Reverse-Engineering User Interfaces to Facilitate Porting to and across Mobile Devices and Platforms

Eeshan Shah Eli Tilevich
Dept. of Computer Science
Virginia Tech
{eeshan9,tilevich}@vt.edu

ABSTRACT

As mobile devices are rapidly replacing desktop computers for a growing number of users, existing user interfaces often need to be ported from the desktop to a mobile device. In addition, successful user interfaces written for one mobile platform are commonly ported to other mobile platforms. Traditionally, porting user interfaces requires that their source code be reverse-engineered and translated, which is difficult and error-prone. In this paper, we present an approach that reverse-engineers user interfaces without having to analyze their source code. Specifically, our approach examines an interface's runtime representation by means of aspect-oriented programming (AOP). An aspect intercepts the program's control flow at the point when all the components of an interface are laid out on the screen, but before the interface is displayed. The aspect analyzes the interface's in-memory representation and extracts a platform-independent model that can then be used to generate equivalent interfaces for other devices and platforms. Our initial proof of concept ports Java Swing interfaces to Android. In this paper, we describe our approach, discuss its main technical challenges, and outline future research directions.

Keywords

Reverse Engineering, Porting, GUI Models, Mobile Devices and Platforms

1. INTRODUCTION

Garner Inc. predicts that by 2013 mobile devices will overtake personal computers (PCs) as the most common means for accessing the Web worldwide [4]. To accommodate this massive transition to mobile computing, existing desktop graphical applications will be ported to run on mobile devices, including smartphones, tablets, and e-readers. In addition, because the marketplace of mobile devices is highly volatile, mobile application vendors often need to port a successful mobile application from one platform to another. Because the majority of modern applications have sophisticated graphical user interfaces (GUIs), the issue of porting GUIs across platforms has come to the forefront of software evolution.

Modern user interfaces are sophisticated and complex, but most of them comprise a standard set of visual widgets such as push buttons, edit areas, list boxes, and radio buttons. To help the programmer build a GUI, major programming environments provide GUI frameworks that encapsulate the appearance and behavior of standard visual widgets as software components. Using a GUI framework, the programmer can seamlessly build a complex GUI. Even though GUI frameworks help express mostly the same functionality, they differ vastly in their design and implementation. Specifically, modern GUI frameworks differ in terms of their implementation languages (e.g., Java, Objective-C, etc.) and design. For example, while emerging frameworks tend to be declarative (e.g., the Android SDK), some of the existing ones are procedural (e.g., Java Swing). All these differences complicate the porting of GUIs across platforms.

In addition, porting GUIs to and across mobile devices and platforms presents additional difficulties. Mobile devices differ in their screen size and input/output facilities. As a result, a GUI may need to be adjusted significantly when ported to run on a different device. For example, software vendors often provide two separate versions of an application for iPhone and iPad to accommodate the differences in screen size and the presence of special hardware (e.g., GPS receiver). All in all, when porting a GUI across mobile platforms, directly mapping the widgets of one GUI to another is likely to be insufficient to achieve the requisite levels of usability.

Traditionally, porting a GUI between platforms has required reverse-engineering its source code, which is difficult and error-prone. The source code controlling the appearance and behavior of a modern GUI is large and complex, written according to the conventions of a given GUI framework and its API. The source code is likely to include conditional logic that lays out the GUI differently based on various factors such as the type of data to be displayed or how the user interacted with the GUI previously. All these complications make it impractical to reverse-engineer source code when porting GUIs between frameworks and platforms. A recent approach has proposed using API wrappers to migrate GUIs between Java Swing and SWT GUI frameworks [3]. This approach would be inapplicable, for example, when porting a GUI from Swing to Android; in Swing, widgets are expressed as Java components, while in swing, widgets are expressed in an XML file.

