Accessibility Validation with RAVEN

Barry Feigenbaum, Ph. D.
IBM
11501 Burnet Road
Austin, TX 78758
001-512-838-4763
feigenba@us.ibm.com

Michael Squillace, Ph. D.
IBM
11501 Burnet Road
Austin, TX 78759
001- 512-823-7423
masquill@us.ibm.com

ABSTRACT
Testing is, for most, a necessary evil in the software life cycle. One very important form of testing is the evaluation of software products according to mandated criteria or guidelines such as those that specify level of accessibility. Such evaluations can be quite tedious, especially if they must be done manually and applied consistently to each and every component of an application. The use of assistive technologies like screen readers to demonstrate the compliance of a software product to a set of regulations is time-consuming, error-prone, and expensive. Validation tools that can perform such evaluations are becoming more popular as integrated development environments become more sophisticated but, in the area of accessibility validation, they are sorely lacking if not nonexistent. This paper introduces the IBM Rule-based Accessibility Validation Environment, an Eclipse-based tool for inspecting and validating Java rich-client GUIs for accessibility using non-invasive, semi- to fully-automatic, rule-based validation and inspection.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Code inspections and walk-throughs, Debugging aids*, *Diagnostics*, *Distributed debugging, Dumps*, *Error handling and recovery*, *Monitors*, *Symbolic execution*, *Testing tools (e.g., data generators, coverage testing), Tracing*.

General Terms: Measurement, Verification.

Keywords
Accessibility, Java, Rich-Client, GUI, Reflection, AOP.

1. INTRODUCTION
Testing is, for most, a necessary evil in the software life cycle but it is critical if a software product is to be successful. One very important form of testing is the evaluation of software products according to mandated criteria or guidelines. Such evaluations can be quite tedious, especially if they must be done manually and applied consistently to each and every component of an application. One need only think of testing an application for compliance with internationalization standards to realize the tedious and monotonous nature of such evaluations.

WOSQ'06, May 21, 2006, Shanghai, China.

Accessibility testing falls into this category as well. The use of assistive technologies like screen readers to demonstrate the compliance of a software product to a set of regulations is time-consuming, error-prone, and expensive. Validation tools that can perform such evaluations are becoming more popular as integrated development environments become more sophisticated but, in the area of accessibility validation, they are sorely lacking if not nonexistent.

This paper introduces the IBM Rule-based Accessibility Validation Environment, an Eclipse-based [1] tool for inspecting and validating Java rich-client GUIs for accessibility. However, it also introduces an approach to building non-invasive, semi- to fully-automatic, rule-based validation and inspection tools for software products.

2. VALIDATING GUI ACCESSIBILITY
Put simply, the *accessibility* of a software application is the degree to which its features and functionality can be accessed by users of that application without regard to the abilities or disabilities of these users. The graphical user interface (GUI) is the most common way in which users interact with software applications and is, at the same time, the most troublesome component of an application when evaluating it for accessibility. Insuring the accessibility of GUI-based applications often involves extra effort on the part of software engineers because they must take advantage of special APIs of a runtime platform for which the application was designed. Such APIs provide ways to export information about basic properties and state changes of GUI components to assistive technologies, notification about events fired within these GUI components, and device-independent access to the GUIs of an application. For example, the Java Swing GUI toolkit [2] provides the *Accessibility API* [3] in order to expose basic properties of Swing components, events fired by these components, and state changes within the GUI to assistive technologies such as screen readers. The software developer must be familiar with and add code to take advantage of the Accessibility API, however, if a Java Swing application is to be deemed accessible. As in the case of internationalization, it is most cost-effective, then, to design accessibility into the application from the outset rather than to catch inaccessible portions of the application during testing or after production and then amending, rebuilding, and redistributing the application.

