Developing VStar Plug-ins

Introduction

Some things clearly belong to the basic set of features needed by a program such as VStar such as the ability to:

- Plot light curves and phase plots
- View data in tabular format
- Load observation data from a variety of sources
- Save and print
- Filter/search
- Perform period analysis

Other features are less obvious. Even some of the items on the list above are not clear-cut. Which period analysis algorithms should be included? What format should files be saved in? What filter expressions should be permitted? It is not possible to anticipate every use to which VStar may be put.

To address this, VStar has a plug-in capability that allows anyone with a moderate knowledge of Java (or other programming languages) to add new functionality to VStar without having to modify its source code, or to have commit rights to the SourceForge source code repository, or to be able to rebuild and redeploy VStar.

In addition, a plug-in may be a good way to prototype an idea or algorithm that may ultimately be included in the VStar source tree. In this way, the VStar user community can more easily engage in the tool's future development, and the tool's flexibility is enhanced.

The plug-in developer is not constrained to use only Java. Any language that can generate Java Virtual Machine class files may be used to develop plug-ins, e.g.

- Scala
- Ruby (JRuby)
- Python (Jython)

- JavaScript (Rhino)
- Groovy
- Lisp (e.g. Clojure)

Plug-in Types

There are currently four types of VStar plug-ins:

1. Observation tool
2. General tool
3. Custom filter
4. Period analysis

All plug-ins must take the form of a jar (Java archive) file. Once this jar file is copied to a particular directory, the plug-in will appear as a menu item next time VStar is started.

An *observation tool plug-in* will appear in the Tool menu and is able to perform arbitrary processing on the currently loaded dataset.

A *general tool plug-in* is intended to be used for general utilities.

A *custom filter plug-in* filters data in the plots and tables as per the View -> Filter… menu item, permitting an arbitrarily complex filter expression. Custom filters will appear in the View -> Custom Filter sub-menu.

A *period analysis plug-in* will appear in the Analysis -> Period Search sub-menu and applies an algorithm such as a Fourier Transform to a particular series (band) of the loaded dataset. By default, the data that is passed to the plug-in is the series that is used as the basis for the light curve mean plot. The results may then be plotted and/or stored in a table. Support is provided for creating a dialog for displaying the results of the period analysis and for selecting a period value with which to create a new phase plot.

Application Programming Interface

VStar's application programming interface (API) for plug-in development is captured as Javadoc.

The file vstar.jar contains VStar's functionality is used for VStar plug-in development. The latest version may be obtained from here:

http://vstar.svn.sourceforge.net/viewvc/vstar/trunk/dist/vstar.jar?view=log

The latest VStar documentation (Javadoc) zip file can be downloaded from here:

http://vstar.svn.sourceforge.net/viewvc/vstar/trunk/doc/vstar_docs.zip?view=log

For a particular stable release of vstar.jar and documentation, the tags directory can be consulted:

http://vstar.svn.sourceforge.net/viewvc/vstar/tags/

or the file area:

https://sourceforge.net/projects/vstar/files/

Insofar as creating plug-ins is concerned, the key classes and interfaces are:

1. `org.aavso.tools.vstar.plugin.PluginBase`
2. `org.aavso.tools.vstar.plugin.ObservationToolPluginBase`
3. `org.aavso.tools.vstar.plugin.GeneralToolPluginBase`
4. `org.aavso.tools.vstar.plugin.CustomFilterPluginBase`
5. `org.aavso.tools.vstar.plugin.period.PeriodAnalysisPluginBase`
6. `org.aavso.tools.vstar.plugin.period.PeriodAnalysisDialogBase`
7. `org.aavso.tools.vstar.plugin.period.PeriodAnalysisComponentFactory`
8. `org.aavso.tools.vstar.data.ValidObservation`

The `org.aavso.tools.vstar.plugin.period.impl` package contains an example period analysis plug-in implementation: `DcDftPeriodAnalysisPlugin` for the Date Compensated Discrete Fourier Transform algorithm. This class is presented internally to VStar as just another plug-in.

