VisualAge for Java, Version 2.0

Tool Integrators for ISVs

Contents

Legal Notices

Notices

Programming Interface Information Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Tool Integrators for ISVs

External Tool Integration

Warning: Only experienced programmers should attempt to use the VisualAge for Java Tool Integrator. The Tool Integrator is ideally suited for use by ISVs.

With the VisualAge for Java Tool Integrator, you can integrate Java applications that reside on the file system such that they can be launched from within the IDE. The Java classes that make up the tool do not reside in the workspace, but are dynamically loaded from the file system. The Tool Integrator provides a standard framework for integration and upgrading of such Java tools.

The Tool Integrator only supports Java applications; that is, the entry point must be a main() method within a specific class. Applets cannot be integrated; instead, use the conventional IDE import facilities. To integrate an external class library or bean, use the Feature Integrator.

The IDE makes the following core class libraries available to the tool at run time:

- JDK 1.1.6
- JFC 1.2 (including Swing)
- VisualAge for Java Tool Integrator API

To ensure that any other classes required by tool are accessible, you should provide them with the tool itself in the file system. Because tools are kept in separate directories, they cannot share classes. Classes that are required by multiple tools must be copied into each tool directory. Unless a class belongs to one of the core class libraries (cited above) or is supplied by you in the tool base directory, there is no guarantee that any other classes will be available to be referenced, even if it is in the workspace. The IDE class path setting can be modified for an individual IDE using the tool, but you should make no general assumptions about accessing non-core class libraries.

An integrated tool is run using the tool's class files, which reside on the file system. These files are not loaded into the IDE, and are not visible to the user in the core IDE tools (for example, the Workbench).

When resolving class references, the VisualAge for Java class loader looks for classes in the following sequence:

1. The tool base directory.
2. The Tool Integrator API.
3. The JFC API.
4. The JDK API.

For classes in the tool base directory, do not use names that conflict with the Tool Integrator API, JFC, or JDK. Results are unpredictable.

If the class is not found in any of the above locations, the class reference is not resolved. The Tool Integrator does not look for classes anywhere else. If you suspect that a class reference is not being resolved and would like more information, make sure to select the 'Show system programs in debugger and console' option in the Debugging page of the Options dialog. The Console window may provide information on unresolved class references.

External Class Integration

Warning: Only experienced programmers should attempt to use the VisualAge for Java Feature Integrator. The Feature Integrator is ideally suited for use by ISVs.

With the VisualAge for Java Feature Integrator, you can conveniently integrate Java class libraries and beans, called features, that reside on the file system. Once a feature has been integrated and added to the workspace, it can be used within IDE the same way that any other classes in the IDE are used. The Feature Integrator allows class libraries and beans not shipped with VisualAge for Java to appear tightly integrated with VisualAge for Java environment, and provides a common framework for integration of such classes.

Keep in mind that the Feature Integrator is a convenience mechanism for classes that are not shipped with VisualAge for Java, and that these classes can also be manually imported into the repository and added to the workspace using conventional IDE facilities.

When the IDE starts, new features are automatically imported into the repository, and are made available in the Quick Start Feature selection dialog for 'Add Feature' and through the Visual Composition Editor palette 'Available...' selection dialog. Features can be loaded from the Quick Start window or from the Visual Composition Editor palette.

Similar to the Feature Integration mechanism, you can integrate file-based Java applications with the IDE by using the Tool Integrator.

Overview of the Tool Integrator API

Warning: Only experienced programmers should attempt to use the VisualAge for Java Tool Integrator API. This API is ideally suited for use by ISVs.

The VisualAge for Java Tool Integrator API provides a powerful Java class library for programmatically managing code and data from within the IDE. While a programmer can use the IDE SmartGuides, menus, and other user interface facilities to work with code, this API lets you manage the environment programmatically.

The API provides methods to perform the following tasks:

- browse the workspace and repository, and query objects
- load editions from the repository into the workspace
- import / export Java source, class files, and repository files
- work with type (class/interface) source code
- launch key IDE elements, such as various prompters and project/package/ type browsers
- manipulate the CLASSPATH of an executing tool
- access tool data

Why would you use the Tool Integrator API? Consider this scenario: you are developing an application in VisualAge for Java that requires a high degree of interaction with the file system. You need to export your code to the file system to properly test it. Using the conventional IDE user interface, you would use the Export

SmartGuide to deploy the class files. With the Tool Integrator API, you can write a tool that encapsulates all the user interface steps into a single class execution.

