Dave Pletcher

EE564 – Midterm Paper

Design of the JavaEditor Class Hierarchy

In this paper, I will discuss the architecture of the Eclipse Platform's JavaEditor class in Eclipse Platform v3.2 and mention the roles of some of the more fundamental classes and interfaces that operate in JavaEditor's context. The total number of such types isn't small, and I found it would be impossible to do a thorough design analysis on such a complicated class in only a few weeks. So, I will attempt to only refer to the most fundamental types and their relationships amongst each other in the context of JavaEditor throughout most of the paper.

The classes and interfaces that JavaEditor inherits from are mostly scattered amongst three plug-ins: the UI part classes/interfaces are defined in org.eclipse.ui, the editor-specific classes/interfaces for generic text editors are found in org.eclipse.ui.texteditor, and JavaEditor (and its two concrete children subclasses) are found in org.eclipse.jdt.internal.ui.javaeditor (which is part of the JDT plug-in). The concrete subclasses are in a package marked internal to discourage clients from attempting to create a new instance directly. The workbench has some sort of editor registry in which JavaEditor should be registered as the associated handler for java source/class files. This way, the workbench will instantiate the editor class and client code will never be forced to directly interact with it. To understand how the JavaEditor class works, we need to take at least a brief look at each class and interface in its super-type hierarchy (Figure 1 in the appendix).

As specified by org.core.eclipse.runtime.IAdaptable, an instance of JavaEditor is adaptable: that is, we can use it to supply and if necessary produce, instances of objects that have been adapted to be compatible with a given class type. I'll restrict my comments on org.core.eclipse.runtime.IExecutableExtension to saying that this means JavaEditor is a plug-in (that automatically executes its own initialization code I think) to an extension point on another plug-in within Eclipse. The interface org.eclipse.ui.IWorkbenchPart specifies a way for clients to access the Workbench, register PropertyListeners, and control the lifecycle of the part. Then, its child interface, org.eclipse.ui.IWorkbenchPart2 was added in Eclipse v3.0 to provide a way to retrieve the part name and description. The class org.eclipse.core.commands.common.EventManager was added near the base of the subtype hierarchy in v3.2. It manages a list of generic Listener objects. I'm not certain of the purpose of this class considering how many specialized Listener classes exist in our context. The interface org.eclipse.ui.part.IWorkbenchPartOrientation provides a way to retrieve the part's visual orientation.

The class org.eclipse.ui.part.WorkbenchPart, ties together all the classes and interfaces described so far, but provides mostly default implementations. However, this is the first functionality we've seen implemented (aside from the overly-trivial EventManager class), and at a glance, appears to provide the kind of functionality you would expect from an abstract workbench part. Examples include a way to fire a property-changed event. Also, the implementation of the getAdapter() method queries the platform's adapter manager when a client has requested an adapter, or more likely

because the original call was on a WorkbenchPart subclass that (along with any other WorkbenchPart subclass in the chain) was unable to supply the requested adapter.

The interface org.eclipse.ui.ISaveablePart is implemented or adapted to by a workbench part that we want to provide implementations of the save and "save as" operations. The interface org.eclipse.ui.IEditorPart inherits from IWorkbenchPart and ISaveablePart and specifies methods to retrieve the IEditorInput and the IEditorSite, as well as initialize a new editor with a provided IEditorInput and IEditorSite.

The class org.eclipse.ui.part.EditorPart is similar to WorkbenchPart in that it is an abstract class providing mainly default functionality. It inherits from WorkbenchPart and IEditorPart, provides a mechanism for ensuring backward compatibility for clients of the setTitle() method, which was replaced with the methods specified in IWorkbenchPart2.

The interface org.eclipse.ui.INavigationLocationProvider is implemented by editor instantiations that add to the workbench's navigation history. The interface org.eclipse.ui.IReusableEditor inherits from org.eclipse.ui.IEditorPart and specifies a method to change an editor's IEditorInput at runtime.

To properly discuss the interface org.eclipse.ui.texteditor.ITextEditor, we must also examine some other interfaces it interacts with. It is specified here that it is intended that a text editor obtain the textual representation of documents via an org.eclipse.ui.texteditor.IDocumentProvider, so we'll have to devote a moment to that interface. The IDocumentProvider specification defines the following context: a domain model (like a workspace or CVS repository's file system structure), as well as domain model elements (generally resources like files and folders). It then lays out the method signatures for the functionality that can be used to map between domain model elements

(resources) and the data they actually contain. It also provides an annotation model for the editor as well as the ability to listen for element state changes (i.e. changes in the availability of resources). So when we see that ITextEditor specifies a getDocumentProvider() method, we can assume that text editor instances aggregate a document provider instance that can be accessed by subclasses. A similar situation exists in how it handles selections: a method is provided to retrieve the editor's selection provider, so that subclasses or clients can listen for selections and access them when needed. In addition, the foundations for range highlighting and getting / setting actions (strategies for responding to specific user input) are specified in this text editor interface.

