Plugin Programmer's Guide

UNICOREpro Client Version 1.0
23 July 2003
Revision 2

Ralf Ratering
Pallas GmbH
Hermuehlheimer Str. 10
D-50321 Bruehl, Germany
ralf.ratering@pallas.com

1 Overview

The plugin interface offers the possibility to add functionality to the standard UNICOREpro Client without manipulating the basic software. This paper describes how different types of plugins can be implemented and how existing Client components can be reused. You will find the complete Client source code available for download at the download section of http://www.unicorepro.com. The source code package also contains the Script, Command, POV-Ray and Auto Update Plugins, that you may want to use as an example or template.
The paper is not meant as a general introduction into UNICORE. If you are not familiar with UNICORE concept and terms please refer to other documentation available at the UNICOREpro web sites.

2 General Client Concepts

The Client software contains several components that can be re-used for plugin programming. These components include graphical interfaces (FileImportPanel, FileExportPanel, RemoteTextEditor), container classes to create your own Abstract Job Objects (ActionContainer), as well as classes to get the outcome of your Abstract Job Object (OutcomeManager) and to access the available resources (ResourceManager). For those of you who want to write their own customized Abstract Job Objects and want to submit them directly from the plugin we have added a Requests section that describes how to use the Observable mechanism for job submission.

2.1 Resource Manager

The com.pallas.unicore.resourcemanager.ResourceManager class is the central instance where all resources like images, Usite and Vsite resources, file chooser dialogs and user information are stored. There are several public static methods to access these resources and the most important ones will be described here:

`public static Client getCurrentInstance();` Get a reference to the current Client instance.
`public static UserDefaults getUserDefaults();` Get a reference to the user defaults.
`public static SystemDefaults getSystemDefaults();` Get the SystemDefaults instance, containing e.g. the default Usite URL.

`public static synchronized String getNextObjectIdentifier();` Get a unique identifier String e.g. for your AJO objects.
`public static ConnectionManager getConnectionManager();` Get a reference to the Client's ConnectionManager
`public static SelectorDialog getSelectorDialog(File selectedFile, Vsite vsite);` Get a file chooser that already exists for a VSite. This method is much faster than building a new one, because the file chooser has already been built and will just be set to visible. If the VSite parameter is null a local file chooser will be returned. The first parameter sets the initially selected file.
`public static SimpleDateFormat getDateFormatter();` Get a DateFormatter with format "MM/dd/yyyy". Use this method to get a format consistent with the rest of the Client GUI.
`public static SimpleDateFormat getTimeFormatter();` Get a DateFormatter with format "HH:mm:ss". Use this method to get a format consistent with the rest of the Client GUI.
`public static SimpleDateFormat getCompleteFormatter();` Get a DateFormatter with format "HH:mm:ss MM/dd/yyyy". Use this method to get a format consistent with the rest of the Client GUI.
`public static SimpleDateFormat getUTCFormatter();` Get a DateFormatter translating local time to UTC. Use this method to get a format consistent with the rest of the Client GUI.
`public static Vector getUsites();` Get a Vector containing the Usite objects that have been loaded from the Unicore Site URL.
`public static Vector getVsites(Usite usite);` Get a Vector containing all Vsite objects that are sent from a given Usite by the listVsites UPL request.
`public static NamedResourceSet getResourceSet(Vsite vsite);` Get a set of available resource objects at a given Vsite.
`public static Vsite getMatchingVsite(Vsite vsite);` Given a Vsite instance, find another Vsite instance in the resource cache that points to the same real Vsite.
`public static boolean isUsiteAvailable(Usite usite);` Test, if a given Usite is accessible from the Client.
`public static boolean isVsiteAvailable(Vsite vsite);` Test, if a given Vsite is accessible from the Client.
`public static boolean usiteExists(Usite usite);` Test, if a Usite exists in the resource cache.
`public static boolean vsiteExists(Vsite vsite);` Test, if a Vsite exists in the resource cache.
`public static boolean isSpecialVsite(Vsite vsite);` Test, if a given Vsite supports some special features like resource brokerage or connection to Globus sites.
`public static ResourceCache getResourceCache();` Get a reference to the resource cache, if you want to manipulate it directly.
`public static String getServicePrefix();` When sending an internal service job to a Vsite use this prefix to make sure the job will not appear in the Client's Job Monitor.
`public static User getUser(Vsite vsite);` Get the User object containing the certificate to be used for a given Vsite.

2.2 Using ActionContainers to encapsulate AJOs

UNICORE is based on the creation of Abstract Job Objects (AJO) that are sent between Client and server. These AJOs consist of ActionGroups that contain directed acyclic graphs of AbstractActions at a very low level. To handle those rather complex job structures the Client uses different ActionContainers that encapsulate ActionGroups that serve different purposes, like file operations, command execution or internal services.

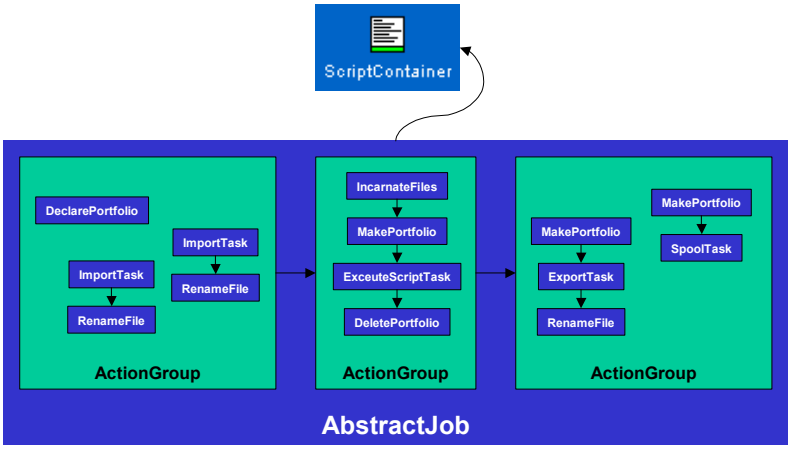


Figure 1: Script Container encapsulating a complex AJO

Although the container classes hide away as much of the AJO programming from the plugin developer as possible, it is still necessary to understand some of the basic AJO concepts to implement complex task plugins. Please refer to the AJO source code and documentation, if you want to create your own customized abstract job objects that are not covered by the container hierarchy.