Package Overview

The API consists of a project called 'IBM IDE Utility class libraries'. When VisualAge for Java is installed, this project is only in the repository. You must add it to the workspace to work with the classes. The project contains two packages:

- com.ibm.ivj.util.base - base workspace and repository access.
- com.ibm.ivj.util.builders - code builders (direct source manipulation). This package contains an initial definition of builder support that is sufficient for code generators. In this case, the generators create an internal source code structure that is then saved in the IDE. Support is provided for preserving simple user changes (using structured comment blocks) across regeneration of the code.

Usage Notes

Code that uses the Tool Integrator API must be executed from within the IDE. There is no supported mechanism for getting to this API from outside the IDE, although tools that execute in a separate process can still make use of the API by writing a request handler that runs within the IDE. The request handler could be structured as a long-running tool extension. Examples include an RMI server or an http server with servlets that communicates with the external process through some private request/response protocol, calling the Tool Integrator API to handle the requests.

When developing code in the IDE that uses the Tool Integrator API, ensure that the classpath properties for the class include 'IBM IDE Utility local implementation' in the 'Extra directories path' list. The classes can be found in <prod dir>\ide\project_resources\IBM IDE Utility local implementation. Once completed tool classes are deployed, the API classes are automatically included on the classpath of each executing tool action.

The tools can store tool-specific data within VisualAge for Java. Any serializable Java object can be tool data.

tool options
: tool data associated with a tool key. Data is unique to each user workspace.

program element data
: tool data associated with a program element (project/package/class/interface) that resides in the workspace. Data is unique to each user workspace.

shared data
: tool data associated with a program element. Data is stored in the repository, so it can be shared in a team environment.

All three of these facilities are intended to be used to store relatively modest amounts of information. It is not intended to be used as a mass storage mechanism. Although the API does not impose any limits on data size, storing large data objects (especially shared data) can lead to performance degradation.

Access to both the workspace and repository is provided at a granularity level of project, package, class, or interface. With the exception of basic code-builder support, no API is provided to manipulate individual methods. Common Java facilities, such as introspection, can be used to access information at the method level.

To begin using the API, begin with the ToolEnv class, a bootstrap class used by a tool to gain access to the API services. ToolEnv.connectToWorkspace() returns a reference to a concrete implementation of the workspace. A repository is obtained from the workspace by calling one of the Workspace.getRepository() methods.

The classes and API are split in terms of code relating to the workspace and elements relating to the repository. The workspace includes projects, packages and types that have been added from the repository. All such program elements have corresponding editions in the repository. Thus, an API object representing a workspace project/package/type is distinct from the API object representing its corresponding edition in the repository. The relationship a workspace object has with other workspace objects is different than with repository objects. In the workspace, a project can include many packages, while a package can only belong to one project. In the repository, a project edition can include many package editions, and a package edition can belong to many project editions.

The repository can be directly queried for names of project and package editions. If a project or package name is supplied, the repository can also be queried for the corresponding list of the project or package editions.

Setting Up a Tool for Integration

A user-defined tool can be set up any time after VisualAge for Java has been installed. After the tool has been set up, the tool becomes visible from within the IDE once the IDE has been restarted.

The high-level steps to set up a tool for IDE integration are:

1. Create a subdirectory for the tool, called the base directory.
2. Copy all application classes, resource files, and HTML help into the base directory.
3. Create a control file that provides the IDE with details on the integration.
4. Create additional control files for other supported languages. (optional)

Create a Subdirectory for the Tool

Create a subdirectory from the ide/tools directory. This will be the base directory for the tool. The subdirectory name should be based on the tool's complete package prefix. When naming the subdirectory, replace the periods (.) in the package name with dashes (-). The following table shows two examples.

Complete package prefix	**Base directory for tool**
com.dingbat.widgets	com-dingbat-widgets
abc.enterprise.databuilder	abc-enterprise-databuilder

Unique base directory names distinguish integrated tools from each other.

Copy all Tool Files into the Base Directory

Copy all class files and resource files into the base directory, ensuring the appropriate subdirectory structure is maintained. You can build the tool itself in the IDE and export the class files to the tool base directory.

In addition, copy any tool documentation files (HTML) into the base directory.

