FlowJo Plugin Developer's Guide

Introduction

This developer's guide describes how to extend the capabilities of the FlowJo application using the FlowJo plugin framework. Using this mechanism, FlowJo can invoke your own custom code to analyze populations with new algorithms or visualizations, or to integrate your analysis with your laboratory information system. By implementing a Java interface and bundling your code into a jar file, the FlowJo application will invoke your code so it can perform calculations, or communicate with external systems, and then provide results back to FlowJo for further analysis with FlowJo's tools.

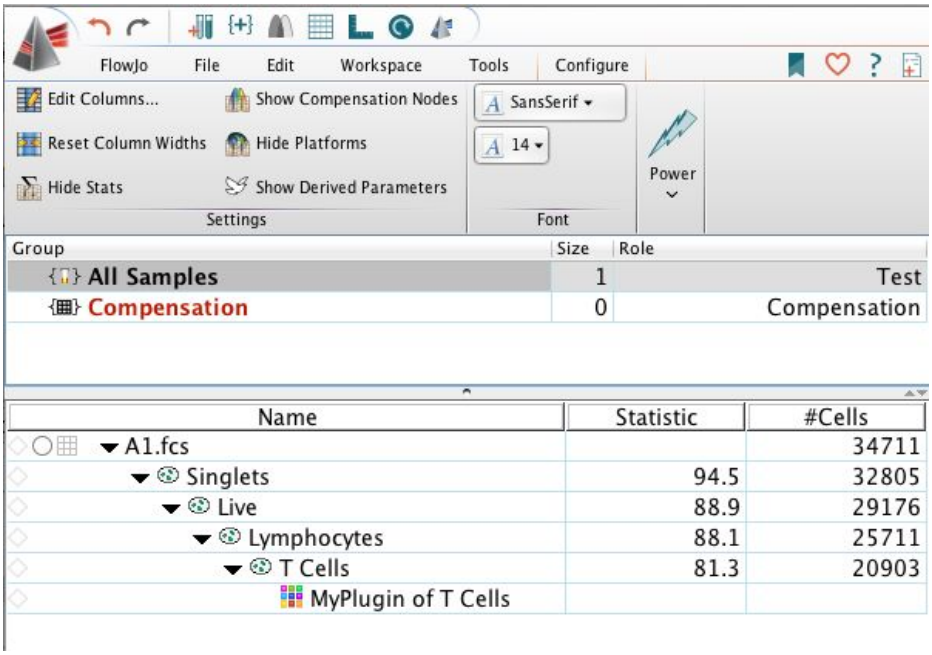
This guide will describe the two types of plugins and the simple interfaces that you implement for them. For each type of plugin, we will show example code that illustrates the basic concepts, and also introduce FlowJo's utility Java classes you can use in your plugin code. We also provide more comprehensive examples to show the entire range of capabilities. Finally this guide describes the steps to develop and deploy your plugin in the FlowJo installation on your local computer. We assume a beginner level of knowledge of the Java programming language and XML, and a basic understanding of the FlowJo application.

Types of Plugins

FlowJo provides different types of plugins to support two levels of use cases: one for analyzing individual populations, and another for access to the entire FlowJo analysis results. The two types of FlowJo plugins differ by the inputs to your code and when your code is invoked. One type of plugin, called a Population plugin, is called with the parameter data for a population as input, and your code is invoked when the plugin is created or the population's events are changed. Another type of plugin, called a Workspace plugin, is called with the entire FlowJo workspace analysis as input, and is invoked when the workspace is opened or saved. In both cases, the plugin framework uses XML as the format for input arguments, as defined by org.w3c.dom.Element. FlowJo has implemented the Element interface with class com.treestar.lib.xml.SElement which provides a number of additional convenience methods to make programming easier.

Population Plugin

To implement a Population plugin, you implement the Java interface com.treestar.lib.core.PopulationPluginInterface. This interface defines methods that allow your plugin to be represented as a 'Plugin Node' in the FlowJo workspace. The Plugin Node is shown as a row in the workspace's Sample table, and like other nodes, can be double-clicked or dragged to other populations and windows. The class that implements PopulationPluginInterface should have a zero-argument constructor that FlowJo will use to instantiate the plugin object.



The methods required to implement the PopulationPluginInterface can broadly be categorized as those which are used for the user interface, those for calculating and returning results, and those for maintaining the state of the plugin.

