Application Designer

Appendices

Version 8.3.1

April 2013

Application Designer

This document applies to Application Designer Version 8.3.1.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: CIT-APPENDICES-831-20130412

Table of Contents

Preface

The following appendices are available:

Appendix A - Call Sequence for Adapter	Describes how an incoming request by the browser client is processed inside an adapter.
Appendix B - Usage of Methods Inherited from the Adapter Class	Gives information about adapter classes and how to use them.
Appendix C - Data Types to be Used by Adapter Properties	Describes the various data types that can be used by adapter properties.
Appendix D - Class Loader Concepts	Gives information about class loader management.
Appendix E - StartCISPage Servlet	Describes the StartCISPage servlet that is used to open intelligent HTML pages.
Appendix F - Using JSwat for Debugging	Describes the graphical debuggerJSwat and its usage.
Appendix G - Using Eclipse with Application Designer 2.4 Functionality	Contains information about the old Eclipse plug-in that was available with previous versions. For your convenience, it is still supported.

I

1 Appendix A - Call Sequence for Adapter

This chapter describes how an incoming request by the browser client is processed inside an adapter. The request contains all the changes of properties that have been made at client side.

Normal Call Sequence

- `init()`
  This method is called only once - when creating the adapter inside a subsession. Before calling this method, Application Designer makes sure that the adapter instance is properly registered inside the Application Designer environment. Therefore - for example - you have access to the session management: use the `findSessionContext()` or `findSubSessionContext()` method in order to look for some values inside the `init()` method. It is not possible to use the `find...SessionContext()` methods inside the constructor of an adapter - since the session is not yet assigned to the adapter instance.

  When navigating between pages (using the `switchToPage()` or `openPopupPage()` method), the corresponding adapter objects are only created once. For example, if you navigate from page "A" to page "B" and back to page "A", the adapter of page "A" does not change. The `init()` method is only called once - at the time the adapter is instanciated.
- `activate(...)`
  This method is implemented by the `Adapter` class already. You only need to overwrite this method if you want to passivate the state between requests. In this case, you can activate this state inside your implemented method of your adapter class. If you use the adapter class to co-operate, for example, with components running in a container of an application server, you should synchronize the state passivation with the container's passivation.
- `reactOnDataTransferStart()`
  This method is called when the transfer of the changed properties starts. You can initialize some internal members at this time. If you overwrite this method, do not forget to include the method of the super-class (`Adapter.reactOnDataTransferStart()`) into your method implementation!
- `setXxx(), setYyy(),...`
  Now, the set methods of the changed properties of the browser client are transferred. It is very important that your implemented set methods never cause an exception or an error.
- `reactOnDataTransferEnd()`
  This method is called after setting the changed properties. Use this method to perform operations you always want to execute when processing a request.
- `invoke()`**-Method**
  If the request has a method call inside, the method is invoked now.
- `processAsDefault()`
  If the request has no method call, this standard method is called.

- `reactOnDataCollectionStart()`
  This method is called when the transfer of adapter properties starts. Use this method, for example, for performance improvements during the following get methods, for example, by building temporary objects.
- `getXxx(),getYyy()`
  All get methods of the adapter - including array elements which may be passed back by - are called.
- `reactOnDataColletionEnd()`
  This method is called when data collection is finished. Temporary objects - which you may have created for performance reasons - can be released for garbage collection now.
- `passivate(...)`
  This method is the counterpart of the activate method.

Call Sequence when a Subsession is Destroyed

- `endProcess()`
  This method is called inside the adapter if the user decides to terminate the subsession. For example, in the Application Designer workplace environment, this method is called whenever the user chooses the close button of a page.

You can deny closing a subsession in your implemented method:

```
public class ABCAdapter
    extends com.softwareag.cis.server.Adapter
{
    ...
    ...
    public void endProcess()
    {
        // veto the endProcess in case of unsaved data
        if (changedDataNotSaved == true)
        {
            this.outputMessage("E","Please save data first");
            return;
        }
        // close subsession
        super.endProcess();
    }
}
```

Call Sequence when a Session is Destroyed

If a session is removed from Application Designer - for example, if the user closes the browser or if a system administrator removes the session - the adapter instances are informed in the following way:

- `destroy()`
  In your implemented method, clean up all resources bound to your adapter instance. You cannot deny the destroying of the session - but you can react.

Error/ Runtime Exceptions

Error and runtime exceptions occurring during the adapter request processing may be handled centrally inside your adapter. For more details, see *Binding between Page and Adapter* in the *Special Development Topics*.

Pay Attention when Overwriting

The methods named above are already implemented with default behavior inside the class `com.softwareag.cis.server.Adapter`. Pay attention when overwriting these methods inside your adapter and always include the super-class's processing into your own implementation. The first statement inside your implementation should call the super-class method:

```
public class ABCAdapter
    extends com.softwareag.cis.server.Adapter
{
    ...
    ...
    public void reactOnDataTransferStart()
    {
        super.reactOnDataTransferStart();
        // now own implementation
        ...
        ...
    }
}
```

2 Appendix B - Usage of Methods Inherited from the Adapter Class

Inside the Application Designer management, adapters have to provide a defined interface to be managed correctly by the system. This interface is declared by `com.softwareag.cis.Server.IAdapter`. In order to have a high level of comfort during developing adapters, you should derive your adapter classes from the super-class `com.softwareag.cis.Server.Adapter`. This class already provides some useful methods.

Access to Lookup Session Context

As you know, session management defines sessions (corresponding to one browser instance) and subsessions (corresponding to one process inside the Application Designer workplace). There is the possibility to bind and look for parameters on both levels:

- `Adapter.findSessionContext()` - returns the context which is on top of all subsessions. All adapters inside one session refer to the same session context.
- `Adapter.findSubSessionContext()` - returns the context which is held per subsession. Only adapters - belonging to the same subsession - share this context.

The result is a context supporting the interface `com.softwareag.cis.context.ILookupContext`. This interface provides two important methods:

```
public Object lookup(String s, boolean reactWithErrorIfNotExist);
public void bind(String s, Object o);
```

The session context is used, for example, to refer to the current user who is logged in, the chosen language, etc. The subsession context is used to share data inside a subsession.

Do not use the context as global variable buffers in a very intensive way. It will end up in programs relying on a lot of context information to be available - and sooner or later no one knows what has to be in the context when starting the program.

Via the methods

- `Adapter.findSessionId()`
- `Adapter.findSubsessionId()`

you can access the internally used representations of session ID and subsession ID.

Access to Application Designer Session Context

Application Designer uses its own lookup session management in order to store information of a session. You can access and manipulate this information by calling your adapter's method:

- `Adapter.findCISessionContext()` - returns a concrete session context object.

Inside the session context, the following parameters are kept:

- date format
- time format
- language
- style
- decimal separator
- and other information.

Have a look at the JavaDoc API documentation for more details.

Access to other Adapters

Access other adapters inside the same subsession by the methods:

- `Adapter.findAdapter(class)` - returns the adapter instance for a given class. Method `init()` is already called when passing back the instance - but only if the adapter was not used before.

Use this method before navigating between pages in order to prepare the adapter that will be used by the next page.

Error Output

You can display error messages inside the status bar (if it is defined in the page layout) by using the methods:

- `outputMessage(String, String (, String))`

First, pass a string for the type of message. This is needed to display a corresponding icon inside the status bar. There are constants defined inside the Adapter for specifying the type:

- `Adapter.MT_ERROR`

- `Adapter.MT_WARNING`
- `Adapter.MT_SUCCESS`

The second string is the message being shown.

The third string - which is optional - is the long text description of the message. It becomes visible by a dialog if the user clicks with the mouse on the message. If you do not specify a long description, the normal message is used.

Page Navigation

Navigate to a page by using the method:

- `switchToPage(String pageName)`

The "pageName" is the URL - either relative or absolute - of the next page.

Opening of Pop-up Dialogs

You can open a page inside a pop-up dialog by using the method:

- `openPopup(String pageName)`.

The "pageName" is the URL - either relative or absolute - of the page that is displayed inside the dialog.

You can specify pop-up parameters of the pop-up you open with `openPopup()` by using the methods:

- `setPopupTitle(String title)`
- `setPopupPageFeatures(String pageFeatures)`

Frame Communication

There are various methods to communicate to other frames:

- `openPageInTarget`
- `openCISPageInTarget`
- `invokeMethodInTarget`

- `refreshTarget`
- `sizeTarget`

Closing of a Page

The default method used for closing a page is `endProcess()`. It is provided by the `Adapter` class. The tasks performed by the `endProcess()` method are:

- The current subsession is closed and de-registered inside the session management.
- The current page is de-registered from the workplace management - if it was registered before.

Calling the `endProcess()` method ensures that all memory resources are released for the corresponding subsession.

The `endProcess()` method is called by clicking inside the page on the close icon at the top right corner of the page. You can also call it directly inside an adapter, e.g. if you want to close the subsession as reaction to the user's entered data.

Multi Language Management

You can access the multi language management using the methods:

- `replaceLiteral(String application, String textid)`
- `replaceLiteral(String application, String textid, String param1)`
- `replaceLiteral(String application, String textid, String param1, String param2)`
- `replaceLiteral(String application, String textid, String param1, String param2, Stirng param3)`

The application is the name for the abbreviation of a defined application area for which literals are defined. In the file-based multi language management, it represents the name of a CSV file that holds the text identified by a text ID.

3 Appendix C - Data Types to be Used by Adapter Properties

The Application Designer management is very flexible by allowing various data types for properties of an adapter.

Supported Data Types

- String
- int, long, short, byte
- float, double
- BigDecimal
- boolean
- CDate
- CTime
- CTimeStamp

Data Types for Managing Date and Time

The `java.util.Time` class is very powerful, but also very complex to use for business applications. Therefore, three classes are introduced to deal with date and time:

- `com.softwareag.cis.util.CDate`
- `com.softwareag.cis.util.CTime`
- `com.softwareag.cis.util.CTimeStamp`

See the JavaDoc documentation for further details.

Dates and times are transferred as strings between Application Designer and the intelligent HTML page:

- YYYYMMDD format for dates.
- HHMMSS format for times.
- YYYYMMDDHHMMSSMMM format for timestamps.

The interpretation and formatting of these strings to valid formats is done automatically.

4 Appendix D - Class Loader Concepts

An explicit class loader management was introduced to support the following scenarios:

- Classes are automatically found in the context of Application Designer without specifying a `CLASSPATH` variable.
- Classes can be stored inside an application project directory - separated from other application projects.
- During development time, easily run new pages together with the latest classes without restarting the server.

This chapter explains the class loader concepts used inside Application Designer.

Design Time - Runtime

The class loader concepts are designed to simplify the development of pages and their logical representations on the server side: adapters.

At runtime, they should only be used if you are not running in a cluster - i.e. if you do not distribute your application server on multiple nodes. When running in a cluster, classes should be located exactly there, where the application server specifications allow them to be located. Inside the Application Designer configuration, you can select which mode you are running in - for details, see *Design Time Mode and Runtime Mode* in the *Configuration* documentation.

After explaining the class loader concepts in this chapter, at the end we explain what to do in order to change a design time environment into a runtime environment.

Class Loader Hierarchy

Application Designer runs as a web application inside a servlet engine - by default, the Tomcat servlet engine is used. The class loader used by the servlet engine is called “web application loader” in the following text.

The Application Designer environment itself is running in the context of the web application loader. This class loader is looking for classes as specified by the servlet engine. Therefore the Application Designer runtime must be accessable by this class loader. For Tomcat, this is achieved by placing the *cis.jar* file inside the *<installdir>/tomcat/webapps/ROOT/WEB-INF/lib* directory.

The following topics are covered below:

Application Class Loader

The application classes (adapter classes) are loaded by the class loader management of Application Designer. This class loader looks for Java classes as follows:

- All *.class* files inside the directory:

  <webapp>/softwareag/appclasses/classes
- All *.jar* files inside the directory:

  <webapp>/softwareag/appclasses/lib
- All *.class* files inside any application project under the directory:

  <webapp>/<project>/appclasses/classes
- All *.jar* files inside any application project under the directory:

  /<webapp>/<project>/appclasses/lib
- All classes that are referenced in the classpath extension that can be defined in the Application Designer configuration (*cisconfig.xml*).

