JUMP

The Unified Mapping Platform

Developer's Guide

Prepared by:

VIVID SOLUTIONS

Document Change Control

REVISION NUMBER	DATE OF ISSUE	AUTHOR (S)	BRIEF DESCRIPTION OF CHANGE
0.1	25-Mar-2003	Jon Aquino	Initial outline
0.2	26-Mar-2003	Jon Aquino	Initial body
0.3	17-Apr-2003	Jon Aquino	JUMP as framework and toolkit, Java2XML
0.4	13-Aug-2003	Jon Aquino	Changed Configuration to Extension
0.5	31-Oct-2003	Djun Kim	Expanded some of Jon's to-do items, added items per wish-list/to-do list.

Licence

For more information, contact:

Vivid Solutions
Suite #1A
2328 Government Street
Victoria BC V8T 5G5
Canada

(250) 385-6040
www.vividsolutions.com

Credits

JUMP was developed by Vivid Solutions Inc. of Victoria, B.C., Canada under a project jointly sponsored by GeoConnections, the British Columbia Ministry of Sustainable Resource Management (MSRM), the Canadian Centre for Topographic Information at Sherbrooke (CTI-S) and the Ontario Ministry of Natural Resources. (OMNR) Additional code was contributed by Refractions Research Inc. of Victoria, B.C., Canada.

Table of Contents

1. INTRODUCTION

The JUMP Developer's Guide describes the architecture of the Unified Mapping Platform (JUMP). This document is intended for developers who wish to extend JUMP by writing their own plugins, cursor tools, renderers, or datasources. Readers looking for a description of JUMP's built-in features should refer to the related documents JUMP Data Sheet and JUMP User Guide.

The major components of the JUMP architecture are shown in Figure 1-1 below.

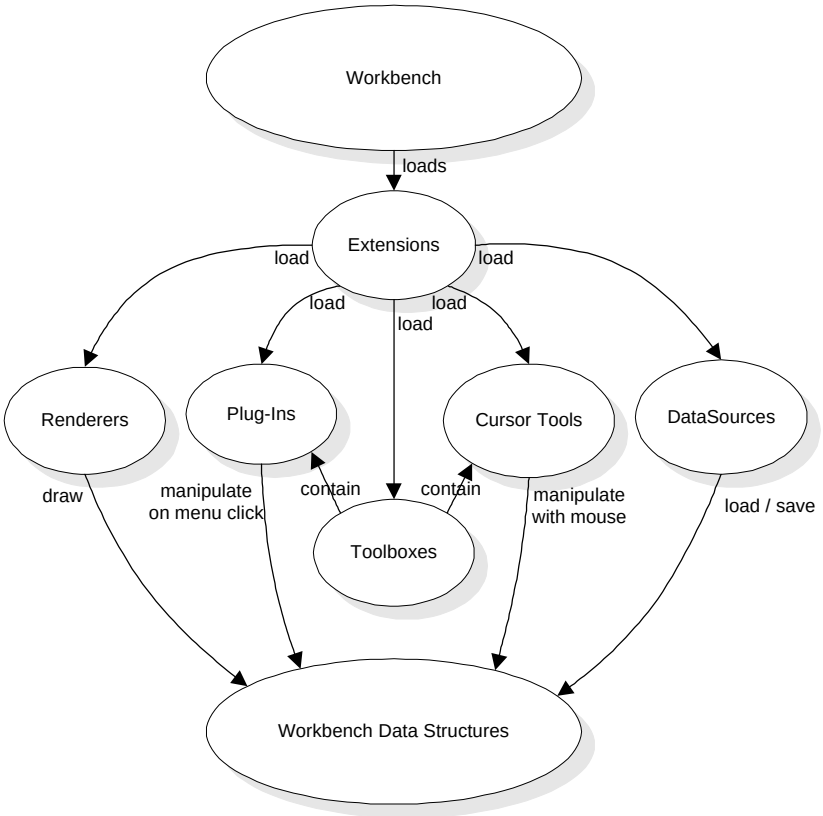


Figure 1-1 – JUMP architecture

On start-up, the Workbench loads extensions, which are JARs that add functionality to the Workbench. This additional functionality may take the form of plugins (menu items), cursor tools (toolbar buttons), renderers (ways to draw the data), and datasources (ways to load and save various data formats).

1.1 Nine Reasons To Use JUMP As A Framework

Up front, here are nine reasons why you might want to consider using JUMP as the framework for your next application. (Note that there are other worthy frameworks out there, for example, Eclipse).

- Splash Screen. Part of the JUMP framework is a splash screen that disappears as soon as your main window appears. See how it's done in the `#main` method in `JUMPWorkbench.java`
- Multiple Document Interface (MDI). The default configuration of the JUMP framework is MDI – your application can have multiple windows. This is a plus for many applications; however, if MDI isn't for you, you can make a regular single-document interface frame and still use the rest of the JUMP framework (splash screen, options dialog, etc.)
- Reusable Actions. Menu actions are defined once; they can then easily be used in several places: main menus, popup menus, even toolbar buttons. All you have to do is implement the `PlugIn` interface. Example: `NewTaskPlugIn.java`.
- Advanced Menuing. The JUMP Workbench makes it easy for you to add logic about when menus should be enabled and disabled. Many frameworks do this, but this framework also allows you to specify the reason for disabling the menu – the user simply holds the mouse cursor over the disabled menu item and gets a tooltip reason, like "Fence needs to be drawn" or "At least 1 layer must be selected". Simply write an anonymous class on the fly that implements the `EnableCheck` interface. Example: `ChangeStylesPlugIn#createEnableCheck`.
- Cursor Tools. If your application needs to handle mouse clicks (e.g., for drawing), the JUMP Workbench makes it easy. Simply subclass `DragCursorTool` (if you want the user to drag out a box) or `MultiClickTool` (if you want the user to click several places, for example, as when drawing a polygon), or one of the other useful classes in the `AbstractCursorTool` hierarchy. Example: `DrawRectangleFenceTool.java`.
- Unobtrusive Warnings. A lot of applications will throw up error dialogs to show minor warnings or information, interrupting the user's flow of work. With the JUMP framework, you can show warnings in the status bar. The warning is yellow and flashes for a second, to get the user's attention, but doesn't interrupt the user's train of thought because there is no OK button to push. Simply call `WorkbenchFrame#warnUser`.
- Error Dialog. Of course, a full-blown error dialog is available if you really need to grab the user's attention. In fact, unhandled exceptions automatically percolate up to the error dialog. The error dialog has a Show/Hide Details button that reveals the Java exception stack trace, which is useful for debugging. (To do: build a framework for reporting errors to the user in a friendly manner. For example, exception must have a friendly message, as well as cause, and recommended actions).
- Window List. Another little bonus you get is a Window menu that lists the open windows. Again, not hard to write, but not fun to write either!
- Options Dialog. The Edit / Options menu will open a tabbed dialog that you can store your program options under, so that the user can set his or her preferences. Example: `InstallGridPlugIn#initialize`.