PopulationPluginInterface methods

// user interface methods

getName() -> String

getIcon() -> Icon

promptForOptions(SElement, List<String>) -> boolean

// calculation methods

invokeAlgorithm(SElement, File, File) -> ExternalAlgorithmResults

useExportFileType() -> ExportFileTypes

getParameters() -> List<String>

// state restoring methods

getElement() -> SElement

setElement(SElement)

getVersion() -> String

We will now look at each of these interface methods in more detail and introduce the ExternalAlgorithmResults class.

User Interface Methods

The three user interface methods are used to display a name and graphical icon in FlowJo's workspace window, and to present an opportunity for your code to prompt the user for any inputs needed to perform the calculations on the input population data. The methods are described in more detail below:

String getName()

This method returns the name of your plugin that is used to construct the plugin node name displayed in the workspace window. The name should not contain any characters that cannot be embedded as an XML attribute ('<', '>', '"')

Icon getIcon()

This method returns a javax.swing.Icon object that is displayed with the plugin node in the workspace window. If this method returns null, FlowJo will use a default icon for the plugin node.

boolean promptForOptions(SElement fcmlQueryElement, List<String> parameterNames)

This method is called when your plugin is first created by the user or when the plugin node is double-clicked in the workspace window. The purpose of this method is to allow your plugin the opportunity to display a user interface to prompt for any options needed to perform the calculations. If this method returns true, FlowJo will create the plugin node and invoke the algorithm. If false, then your plugin node is not created.

The input arguments to the promptForOptions method are used to get more information about the population and sample file on which the new plugin node is being created. The first argument, fcmlQueryElement, is an XML element used by FlowJo to fully describe the sample file, transforms, and gating hierarchy that defines the population. Later in this document we will show how to use the PluginHelper class to extract population details from this XML element, such as total number of sample events. The second input argument, parameterNames, is a list of all the parameter names defined on the sample. This list is useful for presenting a selectable list in the user interface.

Calculation Methods

The three calculation methods defined by PopulationPluginInterface are used to specify the inputs desired by your plugin code, to invoke your plugin calculations at the appropriate time, and to return results of various types back to FlowJo for further processing. When your plugin calculation code is invoked, the algorithm is given as input either an FCS or CSV file containing the event-level data for the desired parameters. Your plugin code can return a variety of results to FlowJo through the returned ExternalAlgorithmResults object. This object wraps different

result types that are used by FlowJo, such as a CSV derived parameter file, an image URL, GatingML definitions, tabular summary data, or derived parameter formulas. The calculation methods are described in more detail below:

ExportFileTypes useExportFileType()
This method returns an enumerated type, ExportFileTypes, to specify what type of input file to provide to your plugin algorithm. The returned ExportFileTypes is defined with the following values:

FCS // The input file is an FCS3.0 file
CSV_SCALE // The input file is a CSV file with parameter data in absolute data scale
CSV_CHANNEL // The input is a CSV file with parameter data in channel (transformed) values
NONE // No data input file is input to the algorithm

List<String> getParameters()
This method returns a list of parameter names to when supplying the data values for the input file to your algorithm. When your algorithm method is invoked, the input data file will contain FCS parameters or CSV columns for each of the parameters in this list. If this method returns null or an empty list, then all parameters will be included in the input file.

ExternalAlgorithmResults invokeAlgorithm(SElement fcmlQueryElement, File dataFile, File outputFolder)
This method is called when the plugin node is created or when the parent population of the plugin node is modified, to allow your algorithm to calculate and return new results to FlowJo. The first argument, fcmlQueryElement, is an XML element used by FlowJo to fully describe the sample file, transforms, and gating hierarchy that defines the population. You can use this XML element with the PluginHelper class, described later, to get additional information about the input data. The second argument is the input data file, either an FCS or CSV file. This data file will only contain the values of the parent population of the plugin node, possibly a subset of the root sample file data. The third argument is the output folder, a location on the file system where the plugin code should write any files that are generated. By writing files in this folder, the FlowJo application will automatically manage those files when the workspace is saved as an ACS file, or the plugin is deployed on a FlowJo Enterprise server. This method returns an instance of ExternalAlgorithResults, which is described in detail below. The ExternalAlgorithResults object encapsulates all the different kinds of results that your plugin can return to FlowJo.

