Speech-Enablement of Eclipse IDE and Eclipse Rich Client Applications Using AspectJ

TV Raman
IBM Research
650 Harry Road
San Jose, CA. 95120
1-408-927-2608
tvraman@almaden.ibm.com

Alison Lee
IBM Research
19 Skyline Drive
Hawthorne, NY 10532
1-914-784-7712
alisonl@us.ibm.com

Abstract: We present an approach to providing broad-based accessibility to graphical applications by employing the capabilities of the Eclipse development platform and aspect oriented programming (AOP) in a way that leverages the strengths of each. We describe the Eclipse platform and the advantages of AOP and contrast AOP with object oriented programming (OOP). Then, we discuss how to use of Eclipse frameworks and AOP in the design of accessible Eclipse-based applications and share some situations of when to leverage these frameworks and when we do not from the perspective of robustness and efficiency.

1 Introduction

Current efforts to provide accessibility typically support third-party reconstruction of the application context. They have enabled users with diverse abilities to gain access but their user experiences remain impoverished compared to that for able-bodied users. For rich user accessibility experiences, assistive technologies need appropriate and meaningful access to application semantics and context. In turn, the technologies need to exploit the particular modalities (e.g., speech) to render the application context appropriate to the user's special needs. Take for instance a monthly calendar rendered by a visual application as a table widget. For a blind user, rather than reading the particular date (e.g., December 17, 2004) as the value 17 at the third row of the sixth column, the assistive technology should be aware that it is a monthly calendar object and speak the 17th as the third Friday of December. Similarly, other users with diverse abilities, such as a visually impaired user, require a different and optimal solution tailored to their special needs.

In order to provide such rich user experiences adapted to users with diverse abilities, it would be necessary to modify the application code and maintain multiple code bases. This is neither feasible nor practical. However, we argue for an alternative approach that views accessibility and different accessibility needs as a cross-cutting concern of usable and accessible software systems. Accessibility enhancements would be realized as different sets of aspects and enhanced application functionality packaged as add-ons. Users with diverse needs would run the standard application in combination with the selected add-ons to experience the application adapted to their needs. This aspect-based approach makes the development and deployment of accessible software both viable and manageable. We describe the aspects-based approach, the access framework, and several illustrations of speech enablement for the Eclipse platform. We discuss how our approach to access-enable the Eclipse platform is general enough to speech-enable the Eclipse IDE and other Eclipse rich-client applications.

Furthermore, when the aspect-based approach is supported by an extensible and open application platform such as Eclipse, developers can leverage the combined capabilities to facilitate the development and deployment of accessible software. We share some situations where accessibility enhancements using aspects is simplified by a framework-rich platform and where accessibility enhancements can leverage a framework-rich platform without the need for using aspects.

In the next section, we provide an overview of prior efforts to provide accessibility in graphical applications. Then, we provide background on aspect-oriented programming, accessibility as a cross-cutting concern, and the Eclipse platform. Next, we describe our approach, specifically for speech enablement. We describe how access enhancements are deployed, how different communities of users with diverse needs use these enhancements, and how Eclipse-based applications can benefit from these enhancements. Finally, we share experiences with how best to leverage AOP in a framework-rich platform such as Eclipse.

2 Prior Work

There have been a number of efforts to provide accessibility to graphical applications [4, 8, 10, 11, 12]. The primitive but naïve approach is to read the content of the display. A second approach externally taps the application context to third-party reconstruction of the application context using APIs to objects and methods (e.g., MSAA). The annotation approach creates a data model for the application and provides transformations to render different user interfaces. The last approach makes use of aspect-oriented programming to create custom extensions and modifications to software systems without directly modifying the original source code.

The first two classes of approaches run alongside but outside of the application context to re-construct a model of the application via different means and to provide users with tools to query their model. Neither of those approaches have direct access to the underlying runtime context from within the application so that the richness of the user experience is limited. The third approach rules out existing applications as a data model is not pre-existing. Emacspeak [12] is the only example using the aspect approach to speech-enable Emacs for access by blind and visually impaired people.

Our Eclipse work extends on the Emacspeak approach in three directions. First, we focus on how to speech-enable a non-speech-aware GUI application and its UI components with spoken input and spoken output. Second, we develop an access framework that allows other Eclipse-based applications to reap the functionality of the enhancements as well as a set of core services to access runtime application context. Third, the Eclipse platform uses an object-oriented programming methodology supported by the Java language, makes extensive use of design patterns, and provides a rich variety of frameworks to support application development. In contrast, large portions of Emacs are built using Lisp, a functional language. Hence, aspect augmentation of the Eclipse platform can leverage its rich set of frameworks.

3 Aspect-Oriented Programming, Accessibility and Eclipse

3.1 Aspect-Oriented Programming in a Nutshell

Software system design involves articulating requirements and decomposing them into a *set of concerns*. Concerns fall into one of two categories: *core* and *cross-cutting*. *Core* concerns capture the central functionalities of a software module while *cross-cutting* concerns capture secondary, peripheral requirements that cut across multiple software modules.