Most probably, your plugin container will fit into the container hierarchy as a sub class of UserContainer, because these containers execute applications that are accessible via application resources. However, if you are planning to build more complex plugins tasks that consist of sub jobs running at different sites or contain complex control structures, you should consider to make your container a sub class of GroupContainer:

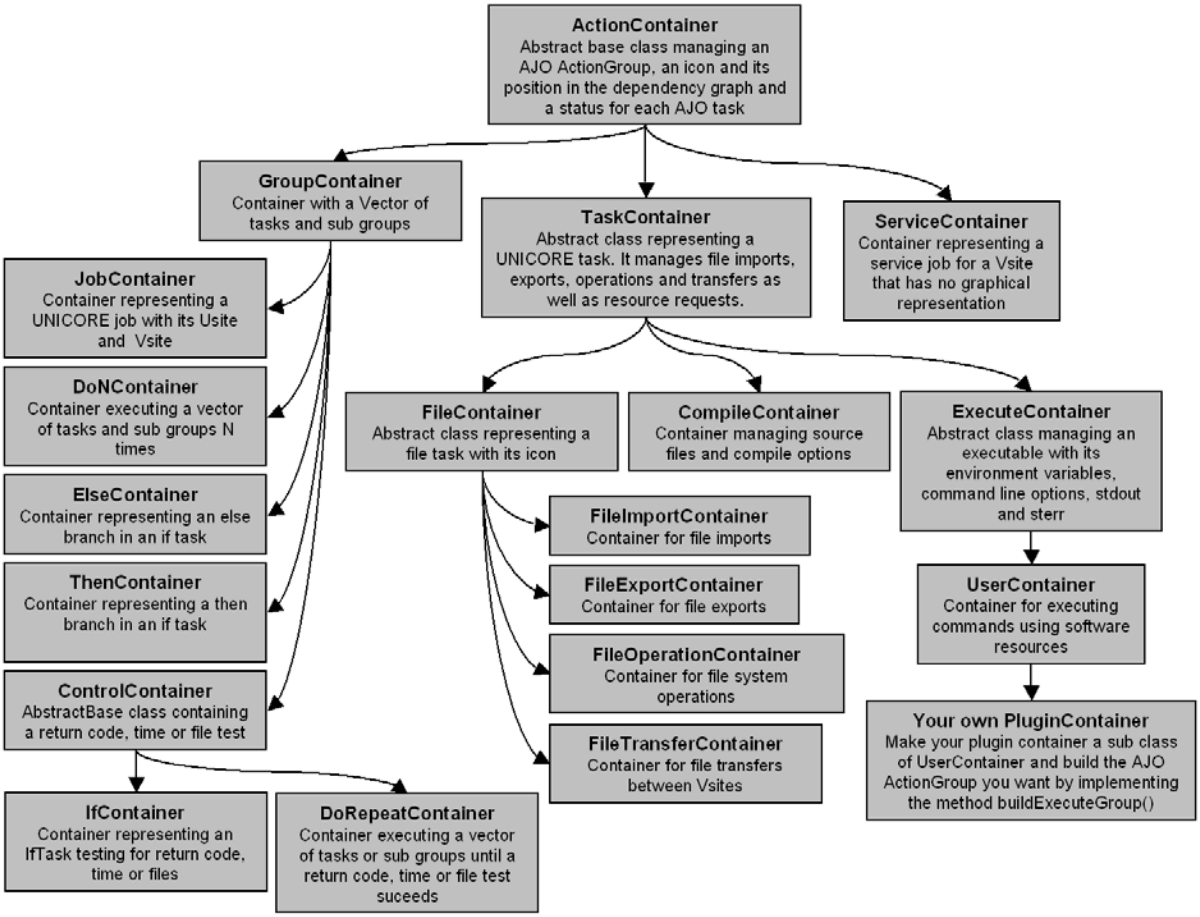


Figure 2: UNICORE Client container hierarchy in package com.pallas.unicore.container

2.3 Accessing Remote File Systems

Within the UNICOREpro Client you have the possibility to access and manipulate remote file systems using the packages com.pallas.unicore.Client.remotefilechooser and com.pallas.unicore.Client.explorer.

RemoteFile

The NJS and TSI at the server side work on org.unicore.ajo.Xfile objects that are wrapped into a sub class of java.io.File called com.pallas.unicore.Client.remotefilechooser.RemoteFile in the Client. A RemoteFile object will be returned from the SelectorDialog or can be generated from scratch. Every RemoteFile is defined by a Storage resource and a path in that Storage space, e.g. Storage "Home" and path "doc/unicore" may be resolved to "/home/username/doc/unicore" at a certain site. But the RemoteFile object will not know about the complete resolved path until the Client has once contacted the Vsite and sent a GetListing request. This is the case, when the RemoteFile object was generated by the Selector Dialog, or when the updatePath() method of RemoteFile was executed.

SelectorDialog

The class com.pallas.unicore.Client.explorer.SelectorDialog contains methods that are similar to those of the standard Swing JFileChooser and should be self-explaining. However, there is one important detail about the way the Client handles SelectorDialog objects: The ResourceManager keeps track of all Selector Dialogs that have been built in a session and stores them in a hashtable. So, when using a SelectorDialog in your plugin you should ask the

ResourceManager for an instance, instead of creating your own one. Use the method ResourceManager.getSelectorDialog(File selectedFile, Vsite vsite) for that.
If you insist on creating your own dialog make sure you to set the pre-emptive mode in the constructor to ResourceManager.getUserDefaults().getPreemptiveFileChooserMode(). This pre-emptive mode will get listings for nodes one level in advance in the file system hierarchy. This is far more convenient for the user, but can lead to performance problems on very large or slow file systems, so the user can choose whether to use this feature or not in the User Defaults.

