Dynamic User Interface Generation: Extended Abstract

Svetoslav Ganov
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712 USA
svetoslavganov@mail.utexas.edu

Enos Jones
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712 USA
enos.jones@mail.utexas.edu

Angela Dalton
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712 USA
adalton@mail.utexas.edu

Dewayne E Perry
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712 USA
perry@ece.utexas.edu

CHI 2008, April 5 – April 10, 2008, Florence, Italy
ACM 1-xxxxxxxxxxxxxxxxx.

Abstract

Advances in hardware technologies facilitated the advent of a wide variety of portable devices—laptops, pocket PCs, PDAs, smart phones—each of which has a different user interface and computational capability. This diversity of devices makes production of cross platform applications challenging. While the Java programming language enables platform independent application development, differences in user interfaces of heterogeneous devices may require specialized approaches. This is often expensive, time consuming, and error prone. In this paper, we develop a generic approach for Java applications with rich user interfaces designated for multiple platforms. We introduce an abstract user interface (AUI) library used in the software development process. We also define a mapping, using XML files, between this abstraction and a concrete user interface library used on the target device. When an application is started on a particular platform we build the concrete user interface by instrumenting the Java bytecode on-the-fly using metadata from the XML files. Our approach eliminates the need for source code modifications and recompilation, thus increasing the speed and reducing the cost of the software development. The overhead for defining a user interface library mapping is amortized over multiple applications.

Keywords
User interface, xml schema, bytecode instrumentation, abstract interface, java virtual machine, ubiquitous computing interface

ACM Classification Keywords
H.5.2 User Interfaces - *Theory and methods, Graphical user interfaces (GUI), Natural language, Prototyping, Voice I/O*

Introduction

Advances in hardware technologies facilitated the advent of a wide variety of portable devices—laptops, pocket PCs, PDAs, smart phones—each of which has a different user interface (UI) and computational capability. The number and diversity of these devices is constantly increasing which results in the development of cross platform applications becoming increasingly challenging.

Software companies aim to achieve fast prototyping and development while minimizing their expenses. Offering a separate product for each target device has a high cost, is time consuming, and is error prone. The software industry faces the problem of developing applications for heterogeneous devices with different UIs, often conceptually disparate. For example, the same application may be run on a laptop using a graphical user interface (GUI) and on a pocket PC using a natural speech user interface (NSUI).

The automatic generation of device specific user interfaces does not have a widely accepted solution, even though it is a subject of intensive research[1]. Simply by looking at the variety of available interfaces the complexity of the problem is apparent. Device interfaces can include displays, keyboards, touch screens, natural speech, and others. Displays range from 96 to 192 dots per inch with areas of a few square inches to multiple square feet. The problem is compounded by the fact that these devices can employ a variety of interface libraries for implementing the same type of user interface.

We present a generic approach for the development of Java applications with rich user interfaces designated for heterogeneous devices. We introduce an abstract user interface (AUI) library used in the software development process. We also define, in XML files, a mapping between this abstraction and a concrete user interface library used on the target device. When an application is started on a particular platform we build the concrete user interface by instrumenting its Java bytecode on-the-fly using metadata from the XML files. The XML files can reside on the device or be available remotely through a look-up service. Our approach eliminates the need for source code modifications, increasing the speed, reducing the cost, and increasing the reliability of the software development. In our approach the work required for defining the UI library mappings is amortized over multiple applications and should be provided by the vendor that supplies the AUI library. Hence, from software developer's point of view, adopting our approach does not add more complexity or consequential manipulations.

The rich ensemble of computational devices in our personal environment employ different user interfaces, while providing enough computational power to run the Java runtime environment. Our approach takes advantage of the common runtime allowing a program to be executed on multiple devices with diverse user interfaces. Application development is not hampered by issues arising from the diverse Human-Computer

Interaction (HCI) models engendered by multiple user interfaces required for the application.

Our Approach

The main goal of this project is to create a general mechanism for building the user interface of Java applications targeted for heterogeneous devices. We provide software developers with appropriate abstractions—universal enough to be mapped to diverse types of UIs and intuitive enough to allow almost effortless adoption. The variety of potential host devices is limited only by the device's ability to run a Java Virtual Machine (JVM). This widens the scope of potential platforms to laptops, pocket PCs, PDAs, and smart phones, each of which runs either a J2ME [2] or a J2SE [3] environment.

The key idea behind our work is the development of an abstract user interface (AUI) layer, implemented as a Java library, which can be mapped to a variety of concrete UI libraries. Our AUI library defines widgets used in the software development process. The AUI widgets are mapped later to concrete widgets of the UI library for the target device.

