Archivists' Toolkit (AT) Plugin Developer Guide

version 1.21

AT Development Team

January 01, 2011

Introduction

Beginning with version **1.5.9**, plugin support was added to the Archivists Toolkit, AT. This guide presents an overview for how to write plugins for the AT. Since the AT uses the JPF framework to provide support for plugins, the reader is first advised to visit the JPF project site and read the tutorial by Jean Lazarou on using the JPF in applications. Like the example applications in the tutorial, the AT makes use of the JPF framework in the simplest way possible. This means that it does not allow for dynamic plugin loading, nor does it have a nice plugin manager for installing and uninstalling plugins.

The plugin framework was added to the AT as way to provide a flexible, simple mechanism by which developers can extend the functionality of the program without having to "touch" the underlying code base. Functionality that can readily be added using plugins, includes custom import/export modules, report generators, and even custom record (Resources, Accessions, Subjects, Names, and Digital Objects) editors. Plugins that can be embedded into existing record editors can also be developed (They will show up as an additional tab in the given editor). Essentially, plugins can provide a convenient method in which to tailor the AT towards an institution's workflow.

Plugin Structure

Here is an example of the representative file structure for a typical AT plugin and the relevant directories found in the AT install directory.

```
[AT_HOME_FOLDER]/
  +-  lib/*.jar
  +-  plugins/


[PLUGIN_HOME_FOLDER]/
  +- lib/
  +- src/
  +- classes/
  plugin.xml
```

Here are the explanations of what is contained within the folders or files:

[AT_HOME_FOLDER]/lib

This contains all the AT jar library, as well as all the other libraries used by the AT. The files in folder should to be in your classpath in order to build plugins. Plugins will automatically have access to all these libraries; so there is no need to add them to the **lib** folder of the plugin.

[AT_HOME_FOLDER]/plugins

Plugins which have been packaged as *.zip files go into this folder and are loaded when the application starts up.

[PLUGIN_HOME_FOLDER]/lib

Libraries which are specific to the plugin go into this folder.

[PLUGIN_HOME_FOLDER]/src

The source code of the plugin goes here. It is not required to be distributed with the packaged plugin.

[PLUGIN_HOME_FOLDER]/classes

The compiled classes of the plugins are kept here. The name of this folder does not matter as the location of the classes are specified in the plugin.xml file. It is also unnecessary to create jars of the compiled plugin classes since everything is going to be packaged up as a zip file anyway.

[PLUGIN_HOME_FOLDER]/plugin.xml

This is the plugin manifest file and is used by the JPF when loading the plugin. The name of this file cannot change.

Sample Plugin Manifest Files

As mentioned above, the AT makes use of the JPF in the most simplistic way. The manifest file for AT plugins are pretty simple. Here is one for a plugin that loads an external library which is not part of the libraries the AT uses. Consult the tutorial on the JPF site for a more in-depth discussion of the manifest file.