Aspect-oriented programming (AOP) is a programming methodology that addresses concern abstraction and the development of software systems that support the separation of concerns without having to sprinkle program code related to such concerns across a large number of modules [3, 6]. Identifying and separating concerns is particularly important for software system implementation in order to preserve independence of the concerns and permit individual concerns, and thus the software system, to evolve without affecting the other unrelated concerns.

While object-oriented programming (OOP) is good at modularizing core concerns, AOP is good at modularizing cross-cutting concerns. Thus, AOP differs from OOP in the way it manages cross-cutting concerns by encapsulating each cross-cutting concern into its own independent unit known as an *aspect*. Aspect is the central unit of AOP. It encapsulates an individual concern, the implementation code known as *advice*, and the weaving rules (i.e., *point-cuts*) that identify points of execution (i.e., *join points*) where advices are integrated together to form the final software system. AspectJ is an implementation of AOP based on the Java language.

3.2 Accessibility as Cross-Cutting Concern of Usable/Accessible Systems

We argue that *accessibility* is a real-world, pervasive, cross-cutting concern of usable and accessible software systems. First, consider the question of modifying large software systems to meet the special needs and abilities of a given user or community of users. This domain is normally referred to as making software accessible. This can be achieved by adding appropriate functionality (e.g., speech output, font enlargement) to an existing code-base (e.g., adding appropriate calls to produce spoken output to support users with visual impairments). Such functionality needs to be reproduced across multiple software systems. Augmenting a software system with such functionality is an excellent example of a cross-cutting concern. Otherwise, doing so without AOP is difficult if not impossible.

Second, users with diverse abilities often need multiple and varying accommodations to address the particular nature of their impairment (e.g., low vision users are supported by a combination of speech and magnification enhancements). They require a customized rather than one-size fits all access solution. Developing and maintaining multiple and customized versions of software systems is neither feasible nor practical. However, aspects and AOP provide an excellent way for users with special needs to reap the benefits of specialized accommodations and customizations without the problems of source code modification or multiple sources.

Presently, accessibility solutions, such as screen readers, are mostly based on third-party software add-ons that attempt to externally augment the user interface with limited access to an application's semantics, state, and functionality. The resulting user experience is impoverished, awkward, and generally requires additional user action and interpretation to reconstruct the application run-time context. In addition, applications developed for a visual presentation are optimized to that modality. To render the same application information effectively in an aural form would require different optimizations and approaches that take into account the modality, the user's abilities and the context. For example, the proper way to speech-enable the date selection displayed as 11/06/2004, is to say “November 6, 2004.” Such an accommodation would require access to the application model which would differentiate between aural and visual presentations. Aspects and AOP would facilitate the development of this functionality by enabling accessibility developers to augment the application to deliver the required access to the application model and allow the application model to provide the appropriate aural rendering.

3.3 Eclipse Platform

Eclipse [1] is an open, extensible platform (see Figure 1) that has evolved from a universal tool platform to being a platform for any end-user applications. Its initial conception was as an open IDE platform for integrating a diversity of development tools. More recently, the rich client platform (RCP) was created from the generally useful portions of the Eclipse workbench (e.g., window-based GUI, UI frameworks, plug-ins, help system, update manager). It allows developers to use it as an extensible desktop application [2] and as a platform for building client applications with rich functionality.

The *plug-in* is the central Eclipse platform mechanism for defining new functionality and extending an existing functionality. It is the smallest unit of function that can be developed and delivered separately. Other than a small kernel known as the platform runtime, all of the Eclipse Platform’s and the Eclipse Rich-client Platform's functionality is located in plug-ins. A plug-in exposes functionality to other plug-ins through *extension points* and/or declares interconnections to other *extension points* provided by other plug-in. Thus, e*xtension points* are well-defined places where plug-ins can contribute and link to functionality.

The plug-in architecture with its plug-ins, extension points, and extensions work in concert with a rich-set of APIs to provide an open system that treats extensibility as a truly first class notion. The rich APIs provide the rich set of frameworks and building blocks for extending the Eclipse IDE and rapid development of rich-client applications.

4 Speech-Enabling Eclipse-based UI Components

4.1 Providing Spoken Output

Spoken output is provided in response to user activities such as selection of UI components such as a node element of a tree widget. Enabling speech in this situation involves being alerted to the user activity. The former can be facilitated with an aspect that hooks into the event notification framework of Eclipse. However, the Eclipse SWT widget toolkit supports an event listener framework. We can attach listener objects on all visible UI widgets for specific user activity.

