Getting started with the Eclipse Platform

Using Eclipse plug-ins to edit, compile, and debug your app

Level: Introductory

David Gallardo (david@gallardo.org), Software consultant

01 Nov 2002

This article gives you an overview of the Eclipse Platform, including its origin and architecture. Starting with a brief discussion about the open source nature of Eclipse and its support for multiple programming languages, this article then demonstrates the Java™ development environment with a simple program example. This article also surveys some of the software development tools available as plug-in extensions, and demonstrates a plug-in extension for UML modeling.

What is Eclipse?

Eclipse is an open source, Java-based, extensible development platform. By itself, it is simply a framework and a set of services for building a development environment from plug-in components. Fortunately, Eclipse comes with a standard set of plug-ins, including the Java Development Tools, or JDT for short.

While most users are quite happy to use Eclipse as a Java IDE, its ambitions do not stop there. Eclipse also includes the Plug-in Development Environment (PDE), which is mainly of interest to software developers who want to extend Eclipse, since it allows them to build tools that integrate seamlessly with the Eclipse environment. Because everything in Eclipse is a plug-in, all tool developers have a level playing field for offering extensions to Eclipse and providing a consistent, unified integrated development environment for users.

This parity and consistency aren't limited to Java development tools. Although Eclipse is written in the Java language, its use isn't limited to the Java language; for example, plug-ins are available or planned that include support for programming languages such C/C++, COBOL, and Eiffel. The Eclipse framework can also be used as the basis for other types of applications unrelated to software development, such as content management systems.

The premiere example of an Eclipse-based application is IBM's WebSphere Studio Workbench, which forms the basis of IBM's family of Java development tools. WebSphere Studio Application Developer, for example, adds support for JSPs, servlets, EJBs, XML, Web services, and database access.

Eclipse is open source

Open source software is software that is released with a license intended to ensure that certain rights are granted to users. The most obvious right, of course, is that the source code must be made available so that users are free to modify and redistribute the software. This protection of users' rights is accomplished with a device called a *copyleft*: the software license claims copyright protection and prohibits distribution unless the user is granted these rights. The copyleft also requires that any re-distributed software be covered by the same license. Since this, in effect, stands the purpose of copyright on its head -- using the copyright to grant rights to the user rather than reserve them for the developer of the software -- copyleft is often described as "all rights reversed."

Much of the fear, uncertainty, and doubt that have been spread about open source software involves the so-called viral nature of some copyleft licenses -- the idea that if you use open source software as part of a program you develop, you will lose your intellectual property because the license will "infect" the proprietary parts you develop. In other words, the license may require that all software bundled with the open source software, including any newly developed software, must be released under the same license. While this may be true of the most well-known copyleft license, the GNU General Public License (under which Linux, for example, is released), there are other licenses that provide a better balance between commercial and community concerns.

The Open Software Initiative is a non-profit organization that explicitly defines what open source means and certifies licenses that meet its criteria. Eclipse is licensed under the OSI-approved Common Public License (CPL) Version 1.0, which "is intended to facilitate the commercial use of the Program ..." (for a link to the complete text of the Common Public License v1.0, see Resources later in this article).

Developers who create plug-ins for Eclipse or who use Eclipse as the basis for a software development application are required to release any Eclipse code that they use or modify under the CPL, but they are free to license their own additions in any way they like. Proprietary code bundled with software from Eclipse does not need to be licensed as open source, and the source code does not need to be made available.

Although most developers will not be using Eclipse to develop plug-ins or to create new products based on Eclipse, the open source nature of Eclipse is important beyond the mere fact that it makes Eclipse available for free (and despite the fact that a commercial-friendly license means that plug-ins can cost money). Open source encourages innovation and provides incentives for developers, even commercial developers, to contribute code back to the common open source code base. There are a number of reasons for this, but perhaps the most essential is that the more developers contribute to the project, the more valuable the project becomes for everyone. As the project becomes more useful, more developers will use it and create a community around it, like those that have formed around Apache and Linux.

Who is Eclipse?

The Eclipse.org Consortium manages and directs Eclipse's ongoing development. Created by IBM after reportedly spending $40 million developing Eclipse and releasing it as an open source project, the Eclipse.org Consortium recruited a number of software tool vendors including Borland, Merant, Rational, RedHat, SuSE, TogetherSoft, and QNX. Other companies have since joined, including Hewlett Packard, Fujitsu, and Sybase. These companies each appoint a representative to a Board of Stewards that determines the direction and scope of the Eclipse project.

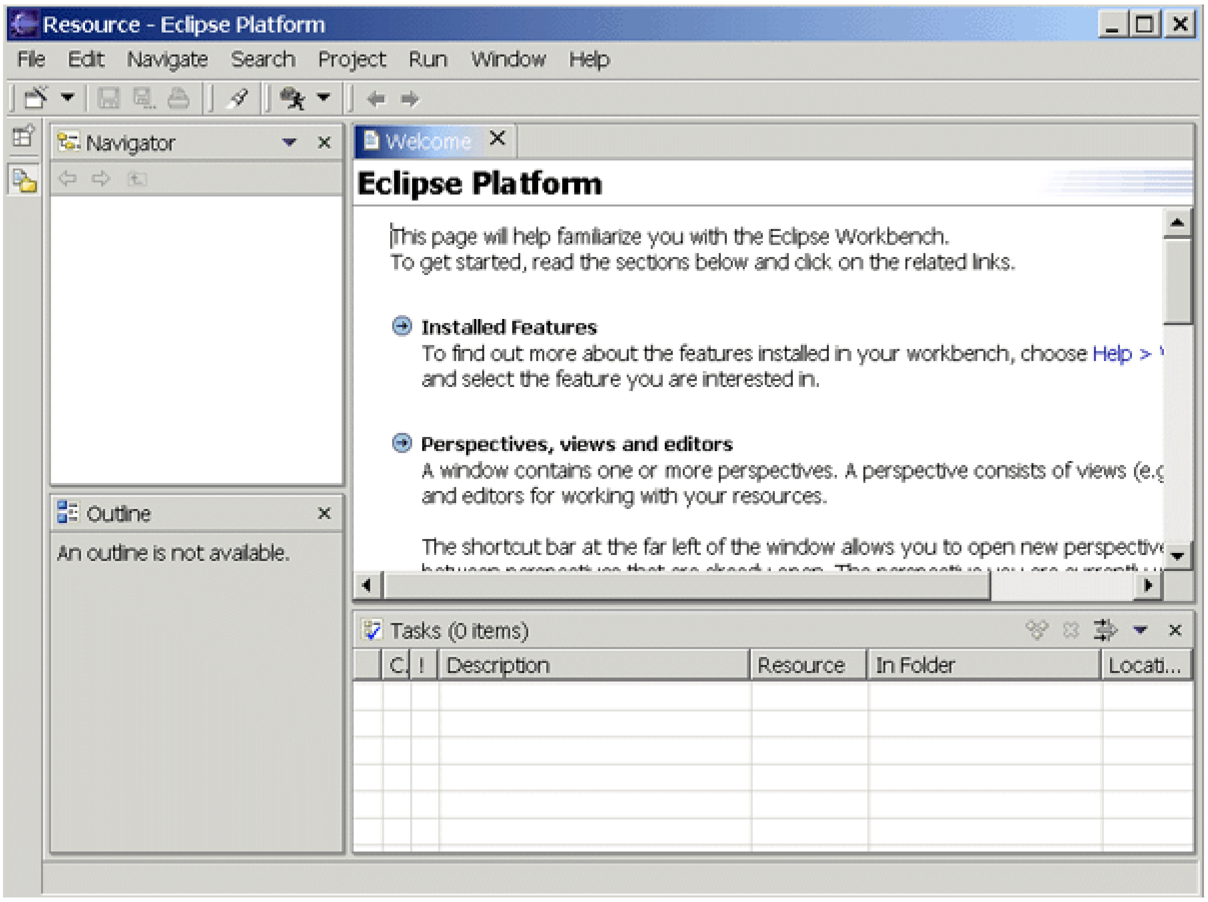
At the highest level, the Project Management Committee (PMC) manages the Eclipse project. The project is divided into subprojects and each of these has a leader. Large subprojects are divided into components, and each of these also has a leader. At present, most of these managerial roles are filled by people from the IBM subsidiary that originally developed Eclipse, Object Technology International (OTI), but as an open source project, anyone is welcome to participate. Responsibility for any specific piece is earned by contributing to the project.