1.2 Five Reasons To Use JUMP As A Toolkit

So you don't want to use JUMP as the primary framework for your application, or you're writing a web app as opposed to a Java application and thus can't use JUMP as a framework. JUMP can still be useful to you as a source of powerful components:

- `Wizard`. There isn't much on the Internet in the way of a free Wizard framework. JUMP has one, and it looks pretty good!
- `AddRemovePanel`. This is a frequently used GUI idiom, and there isn't one in Swing. Try JUMP's — it's easy to use, and you can plug in whatever you want into the panels — even trees!
- `LayerViewPanel`. This is JUMP's core component — the panel that displays all the maps. You get lots of functionality for free: zoom logic, fast draws, and dozens of pre-built cursor tools that you can plug into your own app (drawing, moving, panning, selecting, etc.). Add it to your web app!
- `Java2XML`. Lots of people are interested in this tool. It's a little class that turns the objects you feed it into XML documents, and vice versa. If you don't want to use Castor because you don't want to bother with XML Schema, and Bewitched is giving you inexplicable errors, give `Java2XML` a try! (See Section 10, Java2XML).
- `MultiInputDialog`. Sometimes it's fun to build a dialog; sometimes it's a pain. `MultiInputDialog` will let you whip up a new dialog in minutes rather than hours. Simply give it the types of the fields (e.g., Integer, Boolean, String), the label strings, perhaps an optional image, and presto! Instant dialogs that look pretty good! Examples: the `Boundary Match Data` dialog under the `Tools > Generate` menu, or the (nice looking!) `Validate Selected Layers` dialog under the `QA` menu.

1.3 Quick Answers

If you want to get started right away, here's a few common tasks you may find yourself wanting to do as you start to write JUMP code.

1. JUMP looks like a great framework to build my implementation on. Where do I start with learning to extend JUMP?

   Most JUMP applications provide one or more plugins (that is, actions, for example menu items) via which a user interacts with data. Learning how to write and install a plugin is a great place to begin looking at extending JUMP.

2. OK, so how do I make a menu that pops up a "Hello World" dialog?

   Subclass `AbstractPlugIn`. See Section 2.1 for an example of how to write the "Hello World" dialog as a JUMP plugin.

3. How do I package it into a jar that other JUMP users can drop into their systems?

   Implement `Extension`. See Section 2 Extensions.

4. OK now I want to make a dialog that takes 10 values. But I don't want to spend a lot of time on GUI code — I just want to try it out!

   Whip up a quick GUI with `MultiInputDialog`.

5. My plugin takes a long time, and while it's running, I can get the GUI to blank out. In fact, the GUI seems frozen!

   Subclass `ThreadedPlugIn` rather than `AbstractPlugIn`. Note: `ThreadedPlugIn` is not truly threaded, just more GUI-friendly.

6. I wish Layer stored a timestamp for the date/time it was created!

   Store the time in its `Blackboard` when you create it.

7. When I run my plugin, the undo and redo buttons become disabled!

   Handle undo/`redo`. See Section 11, Undo.

8. I want my plugin to get the selected Features.

   Use `SelectionManager#getSelectedFeatures`

9. Is there an easy way to output stuff from my plugin?

   Use `PlugInContext#getOutputWindow`

10. I want to make a Layer with thick lines and green fill.

    Use `Layer`, `LayerManager#add(Layer)`, `Layer#getBasicStyle`

1.4 CONFIGURING JUMP

Plugin Directory: JUMP's Extension mechanism makes it easy for users to add functionality to the Workbench: they simply need to copy a JAR file into their application's workbench plugin directory. By default, this directory is `lib\ext\` in the directory where JUMP is installed (`lib/ext/` in the Unix/Linux/MacOS world). It is, however, possible to specify another directory by specifying the `–plug-in-directory` command line option when JUMP is started.

Under Windows, this would require a command-line parameter like the following:

```
-plug-in-directory C:\Sandbox\HelloWorld\plugins
```

Under MacOS X, GNU/Linux, or other versions of Unix, the command-line parameter would look like:

```
-plug-in-directory /home/jumpuser/Sandbox/HelloWorld/plugins
```

Properties File: The Workbench allows developers to specify the name of plugin classes in the workbench properties file, an XML file whose location can be specified in a command-line argument for the JUMP Workbench. Thus, the Workbench looks for plugins in two places: in JAR files in the workbench plugin directory, and in classes specified in the workbench properties file. These class names are passed to the JVM's classloader, which searches for them in the Java `CLASSPATH`.


War Journal: In the future, you will be able to specify cursor tools and configurations directly in the Workbench properties file. For now, you can load them indirectly via a plugin.

To specify a workbench properties file in your IDE:

- Create a file somewhere called "workbench-properties.xml" with the format below. Put the name of your class between the `<plug-in>` tags.

```
<workbench>
  <plug-in>example.HelloWorldPlugIn</plug-in>
</workbench>
```

- In your IDE, where you specify `JUMPWorkbench` as the class to run, specify the location of the workbench properties file as a program argument. Under Windows, this would require a command-line parameter like the following:

```
-properties C:\Sandbox\HelloWorld\workbench-properties.xml
```

Under MacOS X, GNU/Linux, or other versions of Unix, the command-line parameter would look like:

```
-properties /home/jumpuser/Sandbox/HelloWorld/workbench-properties.xml
```

- You're done! Now when you run JUMPWorkbench, it will call your plugin's `#initialize` method during startup.