In this paper, we present an approach to reverse-engineering GUIs that does not require the programmer to analyze the GUI source code. Our approach extracts a general GUI model from an application's in-memory representation at runtime. We combine the power of Aspect-Oriented Programming (AOP) and reflection, facilities available for the majority of modern languages. In particular, we use AOP to intercept the application's control flow at the point when the GUI is about to be displayed on the screen. At this point, the GUI must be completely laid out on the screen, containing all the visual widgets placed at the designated screen coordinates. Then, our approach leverages reflection to examine the in-memory representation of the main GUI object and extract a general model that describes the contained widgets and their properties (e.g., size, color, coordinates, etc.). The model is persisted in XML and can be used to port the GUI to another GUI framework, platform, or device.

The approach is simple but powerful, as it obviates the need to parse and examine the program's source code. Because the extracted model is represented in XML, it can be used for generating GUIs in any language or platform. The model is generic and can be manipulated to adjust a GUI for the specific requirements of any target platform. In this paper, we describe our initial proof-of-concept that ports Swing applications to Android and outline future research directions.

The rest of this paper is structured as follows. Section 2 discusses the difficulties of engineering GUIs for mobile devices. Section 3 give an overview of our approach. Section 4 describes our proof-of-concept that ports Swing GUIs to Android. Section 5 compares our approach with related state of the art. Section 6 outlines future work directions.

2. GUI ENGINEERING IN THE MOBILE AGE

The market of mobile devices is highly diverse. Vendors offer devices with varied sets of hardware capabilities and features. In addition, different platforms represent GUIs using their unique GUI toolkits, frameworks, and APIs. These and other factors cause the differences in the look-and-feel of mobile GUIs.

Screen real estate is significantly smaller on phones and tablets than it is on desktops. In addition, users interact with desktop applications differently than they do with portable devices. The mouse and keyboard remain the primary input facilities for interacting with modern desktop applications. By contrast, touch interfaces have become standard for portable devices, including smartphones, tablets, and e-readers.

It is these differences between desktop and mobile platforms that make it impossible to directly map a desktop GUI to a mobile one. Even a GUI that works well for one mobile platform may not be directly portable to another mobile platform. In fact, applications running on both a smartphone and a tablet often have different GUIs to accommodate for different screen sizes and available input facilities.

3. APPROACH OVERVIEW

Our approach entails raising the level of abstraction when porting GUIs. That is, our goal is to manipulate a GUI model whose abstraction level is higher than that of the underlying source code. In the long run, we aim at applying the power of model driven architectures [7] to GUI reverse-engineering and porting. In addition, we aim at completely bypassing source code when analyzing the original GUI. Parsing source code is an unreliable mechanism to determine the interface's layout. When analyzing a GUI based on its source code, the analyzer must take into consideration the program's control flow, which may depend on the program's input or the persisted state of the GUI's previous invocation. In addition, such an analyzer would have to be made aware of the API exposed by the GUI framework in place. These APIs follow different design and coding conventions, further complicating the process of parsing the source code to extract a GUI model.

By contrast, our approach examines the in-memory representations of GUI objects at runtime. It relies on two mainstays of modern programming technologies: AOP and reflection. AOP provides a systematic means to interpose new code within existing software components. Reflection makes it possible to examine object structures at runtime. We use AOP to intercept the program's control flow at the point at which all the GUI components are laid out on the screen, but before they are displayed. For example, Java Swing provides method `setVisible` to display a GUI object. An aspect intercepts the program's flow before `setVisible` is invoked.

Modern GUI's form a tree-like structure, in which the tree's root is the top level window. All the other windows are reachable from the root. Our approach uses reflection to exhaustively walk the in-memory GUI tree and record all the contained window objects. If some GUI action (e.g., push button press) creates a new tree-structured hierarchy (e.g., a new dialog box), its display point is similarly intercepted using AOP and examined through reflection. The exhaustive walk of the GUI tree-structured hierarchy is a recursive algorithm that examines each visited visual components (e.g., a window) for its shape, size, color, etc. The recursive step is repeated for each contained window until the leaf roots are reached.