Unlike normal class loader hierarchies, the application class loader always tries to resolve a class inside its application directories first. Only if the class is not found, the parent class loader is called - the web application loader. The benefit is that application classes are totally separated from the servlet engine classes - e.g. by using XML parser libraries. You are not bound to the parser delivered with the servlet engine.

Inside the Application Designer session management, a session is bound to an application class loader instance. Therefore the application class loader - which was instanciated when the session was created - is kept in the session during its whole life cycle. All objects created inside this session use this instance of the class loader.

In case of changing classes inside the *softwareag/appclasses* or the corresponding application-project subdirectories, you can force to create a new class loader used in all sessions which are created afterwards. This means, that you can upgrade your system without disturbing running sessions. Old sessions are still using their old classes; new sessions are using new classes.

The creation of a new instance of a class loader is triggered inside the monitoring tool. See *Monitoring* in the *Development Workplace* documentation.

By choosing the button **Use latest Version of Applications for new Sessions**, a new class loader instance is generated.

A new class loader instance can also be created during development inside the Layout Painter. See also the "Hello World!" example in the *First Steps* and its section *If you Change the Adapter*.

Initialisation of Your Application

Every time a new instance of a class loader generated, the initialisation process of your application is also performed. This guarantees that, for example, all static variables you may use internally can be correctly initialised by your initialisation procedure.

The initialisation of applications is described in the *Becoming a Member of the Startup Process* part of the *Special Development Topics*.

Guidelines for Development

The guidelines you have to follow during development are quite simple:

- Always put *all* your application/adapter classes inside the *softwareag/appclasses* directory or in the corresponding project directories. When using the project management (which is strongly recommended), store the classes in the project directories so that you can easily copy projects as self-containing units between different Application Designer installations.
- Do *not* put classes into the servlet engine's class loader's class path.
- Avoid class duplicates (a *.class* file in the */classes* subdirectory also contained in a *jar* file inside the */lib* subdirectory).
- Reload the classes by creating a new class loader instance. To see the effects re-logon. (The re-logon can be done by refreshing the browser.)

Classpath Extensions in cisconfig.xml

In the *cisconfig.xml* file, you can define the possibility to explicitly include defined directories or *jar*/*zip*/etc. files in the application class loader. The following example shows a *cisconfig.xml* file containing a class loader extension:

```
<cisconfig ...>
    <classpathextension path="c:/development/centralclasses/classes/"/>
    <classpathextension path="c:/development/centralclasses/libs/central.jar"/>
</cisconfig>
```

Consequence: you can also include classes that are located outside the web application's directory structure into the application class loader of Application Designer.

Pay attention: if defining directories that contain *.class* files, then the path definition inside the classpath extension must end with a slash (/).

Loading Resource Files

The Application Designer application class loader does only load classes to be loaded into the Java virtual machine. It is not able to load resource files that you might access from your code.

Place resource files into the web application class loader, below the directory *<webapps>/WEB-INF/classes/* so that they are loaded in a correct way.

Preparing for Runtime

The following topics are covered below:

Basics

As explained in the previous section, the Application Designer class loader concepts are very useful for design time purposes. What is the price? The Application Designer class loader finds its classes by accessing the file system. It uses for this reason the `cis.home` parameter inside the *<webapp>/WEB-INF/web.xml* file in order to know the file root directory of the web application.

At runtime - especially if your application server distributes the load on several physical nodes - this is dangerous: each node may have its own directory structure and you cannot specify one root directory anymore in which the web application is located.

Consequence: for running in these scenarios, you have to prepare your application accordingly - i.e. you have to place your classes at the places where the application server definition defines them to be located.

The normal directories to put classes in are:

- *<webapp>/WEB-INF/lib* for libraries (*.jar* files).
- *<webapp>/WEB-INF/classes* for single class files (*.class* files).

In addition, you must switch off the flag "useownclassloader" inside the *cisconfig.xml*. Consequently, the Application Designer application class loader will not be used at all - all classes are loaded by the web application loader.

Example

Example: let us assume that you have set up the Application Designer application project "projectxyz". The classes for this project are located in

- *<webapp>/projectxyz/appclasses/classes/*.class* and
- *<webapp>/projectxyz/appclasses/lib/*.jar*

so that the Application Designer class loader can reach them.

For changing to the runtime scenario, just copy the *.*class* and *.*jar* files from your project directory into the corresponding standard directories.

5 Appendix E - StartCISPage Servlet

The StartCISPage servlet is the central servlet that is used in order to open intelligent HTML pages. It was already mentioned several times in this documentation. This chapter describes certain attributes that you can pass inside the servlet call.

Normal Calling of a Page

A normal page is called in the following way:

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html
```

The StartCISPage servlet creates a frameset page around the intelligent HTML page that provides for specific functions that are internally required.

Appending Application Parameters

Application parameters can be passed by just appending the name and the value of the parameters to the URL. Each parameter must be the name of a property that is provided for by the server side adapter.

Example: the adapter provides for a property `company`. When opening a page via

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&company=softwareag
```

then the `setCompany` method of the adapter is called and the value "softwareag" is passed.

This is a very simple and powerful way to pass parameters through the URL.

Controlling the Session Life Cycle

A page relates to adapters living inside a session on server side. A session is opened by default when referencing a page via StartCISPage. By default, it is closed when the initial StartCISPage page is removed - either by closing the browser or by loading a different URL into it.

You can explicitly control this automated removal of sessions with the parameter `ONUNLOADBEHAVIOR`. If you call a page in the following way, the session is not removed when the page is removed:

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&ONUNLOADBEHAVIOUR=NOTHING
```

Controlling the Session ID

By default, a new session ID is internally generated when opening a page by StartCISPage. But you can also pass the session ID and the subsession ID explicitly. This might be of interest if you require to control the Application Designer session management from outside.

Calling a page in the following way

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&SESSIONID=4711&SUBSESSIONID=5
```

will internally open the session with ID 4711 - or use 4711 if it already exists. The same applies on subsession level.

Pay attention: if you use this possibility, then you are responsible for managing session IDs in such a way that they are unique.

Setting Default Parameters

Language

As described in *Multi Language Management*, Application Designer internally holds a language per session. This language can be set from outside:

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&LANGUAGE=E
```

Default Style Sheet

By calling

```
http://<host>:<port>/<webapplication>/servlet/StartCISPage?PAGEURL=/<project>/<pagename>.html&SESSIONCSS=../softwareag/styles/CIS_PARROTT.css&DEFAULTCSS=../softwareag/styles/CIS_PARROTT.css
```

you define that the CIS_PARROTT style sheet is used instead of the default style sheet. Of course, you can reference any style sheet of your own.

Mixing Parameters

All parameters can be mixed without any restrictions.

Setting Parameters with the HTTP Method POST

Instead of adding the parameters to the URL, you can also use the HTTP method `POST` to set the parameters in an HTML form.

Example (similar to the example under ***Appending Application Parameters***, but with `POST`):

```
<html>
<head>
<title>Start Application Designer Demo Application</title>
<script type="text/javascript">
function submitStart() {
document.forms["myform"].submit();
}
</script>
</head>
<body>
 <form id="myform" name="myform" action="servlet/StartCISPage" method="post">
  <input type="hidden" name="PAGEURL" value="/<project>/<pagename>" />
   Company: <input type="input" name="company" value="softwareag" /><br/>
 </form>
 <a href="#" onclick="submitStart()">Start Demo</a>
 <div id="status">Click on Start Demo</div>
</body>
</html>
```

6 Appendix F - Using JSwat for Debugging

JSwat is a graphical debugger, available free of charge - see ***http://www.bluemarsh.com/***. It is not restricted to any development environment - but is a standalone debugging environment. It is flexible to use and can be used e.g. in a customer environment where you do not want to install a full development environment in order to trace and debug your applications. JSwat is written in Java and also runs on Linux-based systems.

We strongly recommend to use this debugger instead of using a `System.out.println()` way for debugging.

Usage of JSwat

JSwat supports remote debugging by the Java JPDA architecture. This means, you start Application Designer with some flags to force the virtual machine that it sends debug information to interested listeners. JSwat acts as listener for this information.

The following topics are covered below:

- Starting Application Designer in Remote Debugging Mode
- Configuring JSwat
- Running the Debug Session

Starting Application Designer in Remote Debugging Mode

There is a special batch file available for starting the default Application Designer in remote debugging mode. Have a look at the file *<installdir>\bin\GUIServer_remote.bat*:

```
cd..
cd tomcat
cd bin
set java_home=..\..\jre
set catalina_home=..
set jpda_transport=dt_socket
set jpda_address=5000
catalina jpda run
```

You see that

- the Tomcat 4.0 (Catalina) environment is started with the option "jpda";
- the transport protocol is "dt_socket" at port "5000"

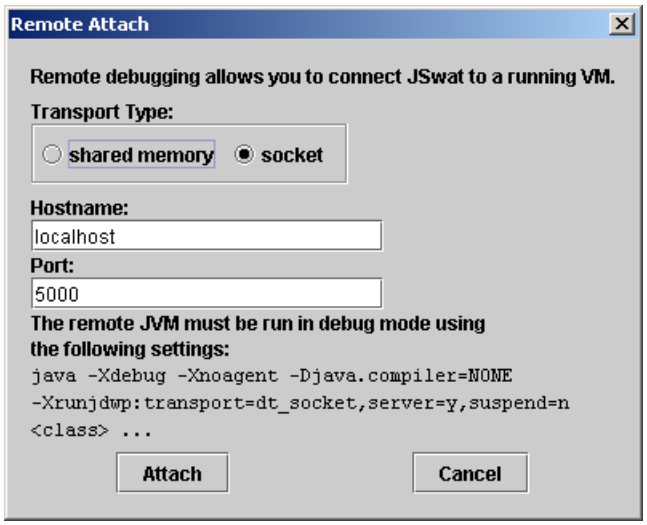
  After starting Application Designer with these options, it runs in debug mode, i.e. remote debuggers can connect by TCP/IP to the virtual machine.

Configuring JSwat

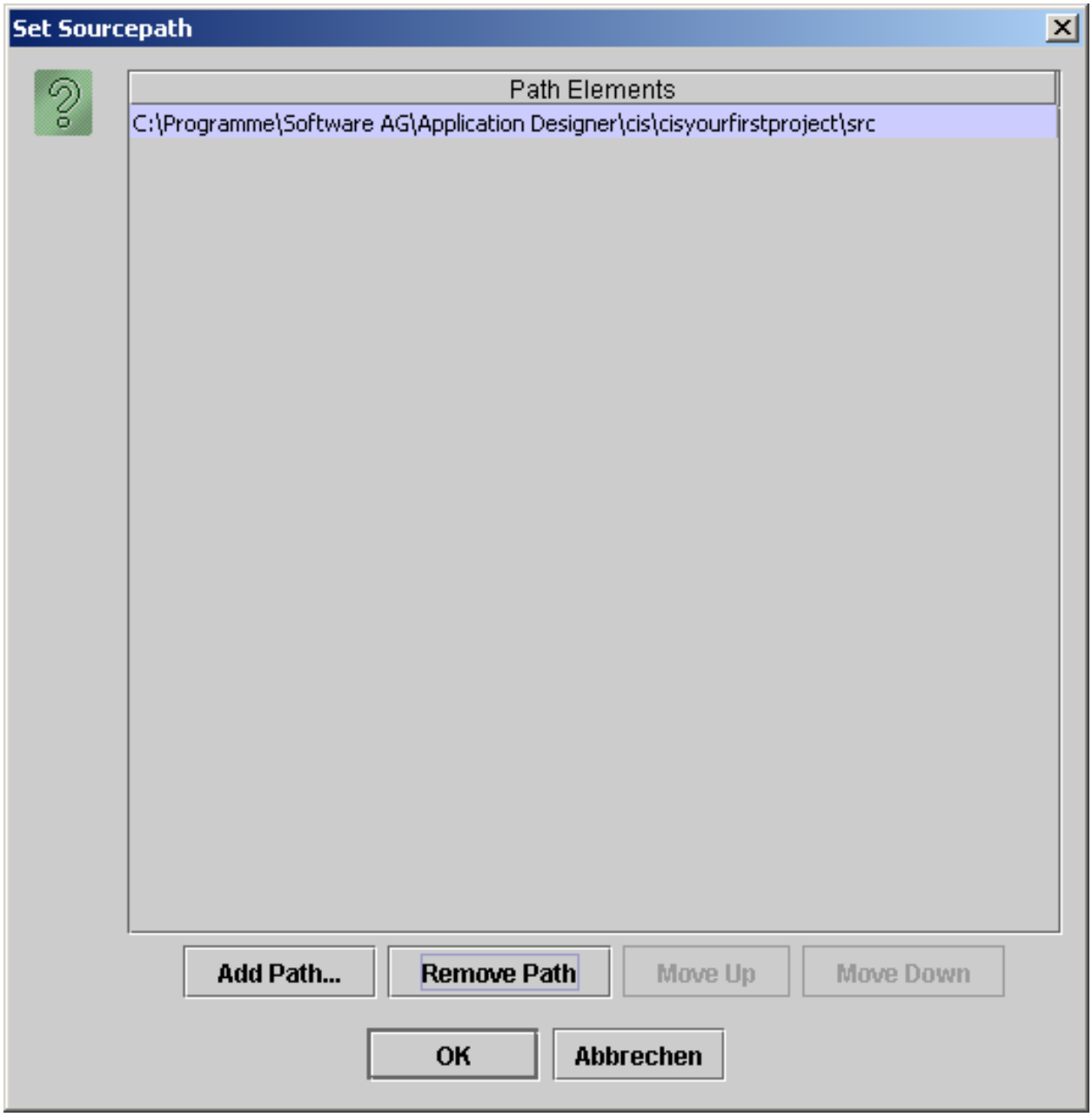
After starting JSwat, configure two items:

- Connect the JSwat debugger to Application Designer, running in debug mode.
- Tell JSwat where the sources of your classes are located.

The connection to Application Designer is done by **Session > Attach to remote...** inside JSwat. Select the parameters according to the definition inside the *<installdir>\bin\GUIServer_remote.bat* file:



The source path is maintained by **Options > Set source path...**. In the dialog, define the location of your sources.



Running the Debug Session

See the JSwat documentation for details on how to use JSwat.

II Appendix G - Using Eclipse with Application Designer 2.4 Functionality

Important: This part contains information about the old Eclipse plug-in that was available with previous versions. For your convenience, it is still supported. However, it is recommended that you use the new plug-in, the Ajax Developer. For detailed information, see the *Ajax Developer* documentation.

This part covers the following topics:

7 Eclipse: A Brief Introduction

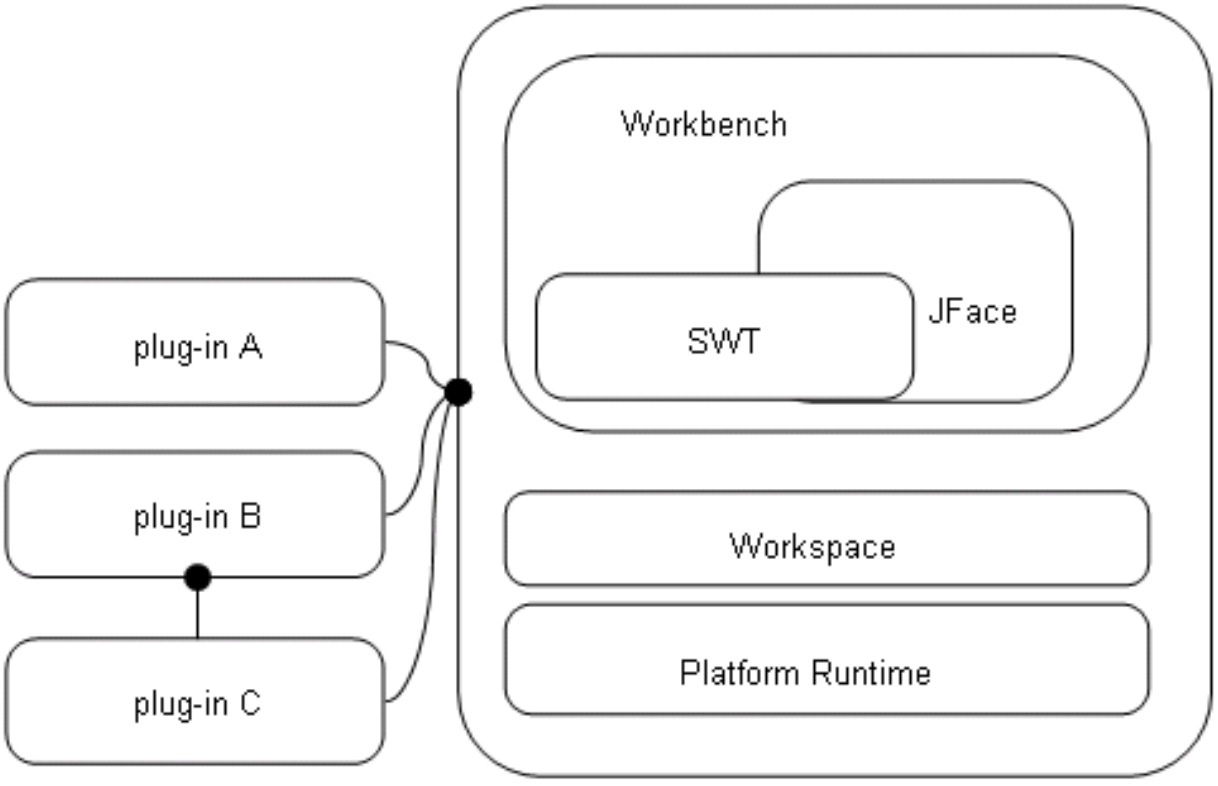
The Eclipse Platform is a very flexible open source development platform for tool integration. It provides a framework for building an integrated development environment from plug-in software components. More and more products nowadays are Eclipse-based, so that it is becoming increasingly important for developers and users to know what Eclipse is and what it offers.

The information in this document gives you a short overview of the basic Eclipse architecture, its components and standard user interfaces, as well as an introduction to what an Eclipse-based product may look like. This will help you understand Software AG products that are based on Eclipse.

Concept

The benefit of Eclipse is that it offers a single integrated platform for all development tasks. Plug-ins provide a feature of the Eclipse development environment and lead to the final Eclipse-based product. Plug-ins can be de-installed without impacting the Eclipse installation as such. Eclipse supports collaboration of development teams and is freely available from *www.eclipse.org*.

As shown in the following graphic, the basic Eclipse installation consists of three parts: the Workbench (which is further subdivided into the Standard Widget Toolkit and the JFace), the Workspace and the Platform Runtime.



- **Workbench**

  The *Workbench* is the user interface of Eclipse. The Standard Widget Toolkit (SWT) holds a set of widgets and graphics for building graphical user interfaces, such as buttons, menus, tree lists etc. With JFace, these elements are grouped into bigger, task-oriented units.

- **Workspace**

  The *Workspace* is the connection to the file system. It is used to create and manage project resources (such as files and folders).

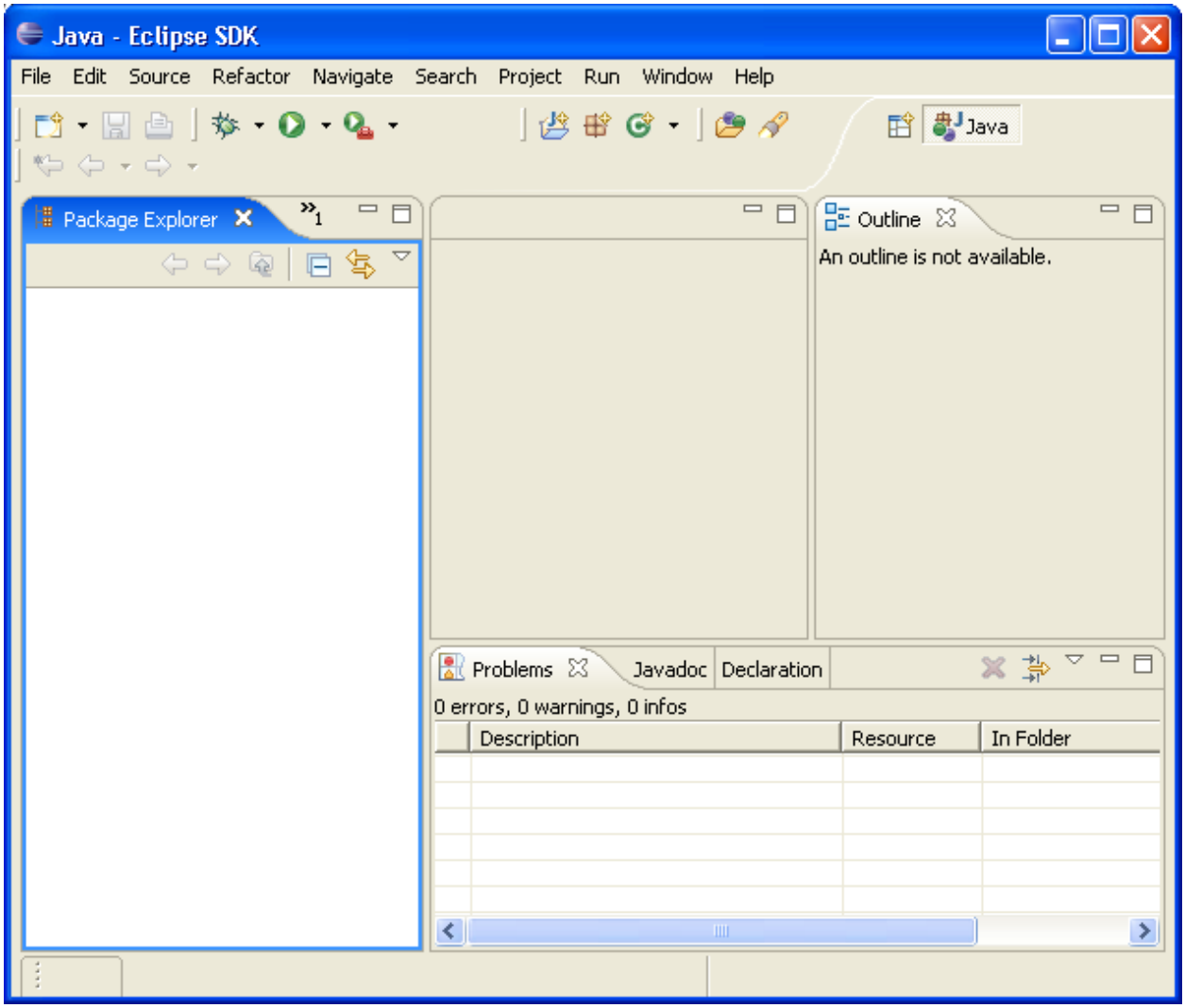
- **Platform Runtime**

  The *Platform Runtime* is the kernel that starts and runs the different components and takes care of the correct loading of plug-ins.