State Restoring Methods
The remaining three methods are used to save and restore the internal state of your plugin. These methods are called prior to the promptForOptions and invokeAlgorithm methods, as well as when the FlowJo workspace is saved or reopened. The plugin implementer should implement these methods so that any internal fields are initialized or restored as needed. It is important to note that the instance of your plugin object that receives the promptForOptions

invocation may not be the same object that receives the invokeAlgorithm call. To retain any internal field values needed by your plugin, you should use the symmetric getElement / setElement methods to remember the state of your plugin. This separation is necessary so your plugin can work in FlowJo Enterprise's client/server architecture, as well as to allow your plugin to execute in a dedicated plugin engine separate from the FlowJo application. The state maintenance methods are described in more detail below:

getElement() -> SElement
This method returns an XML element that represents the state of the plugin that should be saved and restored. The returned XML element is used to retain configuration settings input by the user in the promptForOptions method, to be used later by the invokeAlgorithm or when a FlowJo workspace is saved. FlowJo provides a concrete implementation of org.w3c.dom.Element, named SElement, that provides a number of convenience methods to aid in the creation and modification of the XML element.

setElement(SElement xmlElement)
This method takes as input an XML element as produced by the getElement() method described above, to be used to restore the internal fields of the plugin object. The setElement method should read the same attributes that were written in the getElement method.

getVersion() -> String
This method returns a version string for your plugin, that can be used by your plugin code to enforce compatibility or automatically upgrade different versions of your plugin. FlowJo's plugin framework does not use the returned version string, but will ensure the version string is retained when the plugin node is saved in a FlowJo workspace.

ExternalAlgorithmResults

The ExternalAlgorithmResults is a simple class that encapsulates all the different kinds of analysis results that your population plugin can return to be further used with FlowJo. Your implementation of 'invokeAlgorithm' will return an instance of ExternalAlgorithmResults that is used to create or display additional parameters, subpopulations, statistics, or images generated by your algorithm. The methods to return different analysis results and how the results are used by FlowJo are described below:

setCSVFile(File csvFile)

A common purpose of a population plugin is to define clusters of subpopulations. Your plugin can return a single-column CSV file where each row corresponds to the events in the input data file. The value of the column is the cluster number of the event. FlowJo will use the cluster number of each event to automatically create gates on the defined clusters. This method sets the CSV file to use for cluster parameter values.

addDerivedParameterFormula(String parameterName, String formula)

FlowJo provides the ability to create new derived parameters whose value for each event is determined by a formula. The formula may reference other parameters, as well as math functions and operators, as described here. This method allows your plugin to add a new derived parameter with the given parameter name and formula.

setImageURL(URL imageURL)

Your population plugin can return a graphical image that can be displayed in FlowJo's layout editor by using this method to set the image URL. This method allows your plugin to specify the location of a generated image as a URL to a web or file resource.

setGatingML(String gatingML)

GatingML is a standard XML definition of gates and transforms defined by the ISAC Data Standards Task Force. This method allows you to specify one or more XML gate definitions to be applied to the plugin node's parent population. The root XML element of the gating specification is <gating:Gating-ML>, and your plugin code may add <gating:RectangleGate>, <gating:PolygonGate>, and <gating:BooleanGate> child XML elements. FlowJo will automatically create or update these gates when your plugin is invoked.

setStatValue(double statValue)

Each node in the FlowJo workspace can optionally display a statistical value in the workspace window. This method sets the displayed value in the 'Statistic' column of the workspace window for your plugin node.

setWorkspaceString(String wspString)

Each plugin node in the FlowJo workspace can optionally display short descriptive text in the workspace window. This method sets the displayed text in the '#Cells' column of the workspace window for your plugin node.

setTableHeaders(String[] headers)
Your plugin may generate a table of numerical values that can be displayed in FlowJo's layout editor. This method allows you to specify the text headers for each column in your table. This method is used with 'setValuesTable' described next to define the complete table to be displayed.

setValuesTable(double[][] tableValues)
This method sets the two-dimensional table of numerical values that is displayed when your plugin node is dragged to FlowJo's layout editor.

setErrorMessage(String errMsg)
This method allows your plugin to set an error message that is displayed after your plugin is invoked.