When an event is triggered, the appropriate content to be spoken needs to be obtained. A simple approach, supported by current accessibility technologies, is to speak the visual textual content associated with the target widget. However, this content was generated by the application for visual presentation and is not necessarily appropriate for an aural presentation that would aid the user’s comprehension; recall the tabular calendar example given at the outset of the paper. The appropriate approach is to provide the content to be spoken based on the application’s model. This requires retrieving the application’s model given its view (i.e., target widget). Many

Eclipse applications create its user interactions using the Model-View-Control design pattern provided by JFace. For example, the Eclipse Package Navigator uses JFace to instantiate a TreeViewer object as a control and creates an appropriate model. The control then uses SWT to instantiate a Tree widget to provide the view. The view is populated by the control with content provided by the model.

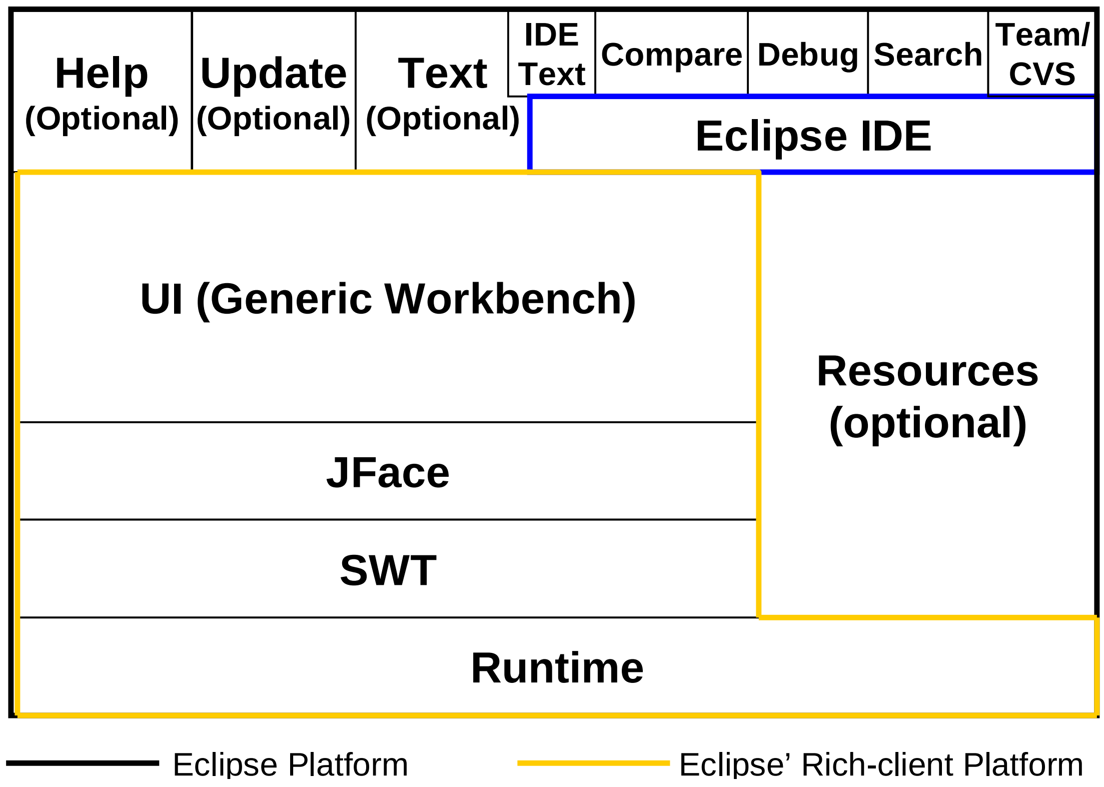


Figure 1: Eclipse platform, rich-client platform, and Eclipse IDE

Once an event triggers and its target widget is identified, we need to map from the widget to its control. Unfortunately, the current SWT implementation does not expose such a mapping. However, an aspect can be used to provide the needed mapping functionality. The aspect hooks in at the join point where the JFace TreeViewer object associates itself to its tree widget view through a method call to hookControl (see Figure 2).

Once the TreeViewer control object is identified, we gain access to the application's model. We use another aspect to inject Java classes that provide content to be spoken. The aural, content providers produce the appropriate spoken content using the application's model. We define a class, rather than an interface, in order to provide default implementations for the aural content to be spoken by all JFace TreeViewers instantiated at runtime. Specific applications can subclass these classes to customize the spoken content for richer user experiences.

The base implementation of the aural content provider uses the audio formatting scheme, first introduced in Emacspeak, to leverage the features of the auditory display to communicate information effectively and succinctly. With this scheme, spoken text can have one or more aural properties applied to it. We provide a simple API that the application's model content provider (i.e., aural content provider) can use to attach aural properties to text. When the text is later spoken, the speech service examines any properties attached to a given utterance and produces the specified variations by manipulating one or more voice characteristics (e.g., pitch, speed, gender, breathiness, and roughness). Just as the visual rendering of user-selected UI component, such as a tree node, is constructed from multiple pieces of information (e.g., decorators, text, icons), the aural rendering can have multiple aural properties that are contributed by different pieces of the application model realized by the hierarchy of models defined by the application. The cumulative effect is correctly rendered when presented to the speech service. Allowing an application and its contributing constituents to attach aural properties to text, we enable the application to produce a conceptual aural display list that is consistently handled by the underlying speech interface.

```
import org.eclipse.jface.viewers.AbstractTreeViewer;
import org.eclipse.swt.widgets.Control;

public privileged aspect ViewerHook {
 private AbstractTreeViewer Control.tree_viewer=null;
 public AbstractTreeViewer Control.getViewer() {
   return tree_viewer;
 }

 before(AbstractTreeViewer v, Control c) :
   target(v) && args(c) &&
   call(protected void hookControl(Control)) {
     c.tree_viewer = v;
   }
}
```