1.5 BUILDING JUMP FROM SOURCE CODE

§ JUMP Configuration (see Section 1.4, Configuring JUMP)
§ How to set up your IDE to compile JUMP. Extensions that your IDE must copy from the source directory to the classes directory: `.java2xml`, `.html`, `.txt`
- Building using Ant
- JavaDoc

Tip: When you are developing JUMP extensions, it can be tedious to generate and install a JAR file each time you test a change in the plugin classes. Instead, you can specify the name of your plugin classes in a workbench properties file (see Section 1.4, Configuring JUMP on page 8)

1.6 TO DO

There are many areas in which this document could be improved. The following to-do list captures some of the areas we have identified as tasks. Please feel free to add to this list, or better yet, adopt a task and contribute to the JUMP Developer's Guide!

§ Add more Doxygen generated class-hierarchy diagrams. (Perhaps get Doxygen to generate the 'dot' output, manually tweak the result and render in PostScript to make the output more legible at small sizes?)
§ The persistent blackboard
§ Future: `getIcon`, `getEnableCheck`
§ How to avoid breaking the undo chain (report nothing to undo or emit an `UndoableCommand`). How to then change your mind and break the undo chain (report irreversible change).
§ Handling errors, etc. — see Section 14, Handling Errors, Exceptions and Warnings.
  o Developers are encouraged to use `WorkbenchFrame#warnUser` to flash a brief message in the status bar rather than throw error dialogs in front of the user

(considered annoying by many). It is generally safer to call #warnUser on WorkbenchFrame rather than LayerViewPanelContext, because the latter assumes that the active frame has a LayerViewPanel on it, which may not be the case.

- Handling selected features, parts, and linestrings — see getSelectionManager.getFeatureSelection.selectItems

- SelectionManager — plugins often operate on a user's selected data. There are three kinds of selections: FeatureSelection, PartSelection, and LineStringSelection. FeatureSelection is just the collection of selected features; PartSelection is the collection of selected parts, etc. These would be good to JavaDoc.
- Adding to the OptionsDialog (future: options will be persistent)
- HTMLFrame ("Output Frame"): This is a quick and easy way to produce output from your plugin
  - You must call #createNewDocument first.
  - Call #surface if you really want to show the Output Frame (note: this can be annoying to the user). Otherwise, the icon will flash (more benign).

======= Very Important (beginner, intermediate, and advanced developers)

- Writing an "extension" (i.e., an extension JAR). I wish we had called "extensions" "PlugIns" and "PlugIns" "actions". See Section 2.1, Example: Building a Simple JUMP Extension.
- Would be good to fully JavaDoc PlugInContext (passed into a PlugIn).
- EnableChecks contain logic to decide whether to enable/disable a menu item (see Section 4.2, Enable-Checks). They display a tool tip showing the reason it is disabled.
- Finding the active window (because it's MDI): PlugInContext#getActiveInternalFrame. Finding whether it contains a map panel (aka LayerViewPanel): instanceof LayerViewPanelProxy.
- Main JUMP structures: LayerViewPanel (the map on the right), LayerTreePanel (the tree on the left), Task, LayerManager, Layer, FeatureCollection, Feature, Geometry.
- ThreadedPlugIn: This is not really threaded — a better name would have been "LongOperationPlugIn". Advantage over regular AbstractPlugIns: you see a (modal) busy dialog, and the GUI doesn't go blank. Disadvantage: The busy dialog will flicker on then off if your plugin is quick.
- Shortcut keys: simply put a "&" before the letter before creating the menu with FeatureInstaller (this is a new feature in JUMP 1.1, not JUMP 1.0.0)
- Undo/redo. Good plugins will be undoable. EditTransaction can ease the pain. See UndoableEditReceiver.

========== Less Important (intermediate and advanced developers)

- How to write a cursor tool (i.e., a toolbar button that changes the mode of the mouse cursor e.g., MoveCursorTool). There's a whole class hierarchy of cursor tools from which to jump off (e.g., DragCursorTool, MultiClickCursorTool). See Section 6, CursorTools for a beautiful graphic of the hierarchy.
- Toolboxes (a floating dialog that holds cursor tools and arbitrary Swing components. Martin thinks them handy.)

- Persisting properties between JUMP sessions: prevents annoyances like dialogs that forget the values entered by the user in a previous session. (Available in JUMP 1.1, not JUMP 1.0.0)
- JUMP's `Wizard` framework. See Section 12, Feature Text Writers
- The WKT, GML, and Coordinate List buttons on the left side of the Feature Info window are examples of feature text writers – they are different ways to display the contents of a Feature.

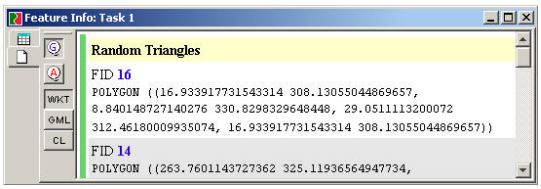


Figure 12-1 – The Feature Info window.

You can easily add custom feature text writers via a plug-in. Simply subclass AbstractFeatureTextWriter (and implement its one abstract method); then in your plug-in's `#initialize` method, say:

```
context.getWorkbenchContext().getFeatureTextWriterRegistry()
        .register(myFeatureTextWriter)
```

This code will add a button to the Feature Info window.

For code examples, see InstallStandardFeatureTextWritersPlugIn.java.

- The Wizard Framework
- Various kinds of events: `FeatureChangedEvent`, `LayerChangedEvent`. Some plugins may find them useful.
- `LayerManagerProxy`: Some plugins can operate on windows that are merely `LayerManagerProxies` (i.e., non-visual views of the data. An example of this is the table-like `AttributeViewer`). These don't need to be full-blown `LayerViewPanelProxies`. Such plugins simply need to check that the active window is an instance of `LayerManagerProxy` in their enable-checks.

`========== Least Important (advanced developers)`

- Writing a `renderer` (i.e., custom drawing). `Styles` are a bit easier to implement than `Renderers` — with `Styles` you just worry about rendering one feature at a time. `Renderers` are very general (e.g., `WMSLayerRenderer` draws satellite images in the background. This was a difficult renderer to write).
- Writing a `DataSource` (connecting to files and databases: the plumbing and the UI)
- Adding tabs to the `Options` dialog
- `Java2XML`. Easy way to get Java objects into XML and back (although you don't have total control over what the XML looks like). Some developers may find it handy, even outside of JUMP. The API has one or two warts, but only long-time `Java2XML` users will notice them.