Workspace Plugin

To implement a Workspace plugin, you implement the Java interface com.treestar.lib.core.WorkspacePluginInterface. Note, the name of this interface was derived by its original purpose of interacting with a FlowJoEnterprise server, but is intended to be used for general purposes. This interface defines methods that are invoked when a FlowJo workspace is opened or saved, and when your FlowJo session is about to end, as well as methods used by the plugin framework to save the reference to your plugin in a FlowJo workspace. Note, that there is no visible representation of your workspace plugin in the FlowJo user interface. Once your plugin is added to a FlowJo workspace, its methods will automatically be invoked at the appropriate times. FlowJo will create a single instance of your plugin class when the workspace is opened, and manage the plugin object for the lifetime of the session. The class that implements WorkspacePluginInterface can be instantiated by FlowJo with one of two constructors. If your plugin defines a constructor with the single argument of type SElement, that will be used to instantiate your plugin object. Otherwise, FlowJo will use a zero-argument constructor to create your plugin object.

openWorkspace(SElement workspaceElement) -> boolean

This method is invoked when the FlowJo workspace is first opened. The input XML element describes the entire FlowJo workspace, including all samples, groups, gating hierarchy, and statistics. This method returns a boolean value indicating whether the plugin should be added to the workspace for subsequent operations. The invocation of openWorkspace occurs before FlowJo opens the workspace window, enabling your plugin to access and possibly modify the workspace XML before the analysis is visible.

saveWorkspace(SElement workspaceElement)

This method is invoked every time that the FlowJo workspace is saved. The input XML element describes the entire FlowJo workspace, including all samples, groups, gating hierarchy, and statistics.

endSession()

This method is called when the user quits the FlowJo application. It provides an opportunity for the plugin to close a database connection, send a notification, or submit logging information.

getServerUrl() -> String

This method returns a string that uniquely identifies your workspace plugin. It is used by the plugin framework to manage the plugin object for the workspace. The returned string should be a valid URL reference, that can be used to construct a Java URL instance without throwing a MalformedURLException.

getElement() -> SElement

This method returns an XML element that represents the state of the plugin that should be saved with a FlowJo workspace. If your plugin defines a constructor with a single SElement argument, this same XML is passed into the constructor.

getVersion() -> String
This method returns a version string for your plugin, that can be used by your plugin code to enforce compatibility or automatically upgrade different versions of your plugin. FlowJo's plugin framework does not use the returned version string, but will ensure the version string is retained when the plugin node is saved in a FlowJo workspace.

PluginHelper

The PluginHelper class provides a number of convenience methods you can use in your plugin code. The input to the PluginHelper's methods is the XML element argument that is passed into to your plugin methods.

The following helper methods are used by your population plugin:

getSampleURI(SElement fcmlElem) -> String
This method returns the sample data file's URI.

getSampleName(SElement fcmlElem) -> String
This method returns the name of the sample file as found in the sample's URI.

getNumTotalEvents(SElement fcmlElem) -> int
This method returns the total number of events in the sample data file (not the number of events in the parent population).

getNumExportedEvents(SElement fcmlElem) -> int
This method returns the number of events in the plugin node's parent population.

The following helper methods are used by your workspace plugin:

getWorkspaceName(SElement workspaceElement) -> String
This method returns the name of the FlowJo workspace.

getWorkspaceAnalysisFolder(SElement workspaceElement) -> File
This method returns a File that is the folder location of the FlowJo workspace file.

collectStats(SElement workspaceElement) -> Map<String, Map<String, Map<String, Double>>>
This convenience method returns a map of maps that collects all the statistics and gate counts for each population in the FlowJo workspace. The key to the outer map is the sample name of all samples contained in the FlowJo workspace. Each sample name in this map refers to a secondary map whose key is a population path name, such as "Singlets/Live/Lymphocytes/TCells". This population name is the key to a tertiary map whose key is the name of a statistic and whose value is the numerical value for that statistic.

SElement

The FlowJo plugin framework uses XML elements as the input arguments and return types of many of its methods. The signature of these methods uses SElement, a concrete implementation of org.w3c.dom.Element that also includes a number of methods to make programming easier. There are numerous methods implemented by class SElement; a list of the most useful is presented below:

Constructor SElement(String elementName)

getString(String attributeName) -> String
setString(String attributeName, String attributeValue)

getInt(String attributeName) -> int
setInt(String attributeName, int intValue)

getDouble(String attributeName) -> double
setDouble(String attributeName, double doubleValue)