The visited window objects are recorded and distilled into a general model. The model is persisted to stable storage, so that later it can be translated into the GUI source code for other platforms. As our intention is to keep the model generic and language independent, we store the model in XML. The exact model's content and format are still being determined. Specifically, our design goal is to create the general model that can naturally support both the initial extraction and subsequent translation to working GUI source code.

Because the model is extracted dynamically, our approach is limited when a given GUI invocation does not utilize all its components. For example, if some push button is only displayed when the input date has certain properties, our approach may not be able to extract this button. It becomes the responsibility of the programmer to invoke the GUI in such a way, so that all the components are utilized. The process may take several invocations, the results of which can be integrated into a single model.

4. PROOF-OF-CONCEPT

We have started this project by exploring how Swing GUIs can be automatically translated to run on Android devices. Figure 1 summarizes our approach. Our automated tool, "Androider," (∼1.5K LOC) intercepts the creation of binary Swing GUI objects by means of AspectJ and Java Reflection.

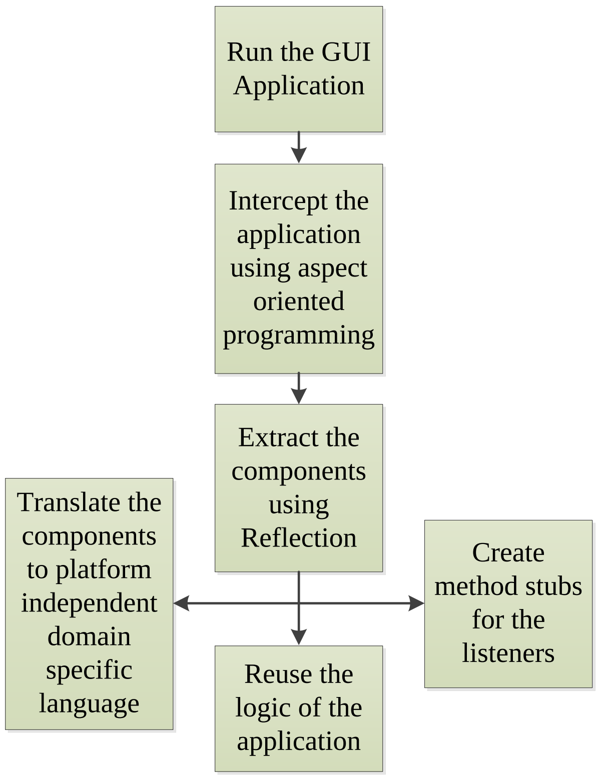
Run the GUI Application
Intercept the application using aspect oriented programming
Extract the components using Reflection
Translate the components to platform independent domain specific language
Create method stubs for the listeners
Reuse the logic of the application

Figure 1: Our approach

Figure 2 shows an AspectJ code excerpt used by Androider to intercept and examine GUI objects at runtime. Androider dynamically analyzes the GUI Swing components to create a generalized GUI model. This model can then be used to automatically generate GUIs for any platform, stationary or mobile. Because well-designed GUIs follow the Model-View-Controller architecture, Androider leverages this design facet to implement a pragmatic translation strategy. Specifically, Androider completely translates the View, partially translates the Controller, and leaves the Model intact. Therefore, any Java-based GUI framework can reuse the original Swing Model as is and need to only adapt the Controller logic for the conventions of the target framework.

We have successfully applied Androider to re-engineer third-party Swing GUIs to run on an Android device. Figure 3 is a screenshot of a calculator application running on a laptop computer, while Figure 4 is a screenshot of the same application running on an Android emulator. Androider automatically adapted this application's GUI, eliminating the need for the programmer to write any code to implement visual components. Androider also generated skeletal implementations for all the required Controllers, so that the programmer only had to fill in low-level event handling logic, which would have been impossible to generate automatically. Finally, the original Model components worked seamlessly with the automatically generated visual components.