2. EXTENSIONS

An `extension` is a collection of classes and supporting resources that provides additional functionality to JUMP. Extensions are packaged as JAR files. From the user's perspective, extending JUMP is as easy as copying an extension JAR file into the JUMP application's workbench plugin directory (see Section 1.4, Configuring JUMP)

Typically, an `Extension` will add `plugins` (menu items) and `cursor tools` (toolbar buttons) to the Workbench. Plugins and cursor tools are discussed more fully in later sections.

The JUMP Workbench will search the JAR file for subclasses of `Extension`. (Note: They must also be named "…`Extension`"). It will then call the `#configure` method on each `Extension` class it finds.

2.1 EXAMPLE: BUILDING A SIMPLE JUMP EXTENSION

In this section we will see how to write a simple JUMP `PlugIn`, how to package it as an Extension and how to install it to make it available to JUMP.

Let's walk through the creation of an extension that writes "Hello, World!" to the Workbench Output Window. First, create the plugin:

```
package example;

import com.vividsolutions.jump.workbench.plugin.AbstractPlugIn;
import com.vividsolutions.jump.workbench.plugin.PlugInContext;

public class HelloWorldPlugIn extends AbstractPlugIn {

   public void initialize(PlugInContext context) throws Exception {
      context.getFeatureInstaller().addMainMenuItem(this,
         new String[] { "Tools", "Test" }, getName(), false, null, null);
   }
   public boolean execute(PlugInContext context) throws Exception {
      context.getWorkbenchFrame().getOutputFrame().createNewDocument();
      context.getWorkbenchFrame().getOutputFrame().addText("Hello, World!");
      context.getWorkbenchFrame().getOutputFrame().surface();
      return true;
   }
}
```

Listing 2-1 – Hello World plugin

Next, create an `Extension` that loads it:

```
package example;

import com.vividsolutions.jump.workbench.plugin.Extension;
import com.vividsolutions.jump.workbench.plugin.PlugInContext;

public class MyExtension extends Extension {

   public void configure(PlugInContext context) throws Exception {
      new HelloWorldPlugIn().initialize(context);
   }
}
```

Listing 2-2 – Hello World Extension

Now, create a JAR file containing these two classes and drop it into the Workbench's plugin directory (see Section 1.4, Configuring JUMP). When you next start JUMP, you will see a new menu item: `Tools > Test > Hello World`. Selecting it will open the Output Window, which will display the "Hello, World!" message.

You might wonder where the Workbench got the menu name "Hello World" — it's not anywhere in the `HelloWorldPlugIn` code. Generating a friendly name from the class name is one of the useful functions provided by `AbstractPlugIn` (and is an incentive to create meaningful plugin class names!)

Tip: When you are developing a plugin, it is tedious to generate and install a JAR file each time you test a change in the plugin classes. Instead, you can specify the name of your plugin classes in a workbench properties file (see Section 1.4, Configuring JUMP on page 8)

2.2 Main Classes

Class	Package
`AbstractPlugIn`, `PlugInContext`, `ThreadedPlugIn`. `Extension`	`com.vividsolutions.jump.workbench.plugin`

2.3 To Do


Note: This section is under construction

3. PLUGINS

A plugin is an object that performs a single action, in response to a menu selection or a button press.

There are dozens of examples of plugins in the JUMP source code, ranging from the simple (`NewTaskPlugIn`) to the complex (`ValidateSelectedLayersPlugIn`). In fact, every menu item in the Workbench (including popup menus) is a plugin. They are all loaded by the `JUMPConfiguration` class.

A plugin has three methods: `#initialize`, `#execute`, and `#getName`. The `#initialize` method is called when the Workbench starts up. `#execute` is called when the plugin is triggered, e.g., by the user selecting a menu item or clicking a toolbar button.

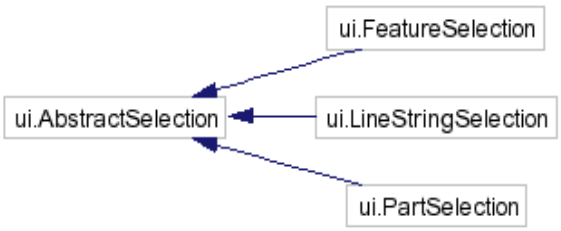
- To add a plugin to the main menu, use
  - `PlugInContext.getFeatureInstaller().addMainMenuItem(…)`
- To add a plugin to the toolbar, use
  - `PlugInContext.getWorkbenchFrame().getToolBar().addPlugIn(…)`

Most Plugins, in order to do something useful, must manipulate Workbench data structures. For more information on how to do this, see Section 5, Workbench Data Structures.

3.1 WORKING WITH SELECTIONS

Selections can be a tricky concept to grasp at first. In JUMP, the term "selection" is always singular, never plural. What's plural is the term "selected items".

A `LayerViewPanel` has a `SelectionManager`, which has three kinds of selection: `FeatureSelection` (in which the user has selected whole features), `PartSelection` (in which the user has selected parts of a `GeometryCollection` feature), and `LineStringSelection` (in which the user has selected `LineStrings` inside a `Polygon` or `GeometryCollection`). So a user can select not only whole features but also pieces of them.



If your plugin needs all features with some sort of selection (whole or otherwise), use:

```
selectionManager.getFeaturesWithSelectedItems().
```

If your plugin needs to be able to distinguish between the different kinds of selection, you'll need to drill down a bit:

```
selectionManager.getFeatureSelection().getSelectedItems()
selectionManager.getPartSelection().getSelectedItems()
selectionManager.getLineStringSelection().getSelectedItems()
```

3.2 THREADING

If your plugin performs long-running, intensive computations, you may want to subclass `ThreadedPlugIn` rather than `AbstractPlugin`. Note `ThreadedPlugIn` is not "truly" threaded since we don't have a locking mechanism yet. A better name might have been '`LongRunningPlugIn`'.