In addition to the basic Eclipse installation, the plug-ins extend the functionality of the Eclipse platform or other plug-ins. Plug-ins can be bundled as installable units. These installable units are called *features*. Each plug-in is connected to Eclipse via extension points, or to another plug-in, or both. Extension points are used to connect to plug-ins outside the Eclipse platform, but they also exist within Eclipse, as Eclipse itself is made of plug-ins.

Components

When you start the Eclipse Software Development Kit (SDK) for the first time, the workbench user interface and its components are displayed (after having closed the **Welcome** page), without any user-defined plug-ins. The Eclipse workbench is a platform for development tools. It provides the user interface structure for Eclipse and facilitates seamless integration of tools. The workbench consists of a collection of windows with menu bars, toolbars, shortcut bars and so-called perspectives. The name of the active perspective is shown in the title of the window. The following graphic shows an example of such a “bare” Eclipse workbench, using the Java perspective:



The workbench usually contains the following menus: **File**, **Edit**, **Navigate**, **Project**, **Window**, and **Help**. Other menus are plug-in dependent, or context-specific, based on the current perspective, editor or view. If you are a developer of plug-ins, you can develop and add new menus, editors, views or wizards.

In the following, the different components of the workbench are briefly introduced:

- Workspace
- Resources
- Wizards
- Views
- Editors
- Perspectives
- Preferences

- Properties

Workspace

As mentioned before, the workspace is the place in the file system where the different **resources** are stored. It consists of one or more projects. A project is a directory with several files and folders and has methods to build dependent resources.

Resources

Resources are items in the workspace, i.e. projects, folders, files and other dependent resources. These are all objects that will be or have been created with Eclipse. They are stored as normal files within the Eclipse workspace. A project holds several folders with files.

Wizards

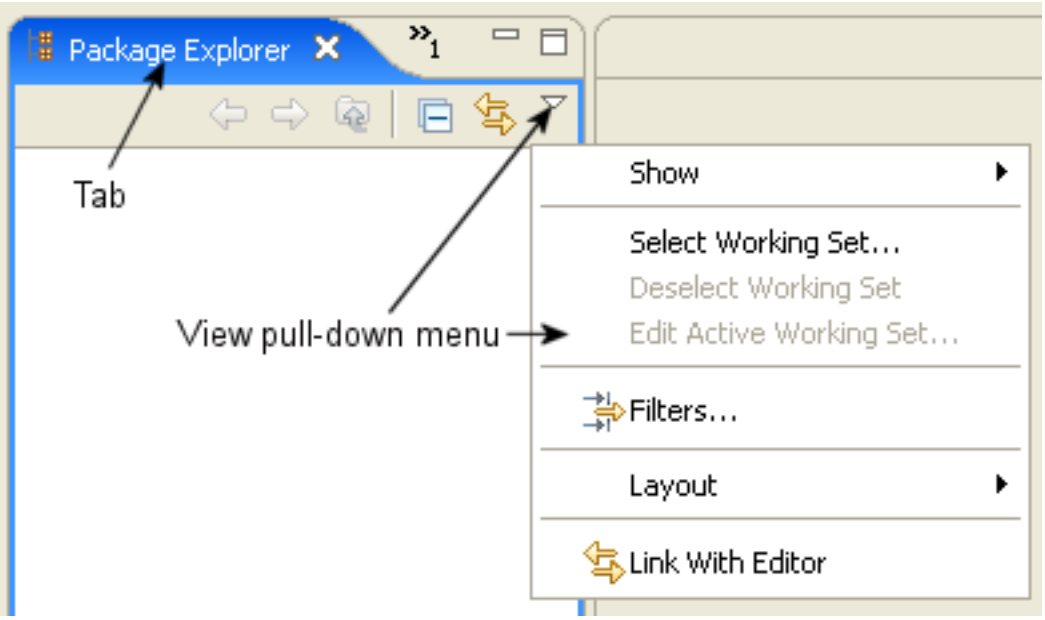
In Eclipse, most data is created using wizards. A wizard is an assistant that guides you step-by-step through a process, for example creating new resources or importing and exporting them.

Examples of wizards are:

- the New Project wizard;
- the New Class wizard;
- the New Package wizard;
- the Checkout wizard.

Views

A view is a visual component of the workbench that shows information, usually in a table or tree. It is used to navigate within a hierarchy of information, to open an editor or to display properties for the active editor. You choose **Window > Show View** to open the view with which you want to work. Several views can be stacked in a so-called tabbed notebook. To activate a view, you select its tab. Views also have their own context menus, which can be opened by right-clicking on the tab. Each view has a pull-down menu, which can be opened by selecting the down arrow to the right of the toolbar, below the tab. It contains functionality like sorting and filtering, which applies to the entire content of the view.



Examples of views are:

- Navigator;
- Package Explorer;
- Outline;
- Problems;
- Properties;
- Error Log.

Editors

An editor is another visual component of a workbench page. It is used to edit a document, to keep changes until the document is saved, or for browsing. Multiple editors may exist even for one document. There are content assistants, simple page and multiple page editors, and syntax high-lighters. Menus, toolbars and options in an editor are context-sensitive and change according to the environment. Eclipse has a list of registered editors, which are consulted first when you open a resource that needs an editor. If none of the editors in the list is suitable for the file type, the workbench checks automatically if any other editor from the underlying operating system is available (external editor). If an external editor is located, it will be launched.