It is often difficult to evaluate the accessibility of an application. Typically, sets of guidelines or checklists are specified by an institution, organization, or government that mandates that applications used by members, employees, or consumers be accessible. Examples of such checklists include the *IBM Java Accessibility Checklist* [4] for Java Swing applications and the W3C's *Web Content Accessibility Guidelines* [5] for evaluating the accessibility of web content. Adopting these guidelines in developing a particular application does not guarantee the accessibility of that application, but it does insure that the application exhibits a high degree of accessibility – it is likely more accessible than not.

Traditionally, there have been three ways in which to determine the accessibility of GUI-based applications where accessibility is specified in terms of the sorts of checklists just noted. First, an assistive technology (AT) such as a screen reader (e.g. Freedom Scientific's *Jaws for Windows* [6]) can be executed while the application to be validated is running. Developers and testers can verify the accessibility of a GUI based upon the information about the GUI reported by the screen reader (e.g. the descriptions for GUI controls it presents, the keyboard shortcuts it recognizes) as they navigate among the GUI components. This can be a tedious task since it requires the developer and/or tester to navigate to each and every component of the GUI and verify that all relevant information about that component is being rendered by the screen reader. Notice, too, that this method of evaluation can only find accessibility violations during the test cycle and, hence, will always dictate code modifications and regression testing.

A second way in which to validate such applications is to use inspection tools such as Sun Microsystems' *Java Ferret* [7]. These tools are configured to run in the Java Virtual Machine (JVM) as if they were ATs and typically report the values of predetermined sets of properties of GUI components. The number of properties that are reported by such tools is usually only a subset of those that are referenced or required by ATs and other properties of the GUI that might hinder accessibility (e.g. the relation of components to one another or the structure of the GUI hierarchy) cannot be considered by such tools. Also, once again, the use of such tools is a distinct and additional task in the test cycle.

Finally, there are validation tools that depend upon the original source code to affirm the accessibility of GUI-based applications. Parasoft's *Web King* [8] analyzes the text of an HTML document to determine deficiencies in accessibility of web pages based on the syntax and structure of the document. These tools, however, cannot adequately evaluate GUI-based applications precisely because they are not evaluating the GUI at runtime, which is precisely when persons using ATs interact with the GUI. What is desired, then, is a noninvasive (i.e. source-code independent), dynamic, semi- to fully-automatic accessibility validation tool that permits a wide variety of validation rules to be specified and that applies these rules to runtime objects.

3. VALIDATION WITH RAVEN

3.1 Concept

The *IBM Rule-based Accessibility Validation Environment* (RAVEN) [9] overcomes many of the deficiencies just enumerated. Distributed as a set of Eclipse plug-ins, RAVEN offers several new perspectives and views that can be utilized along side typical Java- or plug-in development-related perspectives in the Eclipse workbench. A Java (or JVM-based) application can be launched by RAVEN and validation reports are generated while the application is being used. These reports describe the nature of the violation with respect to the validation rules being used and, in some modes, indicate the point at which the troublesome GUI component is instantiated, thus suggesting where violations might be remedied. In this way, accessibility validation is dynamic and somewhat automatic, requiring very little human intervention to produce validation reports.

Figure 1 depicts a portion of a view in the RAVEN External Perspective, the perspective used to launch and validate Java rich-client GUI-based applications. The GUI being validated is the *SwingSet2* example [10] packaged with Sun's distribution of the Java Development Kit (JDK). SwingSet2 is a large Swing application exhibiting the basic features and components included in the Swing GUI toolkit. The figure shows the name of the class to be executed, SwingSet2, the classpath for the application, and indicates the method to be used for the launch, SwingSet2.-main(String[]). (Other modes and views in RAVEN permit validation to be initiated from constructors and other non-static methods of a Java class.)