3.3 CONTEXTS

This section will be about `PlugInContext` and the more general `WorkbenchContext`, which plugins use to get information from the JUMP core (like the current selection or the current `LayerManager`). It might have notes on the usefulness of each method in the contexts; these notes can also be used as JavaDoc.

3.4 MAIN CLASSES

Class	Package
`AbstractPlugIn, PlugInContext, EnableCheck, MultiEnableCheck, ThreadedPlugIn`	`com.vividsolutions.jump.workbench.plugin`
`WorkbenchContext`	`com.vividsolutions.jump.workbench`
`MultiInputDialog, HTMLFrame, LayerNamePanel, EditTransaction, SelectionManager`	`com.vividsolutions.jump.workbench.ui`
`TaskMonitor`	`com.vividsolutions.jump.task`
`UndoableCommand,`	`com.vividsolutions.jump.workbench.model`
`WizardDialog`	`com.vividsolutions.jump.workbench.ui.wizard`


Note: This section is under construction

3.5 TO DO

- Plugins need not create a menu item per se, although that is their typical use; they have full access to JUMP data structures and are free to manipulate them
- `PlugInContext` is a "snapshot", whereas `WorkbenchContext` is "current" e.g., the currently active window
- `MultiInputDialog` is an easy way to put together a quick-and-dirty dialog, especially if it needs lots of fields. `CalculateAreasAndLengthsPlugIn` in `com.vividsolutions.jump.workbench.plugin.analysis` is a good example of this.
- It's best to make your plugin undoable. Classes which will help with this: `Layer#addUndo, EditTransaction`

4. MENU ITEMS

Menu items are a fundamental element of the JUMP user interface. They provide a binding between user actions and plugin functionality.

4.1 ADDING ITEMS TO MENUS

- use `addMainMenuItem` to add an item which is always available
- use `addLayerViewMenuItem` to add a menu item which is only enabled when a Task window has the focus
- use `addLayerNameViewMenuItem` to add a menu item which is only enabled when a Task window has the focus, and that Task window has a Layer Name panel visible

4.2 ENABLE-CHECKS

Have you ever been frustrated by an application disabling a menu item with no explanation? The Workbench provides a mechanism to prevent these mysteries: `enable-checks`. These make it easy for developers to specify when a menu item should be disabled and why.

When you add a plugin as a menu, one of the parameters in `#addMainMenuItem` is an `EnableCheck`. An enable-check has a single method that returns null if the menu item should be enabled or a `String` if the menu item is disabled. This String gives the reason for disabling the menu (e.g., "At least one feature must be selected"). When the user places the mouse cursor over the disabled menu item, the reason will be displayed as a tooltip.

The enable-check is called whenever the parent menu is opened.

You'll find 20 or so commonly used enable-checks already written for you in the `EnableCheckFactory` class. For example:

- At least n features must be selected
- At least n layers must be selected
- A fence must be drawn

You will probably want to specify more than one enable-check for your plugin. The `MultiEnableCheck` class will allow you to combine several enable-checks.

In addition, an enable-check allows you to get at the `JMenuItem` itself (it's passed in as a parameter). So, for example, if you wanted to update the `JMenuItem` text whenever the parent menu was opened, you could do so in an `EnableCheck` (and simply return null).

4.3 ADDING A POPUP MENU

To add a right-click popup menu use the following code.

```
JPopupMenu popupMenu = LayerViewPanel.popupMenu();

featureInstaller.addPopupMenuItem(popupMenu,
                                  featureInfoPlugIn,
                                  featureInfoPlugIn.getName(),
                                  false ,
                                  GUIUtil.toSmallIcon(FeatureInfoTool.ICON),
                                  FeatureInfoPlugIn.createEnableCheck(workbenchContext));
```

4.4 MAIN CLASSES

Class	Package
`addMainMenuItem, addLayerViewMenuItem`	`com.vividsolutions.jump.workbench.ui.plugin.FeatureInstaller`
`EnableCheck`	`com.vividsolutions.jump.workbench.plugin`

4.5 TO DO

- Shortcut keys on menu items: in the string passed to `FeatureInstaller`, prefix the letter with "&"


Note: This section is under construction

5. WORKBENCH DATA STRUCTURES

The main Workbench data structures are features, feature collections, layers, tasks and blackboards.

- A Feature is a representation of a 'geographic' object in the world, including its location, geometry, and other attributes (spatial and non-spatial). In the current Workbench model, each feature has one spatial attribute (its Geometry, imported from the JTS Java Topology Suite) and zero or more non-spatial attributes.
- A FeatureCollection, as the name suggests, is an object that represents a collection of Features. It supports special methods for querying the Features that lie within a given Envelope.
- A Layer is a FeatureCollection with additional stylistic information such as colour line-width, and so on. Layers are the basic objects that JUMP presents to the user for viewing or editing.
- Blackboards: an extremely useful concept described in The Pragmatic Programmer. A blackboard is just a String-to-Object map that anyone can use. We use Blackboards in many places: LayerManager, Layer, LayerViewPanel, Workbench, etc. If you search the code for Blackboard, you'll see some innovative uses. For example, the Options dialog has a single instance, and it's stored on the Workbench Blackboard.
- LayerViewPanel (the map on the right)
- LayerTreePanel (the tree on the left)
- Task, LayerManager: A LayerManager is a container for/registry of Layers. A Task is a thin wrapper around LayerManager that associates a name, project file, and Category.
- Geometry

5.1 MAIN CLASSES

Class	Package
Feature, FeatureCollection, FeatureSchema	com.vividsolutions.jump.feature
Layer, LayerManager	com.vividsolutions.jump.workbench.model
Blackboard	com.vividsolutions.jump.util

5.2 TO DO

§ Getting the "current" LayerManager. Non-TaskFrame windows with LayerManagers.
§ Firing the appropriate events; which events are automatically fired (special condition for feature-changed event). Don't need to call #fireAppearanceChanged when adding/deleting a feature from a Layer's FeatureCollection or modifying a geometry via an EditTransaction. But you do need to call it when modifying a Style, because Styles don't notify their Layer when they change
§ Why we have #wrapFeatureCollection rather than #setFeatureCollection
§ Checking that a layer is editable (AbstractPlugIn#isRollingBackEdits. CursorTool version?). Policy on using selected layers as input vs. using the editable layer. ref: getting selected layer.