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 1.0"
"http://jpf.sourceforge.net/plugin_1_0.dtd">
<plugin id="org.archiviststoolkit.plugin.demo1.atplugindemo1" version="1.0"
class="org.archiviststoolkit.plugin.demo1.ATPluginDemo1">
        <runtime>
                <library id="atplugindemo1" path="/" type="code"/>
                <library id="at2ead" path="lib/at2ead.jar" type="code"/>
        </runtime>
</plugin>
```

Plugin Interface

In order to be recognized as a plugin, a program must extend "org.java.plugin.Plugin" and implement the "ATPlugin" interface. Please take a look at comments in the ATPlugin.java at the end of this document for a good overview of the inner workings. Note: since "org.java.plugin.Plugin" is an abstract class, the doStart() and doStop() method needs to be implemented, but for all practical purposes these can be empty since they are not used within the AT context.

The current plugin interface defines methods that supports three categories plugins. The categories are "import" for plugins that perform importing of legacy data, editor/viewer for plugins that allow viewing/editing of records, and "default" for general purpose plugins. Import plugins are displayed under the import menu, while default plugins go under the Plugin menu. The plugin menu is only visible when plugins are detected in the "plugins" directory. Viewer/editor plugins are used in place of the default AT record editors. Such plugins can be used to implement read only viewers for records or for editors which is more tailored for a particular workflow.

Since a plugin may be capable of performing more than one task, they are able to define multiple tasks and have those displayed under a submenu in either the Plugins or Import menu. For example if an import plugin named "My Importer" defines three task called "Import A", "Import B", and "Import C", then each of those task will be under a submenu called "My Importer". This functionality is accomplished through use of the getTaskList() and doTask() methods.

Plugin Utility Class

Since the AT currently make use of Hibernate to persist Java objects to the backend database, a utility class is provided that enable plugins to save Java objects to the database without having to worry about object relational mapping. This is accomplished using the XStream library to serialize objects to XML, which is then saved to the database. The XML encoded objects are retrieved from the database and converted back to objects. This class also provides methods for saving AT records and for validating them. Take a look at comments in ATPluginUtils.java for more detailed description of what this class does.

Packaging and Installing a Plugin

In order to be recognized as a plugin, the Java classes, plugin.xml, and any other resource files needed **Must Be** packaged as a zip file. Once this zip file is placed in the plugins folder of the AT, it should be available for use once the application is restarted. At this point, there is no support for dynamic loading of plugins in the AT but the underlying JPF framework does support this functionality so future AT versions may have that capability.

Example Plugins

Please go the following site for a list of plugins that have been developed for the AT. If you would like to get a plugin you developed posted to this site, please send an email to info@archiviststoolkit.org with the subject "Add Plugin".

Command Line Interface (CLI) Plugins

Starting with **version 2.0 update 6**, the AT now supports command line interface plugins. These are intended to be loaded by the command line interface for the AT, **atcli**, without having to start-up the main GUI program. Please note that **atcli** doesn't provide any functionality on its' own, but serves merely to load plugins which provide the functionality. It also handles the process of login into the AT database (provided a valid database password is provided), and allocating up to 1GB of memory to run the plugin. It also handles setting of the Classpath. To use the command line interface, one needs to simple install the plugin as usual and execute the following command (Mac OSX command is **atcli.command**, on Linux it is **atcli**).

atcli.exe *"database password" "plugin task" [additional plugin parameters separated by spaces]*

Where "database password" is the password of the AT database, and "plugin task" the task the plugin has registered in the getTaskList() method. Those two parameters are required, otherwise the program will exit. Additional parameters can entered, separated by spaces. See the code below to see how to access this parameters from within the plugin.

See the code below to see what methods need to be defined at minimum for a CLI plugin. Please note that a plugin can be both a CLI and regular AT plugin. This is useful in a situation where the plugin developer wants to provided user with a graphical way to configure the plugin.

Rapid Data Entry (RDE) Plugins

Also in **version 2.0 update 6**, support for Rapid Data Entry plugins have been added. These plugins are sub-plugins which are added to the Rapid Data Entry Screen drop-down menu in the resource editor. They can either launch a dialog, or execute program logic on the resource record, or currently selected resource component.

Since they are sub plugins, they have to be added to a plugin which has defined ATPlugin.EMBEDDED_EDITOR_CATEGORY as a category, and the editor type must contain ATPlugin.RAPID_DATA_ENTRY_EDITOR. The getRapidDataEntryPlugins() method must also be implemented since it returns the Hashmap containing any RDE plugins, keyed by their title.

To be defined has an RDE plugin the class must implement the RDEPlugin interface which is listed below. Please note the multiple RDE plugins can be returned in this Hashmap making the system very versatile.

See the source code for BatchDO, a plugin which automates the creation and attachments of digital objects to resource components, to see how to implement support for RDE plugins.

Authentication Plugins

In **version 2.0 update 10**, support for authentication plugins have been added. These plugins are intended to allow authentication (not authorization) of AT users through a single sign-on system, such as a LDAP directory server. As such, users still need to be entered into the AT database as normal, just that the entered passwords are checked against a central server through the plugin.

With a plugin installed, the login process is as follows. First, authentication credentials are checked against those in AT database, and if that is successful, then the applications continues to load. If that fails, then authentication is tried using the plugin, and if successful, then the application will load as normal. This approach allows for using both AT authenticated, and plugin authenticated users simultaneously.

The minimum requirements for a plugin to be used during the authentication process, is that has to have ATPLugin.AUTHENTICATION_CATEGORY defined as a category and implement the doTask(String task, String[] taskParams) method. The String array taskParams contains the username and password at index 0 and 1 respectively. If authentication succeeded, then "true" should be returned, otherwise false should be returned. See the code for sample ldap authentication plugin, LoginPlugin. This can server as a starting point to implementing your own plugin, but may be already functional out of the box since it simple tries to authenticate against a ldap server.

Demo CLI Plugin Source Code