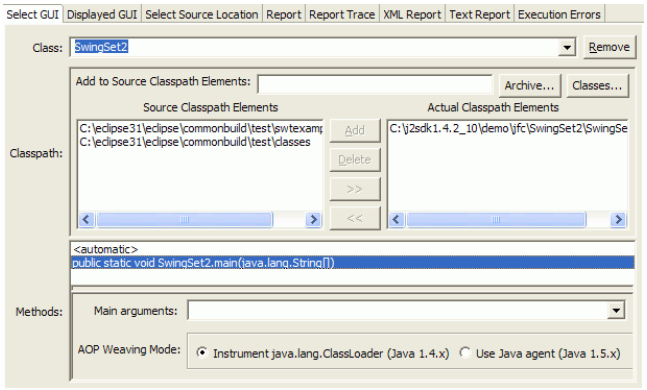

Figure 1: Depicts configuration of the Java GUI View in the RAVEN External Perspective for launching/validating SwingSet2

Figure 2 shows the validation report generated upon the SwingSet2 application being launched and initially rendered. The two errors indicate direct violations of the IBM Java Accessibility Checklist, whereas the warnings indicate places in which accessibility and usability could be enhanced. It is important to note that the SwingSet2 GUI is running when this report is generated. As we use the GUI, other reports would be generated concerning the components of other panels that are opened.

Applications that themselves launch a JVM can also be launched and validated from the RAVEN External Perspective. For example, an Eclipse installation with the RAVEN plug-ins

installed could be validated by RAVEN itself. As it turns out, the tool is fairly accessible, according to the IBM software accessibility checklist. Surprisingly, this is unlike most other accessibility evaluation tools, which are often not very accessible.

Much of the function of RAVEN is made possible via two technologies: the Java *Reflection API* [11] and aspect-oriented programming (AOP)[12]. Another important ingredient of RAVEN is the architecturally-neutral validation engine, which permits the tool to support the validation of not only Java Swing and Eclipse SWT[13] GUIs, but permits the user to extend the reach of the engine and validate, say, GUIs written using the GNOME architecture[14] or the Eclipse Graphical Editing Framework[15]. We shall describe these technologies in turn and, also, gain a better understanding of how accessibility validation with RAVEN works.

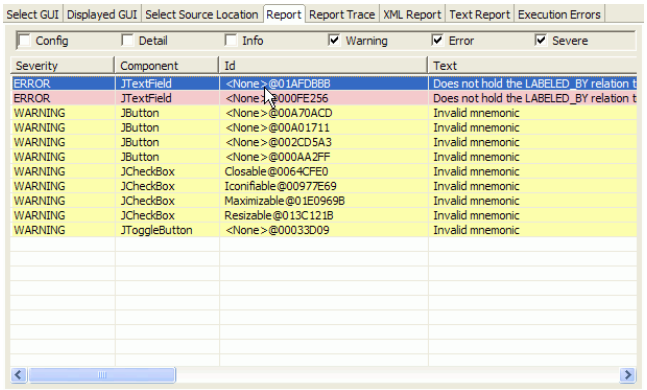

Figure 2 Validation report of the SwingSet2 app

3.2 Reflection and Validation Rules

The Java Reflection API provides a way for applications to dynamically examine Java classes and to determine their data members and methods at runtime. RAVEN uses Java Reflection to process validation rules. These validation rules are given using a simple XML-based [16] markup language specified by RAVEN and used in an XML document known as a *validation document*. This language does not require any Data Type Definition (DTD) [17] or XML schema [18], thus making the validation engine a rather light-weight software component.

The validation engine associates a rule (or set of rules) with a particular Java type. It then walks the GUI hierarchy, applying the appropriate rule or set of rules to each component. Implementations of the `TreeWalker` interface control how a GUI hierarchy (or any hierarchical structure) is traversed, supplying ways to access the parent and children of any given element in the hierarchy.

