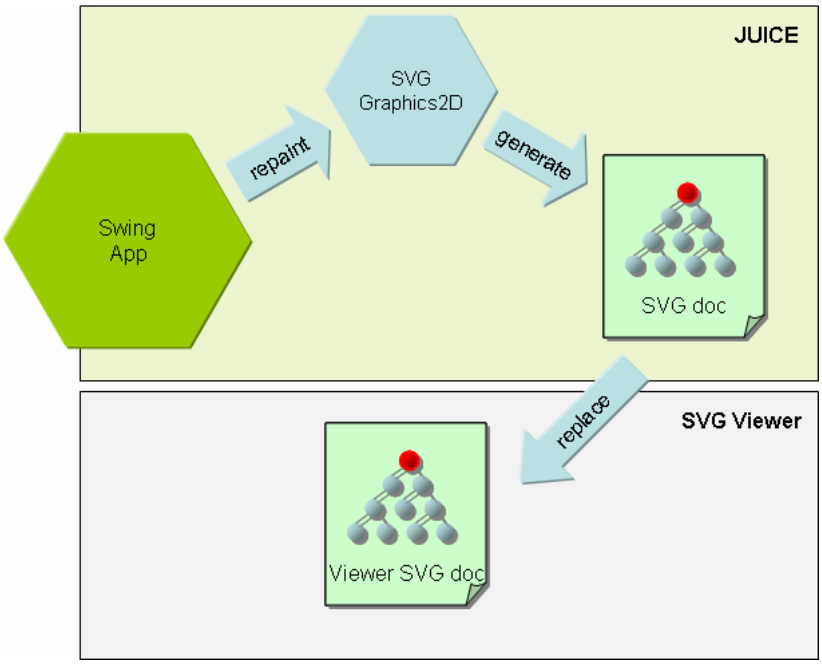

Figure 12: Initial rendering approach

First I need to create an SVG Graphics context; this is done by first creating an SVG document, then passing it into the SVG Graphics context constructor. This follows the example set in the Batik examples website.

Notice that we do not want double buffering. Double buffering redirects drawing to an offscreen buffer; once the process is completed, the offscreen buffer is then transferred to an onscreen buffer to be painted onto the screen.

To understand this process better, below is a code snippet from the `paintWithOffscreenBuffer` method of `JComponent`

```
        Graphics og = offscreen.getGraphics();
        Graphics osg = SwingGraphics.createSwingGraphics(og);
       og.dispose();
.
.
.
                    g.setClip(x,y,bw,bh);
                    g.drawImage(offscreen,x,y,paintingComponent);
                    osg.translate(x,y);
.
.
.
```

Notice that the above code actually invokes a command to draw an image taken from the offscreen buffer into the onscreen buffer. The results for drawing a double buffered JComponent in SVG using SVGGraphics2D is the following:

```
<svg>
    <g style="font-family:sans-serif; font-weight:bold;" transform="translate(24,402)">
      <image xmlns:xlink="http://www.w3.org/1999/xlink" style="clip-
path:url(#clipPath13);" width="310" xlink:show="replace" xlink:type="simple"
preserveAspectRatio="none" height="192" x="0" y="2"
xlink:href="
42uydB7xcZZn/n+m93Xvn9pLeE3roRUFQVixgA1nRFXVXVxQLllUBWf246i4KNhRFLEgVkKYgCDGQkIQkpCckucnt
ZeZO7+3//p73vGfO3MT9N2QF572f8znT7syZM+d8z++pr6jbjOZqjCbbmaI7maI4m2JqjO
ZqjOZpga47maI7maIKtOZqjOZqjCbbmaI7maIKtOZqjOZqjCbbmaI7maI4m2JqjOZqjOZpga47maI7maIKtOZqjOZ
pga47maI7maIKtOZqjOZqjCbbmaI7maI6/0rA2d8H//Lj/+7fQ0OQY9Xd0U/ucJXT6P5x9xGv2bX2WpkfLlIi8RC8
dGKZqNkbxWIQGD4xRIp6goXiKTKUEvzaSTuv/ ...
.
.
.
```

Notice that the drawing information is treated as embedded raster image information—PNG encoded in ASCII—because of the double-buffering. Unlike vector elements, raster images are not scalable. Since this is not desirable, double buffering must be disabled on Swing components. By disabling double buffering, the SVG generated will be pure SVG as shown in the following SVG snippet:

```
<svg>
<g transform="translate(16,25)" style="fill:rgb(102,102,102); font-family:sans-serif;
font-weight:bold; stroke:rgb(102,102,102);">
      <rect x="0" y="0" width="38" style="clip-path:url(#clipPath56); fill:none;"
height="38" />
      <rect x="1" y="1" width="38" style="clip-path:url(#clipPath56); fill:none;
stroke:white;" height="38" />
      <line x1="0" x2="1" y1="39" style="clip-path:url(#clipPath56); fill:none;
stroke:rgb(204,204,204);" y2="38" />
      <line x1="39" x2="38" y1="0" style="clip-path:url(#clipPath56); fill:none;
stroke:rgb(204,204,204);" y2="1" />
    </g>
    <g style="fill:rgb(204,204,204); font-family:sans-serif; font-weight:bold;
stroke:rgb(204,204,204);">
      <rect x="0" y="0" width="720" style="clip-path:url(#clipPath57); stroke:none;"
height="23" />
    </g>
    <g style="fill:rgb(153,153,153); font-family:sans-serif; font-weight:bold;
stroke:rgb(153,153,153);">
      <line x1="0" x2="720" y1="22" style="clip-path:url(#clipPath57); fill:none;"
y2="22" />
    </g>
    <g transform="translate(169,1)" style="fill:rgb(204,204,204); font-family:sans-serif;
font-weight:bold; stroke:rgb(204,204,204);">
      <rect x="0" y="0" width="63" style="clip-path:url(#clipPath58); stroke:none;"
height="21" />
      <text x="6" y="15" style="clip-path:url(#clipPath58); fill:black; stroke:none;"
xml:space="preserve">Tool Tips</text>
      <rect x="40" y="17" width="3" style="clip-path:url(#clipPath58); fill:black;
stroke:none;" height="1" />
    </g>
.
.
.
```

Notice that the items now are all pure vector elements such as rect, line, and text. The vector elements are scalable, making this true SVG, instead of just embedding images inside SVG. Of course drawing pure vector elements is expensive compared to just drawing images, but this is a price that JUICE must pay for the purpose of purity.

The next step is to trigger the repaint of the contained element. Since the `JuiceContainer` itself is a Swing component, it has a `paintComponent` method. In the paintComponent method, the container explicitly calls `paint` with `SVGGraphics` as a parameter on the contained element.

```
SVGGraphics2D g = new SVGGraphics2D(doc);
SVGGeneratorContext genCon = g.getGeneratorContext();
genCon.setComment(COMMENT);
g = new SVGGraphics2D(genCon,false);

panel1.paint(g);
```

This is the first part of the bridge between Swing and Batik SVG viewer. Remember that the SVG viewer we are using is `JSVGCanvas`.

The second part of the bridge is to replace the main group with the latest rendered version. The catch is that the merging must be done in the `UpdateManager` thread.

```
canvas.getUpdateManager().getUpdateRunnableQueue().invokeLater( new Runnable(){
public void run(){
Element oldGroup = canvas.getSVGDocument().getElementById("topLevel");
if(oldGroup != null)
        canvas.getSVGDocument().getDocumentElement().replaceChild(tlgImp,oldGroup);
}
});
```

This approach has a very high success rate for different ranges of applications because it simply takes the top-level rendering information, translates it into SVG, and puts the SVG Document into the `JSVGCanvas`.