Figure 2: Before aspect hooks into calls to the method hookControl made by AbstractTreeViewer to associate the tree widget Control to the tree viewer. The aspect creates an instance variable of type AbstractTreeViewer to remember the tree viewer object associated with the tree widget Control.

4.2 Providing Spoken Input

It takes two sides to have an effective conversation; in this section, we detail how the speech-enabling approach implemented via Aspect Oriented Programming can be used to advantage in turning standard user interaction components into fully interactive widgets that enable the user to provide spoken as well as keyboard/mouse input.

As was observed when generating effective spoken output, AOP enables us to add hooks at the appropriate level to inject new behavior into an existing implementation. Such behavior is implemented via small fragments of aspect code that have full access to the runtime context of the application. This runtime context can be used to advantage in communicating the appropriate pieces of context to the underlying speech recognition system. As a naive example, the fragment of code that speech-enables a list selection can communicate the available choices to the speech input layer, which in turn can construct a grammar that is used to recognize user input while the user is interacting with that particular list box. Further, one can introduce appropriate aspects to dynamically update the grammar as the list of choices get filtered during user interaction.

We illustrate this with an example. Consider a file selection dialog that allows the user to pick from a list of files. Initially, the dialog displays the names of all files in the current directory. When the user enters this dialog, we invoke an appropriate aspect that activates the speech-input component and initializes it with a grammar that matches the current list of filenames. The dialog also presents a set of controls on the screen that enable the user to filter the list by filetype. As an example, the user might filter the list by requesting that only files of type .xml be displayed. This results in the list of choices being pruned; we intercept the call that updates the list of choices with an appropriate aspect that refreshes the speech-input component with the grammar that matches the new list of choices. In a typical Java project, the above step might result in the list of available choices reducing to a single build.xml file; the user might then open this file by a simple "open it" utterance, where the speech-input component uses the fact that there is only one available choice to resolve the anaphoric reference, "it."

This design pattern can be replicated at different levels of user interaction. For example, we can introduce appropriate pieces of aspect-oriented code that attach themselves to the point in the environment that handles switching focus among various applications. As the user switches focus, the aspect can query the application that has focus for a top-level command vocabulary and activate the corresponding grammar. Spoken input can then be handled in a manner that makes most sense within the application that has focus.

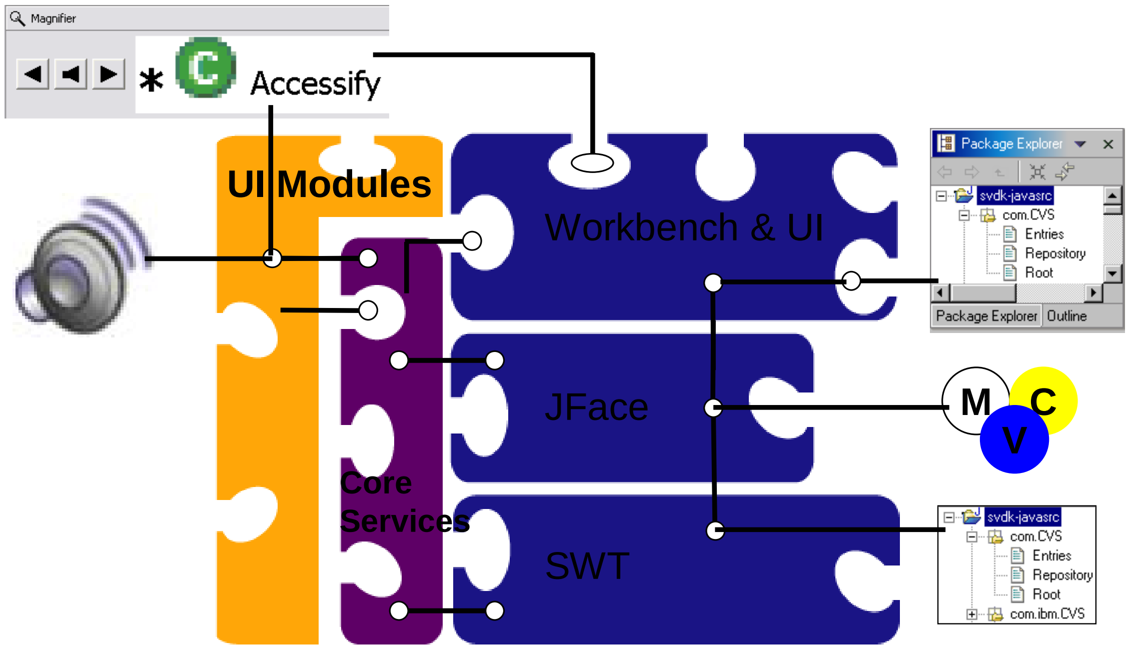


Figure 3: Access framework, consisting of core services and UI modules, embedded within Eclipse platform. Illustration of instantiation of Eclipse Project Navigator and its integration into the Eclipse Platform and access framework.