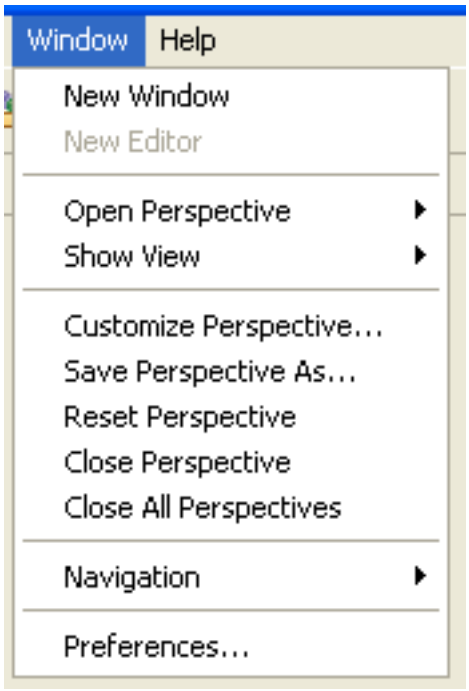
Examples of editors are:

- Java source editor;
- XML editor;
- Ant editor;

- Text editor;
- Plug-in editor.

Perspectives

A perspective can be described as a container that holds several views and editors, bundled for a specific task. Views and editors can be dragged and dropped to other places in the workbench so that the environment fits your needs and you have your personal working perspective. Only one perspective is visible at any time. A perspective can be managed with the commands available in the **Window** menu:

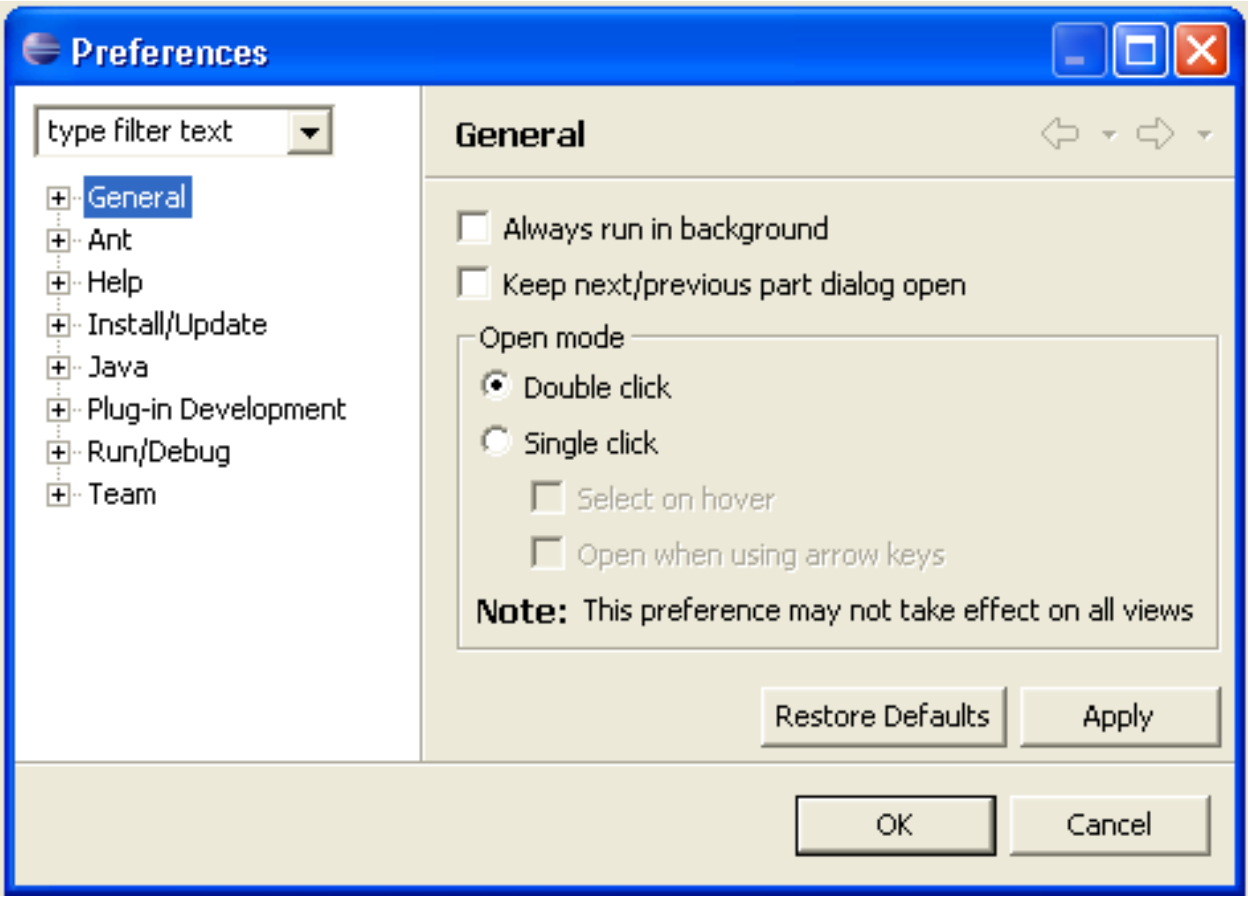


Examples are:

- Resource perspective;
- Java perspective;
- Debug perspective;
- Team synchronizing perspective.

Preferences

The **Preferences** dialog box sets the global preferences for various topics. It is available in the **Window** menu.



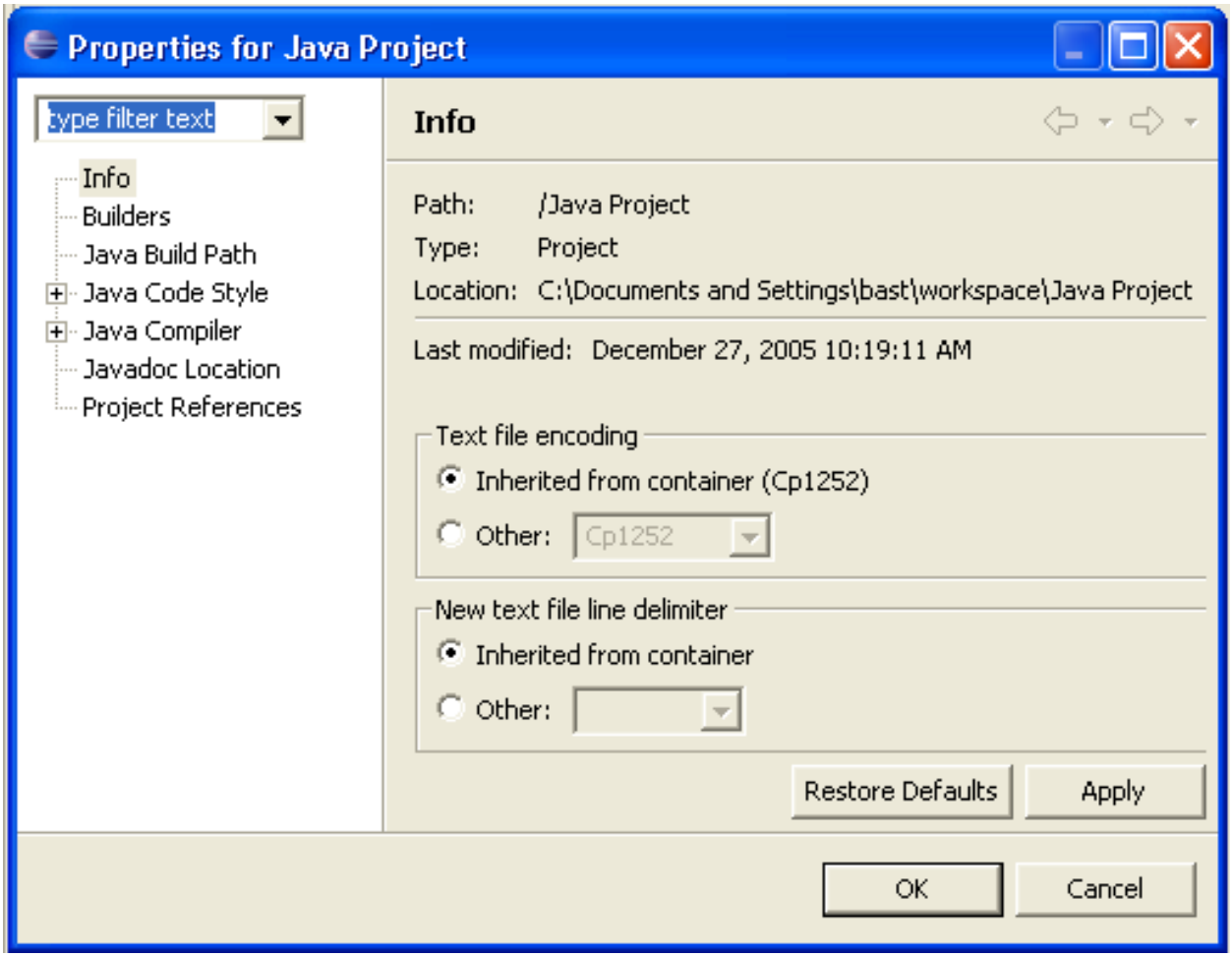
Examples of preferences are:

- Editor settings;
- Java compiler settings;
- Team settings.

The **Preferences** dialog box has a search facility (see the field **type filter text** in the graphic) and a history to navigate backwards and forwards through the pages.

Properties

The **Properties** dialog box shows and changes the properties of a resource or some other object in the active editor or view. The **Properties** command is available in the **File** menu or as the last command in the context menu of a resource.



Examples of properties are:

- the properties of a file;
- the properties of a project.

The **Properties** dialog box has a search facility (see the field **type filter text** in the graphic) and a history to navigate backwards and forwards through the pages.

Using Eclipse-based Products

Developing plug-ins with Eclipse is one task, using Eclipse-based products is another one. For a user of Eclipse-based products, it is generally not necessary to have an in-depth knowledge of the Eclipse user interface, but it is helpful to have an idea of the main concepts, terms and components (as described above), as they keep reappearing in the user interface of the products. Here are some tips and tricks that apply to Software AG's Eclipse-based products.

Navigation

In most Eclipse-based products, navigation is done with the help of the **Navigator** view. It is usually displayed on the left side of a perspective and shows the available resources (projects, folders, files etc.) of the product. If you select a resource in the **Navigator** view and open the context menu, the available commands (for example, for copying, pasting, deleting etc.) are displayed.

Accomplishing tasks

To accomplish certain tasks like creating or editing resources, you select an item in the **Navigator** view, open the context menu and choose the desired command. The corresponding views and editors will usually open in a view on the right side of a perspective. You use them to interact with your product, for example, to enter, edit or add data. If your tasks require a step-by-step process, it is very probable that a wizard will open automatically and guide you through the process (for example, when importing or exporting resources). You just follow the instructions in the dialog boxes.

Logs and Infos

Information about what you are doing is normally available in information and log views at the bottom of your perspective, e.g. error logs, status information, etc.

