Penumbra: Simplifying Eclipse

Modifying the development environment for ease-of-use

FRANK MUELLER AND ANTONY L. HOSKING

Like most IDEs, Eclipse has an overwhelming arsenal of tools. To make these tools available to programmers, Eclipse relies on a collection of menus, tool bars, and views. With the addition of each menu or toolbar, however, the appearance (perspective) of Eclipse grows more complex, making it increasingly difficult to locate the features you consider useful.

To address this growing complexity, we present here an Eclipse plug-in called "Penumbra" (http://www.cs.purdue.edu/s3/projects/eclipse/downloads.html). Designed with the goal to simplify the full-blown environment, Penumbra has been used by computer science students at Purdue University for the last two semesters with great success. (All implementation details described are with respect to Eclipse 2.1.x.)

Frank is a graduate student and Antony an associate professor in the Department of Computer Science at Purdue University. They can be contacted at mueller@cs.purdue.edu and hosking@cs.purdue.edu, respectively.

The fully qualified name of the plug-in is edu.purdue.penumbra. It consists of 60 classes organized in seven packages. Figure 1 shows Penumbra's overall structure. The two most important files in the plug-in are plugin.xml and PenumbraPlugin.java, contained in the main package. Plugin.xml defines all properties of the plug-in. These properties include a listing of the plug-ins required for Penumbra to run properly, as well as detailed declarations of all extensions that the Penumbra plug-in will add to Eclipse. Consider Example 1, which specifies that two views should be added—the Package Explorer and the hierarchy view. Furthermore, it defines the classes that need to be instantiated to display the views, as well as a name and a logo that can be used to refer to them in menus and toolbars. In a similar manner, all other extensions the Penumbra plug-in adds to Eclipse are declared in plugin.xml. The class *PenumbraPlugin* is used as an interface to make all extensions accessible to Eclipse. It contains just a few lines of source code but is crucial for the interaction between Penumbra and Eclipse. Furthermore, *SavitchIn*, a collection of methods to simplify user input presented by Walter Savitch in *Java: An Introduction to Computer Science and Programming,* Third Edition (Prentice-Hall, 2004), is packaged with Penumbra.

A perspective in Eclipse defines the menus, toolbars, and views users see. As such, the Penumbra perspective is undoubtedly one of the most essential parts of the plug-in. The Penumbra perspective is based on the Java perspective, which is the default programming perspective in Eclipse. Instead of designing a perspective from scratch, we adapted the Java perspective found in org.eclipse.jdt.internal.ui. The core of the Penumbra perspective is the class *PenumbraPerspectiveFactory*. When instantiated, a call is made to the method *createInitialLayout*. This method defines where on the screen certain views will be opened. In the case of Penumbra, it opens the Penumbra Package Explorer on the top-left side, and the Penumbra hierarchy view on the bottom left. Furthermore, the console is placed on the bottom. The Java editor is created at this time. It is instantiated once a Java source file is opened. Notice that the perspective factory does not let you modify the menus and toolbars visible to users. Menus and toolbars are not directly connected to a perspective, but rather belong to a particular view. For example, if the Java editor is opened for the first time, it creates menus that let users build and run the code. The Eclipse plug-in structure was designed with the idea that users would only want to add functionality but not remove any of it. This

> "Running projects in Eclipse is, by default, more complicated than in a UNIX console"

design flaw has been addressed in Eclipse 3.0. However, we had to find a somewhat cumbersome solution to work around this design flaw in Eclipse 2.1.x. Again, when a view is opened for the first time, it will create menus and toolbars that are associated with this view. These menus and toolbars are called "ActionSets" in Eclipse. To add an ActionSet, the view has to register it with the current perspective. Unfortunately, it is impossible to keep the views from adding their ActionSets without having to modify them, which would lead to an unnecessary duplication of many classes.

Our alternative solution was to remove these ActionSets right after they had been added. To accomplish this, we designed a class called *PenumbraSwitchPerspectiveAction*. When instantiated, this class switches to the Penumbra perspective. It then scans the array of ActionSets and removes all unwanted menus and toolbars. When Penumbra is installed, it sets Penumbra to be the default perspective and makes a call to *PenumbraSwitchPerspectiveAction*. All subsequent openings of the views will not add the lost ActionSets to the Penumbra perspective, so users will not notice the trick. Besides removing numerous menus and toolbars, a menu called "Penumbra" was added, along with a toolbar with shortcuts to these menu functions. The Penumbra menu contains all functionality that was added by Penumbra or functionality that was otherwise hidden in submenus in the Java Perspective. For example, there is an action to format the source code.