But notice that the whole tree will have to be replaced every time there is a call to repaint the user interface. Remember that Batik has to convert a SVG DOM representation into a GVT, then the GVT has to be rendered onto the screen. Because of the multiple conversions going on, the performance degrades. Note also that multiple repaints are not uncommon for Swing components, this too contributes to the overall performance degradation.

III.2.1.2 Current Approach

III.2.1.2.1 Partial Tree Replacement

III.2.1.2.2 Subtree Mapping

If replacing the whole tree is bad, why don't we replace part of it instead? In theory, replacing only changed subtrees will increase the rendering speed compared to replacing the whole tree.

How do we determine this subtree exactly? One way to do this--while maintaining the same approach as before--is to first perform an XML DOM compare of the new tree with the old one, save the differences, and only replace the differences.

This is not efficient because:

- DOM comparison is a graph isometry problem, and fast comparison algorithms are usually heuristic based.
- Most of the time the newly rendered tree is totally different because the interleaving of drawing instructions.

This means if we perform the fastest DOM comparison method, on every single repaint, we will perform a significant number of operations which will slow down performance considerably. Remember that we actually would like to identify the changed nodes, thus we have to walk the tree.

Suppose we have a DOM compare algorithm that is O(m+n) in time complexity, where m,n is the number of nodes in the larger tree and smaller tree respectively.

Let
N be the set of nodes in the new tree
M be the set of nodes in the old tree
S be the set of nodes in the subtree containing nodes in M that is the same as nodes in N

Notice that $S = M \cap N$; also notice that S is the amount of improvement that we can actually save. We care more about the new tree instead of the old one; however we would like to not replace unchanged nodes if possible. This means that we still need to integrate the difference, and remove the contents of M that is not in S.

Operation	Steps
DOM Comparison	sizeof(N)+ sizeof(M)
Number of nodes to remove	sizeof(M-S)
Number of nodes to integrate	sizeof(N-S)
TOTAL nodal operations	$2 \times$ (sizeof(N)+sizeof(M)-sizeof(S))

Another way to write the total above is :

$$2 \times \text{sizeof}(M \cup N)$$

Thus, the best case is achieved when the size S is maximized. Even in that case we still end up with a full traversal of either the old or new node sets, which is not very good if we factor in multiple repaints coming from multiple containers.

The solution is to map a Swing component to a certain part of the SVG vector tree. That means each Swing component is associated with a node in the SVG tree. The benefits of this solution are that there is no need to determine which subtree has changed, since the components will know where to update already.

The immediate problem is identifying the points to augment this new functionality to the Swing components. At first glance the best way is to replace the graphics context and have the graphics context be aware of the components. But just replacing the graphics context will not work because drawing using the graphics context is a one-way operation, meaning the graphics context does not know nor need the information about who is currently using the graphics context itself.

Below is an example of the drawLine prototype declaration in the AWT Graphics class

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

In drawing a line, the graphics context only requires 2 points (x1,y1) and (x2,y2) and nothing more.

Thus, there are two general alternatives for mapping subtrees to available Swing Components (see **Figure 13: Direct and indirect mapping**):

1. Direct mapping; each Swing component Object will be directly mapped to the specific DOM subtree it is responsible for
2. Indirect mapping; each Swing component Object will be mapped to an XML ID of the DOM subtree instead of the whole DOM subtree object.

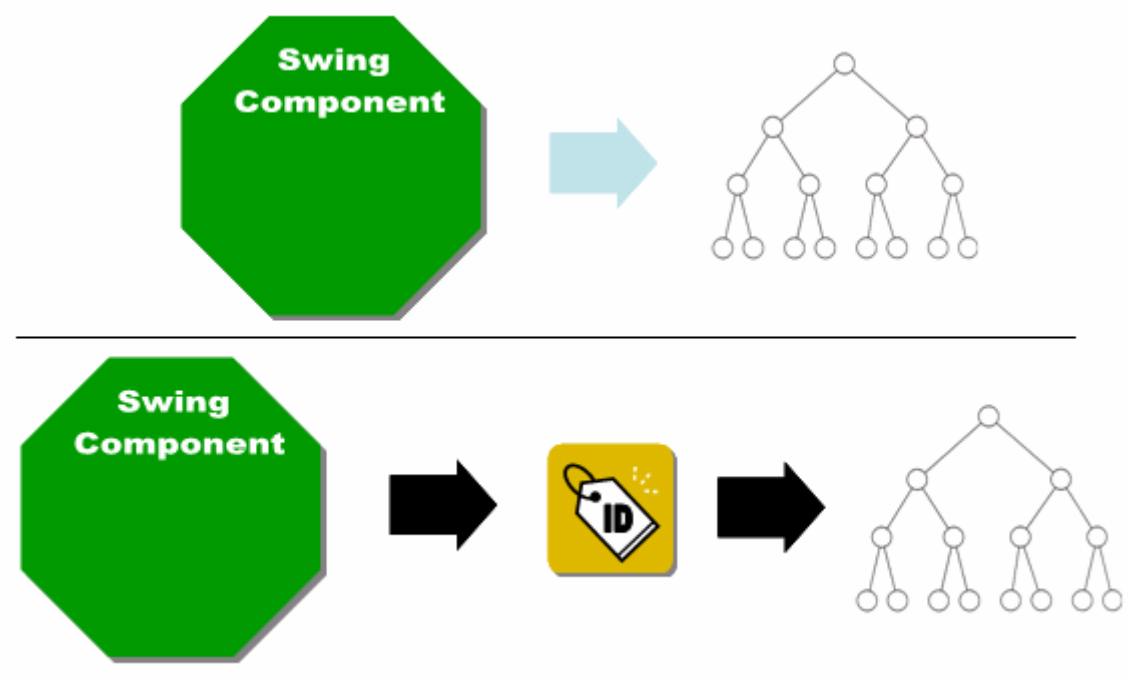

Figure 13: Direct and indirect mapping

The direct mapping approach has a performance advantage because the DOM object itself can be manipulated directly by the Swing component object. But, this approach is not practical for cross-viewer use because some viewers' DOM implementation is not written in Java.

By using indirect mapping we can associate the Swing component with the appropriate subtree while maintaining cross-viewer interoperability and generality because of XML ids. **Figure 14: Current rendering approach** below outlines the approach discussed in this section.