```
   1:package org.archiviststoolkit.plugin.cli;
   2:
   3:import org.archiviststoolkit.ApplicationFrame;
   4:import org.archiviststoolkit.util.StringHelper;
   5:import org.archiviststoolkit.exporter.EADExport;
   6:import org.archiviststoolkit.editor.ArchDescriptionFields;
   7:import org.archiviststoolkit.model.Resources;
   8:import org.archiviststoolkit.mydomain.DomainAccessObject;
   9:import org.archiviststoolkit.mydomain.DomainEditorFields;
  10:import org.archiviststoolkit.mydomain.DomainObject;
  11:import org.archiviststoolkit.mydomain.ResourcesDAO;
  12:import org.archiviststoolkit.plugin.ATPlugin;
  13:import org.archiviststoolkit.swing.InfiniteProgressPanel;
  14:import org.java.plugin.Plugin;
  15:import org.hibernate.Session;
  16:
  17:import javax.swing.*;
  18:import java.awt.*;
  19:import java.util.HashMap;
  20:
  21:/**
  22: * Created by IntelliJ IDEA.
  23: * User: nathan
  24: * Date: Apr 19, 2010
  25: * Time: 8:01:58 PM
  26: *
  27: * This is a sample plugin which demonstrate how to develope
  28: * Command Line Interface, CLI, plugins for the AT. It also demonstrate
  29: * techniques for saving memory which is critical when batch processing
  30: * large amount of records.
  31: */
  32:public class cliPlugin extends Plugin implements ATPlugin {
  33:    /**
  34:     *  get the category(ies) this plugin belongs to. For plugins that
  35:     *  are only used through the CLI then the below is fine. Additional
  36:     *  categories can be defined if this plugin is also to be used with
  37:     *  the main AT program
  38:     */
  39:    public String getCategory() {
  40:        //return ATPlugin.CLI_CATEGORY + " " + ATPlugin.DEFAULT_CATEGORY;
  41:        return ATPlugin.CLI_CATEGORY;
  42:    }
  43:
  44:    // get the name of this plugin
  45:    public String getName() {
  46:        return "Demo CLI Plugin v1.0";
  47:    }
  48:
  49:    /**
  50:     * Method to actually execute the logic of the plugin. It is
  51:     * automatically called by the atcli program so it needs to be
  52:     * implemented
  53:     *
  54:     * @param task The task that is passed in as the second argument in
  55:     * the command line parameters
  56:     */
  57:    public void doTask(String task) {
  58:        // show the command line parameters
  59:        String[] params =
org.archiviststoolkit.plugin.ATPluginFactory.getInstance().getCliParameters();
  60:        for(int i = 0; i < params.length; i++) {
  61:            System.out.println("Parameter " + (i+1) + " = " + params[i]);
  62:        }
  63:
  64:        /*
  65:        * do a test by getting all the resource rocords and print name out
  66:        * their name title and exprt as EAD. Thanks for Cyrus Farajpour
  67:        * for providing some of this code
  68:        */
  69:
  70:        DomainAccessObject access = new ResourcesDAO();
  71:
```

```
  72:         // get the list of all resources from the database
  73:         java.util.List<Resources> resources;
  74:
  75:         try {
  76:             resources = (java.util.List<Resources>) access.findAll();
  77:         } catch (Exception e) {
  78:             resources = null;
  79:         }
  80:
  81:         int recordCount = resources.size();
  82:
  83:         System.out.println("\nNumber of records found: " + recordCount);
  84:
  85:         // Get the path where to export the files to from the command line arguments
  86:         String exportRootPath = params[2];
  87:
  88:         //dummy progress panel to pass into convertResourceToFile
  89:         InfiniteProgressPanel fakePanel = new InfiniteProgressPanel();
  90:
  91:         // print out the resource identifier and title and export as ead
  92:         System.out.println("Exporting EADs ");
  93:
  94:         // Get the runtime for clearing memory
  95:         Runtime runtime = Runtime.getRuntime();
  96:
  97:         // get the ead exporter
  98:         EADExport ead = new EADExport();
  99:
 100:         for(int i = 0; i <= recordCount; i++) {
 101:             try {
 102:                 // load the full resource from database using a long session
 103:                 Resources resource =
(Resources)access.findByPrimaryKeyLongSession(resources.get(i).getIdentifier());
 104:
 105:                 System.out.println("[ " + (i+1) + " ] " +
resource.getResourceIdentifier() + " : " + resource.getTitle());
 106:
 107:                 String fileName =
StringHelper.removeInvalidFileNameCharacters(resource.getResourceIdentifier());
 108:                 java.io.File file = new java.io.File(exportRootPath + fileName +
".xml");
 109:                 file.createNewFile();
 110:                 ead.convertResourceToFile(resource, file, fakePanel, false, true, true,
true);
 111:
 112:                 // since we no longer need this resource, set it to null
 113:                 // close the session in an attemp to save memory
 114:                 resources.set(i, null);
 115:             } catch (Exception e) {
 116:                 System.out.println(e);
 117:             }
 118:
 119:             // close the long session. This is critical to saving memory. If it's
 120:             // left open then hinernate caches the resource records even though
 121:             // we don't need them anymore
 122:             access.getLongSession().close(); // close the connection
 123:
 124:             // run GC to clear some memory after 10 exports, not sure if this is
 125:             // really needed but running GC cost little in time so might as well?
 126:             if(i%10 == 0 && i != 0) {
 127:                 System.out.println("\nRunning GC =>\nFree Memory Before " +
runtime.freeMemory());
 128:                 runtime.gc();
 129:                 System.out.println("Free Memory After " + runtime.freeMemory() + "\n");
 130:             }
 131:         }
 132:
 133:         System.out.println("Finished Exporting ...");
 134:     }
 135:
 136:     /**
 137:      * Method to get the list of specific task the plugin can perform. Only
 138:      * the task at position 0 in the string array is checked in CLI plugins
 139:      *
 140:      * @return String array containing the task(s) the plugin is registered
```