It is stated (in the interface specification) that to provide backward compatibility for clients of text editor, extension interfaces are used to enhance the interface. This makes perfect sense to me since we can't change the text editor's specification without breaking existing code. What confuses me is that none of ITextEditor's extension interfaces actually extend ITextEditor. It seems only natural to me that they would, and I can't think of a scenario in which that would break anything. For now, I'll assume this is done so editors have the option of delegating the responsibilities specified in the extension interfaces to completely different objects. In any case, the text editor interface has 4 extension interfaces (also in package org.eclipse.ui.texteditor) that specify the following features. ITextEditorExtension specifies: configuration of status fields, check for read-only state, listen for ruler context menu events. ITextEditorExtension2 specifies a way to ensure the editor's input is in a persistently modifiable state as well as validate the state of the editor's input. ITextEditorExtension3 specifies that the editor will

implement insert mode management. ITextEditorExtension4 specifies a way to navigate through the editor's annotations and to display revision information for the document.

The class org.eclipse.ui.texteditor.AbstractTextEditor inherits from EditorPart and all the interfaces we have discussed this far. Although this class is fairly lean, it provides default implementations of lots of listeners and actions that subclasses and clients can use or override, and it composes an org.eclipse.jface.text.source.SourceViewer, which is used to separate the responsibility of the actual handling of SWT widgets from the editor classes. It's hard to focus in on exactly what is provided in this class. I'm hoping it will suffice to say that everything you would expect to be implemented at this point in the hierarchy seems to be here, or has foundations here. Sometimes the justification used for how much of a foundation was laid for the development of subclass features with arguable commonality is questionable, but for now I will defer this discussion until the critique section.

The class org.eclipse.ui.texteditor.StatusTextEditor inherits only from AbstractTextEditor, and adds very little compared to its children classes. It seems to have been created as an alternative to popping up dialog boxes to alert users of a relevant document status. If the status of the document is not ok (i.e. doesn't exist in the file-system, or read-only), a status message (and in subclasses, potentially, a means (swt controls) to take action) are displayed in the area you would expect to see the editor widgets. The only other noteworthy bit of information on this class is that it is not abstract.

However, the next class down in the hierarchy, org.eclipse.ui.texteditor.AbstractDecoratedTextEditor is abstract. This class provides

some of the decorative functionality you would expect to see in a good editor, particularly features that a source code editor would use. Whether some functionality found further down in the tree could be moved here is arguable. This will be discussed in the critique and will be fairly speculative due to time constraints. Regardless, some features partially or wholly implemented here include line numbers, change and overview rulers, current line highlighting, and print margins. Generally speaking, I would say this class provides complete implementations for a few features, and a few medium-sized building blocks to allow for customization of additional functionality in subclasses. This is opposed to AbstractTextEditor, which basically just provides a lot of smaller building blocks that cover a broader range of overall functionality.

There is one last interface we must quickly mention before we can talk about the JavaEditor itself, and that is org.eclipse.jdt.internal.ui.viewsupport.IViewPartInputProvider. Its javadoc description basically states that this interface is common to all view parts that provide an input. JavaEditor's implementation returns an adapter to the editor input that conforms to the IJavaElement interface. I didn't absorb all the implications of this, and feel there isn't enough time to research a more precise description.

Well, we can finally talk about the class JavaEditor now, which is part of the JDT plug-in, and in package org.eclipse.jdt.internal.ui.javaeditor. Unfortunately, we've kind of reached an anti-climax here: although JavaEditor provides all the functionality you would find common to Java source editors, it's not a concrete class! It has two subclasses: ClassFileEditor and CompilationUnitEditor, which are both concrete, but we'll have to talk about JavaEditor first. It contains a variety of public inner classes that

implement assorted IActions and Listeners for use by clients and/or subclasses. Here, we see code folding at least partially implemented. This requires the use of a ProjectionViewer, which is an ISourceViewer that can be customized to project only certain partitions of the document onto the view part. Other parts of this context include an IJavaFoldingStructureProvider, which is intended to listen for projection events generated by the viewer and then change the projection structure accordingly, and a ProjectionSupport object that seems to tie everything together. It is arguable whether the foundations for generic projection support could have been provided in AbstractDecoratedTextEditor, and then customized appropriately for each subclass's source code type, but this will be discussed later. Other features that we see must be at least partially implemented here include semantic highlighting, support for marking various program constructs if desired, an outline page to summarize fields/methods for the class we are currently editing, occurrence annotations, and much more that I must omit for the sake of brevity.

Now, to wrap up this section, we'll quickly discuss JavaEditor's two concrete subclasses. ClassFileEditor seems to be used to edit Java class files which contain the source code as well as byte code. If source code is not attached, a source attachment form is created. This allows the user the option of attaching the corresponding source to this class file. However, when I try to open a random Java class file using the editor, I get an internal error complaining that the editor cannot handle the input, so I must be missing part of the picture here. The class CompilationUnitEditor is a little more involved as it is generally used more often than ClassFileEditor, and is probably the editor you actually see when developing Java source code within Eclipse. One thing I see

implemented here is an inner class that automatically adds the closing bracket right after you (as the user) type an opening bracket. There is also support for tab conversions, and reconciliation.