Given that plug-in references vstar.jar, a plug-in can make use of other VStar classes (e.g. `org.aavso.tools.vstar.ui.dialog.MessageBox`).

Creating an Observation Tool plug-in

Suppose you wanted to create a simple Tool plug-in to count the number of observations in the loaded dataset. While not very useful in itself, working through a simple example like this will get you started towards creating more complex, useful plug-ins. Here are the steps to follow:

- Ensure you have installed the Java Development Kit, version 1.6 or higher.
- Download the version of vstar.jar associated with the release of VStar you wish to develop plug-ins for.
- Create a file with your favourite editor or Integrated Development Environment (e.g. Eclipse) with the following content:

```
package my.vstar.plugin;

import java.util.List;

import org.aavso.tools.vstar.data.ValidObservation;
import org.aavso.tools.vstar.plugin.ObservationToolPluginBase;
import org.aavso.tools.vstar.ui.dialog.MessageBox;
```

```java
/**
 * This is a simple observation counting VStar tool plug-in.
 */
public class ObservationCounter extends ObservationToolPluginBase {

   @Override
   public void invoke(List<ValidObservation> obs) {
          int count = obs.size();
          MessageBox.showMessageDialog("Observation Count", String.format(
                 "There are %d observations in the loaded dataset.", count));
   }

   @Override
   public String getDescription() {
          return "Observation counting tool plug-in";
   }

   @Override
   public String getDisplayName() {
          return "Observation Counter";
   }
}
```

Some key things to note are:

- The file must be called `ObservationCounter.java` and located in the directory `my/vstar/plugin` relative to the current directory. Let's call the latter `src`.
- After saving the file, and assuming you are taking a bare-bones approach to editing and compiling Java code , from a DOS prompt or *nix shell, with `src` as your current directory, compile the source code, including the vstar.jar file (specifying the full path to vstar.jar) in the classpath, like this:

```
javac -cp vstar.jar my/vstar/plugin/ObservationCounter.java
```

- Next, create a jar file with the same name as the fully qualified `ToolPluginBase` subclass. Note that this naming scheme is not optional. In this case, we would have:

```
jar -cf my.vstar.plugin.ObservationCounter.jar
        my/vstar/plugin/ObservationCounter.class
```

- Next, move the jar file to the `vstar_plugins` directory. Plug-in jars must be placed into the vstar_plug-ins directory in your home directory (for Mac or *nix that will be $HOME or ~, "C:\Documents and Settings*user*" under Windows).

- Finally, start VStar. A menu item called `Observation Counter` should appear in the Tool menu. Load a dataset from a file or the database and select that menu item. A simple dialog box should appear telling you how many observations are loaded.

Creating a General Tool plug-in

This section shows an example of a General Tool plug-in that converts a Julian Day to a calendar date. The source file needs to be `JDToDateTool.java` and must be located in the directory `my/vstar/plugin` corresponding to the package name `my.vstar.plugin` declared in the code.

```
package my.vstar.plugin;

import javax.swing.JOptionPane;

import org.aavso.tools.vstar.plugin.GeneralToolPluginBase;
import org.aavso.tools.vstar.ui.dialog.MessageBox;
import org.aavso.tools.vstar.util.date.AbstractDateUtil;

/**
 * A general tool plug-in that converts a JD to a calendar date.
 */
public class JDToDateTool extends GeneralToolPluginBase {

	@Override
	public void invoke() {
		String str = JOptionPane.showInputDialog("Enter JD");
		if (str != null && str.trim().length() != 0) {
			try {
				double jd = Double.parseDouble(str);

				String cal =
					AbstractDateUtil.getInstance().jdToCalendar(jd);

				MessageBox.showMessageDialog("JD to Calendar Date",
					String.format("%1.4f => %s", jd, cal));

			} catch (NumberFormatException e) {
				MessageBox.showErrorDialog("JD Error", String.format(
						"'%s' is not a number.", str));
			} catch (IllegalArgumentException e) {
				MessageBox.showErrorDialog("JD Error", String.format(
						"'%s' is an invalid JD: %s", str,
						e.getMessage()));
			}
		}
	}

	@Override
	public String getDescription() {
		return "Convert JD to Calendar Date";
	}

	@Override
	public String getDisplayName() {
		return "JD to Calendar Date...";
	}
}
```

Creating a Custom Filter plug-in

This is an example Custom Filter plug-in that demonstrates that any condition can be used to determine whether or not an observation should be included in the filtered subset, including some relationship between observations.