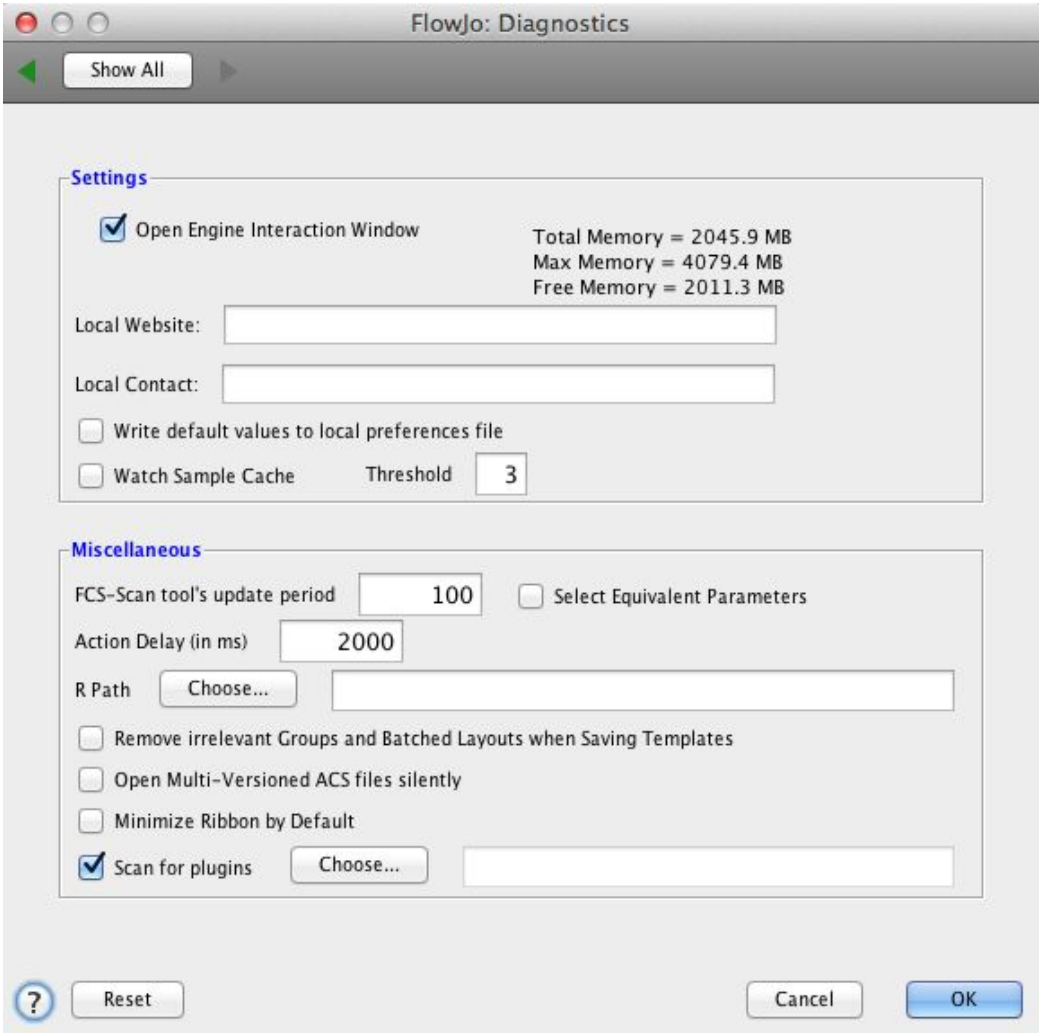
getFloat(String attributeName) -> float
setFloat(String attributeName, float floatValue)

getBool(String attributeName) -> boolean
setBool(String attributeName, boolean boolValue)