This is far from a complete picture of how the JavaEditor works, but I've attempted to include the basics and to provide a general idea of how features are implemented. In the next section, we'll take a quick look at the evolution of these classes and interfaces between Eclipse versions 1.0 – 2.0, and then 2.0 – 3.2.

Evolution of the JavaEditor Type Hierarchy

To analyze the evolution of the JavaEditor component as Eclipse itself matured, we will begin by looking at v1.0. At this time, JavaEditor's super-type hierarchy (see Fig.1) was considerably less bulky as many of the features we now take for granted did not yet exist or were designed differently. In all, I counted four interfaces in JavaEditor's super-type hierarchy that remained unchanged from v1.0 to v3.2: IAdaptable, IWorkbenchPart, IExecutableExtension, and ITextEditor. Everything else was either added or changed sometime after v1.0. We can see that JavaEditor descends directly from AbstractTextEditor and ISelectionChangedListener. Features that seem to be missing include text folding, semantic highlighting, change and overview rulers, encoding support, and quickdiff support. The total number of lines of code in JavaEditor's super-type hierarchy in v1.0 is approximately 3.2kLOC (includes comments/white space), and there are of course many more lines of code in various classes and interfaces that are used in the context of our editor.

By Eclipse Platform v2.0, the number of lines of code in JavaEditor and its ancestry is roughly 5.6kLOC. Notable changes in this version since v1.0 include addition of interface ITextEditorExtension, which specifies support for status fields, editor input status, and ruler context listeners. The experimental interface IReusableEditor was created, and allows the workbench to change the editor's input dynamically. I couldn't figure out why EditorPart seemingly preemptively declared the same "void setInput(IEditorInput)" method specified in IReusableEditor, but because the interface was initially experimental, this makes sense. AbstractTextEditor was nearly doubled in size, but still provides mostly default implementations for many of the methods specified by its super-types. It does provide support for new types of IActions and Listeners. The class StatusTextEditor is also an addition since v1.0. It displays editor status messages in the editor instead of in a dialog, and remains virtually unchanged as of v3.2. JavaEditor was enhanced with encoding support and line numbers, and now aggregates an ISelectionChangedListener instead of inheriting from it.

Presently, with the Eclipse Platform at v3.2._, we are at about 13.6kLOC. The functionality associated with the save-ability of a workspace part has been separated into interface ISaveablePart. Another change is that IWorkbenchPart has essentially been replaced with IWorkbenchPart2, which extends IWorkbenchPart. IWorkbenchPart2 provides part name and part content description to clients and/or subclasses. Also, the class org.eclipse.core.commands.common.EventManager was added near the root of the sub-type hierarchy (between Object and WorkbenchPart). This class is used to manage a list of generic listeners, so no notification mechanism is explicitly defined. I believe, that within the context of the editor, this class is used exclusively by WorkbenchPart to notify

property listeners of property changes. WorkbenchPart had this functionality in v2.0, but it wasn't inherited like it is now. Upon closer inspection, I see that it notifies IPropertyListeners of property changes, but then AbstractTextEditor composes a PropertyChangeListener that it uses to observer its preferences store.

One common theme that seemed to gain in popularity in our context as the Eclipse platform evolved is the use of ArrayLists of "something"-dependent IActions, whose elements' update methods are called by the editor in response to a "something"-changed event. This allows AbstractTextEditor to update the action's availability due to a recent change of state (usually signaled by the capture of some type of event). I originally thought this was used to effectively replace listeners in certain situations by grouping related strategies for event handling, but these features don't seem to be used in that manner.

Three more extension interfaces for ITextEditor emerged at some point since v2.0. They were discussed in the first section. However, I'd like to speculate that perhaps this functionality was singled out into interfaces as a way for subclasses/clients to determine which features are present in an ITextEditor, without using some other kind of introspection. I also must be honest and admit that I didn't have time to make sure the functionality defined in the above mentioned extension interfaces wasn't specified or perhaps just implemented elsewhere in previous Eclipse versions.

According to its source code, org.eclipse.ui.texteditor.AbstractDecoratedTextEditor was added to the hierarchy in v3.0. I believe its existence is justified because it captures for reuse at least the foundations of some of the more common features found in heavy-weight editors. The interface

org.eclipse.jdt.internal.ui.viewsupport.IViewPartInputProvider is fairly confusing, and quickly looking where it's used, seems to support a utility method that inspects a structured selection to find an instance of IJavaElement.