Now that we've looked at some of the theory, history, and politics behind Eclipse, let's take a look at the product itself.

The Eclipse Workbench

The first time you open Eclipse, you see the following welcome screen:

Figure 1. The Eclipse Workbench


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
Welcome
Eclipse Platform
Installed Features
Perspectives, views and editors
Outline
Tasks (0 items)
C ! Description Resource In Folder Locati...

This page will help familiarize you with the Eclipse Workbench.
To get started, read the sections below and click on the related links.

To find out more about the features installed in your workbench, choose Help >
and select the feature you are interested in.

A window contains one or more perspectives. A perspective consists of views (e.g
and editors for working with your resources.

The shortcut bar at the far left of the window allows you to open new perspective

An outline is not available.

The Eclipse Workbench consists of several panels known as *views*, such as the Navigator view at the top left. A collection of panels is called a *perspective*. The default perspective is the Resource Perspective, which is a basic, generic set of views for managing projects and viewing and editing files in a project.

The **Navigator view** lets you create, select, and delete projects. The panel to the right of the Navigator is the **editor area**. Depending on the type of document selected in the Navigator, an appropriate editor window opens here. If Eclipse does not have an appropriate editor registered for a particular document type (for example, a `.doc` file on a Windows system), Eclipse will try to open the document using an external editor.

The **Outline view**, below the Navigator, presents an outline of the document in the editor; the precise nature of this outline depends on the editor and the type of document; for a Java source file, the outline displays any declared classes, attributes, and methods.

The **Tasks view** gathers information about the project you are working on; this can be information generated by Eclipse, such as compilation errors, or it can be tasks that you add manually.

Most of the other features of the workbench, such as the menu or the toolbar, should be similar to those of familiar applications. One convenient feature is a toolbar of shortcuts to different perspectives that appears on the left side of the screen; these vary dynamically according to context and history. Eclipse also comes with a robust help system that includes user guides for the Eclipse workbench and the included plug-ins such as the Java Development Tools. It is worthwhile to browse through the help files at least once to see the range of available options and to better understand the flow of Eclipse.

To continue this short tour of Eclipse, we'll create a project in the Navigator. Right click in the Navigator view, and then select **New=>Project**. When the New Project dialog box appears, select Java on the left. Standard Eclipse has only one type of Java project, named "Java Project". If there were a plug-in installed to supply support for JSPs and servlets, we would see an additional option here for Web applications. For now, select Java Project, enter "Hello" when prompted for the project name, and then press Finish.

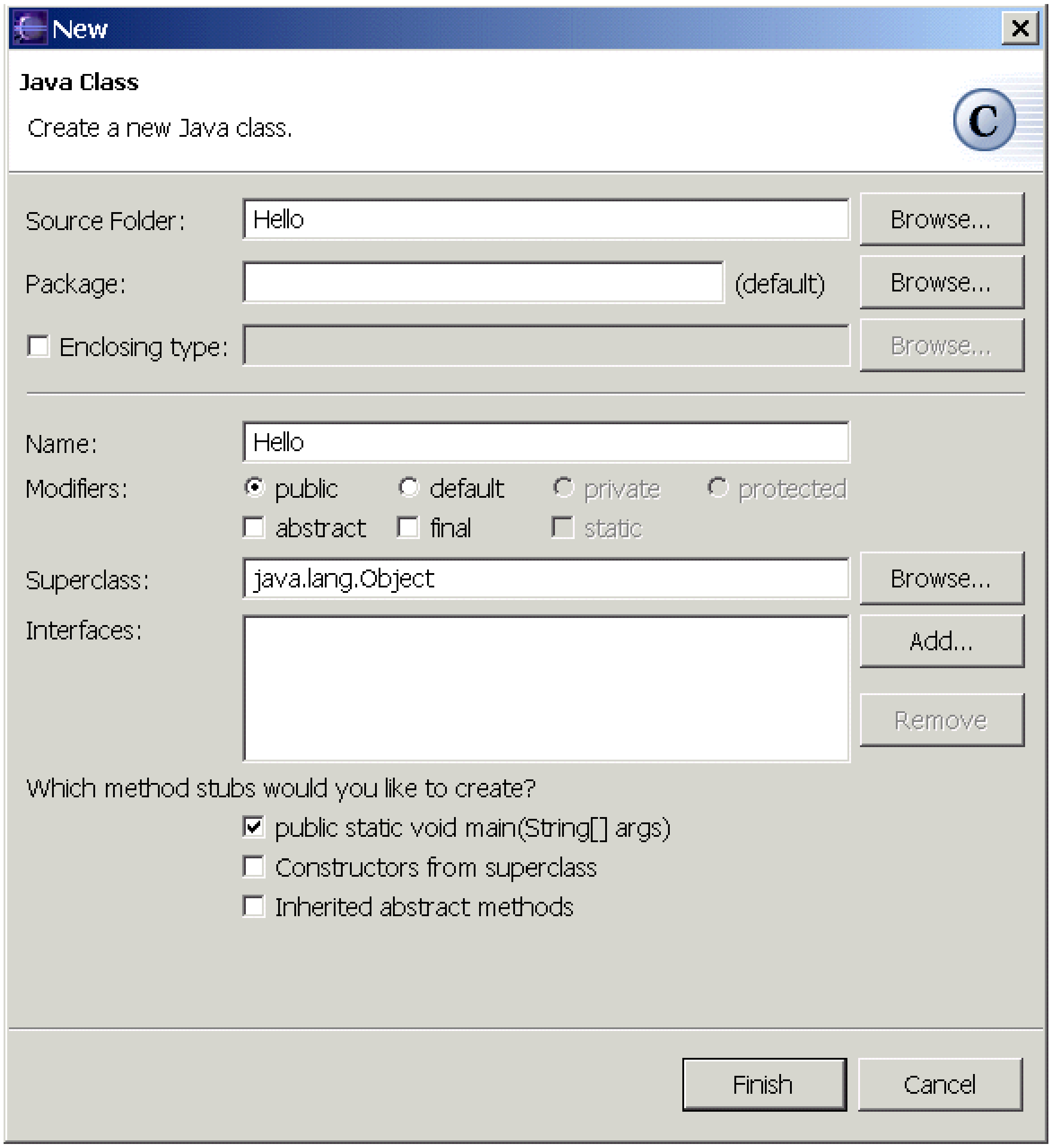
Next, we'll take a look at the Java perspective. Depending on how you like to manage your screen, you can either change the perspective in the current window by selecting **Window=>Open Perspective=>Java** or you can open a new window by selecting **Window=>New Window** and then selecting the new perspective.