We faced several challenging issues while researching this approach. Android and Java use different layouts to display components and widgets on screen. Therefore, a direct mapping of components and their corresponding layouts was impossible. We used built-in Android layouts to create specialized layouts that were equivalent but not the same as built-in Java layouts, and then used these specialized layouts to arrange components and widgets on the screen. Creating these layouts simplified the porting process.

Another hurdle we had to overcome was computing the correct application size and that of its individual components. When generating an Android application from the model, the components' size had to be adjusted, so that the translated GUI would look similar to the original GUI. We also had to insert logic to eliminate the components that did not make sense for mobile platforms like Menu Bars, when translating the model. Reflection will extract all the components that are part of the GUI. Thus, using this approach, we will be able to capture the components that are not visible on the screen and include them in the model representation of the GUI.

As a demonstration, consider a simple Java Swing application in Figure 5. This application how three push buttons, of which the button with the text "Hidden" is not visible. However the button may become visible in response to a future user interaction. As a result, our approach must extract all the buttons when translating this application's GUI to run on a different platform. Figure 6 shows the extracted components that form a tree-shaped structure, with the main frame serving as its root. Each extracted component includes its size and color information. This extracted tree serves as a model of the translated GUI. This model can later be generalized and persisted in any format.

```

public aspect ExtractGUIModel {

/**
 * Intercepts the java.awt.Component.setVisible() method
 */
pointcut setVisible(): call(
            void java.awt.Component.setVisible(boolean));

/**
 * Calls androider methods
 */
after(): setVisible() {
 //obtain the Swing component that is about to be set visible
 Component c = ((Component)thisJoinPoint.getTarget());

 Androider a = new Androider(c, fileName);
 //traverses c using reflection and extracts GUI model
 a.extractGUIModel();
}
}
```

Figure 2: Aspect Code to Intercept and Examine GUI at Runtime.

5. RELATED WORK

UIML is an XML based language for generating UI models [1, 2]. Programmers write a model user interface in UIML and then use UIML bindings to generate the user interface in the desired language. This requires the programmer to