```
141:     * to handel.
142:     */
143:    public String[] getTaskList() {
144:        String[] tasks = {"test"};
145:
146:        return tasks;
147:    }
148:
149:    /*
150:     *
151:     * Method below this point do not need to implemented in a command line plugin,
152:     * but can be for those that also have GUIs, for example for configuration.
153:     *
154:     */
155:
156:    public void setApplicationFrame(ApplicationFrame mainFrame) { }
157:
158:    public void showPlugin() { }
159:
160:    public void showPlugin(Frame owner) { }
161:
162:    public void showPlugin(Dialog owner) { }
163:
164:    public HashMap getEmbeddedPanels() { return null; }
165:
166:    public void setEditorField(ArchDescriptionFields editorField) { }
167:
168:    public void setEditorField(DomainEditorFields domainEditorFields) { }
169:
170:    public void setModel(DomainObject domainObject, InfiniteProgressPanel monitor) { }
171:
172:    public void setCallingTable(JTable callingTable) { }
173:
174:    public void setSelectedRow(int selectedRow) { }
175:
176:    public void setRecordPositionText(int recordNumber, int totalRecords) { }
177:
178:    public String getEditorType() {return null; }
179:
180:    protected void doStart() { }
181:
182:    protected void doStop() { }
183:}
```

ATPlugin Interface Source Code

```
   1:package org.archiviststoolkit.plugin;
   2:
   3:import org.archiviststoolkit.ApplicationFrame;
   4:import org.archiviststoolkit.editor.ArchDescriptionFields;
   5:import org.archiviststoolkit.swing.InfiniteProgressPanel;
   6:import org.archiviststoolkit.mydomain.DomainObject;
   7:import org.archiviststoolkit.mydomain.DomainEditorFields;
   8:
   9:import javax.swing.*;
  10:import java.awt.*;
  11:import java.util.HashMap;
  12:
  13:/**
  28: * The simple plugin interface which all AT Plugins need to be implement in
  29: * order to be loaded into the AT
  30: * <p/>
  31: * Created by IntelliJ IDEA.
  32: *
  33: * @author: Nathan Stevens
  34: * Date: Feb 10, 2009
  35: * Time: 11:03:24 AM
  36: */
  37:public interface ATPlugin {
  38:    // plugin will be displayed under the plugin menu
  39:    final String DEFAULT_CATEGORY = "default";
  40:
  41:    // plugin will be displayed under the import menu
  42:    final String IMPORT_CATEGORY = "import";
  43:
  44:    // plugin will be displayed under the tool menu
  45:    final String TOOL_CATEGORY = "tool";
  46:
  47:    // plugin will be used to view records instead of the default editor
  48:    final String VIEWER_CATEGORY = "view";
  49:
  50:    // plugin will be used to view/edit records instead of the default editor
  51:    final String EDITOR_CATEGORY = "edit";
  52:
  53:    // plugin will be used embedded into an existing domain editor
  54:    final String EMBEDDED_EDITOR_CATEGORY = "embedded";
  55:
  56:    // plugin will be used with the command line interface of the AT. This
functionality will require additional code
  57:    final String CLI_CATEGORY = "cli";
  58:
  59:    /*
  60:    The list of editor types. They can be combined to specify a plugin that
  61:    supports viewing and editing multiple type of records. For example, a
  62:    developer can choose to create a plugin that supports viewing and
  63:    editing of Names, Subjects, and Digital Object Records with the following
  64:    string:
  65:    editorType = NAMES_EDITOR + " " + SUBJECT_EDITOR + " " + DIGITALOBJECT_EDITOR;
  66:    */
  67:
  68:    // plugin that can edit or view resource records
  69:    final String RESOURCE_EDITOR = "resource";
  70:
  71:    // plugin that can edit or view resource components
  72:    final String RESOURCE_COMPONENT_EDITOR = "component";
  73:
  74:    // plugin that can edit or view digital object records
  75:    final String DIGITALOBJECT_EDITOR = "digital";
  76:
  77:    // plugin that can edit or view name records
  78:    final String NAME_EDITOR = "name";
  79:
  80:    // plugin that can edit or view subject records
  81:    final String SUBJECT_EDITOR = "subject";
  82:
  83:    // plugin that can edit or view subject records
  84:    final String ACCESSION_EDITOR = "accession";
  85:
```