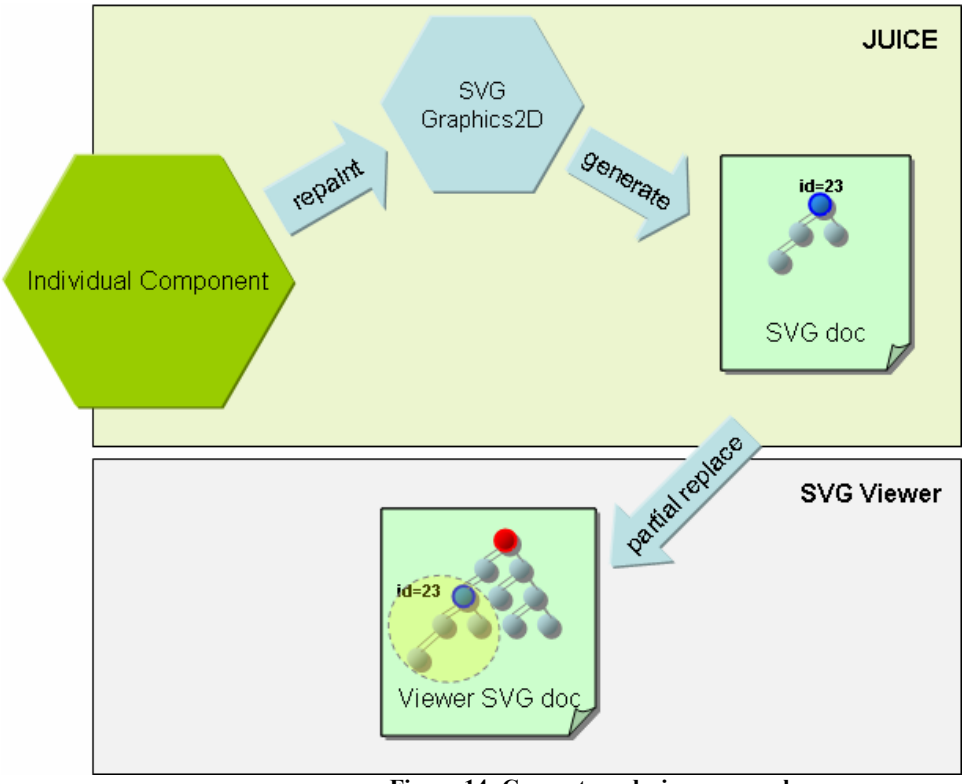

Figure 14: Current rendering approach

III.2.1.2.3 Graphics Interception Point

JUICE improves performance by replacing subtrees instead of whole trees by mapping Swing components to IDs in the SVG tree. Where does JUICE actually perform this?

To find out where to actually intercept these points, we need to examine the Swing/AWT paint workflow. Unfortunately, Sun's documentation on painting only touches the high-level overview on how painting works. For obtaining low-level information about Swing/AWT painting, we will have to take a look under the hood (i.e. the source code) of Swing/AWT.

Let's take as an example a `JButton`. Remember that Swing has three additional methods for rendering a component in addition to the AWT `paint` method.

Each of those methods takes one parameter of type `java.awt.Graphics`. Let 's examine the flow of `paintComponent` for example.

It turns out that `JButton` itself does not have a `paintComponent` method, but `JButton` inherits that particular method from `JComponent` directly.

So now let's take a look at the `paintComponent` method in `JComponent`. There are two relevant questions that pop up immediately:
1. Which other methods call this particular method?
2. Where did the graphics context come from?

Which other methods call this particular method?

Notice that there is a possibility that of an answer for the second question if there is an answer to the first question. A search—limited to `JComponent`—reveals that there are two points in which this occurs:

- the `paint` method
- the `paintWithOffscreenBuffer` method

Both the methods listed above take a graphics context in order to operate. So there is a need to search the Swing classes even deeper. After looking at several classes that calls the `paint` method, I found `_paintImmediately`—a package protected method in the `JComponent` class, which actually calls the method `getGraphics` to obtain the graphics context on the component.

Below is a code snippet from `_paintImmediately`

```
.
.
Graphics pcg = paintingComponent.getGraphics();
```

```
g = SwingGraphics.createSwingGraphics(pcg);
pcg.dispose();
.
.
g.setClip(paintImmediatelyClip.x,paintImmediatelyClip.y,
paintImmediatelyClip.width,paintImmediatelyClip.height);
paintingComponent.paint(g);
.
.
.
```

What does `getGraphics` on `JComponent` ultimately do? It in turn calls `Component`'s implementation which consults the peer of the component. If the component is lightweight, it requests graphics from the parent. If the component is heavyweight, it requests graphics from the peer itself.

There are two events that trigger the paint method. The first event is a `HierarchyEvent`, which is propagated down the component tree when the whole frame is being shown onto the screen. The sequence of instructions can be observed in **Figure 15: Hierarchy events and paint**.

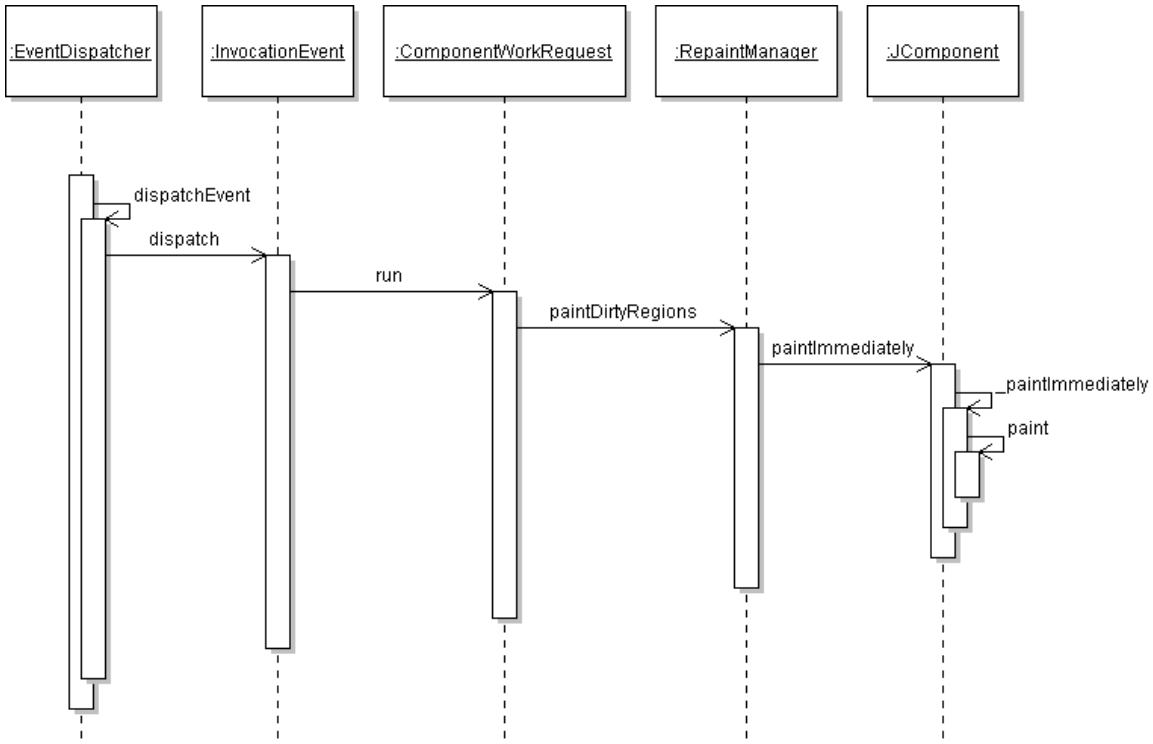

Figure 15: Hierarchy events and paint

The second event would be a `DispatchEvent` which in turn propagates up to the component hierarchy, then back down to trigger paint on the necessary components. This

is shown in the sequence diagram in the next figure, **Figure 16: Dispatch events and paint**.