Next, we simplified the Package Explorer, which has all functionality required but is very complex. It contains menus and filters that novice programmers will not touch, and so only add to the distraction of the overall perspective. Hence, we chose to adapt the default Package Explorer found in org.eclipse.jdt.internal.ui.packageview. We modified the view such that all of its menus were disabled and, furthermore, preset the filters so users can see only the content in their projects. For example, by default, the Java Package Explorer shows all library references, which adds confusion and makes it harder for users to understand the files they are responsible for.

Part of the simplification process was also the need to redesign the way users set up and run projects. As described earlier, it was a design goal to let users easily set up projects and run them, without having to go through many steps. By default, users need to set up a project, then they need to create a class, and they need to remember to include a main method. Furthermore, the CLASSPATH has to be set up to include *SavitchIn*. While these steps seem trivial to experienced programmers, they do, in fact, cause problems for novice programmers. We wrote a project creation wizard that merely asks users to input a name, and the wizard then sets up a project that contains a class with the same name. The class automatically contains a main method, and the CLASSPATH for the project is set to include *SavitchIn*. The class *PenumbraNewProjectCreationWizard* makes use of the Java project and class creations wizards contained in Eclipse. It first creates a project with the CLASSPATH containing *SavitchIn*. It then passes the newly created project to the default wizard for class creation, which is then fed the class name and the request for a main method. The class wizard is automatically executed, and the new project appears in the Package Explorer.

```
<extension
      point="org.eclipse.ui.views">
   <view
      name="Penumbra Package Explorer"
      icon="icons/penumbraLogo.jpg"
      class="edu.purdue.penumbra.ui.PenumbraPackageExplorer"
      id="edu.purdue.penumbra.ui.PenumbraPackageExplorer">
   </view>
   <view
      name="Project Hierarchy"
      icon="icons/penumbraHierarchy.gif"
      class="edu.purdue.penumbra.ui.hierarchy.PenumbraHierarchyView"
      id="edu.purdue.penumbra.ui.TypeHierarchy">
   </view>
</extension>
```

Example 1: *A plugin.xml sample.*

Running projects in Eclipse is, by default, more complicated than in a UNIX console. This is because Eclipse lets you run programs — not just as Java applications. For instance, projects may be executed as applets or as plug-ins in another instance of Eclipse. However, this requires users to specify exactly the environment in which they would like to run the program. Many properties can and must be specified to run programs. In our opinion, users should only have to select a project or class containing a main method and click a single button to run it. However, simplifying the procedure this much would cause other problems. For instance, what if users want to pass command-line arguments? What if they would like to run an applet? Thus, we selected a subset of cases and made them available in a submenu; see Figure 2.

Whenever users select an element in the Penumbra Package Explorer in either the Penumbra hierarchy view or editor window, the selection is noted by the *PenumbraSelectionAction* class. If users choose to click any of the run buttons, it spawns the responsible *PenumbraRunAction* (depending on the menu entry chosen), which attempts to determine which class the user would like to run. If users are working in an editor with a main method, the choice is easy and Penumbra simply executes this class. However, if the user last selected a project that contains multiple classes with main methods, users are prompted to select the class they would like to execute. Users do not have to make any further specifications. If they select the application to be executed with command-line arguments, they are prompted to enter these arguments in a pop-up window. The arguments will be saved in the Eclipse registry so that they do not have to be reentered at each execution. Figure 3 shows the resulting Penumbra perspective.