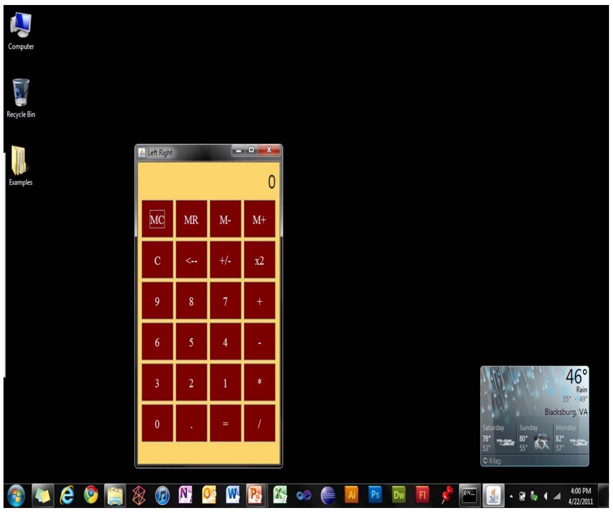

Figure 3: A third-party calculator written in Java running on a laptop computer

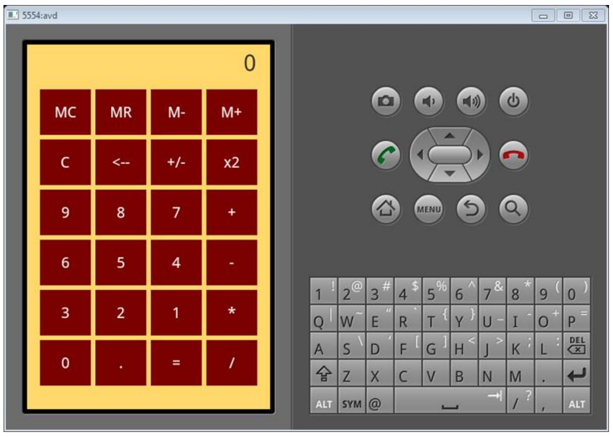

Figure 4: The ported calculator application running on Android emulator

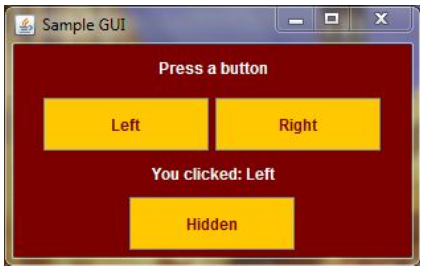

Figure 5: Sample Java Swing GUI

know the UIML language. It has 5 main parts in which the programmer defines the components that make the interface, their arrangement, the data associated with them and their appearance. We are exploring whether UIML may serve as the general model for our approach. An important issue that remains to be determined is whether UIML has adequate support to express, manipulate, and translate modern mobile GUIs.

In another approach, Java Swing is converted to AJAX enabled web based applications [6]. In this approach, Swing GUI is reverse-engineered and automatically translated to AJAX enabled front-end. UiBuilder is another tool used for porting applications across platforms [5], and it is also XML based. The language consists of interactor and constraint tags to define the elements and their position on the screen. At runtime, the XML is translated to the target platform language using a translation library built for this purpose.

In some of these approaches, the programmer has to learn a new language, while other creating wrapper classes create wrapper classes to port applications to use a different GUI framework [3]. Our approach differs in that we reverse-engineer GUIs at runtime by examining in-memory representations of GUI objects. The extracted general GUI models can then be used to generate equivalent GUIs for other platforms. To take advantage of our approach, the programmer does not have to learn a new language. Furthermore, our approach completely bypasses the source code in the process of reverse-engineering a GUI, which is likely to reduce errors when extracting general GUI models.

6. FUTURE WORK

From here on now, we plan to apply what we have learned from our preliminary work to our future research efforts. We plan to investigate how our GUI model, generated while porting Java applications to Android, can be fully generalized, so that it could express GUIs for all major mobile platforms. One way to support this generalization is to create a library of all the supported platforms and devices that run on them. In this library, each entry will contain information such as the type of the device (e.g., smartphone, tablet, e-reader, etc.), the operating system it is running on, its screen size, and its implementation languages. The library can then guide the translation, taking into consideration individual device characteristics and differences. Our vision is to have one general GUI model, which will be extracted and persisted for future use. The model will then be translated into working code as guided by the library.

Specifically, when translating a general model to a particular platform, some components and widgets, not supported by the platform may need to be replaced with equivalent widgets supported on the target platform. For example, a combo box can be replaced with a list of radio buttons and vice versa; a table can be replaced with a simple list formatted as a table. This translation technology should enable new types of mobile applications.

As an example, consider the domain of travel applications. Almost every big city in the US has mobile applications developed to support its visitors. Usually, such applications incorporate a variety of information including maps, landmarks, hotels, restaurants, and entertainment. When an application has to provide so much information to the user, the GUI may be cluttered. We envision being able to mash up such travel applications on the fly, installing only the information needed for a given visitor. All the available information will be stored in a general repository, with the GUI represented as our general model. The required pieces of this model can be translated for the mobile platform at hand and installed on the visitor's device. In essence, in-