Because RAVEN is specifically concerned with accessibility, two different `TreeWalker` implementations are used for each GUI framework supported by RAVEN. For example, the following excerpt is from the implementation of the `TreeWalker` interface for walking a standard hierarchy of Swing GUI components:

```
Object[] getChildren(Object element) {
    Object[] children = new Object[0];
    Object bridgedChild =
       getBridgedChild(element);
    if(bridgedChild != null) {
        children =
          new Object[] {bridgedChild};
    }
    else if(element instanceof Container) {
        children =
     ((Container) element).getComponents();
    }
    return children;
}
Object getParent(Object element) {
    Object parent = null;
    Object bridgedParent =
        getBridgedParent(element);
    if(bridgedParent != null) {
        parent = bridgedParent;
    }
    else if(element instanceof Component) {
        parent =
        ((Component) element).getParent();
    }
    return parent;
}
```

An additional implementation of this interface is also available to walk the hierarchy of Swing components as instances of the `javax.accessibility.Accessible` interface provided by the Accessibility API:

```
Object[] getChildren(Object element) {
    Object[] children = new Object[0];
    Object bridgedChild =
        getBridgedChild(element);
    if(bridgedChild != null) {
        children =
        new Object[] {bridgedChild};
    }
    else if(element instanceof Accessible) {
        AccessibleContext accElement =
           ((Accessible)
          element).getAccessibleContext();
        children =
          new Object[accElement.
            getAccessibleChildrenCount()];
```

```
        for (int c = 0;
             c < children.length;
             ++c) {
            children[c] =
         accElement.getAccessibleChild(c);
        }
    }
    return children;
}
Object getParent(Object element) {
    Object parent = null;
    Object bridgedParent =
         getBridgedParent(element);
    if(bridgedParent != null) {
        parent = bridgedParent;
    }
    else if(element instanceof Accessible) {
        parent =
         ((Accessible) element).
            getAccessibleContext().
            getAccessibleParent();
    }
    return parent;
}
```

Notice that both implementations provide for a so-called `bridgeMap`. This map contains keys that are references to parent components within the hierarchy and values that are references to child components. This map is used by RAVEN because GUIs spanning multiple GUI frameworks are supported by the engine. For instance, we might have a Swing GUI embedded in an Eclipse SWT GUI in which case the bridge map would contain a key referencing the parent SWT control and a corresponding value referencing the Swing component.

Currently, validation rules are *type-based* in that they require that a rule be applied to an object of a particular Java type. (Future work may include a way to specify so-called "event-based" rules so that validation of arguments to method invocations, properties of sources of runtime events, and other more dynamic aspects of the GUI can be performed.) This type is specified in the validation document as a child of the `<rib:components>` tag. The child elements of the tag specifying the type are then used to codify the rule to be applied to the object of that type. Reflection is used to test whether an object is an instance of the requisite type and to invoke access methods or check field values of that object in order to apply the validation rule.

Consider the following example from the Java Swing validation document packaged with RAVEN:

```
<Table>
  <accessibleTable>
    <accessibleCaption
      rib:severity="WARNING"
      rib:message="Missing caption"
      rib:enable="true"
    />
    <accessibleSummary
      rib:severity="WARNING"
      rib:message="Missing summary"
      rib:enable="true"
    />
  </accessibleTable>
</Table>
```

This rule states that any object which is an instance of `javax.swing.JTable` must have an accessible caption and an accessible summary. These properties are fetched (and set) via the `JTable` object's `javax.accessibility.AccessibleTable` object, which, in the Java Accessibility API, is used to export the necessary information about tables in the GUI to assistive technologies. Thus, for any `JTable` object `table` in the GUI, the RAVEN validation engine will invoke `table.getAccessibleTable().getAccessibleCaption()` and `table.getAccessibleTable().getAccessibleSummary()`. If either of these values are `null`, the generated validation report will contain a violation at the level of 'WARNING' and with the message given in the validation document. In this way, the reflection engine determines the validity of a validation document, not the XML parser.