RemoteTextEditor

The RemoteTextEditor in package com.pallas.unicore.Client.editor allows to load, edit and save files from remote file spaces. It also supports undo and redo as well as cut, copy and paste inside the Client and from other applications. It is a subclass of its local equivalent com.pallas.unicore.Client.editor.TextEditor. Both may be used "out of the box" by adding them to you own Component:

```
public class PluginJPAPanel extends JPAPanel {
  private PluginContainer container;
  private RemoteTextEditor textEditor;

  private buildComponents() {
    textEditor = new RemoteTextEditor();
    JScrollPane editorScrollPane =
      new JScrollPane(textEditor);
  }

  public void applyValues() {
    container.setText(textEditor.getText());
  }

  public void updateValues(boolean vsiteChanged) {
    if(vsiteChanged) {
      textEditor.setVsite(container.getVsite());
    }
  }
}
```

Theses are the most important methods to access the RemoteTextEditor:

`public void setVsite(Vsite vsite);`
`public Vsite getVsite();`
Set/Get Vsite to connect to

`public void setText(String text);`
`public String getText();`
Set/Get text from editor text area

`public String getUnixStyleText();` Get text from text area, but remove any Windows style EOL and EOF characters

`public void setFile(File file);`
`public File getFile();`
Set/Get the file that is used to store the editor contents. Please note, that getFile() returns the File object that represents the currently loaded file. This will be an instance of com.pallas.unicore.Client.remotefilechooser.RemoteFile, in case we loaded a file from a remote file system. You may want to store this File object in your plugin container to remember the file source when reloading the task. Please, take a look at the ScriptContainer for an example.

2.4 Managing the Outcome

The com.pallas.unicore.resourcemanager.OutcomeManager keeps track of the outcome of jobs during a session. When a task is executed in an iteration it will produce one org.unicore.outcomes.Outcome in each iteration. The OutcomeManager collects all outcome data belonging to one iteration of an AbstractAction and wraps it into an OutcomeEntry. Additionally all filenames of files that are streamed from the NJS to temporary session directory at the local file system are stored in the OutcomeManager. If your plugin contained custom file exports to the local file system, that are not handled by the client, for instance if used in an additional outcome panel, you should search in the streamed files of the OutcomeManager to get the file.

OutcomeManager

The OutcomeManager contains only static methods to find the outcome information about given AbstractActions or ActionContainers. All methods return Vectors containing one element for each iteration. In case of nested iterations you will get a Vector containing other Vectors.

`public static Vector getOutcomeEntries();` Return a Vector containing all OutcomeEntries belonging to an AbstractAction
`public static Vector getOutcomes();` Return a Vector containing all OutcomeEntries belonging to an ActionContainer
`public static Vector getDetails();` Return a Vector containing all DetailEntries belonging to an ActionContainer
`public static Vector getStreamedFiles();` Return a Vector containing the filenames of all files that have been streamed into the Client's session directory

OutcomeEntry

In an OutcomeEntry all outcome information for one iteration of an AbstractAction is collected. For AbstractTask objects standard out and standard error are streamed as files with the outcome from the NJS and stored in a session directory at the Client. The files belonging to the AbstractTask are stored in the OutcomeEntry, as well as the Outcome object itself, a time stamp containing the last status change, the NJS log and the iteration number:

`public AbstractAction getAction();` Get the AbstractAction belonging to this entry
`public Outcome getOutcome();` Get the org.unicore.outcome.Outcome object belonging to the AbstractAction
`public String getLog();` Get the NJS log belonging to the AbstractAction
`public File getStdoutFile();` Get the File containing standard out in case we have an AbstractTask
`public File getStderrFile();` Get the File containing standard error in case we have an AbstractTask
`public String getIteration();` Get the iteration number for this entry
`public Date getTimeStamp();` Get the time stamp for the last status change of the AbstractAction

2.5 Requests

In some cases it might not be appropriate for your plugin to use the container hierarchy for job construction, because you need some very specialized functionality that requires direct AJO programming. If so, you should first check if the com.pallas.unicore.requests package does already contain a package that fits your needs and if not write your own request as a sub class of ObservableRequestThread.

Observers and Observables

In the UNICOREpro Client AJOs are submitted as separate threads. Every thread can have one or more observers attached to it that will be notified when the request has returned or any other event occurred. To do so requests are sub classes of ObservableRequestThread which implements the IObserver interface and contains AJO submission functionality. If your plugin wants to get notified about request events, it has to implement the IObserver interface.

Useful Requests

Amongst others there are several requests that you might find useful for your plugin:

`GetFilesFromUspace` Get a list of files from another job's Uspace. This request is helpful if you want to get intermediate results from a running job, for instance to visualize them in an additional outcome panel.
`SendFilesToUspace` Send a list of files to another job's Uspace. With this request you can send steering files from your local workstation into the working directory of a running job.
`GetUsites` Get an updated list of currently available Usites
`GetVsites` Get an updated list of currently available Vsites at a certain Usite
`GetResources` Get the updated set of currently available resources at a certain Vsite
`GetJobStatus` Get the current status of an AbstractJob
`GetListing` Get the listing of a directory at a remote file system

Write your own request

There are a few points to follow if you want to submit a customized AJO in your own request:

1. Make your request a sub class of ObservableRequestThread
2. Implement the run() method
3. In run() construct your AbstractJob object and submit it using the inherited nonPolling() method. CAUTION: nonPolling tries to execute the request without polling, but if a time out occurs it will automatically switch into polling mode.
4. Check the returned Reply object of nonPolling() and extract the deserialized Outcome from it in case it is a RetrieveOutcomeReply. Refer to the GetFilesFromUspace source code for an example.
5. Notify all observers when the request has finished.
6. When using the request execute it with start() and make sure you have an Observer attached to it.

3 Plugins

3.1 Plugin Concept

Loading Plugins

Plugins are separate modules that are loaded into the main software at runtime. These modules come as signed Jar archives that have the ending **"Plugin.jar"**. The Client scans two directories for files that match this ending:

1. The installation directory ($INSTALLPATH/lib): In this directory all plugins that are part of the distribution (currently script, command and autoUpdate) are stored. It is also possible for the system administrator to place plugins here that should be available system wide.
2. The user defined directory: This directory is definable via the user defaults dialog in the Client and contains plugins that are only available to a single user.