Create a Control File

Create a control file, named default.ini, and copy it into the base directory. The control file provides integration information to the IDE. This file must be a flat ASCII file, cannot include blank lines, and must follow this format:

```
Name=<tool_name>
Version=<tool_version>
Help-Item=<menu_item_text>,<HTML_filename>
Menu-Group=<menu_group_text>
Menu-Items=<menu_item_group>[;...]
<menu_item_group>=<menu_item_text>,<start_class>,<context_type>
Quick-Start-Group=<Quick_Start_category>
Quick-Start-Items=<Quick_Start_group>[;...]
<Quick_Start_group>=<Quick_Start_text>,<start_class>
```

Parameter values cannot be delimited by quotation marks. In addition, commas (,) and semi-colons (;) cannot be used except when delimiting parameter values, as specified in the syntax above. Only the Name and Version entries are mandatory; all other entries are optional.

Name Identifies the integrated tool by name. The name can be any sequence of alphanumeric characters and punctuation (except commas, semi-colons, and colons), and should not exceed 40 characters in length.

Version
Specifies the released version of the tool.

Help-Item (optional)
Identifies a Help menu entry with the supplied menu text, linking to the main HTML file for the tool. The URL for the file is relative to the tool's base directory.

Menu-Group (optional)
Indicates that all menu items are to be placed in a submenu, and specifies the name of the submenu.

Menu-Items (optional)
Lists one or more menu items. Each menu item includes the text appearing on the menu, the class to be invoked when the menu item is selected, and an optional context type. The context type must be one of '-P' (project), '-p' (package), '-c' (class). If a context type is provided, the menu item is accessible from the Selection menu when that context has been selected. If no context type is provided, the menu item represents a general workspace action (accessible from **Workspace > Tools**). Each *start_class* must be the fully-qualified class name.

Quick-Start-Group (optional)
: Indicates that all Quick Start entries are to be placed under the specified Quick Start category. If multiple tools specify the same *Quick_Start_category*, they appear in the same Quick Start category. If no category is specified, all Quick Start item entries are automatically placed in a default tools Quick Start category.

Quick-Start-Items (optional)
: Lists one or more Quick Start items. Each item includes the text displayed in the Quick Start window, and the class to be invoked when the item is selected. Each *start_class* must be the fully-qualified class name.

A tool can be invoked from both the Tools menu and the Quick Start window.

Examples of Control Files

All examples below assume that the directory ide\tools\com-whammo exists, and includes all required files.

Example 1. Menu Integration

```
Name = Whammo Formatter
Version = 2.0
Menu-Items = Whammo Formatter,com.whammo.WStart1,
```

Example 2. Menu Group and Help Integration

```
Name = Whammo Formatter
Version = 2.0
Menu-Group = Whammo
Menu-Items = Run All,com.whammo.WStart1, ; Run on Selected
Classes,com.whammo.WStart2,-c
Help-Item = Whammo Formatter,index.html
```

Example 3. QuickStart Integration

```
Name = Whammo Formatter
Version = 2.0
Quick-Start-Items = Run All,com.whammo.WStart1;
Configure,com.whammo.WStartConfig
```

Example 4. QuickStart Group and Help Integration

```
Name = Whammo Formatter
Version = 2.0
Quick-Start-Group = Whammo
Quick-Start-Items = Run All,com.whammo.WStart1;
Configure,com.whammo.WStartConfig
Help-Item = Whammo Formatter,index.html
```

Create Control Files for Other Supported Languages (optional)

Create a control file for each language you want to support, and place these files into the base directory. To name these control files, follow the Java naming convention for locale support: <*language*>[_<*country*>].ini. For example, en_GB.ini would be the name of the British English control file, and fr.ini would be the name of the French control file.

When determining which control file to use, the Tool Integrator follows this sequence, using <*language*> and <*country*> values for the default locale:

1. Find the file <*language*>_<*country*>.ini
2. Find the file <*language*>.ini
3. Find the file default.ini

All control files must be in flat ASCII format, although parameter values can use non-ASCII Unicode values with \uxxxx notation.

Even if language-specific control files are supplied, a default.ini file must always be supplied. As well, only a change in the timestamp of default.ini can trigger the Tool Integrator to recognize tool updates.

Setting Up Class Libraries or Beans for Integration

An external class library or set of beans can be set up any time after VisualAge for Java has been installed. In VisualAge for Java, the external class libraries or beans are also called *features*. After features have been set up, they become visible within the IDE once the IDE has been restarted.