The Java perspective, as you might expect, has a set of views that are better suited for Java development. One of these includes, as the top left view, a hierarchy containing various Java packages, classes, jars, and miscellaneous files. This view is the called the **Package Explorer**. Also notice that the main menu has expanded **--** two new menu items have appeared: Source and Refactor.

The Java Development Environment (JDE)

To try out the Java development environment, we'll create and run a "Hello, world" application. Using the Java perspective, right-click on the "Hello" project, and select **New=>Class** as shown in Figure 2. In the dialog box that appears, type "Hello" as the class name. Under "Which method stubs would you like to create?" check "public static void main(String[] args)", and then press Finish.

Figure 2. Creating a new class in the Java perspective



New

Java Class

Create a new Java class.

Source Folder: Hello

Package: (default)

[ ] Enclosing type:

Name: Hello

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

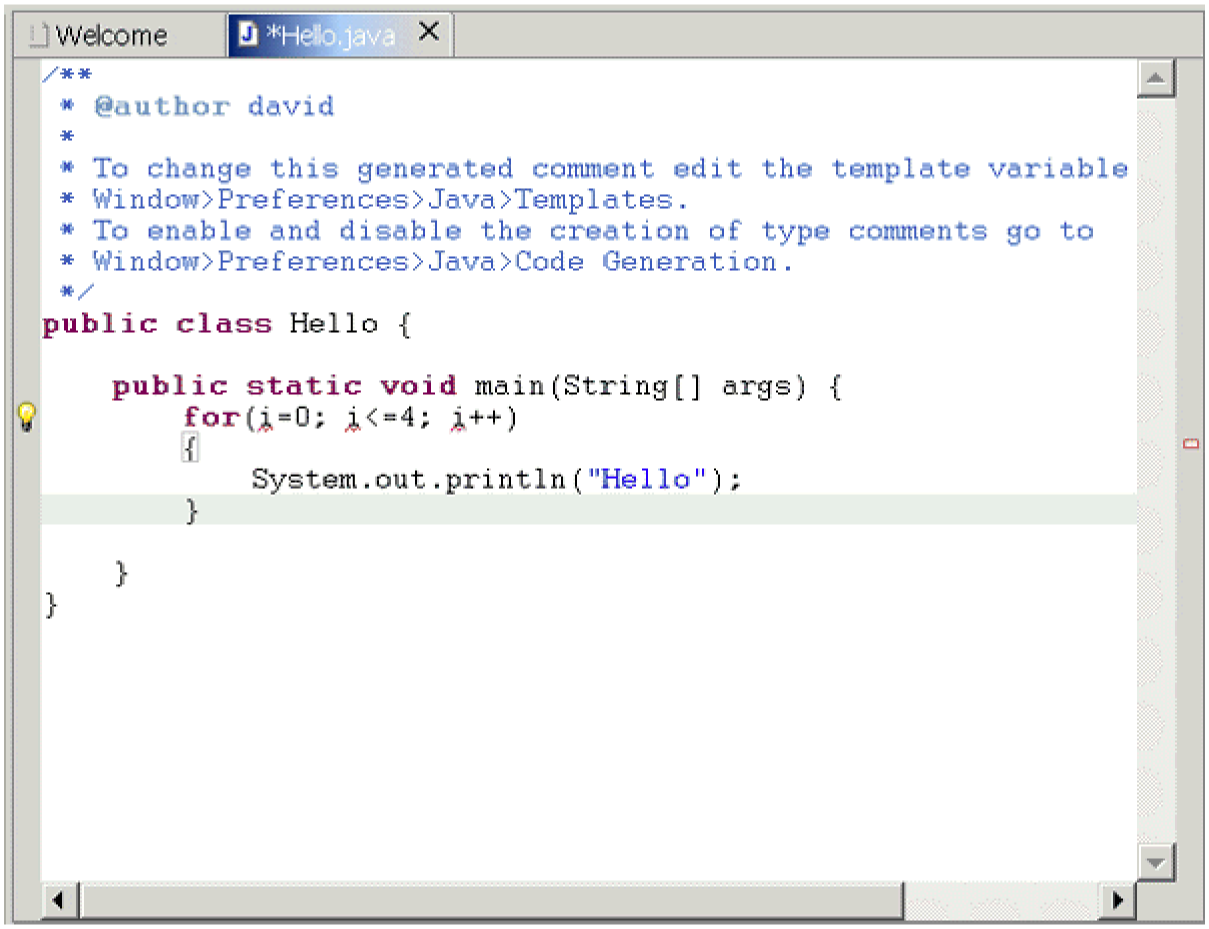
[✓] public static void main(String[] args)

[ ] Constructors from superclass

[ ] Inherited abstract methods

This will create a `.java` file with a `Hello` class and an empty `main()` method in the editor area as shown in Figure 3. Add the following code to the method (note that the declaration for `i` has been deliberately omitted):

Figure 3. The Hello class in the Java editor



```
/**
 * @author david
 *
 * To change this generated comment edit the template variable
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class Hello {

    public static void main(String[] args) {
        for(i=0; i<=4; i++)
        {
            System.out.println("Hello");
        }

    }
}
```