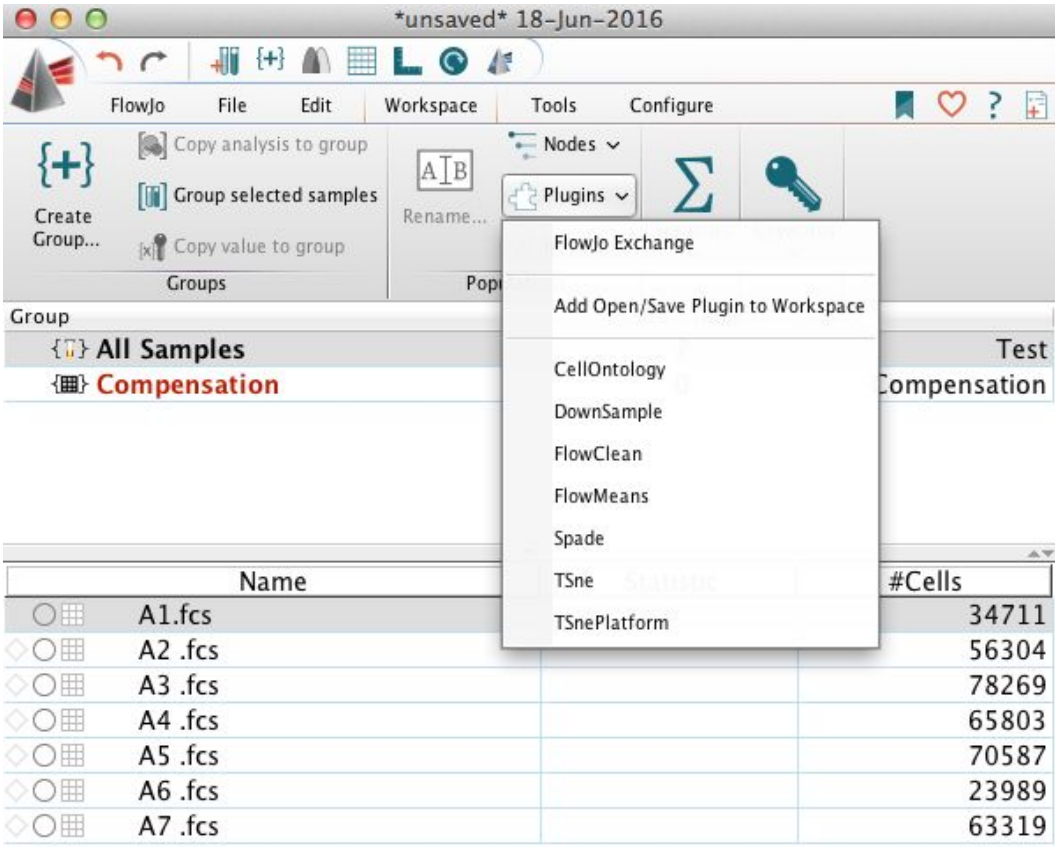
getChild(String childName) -> SElement
getChildren(String childName) -> List<SElement>

Plugin Development and Deployment

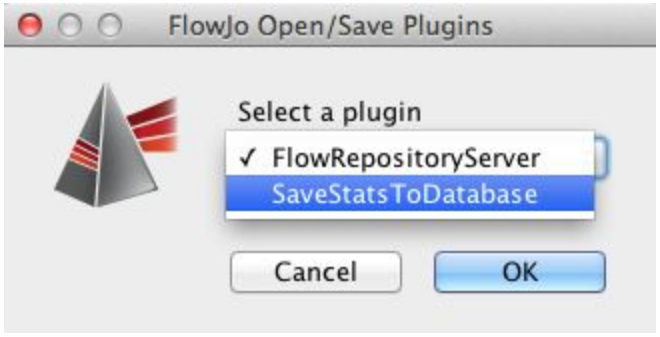
To write your plugin code, you will utilize the Java interfaces and classes that are defined in the fjlib.jar that is delivered with FlowJo. Using your favorite Java development environment, you will implement the PopulationPluginInterface or WorkspacePluginInterface, as well as utilize the ExternalAlgorithmResults and PluginHelper classes. Currently the FlowJo application uses Java 1.6, so your plugin code should be developed using the Java JDK 1.6. Once your code is developed, you will need to bundle all your modules and libraries into a single Java jar file. This jar file is placed in a folder where FlowJo will discover and install your plugins when the application is started. The location of the plugin jar files folder can be specified through the FlowJo Preferences panel.



Once your plugins are discovered by the FlowJo application, you will be able to create your plugins through the FlowJo user interface. The plugin operations are visible in the Workspace ribbon under the Plugins menu. This menu will display a menu item for each population plugin, to allow the user to add a plugin node as a child of the selected population.



In addition, the Plugins menu provides a menu item to ‘Add Open/Save Plugin to Workspace’, which will display a list of available workspace plugins that can be added to the current FlowJo workspace.



When a workspace plugin is added to the current FlowJo workspace, an instance of your plugin is created and added to the workspace. When the workspace is saved, your plugin’s saveWorkspace method will be invoked.