```
 86:     //plugin that can edit or view instant records
 87:     final String INSTANCE_EDITOR = "instance";
 88:
 89:     //plugin that can edit or view assessment records
 90:     final String ASSESSMENT_EDITOR = "assessment";
 91:
 92:     //plugin that loads in the RDE drop down menu
 93:     final String RAPID_DATA_ENTRY_EDITOR = "rde";
 94:
 95:     // plugin that can edit or view all the main AT records.
 96:     final String ALL_EDITOR = "all";
 97:
 98:     /**
 99:      * Method to get the category of the plugin
100:      * i.e. default, import, view or edit
101:      *
102:      * @return String that specify the category type
103:      */
104:     public String getCategory();
105:
106:     /**
107:      * Method to return the name of the plugin. If the plugin category
108:      * is either import or default, this name appears in the menu
109:      *
110:      * @return Returns the plugin names
111:      */
112:     public String getName();
113:
114:     /**
115:      * Method to set the application frame. In the AT, the application
116:      * frame provides a means to access the current worksurface and hence
117:      * the displayed records. This method is called every time a plugin is
118:      * selected in the menu.
119:      *
120:      * @param mainFrame The main AT application frame
121:      */
122:     public void setApplicationFrame(ApplicationFrame mainFrame);
123:
124:     /**
125:      * Method to display the plugin or do anything else the plugin
126:      * requires when selected from the plugin or import menu, if
127:      * it doesn't define a task list.
128:      */
129:     public void showPlugin();
130:
131:     /**
132:      * Method to display a plugin that needs a parent frame.
133:      * This method is not currently used in the AT so it
134:      * can be left blank.
135:      *
136:      * @param owner The parent frame of this plugin
137:      */
138:     public void showPlugin(Frame owner);
139:
140:     /**
141:      * Method to display a plugin that needs a parent dialog.
142:      * Not currently used in the AT, so it can be left blank.
143:      *
144:      * @param owner The parent dialog of this plugin
145:      */
146:     public void showPlugin(Dialog owner);
147:
148:     /**
149:      * Method to return a hashmap containing jpanels for plugins that are not
150:      * dialog or frames. This is used for plugins that are to be embedded
151:      * in a Domain Editor. The hasmap is keyed using the plugin names.
152:      * <p/>
153:      * The format of the plugin name can also be used to specify the location
154:      * and whether to remove a planel already at that location.
155:      * <p/>
156:      * Format to use is panel_name::location::yes, no, main
157:      * Examples >>
158:      * New Editor Panel::0::yes (replace panel at index zero with this one)
159:      * New Editor Panel::0::no (just insert the panel at zero)
160:      * New Editor Panel::0::main (In Subjects editor only remove all other
```