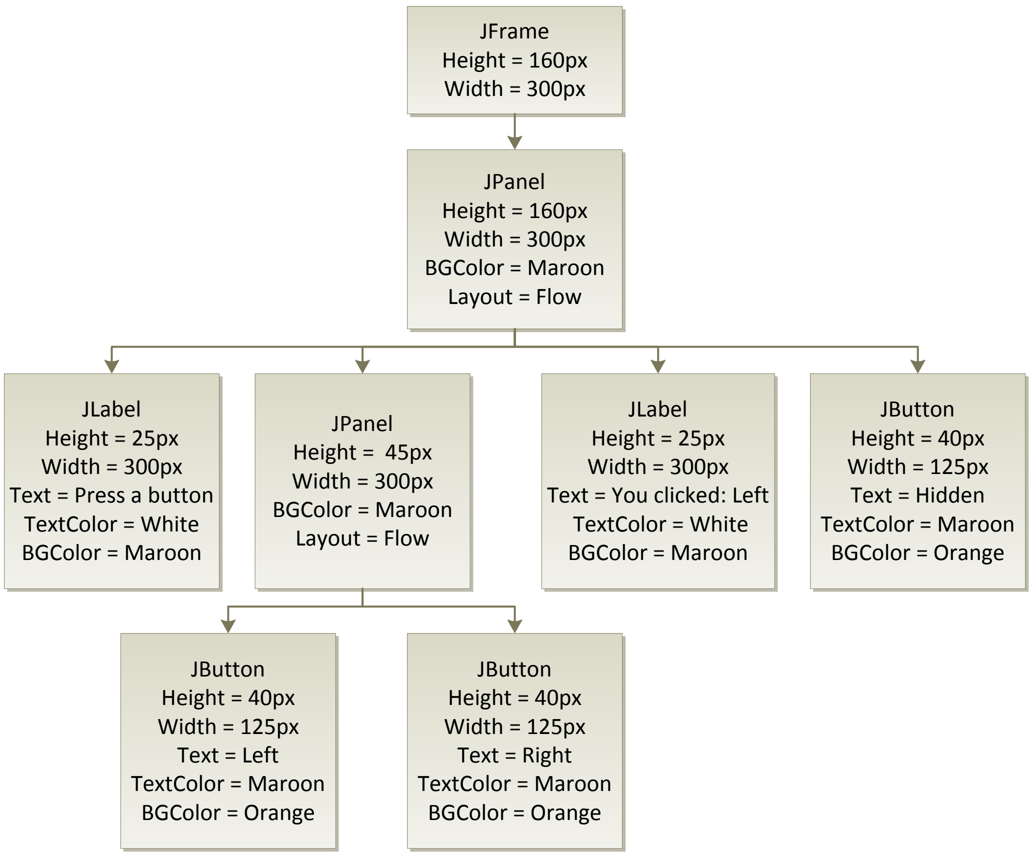

Figure 6: The GUI Tree of Extracted Sample App.'s Components

dividualized mobile applications can be created on the fly for specific users. This is an ambitious vision, but our technology for reverse-engineering GUIs is the first step toward realizing this vision.

7. REFERENCES

[1] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: an appliance-independent XML user interface language. *Computer Networks*, 31(11-16):1695–1708, 1999.

[2] M. F. Ali and M. Abrams. Simplifying construction of multi-platform user interfaces using UIML. In *UIML Europe 2001 Conference*, 2001.

[3] T. Bartolomei, K. Czarnecki, and R. Lammel. Swing to SWT and back: Patterns for API migration by wrapping. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE.

[4] Gartner Inc. Gartner highlights key predictions for IT organizations and users in 2010 and beyond. `http://gartner.com/it/page.jsp?id=1278413`, 2010.

[5] K. Luyten, B. Creemers, and K. Coninx. Multi-device layout management for mobile computing devices. Technical report, Limburgs Universitair Centrum, 2003.

[6] H. Samir, E. Stroulia, and A. Kamel. Swing2Script: Migration of Java-Swing applications to AJAX Web applications. *Reverse Engineering, Working Conference on*, pages 179–188, 2007.

[7] J. Vanderdonckt. A MDA-compliant environment for developing user interfaces of information systems. In *Advanced Information Systems Engineering*, pages 16–31. Springer, 2005.