Another powerful feature of RAVEN is its ability to process rules that contain code from scripting languages like *Jython*[19]. Here is an example from the same validation document that invokes a pre-defined method in Jython to evaluate the mnemonic for all buttons of a Swing GUI:

```
<AbstractButton>
  <mnemonic
    rib:severity="WARNING"
    rib:message="Invalid mnemonic"
    rib:enable="true"
  >
    isValidVirtualKey(propertyValue)
  </mnemonic>
  :
</AbstractButton>
```

and here is the code for the method:

```
def isValidVirtualKey (keyCode):
  return keyCode in VK_CODES
```

The identifier `VK_CODES` is a list of Java virtual key codes as defined in the `java.awt.event.KeyEvent` class. This rule states that any defined mnemonic must be one of these codes. Currently, RAVEN supports embedding either Jython script or Java code.

3.3 Aspect-Oriented Programming

In object-oriented programming (OOP) [20], a class (or set of classes) embodies a set of related concerns of an application. For instance, the `Customer`, `Account`, and `Transaction` classes in a banking application would attempt to capture deposits, withdrawals, and transfers by customers at a bank. These concerns are similar in function and purpose so that they are grouped together in a set of classes.

Other concerns such as logging, debugging, performance, and, in our case, validation are *cross-cutting* concerns. Such concerns cannot be limited to a class or set of classes and, in fact, permeate or are woven throughout the entire application. These concerns are the realm of aspect-oriented programming (AOP) rather than object-oriented programming.

There are two components of an aspect:

1. The set of execution points in the application in which we are interested. An execution point might include a method invocation, a field access, or the throwing of an exception.
2. The behavior or *advice* that is to be performed before, during, or after the execution point in the application.

We use *pointcut* expressions to describe the set of execution points in an application that we want to aspect and *bindings* to associate advice with the set of execution points. Advice is anything that can be written in Java code.

Bindings are expressed differently in different AOP implementations. Consider the following excerpt from `resources/swing-val-aop.xml` packaged with RAVEN:

```
<bind pointcut="call(public void
$instanceof{java.awt.Component}->show())">
  <advice name="validate"
aspect="com.ibm.wac.rib.core.validate.aspect
.ValidationAspect"/>
</bind>
```

The pointcut expression,

```
call(public void
$instanceof{java.awt.Component}->show())
```

indicates that we wish to intercept any call to the `show()` method by any instance of `java.awt.Component`. Upon this interception, we want to validate the GUI once the component in question is shown. (Similar pointcut expressions exist `in swt-val-aop.xml` to validate Eclipse SWT GUIs upon a call to `org.eclipse.swt.widgets.Shell.open()`.

Because AOP engines provide for either load time or runtime weaving (i.e. the process by which the byte code of Java classes is modified to reflect the bindings), RAVEN is able to facilitate source code-independent or *non-invasive* validation. This is important since it means that RAVEN can assist both software developers during the development process and testers or even end-users who may not have access to source code. Put another way, RAVEN can perform validation on GUI-based applications under development or those that have long since been completed. Non-invasive validation is important, too, because it makes accessibility an integral part of the development and testing process. Using the application and validating it for accessibility occur simultaneously – accessibility testing is no longer an additional step or afterthought in the testing cycle.

Finally, when RAVEN's capacity for non-invasive validation is combined with the power of the Java Reflection API, one additional bonus is received. Although RAVEN is an engine for validating GUI-based applications for accessibility, the validation engine itself is generic. Any rules that can be expressed in the simple markup language specified by RAVEN can be applied to any component of an application. For instance, one could use a binding file and the appropriate validation document to test GUIs for internationalization compliance by checking, say, that the strings passed to the `setText` method of all `Button` objects are legitimate values in a known `ResourceBundle`.

3.4 GUI Architectures

As already noted, there are a number of APIs for building and rendering GUIs in Java. Each of these frameworks dictate, among other things, the ways in which components are added to and removed from other components, the ways in which different parent and child components and the properties of these components can be accessed, the way the GUI is rendered, and a basic hierarchical structure of classes via which the GUI is to be built and traversed.

The conception of an *architecture* in RAVEN is an abstraction of these GUI frameworks. By precisely defining an architecture (by implementing the `Architecture` interface), the RAVEN user can specify many aspects of a desired framework such as how and when components (or properties of components) are to be accessed, the methods for traversing these components, the kind of controls that can serve as top-level or root components, and how these components will be rendered. Moreover, the Architecture implementation can be specified externally, thus allowing new and/or different architectures to be supported without changes to RAVEN.