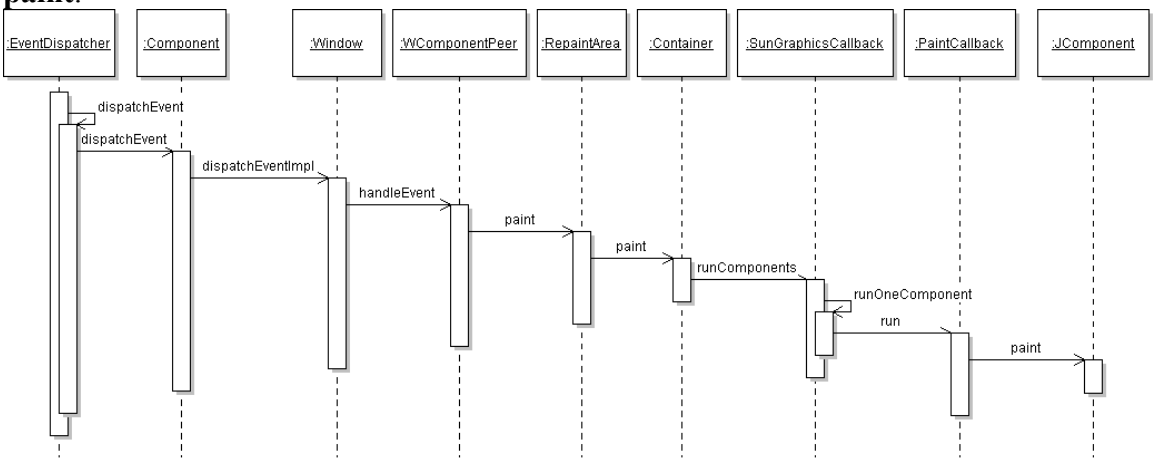

Figure 16: Dispatch events and paint

By observing both the Swing workflows above, the ideal place to swap is not on the peer level, but directly on the `paint` method level. Overriding `paint` makes JUICE react to any event that may trigger repainting.

III.2.1.2.4 Implementation Details

The approach to solve the problems above is simple, yet carefully crafted to maximize the performance of the JUICE Framework. JUICE takes advantage of Swing processes and benefits as much as it can. There are four integral parts to this approach:

1. The Ghost Window (see **Figure 17: The ghost window**): `JuiceWindow`
2. The Special Graphics Context: `JuiceGraphics2D`
3. The Replacement JUICE 'Components', for example: `JuiceJButton` for `JButton`
4. The bytecode transformer

III.2.1.2.4.1 Propagating Hierarchy Events

Swing distinguishes initial painting and subsequent painting as outlined in a JFC article[12] about Swing/AWT painting. Looking into the Swing source code reveals that initial painting is triggered by a `HierarchyEvent` that is propagated when a top level container is being shown or modified. This is where the ghost window, `JuiceWindow`, comes in.

`JuiceWindow` is an invisible `JWindow` extension, which is necessary for several different purposes:

- Keeping the proper Swing hierarchy intact. All lightweight Swing components are assumed to exist within a certain top-level parent/ancestor Swing component.
- Propagation of component visibility and showing; Swing apparently distinguishes these notions, but it does not really matter for our intent and purposes.
- Propagation of hierarchy events. Since JUICE is not actually showing the component on the screen, JUICE needs to make Swing think that the components are contained in a visible top level container. The way to do this is to imitate the way actual Swing top-level containers work.

After looking through Swing source code and performing some experimentation, I found that the `HierarchyEvent` objects generated must be propagated by a top-level container to the child components to ensure Swing compatibility. The propagation of hierarchy events is also necessary to trigger initial painting.

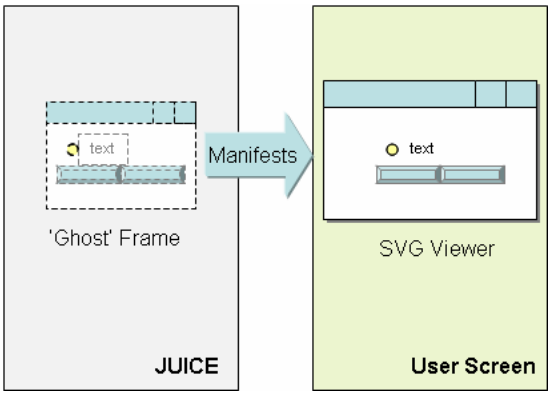

Figure 17: The ghost window

All the user-programmer components are added by JUICE into `JuiceWindow`. The `JuiceWindow` is not displayed on the screen; however, `JuiceWindow` announces its visibility to the child components by propagating the appropriate hierarchy events down the component tree. This triggers the normal Swing repaint flow and invoke the paint method on the child components.

III.2.1.2.4.2 Paint Replacement

The paint replacement workflow can be summarized in **Figure 18: JUICE paint workflow**.

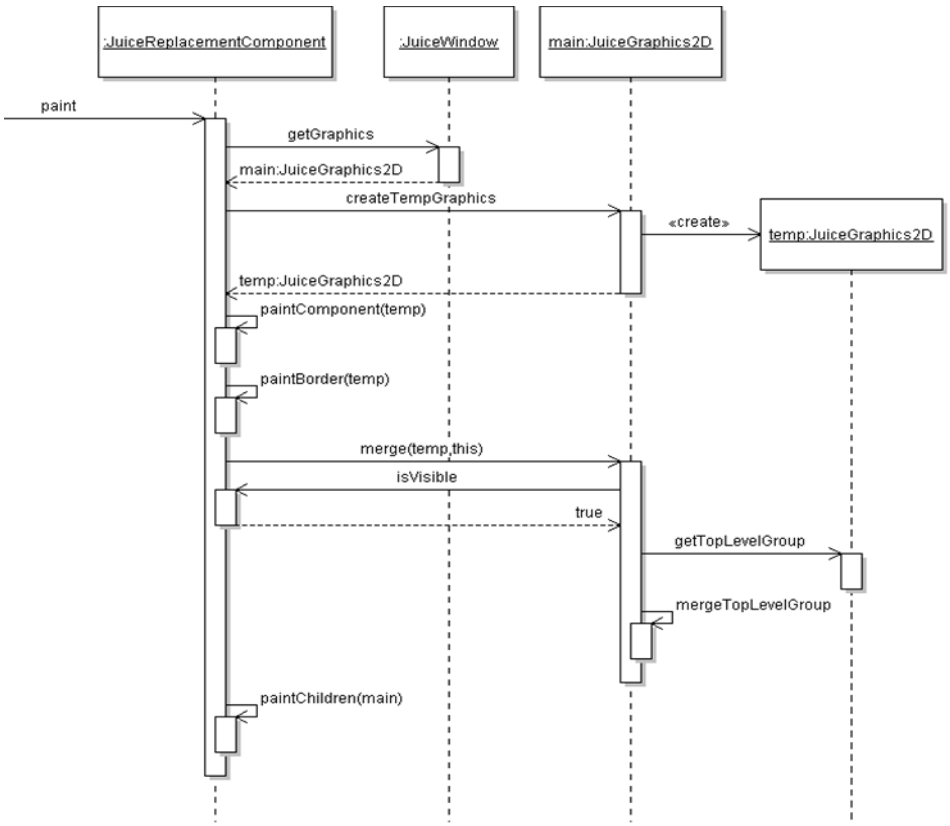

Figure 18: JUICE paint workflow

The special graphics context, `JuiceGraphics2D`, is required for processing the Swing instructions, and embedding them into the SVG tree. This graphics context, coupled with the 'Replacement Juice Components', allow the 'Swing component & subtree ID coupling' to be realized. The special graphics context handles:

- The hierarchy of paint instructions. This means the parent Component↔ child Component painting relationship is preserved.
- Injection of subtrees into the main viewer SVG tree.

An added complication is Z-ordering handling. One of the fundamental difference between a raster substrate and the SVG tree is the way it handles overpainting (painting over). In a raster system, pixels that are painted over are lost. In SVG, vector items that are painted over are still there. An easy way to illustrate this is by thinking of raster as a bitmap. The bits that are replaced are lost, thus they need to be replaced. When a component is hidden, JUICE forces a call to the `paint` method of the hidden component. Observe the sequence diagram in **Figure 19: Component hidden paint workflow** below.