When the Client finds a "*Plugin.jar" in one of these directories, it will check if the plugin signature matches a trusted entry in the Client keystore. If so, it scans the archive contents for a class file ending with **"Plugin.class"**. This class is assumed to be the main plugin class and will be registered by the Client's plugin manager.

As a package name for your plugin you may choose any name you want except for those starting with "com.pallas.unicore", because for security reasons plugin classes are not allowed to reside in this name space.

How to Build a Plugin Jar Archive

1. Write and compile your plugin classes in a package called, lets say "com.yourinstitution.unicore.Client.plugins.yourplugin"
2. Go to the parent directory of your package path and build the Jar-archive with "jar cvf yourPlugin.jar com/yourinstitution/unicore/Client/plugins/yourplugin"
3. Sign the archive with your User Certificate (or any other certificate that you think is appropriate). If you do not know where to find your certificate, simply use the one from the Client keystore ($HOME/.unicorepro/keystore): "jarsigner -keystore keystore -storetype JCEKS yourPlugin.jar aliasOfYourCertificate"
4. Put the signed plugin archive into the Client's plugin search path (see above).

GUI Design Issues

If you are planning to extend the Client GUI in some way, we recommend to follow the Java Look and Feel Design Guidelines available at
http://java.sun.com/products/jlf/ed1/dg/index.htm.

Plugin Types

We distinguish two different types of plugins, **Task Plugins** that add new types of tasks to the standard Client tasks (e.g. CPMD or Fluent) and **Extension Plugins** that add any other new functionality to the Client (e.g. Interactive Access or Resource Broker). Please note, that the standard Client distribution contains two task plugins (Script, Command) and one extension plugin (Auto Update), so these should be a good reference on how to implement the different types. Both types have in common that they have to implement the main plugin class.

3.2 The Main Plugin Class

The main plugin class is responsible for loading, starting and finalizing the plugin. The abstract base class is com.pallas.Client.util.UnicorePlugable and you should extend one of its two sub classes TaskPlugable or ExtensionPlugable.

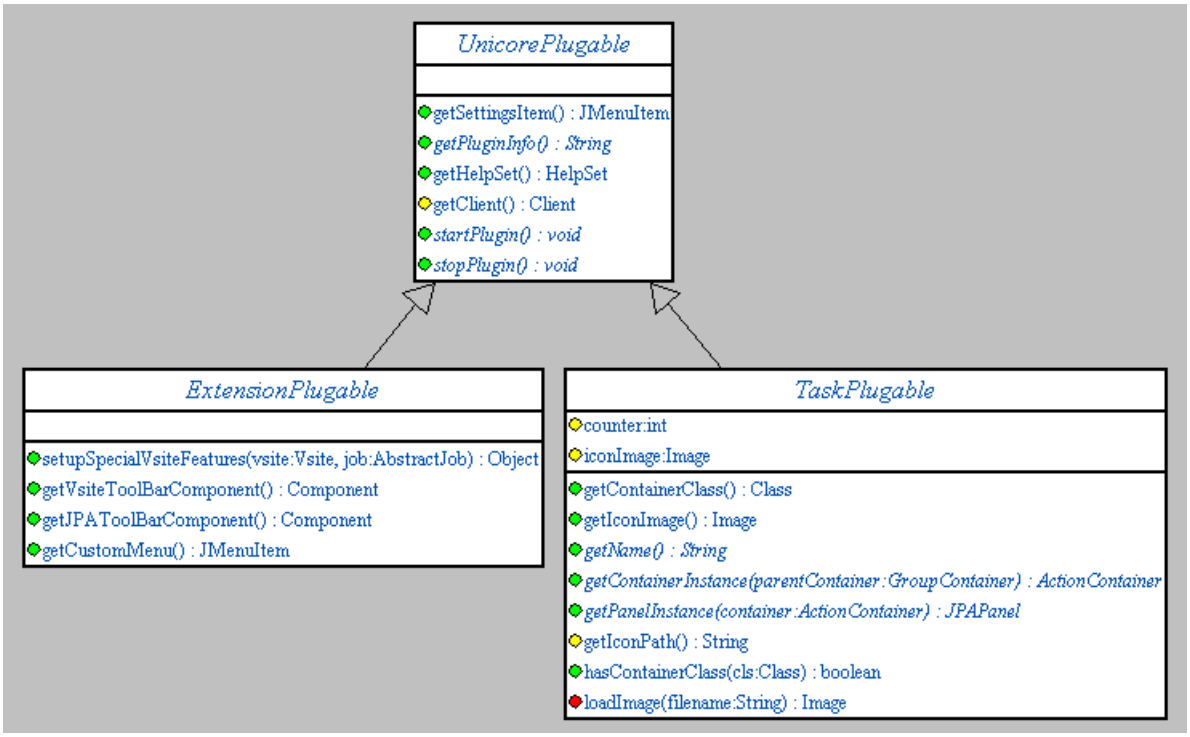


Figure 3:UML Diagram for main plugin classes in package com.pallas.unicore.Client.util

Abstract Methods

The following abstract methods have to be implemented by every plugin:

`public abstract void startPlugin();` This method will be called after the plugin has been loaded. Perform all initialisation like building or reading defaults here.
`public abstract void stopPlugin();` This method will be called before the Client exists. Here any kind of finalization like stopping threads or removing listeners can be performed
`public abstract String getPluginInfo();` Return an info string about the plugin, containing some information about version, author, copyright, etc. This string will be displayed in the Client's about dialog.

.