The size of the JavaEditor source file more than quadrupled since v2.0, going from ~820 to ~3840 LOC. A lot of this has to do with new responsibilities laid upon it by its altered ancestry. There are lots of additional classes and interfaces now that are used exclusively by JavaEditor to provide its new features. One thing I just quickly looked at is semantic highlighting. I don't have an exact understanding of how it works, but there are several classes (new as of v3.0), internal to the JDT plug-in, that are used to support semantic highlighting. The class that installs itself onto JavaEditor and its Viewer is called SemanticHighlightingManager, and it composes a SemanticHighlightingPresenter and a SemanticHighlightingReconciler which work together. I think it's possible that the reconciler determines how to decorate the editor when the text in its document changes and the presenter does the actual decorating.

JavaEditor now contains quite a few non-private inner classes, mostly for listeners and actions that are specific to a JavaEditor. These are usually appropriately configured by subclasses. It often seems that when one looks to find the meat of a significant feature, it's usually another object altogether that's installed on the viewer and whatever else is involved in the context of the feature. One class like this that I happened to notice is SourceViewerDecorationSupport (v2.1). It observes a preference store for changes, and acts on those changes by activating/deactivating features like matching character highlighting, current line highlighting, print margins, and annotations.

The information provided in this section is far from complete, but I tried to give a feel for not just how much the overall structure has changed, but how much more complicated things got as a result of trying to manage the inevitable complexities of the java editor.

Evolution of the Editor / Overview Interaction

In this section I will try to concisely describe the notification mechanisms that are used to update the selection in the editor when a user clicks on a java element in the outline page. In light of the complexity inevitably encountered when doing code traces involving such large components, I have decided to let class-interaction diagrams explain the finer details. They are included in the appendix.

Within Eclipse v1.0 (see Fig. 4), the JavaEditor listens to its aggregated JavaOutlinePage for selection-changed events. The JavaOutlinePage is itself propagating selection-change events that it receives from its JavaOutlineViewer. It uses the inherited method getSelectionChangedListener() to listen to its source viewer, but this is for the purpose of updating the available actions (copy, cut, paste) based on the new selection. Although the response functionality seems to be there, the outline does not react to selection changes in the editor until sometime after v1.0 I think, but the archived version of Eclipse platform v1.0 has a broken eclipse.exe, so I can't be 100% sure. I've learned that the outline does react to selection changes in the type hierarchy's method view, but only because the type hierarchy delegates an object to listen to it, which then sets the proper selection in the editor outline via JavaEditor, via EditorUtility.

The interaction in v2.0 is different because functionality has been added to update the outline (but not the type hierarchy method view) based on the cursor position within

the editor (see Fig. 5). Instead of listening for selection changes on the outline page itself, JavaEditor delegates this duty to an inner SelectionChangedListener class. Again I'm seeing functionality that can be used to set the selection in the outline page, but just like with v1.0, for our purposes, selection changes generated by the editor seem to be ignored. It is the cursor position that dictates the editor overview's selection.

In v3.2, the interaction is of roughly the same complexity as v2.0, but the logic is organized more sensibly (see Fig. 6). The editor overview's selection is still synchronized with the selection in the editor, but the means to achieve this are different. Although we continue to use cursor-listening as a strategy for listening for selections in our source viewer, this functionality is wrapped up in a IPostSelectionProvider many super-classes up, and all JavaEditor needs from the viewer (for our current purpose) is IPostSelectionProvider ancestry. Note that in all versions, the JavaEditor must temporarily uninstall itself as a listener from the outline page when programmatically setting the outline's selection.

Design Critique

To me, the java editor's design initially seemed somewhat overcomplicated. However, the more I studied it I saw that the design was carefully crafted for reuse. When objects were coupled together in a seemingly unintuitive fashion, closer inspection usually showed this was done to loosen their coupling without losing an accurate representation of the context. For instance, the reasons a text editor is configured with a document provider instead of a document are varied. It promotes editor reusability (one editor can display multiple documents), and it separates responsibility: the document

provider is responsible for actually reading the data from the resource and providing it to the editor in a unified format (org.eclipse.jface.text.IDocument). This is an example of loose coupling in that it allows for flexibility regarding the origin of our document's data.

The use of EventManager functionality by WorkbenchPart (v3.2) to notify property listeners of property changes seems to me an arbitrary choice of implementation for that particular listener type. Given how widely this class is used in other classes that provide only one event type (and therefore notify only one type of listener), I suppose it makes sense to reuse this functionality in the most appropriate place (as close to the root of the sub-type hierarchy as possible and in a class that supports only one kind of listener).

The use of the Editor, Viewer, and Page hierarchies to separate responsibilities within the editor seems like a natural design choice. The viewer composes the text widget that displays the document, and provides functionality related to this purpose to the editor. The Editor and Page objects compose a viewer and provide extra functionality related to an editor (like rulers, annotations and highlighting), and to a Page (possibly display some kind of structure, like a tree, to assist in the viewing of structured data) respectively.