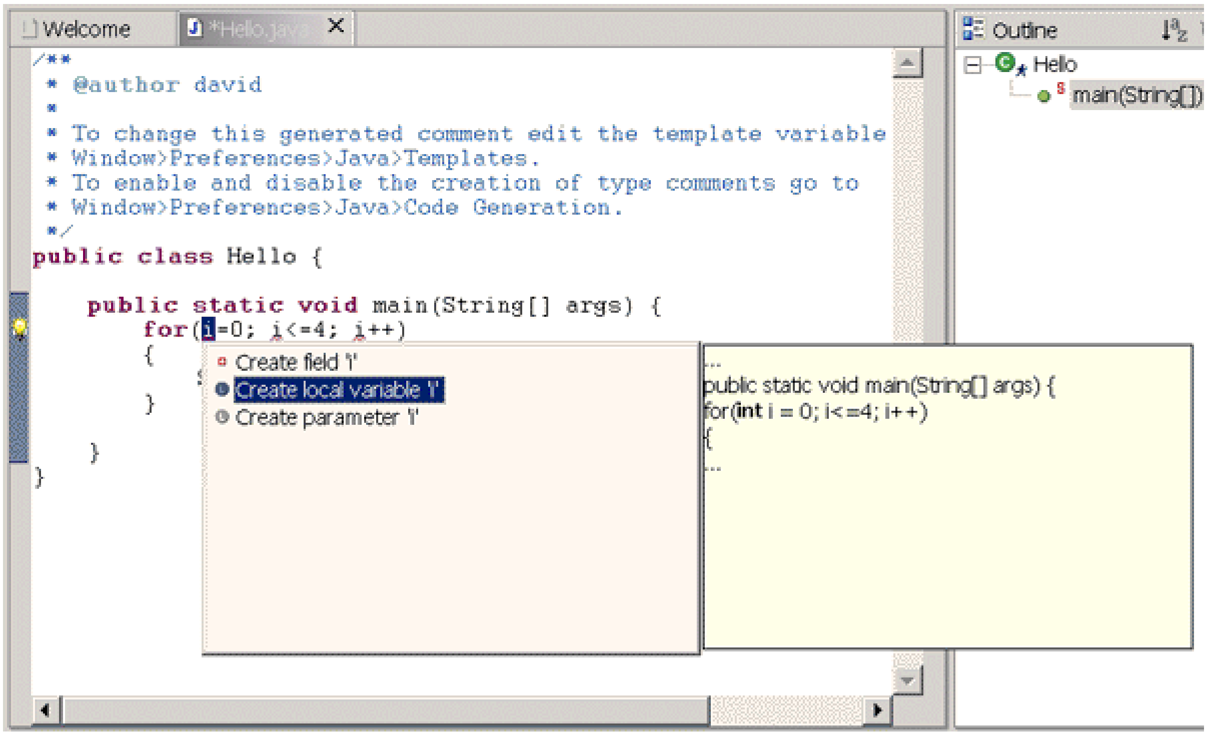
You'll notice some of the Eclipse editor's features as you type, including syntax checking and code completion. In version 2.1 (which I previewed by downloading build M2), when you type an open parenthesis or double quote, Eclipse will provide its partner automatically and place the cursor inside the pair.

In other cases, you can invoke code completion by pressing Ctrl-Space. Code completion provides a context-sensitive list of suggestions selectable by keyboard or mouse. The suggestions can be a list of methods specific to a particular object, or a code snippet to expand, based on various keywords like `for` or `while`.

Syntax checking depends on incremental compilation. As you save your code, it is compiled in the background and checked for syntax errors. By default, syntax errors are underlined in red, and a red dot with a white "X" appears in the left margin. Other errors are indicated with a light bulb in the editor's left margin; these are problems that the editor might be able to fix for you -- a feature called Quick Fix.

The code example above has a light bulb next to the `for` statement, because the declaration for `i` has been omitted. Double-clicking on the light bulb will bring up a list of suggested fixes. In this case, it will offer to create a class field `i`, a local variable `i`, or a method parameter `i`; clicking on each of these suggestions will display the code that would be generated. Figure 4 shows the list of suggestions and the code it suggests for a local variable:

Figure 4. Quick Fix suggestions



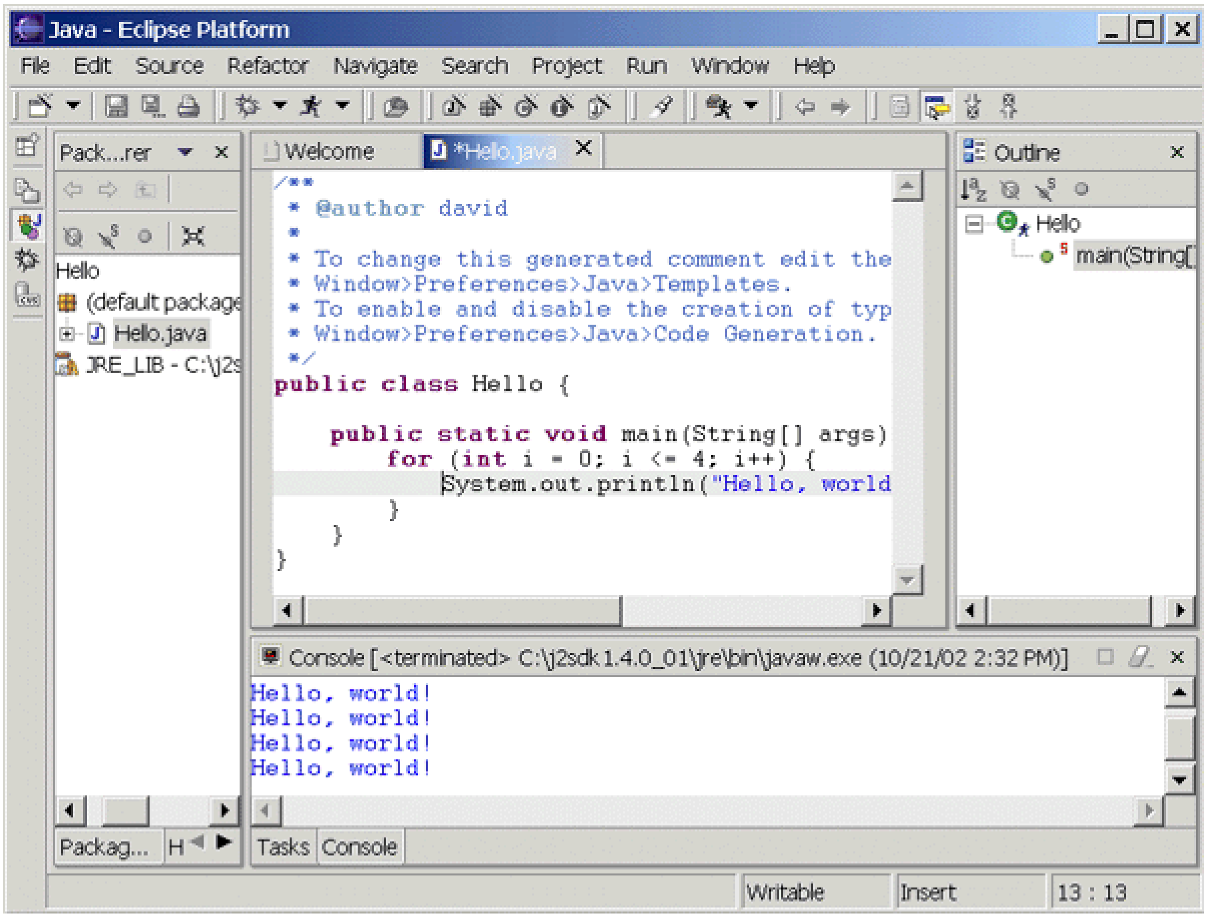
```
/**
 * @author david
 *
 * To change this generated comment edit the template variable
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class Hello {

    public static void main(String[] args) {
        for(i=0; i<=4; i++)
        {

        }

    }
}
```

```
...
public static void main(String[] args) {
for(int i = 0; i<=4; i++)
{
...
```

Double-clicking on the suggestion inserts the code in the proper location in the code.