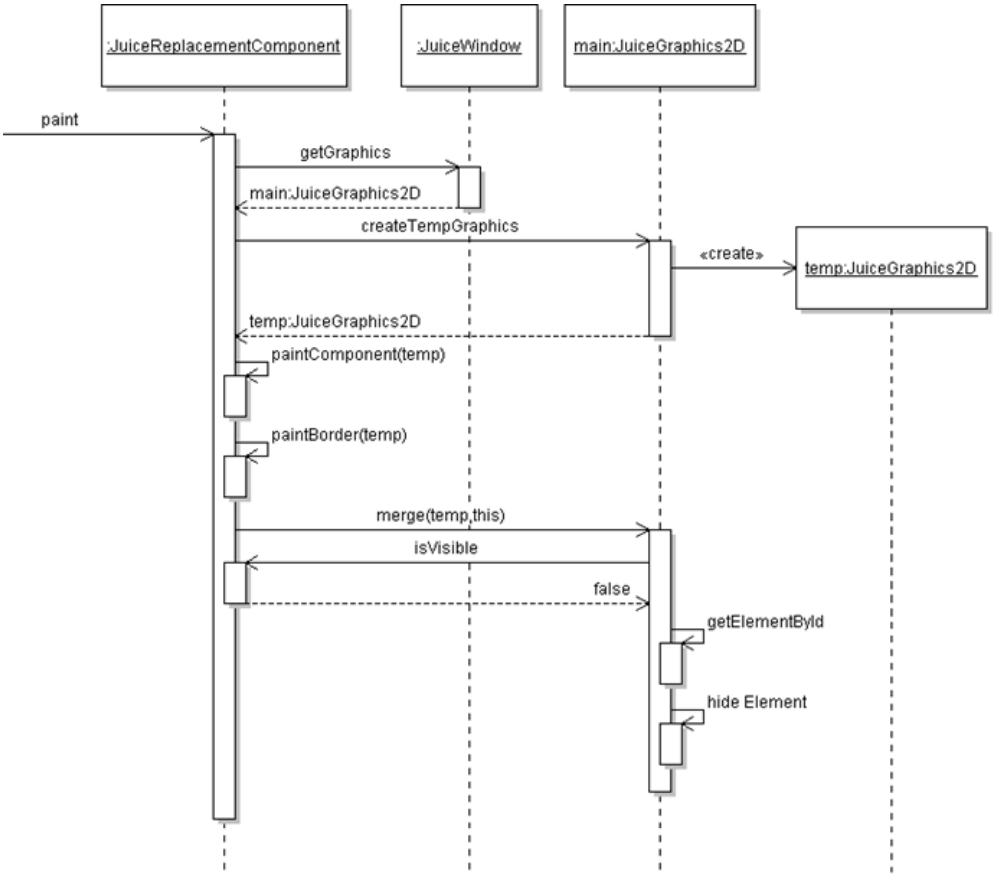

Figure 19: Component hidden paint workflow

Why not just call `repaint`? Calling `repaint` does not necessarily invoke `paint` because there is a possibility that the Swing `RepaintManager` will determine that there is no dirty rectangles, thus there is no need to update the screen; this possibility occurs when another component paints on top of the hidden component, causing it to be hidden. Painting a new object in SVG means appending a new node at the end of the vector tree. Unless removed, those hidden nodes will still exist.

This is best described in the example of Swing's `JTabbedPane`. **Figure 20** is taken from a Swing tutorial page[16] on `JTabbedPane`.

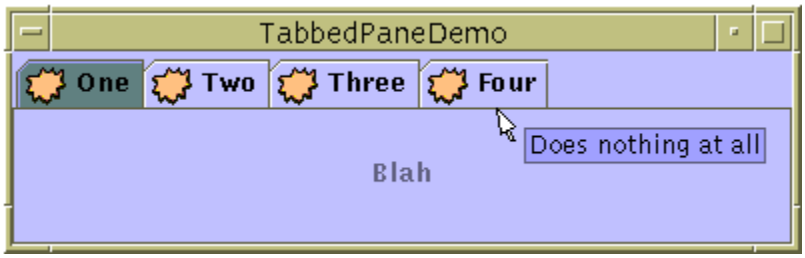

Figure 20: JTabbedPane

Remember that IDs are used to pair a Swing component to an element in the tree. Below is a sample user scenario:

1. The first tab is shown for the first time:
    a. A subtree with the ID for the first tab is generated
    b. The item is appended to the end of the main SVG viewer tree.
    c. The contents of the first tab are shown
2. The user clicks on the second tab
    a. A subtree with the ID for the second tab is generated
    b. The item is appended to the end of the main SVG viewer tree.
    c. The contents of the second tab are shown
3. The user clicks on the first tab again
    a. A subtree with the ID for the first tab already exists
    b. That subtree is updated with whatever content from the first tab
    c. The contents of the first tab, however, are never shown because the first tab SVG subtree comes before the second tab SVG subtree.

The solution to this Z-ordering problem then, is to explicitly remove hidden elements from the tree. This ensures that the completely hidden elements are not displayed. Also, removing hidden elements from the tree will actually yield a performance improvement because the size of the tree is smaller now. Partially obscured components are still drawn using a specified clipping path

The graphics context is also responsible for removal of hidden elements. To increase efficiency, this is done only when a component is hidden. JUICE takes advantage of Swing's workflow here by listening for events fired when the Swing component is hidden. JUICE Components listen for `ComponentEvents` and reacts to component hidden events

All of this graphics rerouting and embedding ID for Swing components is not possible without replacement JUICE Components. Most of these replacement components contain

redirection code that works together with the `JuiceGraphics2D`, while piggybacking on Swing workflows and UI separation model.

The JUICE replacement is a Swing component that extends the original Swing component; for example, a `JuiceJButton` extends `JButton`. Each JUICE replacement component contains:

- A replacement `paint` method; this method swaps the provided `Graphics` object with `JuiceGraphics2D`, calls `paintComponent`, `paintBorder`, and `paintChildren` in the correct order. This is the point where all the the Swing Component and SVG Tree ID mapping occurs.
- A special `ComponentListener` that listens for component hidden events. This listener will signal `JuiceGraphics2D` to remove the subtree associated with the hidden component.

Also remember that all updates such as hiding an element and appending an element to the SVG DOM of the viewer must be done inside the `UpdateManager` thread.

The Java bytecode transformer is responsible of replacing the actual Swing classes with the replacement JUICE classes and will be discussed in the next Chapter.

III.2.1.2.4.3 SVG Viewer Connection

The viewer connection is provided by `JuiceContainer`, which takes instrumented user-implemented classes and provide these classes with a `JuiceGraphics2D` object. The `JuiceGraphics2D` itself is constructed by the `JuiceContainer` with a link to the viewer's SVG Document. The viewer also wants all updates to be queued onto its `UpdateManager` queue; this insertion is done by the `JuiceContainer` as well.

III.2.2 Bridging User Events

In Swing, user interaction events can be categorized into two major groups: keyboard events and mouse events. The DOM Level 2 event model also supports these two types of events. The difference is that Swing/AWT events are delivered to the component, while DOM events are delivered to the particular DOM node--in our case, the particular SVG element. These differences warrant a connection between the SVG element and the Swing component it represents.