The process of dynamic UI construction in a platform dependent way is accomplished on-the-fly through Java bytecode instrumentation. We use XML files to define the semantics for mapping at the bytecode level from the AUI library to a concrete UI library. The UI generation is performed just before a class of the target application developed with our AUI library is loaded into the JVM. From the execution point of view, the application loaded in the JVM is the same as if it was written and compiled with the corresponding concrete UI library. The bytecode of an application written with the AUI library is instrumented on-the-fly to match the bytecode of the same application written with a concrete UI library. This is accomplished by providing our system with the rules for mapping from the AUI to a concrete UI library.

The AUI Library consists of widgets that are at a higher level of abstraction and offer a wider range of properties than those of a standard UI library to facilitate mapping to conceptually different UI libraries. We have developed a set of abstract widgets and used these widgets to manually rewrite an example application as if it had been initially developed with the AUI library. We have developed only the widgets required for implementing the example application, with the goal of exploring the applicability of our approach. Currently the AUI library consists of the following widgets: Action, Conversation, Group, Label, MultipleChoice, Text, and Widget, the super class for all widgets.

System Architecture

Our approach to user interface generation on-the-fly via Java bytecode instrumentation imposes the need for a runtime environment that performs the bytecode manipulations. We provide a runtime environment called BUILD (Builder of User Interfaces for Local Devices) that is responsible for running and instrumenting the bytecode of a launched application. BUILD provides a layer between the Java Virtual Machine and the application developed with the AUI library, allowing us to make bytecode manipulations while loading the application classes.

The architecture of BUILD consists of several interconnected modules organized in a loosely coupled fashion. Figure 1 presents a schematic representation

of the system architecture including the resources consumed or modified by the system.

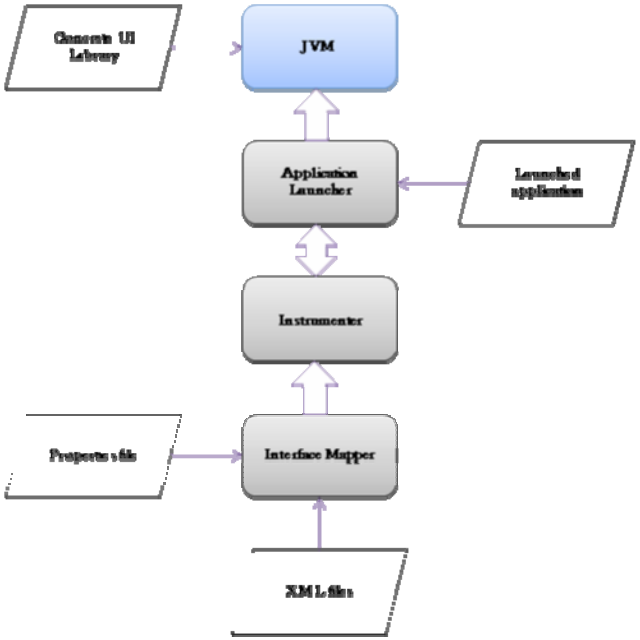

Figure 1. Software architecture

The main modules of BUILD are the *Application Launcher*, the *Instrumenter*, and the *Interface Mapper*.

The *Application Launcher* is responsible for replacing the application class loader with a custom one, initializing the system, and starting the target application. We replace the application class loader with a custom loader to satisfy two requirements:

- Accessing the class definition before it is loaded into the JVM.
- Instrumenting the appropriate classes of the target application.

Satisfying the first requirement allows us to modify the class definition before loading the class into the JVM. Satisfying the second requirement guarantees that all classes of the launched application are consistent when passed through the bytecode manipulation routine. Once the custom class loader succeeds to load a class definition it passes the class definition to the bytecode manipulation routine, and then loads the potentially modified class into the JVM. After the application class loader has been replaced, the *Application Launcher* instantiates and initializes the *Instrumenter*. Finally, the *Application Launcher* invokes the main class of the launched application via reflection.

The *Instrumenter* is responsible for the actual manipulation of the Java bytecode. It implements a simple interface with a single method that takes as a parameter a class definition and returns a modified version of the class if manipulation was necessary. The interface between the *Application Launcher* and the *Instrumenter* allows the user to provide a custom implementation of the *Instrumenter*. The *Instrumenter* instantiates and initializes the *Interface Mapper*.

The *Interface Mapper* is responsible for guiding the *Instrumenter* during the class manipulation phase. It implements a simple interface defining the contract with the *Instrumenter*. This implementation allows a user provided *Interface Mapper* to be used. The *Interface Mapper* reads the interface mapping definitions from XML files and builds appropriate data structures allowing efficient lookup. During the initialization phase of the *Interface Mapper,* a properties file determines which XML files should be

parsed, containing the proper interface mapping definitions.