We illustrate this with an example. Assume that the user is interacting with a calendar and email application. Each of these expose a set of command words and top-level event handlers that are executed upon receiving those command words; as an example, the calendar might include tomorrow and next week within its command vocabulary; whereas the email reader includes commands such as next message and replay. As the user switches focus from the calendar to the email reader, we query the email application for its command vocabulary, and activate the corresponding grammar via an aspect that is triggered upon entering the email application. In fact, the action of switching focus might itself be enabled via spoken input by attaching an aspect to the desktop manager that creates and maintains a top-level grammar which is updated each time a new application is opened.

4.3 Access Framework and Core Services

Our access-enablement of the applications consists of an access framework and accessibility add-ons. Both Eclipse and Activity Explorer were enabled using the same access framework and accessibility add-ons. The access framework integrates, on-demand, the core services and UI modules into the application without the need to modify the application source code or to maintain multiple source versions (see Figure 3). The accessibility add-ons leverage the access framework, core services, and UI modules to render information and provide functionality in support of different accommodations. The core services access and expose application information from within to enable the accessibility add-ons to leverage context information and functionality. The UI modules provide different user interface accommodations (e.g., speech input, magnification).

As a concrete example of functionality introduced as core service and UI module to the access framework, we describe how we introduced auditory icons to make visual icons used to mark notable lines in the Java Editor accessible to users with visual impairment. In the Java Editor, the visual icons highlight lines with compiler errors, warnings, and comments marked with TODOs. We introduce auditory icons as a UI module that is activated when the user is on an editor line containing the visual marker. Eclipse editors such as the Java Editor support code folding structure wherein regions of code such as import statements, comments, types, and methods can be hidden or exposed. In order to ensure that the proper interpretation of the editor's model of which lines contain markers given the visual line number, it is necessary to map visual line number to the model line number. An aspect was created as part of the core service for Eclipse editors to expose this application data. This was necessary because the original AbstractTextEditor API did not expose the specialized object whose method provides this mapping (see Figure 4).

```
import org.eclipse.jface.text.source.ISourceViewer;
import org.eclipse.ui.texteditor.AbstractTextEditor;

public privileged aspect SourceViewerAccess {
 public ISourceViewer AbstractTextEditor.getPublicSourceViewer() {
   return getSourceViewer();
 }
}
```

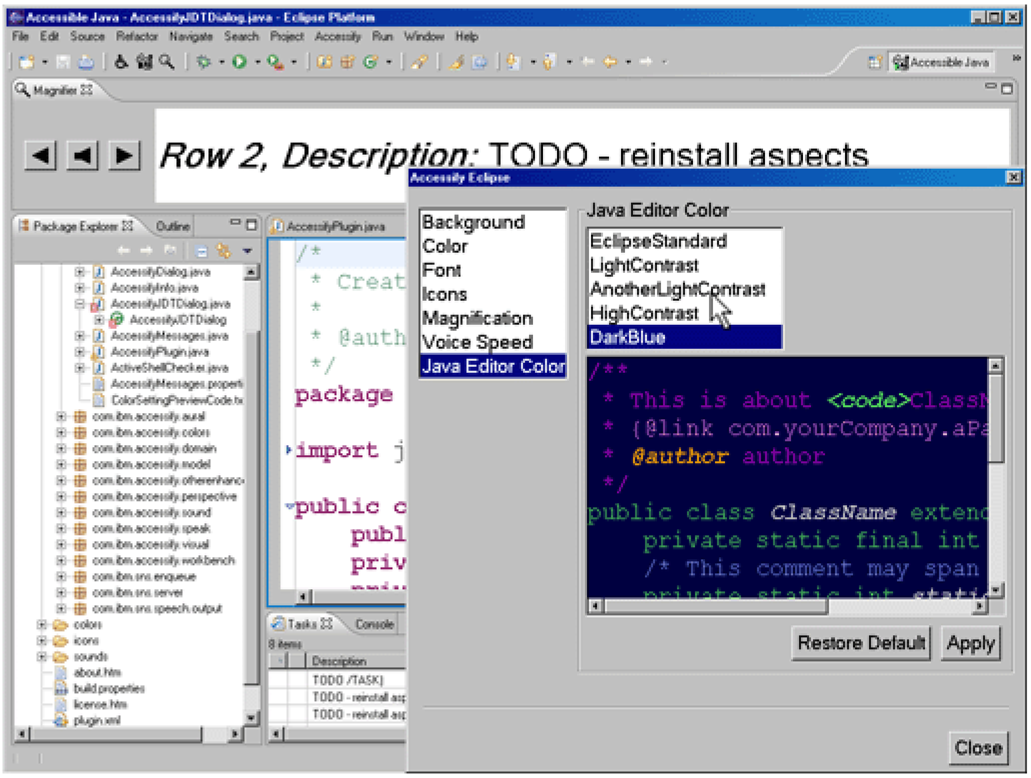
Figure 4: Aspect to expose a private method of the AbstractTextEditor class called getSourceViewer. The aspect uses introductions to add a new method to the AbstractTextEditor object that permits the access framework to pass along the specialized editor object to the UI module to invoke the mapping from visual line number to model line number.