Standard menus and commands

Products are integrated seamlessly into the Eclipse workbench. This means that you do not see where the standard Eclipse workbench ends and where the product-specific user interface starts. Eclipse-based products make use of standard Eclipse menus and commands, and they add their own functionality. So an Eclipse-based product usually still has the Eclipse “flavor”, but also its own components. As a consequence, product documentation describes only product functionality, and not the standard Eclipse functionality. The latter can be found in the standard Eclipse online help. If you miss the description of some functionality in the product documentation, it is thus very likely that you will find it in the Eclipse documentation (see the Eclipse help at *http://www.eclipse.org/documentation/*).

Making life easier

Once you have established a working environment of views, editors and information windows, it is a good idea to save this environment as a customized perspective. To do so, you choose **Window > Save Perspective As**. You can re-open this perspective any time and thus do not have to create it again and again. This saves time and effort. If you want to restore the workbench to its default settings, you choose **Windows > Reset Perspective**.

Further Reading

If you are a new to Eclipse, this set of links will help you:

- *http://www.eclipse.org/* (the official Eclipse website)
- *http://www.eclipse.org/articles/index.php* (technical articles written by members of the development team and other members of the Eclipse community)
- *http://marketplace.eclipse.org/* (solutions for Eclipse)

The generally accepted Eclipse User Interface Guidelines can be found at the following location: *http://wiki.eclipse.org/User_Interface_Guidelines*.

The very useful Eclipse online help is available at: *http://www.eclipse.org/documentation/*.

Further information can be found in the following books:

- "The Java Developer's Guide to Eclipse" from Shavor, D'Anjou, Fairbrother, Kehn, Kellerman and McCarthy (Addison-Wesley)
- "Eclipse - Building Commercial Quality Plug-ins" from Clyberg and Rubel (Addison-Wesley)

Note: It is possible to use the Eclipse user interface using the keyboard only. See the Eclipse online help for detailed information.

8 Setting up Eclipse as Your Development Environment

The information in this section applies when you create your layouts in the development workplace and want to use Eclipse as your development environment for Java.

Creating a Project in the Application Designer Environment

Application Designer has an application project concept in which each project is kept in its own directory.

When you create a project with Application Designer's Project Manager, a new directory is created in your web application directory. For example, when using the standard Windows installation, the root directory of the project is *<installdir>/tomcat/webapps/cis/<yourproject>*.

For further information, see *Project Manager* in the *Development Workplace* documentation.

Creating a Java Project in Eclipse

You must have a source directory into which the adapter classes for your project are written. It is recommended that the name of this source directory is *src*. For example, create the directory *<installdir>/tomcat/webapps/cis/<yourproject>/src* in order to keep the sources. See also *Preferences* in the *Development Workplace* documentation.

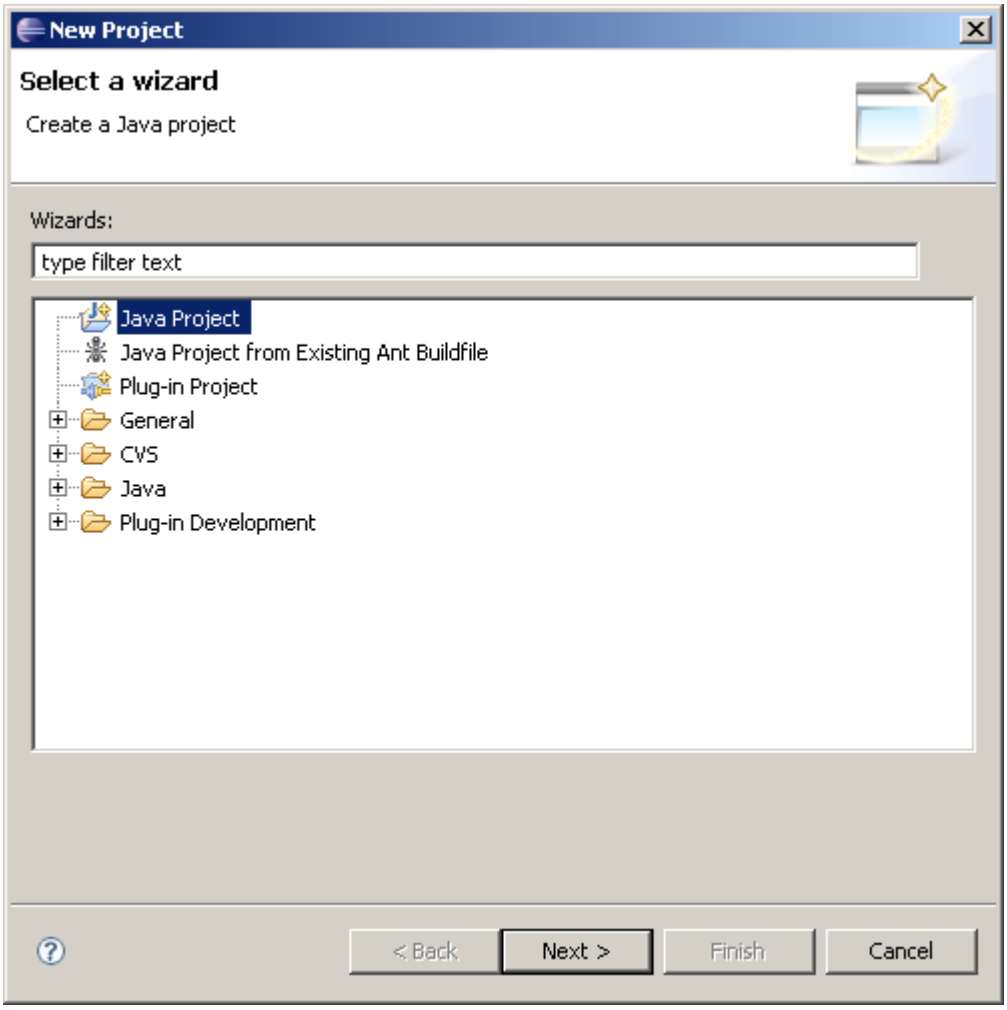
At design time, Application Designer expects the classes to be located in the application project's */appclasses/classes* directory.