Now, in v3.2, the location of most features within the editor's super-type hierarchy seem reasonable, given the course of the design's evolution. One feature that seems to be completely implemented in the context of JavaEditor and possibly its subclasses is code folding. I believe that the super-structure for such a feature could have been specified in a less evolved class in such a way that could have prevented duplication of code in JavaEditor and any other plug-in that provides code folding for java editor or

any other source editor. Projection support within the viewer hierarchy occurs in a descendant of SourceViewer called ProjectionViewer, but wouldn't mainly source viewers desire it? Why not specify out the language-specific functionality in an interface such that with the aid of a third (and possibly more) helper class, all the interface implementer needs to provide is a means to determine what kind of code structures are foldable and whatever other language-specific issues I can't presently think of. The helper class(es) would provide the remaining functionality and yet be completely customizable. Since the feature did not yet exist prior to v3.0, backward compatibility shouldn't have been an issue, but doing it this way would have required the author(s) (who was working on the JDT plug-in) to have submitted a significant portion of code to the org.eclipse.ui.texteditor plug-in. However, there are definitely features that logically belong elsewhere in the hierarchy except that this would destroy backward compatibility. The semantic highlighting manager could have been designed to be abstract and couple together an abstract text editor and source viewer. Then the JDT developers would have extended it into a java-specific, concrete, semantic highlighting manager whose only new functionality would map certain language-specific keywords to an object that describes the highlighting details for that keyword. Even the outline page could have conceivably been partially implemented for all source editors (probably in AbstractDecoratedTextEditor), as long as a feature is included to deactivate it when appropriate. Unfortunately, one of the drawbacks to creating a reusable platform such as Eclipse is that dependent code can get broken in a hurry if the issue of backward compatibility is ignored.

In my opinion, the editor / overview interaction is an example of a feature that evolved nicely. In v1.0, even though all the current functionality doesn't appear to be there, it's still a reasonable design. In v2.0, I think the functionality is essentially the same as in v3.2 (updates to the outline from the editor are toggled off by default), but the logic used to achieve it seems more convoluted than necessary. Selection updates sent from the outline and the type hierarchy view part are handled in a pretty straightforward way, but having the editor send selection updates directly in cursor-listening code seems like we're using code that is much more low-level than us. The solution to this in v3.2 is to wrap the cursor-listening with selection provider functionality, so that the higher-level text editors don't have to worry about cursor changes, only selection changes. Little changes like this can accumulate to effectively manage the complexity of such interactions. Alternately, by over-generalizing code to promote reusability, depending on the circumstances, one may risk presenting an overwhelmingly complicated interface to clients (this may be more accurate in the context of enterprise-scale software development).

I enjoyed researching Eclipse's java editor. I think I bit off more than I could chew in terms of the size and complexity of the component, but once I gained enough familiarity and confidence with the design, I really got a lot out of examining its structure.