The high-level steps to set up classes for IDE integration are:

1. Create a subdirectory for the classes, called the base directory.
2. Copy all related files into the base directory.
3. Create a control file that provides the IDE with details on the integration.
4. Create additional control files for other supported languages. (optional)

Create a Subdirectory for the Feature

Create a subdirectory from the ide/features directory. This will be the base directory for the feature. The subdirectory name should be based on the classes' complete package prefix. When naming the subdirectory, replace the periods (.) in the package name with dashes (-). The following table shows two examples.

Complete package prefix	**Base directory for class library**
com.dingbat.widgets	com-dingbat-widgets
abc.enterprise.databuilder	abc-enterprise-databuilder

Unique base directory names distinguish features from each other.

Copy Related Files to the Base Directory

Two files are required to enable the Feature Integrator facility: projects.dat and default.ini.

The file projects.dat is a VisualAge for Java Version 2.0 repository that must contains Java projects. Only one versioned edition of each project can be included in the repository. You can create this file by importing the classes into an existing IDE project and then exporting the project as a repository (.dat) file. This ensures that the class data available at load time is compiled. Given bytecodes, the IDE can perform optimizations that reduce footprint and increase performance for importing and loading.

The file default.ini is the control file, and is described in the following section.

Resource files associated with the feature can be optionally provided. Follow these steps when providing resource files:

1. Create a subdirectory called project_resources under the feature base directory.
2. Under the project_resources directory, create a subdirectory for each project in the feature, with the same name as the project. The resource files should be copied under each project subdirectory, provided as expected by the referencing Java code.

When the feature is integrated with the IDE, all subdirectories of project_resources are copied into ide/project_resources, overwriting any project resource files that have the same names.

Create a Control File

Create a control file, named default.ini, and copy it into the base directory. The control file provides integration information to the IDE. This file must be a flat ASCII file, cannot include blank lines, and must follow this format:

```
Name=<feature_name>
Version=<feature_version>
Help-Item=<menu_text>,<HTML_filename>
Palette-Items=<category_group>[;...]
<category_group>=<category_name>,<class_name>[,<class_name>...]
Prereq-Features=<base_directory>[< base_directory>]
```

Parameter values cannot be delimited by quotation marks. In addition, commas (,) and semi-colons (;) cannot be used except when delimiting parameter values, as specified in the syntax above. Only the Name and Version entries are mandatory; all other entries are optional.

Name Identifies the integrated feature by name. The name can be any sequence of alphanumeric characters and punctuation (except commas, semi-colons, and colons), and should not exceed 40 characters in length.

Version
Specifies the released version of the feature. Feature versions should not be confused with the versioned edition names of projects and packages in the repository file, as there is no relationship between them.

Help-Item (optional)
Identifies a Help menu entry with the supplied menu text, linking to the main HTML file for the feature. The URL for the file is relative to the feature's base directory.

Palette-Items (optional)
Lists bean categories and their corresponding beans that will be added to

the Visual Composition Editor beans palette. Beans in the same category should be separated by commas, while each category should be separated by a semi-colon (;).

Prereq-Features (optional)
Specifies one or more features that are prerequisites of this feature. A prerequisite feature is specified by its base directory name. Prerequisite features are loaded when this feature is loaded. After the feature has been unloaded, however, prerequisite features must be unloaded separately.

Examples of Control Files

Example 1. Class Library Integration with Help

```
Name = Factory API
Version = 3.0
Help-Item = Factory API, index.html
```

Example 2. Class Library Integration with Help and One Palette Category

```
Name = Factory API
Version = 3.0
Help-Item = Factory Beans, beans.html
Palette-Items = Factory, com.factory.Generator1,com.factory.Generator2
```

Example 3. Class Library Integration with Help and Two Palette Categories

```
Name = Factory API
Version = 3.0
Help-Item = Factory Beans, beans.html
Palette-Items = Factory1, com.factory.Generator1,com.factory.Generator2; Factory2,
com.factory.Mulcher1, com.factory.Mulcher2
```

Create Control Files for Other Supported Languages (optional)

Create a control file for each language you want to support, and place these files into the base directory. To name these control files, follow the Java naming convention for locale support: <*language*>[_<*country*>].ini. For example, en_GB.ini would be the name of the British English control file, and fr.ini would be the name of the French control file.

When determining which control file to use, the IDE follows this sequence, using <*language*> and <*country*> values for the default locale:

1. Find the file <*language*>_<*country*>.ini
2. Find the file <*language*>.ini
3. Find the file default.ini

All control files must be in flat ASCII format, although parameter values can use non-ASCII Unicode values with \uxxxx notation.