Project Checkout via CVS

Another goal of Penumbra was to provide a mechanism that allows for easy distribution of projects, containing skeleton files

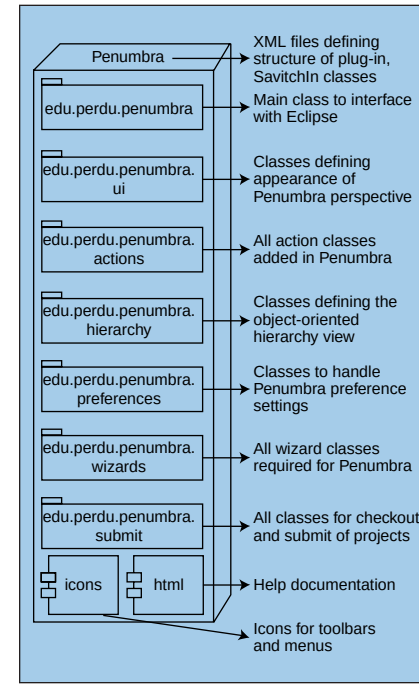

Figure 1: *Package structure of Penumbra.*

defining methods that must be implemented in the current project, as well as starter code and comments. Concurrent Versions System (http://www.cvshome.org/) seemed to be the ideal solution. CVS can be used to distribute, receive, and manage software for a large group of developers. Moreover, Eclipse is already equipped with a CVS plug-in. A problem is that the plug-in provides more functionality than we need, and it is too complex to be used by novice users. A seamless solution should leave users completely unaware of the underlying mechanism used to check out and submit projects. Furthermore, if it is necessary to change the mechanism, it should not affect the way it appears to the students.

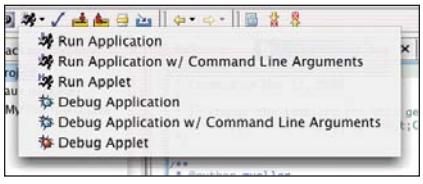

Figure 2: *Penumbra Run menu.*

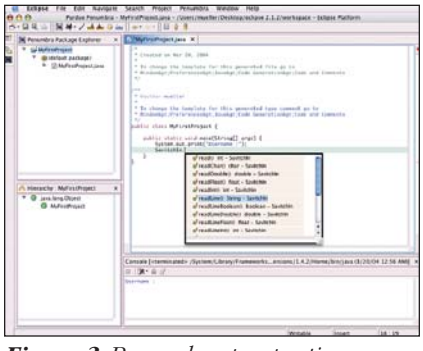

Figure 3: *Penumbra perspective.*

Unlike open-source projects where CVS repositories are used to foster collaboration between team members, we had to design a repository that would permit each user to retrieve and submit only their own projects. Furthermore, the repository has to be accessible to the administrative group (instructors and teaching assistants) to be able to grade the projects. It should also be possible to turn off submission after project deadlines have passed. Figure 4 illustrates the layout we designed. The entire repository is owned by the administrative group, but accessible to all others. Inside the repository, we create a module for each project, which in turn contains modules for each student and an additional module called "skeleton." The project modules are, by default, accessible to anyone but permissions can be modified to disallow access after the deadline has passed. The modules for each student are owned by the students themselves to avoid anyone else gaining access to their source code. The skeleton folder holds the skeleton code that is made available to the students at the beginning of the project. We provide UNIX shell scripts that populate the CVS repository automatically and let instructors easily post new projects. The script to populate the repository has to be run by root, as only instructors have the ability to create folders that belong to the students.

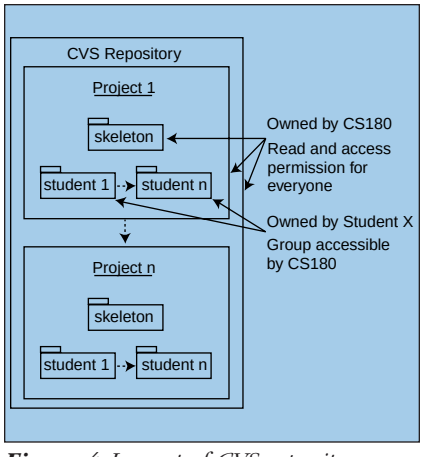

Figure 4: *Layout of CVS repository.*

The use of CVS as a means to check out and submit projects was to be included in a seamless manner. Users should

not have to navigate through the CVS repository to find the project they need to check out or submit. Therefore, we developed new wizards that are based on the default Import and Export wizards, but instead of reading or writing to the file system, they connect to a CVS server. For Penumbra to be able to set up the connection to the CVS repository and use it without the student explicitly knowing about it, we had to gather information from the students, such as user name and the location of the CVS repository. To do so, we created a preference page. The preference page in Figure 5 collects information from users that lets Penumbra automatically set up a connection and use it. By default, all information is set to be used in CS180 (Purdue's introductory programming class), except for the username. Once a user enters the username and clicks OK, Penumbra sets up a new CVS location. Furthermore, the name of the location and the other preferences are stored in the Eclipse registry so that it can be used subsequently without consulting users.