RAVEN is packaged with support for the Java AWT, Java Swing, and Eclipse SWT GUI frameworks. For example, here is an excerpt of the implementation of the `Architecture` interface for the Swing GUI toolkit:

```
String[] getInitPackages() {
    return PACKAGE_LIST;
}
boolean isLinkable(Object comp) {
    return comp instanceof Component;
}
boolean performsLinkOnCreation() {
    return false;
}
boolean isTopDown() {
    return false;
}
void setComponentID(Object comp, String id)
{
    if (comp instanceof Component) {
        ((Component) comp).setName(id);
    }
}
String getComponentID(Object comp) {
    return comp instanceof Component
      ? ((Component) comp).getName() : null;
}
boolean isUIThread() {
    return SwingUtilities.
       isEventDispatchThread();
}
```

This excerpt will seem familiar to anyone who is a Swing programmer. Notice that some methods, such as `getComponentID`, `setComponentID`, and `isTopDown`, are applicable to any hierarchy (not merely GUI hierarchies) and, hence, live in the `Architecture` interface. Other methods, such as `isUIThread`, specifically target hierarchies of GUI components and, therefore, reside in the `GUIArchitecture` sub-interface. Such a division allows RAVEN to remain a generic validation engine, supporting the validation of any hierarchical structure with any set of rules that can be stipulated in the validation document.

4. CONCLUSION

One of the chief principles upon which RAVEN was developed was that accessibility should be an integral part of the development cycle as well as the test cycle. Put another way, accessibility of a GUI-based application should be built into the system from the outset, much as localization or other facets of usability. RAVEN provides a non-invasive, dynamic inspection and validation of Java rich-client GUIs either during the development process or testing.

RAVEN supports a pluggable architecture model via which other GUI frameworks can be supported by the validation engine. Future work will include supporting other types of GUIs, including those produced by Lotus applications, the Eclipse Visual Editor, the Eclipse Graphical Editing Framework, and web-based rich-client GUIs. Validating web content, especially dynamic web content produced by JavaScript or AJAX-based [21] web applications, could be done by examining the actual Document Object Model (DOM) [22] constructed by the browser rather than the static HTML source, as is currently done by other tools such as Parasoft's *Web King*.

5. REFERENCES

[1] See http://www.eclipse.org.

[2] See http://java.sun.com/j2se/1.4.2/docs/guide/swing/-index.html.

[3] See http://java.sun.com/products/jfc/jaccess-1.3/doc/core-api.html.

[4] See http://www.ibm.com/able/guidelines/java/-accessjava.html.

[5] See http://www.w3.org/WAI/intro/wcag.php.

[6] See http://www.freedomscientific.com/fs_products/-JAWS_HQ.asp.

[7] See http://java.sun.com/developer/technicalArticles/GUI/-accessibility2/index.html.

[8] See http://www.parasoft.com/jsp/products/home.jsp?-product=WebKing.

[9] See http://www.alphaworks.ibm.com/tech/raven.

[10] See http://java.sun.com/products/plugin/1.4/demos/-plugin/jfc/SwingSet2/SwingSet2.html.

[11] Forman, I., Forman, N., *Reflection in Action*, Manning 2004.

[12] See http://www.jboss.org/products/aop.

[13] See http://www.eclipse.org/swt/.

[14] See http://www.gnome.org/.

[15] See http://www.eclipse.org/gef/.

[16] See http://www.w3.org/XML/.

[17] See http://xmlfiles.com/dtd/.

[18] See http://www.w3.org/XML/Schema.

[19] See http://www.jython.org.

[20] See http://www.jboss.org/products/aop.

[21] Crane, D., Pascarello, E., *Ajax in Action*, Manning 2005.

[22] See http://www.w3.org/DOM/.