Even if language-specific control files are supplied, a default.ini file must always be supplied. As well, only a change in the timestamp of default.ini can trigger the IDE to recognize feature updates.

Feature Setup at IDE Startup

1. All Java projects in the feature's projects.dat file are copied to the repository. If a problem occurs in either copying the feature to the repository or adding it to the workspace, the feature's default.ini is renamed to default.$$$ in order that the IDE will not encounter this problem every time it starts. As well, an error message is produced.
2. If the feature includes a project_resources directory, all subdirectories of this directory are copied to ide/project_resources. Any existing files in ide/project_resources with the same name are overwritten, so take care when naming your resource files.
3. Information from the default.ini is saved in the workspace registry of installed features.

If a feature is successfully installed, all feature files still remain on the file system. This way, the feature can be easily re-loaded if the workspace needs to be changed or is corrupted.

Adding a Feature to the Workspace and Visual Composition Editor Palette

Once a feature has been successfully integrated, you must still add it to the workspace in order to use it.

1. From the IDE, select F2 to bring up the Quick Start window.
2. Select **Features > Add Feature** and select **OK**.
3. From the selection dialog, select the feature you would like to add to the workspace and then select **OK**.

The feature is now added to the workspace and any beans specified in a control file Palette-Items entry appear on the Visual Composition Editor palette.

You can also add the feature from the Visual Composition Editor palette:

1. From the palette pulldown, select **Available...**
2. From the selection dialog, select the Feature to be added and then select **OK**.

After a feature has been added, it can be deleted through the Quick Start window.

Running the Tool from within the IDE

Once integrated, a user-defined tool that resides on the file system can be launched in two ways. In each case, the main() method for the application is called.

1. From the IDE Tools menu. The tool can have a single menu entry off the Tools menu, or a submenu that contains several menu entries. Each menu entry is associated with a Java application that resides in the base directory of the tool.
2. From the Quick Start window. The tool can have an entry under its own category, or be placed under the default tools Quick Start category.

The tool is launched via a mechanism that emulates a Java interpreter command. Imagine that the class is launched by a command with the following syntax:

java <*class_name*> [*selection_group*]

where:

class_name
The name of the application class file, which must contain a main() method.

selection_group
The selection context with the appropriate switch, if there is a selection context. See below for details.

Selection Context and Selection Group

An application may have been set up such that it can only be launched when a specific program element type is selected. The group of items that are currently selected is called the *selection context*. The selection context can be either one or more projects, one or more packages, or one or more classes. The names of the selected program elements are passed as parameters to the application's main() method, as well as a switch indicating the program element type.

The invocation switches are:

-P (for Projects)
-p (for Packages)
-c (for Classes)

From the selection context, the Tool Integrator constructs a selection group. For example, if the selection context includes two packages, myPack1 and myPack2, the selection group is '-P 'myPack1' 'myPack2''. Items in the selection context may contain spaces because the command creates a single string from the name of the selected item. For example, if the projects 'Banking Facility' and 'Java Class Libraries' have been selected, the selection group is '-P 'Banking Facility' 'Java Class Libraries''. Default package names are passed as they appear in the IDE. For example, the Animator class that is in the default package for Sun JDK Animator would be passed as 'Default package for Sun JDK Animator.Animator'.

Each invocation of a tool passes a single selection context to the application. The same tool can support multiple selection contexts for different invocations.

You are responsible for handling all input parameters passed to the tool as a result of launching the tool with a selection context. You are also responsible for managing any restrictions on the selection context, and to deal with any exceptions that result from handling the input parameters.

Limitation: do not try to concurrently execute two or more tools that use the same classes, where one tool loads the classes from the workspace and the other tool loads the classes from the file system. This may be a typical scenario for tools that generate code against a set of runtime libraries and also use the same set of libraries during tool execution. The libraries must be added to the workspace in order for the generated code to be usable. The tool could have also included the same libraries in the tool installation directory tree for its own execution. This limitation manifests itself as an exception in one or more of the executing programs. Depending on the nature of the code, this is typically either an exception indicating a class could not be found, or a cast exception. Simply terminate the offending programs and run the actions sequentially. See the Release Notes for more details.

Updating and Removing Integrated Tools and Classes

To update a tool or feature, copy any new files into the base directory or appropriate subdirectories, overwriting the existing files. The system timestamp on the control file (default.ini) must have also changed for the updates to take effect. As well, the IDE must be restarted for the updates to take effect.