In this example, an observation is included only if the previous observation has a JD whose integer portion is divisible by 9 and the band of the previous and current observations is Johnson V or Johnson B.

This condition is of dubious value, but it serves to demonstrate that the condition can be arbitrarily complex.

Try this filter on the last 2 years of epsilon Aurigae observations from the AAVSO International database (for example).

```
package my.vstar.plugin;

import java.util.List;

import org.aavso.tools.vstar.data.SeriesType;
import org.aavso.tools.vstar.data.ValidObservation;
import org.aavso.tools.vstar.plugin.CustomFilterPluginBase;

public class CustomFilterTest extends CustomFilterPluginBase {

    @Override
    protected void filter(List<ValidObservation> obs) {
        ValidObservation last = null;

          for (ValidObservation curr : obs) {
               if (last != null) {
                   int lastIntegerJD = (int) last.getJD();
                       SeriesType lastBand = last.getBand();
                       SeriesType currBand = curr.getBand();

                       if (lastIntegerJD % 9 == 0 &&
                           (currBand == SeriesType.Johnson_V ||
                            currBand == SeriesType.Johnson_B) &&
                           (lastBand == SeriesType.Johnson_V ||
                            lastBand == SeriesType.Johnson_B)) {

                              addToSubset(curr);
                       }

                       last = curr;
               }
        }

        @Override
        public String getDescription() {
               return "Includes an observation if a condition holds " +
                       "for the current and previous observation.";
        }

        @Override
        public String getDisplayName() {
               return "V or B plus JD Divisibility Checker";
        }
}
```

Notice that the method `addToSubset()` invoked above exists on the custom filter plug-in base class. See the Javadoc for more details regarding custom filter plug-ins.