Public and Protected Methods

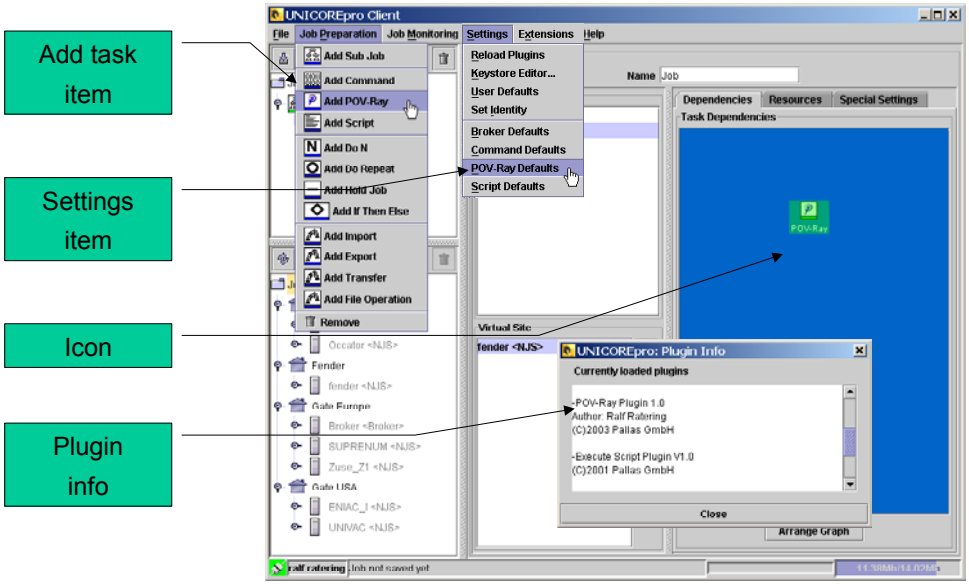
The following public or protected methods may be overwritten or used by your plugin:

`public JMenuItem getSettingsItem();` Return a menu item that will be added to the Client's settings menu. Typically these items are used to popup a plugin specific configuration dialog. The Client does not attach a listener to this item so it is up to the plugin developer to respond to this item. The method returns null by default, meaning that no menu item will be added to the Clients settings menu.
`public HelpSet getHelpSet();` Return a help set that will be plugged into the Client's Java help. The method returns null by default meaning that no help set is available. Please refer to the JavaHelp documentation on how to create your own help set [JavaHelp].

```
protected Client getClient();
```
This method can be used by a plugin to get a reference to the current Client instance.

4 Task Plugins

Task Plugins extend the standard UNICOREpro Client by adding new types of tasks to a job via the Job Preparation menu. Every Task Plugin should implement 3 classes: a main plugin class that is a sub class of TaskPlugable, a sub class of JPAPanel and a sub class of UserContainer.



Gui Components added by a task plugin

Figure 4: GUI components added by a task plugin

4.1 TaskPlugable

The TaskPlugable class will provide methods that integrate the UserContainer into the Client jobs and the JPAPanel (the graphical representation of the container) into the Client GUI.

Abstract Methods

```
public abstract String getName();
```
Get name for plugin task that will be presented to the user in the GUI, e.g. in the job preparation menu.

```
public abstract ActionContainer getContainerInstance(GroupContainer
parentContainer);
```
Build a new instance of the plugin specific ActionContainer. As argument the GroupContainer that is parent of the plugin task will be passed. When implementing this method we recommend to initialise the container name using the following code snippet:

```
YourContainer container = new YourContainer(parentContainer);
container.setName("New_" + getName() + counter);
container.setIcon(new ImageIcon(getIconImage()));
// count instance
counter++;
```

Your plugin task instances will then be counted using the protected integer field `counter`

inherited from class TaskPlugin. Do not use your own counter variable, because the method getContainerInstance will also be used to determine the container class without adding a new plugin task to a job.

Public and Protected Methods

The following public and protected methods may be used or overwritten in your task plugin:

`protected String getIconPath();`
Every task plugin has to provide one single **icon** that represents the task in the job preparation and job monitor trees as well as in the dependency graph. The TaskPlugable class provides methods to load an Image from file and pass it to the Client GUI. We recommend to include an icon into your plugin jar archive and return the path to the icon by overwriting this method.
(e.g. return "com/yourinstitution/unicore/Client/plugins/yourplugin/yourIcon.gif").
The icon should have size 20x21 pixel with a bar of 5 pixel at the bottom and an image that represents your plugin application above. It should be stored in gif format.

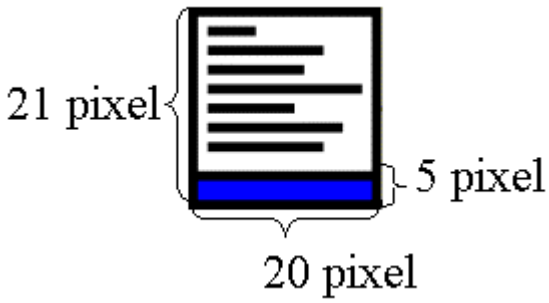


Figure 5: Format for container icon

The Client will automatically fill the bar at the bottom of the image with the correct color corresponding to the current task status.

`public Image getIconImage();`
This method has been used to access the icon image in earlier versions but has been deprecated. Use getIconPath() instead.

`public final Class getContainerClass();`
Final method returning the class of your plugin container. For internal use only.

`public final boolean hasContainerClass(Class cls);`
Final method returning true, if your plugin will provide a container of a certain class. For internal use only.

4.2 Implementing a JPAPanel

The JPAPanel is the graphical representation of the plugin container in the user interface. Here all plugin parameters, file imports and exports are set by the user. Whenever the plugin container node in the job preparation tree will be selected the JPAPanel becomes visible. To implement your own JPAPanel you have to extend a sub class of JPanel com.pallas.unicore.Client.panels.JPAPanel. This class contains three abstract methods that have to be implemented.

Abstract Methods

`public abstract void resetValues();`
Fill the controls in the JPAPanel with the entries from the underlying container. This method will be called, whenever a new JPAPanel was built, e.g. when loading a job or

when copying and pasting.
`public abstract void applyValues();` Apply the current entries in the GUI to the underlying container. This method will be executed before checking the job for correctness. The job will be checked for correctness before it will be submitted, when clicking on the check button or simply when clicking onto the job in the job preparation tree. You may also want to call this method in your plugin to make sure the underlying container is up to date at a certain point of the execution.
`public abstract void updateValues(boolean vsiteChanged);` This method that will be called whenever your plugin task has been selected in the job preparation tree and your JPAPanel becomes visible in the GUI. If the parameter vsiteChanged is true, the Vsite has changed since last the last update, so you may for instance want to refresh some resources that are displayed in your JPAPanel.