Appendix

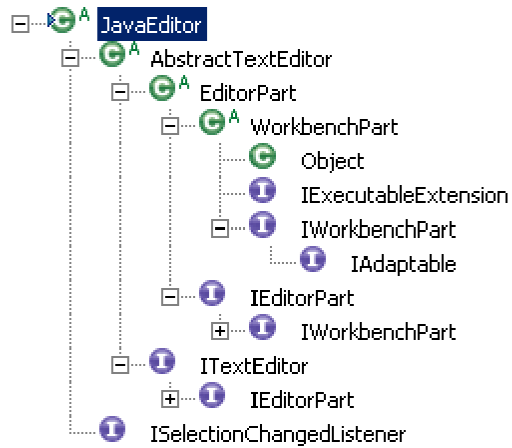

Figure 1

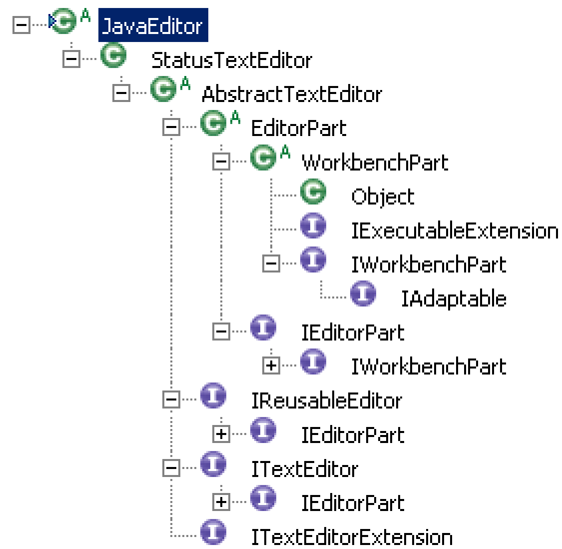

Figure 2

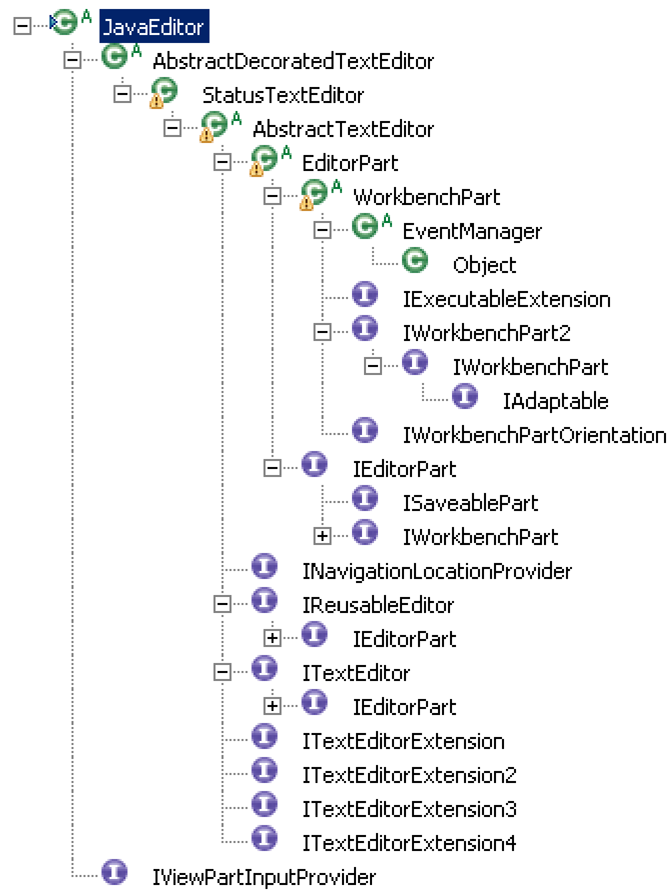

Figure 3

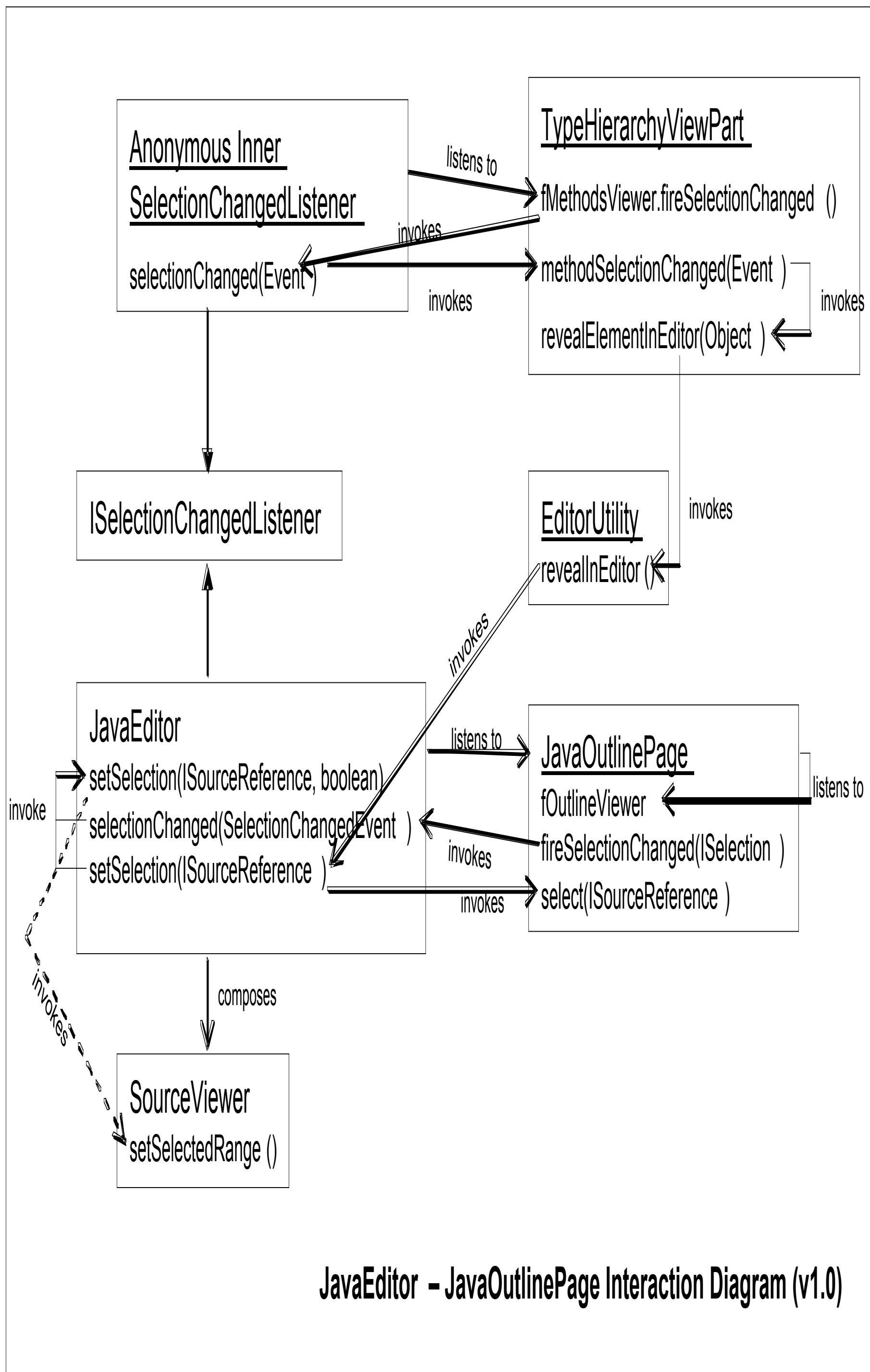

Figure 4.