```
161:      * components in the panel and add this panel)
162:      *
163:      * @return The HashMap containing JPanels and their display name
164:      */
165:     public HashMap getEmbeddedPanels();
166:
167:     /**
168:      * Method to return hashmap containing RDEPlugin objects which are listed
169:      * under the RDE drop down menu in the resources editor. Such plugins
170:      * may launch a dialog or may just execute the some logic agains the
171:      * currently selected respurce component or resource record.
172:      *
173:      * @return The HashMap containing RDEPlugins and display names
174:      */
175:     public HashMap getRapidDataEntryPlugins();
176:
177:     /**
178:      * Method to set the editor field. This is used by embeddable
179:      * plugins so that they can gain access to public method of the
180:      * editor field.
181:      *
182:      * @param editorField The editor field the plugin is embedded into
183:      */
184:     public void setEditorField(ArchDescriptionFields editorField);
185:
186:     /**
187:      * Method to set the editor field. This is used by embeddable
188:      * plugins so that they can gain access to public method of the
189:      * editor field.
190:      *
191:      * @param editorField The editor field the plugin is embedded into
192:      */
193:     public void setEditorField(DomainEditorFields editorField);
194:
195:     /**
196:      * Method to do a specific task in the plugin. This method is
197:      * implemented by plugins that are can do multiple task.
198:      * For example an importer for multiple file types. Its called
199:      * each time a plugin that defines a task list is selected
200:      * from the import or plugin menu.
201:      *
202:      * @param task The task for the plugin to do
203:      */
204:     public void doTask(String task);
205:
206:     /**
207:      * Method to get the list of specific task the plugin can perform.
208:      * This task are displayed in submenus under either the import or
209:      * plugin menu.
210:      *
211:      * @return A list of task this plugin can perform
212:      */
213:     public String[] getTaskList();
214:
215:     /*
216:     * Methods below this point are to be implemented by plugins which
217:     * are used as DomainObject Viewers and/or Editors. The are always
218:     * called by the AT if the plugin category is defined as a
219:     * viewer or editor
220:     */
221:
222:     /**
223:      * Method to return the type of domain objects this plugin can edit or
224:      * view. This method is used by plugins that are implemented as an editor
225:      * for one or more of the main domain objects such Names, Subjects,
226:      * Accessions, Resources, and Digital Objects. Plugins that supports
227:      * the supports the viewing and/or editing of multiple record types
228:      * can be specify in the following manner: A plugin that supports viewing
229:      * and/or editing of Names, Subjects, and Digital Object Records
230:      * will return the following String:
231:      * editorType = NAMES_EDITOR + " " + SUBJECT_EDITOR + " " + DIGITALOBJECT_EDITOR
232:      * If a plugin returns ALL_EDITOR it means it can view
233:      * and/or edit all main AT records.
234:      *
235:      * @return The type or types of records an editor/viewer plugin can open.
```

```
236:     */
237:    public String getEditorType();
238:
239:    /**
240:     * Method to set the domain model. This is always called by the AT
241:     *
242:     * @param domainObject The domain object
243:     * @param monitor      The progress monitor
244:     */
245:    public void setModel(DomainObject domainObject, InfiniteProgressPanel monitor);
246:
247:    /**
248:     * Method to get the table from which the record was selected.
249:     *
250:     * @param callingTable The table containing the record
251:     */
252:    public void setCallingTable(JTable callingTable);
253:
254:    /**
255:     * Method to set the selected row of the calling table. This lets the
256:     * plugin know the current row selection
257:     *
258:     * @param selectedRow The selected row
259:     */
260:    public void setSelectedRow(int selectedRow);
261:
262:    /**
263:     * Method to set the current record number along with the total
264:     * number of records
265:     *
266:     * @param recordNumber The current record number
267:     * @param totalRecords The total number of records
268:     */
269:    public void setRecordPositionText(int recordNumber, int totalRecords);
270:}
```

RDEPlugin Interface Source Code

```
 1:package org.archiviststoolkit.plugin;
 2:
 3:import org.archiviststoolkit.model.ResourcesCommon;
 4:import org.archiviststoolkit.mydomain.DomainObject;
 5:import org.archiviststoolkit.model.Resources;
 6:
 7:import javax.swing.*;
 8:
 9:/**
24: *
25: * This interface is implemented by rapid data entry plugins which are
26: * loaded into the rapid data entry screen drop-down menu in resource
27: * editor
28: *
29: * Created by IntelliJ IDEA.
30: *
31: * @author: Nathan Stevens
32: * Date: Jul 13, 2010
33: * Time: 2:35:06 PM
34: */
35:public interface RDEPlugin {
36:    /**
37:     * Method to set the domain model. This is always called when the plugin
38:     * are loaded
39:     *
40:     * @param parentRecord The parent resource record
41:     * @param resourcesCommon This can either be a resource component or
42:     * the parent resource record
43:     */
44:    public void setModel(Resources parentRecord, ResourcesCommon
45:            resourcesCommon);
46:
47:    /**
48:     * Method to specify whether this plugin launches a dialog, or
49:     * just executes business logic on the currently selected
50:     * resource component
51:     *
52:     * @return boolean to specify wheather this plugin has a dialog
53:     */
54:    public boolean hasDialog();
55:
56:    /**
57:     * Method to run program logic. This method is called when the hasDialog
58:     * method returns false.
59:     */
60:    public void doTask();
61:
62:    /**
63:     * Method to display a dialog to user. It is called when a call
64:     * to the hasDialog method returns true
65:     *
66:     * @param dialog The parent dialog
67:     * @param title  The title of dialog
68:     */
69:    public void showPlugin(JDialog dialog, String title);
70:}
```

ATPluginUtils Source Code