Adding import, export and option panels to your JPAPanel

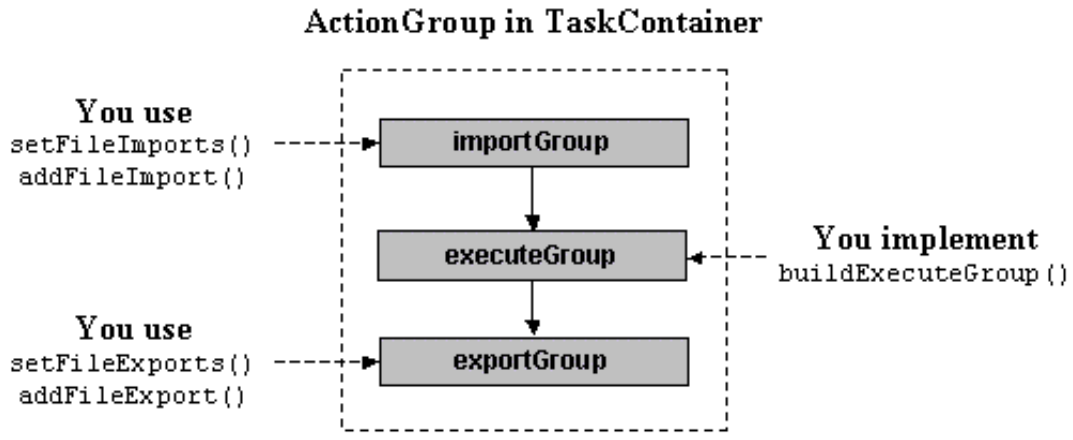
Import and export panels are the graphical interfaces to define imports and exports of files within the UNICOREpro Client, while the option panel allows the user to set execution parameters. You can integrate these panels simply by adding instances of com.pallas.unicore.Client.panels.ImportPanel, ExportPanel and OptionPanel to your JPAPanel. Make sure that you pass the applyValues, resetValues and updateValues events to the panels by calling the corresponding methods in the panels. Refer to the Script Plugin for an example.

4.3 Implementing a Plugin Container

You should make your plugin container a sub class of UserContainer, because these containers can execute binaries as well as Application Resources that are defined in the Incarnation Database of the Vsite. The UserContainer also handles imports and exports by itself, if you have integrated the import and export panels into your JPAPanel.

Imports and exports and execution

Imports and exports are completely handled by the super class of UserContainer TaskContainer. In a TaskContainer an AJO ActionGroup is constructed that consists of an import, an execution and an export ActionGroup. This reflects what most plugin tasks will



look like: They import some files from the local or remote workstation into the USpace,

execute an application and transfer the result files back to the workstation. So for this simple case, all you have to care about in your plugin container is the construction of the execution ActionGroup. The following code snippet shows how to execute an ApplicationResource that is defined at the Vsite using a org.unicore.ajo.UserTask:

```
public class YourPluginContainer extends UserContainer {
  …
  /**
   * This is a simple example for an execution group that
   * executes a software resource using a UserTask.
   **/
  protected void buildExecuteGroup() {
    …
    // CAUTION !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // The container's resource set is just a reference
    // to a resource set object that may be used by other
    // tasks!!
    // So ****CLONE**** the resource set before you add
    // your software resource!!
    ResourceSet taskResourceSet = getResourceSet().getResourceSetClone();

    // CAUTION: Do not forget to set the software resource using
    // setPreinstalledSoftware(Resource softwareResource) before.
    taskResourceSet.add(getPreinstalledSoftware());

    UserTask userTask =
        new UserTask(getName(),
            null,
            taskResourceSet, // ResourceSet contains ApplicationResource
            env,
            getCommandLine(),
            null,
            getRedirectStdout(),
            getRedirectStderr(),
            isVerboseOn(),
            isVersionOn(),
            null, // this should be a Portfolio containing an executable
                  // if no ApplicationResource will be executed
            getMeasureTime(),
            getDebug(),
            getProfile());

    executeGroup =
        new ActionGroup("PluginExecutionGroup");
    executeGroup.add(userTask);
  }
}
```

There are two important points about this snippet:

1. The Client attaches a com.pallas.extensions.NamedResourceSet to each TaskContainer that can be accessed by the method getResourceSet(). Make sure to clone the resource set, before you add your software resource to it. This has been one of the major changes in Client version 3.6: One ResourceSet may be referenced by multiple different tasks. So remember, that before manipulating the ResourceSet that is attached to your task you have to clone it, because other tasks are affected to any changes to the original ResourceSet.
2. Add your Application Resource to the ResourceSet that will be passed to the UserTask. Please refer to the AJO documentation for more details about the UserTask or take a look at the buildExecuteGroup() method in UserContainer.

Checking for errors in the container

Every ActionContainer has a com.pallas.unicore.container.errorspec.ErrorSet that contains information about wrong or missing parameters. This ErrorSet is generated by calls to the

checkContents() routine. In your own checkContents() method you should call the method from the super class and only check the parameters that are specific to your plugin container:

```
public ErrorSet checkContents() {
    // collect errors from super classes
    ErrorSet err = super.checkContents();

    // do your own checking here…
    if(… one of your parameters is wrong …) {
        // add a new Urror to the ErrorSet with the Container identifier
        // and an error message as parameters
        err.add(new UError(getIdentifier(), “Hey, my parameter is wrong!”));
    }
    // update the error set field
    setErrors(err);
    // return the error set
    return err;
}
```

The error message given as a constructor parameter to UError will be displayed in the Client when the Check button is pressed in the job preparation tree or when the user tries to submit an incorrect job.