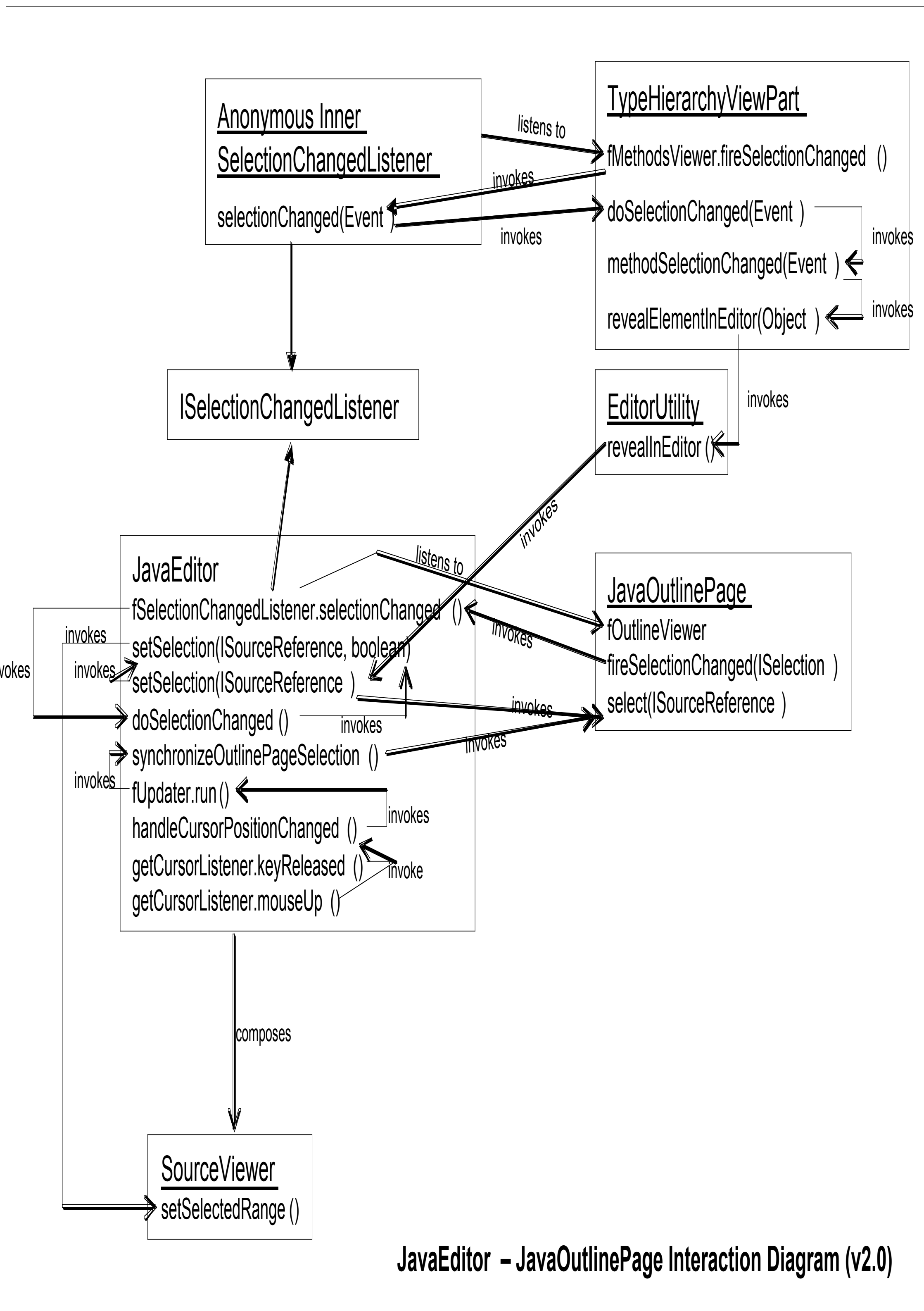

Figure 5.