```
 1:package org.archiviststoolkit.plugin;
 2:
 3:import org.archiviststoolkit.model.ATPluginData;
 4:import org.archiviststoolkit.model.validators.ATValidator;
 5:import org.archiviststoolkit.model.validators.ValidatorFactory;
 6:import org.archiviststoolkit.mydomain.DomainAccessObject;
 7:import org.archiviststoolkit.mydomain.DomainAccessObjectFactory;
 8:import org.archiviststoolkit.mydomain.DomainObject;
 9:import org.archiviststoolkit.util.JGoodiesValidationUtils;
10:import com.thoughtworks.xstream.XStream;
11:import com.jgoodies.validation.ValidationResult;
12:
13:import java.util.Collection;
14:import java.awt.*;
15:
16:/**
32: * This is a utility class to make it easier for plugin developers to save
33: * data to the AT database and perform validation on AT records.
34: *
35: * @author: Nathan Stevens
36: * Date: Feb 11, 2009
37: * Time: 8:07:58 PM
38: */
39:public class ATPluginUtils {
40:    /**
41:     * Method to save text data to the database
42:     *
43:     * @param pluginName The name of the plugin
44:     * @param dataVersion The dataVersion
45:     * @param dataName The name of the data
46:     * @param dataType The type of data
47:     * @param dataString The text data to save
48:     * @throws Exception is thrown if there was a problem saving the data
49:     */
50:    public static void saveData(String pluginName, int dataVersion,
51:                                String dataName, String dataType,
52:                                String dataString) throws Exception {
53:        ATPluginData pluginData =
54:                new ATPluginData(pluginName, false, dataVersion,
55:                        dataName, dataType, dataString);
56:        saveToDatabase(pluginData);
57:    }
58:
59:    /**
60:     * Method that first converts a java object to an xml string
61:     * then save it to the database
62:     *
63:     * @param pluginName The name of the plugin
64:     * @param dataVersion The dataVersion
65:     * @param dataName The name of the data
66:     * @param dataType The type of data
67:     * @param dataObject The object that contains the data or is the data
68:     * @throws Exception is thrown if there was a problem saving the data
69:     */
70:    public static void saveData(String pluginName, int dataVersion,
71:                                String dataName, String dataType,
72:                                Object dataObject) throws Exception {
73:        // use Xstream to convert the java object to an xml string
74:        XStream xstream = new XStream();
75:        String dataString = xstream.toXML(dataObject);
76:
77:        ATPluginData pluginData =
78:                new ATPluginData(pluginName, true, dataVersion,
79:                        dataName, dataType, dataString);
80:        saveToDatabase(pluginData);
81:    }
82:
83:    /**
84:     * Saves plugin data object to the database
85:     *
86:     * @param pluginData The plugin data object
87:     * @throws Exception If there is any problems saving to the database
```

```
 88:     */
 89:    public static void saveToDatabase(ATPluginData pluginData) throws Exception {
 90:        try {
 91:            DomainAccessObject access =
 92:
DomainAccessObjectFactory.getInstance().getDomainAccessObject(ATPluginData.class);
 93:            access.getLongSession();
 94:            access.updateLongSession(pluginData);
 95:        } catch(Exception e) {
 96:            e.printStackTrace();
 97:            throw new Exception("Error Saving Plugin Data to Database ...");
 98:        }
 99:    }
100:
101:    /**
102:     * Method to delete plugin data in the database
103:     *
104:     * @param pluginData The plugin data
105:     * @throws Exception if there was a problem deleting that data
106:     */
107:    public static void deletePluginData(ATPluginData pluginData) throws Exception {
108:        try {
109:            DomainAccessObject access =
110:
DomainAccessObjectFactory.getInstance().getDomainAccessObject(ATPluginData.class);
111:            access.getLongSession();
112:            access.deleteLongSession(pluginData);
113:        } catch(Exception e) {
114:            e.printStackTrace();
115:            throw new Exception("Error Deleting Plugin Data to Database ...");
116:        }
117:    }
118:
119:    /**
120:     * Method to get all the saved data for a certain plugin
121:     *
122:     * @param pluginName The name of the plugin
123:     * @return Collection containing any data they found
124:     * @throws Exception
125:     */
126:    public static Collection getData(String pluginName) throws Exception {
127:        try {
128:            DomainAccessObject access =
129:
DomainAccessObjectFactory.getInstance().getDomainAccessObject(ATPluginData.class);
130:            return access.findByPropertyValue("pluginName", pluginName);
131:        } catch(Exception e) {
132:            e.printStackTrace();
133:            throw new Exception("Error Getting Plugin Data from Database ...");
134:        }
135:
136:    }
137:
138:    /**
139:     * Method to get all the saved data for a particular plugin and data type
140:     *
141:     * @param pluginName The name of the plugin
142:     * @param dataType The data type of the plugin
143:     * @return A collection containing any data that was found
144:     * @throws Exception is thrown of there is a problem find the data
145:     */
146:    public static Collection getData(String pluginName, String dataType) throws
Exception {
147:        try {
148:            DomainAccessObject access =
149:
DomainAccessObjectFactory.getInstance().getDomainAccessObject(ATPluginData.class);
150:            ATPluginData pluginData = new ATPluginData();
151:            pluginData.setPluginName(pluginName);
152:            pluginData.setDataType(dataType);
153:            return access.findByExample(pluginData);
154:        } catch(Exception e) {
155:            e.printStackTrace();
156:            throw new Exception("Error Getting Plugin Data from Database ...");
157:        }
```