Using Application Resources

Keeping a job seamless means the job must not contain site-specific information to ensure it can be executed at any site without modifications. The challenge for seamless application support was to handle different installations and versions of the same software at different sites. The solution to this problem was to use *application resources* (org.unciore.resources.ApplicationResource) that are defined in the Incarnation Database (IDB) and are part of the resource set that is sent from the server to the Client. Application resources tell the Client which applications are supported at that site. If the application required by the plugin is not within this resource set the user will be prompted that the application is not available at that site.
If a matching site was found, its application resource will be attached to the job that is then submitted to the site. The Target System Interface (TSI) will then know that the job requires a special application and will look in its database on how to incarnate the plugin task. Please note, that an application resource is identified by its name and version, so it can be sent to different sites without modifications.

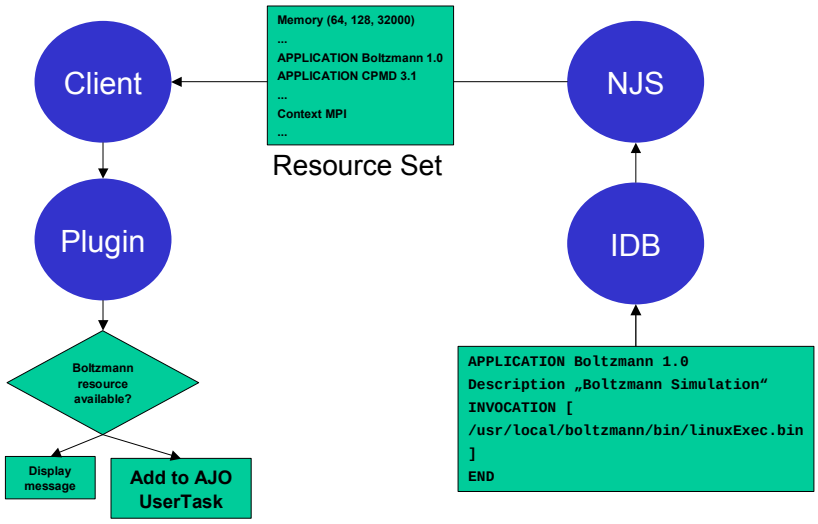


Figure 6: Using application resources in a Plugin User Task

Please, refer to the Incarnation Database documentation to see in detail how application resources are defined in the IDB. This is an example incarnation for the CPMD application:

```
SOFTWARE_RESOURCE APPLICATION CPMD 3.1.4
INVOCATION CPMD-3.1.4 [
  export PP_LIBRARY_PATH=/usr/local/cpmd/lib/PP_LIBRARY;
  /bin/mpprun -n $UC_NODES
  /usr/local/cpmd/bin/cpmd3.4.1.x $CPMD_FILE $PP_LIBRARY;
]
```

A UserContainer contains a field softwareResource that should contain the resource that corresponds to the plugin's application. The field can be accessed via the methods set/getPreinstalledSoftware().
Please note, that plugins need to identify their application resources by the names given in the IDB. As a plugin writer you should specify the application resource name, so that all system administrators that want to support the plugin know, which name they should give the application resource.

Adding additional imports or exports to the plugin task

The easiest way to add your own imports and exports to a plugin task is to overwrite the method buildActionGroup() in the container class:

```
public void buildActionGroup() {
  FileExport export =
    new FileExport(this, FileStorage.NSPACE_STRING, "testfile",
                   "c:/tmp/testfile", true, true);
    addFileExport(export);
    super.buildActionGroup();
}
```

Do not use the setExport() method here, because this one will overwrite all additional exports that have been set by the user. Also note, that the additional exports will not appear in the export panel. If you want the exports to appear in the panel you should write a method in the plugin's JPAPanel class, but take care that the same export is not added multiple times:

```
private void buildAdditionalExports() {
  FileExport export =
    new FileExport(container, FileStorage.NSPACE_STRING, "testfile",
                   "c:/tmp/testfile", true, true)
  container.addFileExport(export);
  exportPanel.resetValues();
}
```

Storing plugin tasks

When the Client writes a job to file the user can choose between binary and XML format. Both mechanisms do essentially the same, except that XML files are human readable and editable. As a plugin writer you should be aware that the plugin container instance will be written to file as a part of the job. An important issue is compatibility between versions here, because if you add or change fields in your container a new version may become incompatible to an older one.
Caution: Do not **change** the type of fields in your container classes, because this will lead to incompatibilities between versions. There are ways to recover old versions in both storing mechanisms, but whenever possible just do not change field types.

- Using binary format

When the Client stores jobs in binary format it uses the standard Java serialization. All fields of the plugin container that should not be stored to file must be marked as transient. This is an important issue, because you may have declared GUI elements as fields in your container, that you do not want to write to file. All non-transient fields have to implement the Serializable interface.
Caution: Make sure to add a long field serialVersionUID to your container. The Java serialization uses these values to identify classes, even if their fields have changed. You can generate these identifiers with the SDK-tool “serialver”:

```
C:\src>serialver –classpath <yourUnicoreClasspath>
com.yourinstitution.unicore.Client.plugins.YourPluginContainer.class
```

When you had to make significant changes to your container class, so that the versions have become incompatible you can overwrite the methods readObject() and writeObject() in the container class. Please, refer to the Java SDK Documentation about further details.

- Using XML

Since version 3.6 the Client will read and write jobs in XML format. The mechanism used here is quite similar to the standard Java serialization. For plugin developers there are some rules to follow:

1. By default the information stored in XML is exactly the same as it is in the standard serialization. That is, no static or transient fields are stored. This should be ok for most plugins.
2. If you do not want this, define a two-dimensional String array persistentXMLFields. This array should contain all fields you want to be stored in the XML file.

When your container versions have become incompatible, it is possible to convert old versions using the XSLT mechanism to transforms old XML versions into new ones.

Executing plugin tasks in loops

The control tasks in the UNICOREpro client include DoN and DoRepeat groups that can execute tasks in iterations. When your plugin ActionGroup will get executed multiple times all portfolios that have been generated by MakePortfolio or DeclarePortfolio in one iteration have to be deleted with a DeletePortfolio task at the end of the action graph. If not, the NJS will complain about an existing portfolio that can not be overwritten in the next iteration.