This section describes the simple way of creating a Java project for Application Designer. If you are an experienced Eclipse user, you can also store the Java project in a different directory and then link the source files from any directory.

To create a Java project for an Application Designer project

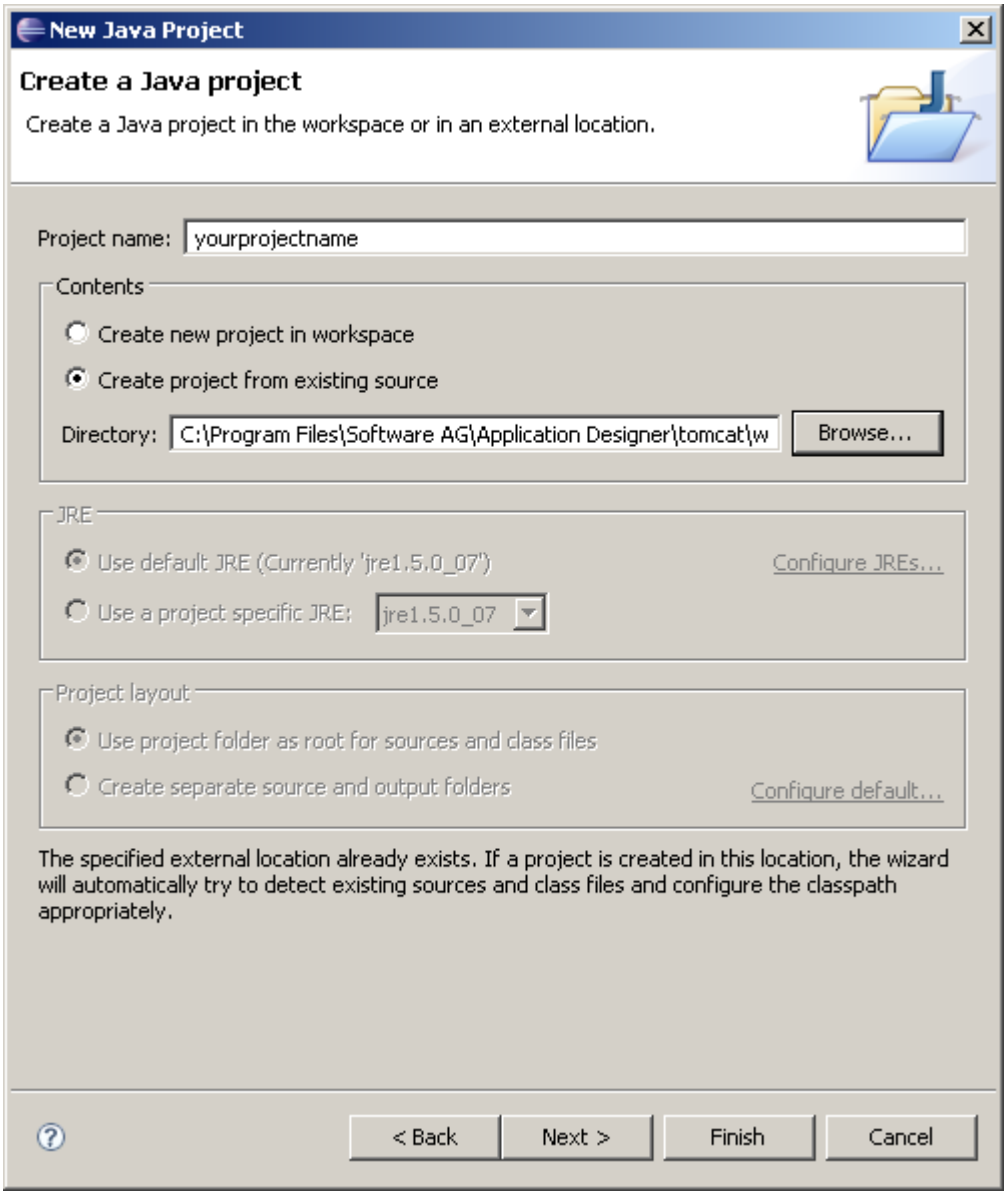
1 From the **File** menu of Eclipse, choose **New > Project**.

The following dialog box appears.



2 In the resulting dialog box, select **Java Project** and choose the **Next** button.

The following dialog box appears.



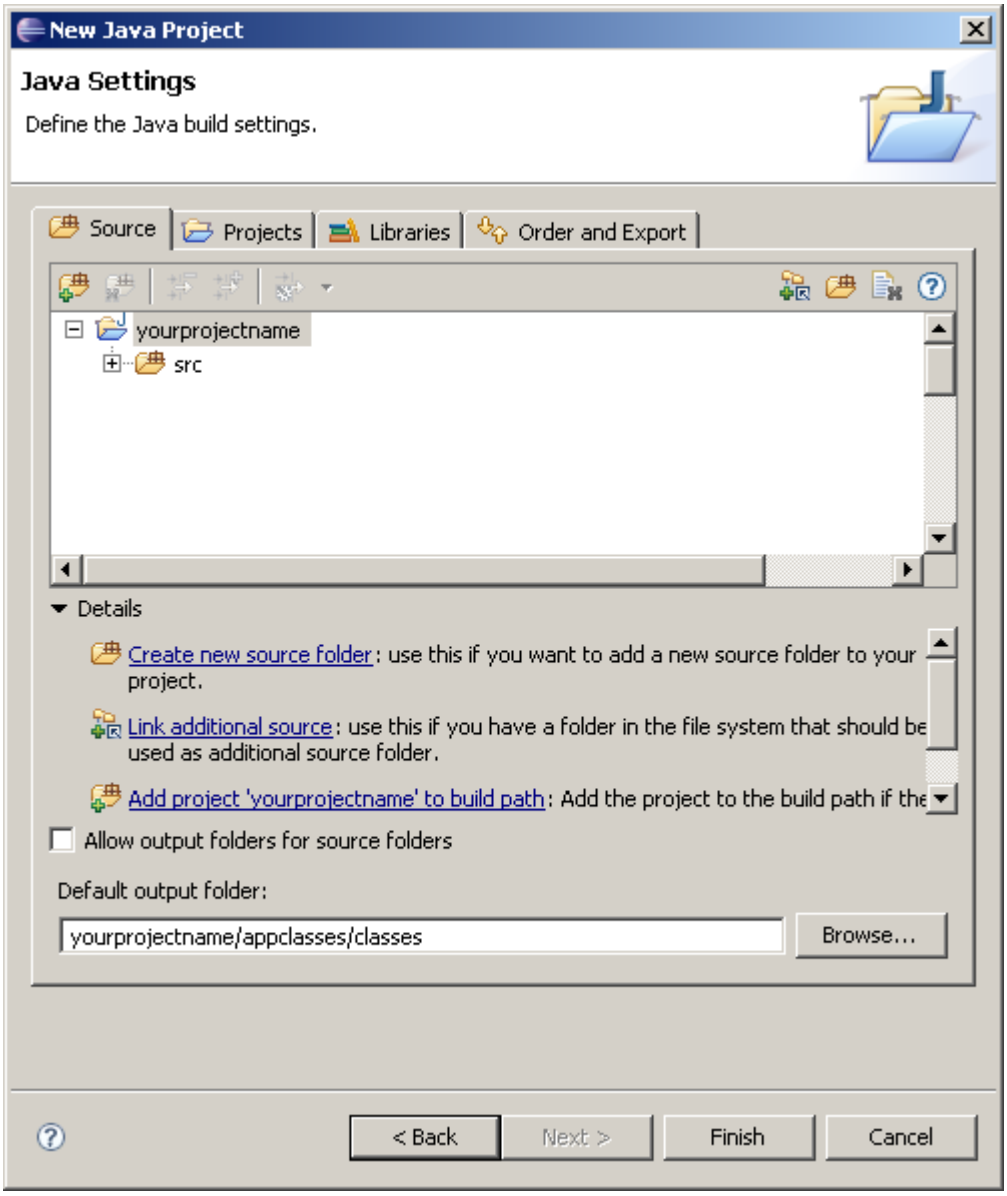
3 Specify a name for your Java project name.

If you want to work with existing Application Designer projects (which you have created in the development workspace), it is recommended that the Java project name and the Application Designer project name are the same.

4 Place the root of the project inside the project directory that was created by Application Designer. Therefore, select the option button **Create project from existing source** and specify the corrsponding directory.

5 Choose the **Next** button.

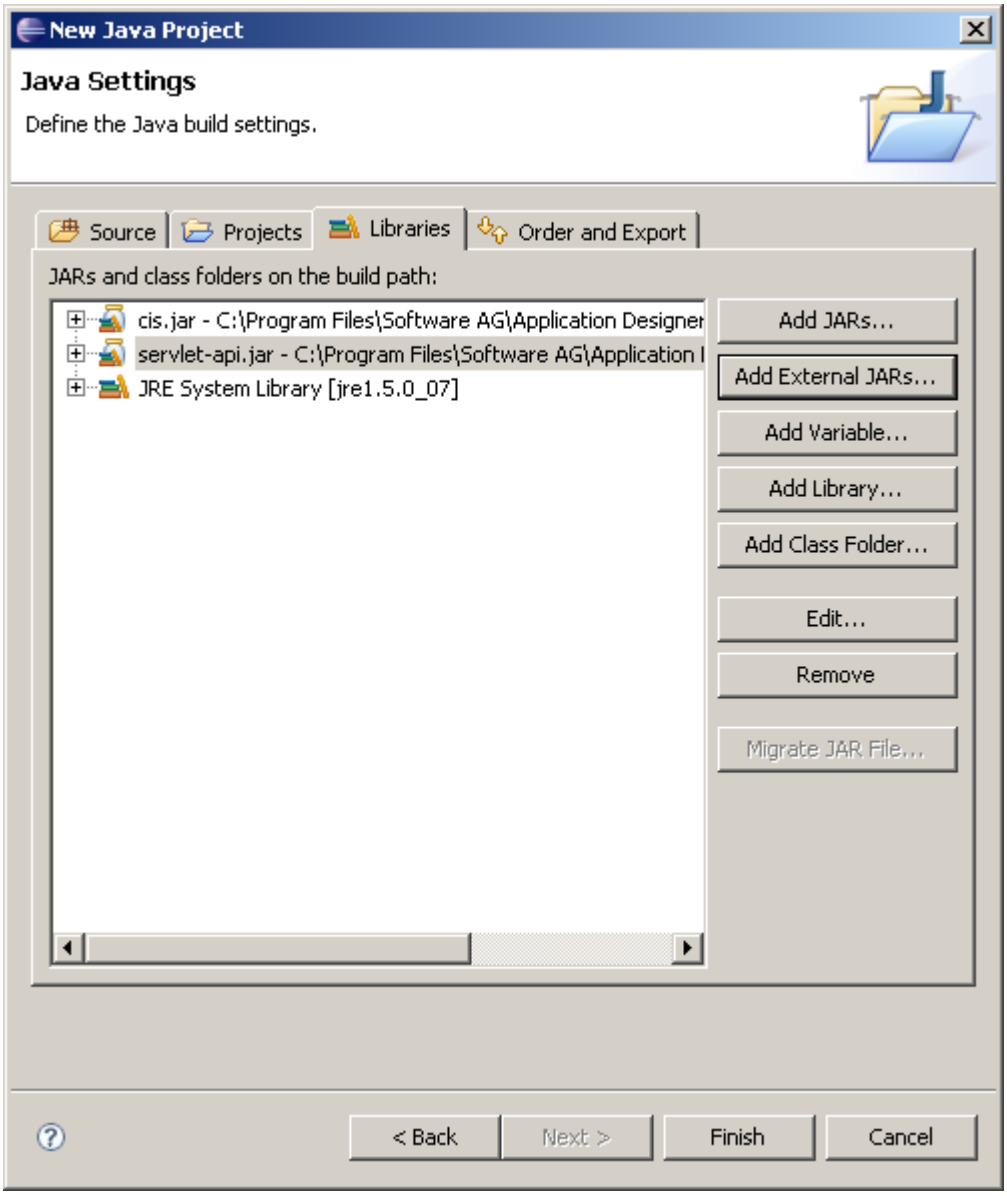
The resulting page is used to define the Java build settings for the Eclipse project.



6 Make sure that Eclipse uses the following directories of your Application Designer project:

- Your source directory (for example, the directory with the recommended name *src*).
- The directory containing the compiled classes of your project. Use the **Browse** button to specify *appclasses/classes* as the default output folder.

7 Go to the **Libraries** page.

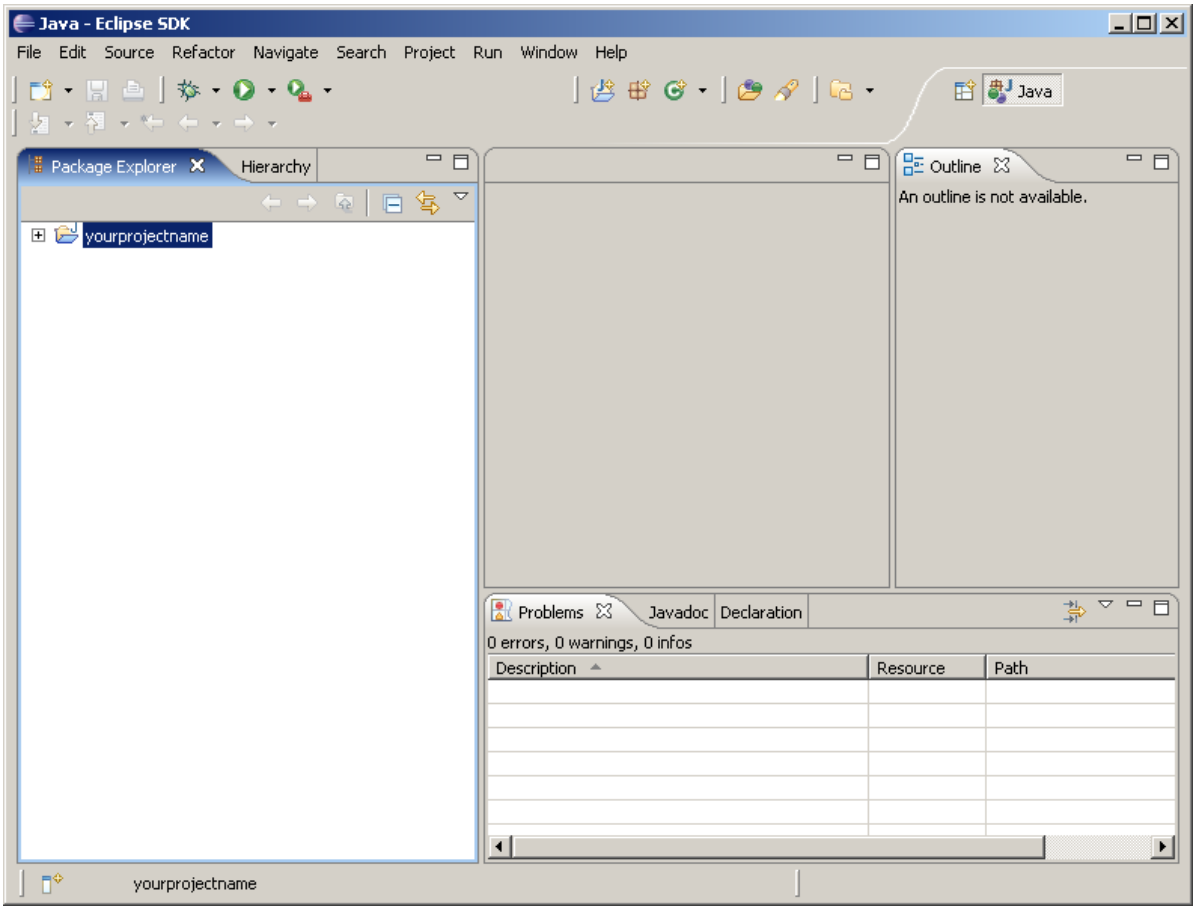


8 Use the **Add External JARs** button to add all libraries which are required by your project. You must add at least the following libraries:

- ***cis.jar***
  This library is located inside your Application Designer installation in the directory *<installdir>/tomcat/webapps/cis/WEB-INF/lib/*.
- ***servlet-api.jar***
  This library is located in the directory *<installdir>/tomcat/common/lib*.

9 Choose the **Finish** button.

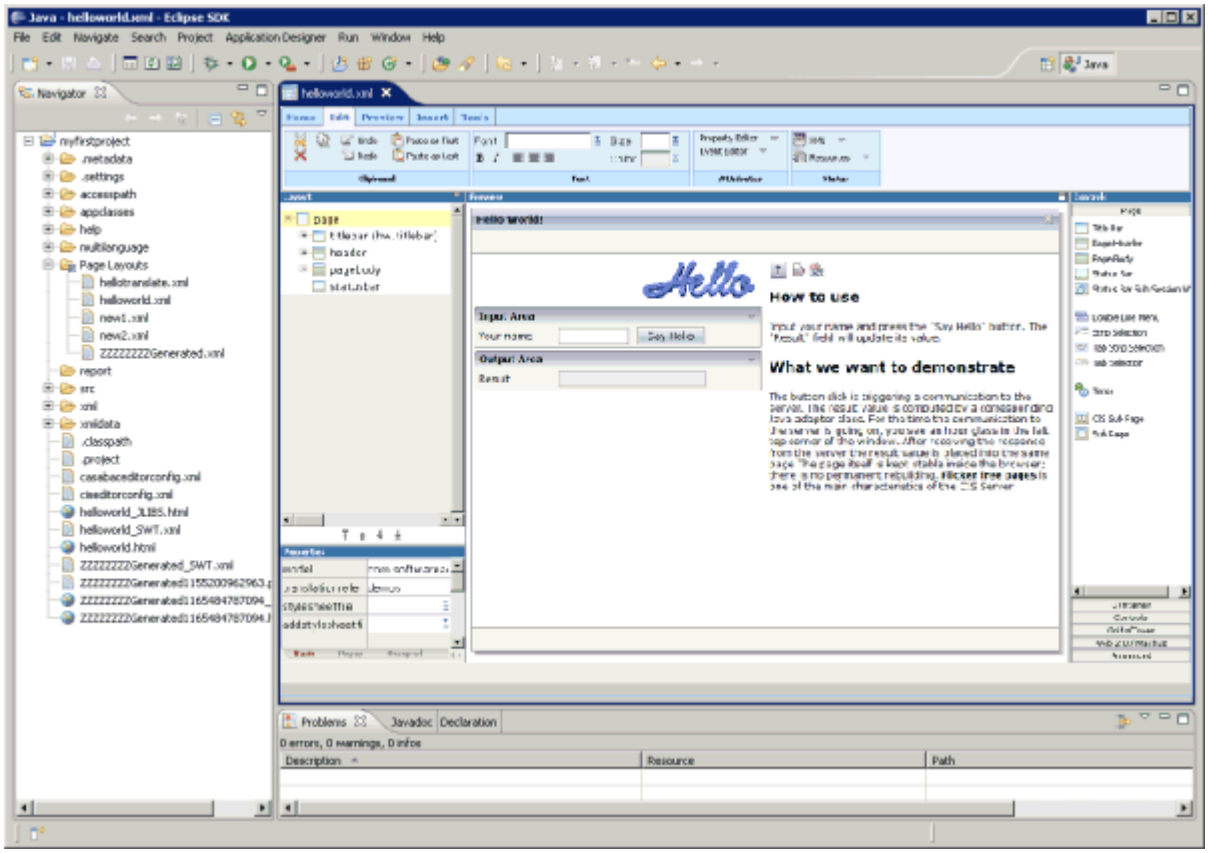
Your project is set up.


Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
Hierarchy
yourprojectname
Outline
An outline is not available.
Problems
Javadoc
Declaration
0 errors, 0 warnings, 0 infos
Description
Resource
Path
yourprojectname

9 Setting Up the Eclipse Plug-in

About the Eclipse Plug-in

Application Designer's Eclipse plug-in allows you to edit Application Designer layouts directly inside Eclipse. When the plug-in has been installed and configured, you can use the Layout Painter and some other Application Designer tools in Eclipse.



Installing the Eclipse Plug-in

The plug-in is delivered with the Application Designer software. After the installation of Application Designer, a directory with the name *EclipsePlugin* is available in your `<installdir>`*/tomcat/webapps/cis* directory.

The Eclipse plug-in consists of two parts which have to be installed separately: the common part and the GUI part. You must first install the common part.

To install the common part of the Eclipse plug-in

It is required that your servlet container has been started.

1 From the **Help** menu of Eclipse, choose **Software Updates > Find and Install**.

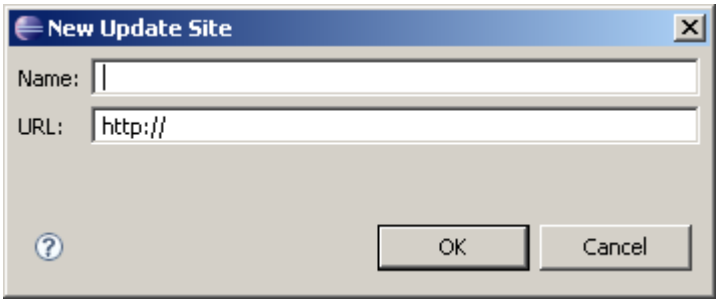
A dialog appears.

2 Select the option button **Search for new features to install**.

3 Choose the **Next** button.

4 On the resulting page, choose the **New Remote Site** button.

The following dialog box appears.



5 In the **Name** text box, specify a name of your choice.

6 In the **URL** text box, specify the path to the Eclipse plug-in. For example:

```
http://localhost:51000/cis/EclipsePlugin/com.softwareag.common
```

7 Choose the **OK** button.

To install the GUI part of the Eclipse plug-in

- Proceed as described above for the common part. In the **URL** text box, however, specify the following path:

```
http://localhost:51000/cis/EclipsePlugin/com.softwareag.cis.gui.swt
```

Creating an Eclipse Project for the Eclipse Plug-in

You create an Eclipse project in the same way as described in *Creating a Java Project in Eclipse*.

Configuring the Eclipse Project

Each Application Designer project has an *project/xml* directory in which the layouts are kept. This directory needs to be defined in Eclipse. To do so, you have to modify the properties of each Eclipse project that you create for the Eclipse plug-in.

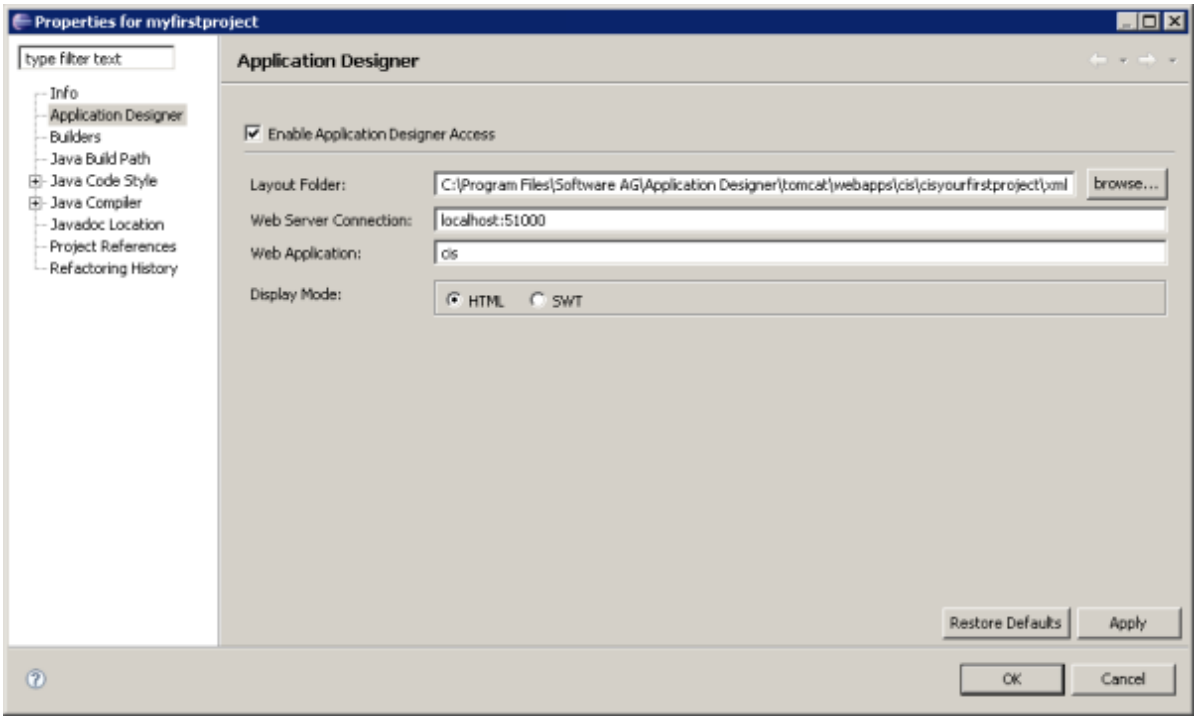
To configure the Eclipse project

1 Select the Eclipse project.

2 Invoke the context menu and choose **Properties**.

3 Enable the check box **Enable Application Designer Access**.

4 In the **Layout Folder** text box, specify the path to your project's *xml* folder.



5 If required, change the properties of the **Web Server Connection** and the **Web Application** in this dialog box. See the Tomcat documentation for more details.

6 Select the display mode for the **Layout Painter** and **Layout Tester**:

- **HTML**
  This mode makes use of the ActiveX plug-in of Eclipse in which Internet Explorer is running.
- **SWT**
  This mode makes use of SWT controls which are shown in an SWT client.

7 Choose the **OK** button.

The **Page Layouts** node is automatically created in the Navigator view. See below.

Elements of the Eclipse Plug-in

In order to work with the Eclipse plug-in, your servlet container must have been started. See *Starting the Servlet Container* in the *Development Workplace* documentation.

The Eclipse plug-in provides the following tools:

- Project Manager
- Layout Painter
- Style Sheet Editor
- Control Editor
- Monitoring
- Layout Tester

You use these tools in a similar way as those in the development workplace. See the *Development Workplace* documentation for further information. The descriptions in the *Development Workplace* documentation also apply to the Eclipse plug-in; the only difference in the Eclipse plug-in is that the tools are invoked in a different way (see below). The screenshots in the *Development Workplace* documentation also apply - with slight differences - to the Eclipse plug-in.

Project Manager

To invoke the Project Manager

1 From the **Window** menu, choose **Show View > Other**.

2 In the resulting **Show View** dialog box, expand the **Software AG Application Designer** node.

3 Select the entry **Tool - Project Manager** and choose the **OK** button.

A list of existing application projects is now shown.

Layout Painter

To create a new layout

1 In the Navigator view, select the **Page Layouts** node. This node contains all XML layout definitions.

2 From the **File** menu, choose **New > Other**.

3 In the resulting dialog box, expand the **Software AG** node.

4 Select the entry **Application Designer Layout** and choose the **Next** button.

5 In the resulting dialog box, enter the name of the file that is to contain your layout definition. The name must end with ".xml".

6 Select the layout template that you want to use.

7 Choose the **Finish** button.

The Layout Painter appears.

To open an existing layout

1 In the Navigator view, select the layout in the **Page Layouts** node.

2 Invoke the context menu and choose **Open With > Layout Painter**.

The Layout Painter appears.

Style Sheet Editor

To invoke the Style Sheet Editor

1 From the **Window** menu, choose **Show View > Other**.

2 In the resulting **Show View** dialog box, expand the **Software AG Application Designer** node.

3 Select the entry **Tool - Style Sheet Editor** and choose the **OK** button.

Control Editor

To invoke the Control Editor

1 From the **Window** menu, choose **Show View > Other**.

2 In the resulting **Show View** dialog box, expand the **Software AG Application Designer** node.

3 Select the entry **Tool - Control Editor** and choose the **OK** button.

A dialog appears, listing all available editor extensions.

4 Choose the editor extension that you want to open.

The contents of the editor extension are loaded into the Control Editor. You can now edit your editor extension.

Monitoring

To invoke the monitoring tool

1 From the **Window** menu, choose **Show View > Other**.

2 In the resulting **Show View** dialog box, expand the **Software AG Application Designer** node.

3 Select the entry **System Monitoring** and choose the **OK** button.

Layout Tester

The Layout Tester can be used to test the currently defined layout according to the display mode which is defined in the **properties** of the Eclipse project: it is either shown as in the browser or as in the SWT client.

To invoke the Layout Tester

1 Select the layout in the Navigator view.

2 Invoke the context menu and choose **Open With > Layout Tester**.

10 Debugging your Project Code

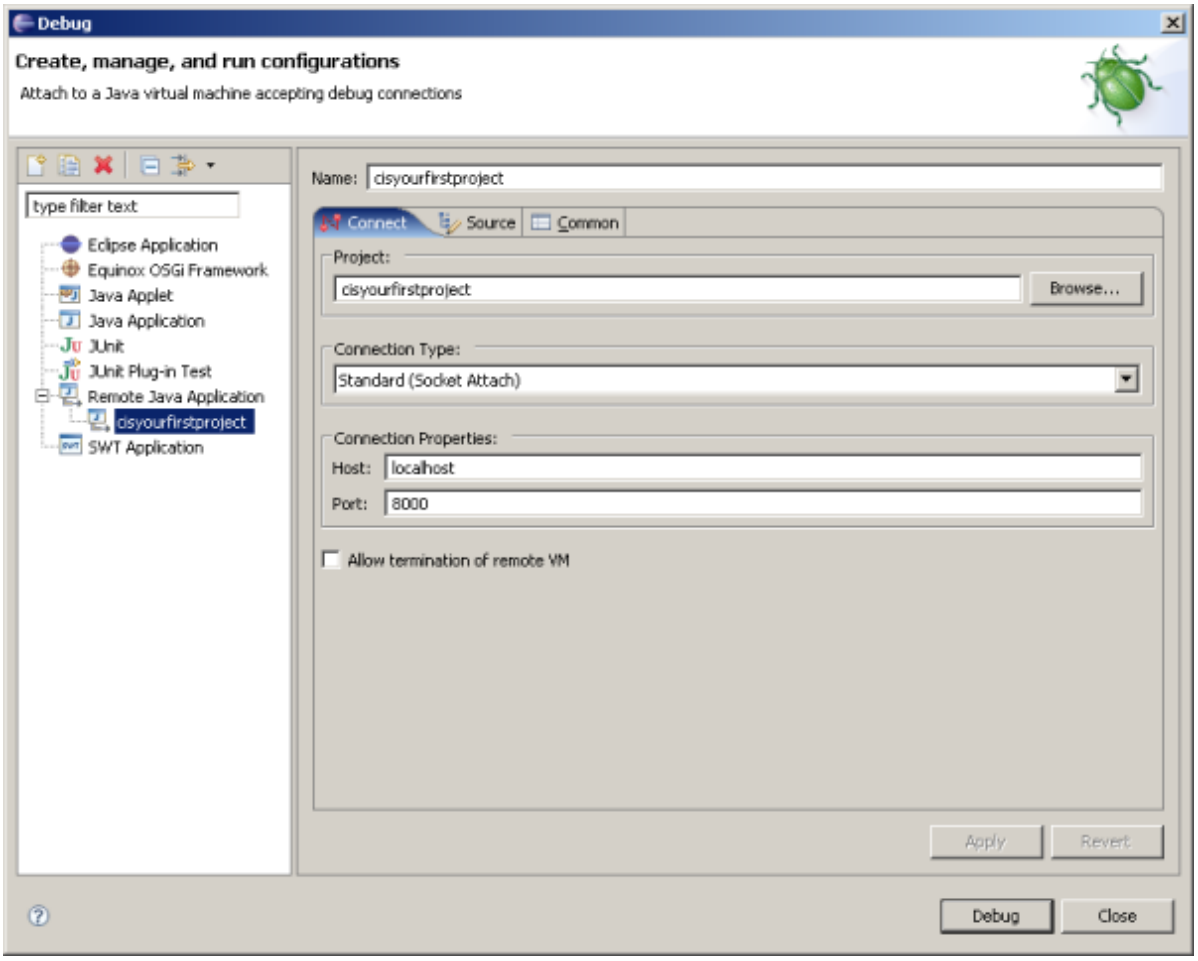
Eclipse contains an excellent debugging environment for debugging your Application Designer applications.

For debugging, you have to start Application Designer in remote debugging mode. This is done by executing the batch file *<installdir>/bin/CIS_debug.bat*. The port for debugging is defined in this batch file. If you want to use a port different from the standard port, you have to modify the batch file accordingly.

When you are using the Eclipse plug-in, it is required that the HTML mode is active. When you are debugging with the Layout Painter, you have to activate HTML in the preview configuration. When you are debugging with the **Layout Tester**, you have to activate the display mode HTML in the **properties** of the Eclipse project.

To configure the debug environment in Eclipse

1. Select the Eclipse project that you want to debug.
2. From the **Run** menu, choose **Debug**.
3. In the tree of the resulting dialog box, select **Remote Java Application**.
4. Choose the "New" button to create a debug configuration for the selected project.



5 Make sure that the port which is specified in **Connection Properties** group box is the same as defined in the file *CIS_debug.bat*.

6 Choose the **Apply** button.

7 After having configured the debug environment, execute the batch file *CIS_debug.bat*.

Or:

Choose the corresponding shortcut from the Windows **Start** menu. See also *Starting the Servlet Container* in the *Development Workplace* documentation.

8 Start debugging in Eclipse.

The debugger will connect to the virtual machine.

11 The Log Viewer

The Log Viewer is developed as an Eclipse plug-in. It is an independent tool designed to view single or multiple log files. A log file records activities and operations that occurred on a server/computer, maintaining an operational history of these activities. A log file is identified by its *.log* extension.

The Log Viewer allows the user to:

1. Specify refresh intervals.
2. Specify the number of lines to be displayed from a file.
3. Customize the way to view files by specifying filter conditions.
4. Save the log files added in the viewer so that the user is able access them easily the next time.
5. Provide extension points so as to enable other Eclipse plugins to contribute their log files for viewing.

Adding a New Log Viewer

To add a new Log Viewer

1 In the Eclipse view, open **Window > Show View > Other**.

2 In the **Show View** screen, select **Software AG > Log Viewer**.

The Log Viewer view appears. You may position it to appear at a convenient location on your screen.

3 To add a new Log Viewer, choose the **Add New Log Viewer** button.

The **Create Log Viewer** dialog appears, in which you can specify the file to be viewed and conditions for viewing.

4 If you are manually selecting a log file, specify the following details:

Specify the following details:

Log file to view is fixed	Select this option if the log file that must be viewed is fixed i.e. if its name does not change. The Log Viewer will view/read this file every time.
Name of the log file varies	Select this option if the name of the log file changes on some criteria; name of the log file might contain timestamp which is changed daily; hence the name of the log file also varies daily.
Select the log file directory	Browse and select the location of the file for the Log Viewer to view. The Log Viewer in this case picks up the file to view/read from this location. Specify this if you have selected the **Name of the log file varies** option.

Log file name pattern	Specify the date pattern of your log files. For example, *${TOMCAT_HOME}\logs\catalina.${yyyy}-${mm}-${dd}.log*. **Note:** Ensure that you know the log file naming conventions used in your application before specifying this information.
Add pattern	Select the date format from the available options in the drop down list.
Name of the view	Enter a name to identify this viewer.
Refresh Interval (in seconds)	The Log Viewer refreshes the view as per the interval you specify here.
Options	Choose the option to either view the complete log file or a few lines of it when opened the first time. If you select the latter option, you need to specify the **Total Lines** to be displayed.

Or:

If you are selecting a predefined log, select the log file from the **Select a Predefined log file** drop down list. On selection, the settings of the predefined file are auto populated in other fields. To learn more about predefined logs, see **About Predefined Logs**.

5 Choose the **Add Filter** button if you wish to add filters to the log files you view. Using these features, you can filter lines based on some conditions such as skipping the entire line or changing the display style in the view.

In the dialog box that appears, add line filters. For example, to view a line highlighted in black and the font color as green, specify the following filters:

- Select Contains text in the **If Line** field and specify the text as Info.
- Select the **Filter Action** as **Change the style of the line**.
- Select the **Foreground color** as green and the **Background color** as black.

6 Choose the **Ok** button to complete adding a new Log Viewer.

Editing or Removing a Log Viewer

You can edit an existing Log Viewer or remove it using the Log Viewer toolbar buttons. The procedure to edit a Log Viewer is the same as to add a Log Viewer except that you cannot edit the log file being viewed. You may access the **Edit Log Viewer**, **Remove Log Viewer** buttons from the Log Viewer toolbar or on right click of your mouse.

About Predefined Logs

The Log Viewer allows you to configure predefined logs. It displays predefined logs for selection in the **Select a Predefined log file** drop down list.

List of Predefined Logs

The following list of predefined log files appear for selection in the Log Viewer.

File Name	Purpose of the File	File Location
Application DesignerApplication ComposerNatural for Ajax Server Log	Runtime log file of the BPEL processing layer. It logs system generated and user defined information.	*\xciwebapps\xciservices\logs*
Application DesignerApplication ComposerNatural for Ajax Deployment Log	Deployment log file of the processing layer. **Note:** If the CentraSite Tomcat is started manually, the log file path is *C:\Documents and Settings\MYUSERNAME\AeBpelEngine\deployment-logs\aeDeployment.log*. Value provided for MYUSERNAME must match the username under which the Tomcat server runs.	*\AeBpelEngine\deployment-lo*
Application DesignerApplication ComposerNatural for Ajax Server Log	Runtime log file of the GUI and processing layers.	*\projects\log\serverLog_${yy*
CentraSite Tomcat Catalina Log	Catalina log file of the CentraSite Tomcat. This log file is only written if the CentraSite Tomcat is manually started using *catalina.bat*. If tomcat is started from **Windows** > **Services**, Catalina Service file is generated according to the parameters passed to the service start. (See Tomcat documentation for details.)	*${XCI_TOMCAT_HOME}\lo*