Once the code compiles without error, you can execute the program by selecting Run from the Eclipse menu. (Note that there is no separate compilation step, because compilation takes place as you save the code. If your code has no syntax errors, it's ready to run.) A Launch Configurations dialog box appears, with appropriate defaults; press the Run button at the bottom right. A new tabbed panel appears in the lower panel (the Console), displaying the program's output as shown in Figure 5:

Figure 5. Output from the program



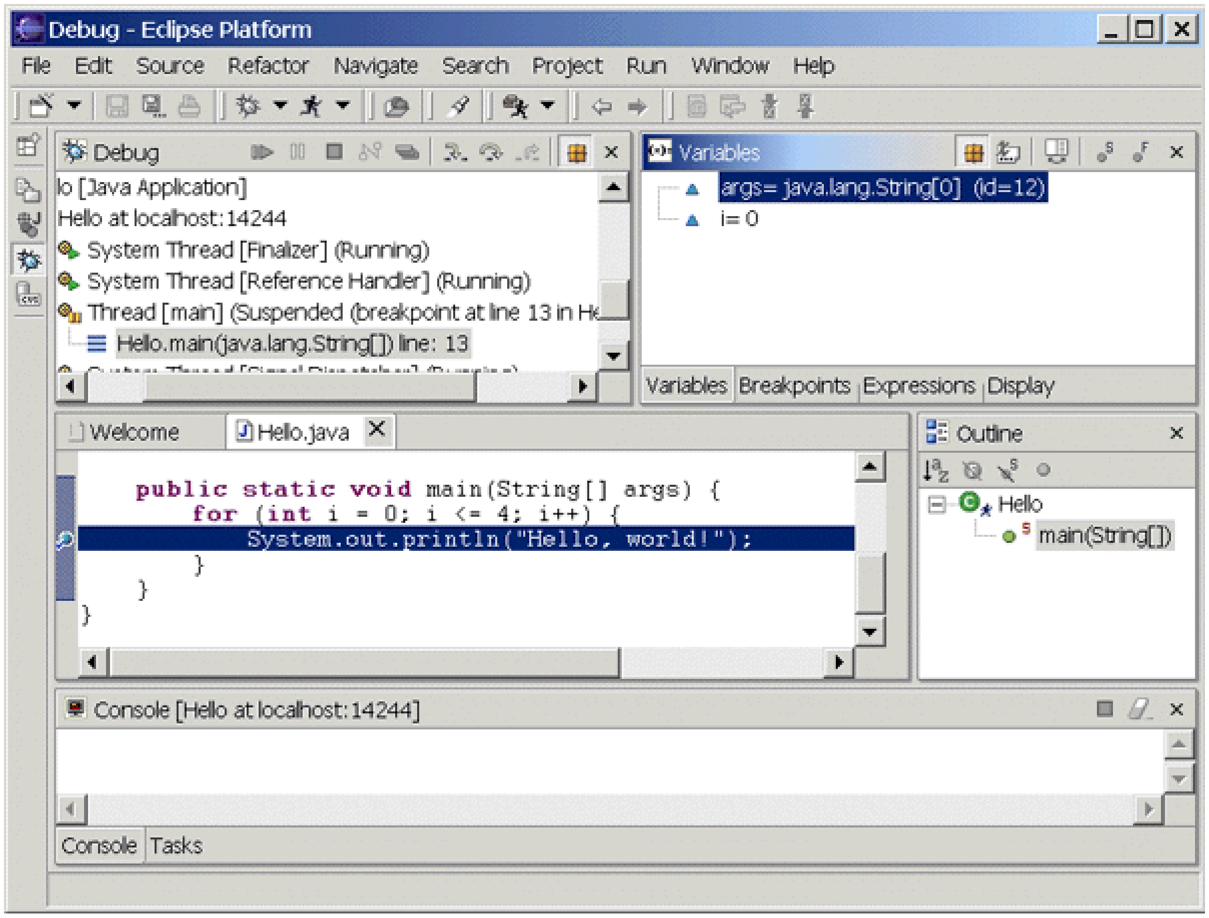
```
/**
 * @author david
 *
 * To change this generated comment edit the
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of typ
 * Window>Preferences>Java>Code Generation.
 */
public class Hello {

    public static void main(String[] args)
        for (int i = 0; i <= 4; i++) {
            System.out.println("Hello, world
        }
    }
}
```

Console [<terminated> C:\j2sdk1.4.0_01\jre\bin\javaw.exe (10/21/02 2:32 PM)]

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

You can also run the program in the Java debugger. First set a breakpoint in `main() System.out.println()` by double-clicking in the gray margin on the left side of the editor view, next to the call to `System.out.println()`. A blue dot will appear. From the Run menu, select Debug. As described above, a Launch Configurations dialog will appear. Select Run. The perspective will automatically change to the Debug perspective, with a number of interesting new views as shown in Figure 6:

Figure 6. The Debug perspective


Debug - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Variables
Variables Breakpoints Expressions Display
Welcome
Hello.java
Outline
Console Tasks

lo [Java Application]
Hello at localhost:14244
System Thread [Finalizer] (Running)
System Thread [Reference Handler] (Running)
Thread [main] (Suspended (breakpoint at line 13 in He
Hello.main(java.lang.String[]) line: 13

args= java.lang.String[0] (id=12)
i= 0

```
    public static void main(String[] args) {
        for (int i = 0; i <= 4; i++) {
            System.out.println("Hello, world!");
        }
    }
}
```

Hello
main(String[])

Console [Hello at localhost:14244]

First, notice the Debug view at the top left of the perspective. This view shows the call stack and has a toolbar in the title bar that allows you to control the execution of the program, including buttons to resume, suspend, or terminate the program, step into the next statement, step over the next statement, or return from a method.

The panel at the top right contains a number of tabbed views including Variables, Breakpoints, Expressions, and Display. Here I've clicked Variables so we can see the current value of `i`.

You can obtain more information about any of the views with the context-sensitive help: click on the title of the view and press F1.

Additional plug-ins

In addition to plug-ins like the JDT for editing, compiling, and debugging applications, plug-ins are available that support the complete development process from modeling, build automation, unit testing, performance testing, version control, and configuration management.

Eclipse comes standard with a plug-in for working with CVS, the open source Concurrent Versions System for source control. The Team plug-in connects to a CVS server, allowing the members of a development team to work on a set of source code files without stepping on each other's changes. Source control from within Eclipse won't be explored here further, because it requires setting up a CVS server, but the capability for supporting a development team, not just standalone development, is an important and integral feature of Eclipse.

Just a few of the third-party plug-ins that are either available or have been announced include:

Version control and configuration management

- CVS

- Merant PVCS
- Rational ClearCase

UML modeling

- OMONDO EclipseUML
- Rational XDE (replaces Rose)
- Together WebSphere Studio Edition

Graphics

- Batik SVG
- Macromedia Flash

Web development, HTML, XML

- Macromedia Dreamweaver
- XMLBuddy

Application server integration

- Sysdeo Tomcat launcher

For a more complete list of available plug-ins, see Resources for a link.