Creating a Period Analysis plug-in

Using the same approach as for the Tool plug-in example, create a file called `PeriodAnalysisPluginTest.java` under `src/my/vstar/plugin`:

```
package my.vstar.plugin;

import java.awt.BorderLayout;
import java.awt.Component;
import java.util.List;

import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;

import org.aavso.tools.vstar.data.SeriesType;
import org.aavso.tools.vstar.data.ValidObservation;
import org.aavso.tools.vstar.plugin.period.PeriodAnalysisComponentFactory;
import org.aavso.tools.vstar.plugin.period.PeriodAnalysisDialogBase;
import org.aavso.tools.vstar.plugin.period.PeriodAnalysisPluginBase;
import org.aavso.tools.vstar.ui.NamedComponent;
import org.aavso.tools.vstar.ui.mediator.Mediator;
import org.aavso.tools.vstar.ui.mediator.message.MeanSourceSeriesChangeMessage;
import org.aavso.tools.vstar.ui.mediator.message.NewStarMessage;
import org.aavso.tools.vstar.ui.mediator.message.PeriodAnalysisSelectionMessage;
import org.aavso.tools.vstar.util.notification.Listener;
import org.aavso.tools.vstar.util.period.PeriodAnalysisCoordinateType;

/**
 * VStar period analysis plug-in test.
 *
 * This plug-in generates random periods and shows them on a line plot, in a
 * table, with the selected period displayed in a label component. A new phase
 * plot can be generated with that period.
 */
public class PeriodAnalysisPluginTest1 extends PeriodAnalysisPluginBase {

	final private int N = 100;

	private double period;

	private double[] domain = new double[N];
	private double[] range = new double[N];

	protected final static String NAME = "Period Analysis Plug-in Test";

	@Override
	public void executeAlgorithm(List<ValidObservation> obs) {
		// Create a set of random values to be plotted. A real plug-in would
		// instead apply some algorithm to the observations to create data
		// arrays (e.g. a pair of domain and range arrays).
		for (int i = 0; i < N; i++) {
			domain[i] = i;
			range[i] = Math.random();
		}
	}

	@Override
	public String getDescription() {
		return "Period Analysis Plug-in Test: generates random periods.";
	}

	@Override
	public JDialog getDialog(SeriesType seriesType) {
```

```
        return new PeriodAnalysisDialog();
}

@Override
public String getDisplayName() {
        return NAME;
}

@Override
protected void meanSourceSeriesChangeAction(
                MeanSourceSeriesChangeMessage msg) {
        // Nothing to do
}

@Override
protected void newStarAction(NewStarMessage msg) {
        // Nothing to do
}

class PeriodAnalysisDialog extends PeriodAnalysisDialogBase {
        PeriodAnalysisDialog() {
                super(NAME);
                prepareDialog();
                this.setNewPhasePlotButtonState(false);
        }

        @Override
        protected Component createContent() {
                // Random plot.
                Component plot = PeriodAnalysisComponentFactory.createLinePlot(
                                "Random Periods", "", domain, range,
                                PeriodAnalysisCoordinateType.FREQUENCY,
                                PeriodAnalysisCoordinateType.POWER);

                // Data table.
                PeriodAnalysisCoordinateType[] columns = {
                                PeriodAnalysisCoordinateType.FREQUENCY,
                                PeriodAnalysisCoordinateType.AMPLITUDE };

                double[][] dataArrays = { domain, range };

                Component table =
                                PeriodAnalysisComponentFactory.createDataTable(
                                                columns, dataArrays);

                // Random period label component.
                JPanel randomPeriod = new RandomPeriodComponent(this);

                // Return tabbed pane of plot and period display component.
                return PeriodAnalysisComponentFactory.createTabs(
                                new NamedComponent("Plot", plot),
                                new NamedComponent("Data", table),
                                new NamedComponent("Random Period",
                                randomPeriod));
        }

        // Send a period change message when the new-phase-plot button is
        // clicked.
        @Override
        protected void newPhasePlotButtonAction() {
                sendPeriodChangeMessage(period);
        }
}
```

```
        /**
         * This class simply shows the currently selected (from plot or table) and
         * updates the period member to be used when the new-phase-plot button is
         * clicked. It's not really necessary, just shows a custom GUI component.
         */
        class RandomPeriodComponent extends JPanel implements
                        Listener<PeriodAnalysisSelectionMessage> {

                private JLabel label;
                private PeriodAnalysisDialog dialog;

                public RandomPeriodComponent(PeriodAnalysisDialog dialog) {
                        super();
                        this.dialog = dialog;
                        label = new JLabel("Period: None selected"
                                        + String.format("%1.4f", period));
                        this.add(label, BorderLayout.CENTER);

                        Mediator.getInstance().getPeriodAnalysisSelectionNotifier()
                                        .addListener(this);
                }

                // Period analysis selection update handler methods.
                public void update(PeriodAnalysisSelectionMessage msg) {
                        if (msg.getSource() != this) {
                                try {
                                        period = range[msg.getItem()];

                                        label.setText("Period: " +
                                                        String.format("%1.4f", period));

                                        dialog.setNewPhasePlotButtonState(true);
                                } catch (ArrayIndexOutOfBoundsException e) {
                                }
                        }
                }

                public boolean canBeRemoved() {
                        return false;
                }
        }
}
```

You should consult the VStar Javadoc for more about the methods overridden or invoked in this example.

To see a more useful period analysis plug-in, see the class `DcDftPeriodAnalysisPlugin` mentioned before.

Closing Remarks

Note that if a plug-in class has a constructor (which neither of the examples given do), it must be public and parameter-less.

A non-trivial plug-in may be dependent upon other jar files not known to VStar. These jars must be placed into the `vstar_plugin_libs` directory which has the same root as the `vstar_plugins` directory.

Currently, plug-ins must strictly be licensed under VStar's license, AGPL 3.0, because they will make use of some of VStar's classes. For this reason,

consideration will be given to placing VStar under a dual license such that use of vstar.jar for plug-in or similar development is subject to the Lesser GPL (LGPL), permitting linking of code against vstar.jar without that code itself becoming subject to AGPL.