To check out a project, users must select the appropriate option from the Penumbra menu or tool bar. This opens the *PenumbraCheckoutWizard*, which connects to the CVS location known to Penumbra and lists all project names (see Figure 6). Users now need to select a project and click OK to start the checkout process. First, Penumbra checks if the project is available, which it does by looking for files in the skeleton folder of the specified project. If there are none, the project is not yet available for check out. Next, the wizard determines if the user has previously checked out the project. This is necessary because users can submit their code and check it out at another location. Hence, users receive the latest version of their project code unless they have not yet submitted anything. If this is the first check out the user has performed on the project, the wizard downloads the code contained in the skeleton directory of the respective project. However, simply obtaining the skeleton code is not sufficient because at the next commit operation, CVS would attempt to overwrite the code in the skeleton directory. After checking out the skeleton code, the checkout wizard disconnects the project from the skeleton module and reimports it as the first version of the students code in his or her project module. All this happens as one uninterruptible step in a matter of a few seconds. Due to the fact that several errors can occur during checkout (network failure, CVS server failure, and so on), the wizard verifies each step to provide students with a detailed error message in case a problem occurs. Once the project is checked out, it appears in the Penumbra Package Explorer and the Penumbra hierarchy view, where users can select the files to open and start working immediately.

Project Submission Via CVS and *turnin*

We developed two mechanisms to submit projects via Penumbra. The default mechanism of choice is submission via CVS. However, there is also the option to submit projects by passing it to a command such as *turnin*. The submission method can be selected in the Penumbra preferences page. Submission via CVS offers greater convenience for a number of reasons. An application such as *turnin* is usually bound to a particular operating system. Consequently, *turnin* only works on UNIX machines whereas using CVS lets users submit the projects from virtually anywhere independent of the operating system they are running. Second, using CVS lets them retrieve previous versions of their project source code. We have integrated a feature that lets users easily view and retrieve previous versions of their source files. Selecting the appropriate menu option yields a list of versions available in the CVS repository, depending on the file selected. Another double-click downloads and opens that version. This feature lets users easily recover source code they previously deleted.

When selecting Submit Project from the Penumbra menu, the *PenumbraExportWizard* class is instantiated. It asks users

to select one of their projects. If the project has been checked out or submitted previously, Penumbra automatically knows where to submit the project. However, if the project has not been checked out or submitted before, users must select from a list of names the user would like to submit the project under. The export wizard then connects to the CVS repository in order to commit the user's project. To save storage space in the repository, only source files will be submitted. All compiled files are filtered out by the wizard. Shortly after the first version of Eclipse was in use, we discovered that users wanted additional confirmation that their projects submitted correctly. By default, we have Penumbra display a message whether or not the submission has been successful. However, users would also like to confirm the dates and times that their files were submitted. Hence, we implemented a feature that would connect to the CVS server and display the information for all files submitted for the selected projects.

Submission via *turnin* works very similarly from the users perspective. It does not offer the benefits of CVS, but it is easier to administer than a CVS repository. The same wizard is invoked and asks the student to select the project to be submitted. The export wizard then filters out the sources file and submits them using the specified *turnin* command. If selected on the Penumbra preference page, the files are zipped up and passed to *turnin*. At the end of the process, a window displays the *turnin* output to let users judge if *turnin* was or was not successful. Since the output of such *turnin* commands differ, it is impossible to provide one solution that allows Penumbra to judge if submission was or was not successful.