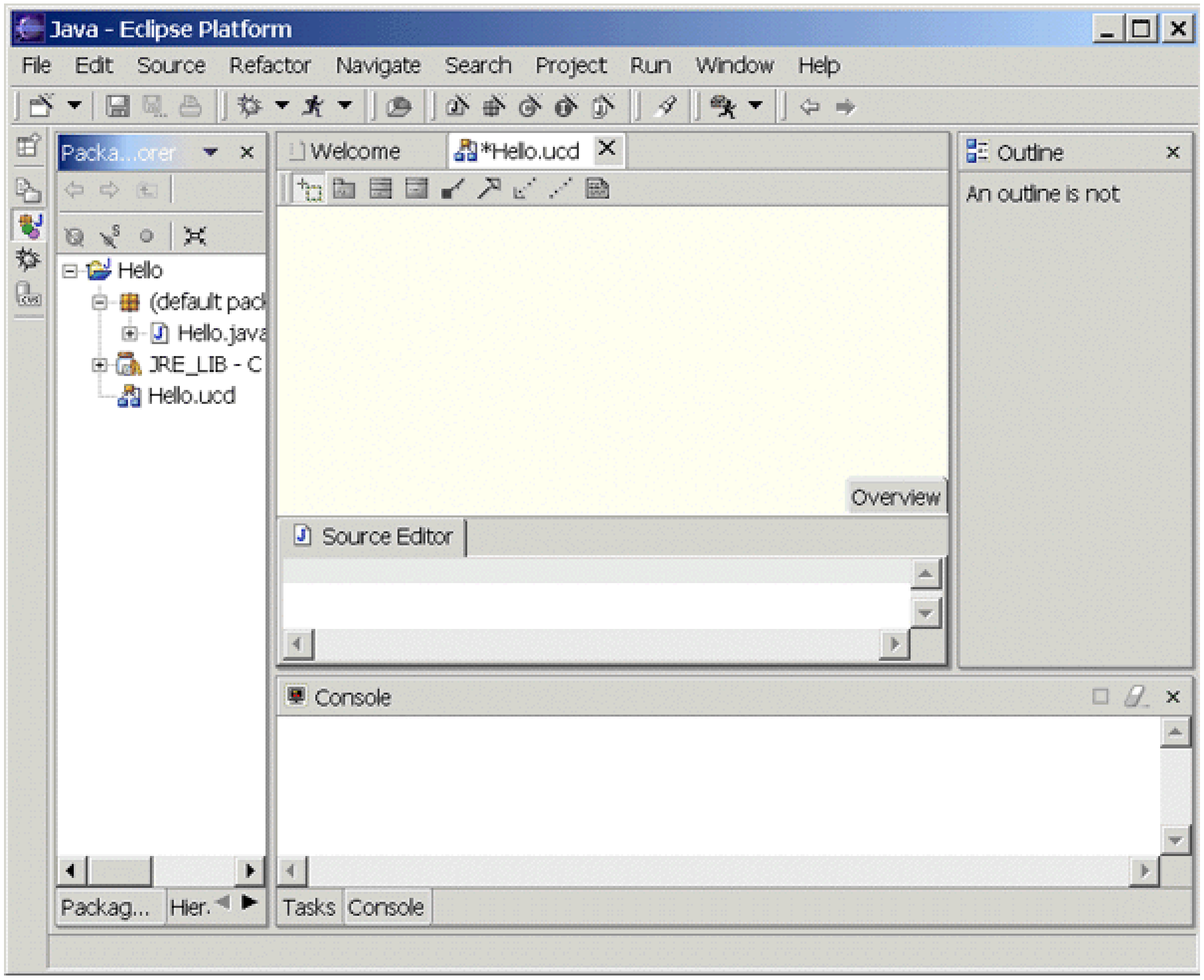
Example: a UML modeling plug-in

To view an example of a plug-in, and see how it integrates with Eclipse, download the popular OMONDO EclipseUML (see Resources for a link); you'll need to register, but the plug-in is free. This plug-in depends on GEF, or the Graphical Editor Framework, another plug-in for Eclipse. GEF is part of the Tools subproject. To download GEF, go to the Eclipse Web site (see Resources), select "downloads", and then click the link for the "Tools PMC downloads page". Note you will need to download the GEF build recommended by OMONDO (GEF version 2.0 for OMONDO version 1.0.2).

Once downloaded, a plug-in is usually installed by unzipping the download file and copying its contents to the Eclipse plug-ins directory. In this case, GEF requires that it be unzipped into the Eclipse directory (it automatically goes into the plug-ins directory from there) while EclipseUML requires unzipping directly into the plug-ins subdirectory of the Eclipse directory. To be safe, you might want to unzip each into a temporary directory and copy the directories as appropriate from there. If Eclipse is running, you'll need to stop and restart it for the plug-ins to be recognized.

Once EclipseUML (and GEF) are installed, you can create a class diagram the same way you create a Java class file. In the Java perspective, right click on the "Hello" project in the Package Explorer, and select **New=>Other** from the pop-up menu. There will be a new option for UML in the left panel of the New dialog. The free version of EclipseUML only supports class diagrams, so the only option on the right is UML Class Diagram. Select UML Class Diagram, and type in a name for the class diagram, such as "Hello":

Figure 7. The Class Diagram editor



In the editor area, a graphical editor will appear with a blank canvas for a class diagram. There are two ways you can create a class diagram: by reverse-engineering existing code by dragging and dropping a Java file from the Package Explorer onto the class diagram, or by using the drawing tools available in the toolbar above the blank diagram. To try out the first method, create a new class called Person (use **File=>New=>Class**) and give it the two private attributes listed below:

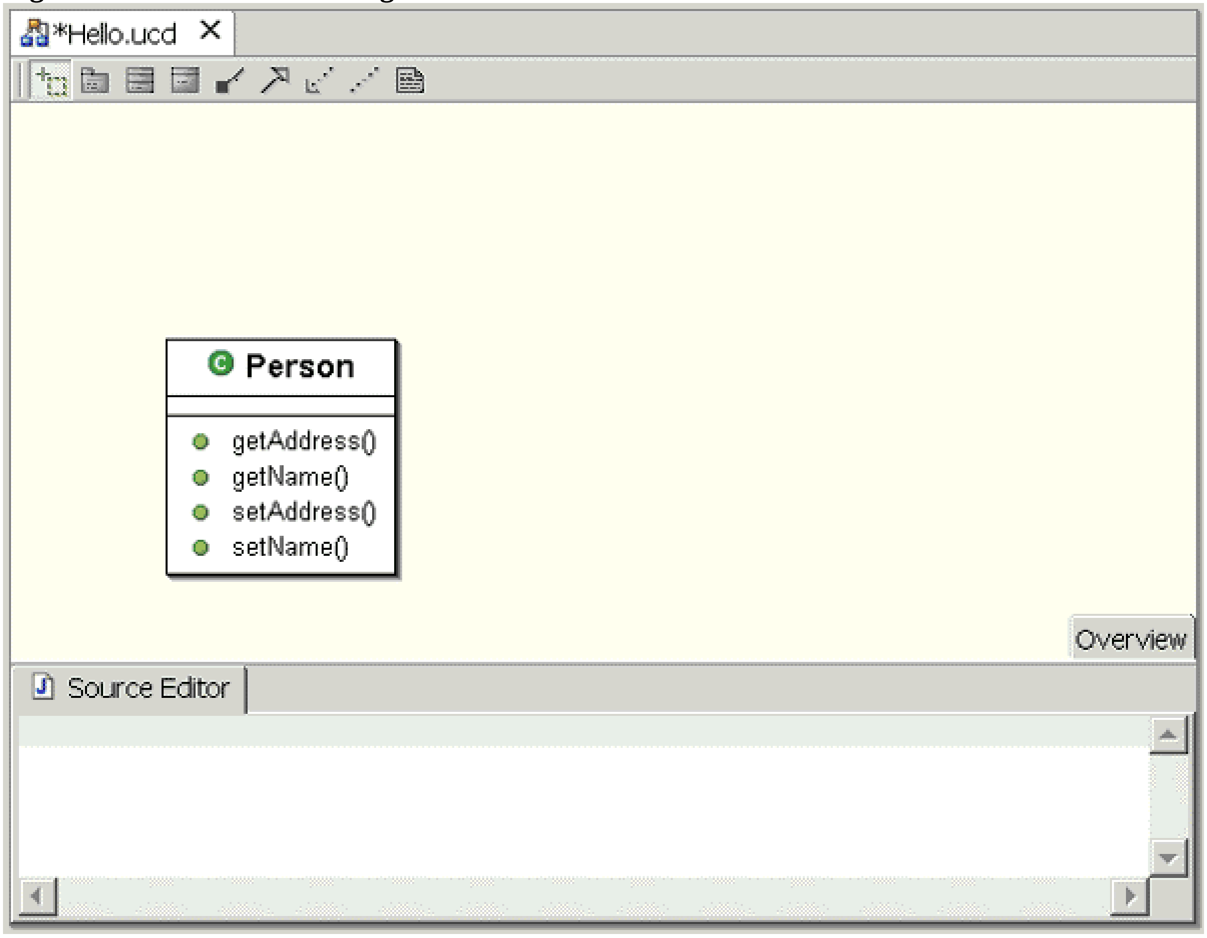
```
/** Person.java
 * @author david
 */
public class Person {
private String name;
private Address address;

/**
 * Returns the address.
 * @return Address
 */
public Address getAddress() {
         return address;
}

/**
 * Returns the name.
 * @return String
 */
public String getName() {
        return name;
}

/**
```

```
 * Sets the address.
 * @param address The address to set
 */
public void setAddress(Address address) {
        this.address = address;
}

/**
 * Sets the name.
 * @param name The name to set
 */
public void setName(String name) {
       this.name = name;
}

}
```

(I should confess that I only typed in the lines for the name and address attributes. The getter and setter methods were automatically generated through Eclipse by right-clicking on the source code, and selecting **Source=>Generate Getter and Setter** from the pop-up menu.)

Save and close `Person.java Hello.ucd`.

Figure 8. The Person class diagram

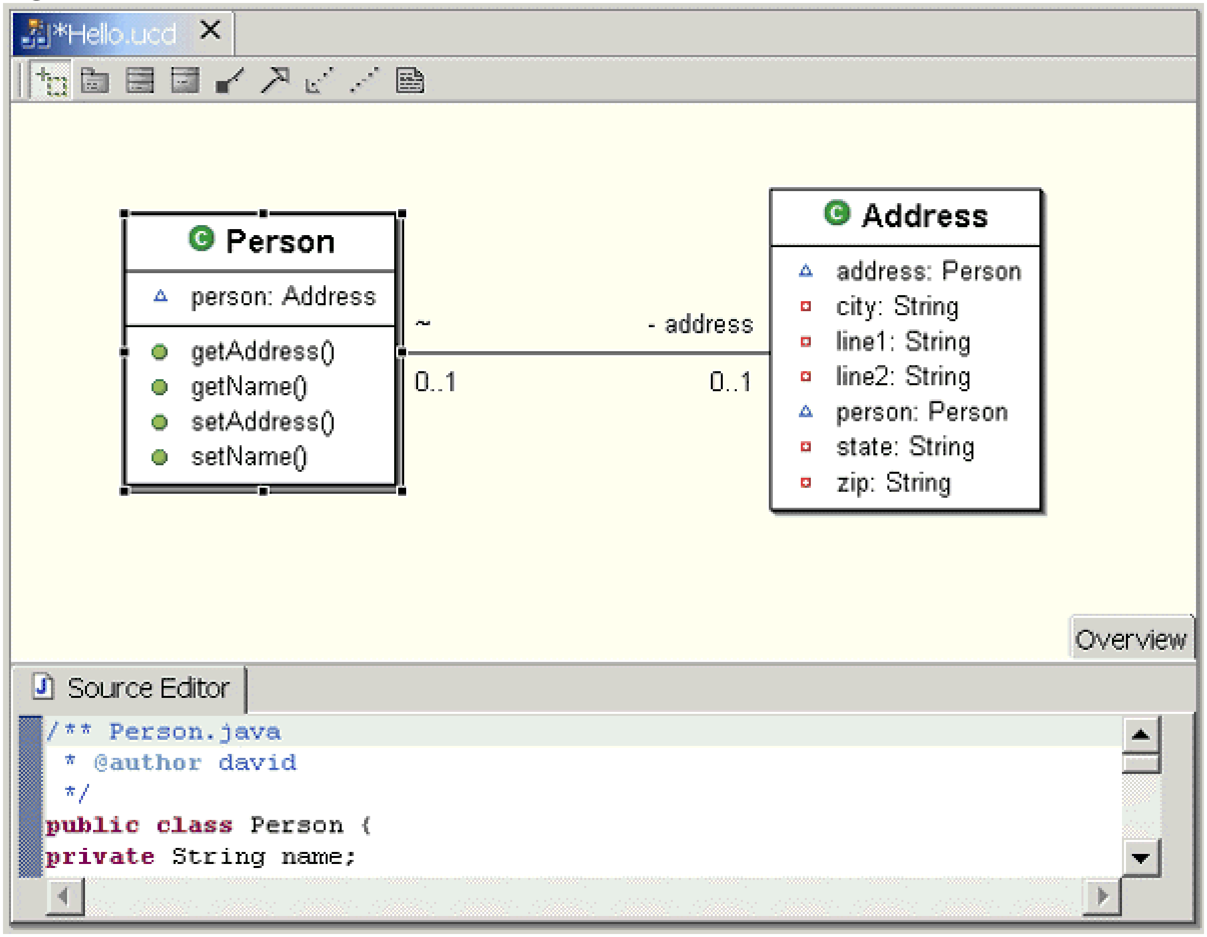


To create a Java class from the UML, click on the "New class" button on the toolbar at the top of the class diagram window, the third button from the left, and then click on the class diagram. When the New class wizard opens, type in Address as the class name and press Finish.