Example Population Plugin

```
package com.plugins;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import javax.swing.Icon;
import com.treestar.flowjo.engine.utility.ClusterPrompter;
import com.treestar.lib.PluginHelper;
import com.treestar.lib.core.ExportFileTypes;
import com.treestar.lib.core.ExternalAlgorithmResults;
import com.treestar.lib.core.PopulationPluginInterface;
import com.treestar.lib.fjml.FJML;
import com.treestar.lib.fjml.types.FileTypes;
import com.treestar.lib.xml.SElement;

/**
 * This test class illustrates the implementation of the External Population Algorithm plugin.
 * It doesn't do anything useful, but attempts to show all the different types of return values
 * that can be used by ExternalAlgorithmResults.
 */
public class ExternalAlgorithmTest implements PopulationPluginInterface {

   private List<String> fParameters = new ArrayList<String>(); // the list of $PnN parameter names to be used
   private SElement fOptions; // an XML element that can hold any additional options used by the algorithm

   @Override
   public String getName() {              return "ExternalAlgorithmTest";    }

   /*
    * Return an XML element that fully describes the algorithm object and can be used to
    * reconstitute the state of the object.
    */
   @Override
   public SElement getElement() {
            SElement result = new SElement(getClass().getSimpleName());
            if (fOptions != null)
                     result.addContent(new SElement(fOptions)); // create a copy of the XML element
            // construct an XML element for each parameter name
            for (String pName : fParameters)
            {
                     SElement pElem = new SElement(FJML.Parameter);
```

```
                pElem.setString(FJML.name, pName);
                result.addContent(pElem);
        }
        return result;
}

/*
 * Use the input XML element to set the state of the algorithm object
 */
@Override
public void setElement(SElement elem) {
        fOptions = elem.getChild("Option"); // could be null
        // clear the parameter list and re-create from the XML element
        fParameters.clear();
        for (SElement child : elem.getChildren(FJML.Parameter))
                fParameters.add(child.getString(FJML.name));
}

@Override
public List<String> getParameters() {
        return fParameters;
}

@Override
public Icon getIcon() {
        return null;
}

/*
 * This method uses class ClusterPrompter to prompt the user for a list of parameters and number of clusters.
 */
public boolean promptForOptions(SElement fcmlQueryElement, List<String> parameterNames)
{
        SElement algorithmElement = getElement();
        // pass the XML element to the cluster prompter
        ClusterPrompter prompter = new ClusterPrompter(algorithmElement);
        if (!prompter.promptForOptions(null, parameterNames, true))
                return false;
        algorithmElement = prompter.getElement();
        setElement(algorithmElement);
        return true;
}

/*
 * This method shows how to return different kinds of values using the ExternalAlgorithmResults object.
 */
@Override
public ExternalAlgorithmResults invokeAlgorithm(SElement fcmlElem, File sampleFile, File outputFolder) {
        ExternalAlgorithmResults result = new ExternalAlgorithmResults();
```

```
		// 1. Create a CSV file that contains random cluster numbers
		// (CSV file is created in the given output folder)
		File outFile = new File(outputFolder, "Random." + sampleFile.getName() + FileTypes.CSV_SUFFIX);
		Writer output;
		// get the number of clusters from the options XML element
		int numClusters = fOptions == null ? 2 : fOptions.getInt("cluster");
		try {
			// get the number of total events in the sample file, using a a plugin helper method
			int num = PluginHelper.getNumExportedEvents(fcmlElem);
			if (num <= 0)
				return null;
			// now write CSV file with the correct number of rows, each row containing a random cluster
number
			output = new BufferedWriter(new FileWriter(outFile));
			for (int i = 0; i < num; i++) {
				int rand = (int)(Math.random() * 100);
				int cluster = rand % numClusters;
				output.write("" + cluster);
				output.write("\n");
			}
			output.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
		// 2. Set the CSV file to be used by FlowJo to create a derived parameter that is then auto-gated
		result.setCSVFile(outFile);

		// 3. Set the number of clusters, displayed in the workspace window as the stat value
		result.setStatValue(numClusters);

		// 4. Set the workspace description string, displayed in the workspace window
		result.setWorkspaceString("" + numClusters + " clusters");

		// 5. Create a 2-dimensional table of values, with a header row
		double[][] table = new double[numClusters][numClusters];
		String[] headers = new String[numClusters];
		for (int i = 0; i < numClusters; i++)
		{
			headers[i] = "Column " + i;
			for (int j = 0; j < numClusters; j++)
				table[i][j] = (i+1) * (j+1);
		}
		// 6. Set the table header and the table values in the result object
		result.setTableHeaders(headers);
		result.setValuesTable(table);

		// 7. Create a new formula that describes a derived parameter
		if (fParameters.size() > 1)
		{
			// the formula is the first parameter's value divided by the second parameter's value
```

```
                String formula = fParameters.get(0) + " / " + fParameters.get(1);
                // add the formula to the result object (can add more than one)
                result.addDerivedParameterFormula(fParameters.get(0) + fParameters.get(1) + "Ratio", formula);
        }

        // 8. Create a URL to an image and set the image URL in the result
        try {
                URL url = new URL("http://www.flowjo.com/wp-content/uploads/2015/09/Enterprise-for-web.png");
                result.setImageURL(url);
        } catch (MalformedURLException e) {
                e.printStackTrace();
        }

        // 9. Create a Gating-ML XML element that describes a gate
        // create the XML elements for a 1-D range gate
        SElement gate = new SElement("gating:Gating-ML");
        SElement rectGateElem = new SElement("gating:RectangleGate");
        rectGateElem.setString("gating:id", "TestGate");
        gate.addContent(rectGateElem);
        // create the dimension XML element
        SElement dimElem = new SElement("gating:dimension");
        dimElem.setInt("gating:min", 50000);
        dimElem.setInt("gating:max", 100000);
        rectGateElem.addContent(dimElem);
        // create the parameter name XML element
        SElement fcsDimElem = new SElement("data-type:fcs-dimension");
        fcsDimElem.setString("data-type:name", fParameters.get(0));
        dimElem.addContent(fcsDimElem);

        // 10. Set the Gating-ML element in the result
        result.setGatingML(gate.toString());
        return result;
    }

    /**
     * This method allows the algorithm to specify if the input file should be a CSV file (CSV_SCALE OR
CSV_CHANNEL), or an FCS file based on algorithm's requirements.
     * @return ExportFileTypes One of three values (ExportFileTypes.FCS, ExportFileTypes.CSV_SCALE,
ExportFileTypes.CSV_CHANNEL) will be returned according to algorithm's requirement.
     */
    @Override
    public ExportFileTypes useExportFileType() {
        return ExportFileTypes.CSV_SCALE;
    }

    @Override
    public String getVersion() {        return "1.0";    }

}
```