```
158:     }
159:
160:     /**
161:      * Method to return a string object or xml encoded object
162:      * found in the database. If more than one data object with
163:      * the same name is found then the first one is return.
164:      *
165:      * @param pluginName The name of the plugin
166:      * @param dataName The name of the data
167:      * @return The data object
168:      * @throws Exception If there is a problem finding the data from the database
169:      */
170:     public static Object getDataByName(String pluginName, String dataName) throws
Exception {
171:         try {
172:             DomainAccessObject access =
173:
DomainAccessObjectFactory.getInstance().getDomainAccessObject(ATPluginData.class);
174:             ATPluginData pluginData = new ATPluginData();
175:             pluginData.setPluginName(pluginName);
176:             pluginData.setDataName(dataName);
177:             Collection collection = access.findByExample(pluginData);
178:
179:             // get the plugin data object returned from the database only return the
first one
180:             if(collection != null) {
181:                 Object[] dataFound = collection.toArray();
182:                 pluginData = (ATPluginData)dataFound[0];
183:                 if(pluginData.getIsObject()) { // xml encoded object so convert it to
an object
184:                     return getObjectFromPluginData(pluginData);
185:                 } else { // just return the plain data string
186:                     return pluginData.getDataString();
187:                 }
188:             } else {
189:                 return null;
190:             }
191:         } catch(Exception e) {
192:             e.printStackTrace();
193:             throw new Exception("Error Getting Plugin Data from Database ...");
194:         }
195:     }
196:
197:     /**
198:      * Method to return an object from plugin data using
199:      * xstream to convert the saved xml to an object
200:      *
201:      * @param pluginData The ATPluginDataContaining the xml encoded object
202:      * @return The converted object or null if conversion can't be done
203:      */
204:     public static Object getObjectFromPluginData(ATPluginData pluginData) {
205:         if(pluginData != null && pluginData.getIsObject()) {
206:             XStream xstream = new XStream();
207:             return xstream.fromXML(pluginData.getDataString());
208:         } else {
209:             return null;
210:         }
211:     }
212:
213:     /**
214:      * Method to save an AT record to the database.
215:      *
216:      * @param record The AT record to save to the database.
217:      * @throws Exception if there is a problem saving the record to the database
218:      */
219:     public static void saveRecordToDatabase(DomainObject record) throws Exception {
220:         try {
221:             Class clazz = record.getClass();
222:
223:             DomainAccessObject access =
224:
DomainAccessObjectFactory.getInstance().getDomainAccessObject(clazz);
225:             access.getLongSession();
226:             access.updateLongSession(record);
227:         } catch(Exception e) {
```

```
228:             e.printStackTrace();
229:             throw new Exception("Error Saving Record to Database ...");
230:         }
231:     }
232:
233:     /**
234:      * Method to valid a AT record. Calling this ensures that no invalid
235:      * records are saved to the database
236:      *
237:      * @param component UI component that is requesting validation of the record
238:      * @param record The AT record to validate
239:      * @return true if the record valide, false otherwise
240:      */
241:     public static boolean validateRecord(Component component, DomainObject record) {
242:         ATValidator validator = ValidatorFactory.getInstance().getValidator(record);
243:     if (validator == null) {
244:       //nothing registered so just return true
245:       return true;
246:     } else {
247:       ValidationResult validationResult = validator.validate();
248:       if (validationResult.hasErrors()) {
249:         JGoodiesValidationUtils.showValidationMessage(
250:             component,
251:             "To save the record, please fix the following errors:",
252:             validationResult);
253:         return false;
254:       }
255:       if (validationResult.hasWarnings()) {
256:         JGoodiesValidationUtils.showValidationMessage(
257:             component,
258:             "Note: some fields are invalid.",
259:             validationResult);
260:       }
261:       return true;
262:     }
263:     }
264:}
```