You can add attributes to the class by right-clicking on the class name and selecting **New=>Attribute**. In the New attribute dialog box, enter the attribute name, type, and visibility. Add methods by right-clicking on the class name and selecting **New=>Method**.

As you change the diagram, the Source Editor window below the diagram will reflect the changes. Finally, you can diagram the relationship between the classes by clicking on the Association button (fifth from the left) and drawing a line from the Person class to the Address class. This will bring up another dialog box where the association properties can be entered (refer to the EclipseUML help for more information about the required information). The diagram should look something like this:

Figure 9. Association



```
/** Person.java
 * @author david
 */
public class Person {
private String name;
```

This UML plug-in demonstrates several characteristics that are typical of Eclipse plug-ins. First, it shows the tight integration between tools. It is never obvious that there are multiple components at work; the integration with the Eclipse Platform and the JDT are seamless. For example, when the Person class was created, it displayed syntax errors because one of its attributes, `Address`, was undefined. These went away once the `Address` class was created in the UML diagram.

Another characteristic is EclipseUML's ability to build on functionality offered by other plug-ins -- in this case, the GEF, which provides tools for developing visual editors.

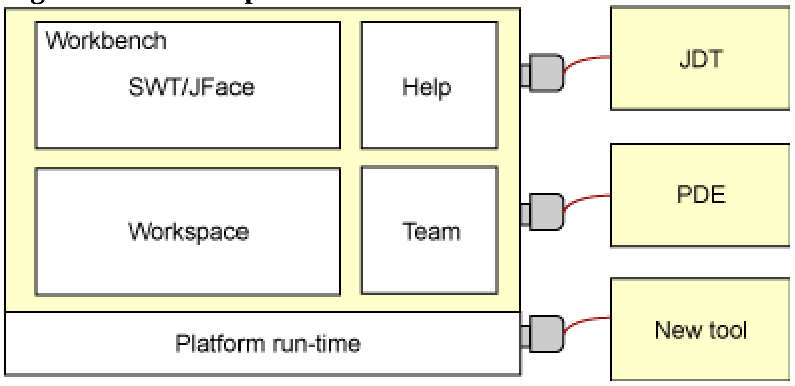
Yet another characteristic involves the way the EclipseUML plug-in is distributed using several tiers of functionality. The basic plug-in, which supports class diagrams, is free, but more sophisticated versions are available for a fee.

Eclipse Platform architecture

The Eclipse Platform is a framework with a powerful set of services that support plug-ins, such as JDT and the

Plug-in Development Environment. It consists of several major components: the Platform runtime, Workspace, Workbench, Team Support, and Help.

Figure 10. The Eclipse Platform architecture



Platform
The Platform runtime is the kernel that discovers at start-up what plug-ins are installed and creates a registry of information about them. To reduce start-up time and resource usage, it does not load any plug-in until it is actually needed. Except for the kernel, everything else is implemented as a plug-in.

Workspace
The Workspace is the plug-in responsible for managing the user's resources. This includes the projects the user creates, the files in those projects, and changes to files and other resources. The Workspace is also responsible for notifying other interested plug-ins about resource changes, such as files that are created, deleted, or changed.

Workbench
The Workbench provides Eclipse with a user interface. It is built using the Standard Widget Toolkit (SWT) -- a non-standard alternative to Java's Swing/AWT GUI API -- and a higher-level API, JFace, built on top of SWT that provides user interface components.

The SWT has proven to be the most controversial part of Eclipse. SWT is more closely mapped to the native graphics capabilities of the underlying operating system than Swing or AWT, which not only makes SWT faster, but also allows Java programs to have a look and feel more like native applications. The use of this new GUI API could limit the portability of the Eclipse workbench, but SWT ports for the most popular operating systems are already available.

The use of SWT by Eclipse affects only the portability of Eclipse itself -- not any Java applications built using Eclipse, unless they use SWT instead of Swing/AWT.

Team support
The team support component is responsible for providing support for version control and configuration management. It adds views as necessary to allow the user to interact with whatever version control system (if any) is being used. Most plug-ins do not need to interact with the team support component unless they provide version control services.

Help
The help component parallels the extensibility of the Eclipse Platform itself. In the same way that plug-ins add functionality to Eclipse, help provides an add-on navigation structure that allows tools to add documentation in the form of HTML files.

The forecast for Eclipse

A critical mass is developing around Eclipse. Major software tool vendors are on board, and the number of open source Eclipse plug-in projects is growing every day.

A portable, extensible, open source framework isn't a new idea (Emacs comes to mind), but because of its mature, robust, and elegant design, Eclipse brings a whole new dynamic into play. IBM's release of $40 million worth of world-class software into the open source arena has shaken things up in a way that has not been seen in a very long time.

Resources

- Documentation, articles, and downloads of Eclipse are available from the Eclipse Project Web site.

- View the contents of the Common Public License v1.0.

- Browse a complete list of plug-ins for Eclipse.

- Download the popular OMONDO EclipseUML; you'll need to register, but the plug-in is free.

- Information about open source software, including certified open source licenses such as the Public Common Licence, is available at the Open Source Initiative Web site.

- The definitive explanation of copyleft is available on the Free Software Foundation's Web site.

- Read the press release announcing the donation of Eclipse to the open source community.

- Learn more about Eclipse in these *developerWorks* articles:
  - "Interview with Marc Erikson about the Eclipse code donation" (*developerWorks*, November 2001).
  - "Working the Eclipse Platform" (*developerWorks*, November 2001).
  - "Getting to know WebSphere Studio Application Developer" (*developerWorks*, November 2001).
  - "Help for reusing your assets" (*developerWorks*, November 2001).
  - "Create native, cross-platform GUI applications" (*developerWorks*, April 2002).
  - "Internationalizing your Eclipse plug-in" (*developerWorks*, June 2002).
  - "Testing your internationalized Eclipse plug-in" (*developerWorks*, July 2002).

  - "Plug a Swing-based development tool into Eclipse" (*developerWorks*, October 2002).
  - "Working XML: Use Eclipse to build a user interface for XM" (*developerWorks*, October 2002).

- Numerous articles about Eclipse have also appeared in the computer press recently. Read why a professor at Carleton University chose to base two ambitious projects on Eclipse in "Promoting shared software through Eclipse" (*ADTmag.com*, November 2002).

About the author

David Gallardo is an independent software consultant and author specializing in software internationalization, Java Web applications, and database development. He has been a professional software engineer for over fifteen years and has experience with many operating systems, programming languages, and network protocols. His recent experience includes leading database and internationalization development at a business-to-business e-commerce company, TradeAccess, Inc. Prior to that, he was a senior engineer in the International Product Development group at Lotus Development Corporation, where he contributed to the development of a cross-platform library providing Unicode and international language support for Lotus products including Domino. You can reach David at david@gallardo.org.