For mouse events, the simplest approach is to use a transparent SVG rectangle on top of the whole application. This means that in the SVG tree, the rectangle must be the last element.

```
<rect x="0" y="0" width="730" style=" opacity:0" id="mainrect" height="649" />
```

The width and height depends on the application's width and height respectively. Because the Batik SVG viewer supports the DOM Level 2 events standard, it is possible for a developer to attach DOM Event listeners to the rectangle during creation.

The code below is taken from the DOM Level 2 specification on EventTarget.

```
interface EventTarget {
  void addEventListener(in DOMString type,
                                    in EventListener listener,
                                    in boolean useCapture);
  void removeEventListener(in DOMString type,
                                     in EventListener listener,
                                     in boolean useCapture);
  boolean dispatchEvent(in Event evt)
                                    raises(EventException);
};
```

The JUICE framework then adds mouse events listeners to the interface. Each listener listens for a particular type of events. The supported mouse events from the DOM Level 2 specification are: click, mouse down, mouse up, mouse over, mouse move, and mouse out . Notice that we do not have a mouse drag as in AWT. Mouse drag has to be simulated in order for it to work. We need to implement a simple state machine for simulating MouseDrag.

The EventListeners will convert the appropriate event into its corresponding Swing/AWT event and artificially inject it into the application's event queue. Converting an event requires translation from the viewer coodinate system to the Swing coordinate system. Remember that SVG is a vector technology, which enables lossless zooming. Batik allows for zooming, and rotating the SVG Document. The workflow is described in **Figure 21** below.

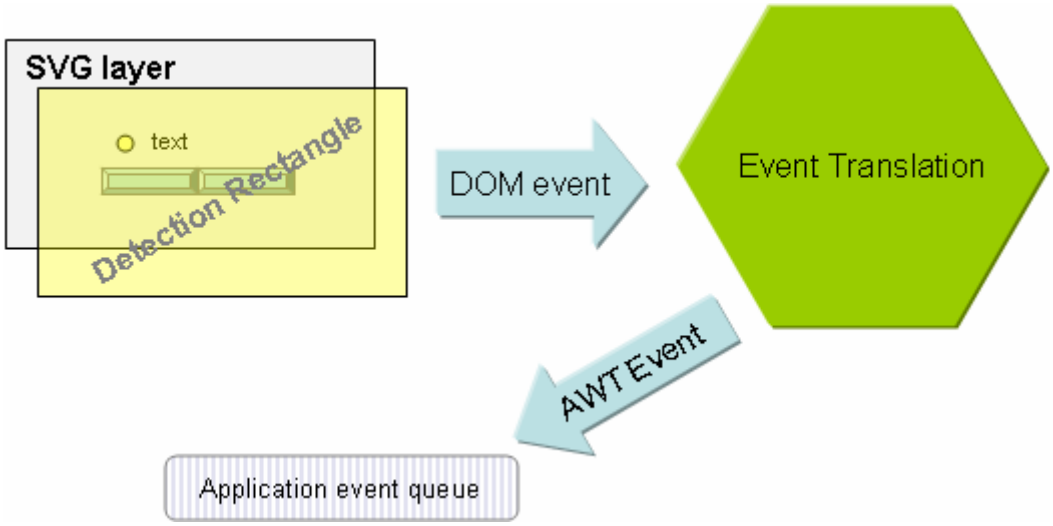

Figure 21: The detection rectangle

Keyboard events are non-existent in DOM Level 2. But, since Batik also supports DOM Level 3 events, the JUICE framework takes advantage of this to intercept, wrap, and route appropriate focus and keyboard events using similar methodology as the mouse events. For keyboard events to work properly JUICE needs to intercept focus events as well. All of this is performed by `JuiceContainer`, since it is the main link to the SVG viewer.

IV JAVA BYTECODE TRANSFORMATION

One of the goals of this project is user transparency. This means the user of this framework should be able to develop code in Swing, and not be aware of any of the existing JUICE Framework classes. The developer then can debug and run the UI separate from the framework. Once the code is ready, the framework will take care of references to JUICE classes using bytecode Transformation.

The overall objective of bytecode transformation is to replace every use of Swing classes with corresponding JUICE Replacement Components.

IV.1 Background

At first we attempted to replace Swing classes during runtime by using a separate classloader that loads my version of the classes. However, these custom classes are considered to be not the same class as the original, because it was loaded using a different classloader. This breaks typecasting and referencing to that particular class. The system classloader does not provide a mechanism for loading classes from bytes on the disk, unless we specify a different classloader. But if we do employ this methodology, it will restrict the programmer as the programmer is forced to use our specific classloader for this to work.

Remember that JUICE needs this for rerouting paint calls from Swing into SVG. Subclassing and overriding the paint methods is the solution, but we do not want to impose this requirement to the programmer. However, bytecode transformation allows changing of the referenced classes without requiring any knowledge of the source code. Bytecode transformation is then the solution to this problem.

IV.2 Transformation Goals

There are several goals for the transformation. The first goal is to replace every extension of a Swing Component with the corresponding JUICE Component. **Figure 22** is an example of this.

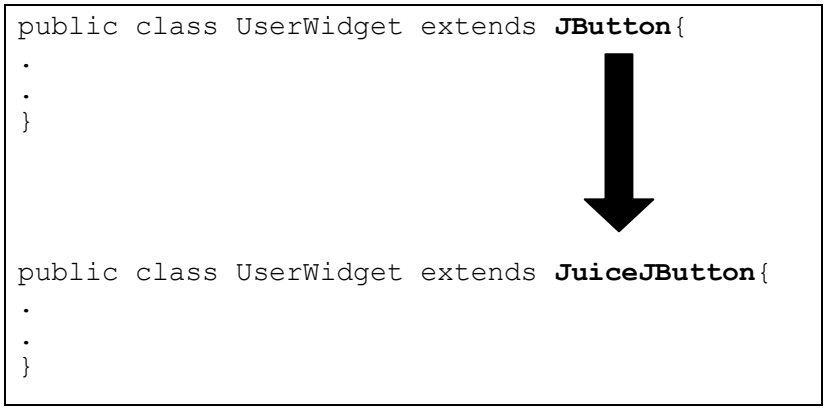

Figure 22: Extension transformation

The second goal is to intercept any constructor calls to a Swing component with the corresponding constructor of a JUICE Component. This is possible to do because the JUICE Components themselves extend the appropriate Swing classes. **Figure 23** is an example of this.

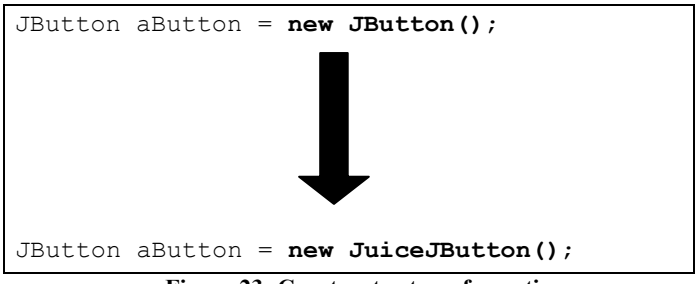

Figure 23: Constructor transformation

And last but not least, calls to any static methods on a Swing component are redirected to operate on the corresponding JUICE Component. This is important, as sometimes static calls will return a Swing component too. **Figure 24** is an example of this.