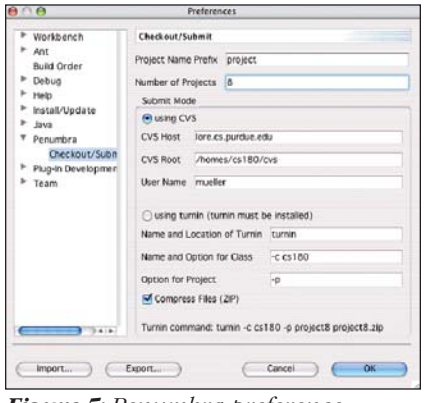

Figure 5: *Penumbra preference settings.*

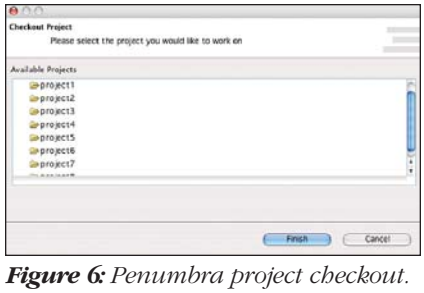

Figure 6: *Penumbra project checkout.*

Penumbra Hierarchy View

The Penumbra hierarchy view was designed to help users better understand the object-oriented concepts of the Java programming language. The goal was to provide users with a view that would visualize the hierarchy of classes and interfaces in a project, but that would also allow for interaction. We chose a tree structure to display the class hierarchy. Throughout Eclipse, views use tree structures to display the contents of projects, packages, and classes. To more seamlessly integrate the Penumbra hierarchy view, we adapted one of these tree structures. By default, Eclipse contains a type hierarchy viewer, which is used to display member methods of classes. This view, which can be found in org.eclipse.jdt.internal.ui.typehierarchy, provided an excellent basis for what we needed to design.

We modified the view so as to display the hierarchy of one project at a time. If a different project or an element of a project is selected in the Penumbra Package Explorer, it causes the hierarchy view to refresh its contents. The package explorer will pass the project root of the selected element to the hierarchy view. The hierarchy view, in turn, recursively traverses the children of the root and builds a tree of class nodes. This tree is then annotated with icons and name labels to be displayed. The fully qualified name of all objects will be displayed to show if a class or interface is part of the default package, another package within the project, or a completely different package. All classes and interfaces, even if they are from different packages, are shown in the same hierarchy, to help the users visualize class dependencies across packages. Figure 7 shows an example hierarchy view. Notice that due to the fact that classes can implement multiple interfaces, some classes will be shown in multiple branches. The nodes in the hierarchy are decorated with the default Eclipse icons.

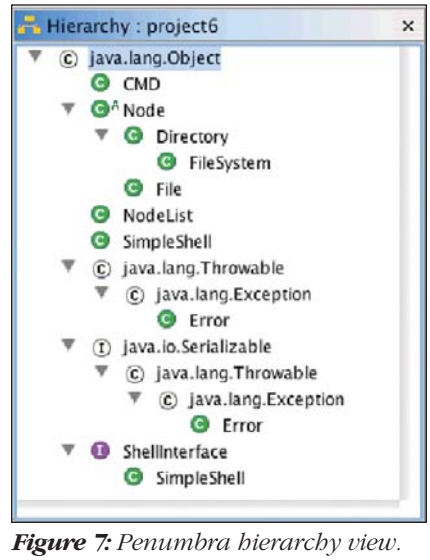

Figure 7: *Penumbra hierarchy view.*

Letting users experiment with the objects in the hierarchy view is key to improving their understanding of object-oriented programming. Eclipse provides the facilities to allow drag-and-drop features. We developed the *PenumbraHierarchyDragAndDropAdapter*, which is the heart of this interactive feature. When a node of the hierarchy tree is selected, dragged, and dropped somewhere in the hierarchy view, the adapter is invoked. The adapter then determines the type of the source and the target. If a source class is dropped onto a class or an interface, it extends or implements that object. If the target is the empty space of the Penumbra hierarchy view, all extensions or implementations are removed from the source class. Similarly, a source interface may extend a target interface. The underlying source files are updated accordingly, all necessary import statements will be added, and the hierarchy view is refreshed to display the change in the hierarchy. Furthermore, users can click on the objects in the hierarchy to open and edit the appropriate source files.

Conclusion

Our work shows that Eclipse can be modified to be an excellent development environment for introductory programming courses. The simplified Penumbra perspective, as well as the added features such as check out and submission, have increased acceptance by students. Furthermore, the development of features such as the Penumbra hierarchy view increases the pedagogical support of Eclipse. Eclipse/Penumbra can be made an even more suitable environment through the implementation of the ideas presented here, although we need to be careful to retain the simplicity of Penumbra while adding new features.

Acknowledgments

Thanks to Sarah Caruthers for her ideas on Penumbra. She also helped in testing Penumbra. Our work on Penumbra was sponsored by an Eclipse Innovation Grant from IBM.