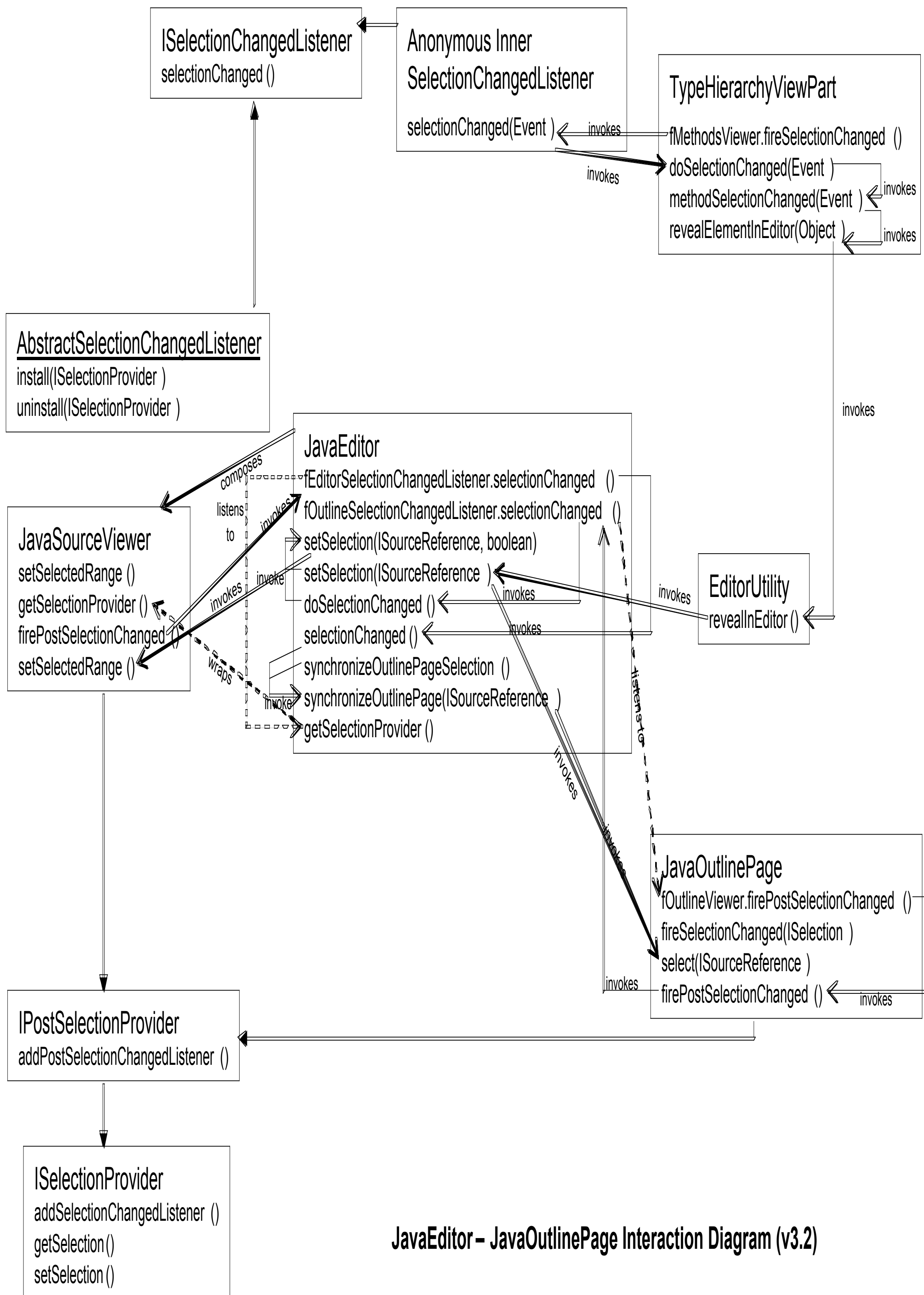

Figure 6.

Resources

1. http://archive.eclipse.org/eclipse/downloads/index.php - from here we can download archived versions of the eclipse platform. They seem to contain most of the source code in the plugins directory. It can be imported by clicking File => Import => Plug-ins and Fragments. Here you can choose the plugins directory specific to the platform version you wish to import. I probably should have used this opportunity to learn more about CVS, but I think I learned more about Eclipse by not having to fiddle with it.

2. The following Eclipse Platform v3.2 classes and interfaces:

Type Name	LOC
IAdaptable	40
IWorkbenchPart	220
IWorkbenchPart2	50
IExecutableExtension	120
IWorkbenchPartOrientation	40
EventManager	100
WorkbenchPart	460
IEditorPart	100
EditorPart	310
INavigationLocationProvider	40
IReusableEditor	30
ITextEditor	200
AbstractTextEditor	5700
text editor extension interfaces	230
StatusTextEditor	220
AbstractDecoratedTextEditor	1900
JavaEditor	3840
CompilationUnitEditor	1850
IDocument	650
IDocumentProvider	220
TypeHierarchyViewPart	1650
SemanticHighlightingManager	600
IStatus	190
JavaOutlinePage	1400
ISourceViewer	200
IPostSelectionProvider	50

All of the files that are longer than 500 lines were probably only partially read, but there were a lot of other classes and interfaces that I referred to in the course of my journey.