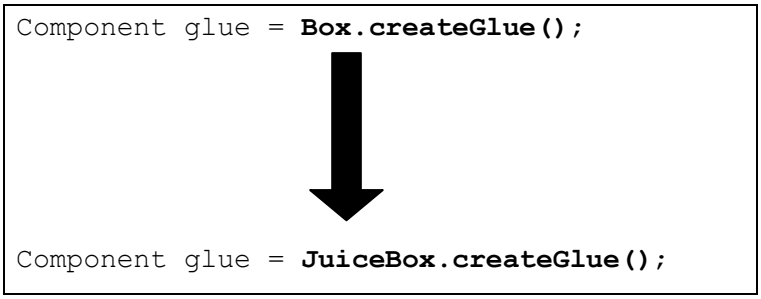

Figure 24: Static method call transformation

There is no need to modify static field access as the JUICE Replacement Classes will inherit the static field values from the extended Swing Classes.

IV.3 Why Bytecode Transformation

There are numerous possibilities in which a user might write a piece of code to invoke a Swing construct. A source code lexical transformer—which at first glance might be simpler to implement—turns out to be much more complex for our intent and purposes, because of these code variations. However, for our current intent, there are not a lot of variations in the bytecode structure; for example, a construction call will always start with a 'new' JVM instruction[13] followed by an '<init>' JVM instruction.

IV.4 Implementation

The user source code is first compiled using any Java compiler that the user chooses. The resulting classes are then instrumented on the fly with the aid of the Byte Code Engineering Library (BCEL)[14], which is also implemented under the Apache License. BCEL provides an object oriented interface to access the underlying Java bytecode; this allows developers to manipulate Java bytecode programatically using Java.

IV.4.1 Transformations

The bytecode transformations that we perform on user code can be put into two main categories: interface transformation, and code transformation. Interface transformations involve changing the superclass along with any implemented interfaces. Code transformation involves modifying the user's code (e.g. inside methods) itself.

IV.4.2 Transformation Set

The transformation set is the set of supported Swing classes that will be transformed into JUICE classes.

The transformation set file itself is better described as a mapping between Swing components and JUICE components. This transformation set file is a simple text file that can be modified as needed. By using a properties file, we enable declarative dynamic remapping of source and target classes to be transformed.

IV.4.2.1 User Extension And Interface Transformations

In Java bytecode, this definition is located on the very top of the *.class file. Below is a snippet from Swing's JButton.class

```
Compiled from "JButton.java"
public class javax.swing.JButton extends javax.swing.AbstractButton implements
javax.accessibility.Accessible{
public javax.swing.JButton();
  Code:
   0: aload_0
.
.
.
```

Notice that the `extends` is in the very top. JUICE only requires one extension and interface transformation: to replace every user extension of a Swing component class with the corresponding JUICE component class. The following code snip is taken from the JUICE `SwingInterfaceScanner` transformer class:

```
if (subRule.containsValue(cg.getClassName()) || cg.isInterface()) {
continue;
}
if (subRule.containsKey(cg.getSuperclassName())) {
es.changeSuperClass(cg.getClassName(), subRule.getProperty(cg.getSuperclassName()));
}
```

The code ignores transformation of Swing interface implementations because we are only interested in Swing class extensions. The transformation set is consulted to determine if a class is included and continues to switch the extended class to the corresponding target class.

IV.4.2.2 User Code Transformations

For the current framework implementation, there are two user code transformations that we are interested in: Swing constructor calls, and Swing static method calls.

IV.4.2.2.1 Swing Constructor Calls

In Java, there is one way to directly invoke a constructor, and that is to use the new keyword followed by the constructor and the appropriate parameters.

```
package mypackage;

import java.util.TreeMap;

public class ChangeMap  {

   public ChangeMap(){

      Map b = new TreeMap();

   }

}
```

The above code will be compiled into Java bytecode, and will look like the following:

```
Compiled from "ChangeMap.java"
public class mypackage.ChangeMap extends java.lang.Object{
public mypackage.ChangeMap();
  Code:
   0:   aload 0
   1:   invokespecial   #9; //Method java/lang/Object."<init>":()V
   4:   new     #11; //class java/util/TreeMap
   7:   dup
   8:   invokespecial   #12; //Method java/util/TreeMap."<init>":()V
   11:  astore_1
   12:  return

  LineNumberTable:
   line 12: 0
   line 14: 4
   line 20: 12
  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      13      0    this       Lmypackage /ChangeMap;
   12      1      1    b       Ljava/util/Map;

}
```

Note that the TreeMap construction is actually a new instruction, which is then followed by an invokespecial of type <init>. The dup instruction in between the two instructs the virtual machine to make a duplicate of the object on the stack. It will exist if we perform an assignment operator. So the JUICE transformer need to take care of new and invokespecial of type <init> on Swing components. Every time there is a construction of a supported Swing component in the list, we will actually replace the construction with the corresponding JUICE Replacement class.

Therefore, replacing Swing constructor calls is a two-step process:

- replace all calls to the ‘new’ JVM instruction
- replace all calls to the ‘<init>’ JVM instruction

IV.4.2.2.2 Swing Static Method Calls

The last transformation is the static factory method calls transformation. How does one determine static factory methods exactly? The answer is that one does not even need to know if a certain static method call is a factory method call or not; just replace every static method call on any Swing component with a static call to the appropriate replacement JUICE component. This is beneficial for the JUICE framework because:

- The JUICE replacement component inherits all the static methods and fields from the appropriate Swing class
- The JUICE replacement component can re-implement the static code if needed

By examining the bytecode structure of these calls, JUICE needs to find and replace invokestatic instructions. Below is the implementation of the replacement call residing in the JUICE `SwingMethodCallScanner` transformer.

```
if (invoke.getOpcode() == Constants.INVOKESTATIC) {
        String className = invoke.getClassName(cpg);
        if (subRule.containsKey(className)) {
                String classReplacementName = subRule
                                .getProperty(className);
                Type[] argTypes = invoke.getArgumentTypes(cpg);

                oneInstructionHandle.setInstruction(factory
                                .createInvoke(classReplacementName, invoke
                                                .getName(cpg), invoke
                                                .getReturnType(cpg), argTypes,
                                                Constants.INVOKESTATIC));

        }
}
```

IV.4.2.2.3 Transformation Chaining

The last piece of the puzzle is how to actually apply the above transformations to the user code. One could always re-invent the wheel and write one's own classfile reader and directory walker using pure BCEL. But we would like to leverage an existing framework called JMangler[15] instead.

JMangler is a bytecode transformation pipelining build on top of BCEL. To use JMangler, knowledge of BCEL transformations is required; however, the advantages of using JMangler are:

- JMangler handles the tedious task of iterating through the class files and directories

- JMangler transformations can be chained and ordered declaratively.
- Transformation code can be tested independent of each other.
- The developer just needs to write the transformation as if he/she is transforming one class

Unfortunately all the examples given on the website are outdated, and could not be applied to the current version. After looking under the hood of JMangler, and assembling bits and pieces of information from forums and mailing lists, I managed to make JMangler work for JUICE.