Although this update process is typically used to replace an existing version of a tool or feature with a newer one, it is possible to make available multiple versions of the tool or feature. The easiest way to support multiple versions is to treat each version as a distinct tool or feature. Essentially, you need to place each version in a unique base directory. It is also advisable to provide a unique Name entry in each control file so that each version can be distinguished by the user in the IDE.

To remove a tool from the IDE, you need to:

1. Delete the tool base directory.
2. Replace your current workspace with a new workspace:
   a. Ensure that all program elements are versioned. If you are working in a team environment and your program elements are ready for general availability to the rest of the team, you may also choose to release them. By releasing, it will quicker to add program elements to the workspace after you have refreshed the workspace.
   b. Shutdown the IDE.
   c. From the product CD, unzip ide.zip into the <prod dir>\ide\program directory. This replaces the workspace file. Reposity data is left unaffected.
   d. Start the IDE.
   e. Add program elements back into the workspace.

To remove a feature from the IDE,

1. Delete the feature base directory.
2. Remove the classes from the workspace.

Example: Using the Tool Integrator API to Associate Data with a Package

Consider the following situation: you want to attach reminders to yourself about packages that you write in the IDE. You can use the Tool Integrator API to write a tool that lets you input text, associate it with a package, and retrieve the text.

This hypothetical tool, called TextReminder, can be launched against a package. It presents a simple dialog with a text area for inputting your comments, and buttons for applying and canceling changes.

When data is associated with a workspace object (in this example, a package), it must be uniquely identified by a key, which is a java.lang.String object. This allows multiple tools to store information associated with the same workspace object. For simplicity, we suggest you use the fully qualified tool name as the key.

Note: Any data associated with an IDE object must implement the Serializable interface. In this example, we use String, which does implement Serializable.

The class for the tool does the following things:

1. Takes the fully qualified name of the selected package as a parameter, `-p packagename`.
2. Gets a workspace object by using the ToolEnv.connectToWorkspace() method.
3. Retrieves the Package object for the given package name by using the Workspace.loadedPackageNamed() method.
4. Checks to see if the package has a text comment associated with it by using the Package.testToolWorkspaceData() method, passing in the key 'TextReminder'. This method returns a boolean indicating whether any data is associated with the package for this tool.
5. If data does exist, retrieves it using the Package.getToolWorkspaceData() method, with the key as a parameter. This returns a ToolData object.
6. The ToolData object consists of the key and the text. The tool extracts the text by using the ToolData.getData() method, which returns a Serializable object. In this case, since we are storing and retrieving String objects, cast the returned object to a String.
7. Outputs the String to the dialog.
8. When changes have been made to the output text, the tool writes the text back to the ToolData object by using ToolData.setData(). Then it applies changes to the package object by using Package.setToolWorkspaceData().
9. If you want remove any associated data (so that Package.testToolWorkspaceData() returns false), use Package.clearToolWorkspaceData().

The resulting class might look like this:

```
private static String key = 'TextReminder';

public static void main(String[] args) {
  boolean saveTheComment = true;
  boolean deleteTheComment = false;
  String pkgName = args[1];

  try {
    Workspace theWS = ToolEnv.connectToWorkspace();
    Package pkg = theWS.loadedPackageNamed(pkgName);
    ToolData toolData = null;
    String userComment = new String();

    if (pkg.testToolWorkspaceData(key)) {
      toolData = pkg.getToolWorkspaceData(key);
      userComment = (String)toolData.getData();
    }
    else
      toolData = new ToolData(key,userComment);

  // Code that allows the user to view/edit
  // the comment omitted.

    if (saveTheComment) { // likely dependent on button click
      toolData.setData(userComment);
      pkg.setToolWorkspaceData(toolData);
    }
    else if (deleteTheComment)  // likely dependent on button click
      pkg.clearToolWorkspaceData(key);
    // otherwise the user is not modifying the comment.
    }
 catch(Exception ex) { }
}
```

Preparation before Testing the Example

To test the example in the IDE before deploying it, you need to ensure that the execution classpath in the IDE has been properly set up, and that an execution context is passed to main() during test execution.

To set up the classpath:

1. Open the Properties dialog on the class, and on the Class Path page, select the **Project path** checkbox and then press **Compute Now**.
2. Select the **Extra directories path** checkbox and then select **Edit**. From the dialog, select **Add Directory...** and then select ide\project_resources\IBM IDE Utility local implementation.

To set up a test invocation context, open the Properties dialog on the class, and on the Program page, enter the command-line arguments expected by the main() method. For example '-p my.package'.