- Always use `EditTransaction` to modify geometries, so that change events are fired and features are automatically unselected.
- When adding/removing features, prefer `#addAll/#removeAll` to #add/#remove so that fewer events are fired.
- The power of blackboards (e.g., setting a boolean, the animated clocks, communicating between plugins)
- "Task": think "project" (for now…future directions) (war journal)


War Journal: Describe Task here.


Note: This section is under construction

6. CursorTools

A `cursortool` is a button on the toolbar that sets the mode of mouse interaction (e.g., selection mode, or draw-polygon mode), like the buttons on the Drawing toolbar of Microsoft Word.

Subclassing `AbstractCursorTool` takes care of all the XOR logic involved in drawing. This class will also automatically generate label names from class name, similar to the way that Menu item names are automatically generated by `AbstractPlugIn`. See Section 3, Plugins.

- § Built-in cursor tools are loaded in `JUMPConfiguration`
- § There are several examples of cursor tools in the Workbench source code

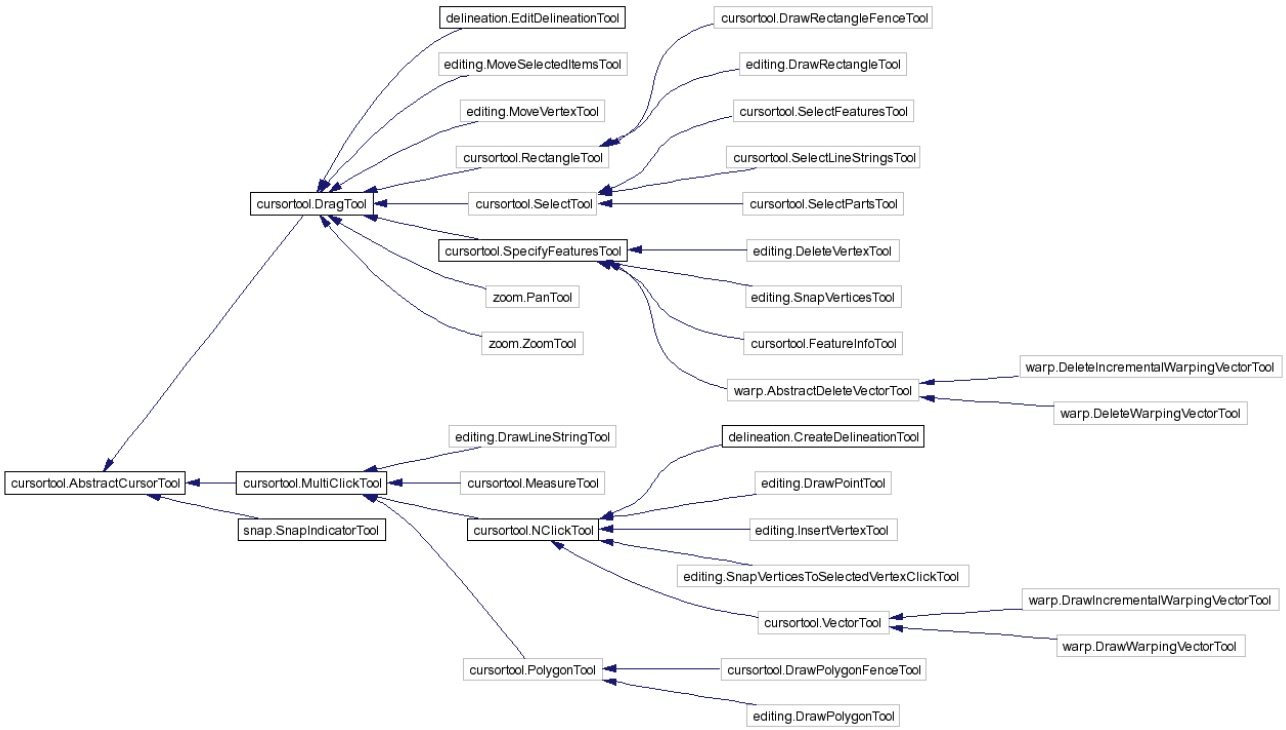
Every `AbstractCursorTool` supplies `#activate`, `#deactivate`, and `#gestureFinished` methods.

- `#activate` is called when the toolbar button is pressed.
- `#deactivate` is called when another `CursorTool`'s button is pressed.
- `#gestureFinished` is called when the `CursorTool`'s gesture is complete.

Compare this with how `PlugIn#initialize`, `PlugIn#execute` are called.

If your application requires more than one or two new CursorTools, it is recommended that you make these available to the User in a `ToolBox` (see Section 7, Toolboxes)

Cursor tools are another example of a commonly supplied resource that developers might supply as part of an Extension. For an example of how to package an install extension elements for a JUMP application, see Section 2.1 Example: Building a Simple JUMP Extension.



6.1 MAIN CLASSES

Class	Package
AbstractCursorTool, DragTool, MultiClickTool, NclickTool, AndCompositeTool, OrCompositeTool, DelegatingTool	com.vividsolutions.jump.workbench.ui.cursortool
WorkbenchToolBar	com.vividsolutions.jump.workbench.ui

6.2 TO DO

- Adding plugins and general buttons to WorkbenchToolBar
- Exception handling
- Undo
- Difference between cursor tools, plugin buttons, and general buttons
- doxygen class diagram
- re: UndoableCommand and EditTransaction in plugins section
- re: SelectionManager in plugins section
- re: Blackboard in Layers section
- Snapping
- Setting colour, line width, line style

 Note: This section is under construction

7. TOOLBOXES

A `toolbox` is a small modeless dialog that can contain cursor-tool buttons and custom GUI components. Toolboxes are always visible because they float above the Workbench, like the Picture toolbar in Microsoft Word.

If your application requires the addition of more than one or two new `cursorTools`, using a Toolbox is recommended: it avoids cluttering the main toolbar (which is already pretty packed!); and it allows the user to interact with data even while the dialog is up (modal dialogs can be annoying).