Below is the current JUICE JMangler properties file:

```
org.cs3.jmangler.bceltransformer.class.transformer1=info.yuwono.classreplace.SwingInterfa
ceScanner
org.cs3.jmangler.bceltransformer.id.transformer1=SwingInterfaceScanner
org.cs3.jmangler.bceltransformer.parameter.transformer1=transform.list
org.cs3.jmangler.bceltransformer.class.transformer2=info.yuwono.classreplace.SwingMethodC
allScanner
org.cs3.jmangler.bceltransformer.id.transformer2=SwingMethodCallScanner
org.cs3.jmangler.bceltransformer.parameter.transformer2=transform.list
org.cs3.jmangler.bceltransformer.class.transformer3=info.yuwono.classreplace.PeepholeOpti
mizer
org.cs3.jmangler.bceltransformer.id.transformer3=PeepholeOptimizer
org.cs3.jmangler.bceltransformer.InterfaceTransformer1=SwingInterfaceScanner
org.cs3.jmangler.bceltransformer.CodeTransformer1=SwingMethodCallScanner
org.cs3.jmangler.bceltransformer.CodeTransformer2=PeepholeOptimizer
org.cs3.jmangler.bceltransformer.Settings.verboseON=false
org.cs3.jmangler.bceltransformer.Settings.startUpMessage=false
org.cs3.jmangler.bceltransformer.Settings.checkPartialOrder=false
org.cs3.jmangler.bceltransformer.Settings.dumpClassSet=false
org.cs3.jmangler.bceltransformer.Settings.classSetSecurityLevel=0
org.cs3.jmangler.bceltransformer.Settings.maxIterations=10
org.cs3.jmangler.hook.EnableLoadClass=false
```

The peephole optimizer is added for fun. This transformer is a modified version of the peephole optimizer from the old BCEL example on the official BCEL site. It is modified to work with JMangler.

IV.4.2.2.4 Performing The Transformations

The transformations are performed offline to avoid any performance hit from doing online transformations. To facilitate transparency and ease of use, JUICE provides the following workflow:

- the user programmer drops code inside a designated source folder.
- the user programmer puts their required libraries (jars) inside a designated library folder.
- the user programmer calls runs the `build.xml` script provided with JUICE to compile and transform their code.

The ideal goal of all this, is that the user can build and debug the user application in Swing, then run it in JUICE easily.

V LIMITATIONS

The JUICE framework has the following limitations:

- JUICE currently only runs under the Apache Batik SVG Viewer.
- Composite elements such as `JScrollPane`, `JSpinner` and `JColorChooser` are not supported. These components have other component declarations inside, and JUICE could not instrument Java system classes because of security and class loading issues.
- JUICE only supports lightweight components because it does not cover the the problem of spawning a new window inside the SVG viewer. For example, `JDialog` is also not supported because its `Window` is a heavyweight object that lives inside the operating system UI display.
- JUICE does not perform as fast as a native SVG widget implementation because of the translations occuring between raster and SVG.

VI CONCLUSION

SVG (Scalable Vector Graphics) is a W3C standard XML markup language that is used to describe vector illustrations. SVG is an ideal format that can be rendered equally well on the web, desktop, and mobile devices. Due to its cross-platform capabilities and support for events, SVG may potentially be used in interactive graphical front-ends.

Although there has been research into the development of widgets and widget libraries for SVG, these prior efforts produced limited widget offerings. None of these offerings to date are as complete as Sun's Swing GUI Toolkit.

Sun's Swing Framework offers a complete desktop GUI solution mainly geared towards the desktop developer. Swing comes with a complete set of widgets with features such as pluggable look-and-feel, localization, cross-desktop-platform, and extensibility.

My framework—the JUICE framework—solves this problem by allowing Swing widgets to be used in SVG. The JUICE framework solves the non-existence of complete widget sets for SVG by extending a complete widget-set: Sun's Swing. The goal of the JUICE framework is to let the developers code their widgets and callbacks in Swing, without worrying about SVG at all.

However, Swing is assumed to be run on a raster display substrate rather than a vector-based one such as SVG. Consolidating the two requires understanding of the internal undocumented Swing painting and event workflows as well as the differences between the two paradigms.

Because of the differences between the two paradigms, performance was an initial problem for JUICE. The preliminary JUICE implementation used a naïve method of bridging Swing into SVG, which rendered the user interface unusable due to bad UI response time. After further research and experimentation, the current JUICE approach managed to increase performance to an acceptable level by piggybacking on the Swing repaint and event workflow, and by using mapping techniques to minimize the updates to the SVG tree.

The research and implementation of JUICE demonstrated the problem of retargeting a raster object-oriented GUI framework into a vector display substrate. JUICE also provided a practical solution to consolidate the different rendering and event handling methodology of both the raster—Swing—Object-oriented UI model and XML vector—SVG —technologies.

JUICE also applied bytecode instrumentation techniques to achieve user transparency; we wanted the users to implement all of their user interface in pure Swing. bytecode transformation allows for altering the user class on the binary class-file level. This means

that JUICE has the ability to take existing compiled Swing applications, and have them run in SVG. JUICE also managed to pipeline and chain the bytecode transformations through the use of JMangler, a third-party framework that is based on BCEL.

VI.1 Future Work

The current implementation of JUICE only runs on the Apache Batik SVG viewer. But, the concept and techniques used can be applied in extending JUICE to support different SVG viewers.

Highly composite Swing widget elements are not currently supported in JUICE. This is due to Java security restrictions that disallow any modifications to Java system API code.

Drag-and-drop is also another challenge that the current JUICE framework does not tackle. Drag-and-drop (DnD) is potentially challenging due to the platform-dependent nature of DnD and cross-platform viewer support that JUICE aims to have.

VII REFERENCES

1. Scalable Vector Graphics (SVG) 1.1 Specification
   http://www.w3.org/TR/SVG11/
2. Apache Batik SVG Viewer
   http://xml.apache.org/batik/
3. Java Swing API and the Java Foundation Classes
   http://Java.sun.com/products/jfc/index.jsp
4. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Lemort, A., and Mertz, C. 2004. Revisiting visual interface programming: creating GUI tools for designers and programmers. In Proceedings of the 17th Annual ACM Symposium on User interface Software and Technology (Santa Fe, NM,USA,October 24 - 27, 2004). UIST '04. ACM Press, New York, NY, 267-276.
5. Marriott, K., Meyer, B., and Tardif, L. 2002. Fast and efficient client-side adaptivity for SVG. In Proceedings of the 11th international Conference on World Wide Web (Honolulu, Hawaii, USA, May 07 - 11, 2002). WWW '02. ACM Press, New York, NY, 496-507.
6. Fettes, A., Mansfield, P. 2004. SVG-Based User Interface Framework. In Proceedings of SVG Open 2004 (Tokyo, Japan, 2004).
7. Antoniou, B., Tsoulos, L. 2004. Converting raster images to XML and SVG. In Proceedings of SVG Open 2004 (Tokyo, Japan, 2004).
8. Lindsey, K., SVG using ECMAScript callbacks for event processing; SVG using ECMAScript connector and Java callbacks for event processing
   http://www.kevlindev.com/geometry/3D/svg3d/index.htm
9. Lewis, C. T., Karcz, S., Sharpe, A., Parkin, I.A.P., Development of an SVG GUI for the visualization of genome data. In Proceedings of SVG Open 2002 (Zürich, Switzerland, 2002)
10. Document Object Model (DOM) Level 2 Core Specification
    http://www.w3.org/TR/DOM-Level-2-Core/
11. Document Object Model (DOM) Level 3 Core Specification
    http://www.w3.org/TR/DOM-Level-3-Core/
12. Painting in AWT and Swing
    http://java.sun.com/products/jfc/tsc/articles/painting/
13. The Java Virtual Machine Instruction Set
    http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions.doc.html
14. The Byte Code Engineering Library
    http://jakarta.apache.org/bcel/
15. The JMangler Project
    http://roots.iai.uni-bonn.de/research/jmangler/

16. How to Use Tabbed Panes
http://java.sun.com/docs/books/tutorial/uiswing/components/tabbedpane.html
17. SWT: The Standard Widget Toolkit
http://www.eclipse.org/swt/