Testing plugin tasks for return codes

Every sub class of TaskContainer can get tested for return code in an If task or a DoRepeat group. The TaskContainer contains a method getExecuteTask() that will return the first ExecuteTask that occurs in the ActionGroup of the TaskContainer. This should be fine for most plugins, because they will probably contain only one ExecuteTask. If your plugin container builds multiple ExecuteTasks, you should overwrite the method getExecuteTask(), to return that task that you want to be tested for return code.

4.4 Adding plugin panels to the outcome area

In the UNICOREpro Client plugins not only can add their own panels to the job preparation area, but it is also possible to add one or more panels to the job outcome area in the job monitor.
These additional panels will be added to the tabbed outcome pane and can then display some plugin specific output, like graphics or tables. It is so possible to directly integrate visualization components for the plugin task into the Client instead of using external viewers for the application output.

IPanelProvider

To add outcome panels to your plugin the plugin container has to implement the interface com.pallas.unicore.Client.panels.IPanelProvider. The interface consists of the following methods:

`public int getNrOfPanels();`
Return the number of additional outcome panels provided by this plugin task

`public JPanel getPanel(int i);`
Return the ith panel. **Caution**: Be aware that your plugin container will be serialized and streamed to the NJS. To avoid unnecessary data transfer you should make your outcome panel a transient field in the container. In the getPanel() method you may check if the outcome panel is still null and build it there:

```
private transient YourOutcomePanel outcomePanel;
public JPanel getPanel(int i) {
      if(outcomePanel == null) {
            outcomePanel = new YourOutcomePanel();
      }
      return outcomePanel;
}
```

Do not build the outcome panel in the constructor or each time getPanel() will be called.

`public String getPanelTitle(int i);`
Return title for ith panel. This title will be displayed at the tabbed pane in the Client's outcome panel.

`public void finalizePanel();`
This method will be called when the job is removed from the Vsite. There may be some finalization necessary (e.g. stopping threads) that can be performed here.

Getting notifications from the Client

If your additional outcome panel implements the interface com.pallas.unicore.client.panels.Applyable you will get notified if your panel becomes visible or new outcome for your task arrives:

`public void updateValues();` This method will be executed whenever the additional outcome panel becomes visible, which is on the tabbed pane event.
`public void resetValues();` This method will be executed whenever a new outcome for the job arrives.

5 Extension Plugins

Extension plugins can extend the Client functionality in arbitrary ways. There are several methods to add new components to the Client GUI. You have to implement only the main plugin class that should be a sub class of ExtensionPlugable.

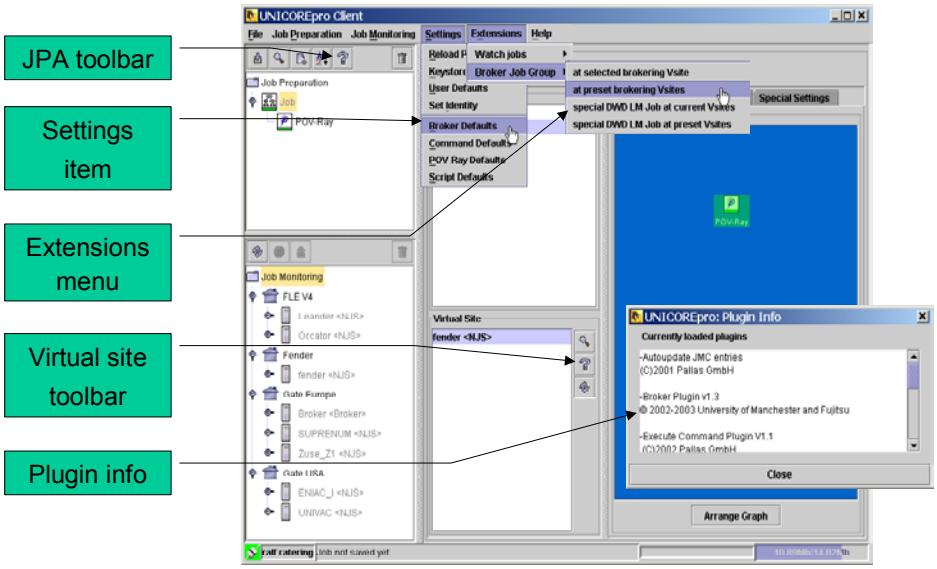


Figure 7: GUI components added by an Extension Plugin

5.1 ExtensionPlugable

This class does not contain any abstract methods, so you may just choose to implement the abstract methods of the base class UnicorePlugable and implement any other functionality you like. But if you want to add your own controls to the Client GUI you should overwrite one or more of the public methods implemented in ExtensionPlugable.

Public methods

`public Component getVsiteToolBarComponent();` Overwrite this method to return a Component that will be added to the toolbar that is attached to the virtual site list in a job group panel. Ideally, the component should contain a button displaying an icon following the Java Look and Feel Design Guidelines. By default the method returns null, meaning that no component will be added.

`public Component getJPAToolBarComponent();` Overwrite this method to return a Component that will be added to the toolbar that is attached to the job preparation tree. Ideally, the component should contain a button displaying an icon following the Java Look and Feel Design Guidelines. By default the method returns null, meaning that no component will be added.
`public JMenuItem getCustomMenu();` Overwrite this method to add a plugin controlled item or sub menu to the Client's extension menu. The Client does not attach a listener to this item, so it is up to the plugin developer to respond to those item events. By default the method returns null, meaning that no menu item will be added.
`public Object setupSpecialVsiteFeatures(Vsite vsite, AbstractJob job)` This method checks if the plugin can enable special features for a Vsite. It may for example create a Globus proxy certifcate for a Globus enabled Vsite.

6 Further Information

We hope that you find all the functionality needed for your plugin in the interface described here. However, if you are missing something, because you are planning to implement something we have not thought of yet, please register and post your request at our issue tracking system at http://roundup.pallas.com/grid-issues. Also all bug reports, comments and suggestions should go there.