7.1 IMPLEMENTING A TOOLBOX

The following steps are required to set up a toolbox:

- Implement `initializeToolbar`.
- (Optionally) Add `CursorTools` to the toolbar.
- Add a panel to the toolbox.
- Write event handlers for the components in the panel.

7.2 MAIN CLASSES

Class	Package
`ToolboxDialog`, `ToolboxPlugIn`	`com.vividsolutions.jump.workbench.ui.toolbox`

7.3 TO DO

- Smart: if any cursor tools are pressed when the toolbox is closed, the toolbox will press the first cursor tool on the main toolbar before closing. Also, toolbox buttons are in same `ButtonGroup` as the buttons on the main toolbar.
- Can add cursor tools, plugins, and general buttons to toolbar just like with main toolbar. See Section 6, CursorTools.
- `ToolboxDialog`: advantage of modelessness. What `AbstractToolboxPlugIn` will do for you.
- Instantly make your toolbox context-sensitive (or rather taskframe-sensitive) by instantiating a ToolboxStateManager. You don't even have to store the ToolboxStateManager anywhere, just instantiate one, passing your toolbox into the constructor

 Note: This section is under construction

8. RENDERERS AND STYLES

A `renderer` is an object that draws on the Workbench using a `java.awt.Graphics`. A `style` is used by a `LayerRenderer` to draw a feature. Styles are a bit easier to write than full-blown renderers because you can focus on rendering one feature.

Most plugins don't need to write their own renderers or styles. They already have control over the colour, line width, etc. If they need to "decorate" features with arrowheads etc., there are a few existing Styles, for example, `ArrowLineStringEndPointStyle`.

The `BasicStyle` specifies style elements such as colours, line widths, etc. `ColorScheme` is used to select from a range of professionally designed colour schemes. You can pass one of the fill patterns from `FillPatternFactory` into `BasicStyle#setFillPattern`, or make your own. The factory contains all of the very cool textures from IBM[1]. For example, you can instantly give a fake mountain texture to your mountain features.

8.1 MAIN CLASSES

Class	Package
`LayerViewPanel, Viewport`	`com.vividsolutions.jump.workbench.ui`
`SimpleRenderer, FeatureCollectionRenderer, ImageCachingRenderer, RenderingManager`	`com.vividsolutions.jump.workbench.ui.renderer`
`Style, BasicStyle, VertexStyle, LabelStyle`	`com.vividsolutions.jump.workbench.ui.renderer.style`
`Java2XML`	`com.vividsolutions.jump.util.java2xml`

8.2 TO DO

- Create your own fill pattern – see `CustomFillPatternExamplePlugIn`
- Check the zoom level
- `Java2XML` to save the state of your renderers: need parameterless constructor; persisting collections; xml tags with no java mapping (to make project file more

[1] See http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/625

human-readable); interfaces and abstract classes still work with `Java2XML`; why I needed to make `LayerCategory#addJava2XMLLayerable`; why I needed to make `Java2XMLDataSource`

- Easy example of colour-theming


Note: This section is under construction

9. DATASOURCES

A `DataSource` is an object that mediates to move data between the Workbench and a file or other data source; it will replace the `Readers/Writers` found in earlier versions of JUMP.

Similarly, `DataSourceQueryChooser` replaces the now-deprecated `Driver`. These are the UI panels associated with a `DataSource`. Note: `Reader` and `Writer` are have not been deprecated in JUMP 1.1, because there is no plan to re-write the existing `Readers/Writers` in the near future.

Developers may need to supply custom `DataSources` in their JUMP Extensions in order to provide access to data for their applications. See Section 2.1, Example: Building a Simple JUMP Extension to learn how to include resources such as custom `DataSources` in your JUMP Extensions.

Tip: Use the `workbench properties` file during development so you don't have to keep generating a JAR file.

9.1 MAIN CLASSES

Class	Package
`DataSource`, `DataSourceQueryChooser`	`com.vividsolutions.jump.io.datasource`

9.2 TO DO

- currently file-based; future: web sources?


Note: This section is under construction

10. JAVA2XML

Java2XML is a utility class that lets you turn objects into XML documents and vice versa. It's dead simple to use, yet quite powerful — you can create complex-looking XML from it.

Java2XML has advantages over existing Java XML bindings: it is simpler than Castor (for example, there is no need for a schema or DTD); unlike Digester, it can both read and write; unlike Bewitched, it gives clear and helpful error messages when a problem occurs. Since there are very few classes (3), it's relatively easy to track down problems.

10.1 EXAMPLE: USING JAVA2XML

Say you have an object Person that you want to turn into XML:

```
public class Person {
   private int age = 50;
   public int getAge() { return age; }
   public void setAge(int i) { age = i; }

   private String name = "Charles";
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }

   private Collection nicknames = Arrays.asList(new String[]{"Charlie", "Chuck", "Chick"});
   public Collection getNicknames() { return Collections.unmodifiableCollection(nicknames);
}
   public void addNickname(String nickname) { nicknames.add(nickname); }
}

Person person = new Person();
```

If you try Java2XML, you'll get a very short XML document:

```
System.out.println(new Java2XML().write(person, "person"));

Output:
<person />
```

This is because you need to write a little mapping file that maps the Java fields to XML tags. Don't worry – it's easy! Just place the following file (Person.java2xml) right beside Person.class:

```
Person.java2xml:
<root>
<element xml-name="age" java-name="age" />
</root>
```

Now your XML document will contain a more complete representation:

```
System.out.println(new Java2XML().write(person, "person"));

Output:
<person>
  <age>50</age>
</person>
```

As you can see, the java2xml file maps an xml tag to a java field. All you have to do is specify the name of each (they can be different).

Why could we just specify "age" for the Java field rather than "getAge" and "setAge"? Java2XML does a case-insensitive search for methods like "setAge*" and "getAge*" (and "isAge*").