The resources required or modified by BUILD are the launched application, the interface mapper properties file, the XML files defining the interface mappings, and the concrete user interface library to which a launched application is to be adapted.

The *Launched Application* is the compiled bytecode of an application developed with the AUI library provided by our framework. Note that we do not require the source code of the launched application since all manipulations are performed at the bytecode level.

As explained in the previous paragraphs the *Interface Mapper* parses a properties file that defines which of the XML files have the mapping definitions that correspond to a particular concrete UI library.

There are three types of XML files that define the user interface mapping and store metadata used by the *Instrumenter* during the class manipulation phase. The mapping files types are:

- Mapping from class to class
- Mapping from method signatures to method signatures
- Mapping from method calls to method calls

The concrete UI library is the library to which a launched application is to be adapted. Even though this resource is not directly required by BUILD, it is presented in the schema to illustrate the need for providing the concrete user interface library because after the instrumentation phase the launched application depends on the concrete library.

Example

In this section we demonstrate how a real life application written with our AUI library and executed with BUILD could be presented with different user interfaces. Since there is no real life application developed with the AUI library, we identified an application written with a concrete UI library and rewrote its source code as if it had been developed with the AUI library.

Our example is the *Workout Generator* application. It is written in Java and interacts with the user through a GUI implemented with the SWT library[18]. The program takes as input a user's biometric characteristics consisting of gender, height, weight, age, metabolism, and experience level. On the basis of this input data the application generates a suitable weekly workout program. Figure 2 shows a screenshot of the original version of the *Workout Generator*.

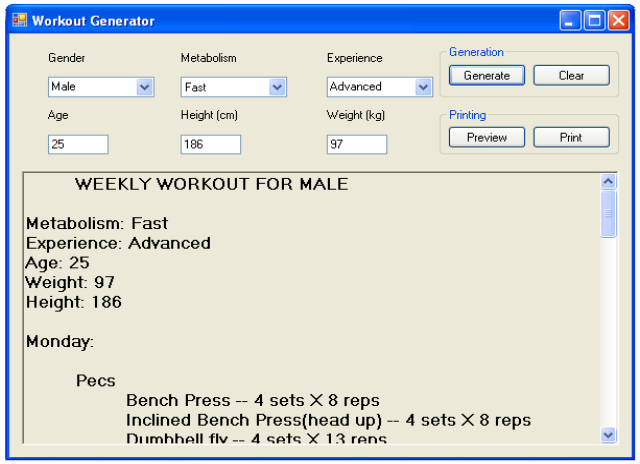

Figure 2. Screenshot of the workout generator

Rewriting the source code of the example application with the AUI library was straightforward. After successfully rewriting the *Workout Generator* we examined the obtained source code to confirm that the use of our AUI library did not add complexity.

The modified application written with the AUI library was successfully executed using two different mappings—one to *SWT* and one to a natural speech library called *BaradVoice*. As a result, the same application is presented users with two conceptually different interfaces—a graphical user interface and a natural speech user interface. The GUI version has exactly the same appearance as the original version presented on Figure 8 and the NSUI version performs consecutive interaction with the user traversing all input widgets first and then providing a choice of the possible actions.

Conclusion

In this paper we presented our approach to automatic user interface generation for heterogeneous devices via Java bytecode instrumentation. We have developed an abstract library that is used during the software development process. We have also defined the mapping from the abstract library to two concrete user interface libraries—SWT and BaradVoice. Our technique uses metadata for the interface mapping during the application loading phase to instrument its bytecode for generating the appropriate user interface. We demonstrated the feasibility of our approach with an example application.

Providing the ability to write applications that will execute effectively on a variety of devices with heterogeneous user interfaces is important as users increasingly adopt new technologies. The method we presented reduces the burden on software developers to rapidly release versions of applications for multiple target devices and benefits users by allowing them to run their applications on any device.

References

[1] Anwar, Z., Al-Muhtadi, J., Yurcik, W., and Campbell, R. Plethora: A Framework for Converting Graphic Applications to Run in a Ubiquitous Environment , Mobile and Ubiquitous Systems: Networking and Services, 2005

[2] Java ME at a Glance, 2007. Retrieved October 15, 2007 from Sun Developer Network(SDN): http://java.sun.com/javame/index.jsp

[3] Java SE at a Glance, 2007. Retrieved October 15, 2007 from Sun Developer Network(SDN): http://java.sun.com/javame/index.jsp