4.4 Putting it Together

The core services of the access framework are packaged as an Eclipse plug-in (i.e., access plug-in) that provides basic functionality for accessing the Eclipse context. We define a number of key functionality that is available to accessibility developers through *extension points*. As we expand the functionality of the core services, this plug-in will evolve with enhancements and new extensions.

Individual accessibility enhancements (e.g., speech output, speech input, magnifier) to address different diverse needs (e.g., blind, low-vision) are modularized into separate plug-ins. Users with particular impairments select the appropriate add-on plug-ins tailored to their special needs. The add-on plug-ins work with the access plug-in to access the Eclipse application context and to deliver the user experiences tailored to the special needs.

In the deployment of an Eclipse application, the standard application, the access plug-in, and add-on plug-ins are available for distribution. Users with diverse needs would simply download the standard Eclipse application, the base access plug-in, and selected add-on plug-ins. When the application launches, the additional plug-ins are loaded by the runtime environment. Accessibility enhancements implemented as aspects are woven into the runtime environment using a load-time weaver such as AJEER [7].


Accessible Java - AccessibilityJDTDialog.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Accessibility Run Window Help
Magnifier
Package Explorer
Outline
AccessibilityPlugin.java
Tasks
Console
Accessibly Eclipse
Restore Default
Apply
Close

Row 2, Description: TODO - reinstall aspects

Background
Color
Font
Icons
Magnification
Voice Speed
Java Editor Color

Java Editor Color
EclipseStandard
LightContrast
AnotherLightContrast
HighContrast
DarkBlue

```
/**
 * This is about <code>ClassN
 * {@link com.yourCompany.aPa
 * @author author
 */
public class ClassName extend
    private static final int
    /* This comment may span
```

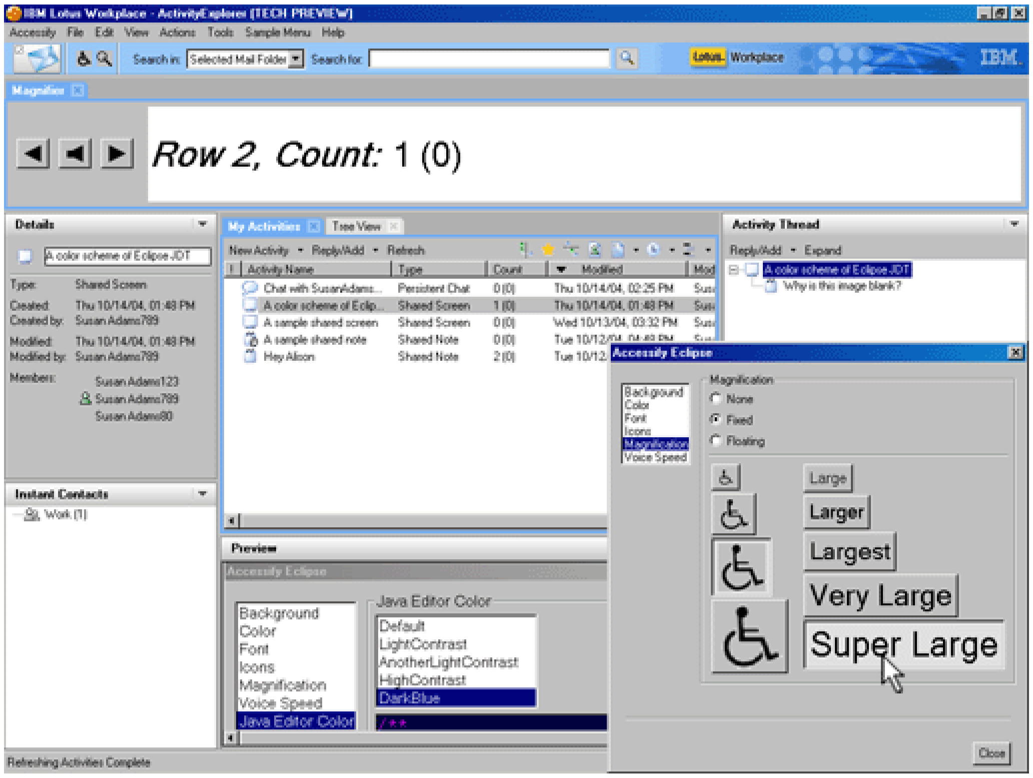
Figure 5: Visual enablement of Eclipse, in particular, magnification of font and currently selected table item. A preference panel permits users to set accessibility options. Users may also have magnified item spoken via keyboard or mouse interaction.