Example Workspace Plugin

```
package com.plugins;

import java.io.File;
import java.text.SimpleDateFormat;
import com.treestar.lib.PluginHelper;
import com.treestar.lib.core.WorkspacePluginInterface;
import com.treestar.lib.fjml.types.FileTypes;
import com.treestar.lib.prefs.HomeEnv;
import com.treestar.lib.xml.SElement;
import com.treestar.lib.xml.XMLUtil;

public class ServerPluginTest implements WorkspacePluginInterface {

   private String opened;
   @Override
   public String getServerUrl() {
            return "http://localhost:8080/ServerPluginTest";
           }

   @Override
   public SElement getElement() {
            SElement result = new SElement("ServerPluginTest");
            if (opened != null && !opened.isEmpty())
                     result.setString("opened", opened);
            return result;
   }

   @Override
   public boolean openWorkspace(SElement workspaceElement) {
            opened = getDateStamp();
            return true;
   }

   private File getOutputFolder(SElement workspaceElement)
   {
            File file = PluginHelper.getWorkspaceAnalysisFolder(workspaceElement);
            if (file != null && file.exists())
                     return file;
            String home = HomeEnv.getInstance().getUserHomeFolder();
            File homeFolder = new File(home);
            if (homeFolder.exists())
                     return homeFolder;
            return null;
   }
   private String getDateStamp()
   {
            SimpleDateFormat dateFormatter = new SimpleDateFormat("EEE, MMM d, yyyy HH:mm:ss z");
```

```
            return dateFormatter.format(System.currentTimeMillis());
    }
    @Override
    public void save(SElement workspaceElement) {
            String name = "ServerPluginTest";
            File folder = getOutputFolder(workspaceElement);
            File saveFile = new File(folder, name + FileTypes.XML_SUFFIX);
            int ct = 1;
            while (saveFile.exists())
            {
                    String nextName = name + "." + ct++;
                    saveFile = new File(folder, nextName + FileTypes.XML_SUFFIX);
            }
            SElement out = getElement();
            SimpleDateFormat dateFormatter = new SimpleDateFormat("EEE, MMM d, yyyy HH:mm:ss z");
            out.setString("saved", dateFormatter.format(System.currentTimeMillis()));
            new XMLUtil().write(out, saveFile);
    }

    @Override
    public void endSession() { }

    @Override
    public String getVersion() {    return "1.0";    }

}
```