Want to add an attribute? It's the same thing again, but you use "attribute" instead of "element":

```
Person.java2xml:
<root>
   <attribute xml-name="name" java-name="name" />
   <element xml-name="age" java-name="age" />
</root>
```

```
System.out.println(new Java2XML().write(person, "person"));

Output:
<person name="Charles">
  <age>50</age>
</person>
```

If you nest the attribute inside the element, guess what the output looks like? "name" is nested inside "age"!

```
Person.java2xml:
<root>
   <element xml-name="age" java-name="age">
      <attribute xml-name="name" java-name="name" />
   </element>
</root>
```

```
System.out.println(new Java2XML().write(person, "person"));

Output:
<person>
  <age name="Charles">50</age>
</person>
```

What about collections? Look at "nickname" below:

```
Person.java2xml:
<root>
   <element xml-name="age" java-name="age">
      <attribute xml-name="name" java-name="name" />
   </element>
```

```
  <element xml-name="nickname" java-name="nickname" />
</root>
```

```
System.out.println(new Java2XML().write(person, "person"));

Output:
<person>
  <age name="Charles">50</age>
  <nickname>Charlie</nickname>
  <nickname>Chuck</nickname>
  <nickname>Chick</nickname>
</person>
```

Java2XML does a case-insensitive search for a getter like "`getNickname*`" and a setter like "`addNickname*`". Note that it's important to leave off the "s" in "nickname" — otherwise Java2XML will look for a setter like "`addNickname`**`s`**`*`" and of course there isn't one.

If you want to add some structure to your XML file, you can add tags that aren't mapped to anything. Look at "aliases" below and note that it isn't mapped to any java name:

```
Person.java2xml:
<root>
  <element xml-name="age" java-name="age">
    <attribute xml-name="name" java-name="name" />
  </element>
  <element xml-name="aliases">
    <element xml-name="nickname" java-name="nickname" />
  </element>
</root>
```

```
System.out.println(new Java2XML().write(person, "person"));

Output:
<person>
  <age name="Charles">50</age>
  <aliases>
    <nickname>Charlie</nickname>
    <nickname>Chuck</nickname>
    <nickname>Chick</nickname>
  </aliases>
</person>
```

Want to turn your XML back into a Java object? There's a class called `XML2Java` that will let you do just that. Easy!

Notes:

- The Java class that you want to turn into XML must have a constructor without any parameters. Otherwise how could `XML2Java` know what to use as parameters?
- You're not confined to storing simple types like Strings and ints in your XML files — you can also store other classes, but of course they too must have a little `.java2xml` file.

- Your getters and setters can even get/set interfaces and abstract classes! How does XML2Java know which class to instantiate then when you read it back in? Java2XML stores the name of the actual class in the XML file.
- Java2XML uses JDOM, which may not be appropriate for large XML files because the whole XML file gets loaded into memory

10.2 JAVA2XML FAQ

1. My IDE often deletes the directory containing my `.class` files – how can I prevent my *`.java2xml`* files from being lost?

   Do the same thing as you do with `.gifs` and `.jpegs`: Have your IDE copy your `.java2xml` files from your source directory to your classes directory every time you do a build. All major IDEs, including JBuilder and Eclipse, will let you set up which filename extensions are copied over.

2. What if one of the objects I'd like to turn into XML is from a third party, e.g., a Swing class like *`java.awt.Font`*? I wouldn't want to modify the Swing jar to add a *`.java2xml`* file beside *`Font.class`*.

   Add a `CustomConverter` for the `Font` class. It's almost as easy as writing a *`.java2xml`* file. Check out `Java2XML#addCustomConverter`.

10.3 MAIN CLASSES

Class	Package
Java2XML, XML2Java	com.vividsolutions.jump.java2XML

10.4 TO DO

- setters and getters work with Maps
- Java2XML can't turn interfaces and abstract classes into attributes. It is possible to turn them into elements though.
- reserved: "class", "null" attributes
- wart: one can't pass a custom class (e.g., HashMap) into Java2XML, but can pass in an Object containing a custom class.

 Note: This section is under construction

11. UNDO

11.1 TO DO

A plugin can indicate via `AbstractPlugIn#reportNothingToUndoYet` that it does not modify the system, or that it is undoable, but has not modified the system yet.

Note: This section is under construction

12. FEATURE TEXT WRITERS

The WKT, GML, and Coordinate List buttons on the left side of the Feature Info window are examples of feature text writers – they are different ways to display the contents of a Feature.

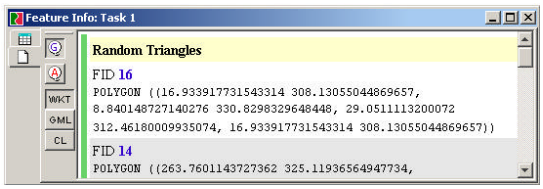


Figure 12-1 – The Feature Info window.

You can easily add custom feature text writers via a plug-in. Simply subclass AbstractFeatureTextWriter (and implement its one abstract method); then in your plug-in's #initialize method, say:

```
context.getWorkbenchContext().getFeatureTextWriterRegistry()
        .register(myFeatureTextWriter)
```

This code will add a button to the Feature Info window.

For code examples, see InstallStandardFeatureTextWritersPlugIn.java.

13. THE WIZARD FRAMEWORK

There isn't much on the Internet in the way of a free Wizard framework. JUMP's looks pretty good!

13.1 MAIN CLASSES

Class	Package
`WizardPanel, WizardDialog`	`com.vividsolutions.jump.workbench.ui.wizard`

13.2 TO DO

- Provide an example of how to use this.


Note: This section is under construction

14. HANDLING ERRORS, EXCEPTIONS AND WARNINGS

Many applications will throw up error dialogs to show minor warnings or information, interrupting the user's flow of work. With the JUMP framework, you can show warnings in the status bar. The warning is yellow and flashes for a second, to get the user's attention, but doesn't interrupt the user's train of thought because there is no OK button to push.

```
WorkbenchFrame.warnUser("The selection is invalid.");
```

Error Dialog Of course, a full-blown error dialog is available if you really need to grab the user's attention. In fact, unhandled exceptions automatically percolate up to the error dialog. The error dialog has a Show/Hide Details button that reveals the Java exception stack trace, which is useful for debugging. (To do: build a framework for reporting errors to the user in a friendly manner. For example, exception must have a friendly message, as well as cause, and recommended actions).

14.1 MAIN CLASSES

Class	Package
WorkBenchFrame, LayerViewPanelContext, ErrorHandler, ErrorDialog	com.vividsolutions.jump.workbench.ui

14.2 TO DO

- Provide an example of how to use this.


Note: This section is under construction