We have used this approach to speech-enable the main UI components used by the Eclipse IDE (see Figure 5) as well as providing other non-speech enablement (e.g., magnification, incremental find for Tree components). More recently, we have re-factored our access plug-in and some of the add-on plug-ins by separating the general enablement from those specific to the IDE application. We used the re-factored plug-ins for an Eclipse rich-client application known as Activity Explorer [9] (see Figure 6). To improve beyond the current accessibility user experiences available for both these applications, the initial set of access and add-on plug-ins can be extended with additional plug-ins and aspects. The current set of plug-ins clearly validates the approach, its generality for other applications, and demonstrates how incremental enhancements can be provided to improve overall user experiences. As with current Eclipse development, these enhancements can come from both the accessibility development and Eclipse development communities.

4.5 To AOP or to Eclipse

Accessibility enablement of Eclipse applications can be done using AOP exclusively. However, there are clear advantages to combine AOP and the Eclipse platform in order to leverage the strengths of each. We highlight some examples where the partnership has led to a more efficient and compact implementation of the accessibility enhancements.

First, we use the plug-in architecture to modularize and deliver the different accessibility enhancements and the application access framework. This simplifies deployment and enables different user communities to select the appropriate accessibility add-on plug-ins they desire. Specifically, a low-vision user may combine speech and visual magnification while a hearing-impaired user may combine visual enhancements and tactile feedback.


IBM Lotus Workplace - ActivityExplorer (TECH PREVIEW)

Row 2, Count: 1 (0)

Details: A color scheme of Eclipse JDT

Type:	Shared Screen
Created:	Thu 10/14/04, 01:48 PM
Created by:	Susan Adams789
Modified:	Thu 10/14/04, 01:48 PM
Modified by:	Susan Adams789
Members:	Susan Adams123, Susan Adams789, Susan Adams80

My Activities

Activity Name	Type	Count	Modified
Chat with SusanAdams...	Persistent Chat	0 (0)	Thu 10/14/04, 02:25 PM
A color scheme of Eclip...	Shared Screen	1 (0)	Thu 10/14/04, 01:48 PM
A sample shared screen	Shared Screen	0 (0)	Wed 10/13/04, 03:32 PM
A sample shared note	Shared Note	0 (0)	Tue 10/12/04, 04:48 PM
Hey Alison	Shared Note	2 (0)	Tue 10/12

Activity Thread: A color scheme of Eclipse JDT; Why is this image blank?

Accessify Eclipse: Magnification: None [✓ Fixed] Floating

Java Editor Color: Default, LightContrast, AnotherLightContrast, HighContrast, [✓ DarkBlue]

Refreshing Activities Complete

Figure 6: Speech and visual enablement of Activity Explorer (an Eclipse rich-client application) using the re-factored accessibility plug-ins first developed for the Eclipse IDE.

Second, we exploit the listener framework available from the SWT widget toolkit to attach appropriate listeners for delivering different user feedback (e.g., visual, aural). There are several benefits. It ensures that all the visible widgets were covered by the listeners. We could also deliver a base-level of accessibility functionality such as audio formatting and spoken output on all selections of UI components. We use a known and supported framework that ensures versatility and robustness of our enhancements. When certain aspects rely too much of an implementation that is not public, we run the risk of the aspects breaking when the implementation changes. Reliance of undocumented features should be avoided as much as possible and certainly when there are public frameworks available. However, in some situations, this is necessary. One situation is when the original API designers did not anticipate other scenarios of use such as the one illustrated in the example shown in Figure 2. In that case, we needed to map from the tree view object to its control object. A second situation is when the original designers were over-cautious with their API design. A case in point is the example highlighted in Figure 4 where the method getSourceViewer was made package private. Access to this method allows mapping from visual line number to model line number.

Third, while use of an existing Eclipse framework is generally useful, there are situations where the AOP approach is chosen instead. This is best illustrated with the example of adding a magnifier view at the top of every instantiated Eclipse perspective; an Eclipse perspective is a window layout identifying a collection of Eclipse views associated with a particular user task (e.g., Java development, CVS repository browsing). An Eclipse perspective implements the interface IPerspectiveFactory with a method createInitialLayout to define its initial layout and views. So, one naive approach is to create a second "accessible" version for each perspective that would add a magnifier window to each these layouts. This is easy to do but causes proliferation of perspectives. Also, when new perspectives become available, it would be necessary to create the second "accessible" perspective.

Alternatively, an aspect can be created in one of two ways. The first approach would use a before *advice* that activates on **execution** of the createInitialLayout method's code body. A before advice is needed to ensure that the magnifier view stretches the full width of the topmost area of the window layout; an after advice can be pre-empted by the execution of the method from obtaining the full width of the topmost area of the window layout. An execution *join point* refers to the implementation code body of the createInitialLayout method which defines its perspective's layout and views. Every plug-in that creates a perspective would be woven with this. The weaver does mechanically more work. However, this approach accommodates finer grain control and permits different advice code to be associated with each *join point*.

The second approach would use a before *advice* that activates at the **call** *join point* to the createInitialLayout method (see Figure 7). As the Eclipse Workbench uses a Perspective Factory design pattern to create the initial layout of the current workbench perspective, there is only one **call** *join-point*. This is mechanically less work for the weaver; hence, efficient. However, if different actions are to be taken for different perspectives, then filtering code must be embedded in the advice code. Unlike the preceding approach which distributes the filtering code with the specific join-point for creating a perspective's layout, this centralizes all the filtering program code.

```
import org.eclipse.ui.IPerspectiveFactory;
import org.eclipse.ui.IFolderLayout;
import org.eclipse.ui.IPageLayout;

public aspect PerspectiveAlter {
 before(IPerspectiveFactory perspective, IPageLayout layout):
   target(perspective) && args(layout) &&
   call(void createInitialLayout (IPageLayout))
 {
   String editorArea = layout.getEditorArea();
   IFolderLayout magnifierfolder =
     layout.createFolder("top", IPageLayout.TOP, 0.2f, editorArea);
   magnifierfolder.addView(
     "com.ibm.accessify.workbench.MagnifierViewPart");
 }
}
```

Figure 7: Before aspect invoked on a method call to createInitialLayout to add magnify view to every Perspective that is opened.

5 Concluding Remarks

Providing rich user experiences to users with diverse abilities frequently requires modifying the application code base or maintaining multiple versions. In this paper, we propose an alternative approach that views accessibility as a cross-cutting concern of usable and accessible software systems. We describe how the implementation of functionality to support users with diverse needs can be facilitated using an aspect approach. Aspect-oriented programming (AOP) is based on the idea of *advising* existing code using *aspects* to create custom extensions and modifications to large software systems without directly modifying the original source code for the application being extended. In addition, the extensions can be loaded *on-demand* leading to an architecture that enables flexible but robust adaptive solutions.

Using the Eclipse IDE application and the Activity Explore rich-client application, we showed how aspect-oriented programming facilitates a general, evolvable, viable, and manageable approach for developing and delivering accessibility functionality that can lead to rich user experiences for different user communities including those with special needs. While most of our examples focused on speech enablement for both input and output, we have also shown that the approach can be used to provide other modality enhancements (e.g., visual). Furthermore, when accessibility development is performed using an extensible and open application framework such as the Eclipse Platform, developers can leverage the combined capabilities to facilitate the development of accessible software. We share some situations of when to leverage these frameworks and when we do not from the perspective of robustness and efficiency.

6 References

1. eclipse.org. http://www.eclipse.org/.

2. Eclipse.org. Eclipse Rich Client Platform UI, http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/rcp/generic_workbench_summary.html.

3. T. Elrad, M. Aksits, G. Kiczales, G. Lieberherr, and H. Ossher, “Discussing Aspects of AOP”. Communications of ACM, 44(10), ACM:New York, pp. 33 – 38, 2001. http://doi.acm.org/10.1145/383845.383854.

4. Freedom Scientific, Inc. Jaws. http://www.freedomsci.com/.

5. G. Kiczales, E. Hillsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “Getting Started with AspectJ”. Communications of ACM, 44(10), ACM:New York, pp. 59 – 65, 2001. http://doi.acm.org/10.1145/383845.383858, http://www.eclipse.org/aspectj.

6. Laddad, R. “AspectJ in Action: Practical Aspect-Oriented Programming.” Manning:Greenwich, 2003.

7. M. Lippert, “An AspectJ-Enabled Eclipse Core Runtime”. In Proceedings of OOPSLA 2003, Poster, ACM:New York, 2003.

8. Microsoft. Microsoft Accessibility: Technology for Everyone. http://www.microsoft.com/enable/.

9. M.J. Muller, W. Geyer, B. Brownholtz, E. Wilcox, and D.R. Millen, “One-Hundred Days in an Activity-Centric Collaboration Environment Based on Shared Objects”. In Proceedings of the 2004 Conference on Human Factors in Computing Systems, ACM:New York, pp. 375 – 382, 2004. http://doi.acm.org/10.1145/985692.985740.

10. E.D. Mynatt and W.K. Edwards. “Mapping GUIs to Auditory Interfaces”. In Proceedings of UIST'92, ACM:New York, pp. 61 – 70, 1992.

11. T.V. Raman. XForms: XML Powered Web Forms. Addison Wesley Professional. 2003.

12. T.V. Raman. “Emacspeak: A Speech-Enabling Interface”. Dr. Dobb's Journal. September 1997. http://emacspeak.sf.net.